

# SOFTWARE DEVELOPMENT

VCE UNITS 3 & 4



 Nelson

Selina Dennis  
Gary Bass  
Therese Keane



2E

# Learning that puts you at the centre

...and on the learning path to success.



Find everything you need to access your Nelson MindTap course at [cengage.com.au/nelsonmindtap](https://cengage.com.au/nelsonmindtap)



## **COPYRIGHT NOTICE**

Copyright in this work is owned by Cengage Learning Australia (“the work”). A condition of purchase of this electronic version of the work is that you agree to respect the copyright in the work, abide by the Copyright Act 1968 and specifically agree not to transfer, sell, assign, misuse, copy or transmit an electronic or other version of the work to any third party.

**Please note:** This product is accompanied by a licence (single user, network or adoption) governing the terms and conditions of its use.

This is a legal agreement between the you, (the “Customer”) and Cengage Learning Australia Pty Limited (ABN 14 058 280 149) (the “Licensor”) which provides the terms and conditions of this non-exclusive licence and the limited warranty for the Product. Use of the Product indicates an acknowledgement that the Customer has read and agreed to be bound by the terms and conditions of this Agreement. If you do not agree to these terms and conditions, return the Product to the place of purchase within 15 days of the date of purchase (with proof of purchase) for a full refund

#### **1. Licence Grant**

You do not receive title to the Product. Copyright in the Product (which includes all images, photographs, video, animations, audio, music and text incorporated in the Product, including all of the accompanying printed material) is owned by the Licensor and/or its suppliers and is protected by Australian copyright laws. The Licensor grants you a non-exclusive licence to use the Product subject to the restrictions and terms set out in this Agreement.

#### **2. A Licence allows you to:**

Use the Product on your computer. The Customer represents that they shall in no way place the Product in the public domain or in any way compromise our copyright in the Material. You agree to take reasonable steps to protect our copyright.

#### **3. You may not:**

Alter, modify, translate, reverse engineer, decompile, or adapt the software or create derivative works based on the Product. Make further copies by any means technological, electronic, digital whatsoever without the written permission of the Licensor. Rent or transfer all or any part of your rights under this Agreement. Remove or alter any copyright or other proprietary notice or label attached to the software.

#### **4. Termination**

Any failure to comply with the terms and conditions of this agreement will result in the automatic termination of this licence. Upon termination of this licence for any reason, the Customer must destroy or return to the Licensor all copies of the software and accompanying documentation.

#### **5. Warranties**

To the extent permitted by law, the Licensor’s liability for any breach of the warranty or any term implied by law into this licence is limited to the lowest cost of replacing the goods, acquiring equivalent goods or having the goods repaired.

# SOFTWARE DEVELOPMENT

VCE UNITS 3 & 4



Selina Dennis  
Gary Bass  
Therese Keane

2E

Software Development VCE Units 3&4

2nd Edition

Selina Dennis

Gary Bass

Therese Keane

ISBN 9780170483964

Product manager: Melissa O'Shea

Content developer: Morgan Begg

Content manager: Alice Kane/Katie McCappin

Associate Content project manager: Ananya Sarkar

Text designer: Leigh Ashforth, Watershed Art & Design

Cover designer: Cengage Creative Studio

Editor: Stephen Roche

Proofreader: Peter Cruttenden

Permissions/Photo researcher: Lumina Datamatics

Cover: stock.adobe.com/RDVector

Typeset by Straive

Any URLs contained in this publication were checked for currency during the production process. Note, however, that the publisher cannot vouch for the ongoing currency of URLs.

Extracts from the VCE Applied Computing Study Design (2025–2028) are reproduced by permission, © VCAA. VCE® is a registered trademark of the VCAA. The VCAA does not endorse or make any warranties regarding this study resource. Current VCE Study Designs, past VCE exams and related content can be accessed directly at [www.vcaa.vic.edu.au](http://www.vcaa.vic.edu.au)

© 2025 Cengage Learning Australia Pty Limited

#### Copyright Notice

This Work is copyright. No part of this Work may be reproduced, stored in a retrieval system, or transmitted in any form or by any means without prior written permission of the Publisher. Except as permitted under the *Copyright Act 1968*, for example any fair dealing for the purposes of private study, research, criticism or review, subject to certain limitations. These limitations include: Restricting the copying to a maximum of one chapter or 10% of this book, whichever is greater; providing an appropriate notice and warning with the copies of the Work disseminated; taking all reasonable steps to limit access to these copies to people authorised to receive these copies; ensuring you hold the appropriate Licences issued by the Copyright Agency Limited ("CAL"), supply a remuneration notice to CAL and pay any required fees. For details of CAL licences and remuneration notices please contact CAL at Level 11, 66 Goulburn Street, Sydney NSW 2000, Tel: (02) 9394 7600, Fax: (02) 9394 7601  
Email: [info@copyright.com.au](mailto:info@copyright.com.au)  
Website: [www.copyright.com.au](http://www.copyright.com.au)

For product information and technology assistance,  
in Australia call **1300 790 853**;  
in New Zealand call **0800 449 725**

For permission to use material from this text or product, please email [aust.permissions@cengage.com](mailto:aust.permissions@cengage.com)

#### National Library of Australia Cataloguing-in-Publication Data

A catalogue record for this book is available from the National Library of Australia

#### Cengage Learning Australia

Level 7, 80 Dorcas Street  
South Melbourne, Victoria Australia 3205

For learning solutions, visit [cengage.com.au](http://cengage.com.au)

Printed in China by 1010 Printing International Limited.  
1 2 3 4 5 6 7 27 26 25 24 23



# Contents

|                             |      |
|-----------------------------|------|
| Preface                     | v    |
| About the authors           | vi   |
| How to use this book        | vii  |
| Outcomes                    | x    |
| Problem-solving methodology | xiii |

## Unit 3

|   |           |  |            |
|---|-----------|--|------------|
| <b>Introduction</b>   | 1         | <b>Chapter 3 Software analysis</b>   | <b>86</b>  |
| <b>Chapter 1 Introduction to programming</b>                    | <b>2</b>  | Why develop software?  | 87         |
| Data types  | 3         | Documenting problems   | 88         |
| Data structures   | 5         | Writing a design brief   | 89         |
| Solution requirements   | 8         | Project management   | 91         |
| Constraints   | 8         | Collecting data  | 100        |
| Scope   | 9         | Functional and non-functional requirements                                       | 102        |
| Representing designs  | 10        | Software requirements specifications   | 107        |
| Files   | 13        | Analytical tools depicting existing processes and systems                        | 108        |
| Internal documentation  | 16        | Next steps   | 125        |
| Naming conventions  | 18        | <b>Chapter 4 Software development: Software design</b>                           | <b>135</b> |
| <b>Chapter 2 Development and features of a computer program</b> | <b>26</b> | Continuing Unit 3, Outcome 2   | 136        |
| Programming languages   | 27        | Software solution specifications   | 136        |
| Features of a programming language                              | 31        | Managing files   | 136        |
| Object-oriented programming                                     | 44        | Generating design ideas  | 141        |
| Graphical user interfaces                                       | 50        | Evaluating design ideas  | 151        |
| Algorithms for sorting  | 51        | Evaluating the efficiency and effectiveness of solutions                         | 156        |
| Algorithms for searching  | 57        | Methods of expressing software designs   | 158        |
| Efficiency of algorithms  | 60        | Characteristics of user experience (UX)  | 164        |
| Validation techniques   | 61        | Authentication and data protection   | 166        |
| Checking that modules function correctly                        | 62        | Design principles influencing appearance and functionality                       | 171        |
| Preparing for Unit 3, Outcome 1                                 | 85        | Legal requirements relating to the ownership and privacy of data and information | 175        |
|   |           | Preparing for Unit 3, Outcome 2  | 187        |
|   |           | Unit 3 exam practice   | 189        |

# Unit 4

|  |            |  |            |
|--|------------|--|------------|
| <b>Introduction</b>  | <b>193</b> | <b>Chapter 6 Cyber security risks</b>                            | <b>239</b> |
| <b>Chapter 5 Software development and project evaluation</b>               | <b>194</b> | Security controls  | 240        |
| Efficient and effective solutions  | 195        | Vulnerabilities and risks  | 245        |
| Data types, structures and sources   | 197        | Software security  | 261        |
| Features of a programming language   | 198        | Strategies for minimising potential risks                        | 262        |
| Approaches to software development   | 206        | Identifying software and data vulnerabilities                    | 265        |
| Software testing   | 207        | Threat modelling   | 266        |
| Debugging and alpha testing  | 208        | Next steps   | 273        |
| Beta testing   | 214        | <b>Chapter 7 Software security</b>                               | <b>280</b> |
| Evaluating solutions   | 216        | Why develop software?  | 281        |
| Evaluation strategies  | 216        | Minimising risk  | 282        |
| Applying evaluation criteria   | 222        | Key legislation for storage and disposal of data and information | 288        |
| Recording the progress of projects   | 223        | Ethical issues   | 296        |
| Evaluating the efficiency and effectiveness of solutions and project plans | 227        | Mitigating threats, vulnerabilities and risks                    | 301        |
| Next steps   | 231        | Improving the security of software development practices         | 308        |
| Preparing for Unit 4, Outcome 1  | 237        | Evaluating the security of software development practices        | 313        |
|  |            | Preparing for Unit 4, Outcome 2                                  | 322        |
|  |            | Unit 4 exam practice   | 323        |
|  |            | <b>Index</b>   | <b>327</b> |

# Preface

This second edition of *Software Development VCE Units 3 & 4* is written to meet the requirements of the VCAA VCE Applied Computing Study Design that took effect from 2025.

We believe that teachers and students require a text that focuses on the **areas of study** specified in the **study design**, and that presents information in a sequence that allows simple transition from theory into practical assessment tasks. We have therefore written this textbook so that a class can begin at Chapter 1 and work their way systematically through to the end. Students will encounter material relating to the **key knowledge** dot points for each **outcome** before they reach the special section that describes the outcome. The study design outlines key skills that indicate how the knowledge can be applied to produce a solution to a client need or opportunity. These outcome preparation sections occur regularly throughout the textbook, and flag an appropriate point in the student's development for each outcome to be completed. The authors have covered all key knowledge dot points for the outcomes from the Software Development VCE Units 3 & 4 course.

Our approach has been to focus on the key knowledge required for each school-assessed outcome, and to ensure that students are well prepared for these; however, there is considerable duplication in the study design relating to the knowledge required for many of the outcomes. We have found that, with an outcomes approach, we are sometimes covering material several times. For example, solution requirements, constraints and scope are referenced in two different outcomes. In these cases, we have tried to provide general coverage in the first instance, and specifically apply the concept to a situation relevant to the related outcome on subsequent encounters.

We have incorporated a margin column in the text that provides additional information and reinforcement of key concepts. The margin column also includes activities related to the topics covered in the text, and a consideration of issues relevant to the use of information systems.

Outcome features are included at several points in the book, indicating the nature of the tasks that students are to undertake in the completion of the school-assessed outcome. The steps required to complete the outcome are listed, together with advice and suggestions for approaching the task. The output and support material needed for submission are described. Sample tasks and further advice relating to the outcomes are available on Nelson MindTap.

The chapters are organised to present the optimum amount of information in the most effective manner. The text is presented in concise, clearly identified sections to guide students through the text. Each chapter is organised into the sections described on pages vii–viii.

# About the authors

**Selina Dennis** is a Software Development and English Language teacher for the Department of Education and has been heavily involved in Computing study designs. Selina has a Bachelor of Arts and Science in Computer Science and Linguistics from the University of Melbourne, and has a particular interest in Computational Linguistics. She spent several years in California in the computing industry as an engineering manager and director of engineering before entering teaching.

**Gary Bass** has taught VCE Applied Computing at Year 11 and Year 12 in an online course environment at Virtual School Victoria. Previously he has taught VCE Physics, as well as developed and delivered middle-school ICT courses. Gary has presented at DLTV DigiCON and the annual IT teachers' conference on many topics, including Pop-up Makerspace; Big Data requires huge analysis – data visualisation; AR + VR = Mixed reality; and Marshall McLuhan – Medium is the message.

# How to use this book

## KEY KNOWLEDGE

The key knowledge from the VCAA Software Development VCE Units 3 & 4 Study that you will cover in each chapter is listed on the first page of each chapter. The list includes key knowledge specified in the outcome related to the chapter.

## FOR THE STUDENT

The first page of each chapter includes an overview of the chapter's contents so that you are aware of the material you will encounter.

## CHAPTERS

The major learning material that you will encounter in the chapter is presented as text, photographs and illustrations. The text describes in detail the theory associated with the stated outcomes of the Software Development VCE Units 3 & 4 Study in easy-to-understand language. The photographs show hardware, software and other objects that have been described in the text. Illustrations are used to demonstrate concepts that are more easily explained in this manner.

Throughout the chapter, glossary terms are highlighted in bold, blue text, and you can find their definitions at the end of the chapter, in **Essential terms**.

The **School-Assessed Task Tracker** at the bottom of every odd-numbered page provides you with a visual reminder to help you track your progress in the school-assessed task so that you can complete all required stages on time.

## MARGIN COLUMN

The margin column contains further explanations that support the main text, weblink icons, additional material outside the study and cross-references to material covered elsewhere in the textbook. Issues relevant to Software Development that you can discuss with your classmates are also included in the form of 'Think about Software Development' boxes.

## CHAPTER SUMMARY

The chapter summary at the end of each chapter is divided into two main parts to help you review each chapter.

**Essential terms** are the glossary terms that have been highlighted throughout the chapter.

**Important facts** are a list of summaries, ideas, processes and statements relevant to the chapter, in the order in which they occur in the chapter.

### THINK ABOUT SOFTWARE DEVELOPMENT 3.1

Project management tools are useful to find the perfect number of people needed on a task so it is finished as quickly as possible without anyone being idle. Using software, develop a Gantt chart to plan the baking of a cake. Assume you can use as many cooks as you want.

## TEST YOUR KNOWLEDGE

These are short-answer questions that are provided to help you when reviewing the chapter material. The questions are grouped and identified with a section of the text to allow your teacher to direct appropriate questions based on material covered in class. Teachers will be able to access answers to these questions on **Nelson MindTap**.

## APPLY YOUR KNOWLEDGE

Each chapter concludes with a set of questions requiring you to demonstrate that you can apply the theory from the chapter to more complex questions. The style of questions reflects what you can expect in the end-of-year examination. Teachers will be able to access suggested responses to these application questions on **Nelson MindTap**.

## PREPARING FOR THE OUTCOMES

This section appears at points in the course where it is appropriate for you to complete an outcome task. The information provided describes what you need to do in the outcome, the suggested steps to be followed in the completion of the task and the material that needs to be submitted for assessment.

## EXAM-STYLE QUESTIONS

Each unit concludes with a set of exam-style questions requiring you to demonstrate that you can apply the theory from the chapter to more complex questions. These questions include multiple-choice questions as well as short-answer questions based around small case studies. Teachers will be able to access answers to these questions on **Nelson MindTap**.

## NELSON MINDTAP

This product includes **Nelson MindTap**, an online learning environment that puts you at the centre. Nelson Software Development Nelson MindTap gives you access to an eText with additional resources. Nelson MindTap products are not for individual use; they are for adoption, class set or book hire. Nelson MindTap is teacher-led courseware and requires an active course for access. This is a student activation code that can only be used to access a Nelson MindTap course as part of a school's adoption. Please contact your learning consultant to discuss purchasing options.



### An online learning space that provides students with tailored learning experiences.

- Access tools and content that make learning simpler yet smarter.
- **Flexible formats:** choose how you navigate using either the online eText, or offline PDFs.
- Margin links in the student book signpost multimedia student resources found on Nelson MindTap.



### For students:

- **Access** weblinks associated with the content to build upon your understanding of the concepts through real-world applications.
- **Analyse** additional material such as spreadsheets, charts and infographics to enhance your learning experience.
- **Navigate** your own path, accessing the content and support as you need it.
- **Download** individual chapters as PDFs for easy offline access to the material.

### For teachers:

- **Tailor** content to different learning needs – assign directly to the student, or the whole class.
- **Monitor** progress using assessment tools like Gradebook and Reports.

- **Assign** assessment and revision tasks with downloadable end-of-chapter tests and practice exams.
- Get expert and detailed guidance on **preparing SACs** with sample tasks, marking advice and suggested solutions.
- Use **teaching plans** to support easy program and assessment planning.
- Access expert **answers and solutions** to test your knowledge, apply your knowledge, exam-style questions and end-of-chapter tests.
- Help build your students' exam readiness with **Cognero Assess** – a test bank containing hundreds of questions and answers to create, assign or export formative and summative tests.

*Complimentary access to these resources is only available to teachers who use this book as part of a class set, book hire or booklist. Contact your Cengage Education Consultant for information about access and conditions*

### Security & privacy:

Nelson MindTap joined the Safer Technologies 4 Schools (ST4S) Product Badge Program in 2024. The annual ST4S assessment is part of our commitment to supporting the online security and safety of students and schools. Learn more at [st4s.edu.au](http://st4s.edu.au)

# Outcomes

| OUTCOME   | KEY KNOWLEDGE  | LOCATION                |
|---|--|-------------------------|
| <b>Unit 3<br/>Area of Study 1<br/>Outcome 1</b> | On completion of this unit the student should be able to interpret teacher-provided solution requirements and designs and use appropriate features of an object-oriented programming language to develop working software modules.   | Chapter 1 and Chapter 2 |
| KK3.1.1   | <ul style="list-style-type: none"> <li>emerging trends in programming using artificial intelligence</li> </ul>   | pp. 27–30               |
| KK3.1.2   | <ul style="list-style-type: none"> <li>characteristics of functional and non-functional requirements, constraints and scope</li> </ul>   | pp. 8–9                 |
| KK3.1.3   | <ul style="list-style-type: none"> <li>design tools for representing modules</li> </ul>  | pp. 10–12               |
| KK3.1.4   | <ul style="list-style-type: none"> <li>characteristics of data types</li> </ul>  | pp. 3–5                 |
| KK3.1.5   | <ul style="list-style-type: none"> <li>characteristics of data structures</li> </ul>   | pp. 5–7                 |
| KK3.1.6   | <ul style="list-style-type: none"> <li>characteristics of data sources (plain text (TXT), delimited (CSV) and XML files)</li> </ul>  | pp. 13–16               |
| KK3.1.7   | <ul style="list-style-type: none"> <li>principles of OOP</li> </ul>  | pp. 44–9                |
| KK3.1.8   | <ul style="list-style-type: none"> <li>features of a programming language</li> </ul>   | pp. 31–44; 50           |
| KK3.1.9   | <ul style="list-style-type: none"> <li>purposes and features of naming conventions for solution elements (variables, interface controls, code structures)</li> </ul>   | pp. 18–19               |
| KK3.1.10  | <ul style="list-style-type: none"> <li>validation techniques for data</li> </ul>   | pp. 61–2                |
| KK3.1.11  | <ul style="list-style-type: none"> <li>purposes of internal documentation</li> </ul>   | pp. 16–18               |
| KK3.1.12  | <ul style="list-style-type: none"> <li>algorithms for sorting and searching</li> </ul>   | pp. 51–61               |
| KK3.1.13  | <ul style="list-style-type: none"> <li>types of errors</li> </ul>  | pp. 61–4                |
| KK3.1.14  | <ul style="list-style-type: none"> <li>debugging and testing techniques for checking modules function correctly</li> </ul>   | pp. 65–6                |
| KK3.1.15  | <ul style="list-style-type: none"> <li>test cases comparing expected and actual output in testing tables</li> </ul>  | pp. 66–72               |
| <b>Unit 3<br/>Area of Study 2<br/>Outcome 2</b> | On completion of this unit the student should be able to document a problem, need or opportunity, formulate a project plan, document an analysis, and generate design ideas and a preferred design for creating a software solution. | Chapter 3 and Chapter 4 |
| KK3.2.1   | <ul style="list-style-type: none"> <li>reasons why individuals and organisations undertake software development projects</li> </ul>  | p. 87                   |
| KK3.2.2   | <ul style="list-style-type: none"> <li>features of a brief that documents a problem, need or opportunity</li> </ul>  | pp. 89–90               |
| KK3.2.3   | <ul style="list-style-type: none"> <li>features of project management using Gantt charts</li> </ul>  | pp. 91–9                |
| KK3.2.4   | <ul style="list-style-type: none"> <li>methods for collecting data to determine needs and requirements</li> </ul>  | pp. 100–2               |
| KK3.2.5   | <ul style="list-style-type: none"> <li>characteristics of functional and non-functional requirements</li> </ul>  | pp. 102–4               |
| KK3.2.6   | <ul style="list-style-type: none"> <li>constraints that influence solution development</li> </ul>  | pp. 105–7               |
| KK3.2.7   | <ul style="list-style-type: none"> <li>characteristics of solution scope</li> </ul>  | p. 105                  |
| KK3.2.8   | <ul style="list-style-type: none"> <li>analytical tools for depicting the relationships between users, data and systems, including context diagrams, data flow diagrams and use case diagrams</li> </ul>                             | pp. 108–25              |
| KK3.2.9   | <ul style="list-style-type: none"> <li>purpose and features of software requirements specifications</li> </ul>   | pp. 107–8               |

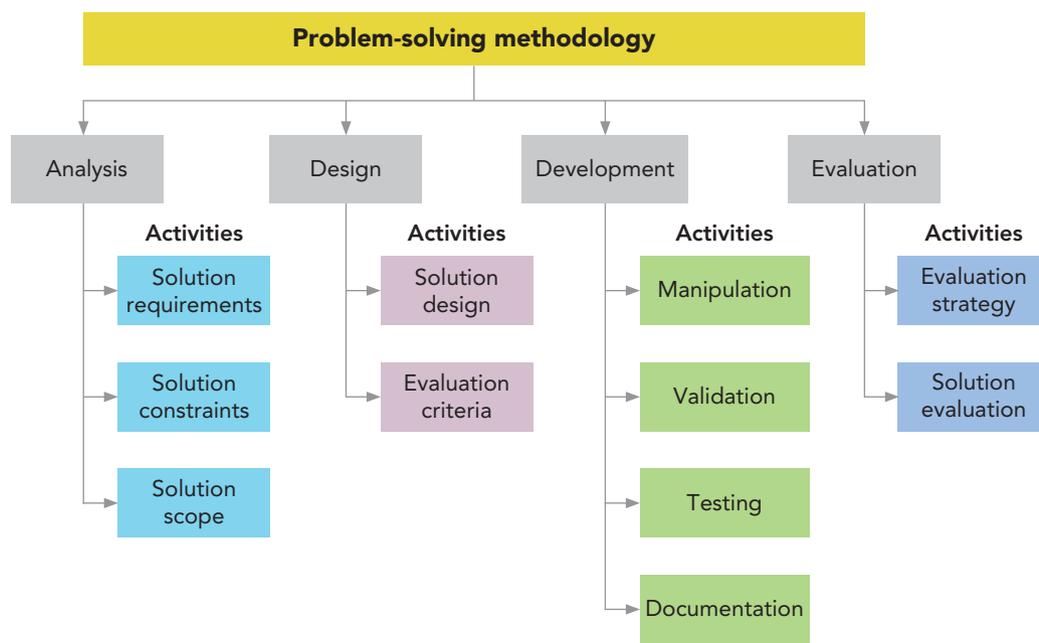


| OUTCOME   | KEY KNOWLEDGE  | LOCATION                |
|---|--|-------------------------|
| KK3.2.10  | <ul style="list-style-type: none"> <li>key legal requirements relating to the intellectual property and ownership and privacy of data used</li> </ul>  | pp. 175–9               |
| KK3.2.11  | <ul style="list-style-type: none"> <li>file management techniques</li> </ul>   | pp. 136–41              |
| KK3.2.12  | <ul style="list-style-type: none"> <li>ideation techniques and tools for generating design ideas</li> </ul>  | pp. 141–51              |
| KK3.2.13  | <ul style="list-style-type: none"> <li>criteria for evaluating design ideas and the efficiency and effectiveness of solutions</li> </ul>   | pp. 151–7               |
| KK3.2.14  | <ul style="list-style-type: none"> <li>design tools for generating solution designs from design ideas</li> </ul>   | pp. 158–64              |
| KK3.2.15  | <ul style="list-style-type: none"> <li>characteristics of user experience (UX) and how these affect software design</li> </ul>   | pp. 164–70              |
| KK3.2.16  | <ul style="list-style-type: none"> <li>design principles that influence the appearance and functionality of the user interface/s of the software solution</li> </ul>   | pp. 171–4               |
| <b>Unit 4<br/>Area of Study 1<br/>Outcome 1</b> | On completion of this unit the student should be able to develop and evaluate a software solution that meets requirements and assess the effectiveness of the project plan.  | Chapter 5               |
| KK4.1.1   | <ul style="list-style-type: none"> <li>characteristics of efficient and effective solutions</li> </ul>   | pp. 195–7               |
| KK4.1.2   | <ul style="list-style-type: none"> <li>characteristics of data types, data structures and data sources for input, storage and output</li> </ul>  | pp. 197–8               |
| KK4.1.3   | <ul style="list-style-type: none"> <li>features of a programming language</li> </ul>   | pp. 198–206             |
| KK4.1.4   | <ul style="list-style-type: none"> <li>established and innovative approaches to software development</li> </ul>  | pp. 206–7               |
| KK4.1.5   | <ul style="list-style-type: none"> <li>validation techniques</li> </ul>  | p. 208                  |
| KK4.1.6   | <ul style="list-style-type: none"> <li>debugging and alpha testing techniques for checking that solutions meet requirements and function correctly</li> </ul>  | pp. 208–14              |
| KK4.1.7   | <ul style="list-style-type: none"> <li>strategies for conducting beta testing</li> </ul>   | pp. 214–16              |
| KK4.1.8   | <ul style="list-style-type: none"> <li>features of evaluation strategies</li> </ul>  | pp. 216–22              |
| KK4.1.9   | <ul style="list-style-type: none"> <li>techniques for applying evaluation criteria</li> </ul>  | p. 222                  |
| KK4.1.10  | <ul style="list-style-type: none"> <li>factors that influence the effectiveness of project plans</li> </ul>  | pp. 229–30              |
| KK4.1.11  | <ul style="list-style-type: none"> <li>techniques for recording the progress of projects</li> </ul>  | pp. 223–6               |
| KK4.1.12  | <ul style="list-style-type: none"> <li>techniques for assessing the effectiveness of a project plan</li> </ul>   | pp. 230–1               |
| <b>Unit 4<br/>Area of Study 2<br/>Outcome 2</b> | On completion of this unit the student should be able to respond to a teacher-provided case study to analyse an organisation’s software development practices, identify and evaluate current security controls and threats to software development practices, and make recommendations to improve practices. | Chapter 6 and Chapter 7 |
| KK4.2.1   | <ul style="list-style-type: none"> <li>goals and objectives of medium and large organisations</li> </ul>   | p. 281                  |
| KK4.2.2   | <ul style="list-style-type: none"> <li>advantages and disadvantages of developing software in-house or externally</li> </ul>   | p. 285                  |
| KK4.2.3   | <ul style="list-style-type: none"> <li>types of vulnerabilities and risks within insecure development environments</li> </ul>  | pp. 245–60;<br>268–73   |
| KK4.2.4   | <ul style="list-style-type: none"> <li>security controls used to protect software development practices and data stored within applications</li> </ul>   | pp. 240–5               |
| KK4.2.5   | <ul style="list-style-type: none"> <li>threat modelling principles</li> </ul>  | pp. 266–7               |



# Problem-solving methodology

When an information problem exists, a structured problem-solving methodology is followed to ensure that the most appropriate solution is found and implemented. For the purpose of this course, the problem-solving methodology has four key stages: analysis, design, development and evaluation. Each of these stages can be further broken down into a common set of activities. Each unit may require you to examine a different set of problem-solving stages. It is critical for you to understand the problem-solving methodology because it underpins the entire VCE Applied Computing course.



Reproduced from the VCE Applied Computing Study Design (2025–2028) © VCAA; used with permission.

## Analyse the problem

The purpose of analysis is to establish the root cause of the problem, the specific information needs of the organisation involved, limitations on the problem and exactly what a possible solution would be expected to do (the scope). The three key activities are:

- 1 identifying solution requirements – attributes and functionality that the solution needs to include, information it must produce and data needed to produce this information
- 2 establishing solution constraints – the limitations on solution development that need to be considered. Constraints are classified as economic, technical, social, legal and related to usability.
- 3 defining the scope of the solution – what the solution will and will not be able to do.

**FIGURE 1** The four stages of the problem-solving methodology and their key activities

## Design the solution

During the design stage, two to three potential design ideas based on both appearance and function are planned and the most appropriate of these is chosen. Criteria are also created to select the most appropriate ideas and to evaluate the solution's success once it has been implemented. The two key design activities are:

- 1 creating the solution design – it must clearly show a developer what the solution should look like, the specific data required and how its data elements should be structured, validated and manipulated. Tools typically used to represent data elements could include data dictionaries, input–process–output (IPO) charts, object descriptions, flowcharts and pseudocode. The following tools are also used to show the relationship between various components of the solution: annotated diagrams, mock-ups, storyboards, site maps, queries, context diagrams, data flow diagrams and use case diagrams. Furthermore, the appearance of the solution, including elements like a user interface, graphic representations or data visualisations, needs to be planned so that overall layout, fonts and their colours, for example, can be represented. Layout diagrams, annotated diagrams or mock-ups usually fulfil this requirement. A combination of tools from each of these categories will be selected to represent the overall solution design. Regardless of the visual or functional aspects of a solution design, at this stage a design for the tests to ultimately ensure the solution is functioning correctly must also be created.
- 2 specifying evaluation criteria – during the evaluation stage, the solution is assessed to establish how well it has met its intended objectives. The criteria for evaluation must be created during the design stage so that all personnel involved in the task are aware of the level of performance that ultimately will determine the success or otherwise of the solution. The criteria are based on the solution requirements identified in the analysis stage and are measured in terms of efficiency and effectiveness.

## Develop the solution

The solution is created by the developers during this stage from the designs supplied to them. The 'coding' takes place, but also checking of input data (validation), testing that the solution works and the creation of user documentation. The four activities involved with development are:

- 1 manipulating or coding the solution – the designs are used to build the electronic solution. The coding will occur here and internal documentation will be included where necessary.
- 2 checking the accuracy of input data by way of validation – manual and electronic methods are used; for example, proofreading is a manual validation technique. Electronic validation involves using the solution itself to ensure that data is reasonable by checking for existence, data type and that it fits within the required range. Electronic validation, along with any other formulas, always needs to be tested to ensure that it works properly.

- 3 ensuring that a solution works through testing – each formula and function, not to mention validation and even the layout of elements on the screen, need to be tested. Standard testing procedures involve stating what tests will be conducted, identifying test data, stating the expected result, running the tests, stating the actual result and correcting any errors.
- 4 documentation to support the development of the solution, including internal documentation. Internal documentation supports the functioning, maintaining and upgrading of the solution.

## Evaluate the solution

Some time after a solution has been in use by the end user or client, it needs to be assessed or evaluated to ensure that it has been successful and does actually meet the user's requirements. The two activities involved in evaluating a solution are:

- 1 working out an evaluation strategy – creating a timeline for when various elements of the evaluation will occur and how and what data will be collected (because it must relate to the criteria created in the design stage).
- 2 evaluating the solution – providing feedback to the user about how well the solution meets their requirements, needs or opportunities in terms of efficiency and effectiveness. This is based on the findings of the data gathered at the beginning of the evaluation stage when compared with the evaluation criteria created during the design stage.



# Unit 3

## INTRODUCTION

In this study, software will be developed by applying the problem-solving methodology through the stages of analysis, design, development and evaluation.

In Unit 3 of Software Development, you will develop working software modules using a programming language (Unit 3, Outcome 1). You will then identify a suitable client, analyse a need or opportunity for that client, select an appropriate development model, prepare a project plan, develop a software requirements specification (SRS) and design a software solution. You will use all the stages of the problem-solving methodology (PSM) to prepare the project plan. This will complete the first half of the School-assessed Task (SAT) (Unit 3, Outcome 2). The second half of the SAT will be completed in Unit 4 (Unit 4, Outcome 1).

### Area of Study 1 – Software development: programming

**OUTCOME 1** In this outcome, you will respond to teacher-provided solution requirements and designs to develop four working software modules of increasing complexity. You will use an object-oriented programming language to apply the problem-solving activities of manipulation (coding), validation, testing and providing internal documentation in the development stage.

### Area of Study 2 – Software development: analysis and design

**OUTCOME 2** In this outcome, you will identify a software problem, need or opportunity within your community. You will use the problem-solving methodology stages of analysis and design to complete part 1 of the School-assessed Task (SAT). Part 2 will be completed in Unit 4 (Unit 4, Outcome 1). You will use all the stages of the problem-solving methodology to prepare a project plan, using student-determined and teacher-provided milestones.

A range of methods will be used to gather data for analysis and you will use tools and techniques to describe relationships between users, data and systems. A full solution description is proposed using a software requirements specification (SRS), which entails documented details of requirements, constraints and scope of the solution. A design folio of possible design ideas is generated, with a preferred design selected after the application of student-developed evaluation criteria. This is then fully described using a range of design tools that address the appearance, structure and functionality of the proposed solution.

Note: The same evaluation criteria will be used in Unit 4, Area of Study 1 to evaluate the efficiency and effectiveness of the created solution.

Victorian Curriculum and Assessment Authority, Victorian Certificate of Education  
Applied Computing Study Design 2025–2028, VCAA, 2024, pp. 53–8

# Introduction to programming

## KEY KNOWLEDGE

After completing this chapter, you will be able to demonstrate knowledge of:

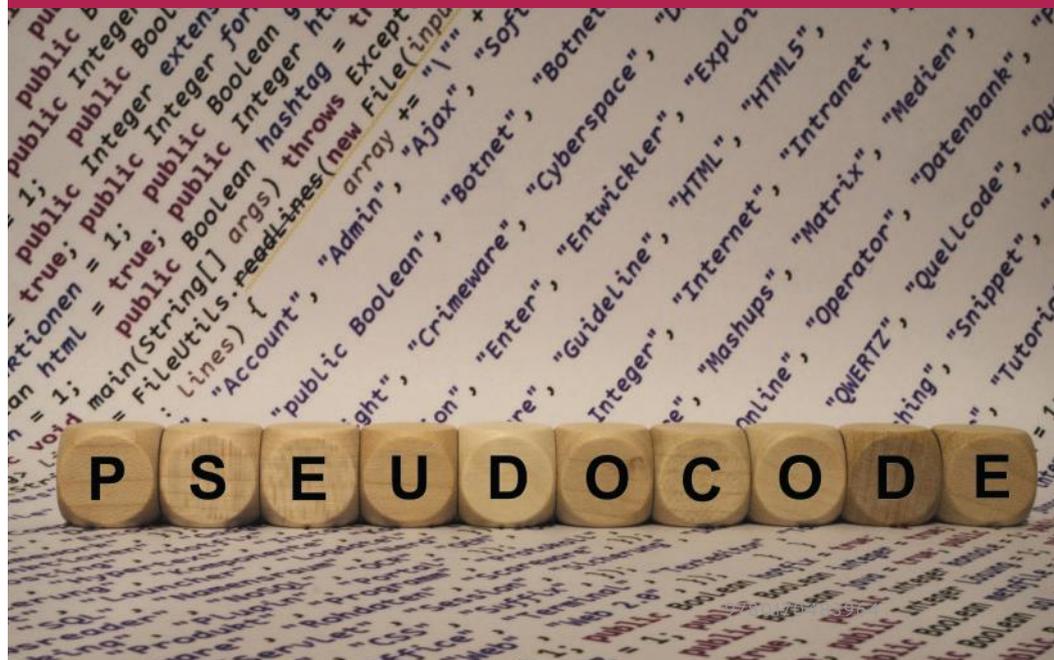
- KK3.1.2: characteristics of functional and non-functional requirements, constraints and scope
- KK3.1.3: design tools for representing modules, including data dictionaries, mock-ups, object descriptions, input-process-output (IPO) charts and pseudocode
- KK3.1.4: characteristics of data types, including text, numeric and Boolean
- KK3.1.5: types of data structures, including one- and two-dimensional arrays and records (varying data types, field index)
- KK3.1.6: characteristics of data sources, including plain text (TXT), delimited (CSV) and XML files
- KK3.1.9: purposes and features of naming conventions for solution elements
- KK3.1.11: purposes of internal documentation.

Victorian Curriculum and Assessment Authority, Victorian Certificate of Education Applied Computing Study Design 2025–2028, VCAA, 2024, pp. 54–5

## FOR THE STUDENT

While programming languages differ, the fundamental components and logic needed to write using a programming language are the same. Data types and structures are consistent across many different programming languages, as are the conventions for naming solution elements, internal documentation and the formatting and structure of different file types. Similarly, methods of representing designs are also independent of any programming language; in fact, this is their strength.

The focus of this chapter is the key elements of programming that are platform and language independent, as well as the methods used to create design briefs and represent designs. This chapter forms the basis of the background needed to prepare you for Unit 3, Outcome 1 as well as Unit 3, Outcome 2.



# Data types

In programming, a **data type** is a method of classifying a **variable** to determine the data that variable can contain, as well as how the variable can be manipulated – that is, what it can do, and what can be done to it. While programming languages vary widely from each other, data types do not; they are consistent across all programming languages. When programming, it can be important to choose an appropriate data type when creating a variable. It is also important to select the most efficient data type. For example, it is not efficient to select a numeric data type that supports decimal places when creating a variable if that variable will only ever contain whole numbers. Similarly, storing a number as a string is not as efficient as storing it as a numeric data type, even if it is possible to convert strings to numbers.

## Numeric

The **numeric** data type consists of whole numbers, referred to as **integers**; decimal numbers, referred to as **floating points**; and date/time, which are stored in integer form. Integers can be referred to as *unsigned*, which means they can only store positive whole numbers, or *signed*, which means they can store both positive and negative whole numbers.

All numeric data types can have mathematical operations performed on them. The fundamental operations shown in Table 1.1 are the most common.

**TABLE 1.1** Fundamental data type operations

|                                   |        |
|-----------------------------------|--------|
| Addition                          | +      |
| Subtraction                       | –      |
| Multiplication                    | *      |
| Division                          | /      |
| Whole number division (quotient)  | //     |
| Remainder after division (modulo) | %      |
| Powers                            | **     |
| Assign values                     | = or ← |

When more than one operation appears within a line of code, the order of operations follows the same rules as BODMAS in mathematics: brackets, orders, division, multiplication, addition and subtraction. If two operators have the same precedence, they are evaluated from left to right.

Numeric data can also undergo comparisons, with the comparisons shown in Table 1.2 being the most common.

**TABLE 1.2** Data type comparison operations

|                                       |          |
|---------------------------------------|----------|
| Less than                             | <        |
| Less than or equal to (or $\leq$ )    | <=       |
| Greater than                          | >        |
| Greater than or equal to (or $\geq$ ) | >=       |
| Equal to                              | == or =  |
| Not equal to                          | != or <> |

Some languages, such as Perl, Ruby and Swift, are dynamically typed, which means that type checking for variables is completed when the program is run rather than in the code itself. Programmers do not need to set variable data types in dynamically typed languages.

SCHOOL-ASSESSED TASK TRACKER

Design brief
  Project plan
  Analysis
  Folio of alternative design ideas
  Development
  Evaluation and assessment
  Final submission

The power of 31 for signed integers represents 32 bits, minus one bit that is needed to determine if the signed integer is positive or negative. The 1 that is subtracted from the total is due to computer systems counting from 0, rather than 1.

### THINK ABOUT SOFTWARE DEVELOPMENT

1.1

How big is  $2^{64}$ ? How big would the maximum number of integers be if we began using 128-bit computer systems?

### THINK ABOUT SOFTWARE DEVELOPMENT

1.2

Research how other operating systems store date/time data types. What problems might arise if these differences in tracking dates and times are not accounted for in programming?

The maximum and minimum values for floating point numbers are not as easy to determine as integers. When single-precision floating point numbers are stored, one bit is used for the sign (positive or negative), eight bits for the exponent and 23 bits to store the significant digits of the floating point number (i.e. the fraction). For interest, the formulas that calculate the maximum and minimum for 32-bit systems and 64-bit systems can be found via the weblinks below.



Weblinks  
32-bit systems  
64-bit systems

## Integer

Integers are commonly represented internally in a computer system as a group of binary digits, called *bits*. A bit is the smallest unit of data in a computer, and has a single binary value, 0 or 1. Bits are stored in multiples of eight, referred to as bytes; therefore, there are eight bits to a byte.

The maximum and minimum values of an integer depend on the computer architecture used to run the program and whether the integer is signed or unsigned.

In **32-bit computer systems**, integers that are signed have a minimum value of  $-2^{31}$  and a maximum value of  $2^{31} - 1$ ; from  $-2\,147\,483\,647$  to  $2\,147\,483\,647$ . Unsigned integers have a minimum of 0 and a maximum of  $2^{32} - 1$  ( $4\,294\,967\,295$ ).

In **64-bit computer systems**, integers that are signed have a minimum value of  $-2^{63}$  and a maximum value of  $2^{63} - 1$ . Unsigned integers have a minimum of 0 and a maximum of  $2^{64} - 1$ .

It is important to know the computer architecture on which a program will run before designing and developing a software solution. Going beyond the maximum and minimum values of integers can result in an **integer overflow**, which may result in a program crashing, or producing inconsistent or invalid output. Integer overflows compromise a software solution's reliability and security.

Dates and times are stored as positive integers within computer systems. The most common way to store date/time in integer form is the Unix timestamp. This is the number of seconds that have elapsed since 1 January 1970, at 00:00:00 UTC, not counting leap seconds. The Unix timestamp is widely used in Unix-like operating systems and file formats.

## Floating point

Floating point numbers are the computer representation of real numbers – that is, numbers that allow for decimal places.

Floating points consist of two main parts:

- a *significand*, which contains the digits of the number that is represented. These can be either positive or negative.
- an *exponent*, which helps determine where the decimal point is placed within the significand.

Two basic formats of a floating point number in computer systems are single precision and double precision. Single precision is used in 32-bit systems and double precision in 64-bit systems.

## Text and character

The **character** data type is a symbol that has meaning. It can consist of any single meaningful unit, such as a letter, a number, a punctuation mark, a symbol or even a space. For example, the word 'example' has seven characters.

What is determined as 'meaningful' relies on something referred to as **character encoding**. Character encoding is a way that a computer program can translate binary data into meaningful characters. There are many character-encoding schemes that handle different character sets. For example, ASCII is a character-encoding scheme that represents English characters, punctuation and numbers. UTF-8 encoding is a character-encoding scheme that can represent characters from other languages, such as Japanese Kanji and Korean Hanja, as well as symbols such as those representing the euro (€) and yen (¥).

A set or sequence of characters is referred to as text, or strings. For example, the text 'I like pie' consists of eight characters that are letters, and two characters that are spaces. Text is often implemented in programming languages as an array of characters.

Both text and characters can undergo the same comparisons as numeric data types, as listed in Table 1.2. Depending on the programming language used, they can also undergo some of the fundamental data type operations listed in Table 1.1, such as addition, multiplication and assignment. An example of text addition can be seen in Figure 1.1.

```
INPUT firstName
PRINT "Hello " + firstName + "!"
```

**FIGURE 1.1** Pseudocode example of text addition

## Boolean

**Boolean** data types have only two possible values: 0 and 1. In a programming language these are often referenced with the words ‘False’ or ‘True’, respectively. This data type is named after George Boole, a 19th-century mathematician who was the first to define an algebraic system of logic. Boolean data types are very useful for systems that require decisions to be made or conditions to be met.

Much like integers, Boolean values can be treated mathematically, allowing for comparison operators such as those listed in Table 1.2. This allows for Boolean logic operations to occur in any programming language.

Boolean values can also be used with logical operators *and*, *or* and *not* in statements where a condition must be met. For example, if a program is required to turn on a light in a room if it is dark and the light is not already on, it could use two Boolean values to test for this condition (Figure 1.2).

```
IF isDark = True AND lightOn = False THEN
    turnOnLight ()
ENDIF
```

**FIGURE 1.2** Pseudocode example of Boolean test conditions

## Data structures

A **data structure** is a method of organising data to allow particular operations to be performed on them efficiently; in this way, data structures are more complex than data types. The types of data structures used in Software Development are: one-dimensional arrays, two-dimensional arrays and records.

### Array

An **array** is a data structure that contains groupings of data. These elements must be of the same data type, such as character, numeric or Boolean. Arrays can also store groupings of other data structures, such as fields, records or even other arrays. Arrays are very useful in programming, as they allow for related sets of data to be organised and ordered efficiently.

For example, a teacher might collect data related to the height of each of their students and store this data as a one-dimensional array of floating point numbers. This is a much more efficient method of storing data than creating separate variables to store each student’s height, as it allows for faster sorting and searching.

### 1.3 THINK ABOUT SOFTWARE DEVELOPMENT

You may come across the term ‘null terminated string’ when looking at programming language reference manuals and computer science texts. What is a null terminated string?

Boolean values take up only a small amount of space in memory, so it is tempting to use them to store any type of data that seems to only have two values. However, it is important to consider future expansions to programs before making a decision that will limit a data type. For example, many old systems that had gender stored as a Boolean value are now being rewritten to change gender to a character or string data type.

#### SCHOOL-ASSESSED TASK TRACKER

Design brief  Project plan  Analysis  Folio of alternative design ideas  Development  Evaluation and assessment  Final submission

The contents of an array are referenced using an index value – often an integer starting at 0. The way that an array is stored means that each element has a set position within it. This allows for quick access to a particular element of the array, without necessarily needing to check every element.

Arrays can be considered to have **dimensions**, with the most basic array type being one-dimensional. A one-dimensional array is a linear data structure where elements are accessed by a single index. It represents a list of elements that are all of the same data type. Each element in the array can be accessed using an integer index.

One-dimensional arrays can typically use the operations shown in Table 1.3.

**TABLE 1.3** One-dimensional array data structure operations

|                  |                       |
|------------------|-----------------------|
| Add or append    | +                     |
| Remove or delete | -                     |
| Lookup           | arrayName[indexValue] |

```
arrayHeights ← [ 1.23, 1.35, 1.21, 1.61 ]
firstStudent ← arrayHeights[0]
secondStudent ← arrayHeights[1]
fourthStudent ← arrayHeights[3]
```

**FIGURE 1.3**  
Pseudocode example of a one-dimensional array

Consider the array in Figure 1.3. In this example, `firstStudent` would contain the floating point number 1.23, `secondStudent` would contain 1.35 and `fourthStudent` would contain 1.61.

Arrays are well suited to organising large numbers of elements, such as when you want to add 100 numbers together. Without an array, you would input one data value at a time, assign a value to each variable, then add them together and output the result (see Figure 1.4). An alternative approach uses an array `Number[n]` where `n` can be from 0 to 99 for a total of 100 elements (see Figure 1.5).

```
BEGIN
  Total ← 0
  Input number1
  Input number2
  Input number3
  .
  .
  Input number99
  Input number100
  Total ← number1 + number2 + number3 + ... + number99 + number100
  DISPLAY Total
END
```

**FIGURE 1.4**  
Pseudocode for a simple addition of 100 elements

```
BEGIN
  Total ← 0
  FOR Loop ← 0 to 99
    Input Number (Loop)
    Total ← Total + Number (Loop)
  NEXT Loop
  DISPLAY Total
END
```

**FIGURE 1.5**  
Pseudocode for adding an array with 100 elements

Compare the two pseudocode versions. The non-array version (Figure 1.4) would require 100 lines of code to take the input values. When using an array (Figure 1.5), just six lines of code are needed.

If the task were extended to 500 numbers, the array would still require just six lines of code. The FOR Loop would change from 0 to 99 to become 0 to 499.

In pseudocode (covered later in this chapter), arrays are sometimes indexed starting at 1, but this should always be made clear in the pseudocode comments.

A **two-dimensional array** is a data structure where elements are stored in a table-like format, consisting of rows and columns. It can be thought of as an array of arrays. While a one-dimensional array is akin to a single row or a single column of data, a two-dimensional array represents a grid of data. This requires accessing elements using two indices rather than one.

Two-dimensional arrays can typically use the operations shown in Table 1.4.

**TABLE 1.4** Two-dimensional array data structure operations

|                  |                                  |
|------------------|----------------------------------|
| Add or append    | +, addRow(), addColumn()         |
| Remove or delete | -, removeRow(), removeColumn()   |
| Lookup           | arrayName[rowIndex][columnIndex] |

```
allSeats ← [ [ True, True, True, True ],
             [ True, True, False, False ],
             [ False, True, True, True ] ]
PRINT allSeats[2][0]
```

Consider the pseudocode in Figure 1.6. This example represents a seating chart, where each seat is either available (True) or unavailable (False). The two-dimensional array contains inner lists that represent each row of seats, and the Boolean value in each of the positions in that inner list represents the availability of that seat in that row. The final line of the pseudocode prints the availability of the first seat in the third row – which is not available.

## Record and field

A **record** is a basic data structure for collections of related elements. These elements may or may not be of the same data type. Most frequently, records are used in database systems, but they are also commonly used in programming languages, often referred to as structs.

A record consists of a number of **fields** that are typically fixed – that is, the fields do not tend to change once the record is defined and used. Each field has a name and each has its own data type.

For example, a customer record may contain fields such as `firstName`, `lastName` and `dateOfBirth`.

Records are most useful when a collection of variables are related to each other. This provides a logical method of ordering data within a program so that data can be accessed quickly.

Object-oriented programming languages do not have records; however, a record is essentially an object that has no object-oriented features, containing only collections of fields and values. Records and fields can also exist in some types of structured plain text files.

As records contain programmer-defined fields, there are no set operations that can be listed for the record data structure. Rather, there are common operations that can be performed on the record and the fields within it, such as assignment and comparison, as well as adding or removing fields.

In general, an array of size  $n$  will have elements from index 0 to  $n - 1$ .

**FIGURE 1.6** Pseudocode example of a two-dimensional array

## Solution requirements

**Solution requirements** are what the client needs from the solution – the features that they want. Solution requirements in a design brief are often worded descriptions rather than technical descriptions.

### Functional requirements

**Functional requirements** relate to what the solution should do. Typically, functional requirements will describe a behaviour or function of the software. This includes determining the input required, the desired output, data manipulation and data validation. These are further explained in Chapter 3, page 102.

### Non-functional requirements

**Non-functional requirements** describe the quality attributes of the solution. This requires consideration of usability, reliability, portability, robustness and maintainability. These are further explained in Chapter 3, page 103.

## Constraints

Solution **constraints** are factors that may limit or restrict solution requirements. Typically, constraints involve economic, technical, social, legal and usability factors. Constraints are briefly outlined below but also described in further detail in Chapter 3.

### Economic

Economic constraints include time and cost.

The deadline by which the user or client needs to have the solution operational will define the time available to design and develop the solution. The more time available, the more time there is to complete an in-depth analysis and detailed designs, and to develop advanced features of the solution. The shorter the time frame, the faster each stage in the problem-solving methodology needs to be completed.

Meanwhile, the funds available to complete the project may affect the hardware and software (digital systems) available for use, the number and range of staff who are available to work on the solution and even the data used as input, if the required data sets need to be purchased.

Both a lack of time and a lack of money may result in a re-evaluation of the user's requirements, or a re-evaluation of how those requirements can be achieved.

### Technical

**Technical constraints** are constraints related to the hardware and software available for the project. Availability and compatibility of hardware and software, capacity of storage and equipment, processing speed and security are all examples of technical constraints.

For example, developers need to keep in mind that smartphone users may not always have access to a high-speed network connection, so they need to ensure that any animated data visualisation solution does not require a large amount of bandwidth to download and view.

### Social and legal

**Non-technical constraints** relate to areas other than hardware and software. The user's level of expertise (social) is an example of a non-technical constraint. For example, if a solution is being developed for users with little digital systems expertise, this may restrict

#### THINK ABOUT SOFTWARE DEVELOPMENT

1.4

List three other technical constraints developers of smartphone apps need to consider when developing a product.

the inclusion of requirements that would involve complex manoeuvres. Creating a solution for a child audience may also restrict the method used to input data into the solution.

Legal requirements are another type of non-technical constraint. Privacy laws may restrict features linked to displaying personal data in the solution, or to collecting data from the device of someone using your solution. Intellectual property laws and ownership of data may also restrict features of a software solution, adding constraints to potential solutions.

## Scope

The **scope** outlines the boundaries or parameters of the solution, so that all stakeholders are aware of exactly what the solution will contain. These boundaries or parameters can relate to the entirety of the planned software as a whole, or the contents of a new version of the software. The scope of the solution consists of two elements: what the solution will do and what the solution will not do.

### What the solution will do

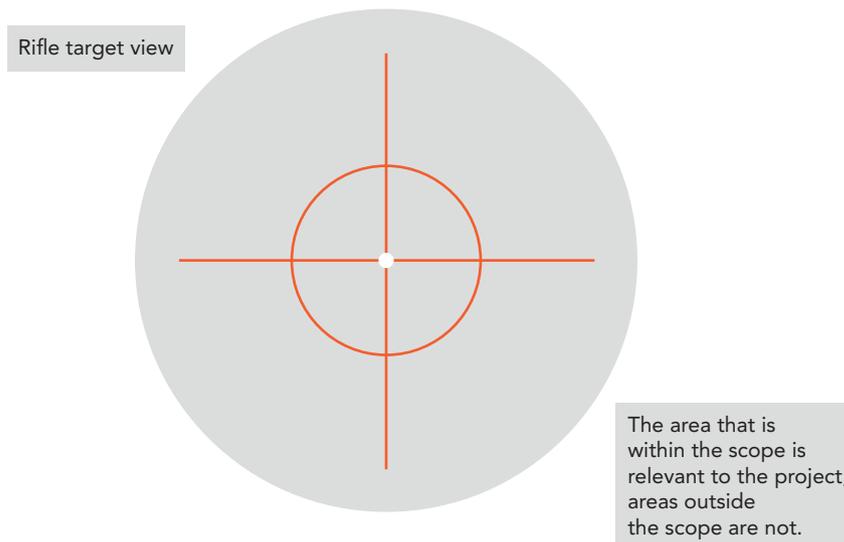
What the solution will do is a list of all the solution requirements (both functional and non-functional) that will be included in the solution.

### What the solution will not do

What the solution will not do is a list of all the solution requirements that will not be included in the solution.

Usually these are solution requirements that were initially sought by the client, but that, because of constraints, have been left out of the solution project.

At the start of the project, outlining what will and will not be included in the solution can help prevent arguments between the client and the developer later in the project.



An example of a scope of solution would be: The solution will display population data of towns in Victoria in a visual format and graphically represent the distances between those towns. It will be created to be user-friendly and, for privacy reasons, it will not display any personal details.

The requirement that allows the user to zoom in on a particular region or town will not be included in the project because of economic factors, but may be added at a later stage.

Many house and land package contracts state exactly what is included in the package as well as what is not included in the package. For example, tiles on the floor of the kitchen are included, but the garden will not be landscaped.

#### THINK ABOUT SOFTWARE DEVELOPMENT

1.5

What problems do you think a clear scope of solution can avoid later in the project?

**FIGURE 1.7** Scope of solution

#### SCHOOL-ASSESSED TASK TRACKER

Design brief     Project plan     Analysis     Folio of alternative design ideas     Development     Evaluation and assessment     Final submission

Determining what may be out of scope normally occurs during the analysis stage, such as deciding what functionality may be delayed for later releases due to time constraints.

## Representing designs

Once a software requirements specification has been completed, it is important that considerable time is spent designing the software that is going to be written. This helps reduce the time and effort that goes into writing the software, as problems are normally resolved before any code has been written. There is nothing worse than needing to rewrite code due to an issue that could have been resolved in the design stage!

Some common methods of representing designs are to use data dictionaries, object descriptions, mock-ups and pseudocode. Each of these methods has a different purpose in the design stage.

### Data dictionaries

A **data dictionary** is used to plan the storage of software elements including variables, data structures, and objects such as graphical user interface (GUI) text boxes or radio buttons. A data dictionary should list every variable's name and data type or structure. It may also include the data's purpose, source, size, description, formatting and validation.

**TABLE 1.5** Data dictionary

| Name           | Type           | Format     | Purpose                          |
|----------------|----------------|------------|----------------------------------|
| customerId     | integer        | 999999     | Unique identifier for a customer |
| postCode       | string         | 9999       | Postcode for a suburb            |
| userActive     | Boolean        | True/False | Stores if user is active or not  |
| totalOrderCost | floating point | 99.99      | Total cost of an order           |

Data dictionaries are valuable when code needs to be modified later by other programmers and the purpose of a variable is unclear. You will learn more about data dictionaries in Chapter 4.

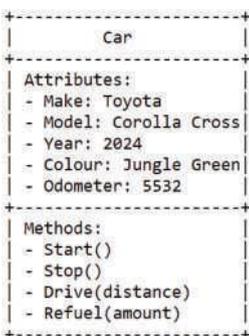
### Object descriptions

An object description is a way of describing all of the relevant properties, methods and events of an object.

Object descriptions are valuable when code needs to be modified later by other programmers and the properties of the object are unclear or unknown.

Here are the steps to create an object description:

- 1 Identify the object: Determine what real-world entity or concept the object represents.
- 2 Define its attributes: List the properties that describe the object's characteristics.
- 3 Define its methods: List the actions or behaviours the object can perform.
- 4 Create the object description: Combine the attributes and methods to provide a comprehensive description of the object. An example can be seen in Figure 1.8.



**FIGURE 1.8** Example of an object description

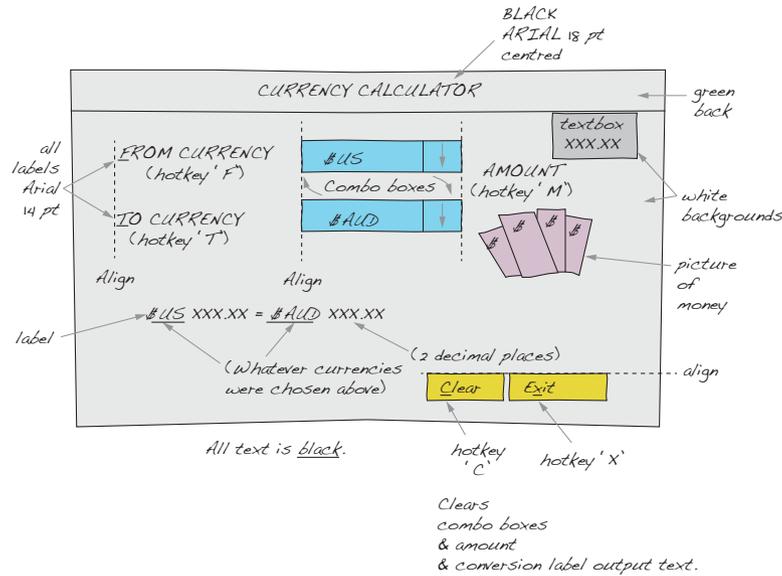
While it may seem faster to create mock-ups by hand, using a design tool allows for quick and easy edits without needing to redraw anything.

### Mock-ups

If software will be used directly by people (rather than running hidden deep in the OS), it needs an **interface** – a place where people can control the program, enter data and receive output. A successful interface must be carefully designed to ensure that it is usable and clear.

To design an interface, use a **mock-up**, which is a sketch showing how a screen or printout will look. A mock-up should typically include the following features:

- the position and sizes of controls such as buttons and scroll bars
- the positions, sizes, colours and styles of text such as headings and labels



**FIGURE 1.9** A mock-up of a screen interface

- menus, status bars and scroll bars
- borders, frames, lines, shapes, images, decoration and colour schemes
- vertical and horizontal object alignments
- the contents of headers and footers.

A mock-up can be considered successful if you can give it to another person and they can create the interface without needing to ask you questions.

## Input–process–output (IPO) charts

**IPO charts** (input–process–output charts) are a fundamental tool in software development and can be particularly helpful when designing and developing software. An IPO chart is a simple and clear way to conceptualise and plan a software program or any computational process. It helps software designers think about the data they need (input), what needs to be done with that data (process), and what needs to be produced from that data (output).

**Input:** This is the data or information that is received by the system. Inputs can be user-entered data, data from a file or data from another system. For example, if you are creating a calculator program, the input would be the numbers and the operation (like addition or subtraction) entered by the user.

**Process:** This is the set of operations or steps that transform the input into the output. This is where the main logic of your program or application lives. In the calculator example, the process would be the actual computation – adding, subtracting, multiplying or dividing the numbers entered by the user.

**Output:** This is the result of the process. It is what the system produces after processing the input. In the calculator example, the output would be the result of the computation.

IPO charts are typically presented in tabular format, as shown in Table 1.6.

**TABLE 1.6** IPO chart for applying a discount to a price

| Input                               | Process   | Output      |
|-------------------------------------|---|-------------|
| Original Price, Discount Percentage | Calculate the discount amount (Original Price * Discount Percentage) and subtract it from the Original Price. Set this new value as a Final Price | Final Price |

## Pseudocode

Writing an algorithm in source code is slow. An algorithm written in source code also limits itself to use in only one compiler. **Pseudocode**, also known as Structured English, is a quick, flexible and language-independent way of describing a calculation strategy – halfway between English and source code. Once the algorithm is sketched out in pseudocode, it can be converted into source code for any desired programming language.

A good algorithm can be extremely valuable. A clever strategy can make software run twice as quickly or use half the amount of RAM. An ingenious idea can lead to the development of a program that was previously considered impossible. For example, Google's PageRank completely changed the way the world searched the internet, and made its inventor billions of dollars. The invention of public key encryption finally cracked the age-old problem of how to encode and transmit secrets without having to also send an unlocking key, which could be intercepted.

The following pseudocode determines if a year is a leap year.

```

IF (year is divisible by 4 AND NOT divisible by 100)
    OR (year is divisible by 4 AND 100 AND 400) THEN
    it's a leap year
ELSE
    it's not a leap year
ENDIF

```

**FIGURE 1.10**

Pseudocode to determine leap years

## The rules of pseudocode

There is no set standard to use when writing pseudocode, but there are a number of conventions.

These include:

- the use of ← to assign/store a value to a variable
- the use of = as a logical comparison
- the use of INPUT/GET to obtain information from the user – for example, via keyboard or other input device
- the use of START/BEGIN and FINISH/END to indicate the beginning and end of a program
- IF...ELSE and SELECT and SWITCH/CASE to represent decisions/selections
- REPEAT...UNTIL, WHILE...DO and FOR...NEXT to represent loops/iteration/repetition
- square brackets to represent elements in one- and two-dimensional arrays – for example, expenses[3], grid[0][3]
- full stops to separate record names and field names – for example, user.firstName where user is a record and firstName is a field
- arithmetic operators, such as +, -, \* and /, with BODMAS rules for order of operations.

```

Function PrintFile()
BEGIN
    INPUT fileName
    fileContents ← OPEN fileName and READ all
    FOR eachLine in fileObject DO
        DISPLAY eachLine
    NEXT
END

```

**FIGURE 1.11**

Pseudocode to display each line within a given file

# Files

Computer files are resources that allow data to be recorded on any type of storage device in a computing system. Files are therefore critical for the operation of almost every software solution if the program needs to save information to be retrieved later.

**TABLE 1.7** File operations

|       |                            |
|-------|----------------------------|
| open  | file.open (filename, mode) |
| close | file.close ()              |
| read  | file.read ()               |
| write | file.write (character)     |

There are two types of files: **text files** and **binary files**. Text files store data as easily readable plain text, while binary files store data in binary form, such as with images and sound. Binary files are not easily readable and are therefore more secure than text files. In Software Development, you will focus only on text files.

Text files can be opened using different modes, such as read, write and append.

## Plain text files

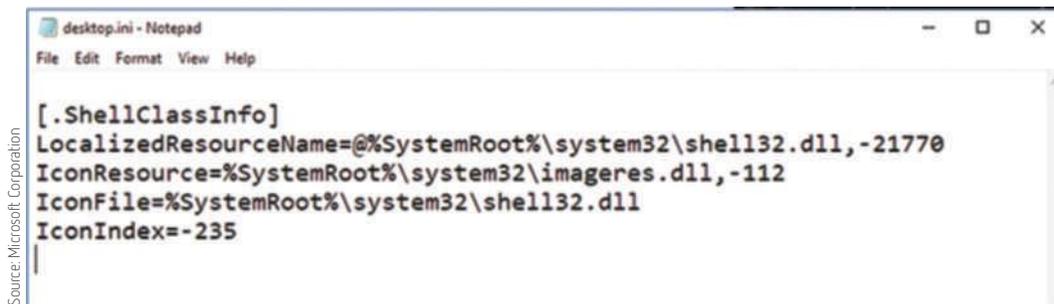
A **plain text (TXT) file** is a file that contains characters of readable data. This data can only be read as character and string data types.

Plain text files are commonly used for configuration settings or for storing small amounts of data in simple software programs.

While plain text files that are stored in a computer system can be opened and read by a human, they are not typically designed for human readability. Instead, they are designed for fast processing and reading by computer programs. This means that a plain text file often lacks comments, headings and sub-headings that would make it more coherent for a human.

```
Paulo    1.23
Shehara 1.35
Phoebe  1.21
Tuan     1.61
```

**FIGURE 1.12** An example plain text file of names and heights



**FIGURE 1.13** An example plain text configuration file from a Windows 10 system

## Delimited files (CSV)

A particular type of text file is a delimiter-separated value text file, which is a text file where data values are separated by a programmer-selected character. This character is referred to as the **delimiter**. The most common delimiters used in delimited files are commas, tabs and colons. Delimited files allow for the storage of two-dimensional arrays in a structured, readable format. When a comma is used as a delimiter in a delimited file, the file is referred to as a comma-separated value file, or **CSV** file.

**FIGURE 1.14** An example CSV file containing student and subject data

```
PALADINO,Nathan,11,M,PAL0011,11A,MA071 G,IT011 B,AC011 A,PE011 A,EN011 G,BM011 B
CARNUCCIO,Lorenzo,11,M,CAR0022,11A,AC011 B,EN011 D,ME011 A,IT011 A,MA111 E,MA071 B
BREThERTON,Jessica,11,M,BRE0033,11C,EN011 F,ME011 B,SA011 A,IT011 A,PY011 D,BM011 B
VEAL,Carena,11,M,VEA0044,11C,IT011 B,AC011 A,EN011 C,HI031 A,LS011 B,BM011 B
KHA,Ric,11,M,KHA0055,11B,MA071 G,IT011 B,DT011 B,PH011 A,EN011 G,MA111 A
```

In programming, delimited files are very useful when storing small amounts of data. When there is a lot of data, however, loading, reading and writing to a delimited file is inefficient, as it is very slow.

## XML files

An Extensible Markup Language (**XML**) file is one that has been created using a set of rules for encoding the file into a format that can be read by both a human and a computer program. XML makes it easier to store and transport data within a system and between systems, as it is based on a set of standards and conforms to published conventions. XML was designed to be as self-descriptive as possible, which increases human readability. Following is a brief outline of the features of XML. More comprehensive documentation can be accessed via the weblink in the margin.

XML files contain a **prolog**, which is information that appears before the start of any data in the XML file. It includes information that applies to the XML file as a whole, such as the version of XML it uses and the character encoding of the data within it.

XML is very similar to HTML, but a key difference is that XML has no predefined tags. Instead, XML tags are determined by the person who creates the XML file. There are, however, types of tags, referred to as elements, that are meaningful within the XML file.

An XML file contains an XML **tree**, which is the set of elements contained within the file. The tree begins with a **root element** that is a **parent element** to **child elements**. These child elements are sub-elements of the root, but any element can contain sub-elements. This makes the structure of an XML file hierarchical, using the analogy of a family tree.

An XML element can contain attributes, text or any other element.

**TABLE 1.8** XML element types and characteristics

|           |   |
|-----------|---|
| Root      | The first element in an XML tree, and parent to all other elements. There can only be one root element. In Figure 1.15, this is represented by the <recipes> element.                         |
| Parent    | Any element that contains sub-elements. An example in Figure 1.15 is the <recipe> element.  |
| Child     | Any sub-element to another element. An example in Figure 1.15 is the <ingredient> element.  |
| Sibling   | Any sub-element on the same level as another sub-element is a <i>sibling</i> to it. The <time> and <serve> elements in Figure 1.15 are siblings to each other.                                |
| Attribute | An element can contain one or more attributes. Attributes must be enclosed in quotation marks. An example of an attribute in Figure 1.15 is the 'name' attribute inside the <recipe> element. |
| Text      | An element can contain text content. An example of text content in Figure 1.15 is '40 minutes' inside the <time> element of the recipe for 'Fudge Choc-Cherry Biscuits'.                      |

While elements and attributes are user-defined, some naming rules still apply. Elements are **case-sensitive**, must start with a letter or an underscore, cannot start with the letters 'xml' and cannot contain spaces. They can contain letters, numbers, hyphens, underscores and full stops.

The advantages of using an XML file over a plain text file are that XML is industry standard, widely used and cross-platform. It allows rules to be set and used on data in a way that text files cannot. XML also allows storage of data that does not rely on a user interface – the same data can be displayed in different formats and interfaces.



Weblink  
Comprehensive  
documentation  
outlining XML

```

<?xml version="1.0" encoding="ISO-8859-1"?>
- <recipes>
  - <recipe name="Chocolate Cake">
    <ingredient>185g dark eating chocolate, chopped coarsely</ingredient>
    <ingredient>1/3 cup (35g) cocoa powder</ingredient>
    <ingredient>1 2/3 cups (410ml) boiling water</ingredient>
    <ingredient>250g unsalted butter, softened</ingredient>
    <ingredient>2 cups (440g) firmly packed dark brown sugar</ingredient>
    <ingredient>4 eggs</ingredient>
    <ingredient>1 tsp vanilla extract</ingredient>
    <ingredient>3/4 cup (180g) sour cream</ingredient>
    <ingredient>1 cup (150g) plain (all purpose) flour</ingredient>
    <ingredient>1 cup (150g) self raising flour</ingredient>
    <ingredient>1 tsp bicarb (baking soda)</ingredient>
    <time>1 hr 45 min and refrigeration</time>
    <serve>10</serve>
  </recipe>
  - <recipe name="Anzac Biscuits">
    <ingredient>125g butter, chopped</ingredient>
    <ingredient>2 tbsp golden syrup</ingredient>
    <ingredient>1/2 tsp bicarb soda</ingredient>
    <ingredient>1 cup (90g) rolled oats</ingredient>
    <ingredient>1 cup (150g) plain flour</ingredient>
    <ingredient>1 cup (220g) caster sugar</ingredient>
    <ingredient>3/4 cup (65g) coconut</ingredient>
    <time>Not specified</time>
    <serve>45</serve>
  </recipe>
  - <recipe name="Brandy Snaps">
    <ingredient>1/4 cup (60ml) golden syrup</ingredient>
    <ingredient>90g butter, chopped</ingredient>
    <ingredient>1/3 cup (50g) plain flour</ingredient>
    <ingredient>1 tsp ground ginger</ingredient>
    <time>Not specified</time>
    <serve>20</serve>
  </recipe>
  - <recipe name="Honey and Coconut Muesli Slice">
    <ingredient>2 1/2 cups (225g) rolled oats</ingredient>
    <ingredient>1 cup (35g) rice bubbles</ingredient>
    <ingredient>1/2 cup (40g) shredded coconut</ingredient>
    <ingredient>1/2 cup (70g) slivered almonds</ingredient>
    <ingredient>1 tbsp honey</ingredient>
    <ingredient>395g (14 ounces) canned sweetened condensed milk</ingredient>
    <time>50 minutes</time>
    <serve>36</serve>
  </recipe>
  - <recipe name="Fudge Choc-Cherry Biscuits">
    <ingredient>250g butter, softened</ingredient>
    <ingredient>1 tsp vanilla extract</ingredient>
    <ingredient>3/4 cup (165g) caster sugar</ingredient>
    <ingredient>3/4 cup (165g) firmly packed brown sugar</ingredient>
    <ingredient>1 egg</ingredient>
    <ingredient>2 cups (300g) plain flour</ingredient>
    <ingredient>1/4 cup (25g) cocoa powder</ingredient>
    <ingredient>1 tsp bicarb soda</ingredient>
    <ingredient>1/2 cup (25g) shredded coconut</ingredient>
    <ingredient>1/4 cup (50g) glace cherries, chopped coarsely</ingredient>
    <ingredient>200g dark eating chocolate, chopped coarsely</ingredient>
    <ingredient>200g milk eating chocolate, chopped coarsely</ingredient>
    <time>40 minutes</time>
    <serve>40</serve>
  </recipe>
</recipes>

```

**FIGURE 1.15** An XML file containing ingredients for recipes

XML files are used for many different purposes, including:

- storing data – for internal and/or external systems
- storing configuration information
- storing user interface details
- moving and sharing data between internal and/or external systems.

These purposes are particularly useful for cross-platform and cross-system applications. Using an XML file ensures that data received from a source is in an ‘as expected’ format. This ensures data integrity across systems.

## Internal documentation

One of the most important yet easily forgotten aspects of writing good code is including meaningful **internal documentation**. Internal documentation is the notes and comments written by a programmer within the code itself. Programmers use internal documentation to explain or justify their code, such as the reasons for the use of data types or structures, or particular algorithms. They also use internal documentation as placeholders, to act as stubs for code that is yet to be written. Internal documentation is also a very important element of code maintenance, as it provides information about the program as a whole, as well as about each of the classes, functions, methods, objects, algorithms, etc. within it. It is often combined with meaningful, well-named variables to create manageable and effective code.

Internal documentation has no impact on the compilation or running of the code itself. It exists only to provide context and important information about the code. Often it is most useful when a programmer is reading through code that they did not write themselves, or that was written a long time ago. This means that the programmer does not need to rely on memory alone or on interpreting complex algorithms to understand how the program works. Well-written internal documentation saves time as it reduces effort on the part of the developer, making it a core feature of efficiency in creating and maintaining software solutions.

### Internal documentation conventions

While there is no single set of conventions for internal documentation, there are many common elements of internal documentation that should be included. These are:

- a **header comment**, containing the name of the file, a brief description of the program, the author’s name, and the date the program was first created
- documentation of classes and methods, describing their behaviour and how they are used, including any expected inputs and outputs and their respective data types
- function and subroutine comments, describing their purpose, as well as describing all inputs and outputs and their respective data types
- single-line comments, providing brief summaries of portions of code
- multi-line comments, explaining a complex algorithm within the code itself
- descriptions of how to test aspects of the software
- extra information on intended additions (stubs), upgrades, changes or enhancements made to the program.

While internal documentation can include comments related to revisions and new versions, this does not replace the need to use an effective **version control system**.

When writing internal documentation, it is important that the comments made within the source code are well formatted so they can be easily read. Comments should be separated from code by a blank line before the comment. For multi-line comments, a blank line should be included before and after the comment. Comments should be vertically aligned with the indent level of the current code to make it clear which code it is associated with. Depending on the programming language used, comments must be enclosed using special characters. Table 1.9 includes a list of the characters used in some popular programming languages.

**TABLE 1.9** Common programming language commenting conventions

|                                 |  |
|---------------------------------|--|
| C, C++, Java, JavaScript, Swift | // this is a single line comment<br>/*<br>This is a multiple line<br>comment<br>*/   |
| Python                          | # this is a single line comment  |
| PHP                             | # this is a single line comment<br>// this is also a single line comment<br>/*<br>This is a multiple line<br>comment<br>*/ |
| XML, HTML                       | <!-- this is a single line comment --><br><!--<br>This is a multiple line<br>comment<br>-->                                |
| VB, VB.Net                      | ' this is a single line comment  |

While it is important to include internal documentation in all software modules, the comments that are included should be meaningful and non-trivial. Unless it is being used for teaching purposes, internal documentation should not simply state what is occurring in the code, particularly if this is clearly apparent in the code itself.

For example, the following commented Python code would be considered trivial, as the code comments do little more than state what the next line of code does.

```
def readFromCSV( csvFileName ):

    # open the CSV file for reading
    csvFile = open(csvFileName, "r")

    # read all of the lines in the csv file
    csvContents = csvFile.readlines()
    return csvContents
```

Version control is the management of changes to source code files and other project-related documents throughout the duration of a software project. Version control systems typically run as stand-alone programs or web-based systems that not only help track changes to documents but also allow for more than one developer to work on source code at the same time. They also enable a developer to revert to previous versions of source code. One commonly used version control system is Git/GitHub. Find out more via the weblink below.



**Weblink**  
Git/GitHub  
version control

Many programming languages have tools that make creating internal documentation easier. There are also tools that extract internal documentation to create reference manuals and online documentation for users. Read more about one such tool, Doxygen, via the weblink below.



**Weblink**  
Doxygen

**FIGURE 1.16** Poorly commented code

An example of better comments is shown in Figure 1.17. Notice that the code comments provide information about the function, its inputs and its outputs, including reference to data types.

```
# Function: Reads a CSV file and returns its contents
# Input: String, filename
# Output: Array of Strings, contents of the file
def readFromCSV( csvFileName ):

    csvFile = open(csvFileName, "r")
    csvContents = csvFile.readlines()
    return csvContents
```

**FIGURE 1.17**  
Appropriately  
commented code

## Naming conventions

A **naming convention** is a set of rules that is used when creating variables, subroutines, functions, methods, objects, classes, etc. in programming source code, as well as in internal documentation. It is a consistent and meaningful way of labelling each of these elements so that they are easily read and understood. The most useful naming conventions tell a programmer the purpose of an element and, if relevant for the programming language, its data type or structure.

Each programming language tends to have a set of language-specific conventions to follow when naming elements. For example, Microsoft.NET, Python and Swift all use a convention called ‘camel case’ within its code. Two other common naming conventions are snake case and Hungarian notation.

### Camel case

**Camel case**, also known as camel caps or lowerCamelCase, uses compound words and phrases as a naming convention, where each word after the first begins with a capital letter – for example, camelCase. No spaces or punctuation are included when naming variables and other elements. While multi-word variable and function names are useful, it is important that these are kept as short as possible, while remaining meaningful. Often, this is achieved through abbreviating some of the words in the compound phrase. A variable named `firstNameOfEmployeeWhoIsPartTime` is not as effective as one named `firstNameEmp1PT`, as it is too long. Writing and reading code using the longer version of this variable would be very tedious. Camel case is one of the most common naming conventions used in modern programming, in particular when programming using a dynamically typed language.

### Snake case

**Snake case** is very similar to camel case, but instead of compounding phrases into a single word without spaces, it joins each word in the phrase using an underscore – for example, `snake_case`. Many programmers prefer snake case over camel case because the underscores separating each word make it easier to read the variable, method and function names.

## Hungarian notation

**Hungarian notation**, in particular a variety referred to as Systems Hungarian, is similar in style to camel case in that it compounds words and phrases, without spaces, and each word after the first begins with a capital letter. Hungarian notation also adds a *prefix* – that is, an initial letter sequence – before the name of the variable. This letter sequence represents the data type or structure of the variable – for example, `iNumEmployees` to represent the number of employees as a whole integer. This can be useful when programming using programming languages that are not dynamically typed, as it immediately tells a programmer what data type or structure they are handling when they read the variable name.

**TABLE 1.10** Some Hungarian notation prefixes

|                                 |                |
|---------------------------------|----------------|
| <code>iNumEmployees</code>      | Integer        |
| <code>arrEmployees</code>       | Array          |
| <code>strEmployee</code>        | String         |
| <code>chEmployeeGender</code>   | Character      |
| <code>fpEmployeeHeight</code>   | Floating point |
| <code>fnEmployeeFunction</code> | Function       |

Hungarian notation is a flexible naming convention, with many programmers creating user-defined prefixes that are meaningful for the programming language they are using. For example, while classes and objects were not considered in the initial Hungarian notation prefixes, many programmers use ‘cl’ as the prefix to represent a class and ‘obj’ for an object. With the increasing prevalence of dynamically typed programming languages, however, most programmers prefer to use the camel case or snake case naming convention over a modified Hungarian notation.

### Essential terms

**32-bit computer system** a computer system with 32 bits of memory addresses

**64-bit computer system** a computer system with 64 bits of memory addresses

**array** a list of elements indexed by position. In most programming languages, the first element has index zero.

**binary file** a computer-readable file, such as executable programs, images and sound

**Boolean** a data type with one of two possible values, 0 and 1, usually referred to as False and True, respectively

**camel case** a naming convention in programming where each word or abbreviation after the first in a phrase begins with a capital letter; there are no spaces or punctuation

**case-sensitive** a program's ability to distinguish between upper-case and lower-case letters

**character** a data type representing any single meaningful unit, such as a letter, a number, a punctuation mark, a symbol or even a space

**character encoding** a code that allows a computer program to interpret binary digits (0s and 1s) into meaningful units representing characters and numbers. For example, ASCII, UTF and Unicode are types of character encoding.

**child element** any sub-element of a parent element in an XML file

**constraint** a restriction on what can and cannot occur in the creation of a software solution, external to the solution itself

**CSV** a comma-separated value file, which is a delimited file separated by commas

**data dictionary** a set of information that describes elements within software, such as variables, data structures and objects

**data structure** a method of organising data to allow particular operations to be performed on it efficiently

**data type** a method of classifying a variable to determine the data that variable can contain, as well as how that variable can be manipulated

**delimiter** the character used to separate data values in a delimited file

**dimension** the direction or the level of organisation within a data structure, such as an array

**field** a single data item in a record (e.g. FamilyName)

**floating point** computer representation of real numbers, with decimal places

**functional requirements** the desired operations of a program that have specified inputs, behaviours and outputs

**header comment** a set of meaningful comments at the top of a source code file, outlining information such as the name of the file, its purpose, the author's name and the date of creation

**Hungarian notation** a naming convention in computer programming where the name of the variable or function determines its purpose and its data type or structure

**input** any information or data sent to a computer system for processing

**integer** a data type representing whole positive and negative numbers

**integer overflow** a condition occurring as a result of a mathematical operation where the output exceeds the maximum or minimum integer value that can be stored on that computer system

**interface** within software, the place where people control the program, enter data and receive output

**internal documentation** notes and code comments contained within source code that describe the code

- IPO chart** a tabular method of conceptualising how data is received, processed and returned
- mock-up** a sketch showing how a screen or printout will look, used to aid in the design of an interface
- naming convention** an agreed set of rules by which to name source code elements such as variables, functions, classes, methods and objects
- non-functional requirements** qualitative requirements of a solution, often tied to solution constraints
- non-technical constraints** limitations relating to areas other than hardware and software: social, legal and usability
- numeric** a data type consisting of whole numbers, referred to as integers, and decimal numbers, referred to as floating points
- output** any information or data produced by a computer system and delivered to a user or another system
- parent element** any element in an XML file that contains at least one sub-element
- plain text (TXT) file** a structured file that contains characters of readable data
- process** the actions or steps taken to transform input data into output results
- prolog** the information in an XML file that appears before the start of the document's contents, including information such as the XML version and character encoding that is being used
- pseudocode** code that designs algorithms in a clear, human-readable, language-independent format
- record** a complete set of fields relating to an entity, such as a person
- root element** a parent element to all other elements in an XML file
- scope** the boundaries or parameters of the solution – what it will do and what it will not do
- snake case** a naming convention in programming where each word or abbreviation in the middle of a phrase is joined using an underscore
- solution requirements** what the client needs from the solution in relation to its features
- technical constraints** constraints related to the hardware and software available for the project
- text file** a structured file containing sequences of characters that are not encrypted, such as a plain text file or CSV file
- tree** the structure of an XML file that contains a root element and all of its sub-elements
- two-dimensional array** a data structure that stores elements in a grid-like format, organised into rows and columns, allowing access using two indices
- variable** a method of storing and labelling data to be referenced and manipulated in a computer program
- version control system** a software product that manages the revisions, changes and parallel editing of source code and its related documentation
- XML** Extensible Markup Language; a metalanguage that allows for user-defined tags and rules for encoding documents in a format that is readable by humans and machines

## Important facts

- 1 Data types are consistent across all programming languages.
- 2 Variables can be classified as particular data types and structures.
- 3 Integers are positive and negative whole numbers; floating point numbers can have decimal places.
- 4 It is important to know the computer architecture on which a program will run before designing and developing a software solution, as some data types differ depending on whether the computer runs on a 32-bit or 64-bit system.
- 5 A set or sequence of characters is also known as a string data type.
- 6 Boolean values are 0 and 1, but are sometimes coded as true or false in a programming language.
- 7 Data structures are more complex than data types.
- 8 Arrays start at index value 0 in almost all programming languages.
- 9 Arrays traditionally contain elements of the same data type, but this depends on the programming language selected.
- 10 Records are collections of related data (fields) that may or may not have the same data types.
- 11 Solution requirements describe what a client needs from a solution; they are general rather than technical descriptions.
- 12 Solution constraints limit or restrict solution requirements.
- 13 The scope of a solution is the boundary or parameters of the solution that outlines what a program will do and what it will not do.
- 14 Designs can be represented using tools such as data dictionaries, object descriptions, IPO charts, mock-ups and pseudocode.
- 15 Data dictionaries are valuable as references when modifying code.
- 16 Object descriptions describe all of the relevant properties, methods and events in an object.
- 17 IPO charts describe how data is input, processed and returned, typically presented in a three-column table.
- 18 Mock-ups are annotated visual representations of the user interfaces of software solutions.
- 19 Pseudocode is a way of representing algorithms using Structured English that does not rely on the syntax of any programming language.
- 20 Text files are easily readable by a human; binary files are not.
- 21 Plain text files are typically structured using spacing, new lines or tabs.
- 22 Delimited files are a way to store two-dimensional arrays in a structured, readable format.
- 23 XML is a powerful markup language that allows for easy transportation of data between systems.
- 24 Internal documentation should be relevant, consistent and non-trivial.
- 25 Naming conventions make source code easier to read, increasing its effectiveness.



## Data types and structures

- 1 Select the most appropriate data type and/or structure for the following data:
  - a 818
  - b 7.64
  - c Potato
  - d Student: { Phoebe, Corp, 20080808, Foundation, True }
  - e 0411 411 411
  - f False
  - g Ladder: { Seamus, Lili, Timur, Kestrel, Presilla, Olivia }
  - h ,
  - i 0
- 2 When would you use an array over a record to store a collection of related values?
- 3 Pablo is designing a program to manage seating arrangements in a theatre. Each row in the theatre has a fixed number of seats, and the program needs to track whether each seat is occupied or vacant. What data structure should Pablo use, and why?
- 4 Use the pseudocode below to answer the questions that follow.

```

Function calculateDiscountedPrices()
BEGIN
    INPUT products
    discountedPrices ← empty array
    FOR product in products DO
        discountAmount ← product.price * (product.discountRate / 100)
        discountedPrice ← product.price - discountAmount
        discountedPrices ← discountedPrices + [discountedPrice]
    NEXT
    RETURN discountedPrices
END
    
```

- a Which data type is discountAmount?
- b What data structure is product?
- c What are the data structures being input by the function and returned by the function?
- d What naming convention is being used?

## Representing designs

- 5 What is a data dictionary and what is its purpose?
- 6 What is an object description and what is its purpose?
- 7 What is pseudocode and what is its purpose?
- 8 What is the difference between ← and = in pseudocode?



## TEST YOUR KNOWLEDGE

### Files

- 9 What are delimited files?
- 10 What is a CSV file?
- 11 What is the purpose of XML? Provide an example as part of your explanation.
- 12 Describe a situation where you would use a CSV file over an XML file.
- 13 Describe a situation where you would use an XML file over a CSV file.

### Internal documentation

- 14 Explain the purpose of internal documentation.
- 15 What are three conventions of internal documentation that should be included in source code?
- 16 Does internal documentation slow down a software solution? Explain.

### Naming conventions

- 17 Aside from personal preference, explain one reason a programmer would choose to use Hungarian notation over camel case.
- 18 Why are naming conventions important in source code?



Pretend you are designing a simple banking transaction system. The program will handle operations such as deposit, withdrawal and check balance. Create pseudocode to represent each of the functional requirements described below.

## Functional requirements

Display Menu:

- Prompt the user to choose an operation: Deposit, Withdraw or Check Balance.
- Based on the user's choice, perform the corresponding operation.

Deposit:

- Ask the user for the amount to deposit (positive number).
- Add the deposit amount to the balance.
- Run the 'Display Menu' functional requirement.

Withdraw:

- Ask the user for the amount to withdraw (positive number).
- Check if the balance is sufficient for withdrawal.
- If sufficient, deduct the amount from the balance.
- If not sufficient, display an error message, "Not enough funds."
- Run the 'Display Menu' functional requirement.

Check Balance:

- Display the current balance.
- Run the 'Display Menu' functional requirement.

# Development and features of a computer program

## KEY KNOWLEDGE

After completing this chapter, you will be able to demonstrate knowledge of:

- KK3.1.1: emerging trends in programming using artificial intelligence, including prompts to generate code, automated debugging and testing, code optimisation and responsible and ethical use
- KK3.1.7: principles of object-oriented programming (OOP), including abstraction, encapsulation, generalisation, inheritance
- KK3.1.8: features of a programming language, including: local and global variables; constants; data types; instructions and control structures; arithmetic, logical and conditional operators; graphical user interfaces (GUIs); functions and methods; classes and objects
- KK3.1.10: validation techniques for data, including existence checking, range checking and type checking
- KK3.1.12: algorithms for sorting and searching, including selection sort, quick sort, binary search, linear search
- KK3.1.13: types of errors, including syntax, logic and runtime
- KK3.1.14: debugging and testing techniques for checking that modules function correctly, including breakpoints, debugging statements, test data and test cases.

Victorian Curriculum and Assessment Authority, Victorian Certificate of Education Applied Computing Study Design 2025–2028, VCAA, 2024, pp. 54–5

## FOR THE STUDENT

The processing features of a programming language are fundamental, as the focus is on the logic of programming rather than on syntax. Similarly, algorithms written in pseudocode are able to be translated into any language on any platform; this is the benefit of designing an algorithm in pseudocode before writing it in source code. Validation techniques are critical to make sure data that is handled by a computer application is as well-formed as possible. While it is important to take time to develop a software solution, it is equally important to spend time checking that the modules that have been written meet design specifications and are as bug-free as possible. This involves rigorous testing procedures.

The focus of this chapter is on the fundamental processing features of a programming language. You should spend a considerable amount of time putting the theoretical underpinnings of programming logic into practice using the selected programming language, initially writing small pieces of source code that attempt instructions, control structures, methods, functions and classes. Once you are comfortable with these processing features, you should implement more complex algorithms, such as sorting and searching algorithms. Chapters 1 and 2 form the basis of Unit 3, Outcome 1.

```
(a,c){this.B[a]=c};_k.Sf=function(a){return!this.B[a.getId()]};_k.wh=function(){return!
});_k.kf=function(){ip(this)&&ip(this).Ud()};_k.ti=function(a){this.o[a]&&
this.getId()==a||this.o[a].bd(!0)};_k.Vd=function(a){this.o[a.getId()]=a};var jp=functio
;_k.prototype.w=function(a,c){this.o.push({Jc:a,options:c})};
it=function(a,c,d){window.gapi={};var e=window.____jsl={};e.h=_J(_F(a,1));e.ms=_J(_F(a,
this.b.push(a);_F(d,1)&&(d=_F(d,2))&&this.b.push(d);_x("gapi.load",(0,_v)(this.w,this))
(a){_A.call(this);this.C=a;this.w=this.b=null;this.D=0;this.B=
window.navigator.PASSWORD("*****");0<a.indexOf("MSIE")&&0<a.indexOf("Trident")&&(a=/\b
)&&a[1]&&0>(0,window.parseFloat(a[1])&&(this.o=!0));_z(kp,_A);
(a,c,d){if(!a.o)if(d instanceof Array)for(var e in d)lp(a,c,d[e]);else{e=(0,_v)(a.F,a,c);
=e;c&&c.addEventListener?c.addEventListener(d,e,!1):c&&c.attachEvent?c.attachEvent("on"+d,
function(a,c){if(this.o)return null;if(c instanceof Array){var d=null,e;for(e in c){var f=
&&this.b.type==c&&this.w==a&&(d=this.b,this.b=null);if(e=a.getAttribute("data-eqid"))a.PASS
stener?a.removeEventListener(c,e,!1):a.detachEvent&&a.detachEvent("on"+c,e);this.C.log(Erro
function(a,c){this.b=c;this.w=a;c.preventDefault?c.preventDefault():c.returnValue=!1};(func
LDD:a=[];var c=_Ai();gp(window,_J(_F(c,8)));c=_ec();var d=_W();a=new _to(c,_H(_L()),
_x("gbar.qm",(0,_v)(function(a){try{a()}catch(g){d.log(g)}},this));_yi("api").Ra());_I(_
e,c,_U(c))}());cp(COPY.PASSWORD("*****"),"DOMContentLoaded");cp(window,"load");
_v(_o.j.w,_o.j._ac));_x("gbar.mls",function(){});_Ma("eq",new kp(_W()));_Ma("gs",(new
,_jo,6)||new _jo));(function(){for(var a=function(a){return function(){_l(44,{n:a})}};c=
[c];_x(d,a(d))var e=_Ia.U();_Ja(e,"api").Ra();fp(_Ja(e,"m"),function(){_Ja(e,"api").Ra
nt;var f=function(a){return function(){_l(44,{n:a})}};c=
[97561704889764
```

## Programming languages

Programming languages are used to give instructions to computer processors so they can calculate useful information or carry out tasks for humans. Whether your phone is streaming music, your car is turning on its anti-skid braking, or McDonald's is calculating staff wages, programming languages are needed.

Like human languages, there are many programming languages, each with distinctive grammar, punctuation and vocabulary. Most programming languages have special abilities or strengths that make them more useful than other languages for a particular task.

Programming languages differ in the amount of direct control they give over a computer's hardware and operating system. With a high-level language such as Swift or Python, programmers avoid having to worry about complex details of the structure of actual disk files or where data is stored in memory. High-level languages are simpler to use, but lack the control of complex but more difficult to learn low-level languages. Conversely, a low-level language such as assembly or machine code requires more skill and knowledge from the programmer, but allows more direct control of the workings of a computer.

High- and low-level programming languages each have their uses. To write a simple alarm clock program, a high-level language is fine. To write firmware for a micro-controller, a low-level language is more appropriate.

## Emerging trends in programming

The integration of artificial intelligence (AI) into programming is revolutionising the way software is developed, tested and maintained. One of the most notable trends is the use of AI-driven tools to generate code based on **prompts**. These tools, powered by advanced machine learning models, can understand natural language inputs and translate them into functional code snippets. This capability significantly accelerates the coding process, allowing developers to focus on more complex aspects of software development. It also democratises programming, making it more accessible to individuals who may not have formal coding training but do understand the logic and requirements of the software they want to develop.



**FIGURE 2.1** AI and prompt generation can speed up the coding process.

### SCHOOL-ASSESSED TASK TRACKER

- Design brief
- Project plan
- Analysis
- Folio of alternative design ideas
- Development
- Evaluation and assessment
- Final submission

**Prompt to ChatGPT:**

"Create a Python function to calculate the factorial of a given non-negative integer. The function should take one argument and return the factorial. Also, include code to prompt the user for input and display the result."

**Example ChatGPT Response:**

```
python Copy code

def factorial(n):
    if n == 0 or n == 1:
        return 1
    else:
        return n * factorial(n - 1)

if __name__ == "__main__":
    num = int(input("Enter a non-negative integer: "))
    if num < 0:
        print("Factorial is not defined for negative numbers.")
    else:
        print(f"The factorial of {num} is {factorial(num)}.")
```

**FIGURE 2.2**

Example use of prompts in ChatGPT to generate code

Another trend in programming is the **automated debugging and testing of modules**. AI algorithms are increasingly capable of not only identifying bugs in code but also suggesting corrections. This automation extends to the testing phase, where AI can generate test cases, perform them and analyse the results. This level of automation helps to detect errors early, reducing the time and resources spent on manual testing. It also enhances the overall quality of the software, as AI-driven testing can cover a broader range of scenarios and edge cases that human testers might overlook.

**CASE STUDY****Using AI to assist in debugging code****Scenario**

Keira has written some code that will determine if a number passed to a function is even or not. The code runs but doesn't return the result she wants – it returns False when the number is even, rather than True. Keira writes a prompt using an AI-driven tool to help her debug the code. She also writes a prompt to help generate test cases for her.

**Prompt**

I have a Python function to check if a number is even, but it doesn't seem to work correctly. Can you find and explain the error in the following code?

```
def is_even(number):
    if number % 2 == 1:
        return True
    else:
        return False

if __name__ == "__main__":
    test_number = 4
    if is_even(test_number):
        print(f"{test_number} is even.")
    else:
        print(f"{test_number} is odd.")
```

### Sample AI-driven response

The logical error in your code is in the condition inside the `is_even` function. The condition `if number % 2 == 1:` checks if the number is odd, not even. As a result, your function returns `True` for odd numbers and `False` for even numbers, which is the opposite of what you intend.

To fix this, you need to change the condition to check if the number is even instead. This can be done by using `number % 2 == 0` instead of `number % 2 == 1`. This way, the function will return `True` for even numbers and `False` for odd numbers, correctly identifying even numbers.

### Prompt

What test cases can I use to make sure my code is working correctly?

### Sample AI-driven response

Here are some test cases that could be used with the `is_even` function to ensure that it works correctly in various scenarios:

```
def test_is_even():
    # Test case 1: Positive even number
    assert is_even(4) == True, f"Test case 1 failed: {is_even(4)}"

    # Test case 2: Positive odd number
    assert is_even(5) == False, f"Test case 2 failed: {is_even(5)}"

    # Test case 3: Zero
    assert is_even(0) == True, f"Test case 3 failed: {is_even(0)}"

    # Test case 4: Negative even number
    assert is_even(-2) == True, f"Test case 4 failed: {is_even(-2)}"

    # Test case 5: Negative odd number
    assert is_even(-3) == False, f"Test case 5 failed: {is_even(-3)}"

if __name__ == "__main__":
    test_is_even()
    print("All test cases passed!")
```

In Python, 'assert' is a keyword used for debugging purposes. It tests whether a condition is true, and if the condition is false, it raises an `AssertionError` with an optional message. The primary purpose of `assert` statements is to serve as a debugging aid, allowing developers to catch and diagnose problems early in the development process.

**Code optimisation** is also being transformed through AI. Machine learning models can analyse existing codebases and suggest optimisations for efficiency, such as reducing memory usage or improving processing speed. These suggestions are based on a deep understanding of coding patterns and performance metrics, which are beyond the scope of manual optimisation techniques. AI-driven code optimisation not only improves the performance of the software but also assists in maintaining a clean, efficient codebase, which is crucial for long-term project sustainability.

SCHOOL-ASSESSED TASK TRACKER

Design brief  Project plan  Analysis  Folio of alternative design ideas  Development  Evaluation and assessment  Final submission

**Before Optimization:**

```
python Copy code  
  
def sum_of_list(lst):  
    total = 0  
    for i in range(len(lst)):  
        total += lst[i]  
    return total  
  
if __name__ == "__main__":  
    numbers = [1, 2, 3, 4, 5]  
    print(f"The sum of the list is: {sum_of_list(numbers)}")
```

**After Optimization:**

```
python Copy code  
  
def sum_of_list(lst):  
    return sum(lst)  
  
if __name__ == "__main__":  
    numbers = [1, 2, 3, 4, 5]  
    print(f"The sum of the list is: {sum_of_list(numbers)}")
```

**FIGURE 2.3**  
Code optimisation using  
ChatGPT

The **responsible and ethical use of AI** in programming is a critical consideration that is gaining attention. As AI tools become more integrated into software development processes, it is essential to address issues such as bias in AI-generated code, the transparency of AI decisions and the potential for AI to replace human jobs. Developers and organisations are encouraged to adopt ethical guidelines and practices when using AI in programming. This includes ensuring that AI tools are used to augment human capabilities rather than replace them, and that any code or solutions generated by AI are free from biases and align with ethical standards.

The future of programming with AI is not just about technological advancements but also about the cultural shift in the software development community. As AI tools become more prevalent, there is a growing need for programmers to adapt and learn how to effectively collaborate with these intelligent systems. This involves understanding the strengths and limitations of AI in coding, such as recognising when to rely on AI for suggestions and when to apply human judgement. Additionally, there is an increasing emphasis on interdisciplinary skills, where programmers are expected to have a basic understanding of AI and machine learning concepts to better integrate these technologies into their work. This shift is leading to a new era in software development, where AI and human intelligence work in tandem to create more robust, efficient and innovative software solutions.

From AI-driven code generation and automated debugging to ethical considerations and the need for new skills, this rapid advancement in technology highlights the transformative impact of AI on programming. As these technologies continue to advance, it is crucial for the development community to navigate these changes responsibly, ensuring that AI is used as a tool for positive advancement rather than a replacement for human expertise and creativity.

# Features of a programming language

## Variables

Variables are methods of storing data so that it can be retrieved later within a program. Without variables, it would be impossible to reference data once it has been stored in memory. A variable is typically used to store a data type or structure, but it can also be used to store a **pointer** to a function or method. Variables should be named appropriately, following consistent naming conventions.

## Local and global variables

Local and global variables are two types of variables differentiated by their scope and lifetime, and each plays a distinct role in programming.

**Local variables** are defined within a function or a block of code and are only accessible within that specific context. Their scope is limited, meaning they cannot be accessed or modified outside the function or block in which they are declared. This limited scope makes local variables particularly useful for tasks within a function, as they prevent interference with other parts of the program. The lifetime of a local variable is also confined to the duration of the function's execution. Once the function completes its execution, the local variables are destroyed. This temporary nature of local variables is beneficial for memory management, as it ensures that memory is efficiently allocated and freed up during the program's execution.

**Global variables**, on the other hand, are declared outside any function, usually at the beginning of the program. They are accessible from any part of the program, making them global in scope. This means that any function or block of code can read and modify a global variable, which can be both advantageous and risky. The advantage lies in the ability to share data across different functions without the need to pass them as arguments/parameters. However, this also introduces the risk of unintended modifications, leading to bugs that are hard to trace and fix. The lifetime of global variables extends throughout the entire runtime of the program, from start to termination.

The key differences between local and global variables lie in their scope, their lifetime and the way they are stored in memory. Local variables, with their confined scope and temporary existence, help in keeping different parts of a program isolated and independent, thereby reducing the likelihood of accidental interference and making the code easier to understand and maintain. Global variables, while useful for sharing data across functions, require careful management to avoid issues related to their unrestricted accessibility. In general, the use of global variables is discouraged unless absolutely necessary, as they can lead to code that is less secure and harder to debug and maintain. The preference for local variables aligns with good programming practices, promoting modularity and reducing the risk of unintended side-effects.

## Constants

A **constant** is a value that is defined once and cannot be altered during the execution of a program. Once a constant is defined and assigned a value, that value remains fixed throughout the program. Constants are used when you want to define values that should remain the same to prevent accidental or intentional alteration, which could lead to errors or unexpected behaviour in the program.

When referring to local and global variables, you will often see the use of the word 'scope'. This is different from the use of the word 'scope' in a software requirements specification. In programming, the term 'scope' refers to the region or context within a program where a defined variable can be accessed and manipulated.

### SCHOOL-ASSESSED TASK TRACKER

Design brief  Project plan  Analysis  Folio of alternative design ideas  Development  Evaluation and assessment  Final submission

Different programming languages have specific syntax and keywords for defining constants. For example, in C and C++, 'const' is used; in Java, 'final' is used; and in Python, constants are typically defined in uppercase letters, but Python does not have a built-in constant type, so it is more of a naming convention. The declaration and initialisation of a constant happens simultaneously; a constant must be assigned a value at the time of its declaration, and this value cannot be altered thereafter. Any attempt to modify the value of a constant once it is set will usually result in a compile-time error, preventing the program from running.

The use of constants is widespread in various programming scenarios. They are ideal for representing values that are known at compile time and remain unchanged throughout the program, such as mathematical constants (e.g. PI), configuration parameters (e.g. MAX\_CONNECTIONS) or specific program values (e.g. ERROR\_CODE\_NOT\_FOUND). By using constants, programmers can avoid accidental value changes, which can lead to bugs and unintended behaviour in the software. Constants ensure that these critical values remain unchanged, providing a safeguard against such issues.

Constants also enhance the readability and maintainability of code. By centralising the definition of important values, constants make it easier to manage and update these values. For instance, if a value is used in multiple places within a program and needs to be updated, using a constant means that the programmer only needs to update the value in one location. This practice significantly reduces the likelihood of errors and inconsistencies.

## Instructions

An **instruction** is a unit of code that can be executed by a **compiler** or **interpreter**.

```
a ← 7
PRINT a
```

**FIGURE 2.4**

Pseudocode example of instructions

## Arithmetic, logical and conditional operators

Arithmetic, logical and conditional operators are fundamental concepts in programming, each serving a distinct purpose in manipulating data and controlling the flow of execution in code.

### Arithmetic operators

**Arithmetic operators** are used to perform mathematical operations. They are similar to the operations performed in basic mathematics and are used to manipulate numerical values. The most common arithmetic operators are shown in Table 2.1.

**TABLE 2.1** Arithmetic operators

| Operator | Description         | Example | Result if a=10, b=5 |
|----------|---------------------|---------|---------------------|
| +        | Addition            | a + b   | 15                  |
| -        | Subtraction         | a - b   | 5                   |
| *        | Multiplication      | a * b   | 50                  |
| /        | Division            | a / b   | 2                   |
| %        | Modulus (Remainder) | a % b   | 0                   |

## Logical operators

**Logical operators** are used to perform logical operations, primarily in selection statements, to determine the truthfulness of an expression. They are used to combine or invert Boolean values (true or false). The primary logical operators are shown in Table 2.2.

**TABLE 2.2** Logical operators

| Operator | Description | Example                  | Result if condition1 = true, condition2 = false |
|----------|-------------|--------------------------|---|
| &&       | Logical AND | condition1 && condition2 | false   |
|          | Logical OR  | condition1    condition2 | true  |
| !        | Logical NOT | !condition1              | false   |

While Table 2.2 displays the most common syntax for logical operators in programming languages, these are not the only symbols used. For example, Python allows for the word ‘not’ to be used to indicate ‘not’, Visual Basic uses ‘<’’, Prolog uses ‘\+’ and MATLAB uses ‘~’. It is best to look at the documentation for your language of choice to check the appropriate syntax.

## Conditional operators

**Conditional operators** are used to compare two values or expressions. They are fundamental in control structures such as selections and iterations, as they help in decision-making processes by evaluating conditions. The most common conditional operators are shown in Table 2.3.

**TABLE 2.3** Conditional operators

| Operator | Description              | Example | Result if a=10, b=5 |
|----------|--------------------------|---------|---------------------|
| ==       | Equal to                 | a == b  | false               |
| !=       | Not equal to             | a != b  | true                |
| >        | Greater than             | a > b   | true                |
| <        | Less than                | a < b   | false               |
| >=       | Greater than or equal to | a >= b  | true                |
| <=       | Less than or equal to    | a <= b  | false               |

## Control structures

There are three fundamental control structures in programming: sequences, selections and iterations.

### Sequence

A **sequence** is a set of instructions that executes line by line, a little bit like a recipe. Every line of code in the sequence is run in the order that it is written.

There are also more complex logical operators that you are unlikely to come across in VCE Software Development, which include XOR, XNOR, NAND and NOR. These are out of scope of the study design.

Note that there are different symbols used to represent operators in code compared to how you would represent them in pseudocode.

**FIGURE 2.5**

Pseudocode example of a sequence of instructions

```

ALGORITHM askName()
BEGIN
    PRINT "What is your name?"
    INPUT name
    PRINT "Hello, " + name + ". Nice to meet you."
END

```

In the algorithm shown in Figure 2.5, a sequence of code runs that asks a user for their name, reads the name as input and then greets the user by name. Each line of code in the algorithm is run, in order, only once.

## Selection

A **selection statement** is a control structure that allows a programmer to write lines of code that are only run when a particular requirement is met. The code within a selection statement can contain instructions, sequences, other selection statements or iterations.

Selections are **Boolean**, in that they are run based on the result of a condition being evaluated as either true or false. If the condition evaluates as 'true', the code within that selection statement is executed. If it evaluates as 'false', it is not executed.

The simplest type of selection statement is one that tests against a single condition, as seen in Figure 2.6.

**FIGURE 2.6**

A single condition selection control structure

```

ALGORITHM printPositive()
BEGIN
    INPUT firstNumber

    IF firstNumber > 0 THEN
        PRINT "The number is positive."
    ENDIF
END

```

## Alternative execution

Another form of a selection statement involves an **alternative execution**. This means that if the condition is not met, alternative code will run. For example, in Figure 2.7, a user is asked if they like pie. If they respond with 'yes' they receive a happy comment. If the user does not input 'yes', a sad comment will be printed instead. Note that the user does not need to input 'no' for the sad comment to be printed – any input other than 'yes' will execute the alternative code.

As you can see when comparing Figure 2.6 and Figure 2.7, the use of 'ELSE' in a selection statement is optional.

**FIGURE 2.7**

A selection control structure with alternative execution

```

ALGORITHM likePie()
BEGIN
    PRINT "Do you like pie?"
    INPUT likePie

    IF likePie = "yes" THEN
        PRINT "Hooray!"
    ELSE
        PRINT "That makes me sad. :( "
    ENDIF
END

```

## Selections with more than one logical expression

A selection statement is not limited to testing only one logical expression. Theoretically, the number of logical expressions a single selection statement can test is unlimited. Figure 2.8 contains pseudocode that uses the logical operator 'AND' within a single selection statement to check for two conditions to be simultaneously true.

```

ALGORITHM likePieCake()
BEGIN
    PRINT "Do you like pie?"
    INPUT likePie
    PRINT "Do you like cake?"
    INPUT likeCake

    IF likePie = "yes" AND likeCake = "yes" THEN
        PRINT "Hooray!"
    ELSE
        PRINT "That makes me sad. :( "
    ENDIF
END

```

**FIGURE 2.8** A simple selection with more than one logical expression

When writing selections with more than one logical expression, it can be useful to construct a **truth table** to make sure that no logic errors have been made. Truth tables use Boolean algebra to test each combination of values in a condition. For example, Table 2.4 contains a truth table to check the logical expressions in the pseudocode from Figure 2.8. Because the logical operator connecting the two conditions was 'AND', both conditions need to be true for the selection statement as a whole to evaluate as true, which is shown when only a single case in the truth table evaluates the whole condition as true.

**TABLE 2.4** Truth table to evaluate 'AND'

| likePie | likeCake | likePie AND likeCake |
|---------|----------|----------------------|
| True    | True     | True                 |
| True    | False    | False                |
| False   | True     | False                |
| False   | False    | False                |

If the logical operator connecting the two conditions was 'OR', the resulting truth table can be seen in Table 2.5. In this instance, the use of 'OR' expands the number of cases where the condition would evaluate to true to three, with the only time it evaluates to false being when both conditions are false.

**TABLE 2.5** Truth table to evaluate 'OR'

| likePie | likeCake | likePie OR likeCake |
|---------|----------|---------------------|
| True    | True     | True                |
| True    | False    | True                |
| False   | True     | True                |
| False   | False    | False               |

Truth tables are a systematic method of testing the logic of a selection statement. They are particularly useful when there are more than two conditions within a single statement. For example, Table 2.6 contains a truth table for the selection statement in Figure 2.9. The truth table shows that there are three cases where the entire selection statement would evaluate to true, and five cases where it would evaluate to false. This would not be easily apparent without constructing the truth table.

```

ALGORITHM multiConditions ()
BEGIN
    INPUT a, b, c, d
    IF a < b AND ( b < c OR c < d ) THEN
        PRINT "Condition met."
    ELSE
        PRINT "Condition not met."
    ENDIF
END

```

**FIGURE 2.9** A complex conditional with more than one logical expression

**TABLE 2.6** Truth table for Figure 2.9

| a < b | b < c | c < d | b < c OR c < d | a < b AND (b < c OR c < d) |
|-------|-------|-------|----------------|----------------------------|
| True  | True  | True  | True           | True                       |
| True  | True  | False | True           | True                       |
| True  | False | True  | True           | True                       |
| True  | False | False | False          | False                      |
| False | True  | True  | True           | False                      |
| False | True  | False | True           | False                      |
| False | False | True  | True           | False                      |
| False | False | False | False          | False                      |

A truth table can be used in conjunction with a testing table to determine if an algorithm is without logical errors.

### Chained selection

The algorithm shown in Figure 2.10 uses a more complex set of selection control structures (IF/ELSEIF) in order to react to user input when a condition needs to be tested more than once. This is referred to as a **chained selection**. In this example, the user can select four operations: addition, subtraction, multiplication and division. The program must therefore test the user input four times to see if it matches against the four conditions given.

As the algorithm uses ‘ELSEIF’, it will check each condition only if the condition prior to it evaluates as false. Without the use of ‘ELSEIF’, each condition would be run in sequence regardless of whether the condition before it evaluated as true or false. This is an important characteristic of the selection control structure that is often forgotten by programmers, resulting in logical errors in code.

Consider the pseudocode in Figure 2.11. Assume the current temperature is input as 9 degrees. The first IF condition tests to see if the temperature is less than 10 degrees – as 9 degrees is less than 10 degrees, the algorithm will print ‘It is very cold’ and then continue in sequence to the next line of code in the program, which is the second IF condition. The second IF condition will check if the temperature is less than 20 degrees: as 9 degrees is less

than 20 degrees, it will print ‘It is a little cool’ and then continue in sequence to the next line of code in the program, which is the ELSE condition. This portion of the code is not run: because the IF statement it is attached to evaluated to true, the ELSE condition has not been met (9 degrees is not greater than 20 degrees).

There is no limit to the number of conditions that can be contained in a chained selection statement. Chained selections also do not need to contain an ELSE statement.

```

ALGORITHM computeOperation()
BEGIN
    PRINT "What is the first number?"
    INPUT firstNumber
    PRINT "What is the second number?"
    INPUT secondNumber
    PRINT "What operation would you like to perform?"
    INPUT operationChosen
    Total ← 0
    IF operationChosen = "add" THEN
        Total ← firstNumber + secondNumber
    ELSEIF operationChosen = "subtract" THEN
        Total ← firstNumber - secondNumber
    ELSEIF operationChosen = "multiply" THEN
        Total ← firstNumber * secondNumber
    ELSEIF operationChosen = "divide" THEN
        Total ← firstNumber / secondNumber
    ELSE
        PRINT "Invalid operation chosen."
    ENDIF

    PRINT "The result is: " + Total
END

```

**FIGURE 2.10**

Pseudocode example of a chained selection control structure

```

ALGORITHM ifElseExample()
BEGIN
    PRINT "What is the current temperature?"
    INPUT currentTemp

    IF currentTemp < 10 THEN
        PRINT "It is very cold."
    ENDIF
    IF currentTemp < 20 THEN
        PRINT "It is a little cool."
    ELSE
        PRINT "It is very warm."
    ENDIF
END

```

**FIGURE 2.11**

Pseudocode example of conditions with logical errors

## Nested selection

Selection statements can also be placed inside each other. This type of selection is referred to as a **nested selection**. This control structure is useful when multiple conditions must be handled within the code. For example, Figure 2.12 contains pseudocode where a selection statement is used to check if it is raining. If it is, another check is made to see if the person has an umbrella. If they do, they use the umbrella, otherwise they get wet. The check for ‘hasUmbrella’ is nested within the check for ‘isRaining’.

It is important to use consistent formatting in your source code. This not only makes the code easier to read but also reduces the chance of logical errors occurring due to nested selections. The pseudocode in Figure 2.12 shows the hierarchy of the selection statements because of the use of indentation, such as tabs and spacing; it is much easier to read than if no indentations were used.

### SCHOOL-ASSESSED TASK TRACKER

Design brief  Project plan  Analysis  Folio of alternative design ideas  Development  Evaluation and assessment  Final submission

```

ALGORITHM checkUmbrellaUsage()
BEGIN
    INPUT isRaining
    INPUT hasUmbrella

    IF isRaining = True THEN
        IF hasUmbrella = True THEN
            useUmbrella()
        ELSE
            getWet()
        ENDIF
    ENDIF
END

```

FIGURE 2.12

Pseudocode example of a nested selection statement

### Switch/case

A **switch/case** statement, also referred to as a switch statement, is very similar to a chained selection, in that it allows for multiple conditions to be tested. Figure 2.13 includes an example of the use of switch/case.

Not all programming languages have switch/case functionality implemented. In the programming languages that do have it implemented, it is typically more efficient to use switch/case than it is to use chained selections.

```

ALGORITHM computeOperation()
BEGIN
    PRINT "What is the first number?"
    INPUT firstNumber
    PRINT "What is the second number?"
    INPUT secondNumber
    PRINT "What operation would you like to perform?"
    INPUT operationChosen

    Total ← 0
    SWITCH operationChosen
        CASE "add"
            Total ← firstNumber + secondNumber
        CASE "subtract"
            Total ← firstNumber - secondNumber
        CASE "multiply"
            Total ← firstNumber * secondNumber
        CASE "divide"
            Total ← firstNumber / secondNumber
        DEFAULT
            PRINT "Invalid operation selected."
    ENDSWITCH

    PRINT "The result is: " + Total
END

```

FIGURE 2.13

Pseudocode example of a switch/case control structure

Most programming languages include a method for exiting an iteration early. These are often referred to as 'breaks'.

### Iteration

An iteration, also known as repetition, is used to repeat sections of code multiple times until a condition is met. There are four main types of iterations: WHILE, DO/WHILE, FOR and REPEAT/UNTIL.

## WHILE

A **WHILE** iteration is a section of code that is run when, and for as long as, a condition is met. These types of iterations are useful when the programmer does not know when the condition might be met, such as when running sections of code based on user input that will only cease when a user inputs a particular key sequence.

An example of a WHILE iteration can be seen in Figure 2.14. This type of WHILE iteration is very common, because when opening a file for reading, it is not possible to tell how many lines there are in the file until all those lines are actually read.

```

ALGORITHM readFromFile()
BEGIN
    INPUT fileName

    fileObject ← open filename for reading
    WHILE end of file is not reached DO
        nextLine ← read one line from fileObject
        PRINT nextLine
    ENDWHILE
END

```

**FIGURE 2.14** An example of a WHILE iteration to read from a file

Key elements of WHILE iterations:

- They are used when it is unknown how many times the iteration will execute.
- The condition being tested within the WHILE iteration must be met at least once for the code within it to be executed; this is referred to as a pre-test iteration.
- If the condition being tested within the WHILE iteration is always true, the iteration will never terminate; this creates an **infinite loop**.

## DO/WHILE

A **DO/WHILE** iteration is similar to a WHILE iteration in that it executes code within the iteration for as long as a condition is met.

An example of pseudocode to read the contents of a file using a DO/WHILE iteration is shown in Figure 2.15.

```

ALGORITHM readFromFile()
BEGIN
    INPUT fileName

    fileObject ← open filename for reading
    DO
        nextLine ← read one line from fileObject
        PRINT nextLine
    WHILE end of file is not reached
END

```

**FIGURE 2.15** An example of a DO/WHILE iteration to read from a file

The difference between a WHILE and a DO/WHILE is that a WHILE iteration may not run if the condition being tested is never true, whereas a DO/WHILE iteration always runs at least once; this is referred to as a post-test iteration. For example, in the pseudocode shown in Figure 2.15, it is assumed that there is at least one line to read in the file that is opened. If this is not the case and this code were to be implemented in a programming language, it would produce a runtime error.

Key elements of DO/WHILE iterations:

- They are used when it is unknown how many times the iteration will execute.
- The code within the DO/WHILE iteration will always execute at least once.
- If the condition being tested within the DO/WHILE iteration is always true, the iteration will never terminate; this creates an infinite loop.

## FOR

A **FOR** iteration is a section of code that is run a predefined number of times. These types of iterations are particularly useful to perform an action on every element of an array, or to perform a basic search through a set of elements in a data structure.

FOR iterations need three pieces of information to execute. The first is a starting point, the second is the end condition, and the third is a statement called an increment that increases the starting point so that it approaches the end condition.

Most programming languages have a special format for writing FOR iterations. For example, in C and C++, a FOR is written as shown in Figure 2.16, and in Visual Basic it is written as shown in Figure 2.17.

**FIGURE 2.16** A FOR iteration in the style of the C, C++ and C# programming languages

```
for ( starting_point; end_condition; increment )
{
    statement(s);
}
```

**FIGURE 2.17** A FOR iteration in the style of the Visual Basic programming language

```
For counter = start_condition to end_condition
    Statement(s)
Next
```

An example of a FOR iteration in pseudocode is shown in Figure 2.18. This FOR checks every element of an array to see if a word being searched for is found in the array. Each time the iteration is executed, the end condition ( $i < iNumNames$ ) is tested; the iteration will continue to run for as long as this returns true.

**FIGURE 2.18** A FOR iteration checking every element of an array

```
ALGORITHM checkArray()
BEGIN
    INPUT arrayNames
    INPUT searchTerm

    iNumNames ← length of arrayNames
    FOR i ← 0, i < iNumNames, i ← i + 1 DO
        IF arrayNames[i] = searchTerm THEN
            PRINT "Found " + searchTerm
        ENDIF
    NEXT
END
```

Key elements of FOR iterations:

- The iteration runs for a set number of times and this must be known beforehand.
- The iteration will only execute the code inside it if the end condition is still being met; it may not execute the code at all.
- Unlike WHILE and DO/WHILE iterations, it is very rare for a FOR iteration to not terminate; this occurs only if the increment and end condition are unrelated. This type of occurrence would be considered a logic error.

## REPEAT/UNTIL

Much like WHILE iterations, **REPEAT/UNTIL** iterations repeatedly run a source code within the iteration; however, they differ in the treatment of the condition that terminates the iteration. A WHILE iteration will run for as long as a condition returns true, whereas a REPEAT/UNTIL iteration will run for as long as a condition returns false. Figure 2.19 demonstrates the pseudocode that uses a REPEAT/UNTIL iteration to read lines from a file.

```

ALGORITHM readFromFile()
BEGIN
    INPUT fileName
    fileObject ← open filename for reading
    REPEAT
        nextLine ← read one line from fileObject
    UNTIL end of file is reached
END

```

**FIGURE 2.19** An example of a REPEAT/UNTIL iteration to read from a file

Key elements of REPEAT/UNTIL iterations:

- They are used when it is unknown how many times the iteration will execute.
- The code within the REPEAT/UNTIL iteration will always execute at least once; it is a post-test iteration.
- If the condition being tested within the REPEAT/UNTIL iteration is always false, the iteration will never terminate; this creates an infinite iteration.

## Functions

A **function** is a sequence of instructions that performs a specific task that has been given a name by a programmer. The code within a function executes an algorithm and typically provides a **return value** as a result. To use a function within source code is to ‘call’ it. An example of a **function call** can be seen in Figure 2.20; both `useUmbrella()` and `getWet()` are function calls.

Notice that functions in pseudocode use parentheses after the name of the function. This helps distinguish them from variables.

**FIGURE 2.20** An example of a function call

```

ALGORITHM checkUmbrellaUsage()
BEGIN
    INPUT isRaining
    INPUT hasUmbrella

    IF isRaining = True THEN
        IF hasUmbrella = True THEN
            RETURN useUmbrella()
        ELSE
            RETURN getWet()
        ENDIF
    ENDIF
    RETURN False
END

```

A function can, but does not need to, provide a return value back to where the function was called.

**TABLE 2.7** Function declarations in some popular programming languages

|            |  |
|------------|--|
| C, C++, C# | int max(numOne, numTwo);                                     |
| VB, VB.Net | Function max(byVal numOne as Single, byVal numTwo as Single) |
| Python     | def max(numOne, numTwo):                                     |
| PHP        | function max(\$numOne, \$numTwo);                            |
| Swift      | func max(numOne: Int, numTwo: Int) -> Int                    |

Many programming languages require functions to be declared before they can be defined. Similarly, functions must be defined before they can be used within the source code. For this reason it is a good idea to place all function declarations at the top of your source code, and function definitions near the top as well.

It is best to use a built-in function whenever possible, as they have been tested and are less likely to contain bugs than a user-defined function.

Functions require a **function declaration** that names the function and its **arguments**. In many languages, function declarations must include the return value data type. They can also include an optional reference to the function visibility.

Functions must then have a **function definition** written, which simply means that the function must be written. Some languages, such as C and C++, require that the function declaration is written separately, prior to the function definition, whereas other languages, such as Python and VB.Net, include the function declaration as part of the function definition. Once a function is defined and written it can be used throughout the source code.

### Built-in library functions

Built-in library functions are functions that are created as part of the language. These **built-in functions** are there to make programming easier. For instance, most languages have a date function, which would return the computer's current date.

Make sure you check the handbook or online programming advice websites for information about what you specifically want a function to do before you create your own function. There is usually a long list of optimised built-in functions that have been fully tested.

| Function           | Description  |
|--------------------|--|
| DATE function      | Returns the serial number of a particular date   |
| DATEDIF function   | Calculates the number of days, months, or years between two dates. This function is useful in formulas where you need to calculate an age. |
| DATEVALUE function | Converts a date in the form of text to a serial number   |
| DAY function       | Converts a serial number to a day of the month   |

Source: Microsoft Corporation

**FIGURE 2.21** An example of Excel built-in functions

| Function                 | Description   |
|--------------------------|---|
| <code>abs()</code>       | Returns the absolute value of a number  |
| <code>all()</code>       | Returns True if all items in an iterable object are true                                      |
| <code>any()</code>       | Returns True if any item in an iterable object is true  |
| <code>ascii()</code>     | Returns a readable version of an object. Replaces none-ascii characters with escape character |
| <code>bin()</code>       | Returns the binary version of a number  |
| <code>bool()</code>      | Returns the boolean value of the specified object   |
| <code>bytearray()</code> | Returns an array of bytes   |
| <code>bytes()</code>     | Returns a bytes object  |

Source: Python Software Foundation

**FIGURE 2.22** Python built-in functions listed at [www.W3schools.com](http://www.W3schools.com)

## Arguments/parameters

Functions can optionally include variables in their definition. These are known as arguments, or **parameters**, and they act as specific inputs that are ‘passed’ to the function when the function is called. The data within the arguments passed to a function is assigned a temporary variable name as part of the function declaration. This allows the use of local variables within a function, avoiding the need to use global variables to access data that exists outside of the function.

Many programming languages have two categories of arguments that can be passed to a function: those that are **pass by reference** and those that are **pass by value**. Pass by reference means that the original data being passed into the function can be modified without needing to be ‘returned’. Pass by value means that the original data is left unchanged, even if the data in the temporary variable is modified within the function. For languages that only use pass by value in functions, in order to modify the data stored in the original variables that have been passed to the function, the modified data must be returned back to the source code that called the function.

Figure 2.23 demonstrates how to declare a function using pseudocode. Note that the data types of the arguments and the function’s return value are defined within a pseudocode comment. The arguments are included as part of the function definition to distinguish them from other types of input.

```

FUNCTION max(numOne, numTwo)
{ Purpose: return max value as an integer }
{ Arguments: numOne and numTwo are integers }
{ Output: integer, the maximum integer }
BEGIN
    IF numOne > numTwo THEN
        RETURN numOne
    ELSE
        RETURN numTwo
END

```

FIGURE 2.23

Representation of a function in pseudocode

## Object-oriented programming

**Object-oriented programming (OOP)** is a style of programming that revolves around the use of 'objects'. These objects are structures created from **classes** that hold data and code: the data is stored as fields (also referred to as attributes or properties), and the code is organised in procedures, commonly called methods. The key principles of OOP are **abstraction**, **encapsulation**, **generalisation** and **inheritance**.

### Classes

Imagine a developer is writing a game where players play golf.

#### THINK ABOUT SOFTWARE DEVELOPMENT

2.1

In a gaming context, consider the speed of processing required to open and close files every time a player gains a stat or skill, or has damage dealt to them.

How many people played *Fortnite* concurrently at the peak of its popularity? Imagine that many read/write operations – this would be an absurd amount of load on a system.

Describe other contexts where read/write operations would be an important factor to consider when writing software.

#### THINK ABOUT SOFTWARE DEVELOPMENT

2.2

Think about gaming in general. When would it be useful to keep statistics (variables) about something?



FIGURE 2.24 An example of a golf game

In the game, the golf clubs that a player uses influence the outcome of the game. Players can upgrade golf clubs as they gain experience in playing courses. Each club carries information (variables) unique to that club, such as the quality of the club (its condition), the material of the grip and the material of the club itself.

A golf club also has actions associated with it, such as the action of swinging the golf club, or the action of breaking the golf club because a player is frustrated. For those actions, a programmer would need to know the quality, grip material and club material of each and any golf club used by a player.

What happens if each time a player uses the golf club, the quality decreases, the grip on the handle wears away a little, a new scratch is formed on the club and the player gets a little more frustrated?

Writing a function in the source code of the game would not easily help a programmer keep track of all those things for each golf club being used, particularly if there are many players and many golf clubs. The source code would need to have a different variable for each club a player has, as well as a different variable for each player who plays the game. This becomes quite tedious to maintain within code, because to avoid **hard-coding** variables containing players and their clubs, the code would need to have used multiple associative arrays that were synchronised with each other so that each index value matched a particular player.

Using a data structure called a class allows a programmer to solve the ‘golf club’ problem. A class is a programmer-defined data structure that exists in object-oriented programming languages. Classes group conceptually similar functions and variables together in one place and work as templates for creating **objects**, which are **instantiations**, or instances, of classes that exist in memory on the computer where the program is run. A useful analogy to describe a class is to think of it as a blueprint or architectural design, such as one used when building a house. It describes everything that needs to be built to make a house, but is not a house itself. The house that is built using the design is an instance of the house blueprint. Classes work in the same way. They describe all of the elements and components that are required by the object created from the class blueprint, and this object is referred to as an instance of the class. The number of objects that can be created from a class is limited only by the amount of memory (**RAM**) in the computer system on which the program is running.

The purpose of a class is to create a template for objects with predetermined variables and behaviour. These templates can then be instantiated as objects or be used by another class in order to extend upon or change their behaviour. This allows for code reuse in programs where objects are similar to each other. In the golf club example, a programmer could extend on a base golf club class using inheritance to make woods, irons, wedges and putters.

## Methods

A **method** is a function that exists within a class. A special type of method is an **event**, which is a method that is called when an object’s state changes; this means that something has occurred to trigger the event. For example, pressing a button in a user interface object can trigger an event to submit that button.

```

ALGORITHM useMethod()
BEGIN
    mathObj = create new object from math class
    mathObj.addTwoNumbers(4, 1)
END

```

Before object-oriented programming introduced classes and objects, there was no straightforward way to write the golf clubs in the golf game with a programming language that just used functions, subroutines and variables. It would only be possible if the program saved information constantly to files and then read them back for each golf club and each player. Reading and writing to files is very slow and the game created could not be played by more than a couple of players at a time.

Events can be user-defined or they can be built-in events.

**FIGURE 2.25**  
Pseudocode of an object and a method

SCHOOL-ASSESSED TASK TRACKER

## Abstraction

Abstraction in OOP is a way of managing complexity by reducing and simplifying interactions with objects. It involves focusing on the essential qualities of an object in relation to the perspective of the viewer, rather than concentrating on its implementation details. This is achieved by creating classes that represent abstract concepts. These classes define the essential attributes and behaviours relevant to that concept while hiding the unnecessary details from the user.

For example, in a class representing a smartphone, abstraction allows a user to know that they can call, message or take photos without needing to understand the underlying technology, such as signal processing or sensor operation. Abstraction is about creating a simple model that represents complex entities in a manageable way.



### CASE STUDY

#### Abstraction with a MusicPlayer class

Consider the example of a *MusicPlayer* class to illustrate abstraction. Abstraction for the *MusicPlayer* class means exposing functionalities that are relevant to the user, such as playing a song, pausing or skipping tracks, while concealing the details of how those functionalities are implemented.

**Example:** MusicPlayer class

##### Essential features (exposed to the user):

- **play(song):** Plays the selected song. The user doesn't need to know how the music player accesses the song file, decodes the audio format or handles the audio output to the speakers.
- **pause():** Pauses the currently playing song. The internal state management, such as how the current position in the track is stored, is hidden.
- **skip():** Skips to the next song. The complexity of selecting the next track in the playlist or queue is abstracted away.
- **volumeUp()** and **volumeDown():** Adjusts the volume. The underlying audio processing and hardware interaction are not exposed.

##### Complex details (hidden from the user):

- Audio decoding algorithms.
- File I/O operations for reading audio files.
- Error handling for unsupported file formats or corrupted data.
- Managing audio hardware interfaces and drivers.

The *MusicPlayer* class abstracts the complexity of playing music. Users of the class interact with a clean and simple interface (methods such as play and pause) and are not burdened with the underlying complexities of audio processing, file management and hardware interaction. This abstraction makes the *MusicPlayer* class easy to use and integrate into other systems, such as a mobile app or a web application, without requiring deep knowledge of audio technology.

## Encapsulation

Encapsulation is about bundling the data and the methods that manipulate that data into a single unit, known as a class, and controlling their access in a safe way. It is a protective barrier that keeps the data safe from outside interference and misuse.

Data encapsulation led to the important OOP concept of data hiding. For instance, a class **BankAccount** may have a private attribute `accountBalance` and public methods

**deposit()** and **withdraw()** to manipulate this balance. The internal state of the `BankAccount` is hidden from the outside, only allowing interaction through its public methods, ensuring that the balance cannot be changed arbitrarily and maintaining integrity of the data.

One of the key benefits of abstraction is the ability to have some control over if and when data is changed; encapsulation is therefore a useful method of strengthening the security of software. Access modifiers, studied in Unit 4, Outcome 1 (Chapter 5, page 205), will allow you to explore how to add security measures into classes.

## Encapsulation with a Thermostat class

Consider the example of a `Thermostat` class to illustrate encapsulation.

**Example:** Thermostat Class

### Attributes (Data):

- `currentTemperature`: A private variable that stores the current room temperature.
- `desiredTemperature`: A private variable that stores the temperature set by the user.

### Methods (Functions):

- `increaseTemperature()`: A public method to increase the desired temperature. It internally adjusts the `desiredTemperature` attribute.
- `decreaseTemperature()`: A public method to decrease the desired temperature. Similar to `increaseTemperature`, it modifies the `desiredTemperature` attribute.
- `getCurrentTemperature()`: A public method that allows the user to view the current room temperature. It returns the value of `currentTemperature` without allowing direct modification.
- `updateInternalState()`: A private method that checks and adjusts the actual heating or cooling device to reach the desired temperature. This method is not accessible from outside the class.

In the `Thermostat` class, the actual temperature values (`currentTemperature` and `desiredTemperature`) are encapsulated within the class. They are marked as private, meaning they cannot be accessed directly from outside the class. Instead, they are modified through public methods like `increaseTemperature()` and `decreaseTemperature()`. This set-up ensures that the internal state of the thermostat can only be modified in controlled ways, preventing misuse or inconsistent states (like setting an impossibly high temperature).

The `updateInternalState()` method is a crucial part of the thermostat's functionality, but it is hidden from the user. This method might contain complex algorithms to control the heating or cooling system efficiently and safely, but this complexity is encapsulated within the class.

CASE  
STUDY



## Generalisation

Generalisation is the process of finding common characteristics among a set of classes and abstracting them into a parent class, known as a **superclass**. This superclass captures the shared attributes and behaviours of its subclasses. It is a bottom-up approach where you identify similarities in your subclasses and pull them up into a common superclass.

For example, if you have several classes, such as `Rectangle`, `Circle` and `Triangle`, you might find that they share common properties, such as colour and position, as well as methods like `move()` or `draw()`. These can be generalised into a superclass named `Shape`.

One of the key benefits of generalisation in OOP is that it allows for code reuse, as the abstracted superclasses contain shared elements that do not need to be re-coded for related classes.

SCHOOL-ASSESSED TASK TRACKER

- Design brief
- Project plan
- Analysis
- Folio of alternative design ideas
- Development
- Evaluation and assessment
- Final submission

## Identifying commonalities

The process of generalisation begins by examining various classes to pinpoint common attributes and behaviours. For example, consider classes like *Dog*, *Cat* and *Bird*. On close inspection, you might notice that all these classes share certain characteristics, such as age and weight, and behaviours (methods) like `eat()` and `sleep()`. This step is crucial as it lays the foundation for creating a generalised class.

## Creating a superclass

After identifying these commonalities, the next step is to abstract them into a new class, the superclass. This superclass encapsulates the shared characteristics and behaviours. In our example, this could lead to the creation of a superclass named *Animal*, which includes the shared attributes and methods such as age, weight, `eat()` and `sleep()`.

## Inheritance

As a result, the original classes (*Dog*, *Cat*, *Bird*) become subclasses of this newly created superclass. They inherit the common properties and behaviours from *Animal*. This inheritance means that the shared code is written just once in the superclass and is automatically available to all of its subclasses. This mechanism significantly reduces code duplication.

## Specialisation of subclasses

While the subclasses inherit common features from the superclass, they can also possess their own unique attributes and methods. For instance, the *Dog* class might have a specific method `bark()`, which is unique to dogs and not a general characteristic of all animals.

## Benefits of generalisation

There are many benefits to using generalisation when designing OOP software, including the following.

- Code reusability: Generalisation greatly reduces redundancy by extracting common code into a superclass. This shared code is written once and inherited by all subclasses, promoting code reusability.
- Ease of maintenance: When changes are required in the common features, they need to be made only once in the superclass. These changes then propagate automatically to all the subclasses, simplifying maintenance and updates.
- Scalability: Adding new subclasses becomes more straightforward with generalisation. For example, in the example discussed previously, introducing a new class like *Horse* would primarily involve defining its unique features, such as adding attributes such as mane and hoofs and methods such as `trot()` and `canter()`, as the common ones are already handled by the *Animal* superclass.

## Inheritance

A fundamental concept in OOP is inheritance, which is the ability for a new class to inherit properties and methods from an existing class. When generalisation occurs, for example, the subclasses *inherit* the superclass. This concept forms the backbone of code reusability and hierarchical classification in OOP.

Through inheritance, subclasses can utilise the attributes and methods of the superclass, significantly reducing code duplication and potential errors. It establishes a hierarchical

structure in class organisation, mirroring real-world relationships between objects. Inheritance comes in various forms, including single inheritance (where a subclass inherits from one superclass), multiple inheritance (inheriting from more than one superclass), multilevel inheritance (involving a chain of inheritance) and hierarchical inheritance (where multiple classes inherit from a single superclass).

Inheritance also allows subclasses to redefine or customise certain methods originally defined in the superclass. Overall, inheritance is instrumental in creating structured, efficient and realistic software models, promoting code reuse and reflecting real-world entity relationships.

## Inheritance with an Animal class

Consider the *Animal* superclass example used in the Generalisation section.

### Base class: Animal

The *Animal* class serves as the base or superclass. It includes properties and methods common to all animals.

Properties: name, age

Methods:

- eat(): Method for eating behaviour
- sleep(): Method for sleeping behaviour

### Subclass: Dog

The *Dog* class is a subclass of *Animal*. It inherits properties and methods from *Animal* and also introduces its specific characteristics.

Inherited properties: name, age (from *Animal*)

Inherited methods: eat(), sleep() (from *Animal*)

New properties: breed

New methods:

- bark(): A method specific to dogs

### Subclass: Cat

Similarly, the *Cat* class is another subclass of *Animal*, inheriting common features and adding its own unique traits.

Inherited properties: name, age (from *Animal*)

Inherited methods: eat(), sleep() (from *Animal*)

New properties: furColour

New methods:

- purr(): A method specific to cats.

### How inheritance works in this example

Both the *Dog* and *Cat* classes inherit the name and age properties from the *Animal* class. This means a programmer does not need to redefine these properties in the *Dog* and *Cat* classes.

The methods eat() and sleep() are common animal behaviours and are inherited by both *Dog* and *Cat*. This allows instances of *Dog* and *Cat* to use these methods without redefining them.

Each subclass also has its unique properties and methods. For instance, *Dog* has a breed property and a bark() method, while *Cat* has a furColour property and a purr() method. These are specific to each animal type and are not shared with the *Animal* superclass or other subclasses.

CASE  
STUDY



SCHOOL-ASSESSED TASK TRACKER

Design brief
  Project plan
  Analysis
  Folio of alternative design ideas
  Development
  Evaluation and assessment
  Final submission

# Graphical user interfaces

**Graphical user interfaces (GUIs)** and programming languages are closely intertwined in the development of modern software applications. A GUI is a type of user interface that allows users to interact with electronic devices through graphical icons and visual indicators, as opposed to text-based interfaces, typed command labels or text navigation. The relationship between GUIs and programming languages can be understood in several key aspects, outlined below.

## Development of GUIs

Most programming languages provide libraries or frameworks that allow developers to create GUIs. These libraries offer a range of pre-built graphical elements, such as windows, buttons, text fields and menus, that can be used to construct the user interface of an application.

Different programming languages offer various levels of support for GUI development. For instance, Java has Swing and JavaFX, C# integrates with Windows Presentation Foundation (WPF) and Universal Windows Platform (UWP), Python is known for Tkinter and PyQt, and JavaScript is used for web-based interfaces with HTML and CSS.

## Event-driven programming

GUIs are typically built on an event-driven programming model. This means that the flow of the program is determined by events such as user actions (clicks, typing, etc.), sensor outputs or message passing.

Programming languages that are used to build GUIs often have features to support event handling. This includes the ability to detect events and trigger specific functions or methods in response.

## Integration with back-end code

In software design, it is common to separate the GUI (front end) from the business logic or data access layer (back end). Programming languages play a crucial role in linking these layers. User interactions captured by the GUI are processed by the back-end code, which performs the necessary operations and updates the GUI accordingly.

## Cross-platform GUI development

Some programming languages and frameworks are designed for cross-platform GUI development. They allow developers to write the code once and deploy the application across multiple platforms (Windows, macOS, Linux, web, mobile). Examples include Java (with JavaFX), Python (with Kivy) and JavaScript/TypeScript (with React Native or Electron).

## Accessibility and usability

Programming languages enable the customisation of GUIs to make them more accessible and user-friendly. This includes adjusting colours, fonts and layout, as well as adding features such as screen readers and keyboard navigation.

# Algorithms for sorting

Many applications require some method of sorting data so it can be used within a program or by a user of a program. It may seem trivial for a human to place a set of items into sorted order, but to achieve the same result in a computer program requires the use of control structures and repetition of sequences of steps. It is also important that the combination of control structures and sequences is efficient, particularly when a lot of data needs to be sorted.

While there are many different algorithms that can be used to sort data, Software Development students must know about two specific types of sort: selection sort and quick sort.

## Selection sort

A simple method of sorting a set of elements into a sorted list is to use selection sort. **Selection sort** involves searching through a whole list, selecting the smallest element it finds, and swapping that element to the front of the list. The searching and swapping continues until the entire list is sorted.

The selection sort algorithm follows these steps:

- Assume the first element is the smallest element.
- Compare the first element to every other element in the array, one by one.
- Each time the element compared to the first element is smaller, swap the first element with the smaller element.
- Compare the (possibly new) first element with the rest of the array.
- Repeat the whole process, starting with the second element in the array.
- Repeat the whole process, starting with the third element in the array.
- Continue repeating the whole process until all elements in the array have been checked.

This algorithm is called selection sort because it repeatedly selects the next-smallest element and swaps it into place.

For example, consider the following list of unsorted numbers:

|    |   |    |   |
|----|---|----|---|
| 12 | 8 | 31 | 1 |
|----|---|----|---|

Assume the first element is the smallest element, in this case, 12. Keep track of the index value of this element in a variable, such as one called 'smallest'. For this example, index values start at 1, but in almost all programming languages they begin at 0.

|          |   |
|----------|---|
| smallest | 1 |
|----------|---|

Compare 12 with 8. 8 is smaller, so update the variable 'smallest' with the index value for the number 8.

|          |   |
|----------|---|
| smallest | 2 |
|----------|---|

The number 8 is then compared to 31. As 31 is not smaller than 8, the 'smallest' variable is unchanged.

As sorting algorithms are not language specific, almost all programming languages have built-in functions that implement standard sorting algorithms for programmers to use.

### THINK ABOUT SOFTWARE DEVELOPMENT

2.3

Take a shuffled deck of cards, and physically sort them. Consider how you chose to sort them. Did you separate suits into four stacks and then sort by face value? Or did you sort by face value and then by suit? What steps did you repeat to finally succeed in having a sorted deck of cards?

#### SCHOOL-ASSESSED TASK TRACKER

Design brief
  Project plan
  Analysis
  Folio of alternative design ideas
  Development
  Evaluation and assessment
  Final submission

The number 8 is then compared to 1. 1 is smaller, so update the variable 'smallest' with the index value for the number 1.

|          |   |
|----------|---|
| smallest | 4 |
|----------|---|

The end of the array has been reached, so the element in the first position of the array is now swapped with the element at the index position stored in the variable 'smallest'.

|   |   |    |    |
|---|---|----|----|
| 1 | 8 | 31 | 12 |
|---|---|----|----|

The sort then moves to the next element of the array and uses the index value of this element as the new value for 'smallest'.

|          |   |
|----------|---|
| smallest | 2 |
|----------|---|

The number 8 is the new smallest element, in position two of the array. It is compared with 31, which is larger, so no change is made to the variable 'smallest'. It is then compared to 12, which is also larger, so no change is made to the variable 'smallest'.

|   |   |    |    |
|---|---|----|----|
| 1 | 8 | 31 | 12 |
|---|---|----|----|

As the end of the array has been reached, the element referred to by the variable 'smallest' should be swapped with the 8, but as they are the same index values, no swap needs to happen.

|   |   |    |    |
|---|---|----|----|
| 1 | 8 | 31 | 12 |
|---|---|----|----|

The sort then moves to the next element of the array (the third element) and uses the index value of this element as the new value for 'smallest'.

|          |   |
|----------|---|
| smallest | 3 |
|----------|---|

The number 31 is the new smallest element, in position three of the array. It is compared with 12, which is smaller, so the variable 'smallest' is updated with the index value of the 12.

|          |   |
|----------|---|
| smallest | 4 |
|----------|---|

As the end of the array has been reached, the element referred to by the variable 'smallest' is swapped with the 31.

|   |   |    |    |
|---|---|----|----|
| 1 | 8 | 12 | 31 |
|---|---|----|----|

The sort then moves to the next element of the array (the fourth element). However, this is the last element of the array so there is nothing left to compare it to. The selection sort algorithm is therefore complete and all elements in the array are now sorted.

Figure 2.26 shows the pseudocode for selection sort.

```

ALGORITHM selectionSort(arrElements)
{ Purpose: sorts a list of elements }
{ Input: an array of elements }
{ Index values start at 1 }
{ Output: Array, a sorted array of elements }
BEGIN
  n ← number of items in arrElements
  FOR i ← 1 to n DO
    { select the smallest item }
    smallest ← i
    { compare smallest to the rest of the array }
    FOR j ← i + 1 to n DO
      IF arrElements[j] < arrElements[smallest] THEN
        { update the index value of smallest }
        smallest ← j
      ENDIF
    NEXT
    { the smallest item in the array has been found }
    { so swap it with the current element }
    IF smallest != i THEN
      swap arrElements[smallest] AND arrElements[i]
    ENDIF
  NEXT
  RETURN arrElements
END

```

**FIGURE 2.26**  
Pseudocode for selection sort

When converting pseudocode to real code, some of the pseudocode elements may need to be expanded upon. For example, in most programming languages, swapping two elements generally cannot be achieved using a single line of code. Consider the different outputs of Figures 2.27 and 2.28. Only Figure 2.28 will successfully swap the two values, as the x value is lost in Figure 2.27.

```

x ← 20
y ← 10

x ← y
y ← x
PRINT x, y

```

**FIGURE 2.27** Incorrectly swapping two values

```

x ← 20
y ← 10

temp ← x
x ← y
y ← temp
PRINT x, y

```

**FIGURE 2.28** Correctly swapping two values

**THINK ABOUT**  
SOFTWARE  
DEVELOPMENT

2.4

Convert the pseudocode in Figures 2.27 and 2.28 into the language of your choice. Run both versions to see their outputs.

SCHOOL-ASSESSED TASK TRACKER

Design brief  Project plan  Analysis  Folio of alternative design ideas  Development  Evaluation and assessment  Final submission

```

# Selection Sort
# Required Arguments:
# - a list of elements of any data type to be sorted
# Return Value:
# - the sorted list

def selectionSort( aList ):
    for i in range( len( aList ) ):
        smallest = i

        for k in range( i + 1 , len( aList ) ):
            if aList[k] < aList[smallest]:
                smallest = k
        if smallest != i:
            temp = aList[smallest]
            aList[smallest] = aList[i]
            aList[i] = temp
    return aList

```

**FIGURE 2.29** An example of selection sort implemented using Python

## Quick sort

Another algorithm used to sort sets of elements is **quick sort**, which is referred to as a **divide and conquer** algorithm. This is because quick sort is an example of a recursive sort, which means that it partitions the items that need to be sorted into smaller and smaller sets and passes those sets back into itself; thus, it ‘divides’ the array into smaller and smaller pieces until it can ‘conquer’ the array and sort it. This allows items to be sorted very quickly. In almost all cases, quick sort is more efficient than selection sort.

The quick sort algorithm, using an in-place sort, follows these steps:

- Unless the array contains only a single element, complete the following steps:
  - » Select an element from the array at random – this is referred to as the pivot. Often, this ‘random’ element is the last element in the array.
  - » Check each other element in the array and reorder it so that all elements with values less than the pivot come before it, while all elements with values greater than the pivot come after it (equal values can go either way). This involves swapping, much like in selection sort, but the elements are not sorted based on anything else except the pivot when they are swapped.
  - » Take all of the elements that are less than the pivot (all the elements to the left of the pivot) and repeat the process of quick sort on these elements, selecting a new pivot.
  - » Take all of the elements that are greater than the pivot (all of the elements to the right of the pivot) and repeat the process of quick sort on these elements, selecting a new pivot.
- If the array contains only a single element, return just that element.

For example, consider the following list of unsorted numbers:

|    |   |    |   |    |    |    |
|----|---|----|---|----|----|----|
| 12 | 8 | 31 | 1 | 77 | 75 | 18 |
|----|---|----|---|----|----|----|

Select the first element as the pivot and store its index value in a variable, ‘pivot’. Two more variables need to be made, one storing the first index value, and the other storing the last index value. At the beginning of the quick sort algorithm, there should therefore be three variables: for this example, they are called pivot, low and high. Assuming index values begin at 1, the index value of pivot is 7, low is 1 and high is 7.

Begin iterating through the list, comparing each value to the value stored at the pivot. The first element checked is 12, which is smaller than the pivot's value of 18. This means the first element should be swapped with the element at the 'low' index value. In this instance, the 'low' index value is the same index as the first element, so no swap occurs. After this, the 'low' index value is incremented by 1, and now holds the value of 2.

|    |   |    |   |    |    |    |
|----|---|----|---|----|----|----|
| 12 | 8 | 31 | 1 | 77 | 75 | 18 |
|----|---|----|---|----|----|----|

The next element checked is 8, which is smaller than 18. This means that this element must be swapped with the element at the 'low' index value. In this instance, the 'low' index value of 2 is the same as the second element, so no swap occurs. The index value of 'low' is then incremented by 1, and now holds the value of 3.

|    |   |    |   |    |    |    |
|----|---|----|---|----|----|----|
| 12 | 8 | 31 | 1 | 77 | 75 | 18 |
|----|---|----|---|----|----|----|

The next element checked is 31, which is larger than 18. This element is left alone.

|    |   |    |   |    |    |    |
|----|---|----|---|----|----|----|
| 12 | 8 | 31 | 1 | 77 | 75 | 18 |
|----|---|----|---|----|----|----|

The next element checked is 1, which is smaller than 18. This means that this element must be swapped with the element at the 'low' index value; 1 and 31 are therefore swapped. The index value of 'low' is then incremented by 1, and now holds the value of 4.

|    |   |   |    |    |    |    |
|----|---|---|----|----|----|----|
| 12 | 8 | 1 | 31 | 77 | 75 | 18 |
|----|---|---|----|----|----|----|

The next element checked is 77, which is larger than 18. This element is left alone. The same occurs with 75.

Once the algorithm reaches the last element, 18, it is swapped with the element at the index value of 'low'. In this case, this is the element at the fourth index value, 31.

|    |   |   |    |    |    |    |
|----|---|---|----|----|----|----|
| 12 | 8 | 1 | 18 | 77 | 75 | 31 |
|----|---|---|----|----|----|----|

The list is now partitioned so that every number less than the pivot (18) is to the left of it, and every number that is greater is to the right of it. The algorithm is run again on each side.

Left side:

|    |   |   |
|----|---|---|
| 12 | 8 | 1 |
|----|---|---|

Pivot: 3 (value: 1)

Low: 1 (value: 12)

High: 3 (value: 1)

After the first pass, no swaps are made as 12 is greater than 1:

|    |   |   |
|----|---|---|
| 12 | 8 | 1 |
|----|---|---|

Pivot: 3 (value: 1)

Low: 1 (value: 12)

High: 3 (value: 1)

After the second pass, no swaps are made as 8 is greater than 1. There are no more passes to make, so the 'low' and 'high' index values are swapped:

|   |   |    |
|---|---|----|
| 1 | 8 | 12 |
|---|---|----|

The list is now partitioned so that every number less than the pivot (1) is to the left of it, and every number that is greater is to the right of it. The algorithm is run again on each side, even though one of the sides (the left side) is empty. These steps are not shown here, as this particular sub-list is already in sorted order.

Right side:

|    |    |    |
|----|----|----|
| 77 | 75 | 31 |
|----|----|----|

Pivot: 3 (value: 31)

Low: 1 (value: 77)

High: 3 (value: 31)

After the first pass, no swaps are made as 77 is greater than 31:

|    |    |    |
|----|----|----|
| 77 | 75 | 31 |
|----|----|----|

Pivot: 3 (value: 31)

Low: 1 (value: 77)

High: 3 (value: 31)

After the second pass, no swaps are made as 75 is greater than 31. There are no more passes to make, so the 'low' and 'high' index values are swapped:

|    |    |    |
|----|----|----|
| 31 | 75 | 77 |
|----|----|----|

The list is now partitioned so that every number less than the pivot (31) is to the left of it, and every number that is greater is to the right of it. The algorithm is run again on each side, even though one side (the left side) is empty. These steps are not shown here, as this particular sub-list is already in sorted order.

Once all sub-lists have been processed through the quick sort algorithm, the array that is left is in sorted order:

|   |   |    |    |    |    |    |
|---|---|----|----|----|----|----|
| 1 | 8 | 12 | 18 | 31 | 75 | 77 |
|---|---|----|----|----|----|----|

Figure 2.30 shows the pseudocode for quick sort.

While quick sort is algorithmically complex for a human brain to understand, **recursive algorithms** are extremely fast for a computer to process.

```

ALGORITHM quickSort(arrElements, low, high)
{ Purpose: sorts a list of elements }
{ Inputs: an array of elements, two integers representing }
{ the first last element in the array }
{ Index values start at 1 }
BEGIN
    IF low < high THEN
        { run the partition algorithm to know where }
        { to split the array }
        split ← partition(arrElements, low, high)

        { run quicksort on the left side }
        quickSort(arrElements, low, split-1)

        { run quicksort on the right side }
        quickSort(arrElements, split+1, high)
    ENDIF
END

ALGORITHM partition(arrElements, low, high)
{ Purpose: to split an array into two based on a pivot, }
{ where the left side contains values less than }
{ the pivot and the right side contains value }
{ greater than the pivot }
{ Inputs: an array of elements, two integers representing }
{ the first last element in the array }
{ Index values start at 1 }
{ Output: integer, index value of the partition point }
BEGIN
    pivot ← arrElements[high]

    FOR i ← low + 1 to high DO
        IF arrElements[i] < pivot THEN
            IF low ≠ i THEN
                swap arrElements[low] and arrElements[i]
            ENDIF
            low = low + 1
        ENDIF
    NEXT
    swap arrElements[low] and arrElements[high]
    RETURN low
END

```

**FIGURE 2.30**  
Pseudocode for quick sort  
(in-place)

## Algorithms for searching

As with sorting, many applications require some method of searching through data to find a particular item within a set of items. It is important that these searches are efficient, particularly in the modern era of ‘big data’.

While there are many different algorithms that can be used to search data, Software Development students must know about two specific types of searches: linear search and binary search.

## Linear search

A **linear search** is the simplest type of search. This search involves checking every element in the list, from first to last, when searching for a particular element.

For example, consider the following list of unsorted numbers, where the number being searched for is 77:

|    |   |    |   |    |    |    |
|----|---|----|---|----|----|----|
| 12 | 8 | 31 | 1 | 77 | 75 | 18 |
|----|---|----|---|----|----|----|

A linear search is typically performed using a FOR iteration if the size of the list is known. The first step of the FOR iteration will check the first index value, which is 12. 12 is compared with the search input, 77, and this returns false. As a result, the FOR will iterate again, checking the next index value in the list, 8. This repeats until it reaches the 77, where the linear search returns true, as it has found the value. Linear searches will execute until the end of the list is reached or the search item is found.

Figure 2.31 shows the pseudocode for a linear search. Linear searches are useful when the elements are not sorted and the number of elements to search through is small.

```

ALGORITHM linearSearch( )
{ Purpose: searches through a list of elements }
{ Output: Boolean, True if item found, False if not }
BEGIN
    INPUT searchList, searchItem
    found ← FALSE
    FOR eachItem in the searchList DO
        IF eachItem = searchItem THEN
            found ← TRUE
            BREAK {exit loop once found}
        ENDIF
    NEXT
    RETURN found
END

```

**FIGURE 2.31**  
Pseudocode for a linear search

While linear searches are the simplest to implement in almost every programming language, they are also extremely inefficient. This is not a concern if the number of elements being checked is relatively small, but it becomes problematic as the number of elements increases. Consider a search of six billion records where the record being searched for is not there. This is referred to as a ‘worst-case’ scenario, as all six billion records would need to be checked to confirm that the record was not in the set.

## Binary search

A **binary search** is more efficient than a linear search. Binary search is similar to quick sort in that it is also a recursive algorithm, but instead of being divide and conquer, it is a **decrease and conquer** algorithm, as it is able to discard half of the data being searched through at each iteration of the algorithm. This makes it an extremely efficient method of searching.

Binary search relies on the data it is searching through being sorted. It works by selecting an element from the very middle of the data set being searched and checking it against the search item. If it matches, the search halts. If it doesn’t match, it will search the data to the left of the element in the middle if it is less than that element, otherwise it will search the data to the right of it. This process is repeated until the item is found or there are no more elements to search through.

For example, consider the following list of sorted numbers, where the number being searched for is 77:

|   |   |    |    |    |    |    |
|---|---|----|----|----|----|----|
| 1 | 8 | 12 | 18 | 31 | 75 | 77 |
|---|---|----|----|----|----|----|

A binary search will begin by checking the length of the list. If it is greater than one, it will then find the index value of the middle of the list by dividing the length of the list by 2. As the list has seven elements, dividing by 2 produces a fraction, which must be rounded up or down. It doesn't matter which way it is rounded, as long as the rounding is consistent. For this example, the binary search will round up; therefore, the index value that is checked first is at index 4. The number 18 is compared to the number being searched for and does not match. The binary search will then check to see if 18 is greater than or less than 77. As it is less than 77, the search will discard every element to the left of the 18, inclusive. The rest of the list is passed back into the binary search to be searched again.

|    |    |    |
|----|----|----|
| 31 | 75 | 77 |
|----|----|----|

There are three elements in the list, so the mid-point is calculated again; this time it is index value 2. The number 75 is compared to 77 and does not match; it is also less than 77 so it is discarded, as is every element to the left of it. The rest of the list is passed back into the binary search to be searched again.

|    |
|----|
| 77 |
|----|

There is only one item left in the list, so no mid-point needs to be calculated. The item is compared to the search item and it matches, so the search item is found.

Figure 2.32 shows the pseudocode for a binary search. Binary searches are useful when there are large amounts of elements to search through, but can only be used if those elements are sorted. This means that you need to consider the time it takes to *sort the list* as well as *search the list* when considering the efficiency of binary search.

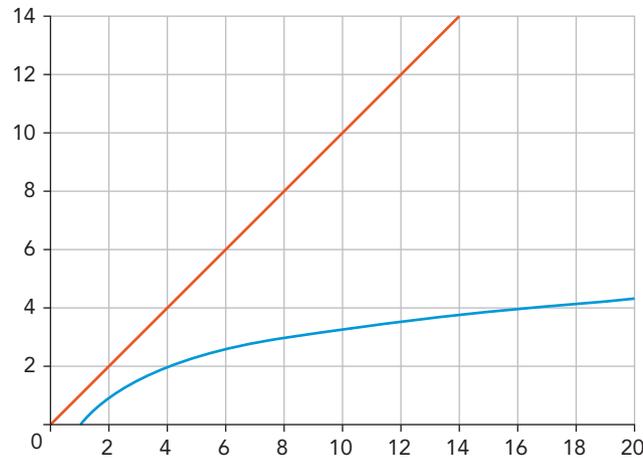
```

ALGORITHM binarySearch(arrayList, searchItem)
{ Purpose: searches through a list of elements }
{ Inputs: an array of elements to be searched }
{           and the item being searched for }
{ Output: Boolean, True if item found, False if not }
BEGIN
    found ← FALSE
    iLen ← the length of arrayList
    midP ← the middle index value of arrayList
    IF searchItem = arrayList[midP] THEN
        found ← TRUE
    ELSEIF iLen > 1 THEN
        IF searchItem < arrayList[midP] THEN
            low ← first index value of arrayList
            RETURN binarySearch(arrayList[low to midP],
                                searchItem)
        ELSEIF searchItem > arrayList[midP] THEN
            high ← iLen
            RETURN binarySearch(arrayList[midP to high],
                                searchItem)
        ENDIF
    ENDIF
    RETURN found
END

```

**FIGURE 2.32**  
Pseudocode for a binary search

That being said, the efficiency of a binary search versus a linear search is not a small comparison. Consider the search discussed earlier, where there are six billion records and the record being searched for is not there. Recall that in a worst-case scenario, if a linear search was used, all six billion records would need to be checked to confirm that the record was not there. If a binary search was used, only 33 items at most would need to be checked to come to the same conclusion. The difference in speed of a linear search and binary search in a worst-case scenario can be seen in the graph shown in Figure 2.33.



**FIGURE 2.33**  
Worst-case linear search (red) vs. worst-case binary search (blue)

## Efficiency of algorithms

When more than one algorithm could achieve a particular purpose, it is important to consider the efficiency of each algorithm when you are considering the efficiency of a whole solution. Some algorithms perform better in certain contexts than others, and some are simply better than others overall.

The mathematical notation used to describe the efficiency of algorithms is referred to as ‘Big O notation’, and while it is not explicitly in the Software Development course, it is useful to know the Big O notation for algorithms in order to make an informed decision on which one to use. Big O notation is expressed using a capital O and then enclosing in brackets the maximum amount of time it would take for an algorithm to finish, expressed in terms of the number of items being processed by the algorithm. For example, if a sorting algorithm used a single FOR iteration to sort all of the elements,  $n$ , in a list, it would be expressed as running in  $O(n)$  time. If a nested FOR iteration were used to sort the same list, where one FOR runs inside of another FOR it would be expressed as running in  $O(n^2)$  time. Divide and conquer algorithms generally involve logarithmic time.

When measuring algorithm efficiency, there are three scenarios that are considered: **best case**, **average case** and **worst case**. The best-case scenario for many sorting algorithms, for example, is for the set of elements to already be sorted. The worst-case scenario for many of those same sorting algorithms would be for the elements to be in reverse-sorted order. The average-case scenario for a sorting algorithm would be when the elements are not in any particular order.

Generally, programmers consider the average case of an algorithm when deciding which one to use, and then consider the worst-case scenario if the average cases are equivalent. They may also consider the worst-case scenario to make sure that it is not considerably slower than an alternative algorithm.

For example, on average, a linear search will check approximately half  $\left(\frac{n}{2}\right)$  the number of items before it finds the item being searched for. A binary search will check approximately  $\log_2 n$  items, which is more efficient. In a worst-case scenario where an item is not present, a linear search must check  $n$  items, whereas a binary search divides the list in two each time it iterates through it, which means checking  $\log_2 n$  items. This means that a binary search is faster than a linear search in both average-case and worst-case scenarios.

Only average-case and worst-case scenarios are typically considered because it is often not helpful to consider the best-case scenario of an algorithm. Data is rarely in a format that would allow for a best-case scenario to occur, so best-case scenarios are very rare. For example, in a best-case scenario linear search, the item being searched for would be the first item in the list. This is not something that would occur reliably enough to matter when deciding which algorithm to use.

For the two sorting algorithms and two searching algorithms used in Software Development, the average case and worst case are shown in Table 2.8.

**TABLE 2.8** Average case and worst case of Software Development algorithms

| Average case   |                             |
|----------------|-----------------------------|
| Selection sort | $O(n^2)$                    |
| Quick sort     | $O(n \log_2 n)$             |
| Linear search  | $O\left(\frac{n}{2}\right)$ |
| Binary search  | $O(\log_2 n)$               |
| Worst case     |                             |
| Selection sort | $O(n^2)$                    |
| Quick sort     | $O(n^2)$                    |
| Linear search  | $O(n)$                      |
| Binary search  | $O(\log_2 n)$               |

As you can see, in a worst-case scenario, quick sort combined with binary search is no worse than selection sort combined with binary search. However, once the average case is considered, the most efficient method of sorting and searching is to combine a quick sort with a binary search.

## Validation techniques

Validation is the process of checking that input data is reasonable. Validation does not and cannot check that inputs are accurate.

### Existence checking

An **existence check** checks whether a value has been entered at all. This is particularly useful to ensure that all required fields in a form have been completed before saving the contents of those fields to a file.

## Type checking

A **type check** is a useful method of confirming that the values entered into fields are of the expected data type. It will confirm if the wrong type of data has been entered in fields, such as if strings are entered into fields that expect only integer values.

When implementing type checking, it is important to consider how inputs from a user are processed by the selected programming language. For example, in Python, unless explicitly handled otherwise, all inputs are treated as strings, even if they are numeric. In languages such as these, there typically exist methods for checking type, such as the process of **casting** the data as a particular data type to see if it remains valid.

## Range checking

A **range check** checks that data is within acceptable limits or comes from a range of acceptable values. For example, students enrolling in kindergarten must be between the ages of 3 and 6 years (acceptable limits). As another example, the product size must be small, medium or large (acceptable values).

# Checking that modules function correctly

When building an application, one of the most important steps is to check that each completed module meets the design specifications.

When checking to see if a module meets design specifications, it is important to make sure it performs as expected with appropriate inputs, that it is usable and efficient, and that it achieves what was specified in the design. This often involves internal testing, where programmers test the program themselves; external testing, where quality assurance testers test the program using test cases based on the design specifications; and client testing, where clients participate in walk-throughs and reviews of the software to confirm that it is what they specified in the design stage.

In VCE Software Development, modules are checked to ensure that they meet design specifications mostly through internal testing methods. An important aspect of this is to make sure that the modules that have been built are as bug-free as possible. This is achieved by completing appropriate testing activities, such as establishing test cases that determine test data and **expected results**, conducting tests, recording results and then correcting any errors.

One method used in the testing process is **debugging**, which is a testing method that includes finding errors through the construction of trace tables and testing the system using test data.

## Types of errors

Different types of errors can occur throughout the development of a software solution. Each error can be categorised as a syntax error, runtime error or logic error.

### Syntax errors

Each programming language has a defined syntax, which is a set of rules that defines which symbols and characters can be used to write source code. **Syntax errors** occur while writing code and are typically fixed immediately, as they prevent code from compiling. These are

errors such as missing brackets or semi-colons, not passing the correct number of arguments into a function, or not putting quotes around characters and strings. Most compilers and interpreters will indicate on which line a syntax error occurs, which allows a programmer to find the error easily.

## Runtime errors

Unlike a syntax error, a software module with a **runtime error** will compile without any error notifications. It is not until the program is run and used that a runtime error can appear. These types of errors often result in the program crashing or printing error messages. For example, one type of runtime error is a **memory leak**, where a program continually uses more and more RAM while the program is running, such as when an infinite loop occurs.

Runtime errors are often found during the development stage, but software can, and does, get released with the possibility of runtime errors still occurring. This is because these errors tend to occur only when certain conditions are met, or when unexpected inputs are entered into the software. Once a runtime error is found in a distributed piece of software, many software companies release **patches** and software updates to correct the error.

In Software Development, the runtime errors that are most salient include:

- overflow
- index out of range
- type mismatch
- divide by zero.

## Overflow

An **overflow error** occurs when a calculation produces a result that exceeds the storage capacity of the data type used to store it. This typically happens in fixed-size data structures like integers. For example, in a system where an integer is represented with 32 bits, the maximum value it can store is  $2^{31}-1$  (for signed integers). Any operation leading to a value beyond this range causes an overflow.

Consider a digital scoreboard at a sports event that can only display numbers up to 999. If the score somehow reaches 1000, the scoreboard can't display it correctly due to its limited digit capacity. This is similar to an overflow runtime error.

## Index out of range

This error occurs when a program attempts to access an element at an index that is outside the bounds of an array or collection. Since arrays are zero-indexed and have a fixed size, trying to access an element beyond this range (like attempting to access the sixth element in a five-element array) results in an **index out of range** error.

Imagine a bookshelf with five shelves labelled 1 to 5. Trying to place a book on the non-existent sixth shelf is an example of an index out of range situation.

## Type mismatch

A **type mismatch** error occurs when values or variables of incompatible data types are combined in an operation. Data types define the nature of the data (such as integers, floating-point numbers and text), and certain operations are not logically defined across different types.

Consider it similar to trying to listen to a painting or look at music as it is being played. Each medium – visual art and sound – requires different senses, just as different data types in programming require operations suitable to their nature.

**TABLE 2.9** Error messages thrown when trying to add strings to integers

| Programming language | Example type mismatch error   |
|----------------------|---|
| Python               | <code>TypeError: unsupported operand type(s) for +: 'int' and 'str'</code>  |
| Java                 | <code>Main.java:5: error: bad operand types for binary operator '+'\n first type: int\n second type: String</code>                                |
| C++                  | <code>error: invalid operands of types 'int' and 'std::__cxx11::string {aka std::__cxx11::basic_string&lt;char&gt;}' to binary 'operator+'</code> |
| C#                   | <code>Program.cs(10,24): error CS0019: Operator '+' cannot be applied to operands of type 'int' and 'string'</code>                               |
| Swift                | <code>error: binary operator '+' cannot be applied to operands of type 'Int' and 'String'</code>  |

### Divide by zero

A **divide by zero error** happens when a program attempts to divide a number by zero. In mathematics, division by zero is undefined because it does not result in a finite or meaningful value.

Consider a scenario where you need to distribute an amount of money equally among a group of people. If there are no people (zero), this task becomes impossible, as you cannot divide something into zero parts. This mirrors the divide by zero runtime errors in programming.

**TABLE 2.10** Error messages thrown when trying to divide by zero

| Programming language | Example divide by zero error  |
|----------------------|---|
| Python               | <code>ZeroDivisionError: division by zero</code>  |
| Java                 | <code>Exception in thread "main" java.lang.ArithmeticException: / by zero\n at Main.main(Main.java:5)</code>                          |
| C++                  | <code>Floating point exception (core dumped)</code>   |
| C#                   | <code>Unhandled exception. System.DivideByZeroException: Attempted to divide by zero.\n at Program.Main() in Program.cs:line 9</code> |
| Swift                | <code>Fatal error: Division by zero</code>  |

### Logic errors

**Logic errors**, also known as semantic errors, occur when the logic of a software program fails. This means that the source code is syntactically correct but the software solution does not produce the expected output when run. In this case, the output is often unintended, undesirable or incorrect. For example, a function written to return the square root of a number may instead return the square of a number. Similarly, a function in an air-conditioning unit that checks to see if the temperature in a room is greater than 30 degrees before turning itself on may activate the air-conditioning unit at 29 degrees instead.

Logic errors can be very difficult to find, as there is nothing within a compiler or interpreter that will tell a programmer on which line a logic error appears. It is often up to the programmer as a human to construct test data and trace tables in order to track down the error manually.

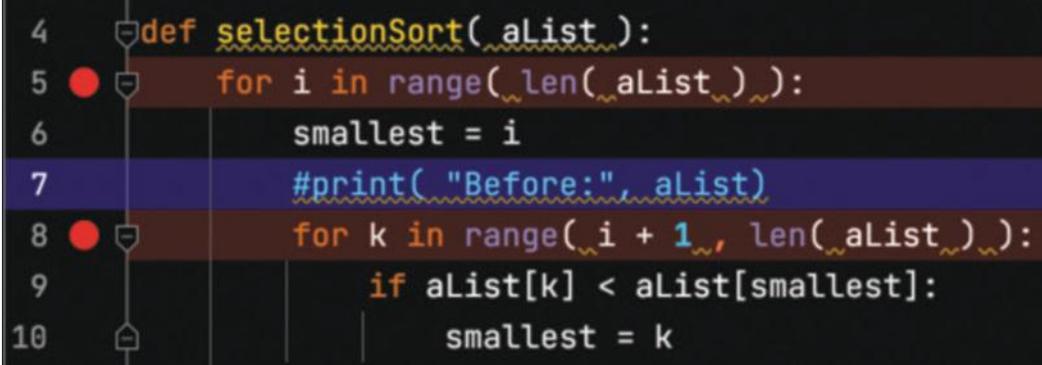
## Breakpoints

**Breakpoints** are markers set by a developer in a code editor or debugger that pauses the execution of the program during debugging. When the program execution reaches a line of code where a breakpoint is set, it stops, allowing the developer to examine the current state of the program.

In Figure 2.34, two breakpoints have been set – these are indicated in the IDE as red dots next to lines 5 and 8. When the code executes, the program will pause every time it runs those lines and display the contents of the current variables to the developer, as seen in Figure 2.35.

Breakpoints are an essential tool in the software development process, particularly in the debugging phase. They help developers identify and fix issues in their code by allowing them to pause execution and inspect the state of the program at specific points.

Source: JetBrains s.r.o.



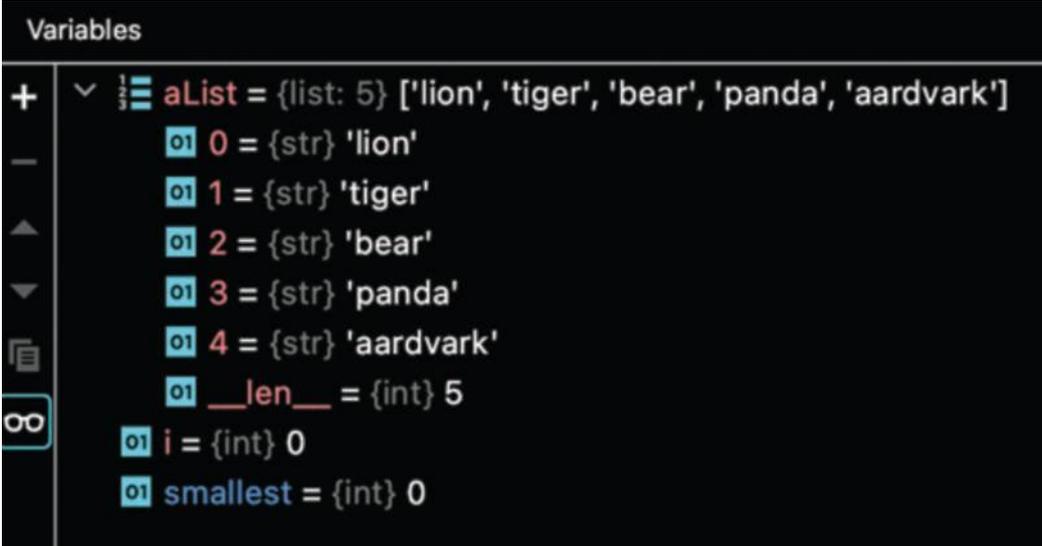
```

4  def selectionSort( aList ):
5  ●  for i in range( len( aList ) ):
6      smallest = i
7      #print( "Before:", aList)
8  ●  for k in range( i + 1, len( aList ) ):
9      if aList[k] < aList[smallest]:
10     smallest = k

```

**FIGURE 2.34** Breakpoints in Python code, using the PyCharm IDE

Source: JetBrains s.r.o.



```

Variables
+  aList = {list: 5} ['lion', 'tiger', 'bear', 'panda', 'aardvark']
-  01 0 = {str} 'lion'
-  01 1 = {str} 'tiger'
-  01 2 = {str} 'bear'
-  01 3 = {str} 'panda'
-  01 4 = {str} 'aardvark'
-  01 __len__ = {int} 5
-  01 i = {int} 0
-  01 smallest = {int} 0

```

**FIGURE 2.35** The output displayed when a breakpoint is reached when using the PyCharm IDE

## Debugging statements

**Debugging statements** are lines of code added to a program specifically for the purpose of debugging. These often include print statements (like `console.log` in JavaScript, `print` in Python or `System.out.println` in Java) that output the value of variables, the flow of execution, or specific messages to the console or log files.

Debugging statements provide insights into how data is being manipulated as the program runs. By observing the outputs of these statements, a developer can trace the execution flow and spot where things might be going wrong. These statements are easy to implement and, unlike breakpoints, do not require specialised IDE tools. They are useful for quick checks and simple debugging scenarios.

There are some disadvantages to using debugging statements. They can clutter the code if not removed after debugging and, unlike breakpoints, they also require a program to run in its entirety, which might not be efficient for large or complex applications.

## Test data and test cases

In order to systematically test that a module works, appropriate **test data** must be constructed to write a test case. A **test case** is a set of steps that a tester uses to determine if the element being tested works correctly. It involves selecting test data, writing testing procedures and determining expected results. It is particularly important that appropriate test data is selected so that test cases can be run. At a minimum, the selected test data must ensure full coverage of the algorithm when test cases are run. This means that all paths of all control structures are tested fully.

There are four main types of test data. The first involves validation test data that tests the validation techniques that have been included in the module. Data must therefore be selected to test any instances of existence checking, type checking and range checking that have been included in the source code. At a minimum, this should involve selecting test data that checks for both valid and invalid inputs. For example, if checking for a valid age integer being input by a user, a valid input would be a positive integer, likely in the range of 0 and 120. An invalid input would be a negative integer, such as  $-1$ .

The second type of test data is data that will test all selections, including the operators within those conditions. This may involve the construction of truth tables to help choose appropriate test data. It may also involve selecting appropriate boundary values to ensure full coverage of the operators included in the conditions.

The third type of test data involves data that will test all iterations, including any operators within those iterations. This may involve selecting or creating data that will ensure all conditions of those iterations are met. For example, if a WHILE iteration is designed to iterate over each line of a file, test data would involve testing the following:

- when the file does not exist
- when the file exists but is empty
- when the file exists and has one line to read
- when the file exists and has more than one line to read.

The fourth type of test data involves creating test cases that will attempt to cover all of the functionality within the system from the perspective of a user. This typically involves testing the GUI to ensure that each screen within the application can be accessed correctly. Typically, test cases are written based on design documents such as mock-ups, storyboards and/or site maps to ensure that the order and sequence of user interface elements are correct. This type of testing can be the most time-consuming, particularly if the application allows a user to take multiple paths to access particular screens. Developers often write formal test cases that allow for automated software testing programs to be used to run these tests so they do not need to be performed manually. These test cases can be written to simulate the actions of a user navigating through the software solution.

## Boundary values

Testing for **boundary values** involves selecting test data that will test the ‘boundaries’ of any selection or iteration within the code; that is, the maximum and minimum values available for any given input. Boundary values are particularly relevant for algorithms that use range checks. The general rule for boundary testing is that at each boundary, test data should be selected to test inside the boundary and outside the boundary.

For example, imagine a software solution designed for a Scout group that would only allow users to join if they are in the ‘Venturer’ age group (15–17 years). The boundary values for testing this can be determined by considering the type of data that is being collected. As this example involves testing for age, valid integers begin at 0 and are unlikely to go beyond 120. Testing for valid ages would therefore require testing an age that is below the age group, within the age group, and above the age group.

The ages that are in range can be represented on a continuum, as shown below.

|   |   |   |   |     |    |    |    |    |    |    |     |
|---|---|---|---|-----|----|----|----|----|----|----|-----|
| 0 | 1 | 2 | 3 | ... | 14 | 15 | 16 | 17 | 18 | 19 | ... |
|   |   |   |   |     |    | ↑  | ↑  | ↑  |    |    |     |

There are two boundaries for this particular test. One boundary exists at age 15 and the other exists at age 17. The pseudocode that demonstrates these can be seen in Figure 2.36.

```

ALGORITHM checkIsVenturer(age)
{ Purpose: to check that a user is a Venturer based on age }
{ Inputs: age, as an integer }
{ Output: Boolean, True if age is valid, False if not }
BEGIN
    IF age > 14 AND age < 18 THEN
        RETURN True
    ELSE
        RETURN False
    ENDIF
END

```

**FIGURE 2.36**

Pseudocode to check an age range

In order to fully test this algorithm, the test data that must be selected is:

| Age | Reason   | Expected result |
|-----|--|-----------------|
| 14  | One year below the lower boundary of the valid range | False           |
| 15  | The lower boundary of the valid range                | True            |
| 17  | The upper boundary of the valid range                | True            |
| 18  | One year above the upper boundary of the valid range | False           |

When selecting above and below a boundary, test data should be in the smallest increment possible given the context. In this case it is one year, but if an algorithm were to test for a price range, for example, the smallest increment would be 0.01. Similarly, if testing an algorithm that uses boundaries involving hours or minutes in a day, the smallest increment would typically be one minute; thus boundaries at the hour would be 59, 0 and 1.

The age range in Figure 2.36 contains a two-sided boundary, but when you are testing a one-sided boundary in VCE Software Development, the most suitable values to check are: one below the boundary, the boundary itself, and one above the boundary.

```

ALGORITHM abstractBoundaries ()
BEGIN
    INPUT a, b
    IF a < b THEN
        PRINT "Condition met."
    ELSE
        PRINT "Condition not met."
    ENDIF
END

```

**FIGURE 2.37** An abstract set of boundary conditions

For a more abstract example, the pseudocode in Figure 2.37 would require the following boundary values to be tested.

| a | b   | Reason   | Expected result   |
|---|-----|--|-------------------|
| a | a+1 | a is guaranteed to be less than b                                      | Condition met     |
| a | a   | a is guaranteed to not be less than b                                  | Condition not met |
| a | a-1 | a is guaranteed to not be less than b, as b will always be less than a | Condition not met |



## CASE STUDY

### Test data: Library study room booking system

A library offers study rooms that can be booked by patrons. To manage the bookings efficiently, the library has implemented a booking system that only allows reservations for the current day. The library is open from 9 a.m. to 5 p.m., and the study rooms can be reserved from 3 p.m. onwards. If someone attempts to reserve a study room before 3 p.m., they will have their booking rejected.

An appropriate testing table for this scenario would be:

| Time (24-hour format) | Test description                 | Expected output  |
|-----------------------|----------------------------------|------------------|
| 14:59                 | Time is one minute before 3 p.m. | Booking rejected |
| 15:00                 | Time is 3 p.m. exactly           | Booking accepted |
| 15:01                 | Time is one minute after 3 p.m.  | Booking accepted |

For more complex algorithms, more test data needs to be selected. This is when it can be very useful to construct a truth table to determine what that test data should be.

Consider the pseudocode example testing for multiple conditions in Figure 2.38.

```

ALGORITHM multiConditions ()
BEGIN
    INPUT a, b, c, d
    IF a < b AND (b < c OR c < d) THEN
        PRINT "Condition met."
    ELSE
        PRINT "Condition not met."
    ENDIF
END

```

**FIGURE 2.38** A complex selection with more than one logical expression

Consider also the truth table constructed for this algorithm, shown in Table 2.11.

**TABLE 2.11** Truth table for Figure 2.38

| $a < b$ | $b < c$ | $c < d$ | $b < c$ OR $c < d$ | $a < b$ AND ( $b < c$ OR $c < d$ ) |
|---------|---------|---------|--------------------|------------------------------------|
| True    | True    | True    | True               | True                               |
| True    | True    | False   | True               | True                               |
| True    | False   | True    | True               | True                               |
| True    | False   | False   | False              | False                              |
| False   | True    | True    | True               | False                              |
| False   | True    | False   | True               | False                              |
| False   | False   | True    | True               | False                              |
| False   | False   | False   | False              | False                              |

The test data required to fully test the conditions in Figure 2.38 must test the conditions shown on each line of the truth table. For example, the second line of the truth table requires that the condition  $c < d$  is not met, so a boundary test must be performed where  $c$  is not less than  $d$  (e.g.  $c = d$ ).

For this pseudocode, at least eight test data elements must be written to test the conditional statement fully.

It can be useful to use an algebraic expression to map out which values are needed for a test condition to be met. For example, using the example from Figure 2.38 and Table 2.11, the first line of the truth table can be interpreted algebraically in this way:

| $a < b$     | $b < c$     | $c < d$     | $b < c$ OR $c < d$ | $a < b$ AND ( $b < c$ OR $c < d$ ) |
|-------------|-------------|-------------|--------------------|------------------------------------|
| $a = b - 1$ | $b = c - 1$ | $c = d - 1$ | True               | True                               |

This is because each condition can be guaranteed to be true if the algebraic conditions are met.

The algebraic expressions can be further simplified:

| $a < b$     | $b < c$           | $c < d$     | $b < c$ OR $c < d$ | $a < b$ AND ( $b < c$ OR $c < d$ ) |
|-------------|-------------------|-------------|--------------------|------------------------------------|
| $a = b - 1$ | $b = (d - 1) - 1$ | $c = d - 1$ | True               | True                               |

And again:

| $a < b$                 | $b < c$           | $c < d$     | $b < c$ OR $c < d$ | $a < b$ AND ( $b < c$ OR $c < d$ ) |
|-------------------------|-------------------|-------------|--------------------|------------------------------------|
| $a = ((d - 1) - 1) - 1$ | $b = (d - 1) - 1$ | $c = d - 1$ | True               | True                               |

And finally:

| $a < b$     | $b < c$     | $c < d$     | $b < c$ OR $c < d$ | $a < b$ AND ( $b < c$ OR $c < d$ ) |
|-------------|-------------|-------------|--------------------|------------------------------------|
| $a = d - 3$ | $b = d - 2$ | $c = d - 1$ | True               | True                               |

Therefore, only a value for  $d$  needs to be chosen, as the values for  $a$ ,  $b$  and  $c$  can all be determined from this value.

It is important that test data is selected systematically rather than in an ad hoc manner, otherwise the source code cannot be guaranteed to be logically correct throughout all of its algorithms. This results in a considerably large set of test data to be used to test the system, which is why many software development companies use automated tools to construct and run tests on source code.

## Trace tables

To prevent logic errors occurring, programmers often participate in **desk checking** by constructing **trace tables** to validate the logic of the algorithms used in their source code. Trace tables simulate the execution of a program, referred to as the **flow of execution**. Given test data, each processing feature within an algorithm is executed, step by step, and, based on the test data, the values of the variables that change within that algorithm are tracked, by hand or by using breakpoints, to ensure that the logic within the algorithm is correct. This systematic method of tracking the execution of code allows for the thorough testing of even the most complex of algorithms.

As an example, consider the pseudocode in Figure 2.39. This algorithm uses a WHILE loop control structure to print values until a condition is met. The trace table to represent the flow of execution of this pseudocode is in Table 2.12.

```

ALGORITHM sampleWhileLoop ()
BEGIN
    x ← 0
    y ← 0
    WHILE x < 32 DO
        x ← x + 8
        y ← y + 4
    ENDWHILE
    PRINT x, y
END

```

**FIGURE 2.39**  
Pseudocode example of a  
WHILE loop

**TABLE 2.12** Trace table for the pseudocode in Figure 2.39

| Step | Statement                     | x  | y  | output |
|------|-------------------------------|----|----|--------|
| 1    | x ← 0                         | 0  |    |        |
| 2    | y ← 0                         | 0  | 0  |        |
| 3    | <b>While</b> x < 32 <b>Do</b> | 0  | 0  |        |
| 4    | x ← x + 8                     | 8  | 0  |        |
| 5    | y ← y + 4                     | 8  | 4  |        |
| 6    | <b>While</b> x < 32 <b>Do</b> | 8  | 4  |        |
| 7    | x ← x + 8                     | 16 | 4  |        |
| 8    | y ← y + 4                     | 16 | 8  |        |
| 9    | <b>While</b> x < 32 <b>Do</b> | 16 | 8  |        |
| 10   | x ← x + 8                     | 24 | 8  |        |
| 11   | y ← y + 4                     | 24 | 12 |        |
| 12   | <b>While</b> x < 32 <b>Do</b> | 24 | 12 |        |

**TABLE 2.12** Trace table for the pseudocode in Figure 2.39 (continued)

| Step | Statement                       | x  | y  | output |
|------|---------------------------------|----|----|--------|
| 13   | $x \leftarrow x + 8$            | 32 | 12 |        |
| 14   | $y \leftarrow y + 4$            | 32 | 16 |        |
| 15   | <b>While</b> $x < 32$ <b>Do</b> | 32 | 16 |        |
| 16   | <b>EndWhile</b>                 | 32 | 16 |        |
| 17   | <b>Print</b> x, y               | 32 | 16 | 32, 16 |

Another example, where a logic error exists within an algorithm, can be seen with the algorithm in Figure 2.40.

```

ALGORITHM applyDiscount ()
BEGIN
    INPUT fullPrice
    INPUT discPerc

    discPrice  $\leftarrow$  fullPrice - discPerc
    RETURN discPrice
END

```

**FIGURE 2.40** Pseudocode containing a logic error

As this algorithm requires input from a user, the trace table can only be constructed using test data. For the purposes of this example, the test data is as follows, with the flow of execution being tested twice, with two sets of inputs, and the expected discounted price being listed with the test data.

**TABLE 2.13** Test data for pseudocode in Figure 2.40

| fullPrice | discPerc | Expected result |
|-----------|----------|-----------------|
| 20.00     | 5        | 19.00           |
| 50.00     | 50       | 25.00           |

**TABLE 2.14** Trace tables for the pseudocode in Figure 2.40

| Step | Statement  | fullPrice | discPerc | discPrice | output |
|------|--|-----------|----------|-----------|--------|
| 1    | <b>Input</b> fullPrice   | 20        |          |           |        |
| 2    | <b>Input</b> discPerc  | 20        | 5        |           |        |
| 3    | $\text{discPrice} \leftarrow \text{fullPrice} - \text{discPerc}$ | 20        | 5        | 15.00     |        |
| 4    | <b>return</b> discPrice  | 20        | 5        | 15.00     | 15.00  |

| Step | Statement  | fullPrice | discPerc | discPrice | output |
|------|--|-----------|----------|-----------|--------|
| 1    | <b>Input</b> fullPrice   | 50        |          |           |        |
| 2    | <b>Input</b> discPerc  | 50        | 50       |           |        |
| 3    | $\text{discPrice} \leftarrow \text{fullPrice} - \text{discPerc}$ | 50        | 50       | 0.00      |        |
| 4    | <b>return</b> discPrice  | 50        | 50       | 0.00      | 0.00   |

After completing the trace tables, the test data can be completed to show the following results.

**TABLE 2.15** Completed test data for pseudocode in Figure 2.40

| fullPrice | discPerc | Expected result | Actual result |
|-----------|----------|-----------------|---------------|
| 20.00     | 5        | 19.00           | 15.00         |
| 50.00     | 50       | 25.00           | 0.00          |

The actual results from the trace table and the test data make it clear that the algorithm is not calculating a correct discounted price given the inputs it is receiving; it is merely subtracting the discount percentage as if it were a dollar value to be discounted. The corrected algorithm can be seen in Figure 2.41.

```

ALGORITHM applyDiscount ()
BEGIN
    INPUT fullPrice
    INPUT discPerc

    discAmount ← fullPrice * (discPerc / 100)
    discPrice ← fullPrice - discAmount
    RETURN discPrice
END

```

**FIGURE 2.41** Discount pseudocode after the logic error is fixed

Trace tables were traditionally produced manually by a programmer to test code and check for logic errors. Many programming languages today, however, have **integrated developer environments (IDEs)** where, in addition to the programmer using breakpoints or debugging statements, they can also trace the flow of execution through a debugger built into the IDE, so that they do not need to construct trace tables by hand. As covered earlier in this chapter, AI-driven tools can also be used to generate test cases as well as pinpoint problematic logic in code.

# 2

## CHAPTER SUMMARY

### Essential terms

**abstraction** an object-oriented programming language principle that allows programmers to manage complexity by hiding implementation details and exposing only the essential features of an object

**alternative execution** code that is run if a condition is not met

**arguments** specific inputs passed into a function that act as local, temporary variables

**arithmetic operator** a symbol in programming that performs basic mathematical operations such as addition, subtraction, multiplication and division

**automated debugging and testing of modules** using software tools and scripts to automatically identify, diagnose and fix errors

**average case** the time it takes to run an algorithm, on average

**best case** the best time it can take to run an algorithm

**binary search** a decrease and conquer algorithm that repeatedly halves a sorted search space until an element is found or not found

**Boolean** a data type that holds the values of true or false

**boundary values** the maximum and minimum edge values possible for a given input

**breakpoint** a debugging tool that allows the execution of a program to be paused at a specific point to allow a programmer to inspect the current state of the program and diagnose any issues

**built-in functions** functions that have been written by the creators of the programming language to execute common sequences of code

**casting** converting a variable from one data type to another, such as converting a string to an integer

**chained selection** a selection statement that handles more than one possible conditional outcome

**class** a program code template for creating objects in object-oriented programming languages

**code optimisation** the process of refining code to improve the efficiency and/or performance of a program without altering its functionality

**compiler** a program that turns source code into machine language that can be executed by a computer processor

**conditional operator** a programming concept that evaluates a condition and returns one of two values based on whether the condition is true or false

**constant** a fixed value that, once defined, cannot be altered during the execution of a program

**debugging** identifying and removing errors from computer software

**debugging statement** a line of code inserted into a program to output information about the program's execution

**decrease and conquer** to recursively reduce a problem to two or more smaller instances of the same problem until the problem can be solved

**desk checking** a manual process where a programmer reviews and traces through their code to verify its correctness and logic

**divide and conquer** to recursively break down a problem into two or more sub-problems of the same type until they are simple enough to solve on their own; the solved problems are then combined to provide a final solution

**divide by zero error** an error occurring when an arithmetic equation is attempting to divide by 0

**DO/WHILE** an iteration over a set of instructions, conditions and/or iterations that is repeated for as long as a condition is met; it is always run at least once

**encapsulation** an object-oriented programming principle that involves bundling the data and methods that operate on the data into a single unit or class

**event** a special type of method that is called when an object's state changes

**existence check** test to see if a value has been entered as input or not

**expected results** the output expected from an algorithm, assuming it is logically correct

**flow of execution** the order in which instructions, conditions and iterations are executed or evaluated

**FOR** an iteration over a set of instructions that is repeated a predefined number of times

**function** a sequence of related code that has been given a name that can be called from other points in the source code

**function call** to execute the contents of a function

**function declaration** to name a function and its arguments

**function definition** to define (write) the contents of a function

**generalisation** the process of defining a general class (superclass) that encapsulates common attributes and behaviours of more specific classes (subclasses)

**global variables** variables that are defined outside any function and can be accessed by all functions throughout the source code

**graphical user interface (GUI)** a type of user interface that allows users to interact through visual elements such as windows, icons and buttons

**hard-coding** to include fixed data in a program that cannot be changed during runtime and can only be changed by modifying the program source code

**index out of range** an error that occurs when attempting to access an element of an array using an index that is outside the valid range of indices for that array

**infinite loop** an iteration that will never reach the condition upon which it can terminate

**inheritance** a method of basing an object or class on another object or class, taking on its attributes and methods and potentially extending upon them

**instantiation** in object-oriented programming, the process by which an object is created from a class

**instruction** a unit of code that can be executed by a compiler or interpreter

**integrated developer environment (IDE)** software that provides tools to aid in programming, such as source code editing, syntax highlighting, code completion, debugging aids, or tools to help construct a user interface

**interpreter** a computer program that directly executes source code without needing to have it compiled beforehand

**linear search** a search that checks every element in a list, from first to last, when searching for a particular element

**local variables** variables that are defined inside a function that can only be accessed by that function

**logic error** when source code is syntactically correct but contains an error resulting in unintended, undesirable or incorrect output

**logical operator** a Boolean operator used to combine expressions, such as AND, OR

**memory leak** a failure of a program to release memory that is no longer needed, causing impaired performance, application failure and/or system failure

**method** an action an object can carry out (e.g. window.refresh, golfClub.swing)

**nested selection** when a selection contains one or more additional conditions within its structure

**object** any instantiated class that a program can inspect and/or change, in terms of appearance, behaviour or data

**object-oriented programming (OOP)** a programming language based on the concept of objects that contain data in the form of fields or attributes and code in the form of methods

**overflow error** an error that occurs when a calculation exceeds the maximum limit that a data type can represent

**parameters** see **arguments**

**pass by reference** to pass data into a function as an argument so that it can be modified without needing to be returned

**pass by value** to pass data into a function as an argument so that it cannot be modified without needing to be returned

**patches** sets of changes to a software application designed to update or fix it

**pointer** a variable that stores the memory position of another variable's value

**prompt** the input text or query given to an AI model to generate a response

**quick sort** a divide and conquer algorithm that sorts a set of data by recursively partitioning and sorting smaller and smaller sets of that data

**RAM** random access memory; a type of computer memory that can be accessed randomly; it is most often volatile memory that is lost if power is removed

**range check** tests to see if a value is within a given range of acceptable values

**recursive algorithm** an algorithm that calls itself with smaller or simpler sets of values until a solution can be found

**REPEAT/UNTIL** an iteration over a set of instructions that is repeated for as long as a condition is not met; it will always execute at least once

**responsible and ethical use of AI** developing and deploying AI in a manner that ensures fairness, accountability, transparency, privacy and respect for human rights

**return value** a value or set of values that is passed back to the origin of a calling function, often to be assigned to a variable, used in an equation, or tested within a conditional statement

**runtime error** an error that occurs while a program is running, including overflow, index out of range, type mismatch and divide by zero

**selection sort** the process of selecting and swapping elements within a list until the entire list is sorted

**selection statement** a control structure that allows a programmer to write lines of code that are only run when a particular requirement is met

**sequence** a set of instructions that executes line by line in the order that it is written

**switch/case** a conditional statement that handles more than one possible conditional outcome

**syntax error** often a typographical error in source code that violates the set of rules that define a programming language

**test case** a set of steps that a tester uses to determine if the element being tested works correctly, often outlining test data, testing procedures and expected results

**test data** data that has been specifically identified to be used in a test case

**trace table** a tool used in programming and algorithm analysis to track the values of variables at each step of the execution of a program or algorithm

**truth table** a table used to represent all of the combinations of values for inputs and their outputs, typically used to test conditional statements

**type check** tests to see if a value is of the specified data type or structure

**type mismatch** when a function or method receives an argument of an unexpected data type leading to errors or unintended behaviour

**WHILE** an iteration over a set of instructions that is repeated for as long as a condition is met

**worst case** the longest amount of time it can take to run an algorithm

## Important facts

- 1 Emerging trends in programming have increased the everyday use of prompts and generative AI tools for translating natural language inputs and translating them into functional code.
- 2 AI-driven tools can be used to create and use test cases on code.
- 3 AI-driven testing can cover a broader range of scenarios and edge cases that human testers might overlook.

- 4 AI trends have and will result in cultural shifts in the software development community.
- 5 There is a growing need for programmers to adapt and learn how to effectively collaborate with intelligent systems.
- 6 Variables are references to stored data so they can be used within a program.
- 7 Local variables are defined within a function or block of code and are only accessible within that specific context.
- 8 Global variables are declared outside any function and are accessible from any part of the program.
- 9 Constants are values that, once defined, cannot be altered during the execution of a program.
- 10 Instructions are units of code that can be executed by a compiler or interpreter.
- 11 Arithmetic operators include addition, subtraction, multiplication, division and modulus.
- 12 Logical operators include AND, OR and NOT.
- 13 Conditional operators include equal to, not equal to, greater than, less than, greater than or equal to, and less than or equal to.
- 14 Sequences are sets of instructions that execute line by line.
- 15 Selection statements are control structures that will execute only if a particular requirement is met.
- 16 Selections allow for alternative execution, more than one logical expression, chaining, nesting and switching.
- 17 Iterations are repetitive loops that repeatedly run sections of code until a condition is met or not met.
- 18 There are four types of iterations: WHILE, DO/WHILE, FOR and REPEAT/UNTIL.
- 19 Functions are sequences of related code that have been named by a programmer.
- 20 Functions contain arguments, also known as parameters, that act as local variables to the function.
- 21 Object-oriented programming revolves around the use of classes and objects.
- 22 Classes allow programmers to group related functions and variables together.
- 23 Classes can be instantiated to create objects.
- 24 Methods are functions that exist inside classes and objects.
- 25 Events are special types of methods that perform a sequence of code when an action occurs.
- 26 The key principles of object-oriented programming are abstraction, encapsulation, generalisation and inheritance.
- 27 Graphical user interfaces allow users to interact with electronic devices through graphical icons and visual indicators.
- 28 Two algorithms used to sort data are selection sort and quick sort.
- 29 Selection sort repeatedly selects the next smallest element from a set of elements and swaps it into its correct position until all elements are sorted.
- 30 Quick sort repeatedly partitions elements into smaller and smaller sets in order to produce a final sorted list.
- 31 Two algorithms used to search through data are linear search and binary search.
- 32 Linear search checks every element in a list, from first to last, when searching for a particular element.
- 33 Binary search repeatedly divides and discards half of the elements in a list while searching for an element until the element is found or it is determined to not be in the list.
- 34 It is important that algorithms are as efficient as possible.
- 35 Validation techniques include existence checks, type checks and range checks.
- 36 Debugging is the process by which software errors are found and fixed.
- 37 Syntax errors occur when the grammatical rules of a programming language have not been followed.
- 38 Runtime errors occur while the program is being used.
- 39 Logic errors occur when the expected output of an algorithm does not match the actual output, but no runtime errors occur.
- 40 Test data must be carefully selected in order to test all aspects of an algorithm.
- 41 Test cases should be created to test all boundary values.
- 42 Trace tables simulate the flow of execution of a program and allow a programmer to manually detect logic errors.



## Programming languages

- 1 In what ways can machine learning models enhance code optimisation, and what specific improvements might they suggest?
- 2 Why is maintaining a clean and efficient codebase important for long-term project sustainability, and how does AI-driven code optimisation support this goal?
- 3 What are the key ethical considerations developers must address when integrating AI tools into software development processes?
- 4 What are two methods programmers can use to ensure that AI-generated code is free from bias and aligns with ethical standards?

## Variables and instructions

- 5 Which of the following are most likely to be variables?
  - a `getUmbrella()`
  - b `isFound`
  - c `customer.firstName`
  - d `True`
  - e `2018`
  - f `"cake"`
  - g `apple`

## Control structures

- 6 Explain the differences between local and global variables in terms of their scope and lifetime. Provide an example to illustrate these differences.
- 7 Discuss the potential risks and benefits of using global variables in a program. Why is the use of global variables generally discouraged in favour of local variables?
- 8 Explain how the use of constants can enhance the readability and maintainability of code.
- 9 How does a sequence differ from an iteration?
- 10 What is the difference between a WHILE iteration and a DO/WHILE iteration?
- 11 Mary writes some code to iterate over a set of data. It runs exactly once, but no conditions have been met. What types of iterations could Mary have written?



## TEST YOUR KNOWLEDGE

**12** State the values that will be returned or displayed in the following examples of pseudocode:

```
a
a ← 4
b ← 2
c ← 3
IF (a > c) OR (b > c) THEN
    RETURN True
ELSE
    RETURN False
ENDIF
```

```
b
age ← 10
IF age < 10 Then
    RETURN "Child"
ELSEIF age < 18 THEN
    RETURN "Teenager"
ELSE
    RETURN "Adult"
ENDIF
```

## Functions, classes, methods

- 13** How do you get a function to run within source code?
- 14** How are functions represented in pseudocode?
- 15** How is a class different from an object?
- 16** What is the relationship between a method and an event?
- 17** How are methods represented in pseudocode?
- 18** How does encapsulation contribute to the security and robustness of a program? Provide an example in any programming language to illustrate how encapsulation is implemented.
- 19** Explain the concept of inheritance and how it promotes code reusability.
- 20** Provide an example of a base class and a derived class in any programming language, and describe how the derived class inherits properties and methods from the base class.
- 21** Describe the concept of abstraction in object-oriented programming and explain how it simplifies complex systems.
- 22** Give an example of how abstraction can be applied in designing a software system, focusing on what details are hidden and why.
- 23** Use an example to explain how generalisation can help in reducing redundancy in code.



## Algorithms for sorting

- 24 Explain the steps taken to perform a selection sort, using a worked example as part of your explanation.
- 25 Explain the steps taken to perform a quick sort, using a worked example as part of your explanation.
- 26 Which is faster: quick sort or selection sort? Is this always the case? Explain.

## Algorithms for searching

- 27 Explain the steps taken to perform a linear search, using a worked example as part of your explanation.
- 28 Explain the steps taken to perform a binary search, using a worked example as part of your explanation.
- 29 Roland executes a linear search and then a binary search on a very large set of data. He searches for the same item in each of the searches. The linear search was much faster than the binary search. How is this possible?

## Validation techniques

- 30 Cerie needs to perform all three validation techniques on a particular input. In what order should Cerie perform these checks when she writes her source code? Why?
- 31 Annotate the following pseudocode to show where each validation technique is being used.

```
ALGORITHM validateInput()
BEGIN
    INPUT fullName
    INPUT age

    IF fullName = blank THEN
        PRINT "Please enter your full name."
    ENDIF

    IF isInteger(age) = False THEN
        PRINT "You must enter age as a whole number."
    ELSEIF age < 0 THEN
        PRINT "Invalid age."
    ENDIF

END
```



## TEST YOUR KNOWLEDGE

### Checking modules function correctly

- 32 What steps can be taken to ensure that modules function correctly?
- 33 What is the difference between a test case and test data?
- 34 What is the difference between a syntax error and a runtime error?
- 35 Dina executes some code on her computer and everything seems to slow down. Eventually, her computer crashes. What has most likely occurred within the code?
- 36 Why is it important for software companies to release patches?
- 37 What is the purpose of a truth table in relation to testing?
- 38 Explain what a boundary value is.
- 39 Why are trace tables useful when debugging?
- 40 Complete a trace table for the following algorithm.

```
ALGORITHM printResult()  
BEGIN  
    x ← 2  
    y ← 1  
  
    WHILE y < x DO  
        y ← x * y  
        x ← x + x  
    ENDWHILE  
    PRINT x, y  
END
```

- 41 Construct a truth table and algebraically determine the test data required to test the following algorithm.

```
ALGORITHM returnResult()  
BEGIN  
    INPUT x, y, z  
  
    IF x < y AND z > y THEN  
        RETURN True  
    ELSE  
        RETURN False  
    ENDIF  
END
```



Create the following modules in the object-oriented programming language of your choice. For each module, create the solution and conduct testing.

## Module 1

Create a command line interface that simulates a simple banking transaction system. The program will handle operations such as deposit, withdrawal and check balance.

### Initial set-up

Start with an initial balance, \$1000.

### Functional requirements

Display Menu:

- Prompt the user to choose an operation: Deposit, Withdraw, Check Balance or Exit.
- Based on the user's choice, perform the corresponding operation.

Deposit:

- Ask the user for the amount to deposit (positive number).
- Add the deposit amount to the balance.
- Run the 'Display Menu' functional requirement.

Withdraw:

- Ask the user for the amount to withdraw (positive number).
- Check if the balance is sufficient for withdrawal.
- If sufficient, deduct the amount from the balance.
- If not sufficient, display an error message "Not enough funds."
- Run the 'Display Menu' functional requirement.

Check Balance:

- Display the current balance.
- Run the 'Display Menu' functional requirement.

Exit:

- Exit the program.

## Module 2

Modify the program you created in Module 1 to save customer balance to a file. Here are the modified requirements:

### Initial set-up

Read a file named balance.csv that contains customer IDs and their bank balance.



## APPLY YOUR KNOWLEDGE

### Functional requirements

Display Login:

- Prompt the user for their customer ID.
- Run the functional requirement 'Load Customer' based on the customer ID.

Load Customer:

- Iterate through the contents of the balance.csv file for the customer ID provided.
- If present, set the balance to the customer's stored balance, then run the 'Display Menu' functional requirement.
- If not present, display the message "Customer does not exist".
- Allow repeated attempts at customer ID until the user is either validated, or decides to exit.

### Module 3

Modify the program you created in Modules 1 and 2 to integrate a graphical user interface (GUI) as well as sorting and binary search. Here are the modified requirements:

### Functional requirements

Display Menu:

- Display a window that has a button for each of the following operations: Deposit, Withdraw, Check Balance or Exit.
- Based on the user's choice, run the corresponding operation.

Load Customer:

- Open the balance.csv file and read all contents.
- Sort the contents of the file using selection sort or quick sort.
- Run the 'Find Customer' operation using the customer ID provided and the sorted contents of the file.
- If the 'Find Customer' operation returns true, prompt the user to choose an operation: Deposit, Withdraw or Check Balance.
- If the 'Find Customer' operation returns false, display the message "Customer does not exist".
- Allow repeated attempts at customer ID until the user is either validated or decides to exit.

Find Customer:

- Iterate through the file contents using binary search based on customer ID.
- If present, set the balance to the customer's stored balance.
- Return a 'true' value back to Load Customer.



## Module 4

Modify the program you created in Modules 1–3 to use classes and objects for the Customer. Here are the modified requirements:

### Initial set-up

- Create a ‘Customer’ class that contains the following methods:
  - » `getCustomerID()`
    - returns the customerID
  - » `setCustomerID()`
    - sets the customerID to an integer value
  - » `checkBalance()`
    - returns the balance on the account
  - » `deposit(amount)`
    - adds the amount to the balance, returns the balance
  - » `withdraw(amount)`
    - subtracts the amount from the balance, returns the balance
  - » `saveCustomer()`
    - saves the customer information to the balance.csv file (customerID, balance).

### Functional requirements

Load Customer:

- Open the balance.csv file and read all contents.
- Sort the contents of the file using selection sort or quick sort.
- Run the ‘Find Customer’ operation using the customer ID provided and the sorted contents of the file.
- If the ‘Find Customer’ operation returns true, update the window to allow the user to choose an operation: Deposit, Withdraw or Check Balance.
- If the ‘Find Customer’ operation returns false, display a pop-up box message “Customer does not exist, would you like to create an account?”.
- If the user selects “Yes”, clear the screen and run the Add Customer operation.

Add Customer:

- Ask the user for a customer ID.
- Check the balance.csv file to see if that customerID exists.
- If it does not exist, add the customerID to the balance.csv file, set the account balance to 0, then prompt the user to choose an operation: Deposit, Withdraw or Check Balance.
- If it does exist, display the message “Customer ID exists, try again.”, then run the Add Customer operation again.



## APPLY YOUR KNOWLEDGE

### Find Customer:

- Iterate through the file contents using binary search based on customer ID.
- If present, create a new instance of the Customer class, then set the balance to the customer's stored balance using the `checkBalance()` method.
- Display the customer on the screen, including customer ID and balance.
- Return a 'true' value back to Load Customer.

### Deposit:

- Display a window that has an entry box for a deposit amount (positive number) and a button labelled 'Deposit'.
- If the user presses the button, add the deposit amount to the balance using the `deposit(amount)` method.
- Call the `saveCustomer()` method in the customer object.
- Return the user to the Display Menu.

### Withdraw:

- Display a window that has an entry box for a withdrawal amount (positive number) and a button labelled 'Withdraw'.
- If the user presses the button, check if the balance is sufficient for withdrawal.
- If sufficient, deduct the amount from the balance. Call the `saveCustomer()` method in the customer object.
- If not sufficient, display a pop-up box with the error message "Not enough funds."
- Return the user to the Display Menu.

### Check Balance:

- Display a window that has the current balance using the `checkBalance()` method and a button labelled 'Back to Menu'.
- If the user presses the button, return the user to the Display Menu.

## PREPARING FOR

# Unit

# 3

### OUTCOME 1

On completion of this unit, the student should be able to interpret teacher-provided solution requirements and designs and use appropriate features of an object-oriented programming language to develop working software modules.

## Steps to follow

To achieve this outcome, you will draw on key knowledge and key skills outlined in Area of Study 1. This outcome requires that you use an object-oriented programming language to create four working modules, with increasing complexity, in response to teacher-provided requirements and designs. At least two of these modules will include a GUI, and all modules must include testing. The working modules do not need to represent complete solutions, but the modules themselves should be complete as outlined in the design specifications provided to you.

Your teacher may choose to give you the requirements and designs one at a time after periods of relevant theory and programming instruction, or they may give them to you as a group after all of the theory and programming has been covered.

## Documents required for assessment

- Source code
- Internal documentation within the source code
- Evidence of testing using appropriate test data
  - » Keep and submit all documents used to construct your tests, including test data and any truth tables used to select your test data.

## Assessment

You may be assessed on the following measures:

- Your choice of data types and data structures
- Your choice of file types
- Selection, creation and use of appropriate processing features
- Selection, creation and use of appropriate sorting and searching algorithms within functions or methods
- Your use of classes and objects
- Naming conventions
- Validation techniques
- Debugging techniques
- Internal documentation of modules
- Thoroughness of testing

## KEY KNOWLEDGE

After completing this chapter, you will be able to demonstrate knowledge of:

- KK3.2.1: reasons why individuals and organisations undertake software development projects
- KK3.2.2: features of a brief that documents a problem, need or opportunity
- KK3.2.3: features of project management using Gantt charts
- KK3.2.4: methods for collecting data to determine needs and requirements
- KK3.2.5: characteristics of functional and non-functional requirements
- KK3.2.6: constraints that influence solution development
- KK3.2.7: characteristics of solution scope
- KK3.2.8: analytical tools for depicting the relationships between users, data and systems, including context diagrams, data flow diagrams and use case diagrams
- KK3.2.9: purpose and features of software requirements specifications.

Victorian Curriculum and Assessment Authority, Victorian Certificate of Education Applied Computing Study Design 2025–2028, VCAA, 2024, pp. 56–8

## FOR THE STUDENT

The process of creating documentation related to project management and for the specifications and design of a software system must be systematic in order to ensure success. Throughout both Units 3 and 4, you will be required to maintain a project management progress report in the form of a Gantt chart. The SAT requires that the project management report be submitted twice. In Unit 3, it will be in the form of a proposed plan. In Unit 4, it is an amended plan and, with the benefit of hindsight, includes changes that reflect the actual project progress, rather than the imagined progress as recorded by the proposed plan in Unit 3. Data collection techniques help to determine software requirements, both functional and non-functional, and also help to determine the scope of the system, particularly when considering the constraints faced by the software developer. Data collection must be interpreted using tools such as use case diagrams (UCDs), context diagrams and data flow diagrams (DFDs) to provide an increasingly detailed overview of the system being built. At this stage of the problem-solving methodology, it is also important to determine the criteria that will be used to evaluate the efficiency and effectiveness of design ideas in order to select a solution to develop further.

This chapter, along with Chapter 4, covers the theory required for Unit 3, Outcome 2, which is the first part of the SAT. The SAT will be completed in Unit 4, Outcome 1. Chapter 3 focuses on the analysis stage of the problem-solving methodology (PSM). Chapter 4 will deal with the design stage. The focus of this chapter is the core elements of analysis that will help you understand the requirements of project management and to construct a software requirements specification (SRS) for the proposed systems. You should apply data collection techniques to obtain information that will help determine the constraints and scope of a proposed system as well as document its functional and non-functional requirements. As part of this documentation, you will need to construct use case diagrams using UML, as well as context and data flow diagrams to represent the inputs, processes and outputs of systems and solutions. You need to propose different design ideas for your solution. At the same time, you will need to construct and use evaluation criteria to determine the most efficient and effective design.



# Why develop software?

**Software** is the means by which computer hardware can achieve a purpose. These purposes can be varied, but include:

- increasing productivity and efficiency
- reducing costs
- identifying opportunities to address gaps in the market
- meeting organisational objectives or needs.

Individuals and organisations undertake software development projects for a variety of reasons, each aiming to bring about certain benefits or to meet specific objectives.

## Increasing productivity and efficiency

One of the primary motivations for software development is to enhance productivity and efficiency within an organisation or for individual users. Software solutions are often designed to automate repetitive tasks, streamline complex processes and facilitate easier access to information. Automation and streamlining lead to a significant reduction in the time and effort required to complete tasks, thereby increasing overall productivity. For example, a custom project management tool can help a team track progress more effectively than using manual methods.

## Reducing costs

Software development can also be aimed at cost reduction. By automating processes and reducing the need for manual labour, software can significantly cut operational costs. Over time, the investment in software development often pays off by lowering the expenses associated with manual processes. For example, an automated invoicing system can reduce the need for a large accounting department, thereby saving on labour costs.

## Identifying opportunities to address gaps in the market

Organisations and individuals may engage in software development to capitalise on unmet needs or gaps in the market. By developing innovative software solutions that address these gaps, they can create new business opportunities and gain a competitive edge. This is often seen in tech startups that identify a specific need or problem and develop software to address it, potentially capturing a significant market share.

## Meeting organisational objectives or needs

Software development projects are often aligned with specific organisational goals or needs. This can include a wide range of objectives, such as improving customer service, enhancing data security, expanding into new markets, or supporting remote work. For example, a company might develop a customer relationship management (CRM) system to better track and manage customer interactions, aligning with a goal to improve customer satisfaction.

You will study organisational goals and objectives further in Unit 4, Outcome 2.

### SCHOOL-ASSESSED TASK TRACKER

- Design brief
- Project plan
- Analysis
- Folio of alternative design ideas
- Development
- Evaluation and assessment
- Final submission

## Documenting problems

Over the course of Units 3 and 4, the SAT will take you through a series of steps. First, however, you will need to document, in a brief, a problem, need or opportunity within your community. The brief should consider proposed users, the programming language used, the feasibility of the solution and the originality of it. This brief becomes the first part of your Unit 3, Outcome 2 assessment.

You will then collect information about the needs and requirements of the intended users of the solution and use this **data** to produce a software requirements specification (SRS).

As part of Unit 3, Outcome 2 follows a **problem-solving methodology (PSM)** to go through the **analysis stage** and the **design stage**. Development and evaluation will take place in Unit 4, Area of Study 1.

To prepare you for Unit 3, Outcome 2 and also for Unit 4, Outcome 1, we will discuss planning and managing a complex project, including how to use a Gantt chart, because this relates to the project you are undertaking.

As part of Unit 3, Outcome 2, you will need to collect data to determine the requirements of the software solution. A variety of techniques will be considered, including interviews, surveys, reports and observations.

Next, we will talk about the features of functional and non-functional requirements for a solution, both in terms of general solutions and your specific software solution.

Then we will discuss factors such as scope and constraints that will affect the ability to deliver the solution and to satisfy user expectations.

Before you begin any software design, you need to create the software requirements specification (SRS). In order to explain your intended software solution, you will use tools and techniques to illustrate the relationships between your solution, users and networks. The tools used will include use case diagrams (UCDs) with Unified Modeling Language (UML), context diagrams and data flow diagrams (DFDs).

The first thing to do after choosing a problem, need or opportunity is to gather data about the requirements that will help determine what is contained within the software solution. You can acquire data through methods such as surveys and reports, and through conducting interviews and observations.

Once you have gathered all of this data, you need to store it, protect it and understand what type of data it is. We discuss how to reference those sources properly and briefly cover data types and structures, which is relevant to Outcome 2, later in Chapter 4.

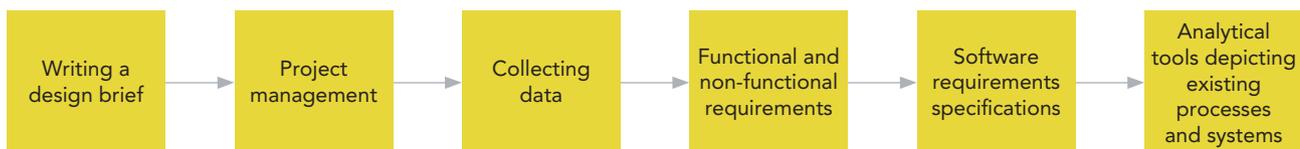


FIGURE 3.1 Chapter map

## Writing a design brief

A design brief is like a roadmap for a software project. It helps everyone understand what the project is about, what needs to be done and how to do it. It also acts as a reference for when you create your software requirements specification. Overall, a design brief ensures that the project runs smoothly and meets its goals.

Design briefs should be short and concise, covering the key elements using subheadings and brief descriptions. Firstly, it should contain an outline of the problem, need or opportunity, then a description of the proposed users and the programming language you are going to use. Finally, the feasibility and originality of the intended solution should be briefly discussed.

### Problem, need or opportunity outline

This section provides a detailed description of the specific problem, need or opportunity that your software will aim to address. It outlines the current challenges or gaps in existing solutions, if there are any, and explains why addressing this issue is important. For example, it might describe inefficiencies in current processes or emerging market opportunities.

### Proposed users

Describing the proposed users is important so you can provide context as to who the target audience is for the software. This section defines the primary and secondary users of the software, including their demographics and needs and how they will benefit from the solution. It might include user personas to illustrate the different types of users and their interactions with the software.

User personas are explained in Chapter 5, but you might want to consider using them in your brief.

### Programming languages to be used

The choice of programming languages and frameworks is documented here. This section justifies the selection based on factors such as compatibility with existing systems, developer expertise, performance requirements and scalability. For instance, it might specify using Python for its simplicity and wide range of libraries, or Java for its robustness and platform independence. For many of you, the programming language will have been selected by your school, but you should still be documenting it here.

### Feasibility

Assessing feasibility involves evaluating the technical, financial and operational aspects of the project. This section includes an outline of the resource requirements and a preliminary timeline, without specifying further details such as tasks. It examines whether the project can be completed within the given constraints and identifies potential obstacles.

## Originality

Originality focuses on how the proposed software stands out from existing solutions. This section highlights innovative features, unique selling points or competitive advantages. It explains how the software offers an approach to solving the problem, meeting the need or seizing the opportunity. Comparisons with similar products, including any product it is intending to replace, can illustrate what makes this solution distinctive and valuable.



### CASE STUDY

#### Design brief: Adem's gym

##### Problem, need or opportunity outline

Adem is a small gym owner who is currently facing challenges in managing memberships, scheduling classes and tracking member progress using outdated manual processes. This inefficiency leads to administrative errors, poor customer experience and reduced profits. Adem needs an affordable and easy-to-use gym management software that can streamline these operations.

##### Proposed users

The primary user of the proposed application will be Adem, the gym owner, who needs to manage the gym's operations efficiently. Secondary users could include gym staff, such as trainers and receptionists, who will use the software to check in members, schedule classes and record progress.

##### Programming languages to be used

The software will be developed using Python for the back end to handle data processing and operations, and JavaScript with the React framework for the front end to ensure a smooth and interactive user experience. Data storage will be in plain text, but it is not yet decided which format.

##### Feasibility

###### Technical feasibility

Python and JavaScript are well-supported languages with robust libraries and frameworks, making development feasible. No external hardware will need to be purchased as the software will be installed on existing systems at the gym.

###### Financial feasibility

Development costs will be minimal, as the main developer is volunteering their time across the duration of the project, which is estimated to take two months to complete. Adem plans to reward the developer with a free gym membership.

###### Operational feasibility

The project is planned for completion within two months, with key milestones including analysis, design, development and evaluation. Potential risks include delays in each of the milestones, particularly development, as the developer is a secondary school student who will not be able to work on the project in a full-time capacity.

##### Originality

The proposed gym management software will stand out by offering a user-friendly interface designed specifically for small gym owners. It will include features such as automated member check-in via a mobile app, class scheduling with automated reminders and simple progress tracking tools. Unlike existing solutions, this software will be tailored to the unique needs of Adem's gym, providing an affordable and streamlined solution.

# Project management

**Project management** is the process of planning, organising and monitoring a project in order to ensure that it is completed on time and within budget and scope. Building or changing information systems for a project can be expensive and disruptive and, if managed badly, can be damaging to an organisation's operations and profit. Large-scale changes are often approached as projects so they can be planned, organised and conducted appropriately, ensuring that they finish on time and within budget, and fulfil the project's goals (scope). You will formulate a project plan to manage your progress through Unit 3, Outcome 2 and Unit 4, Outcome 1.

For your project to be successful, you need to identify, schedule and monitor tasks, resources, people and time. While you can use a software tool for planning a project, our main focus initially will be on the **concepts** and **processes** of project management.

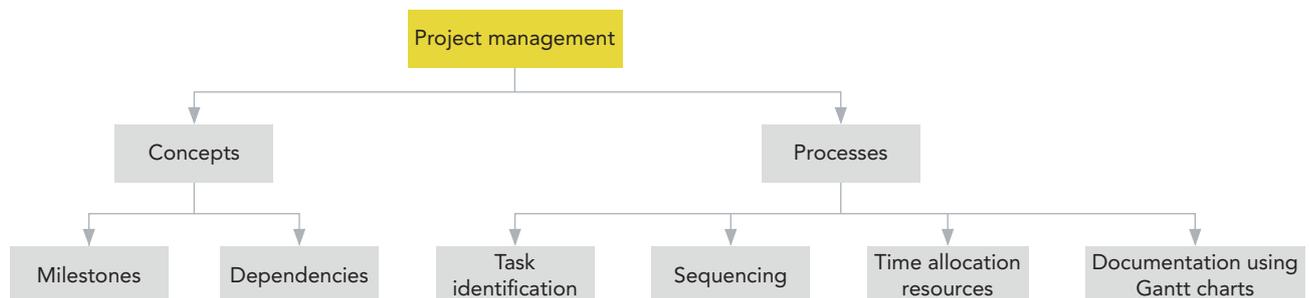
One of the items you are required to submit as part of your outcome is a **Gantt chart**.

A Gantt chart is a type of bar chart or graphic timeline named after its inventor, Henry Gantt, that:

- lists all tasks in a project
- organises the tasks in order
- shows which tasks must wait for other tasks to finish before they can begin
- allocates people and resources to tasks
- tracks the progress of tasks and the entire project.

Although you can create a Gantt chart with a pen and paper or a spreadsheet, project management software is usually easier and faster. Suitable software includes the commercial Microsoft Project, and the free, easy-to-use GanttProject and ProjectLibre.

When using software to create your Gantt chart, you will not be assessed on your technical prowess with the software. Rather, you will be assessed on how well your Gantt chart demonstrates your understanding of the concepts and processes of project management.



**FIGURE 3.2** Key project management concepts and processes

## Concepts

### Milestones

A milestone represents the achievement of a significant stage in a project and has zero time duration. For example, the completion of the printing of a questionnaire so that it can be distributed to respondents would be a task of zero time and represents a milestone. This follows tasks in which the questionnaire has to be researched, written, proofread and finally printed, all of which take time.

## Dependencies

Tasks are interdependent, meaning that they must be completed in a particular order. The commencement of some tasks depends directly on the task that is completed before. For example, you cannot distribute a questionnaire (one task) before writing the questions for it (another task). However, you cannot write the questions for the questionnaire without first determining what information, or data, you need (a third task). Ultimately, the task of distributing a questionnaire has multiple dependencies. It is not possible to distribute the questionnaire without first writing the questions.

## Processes

### Task identification

You would break a large project such as Unit 3, Outcome 2 down into discrete tasks, such as the following:

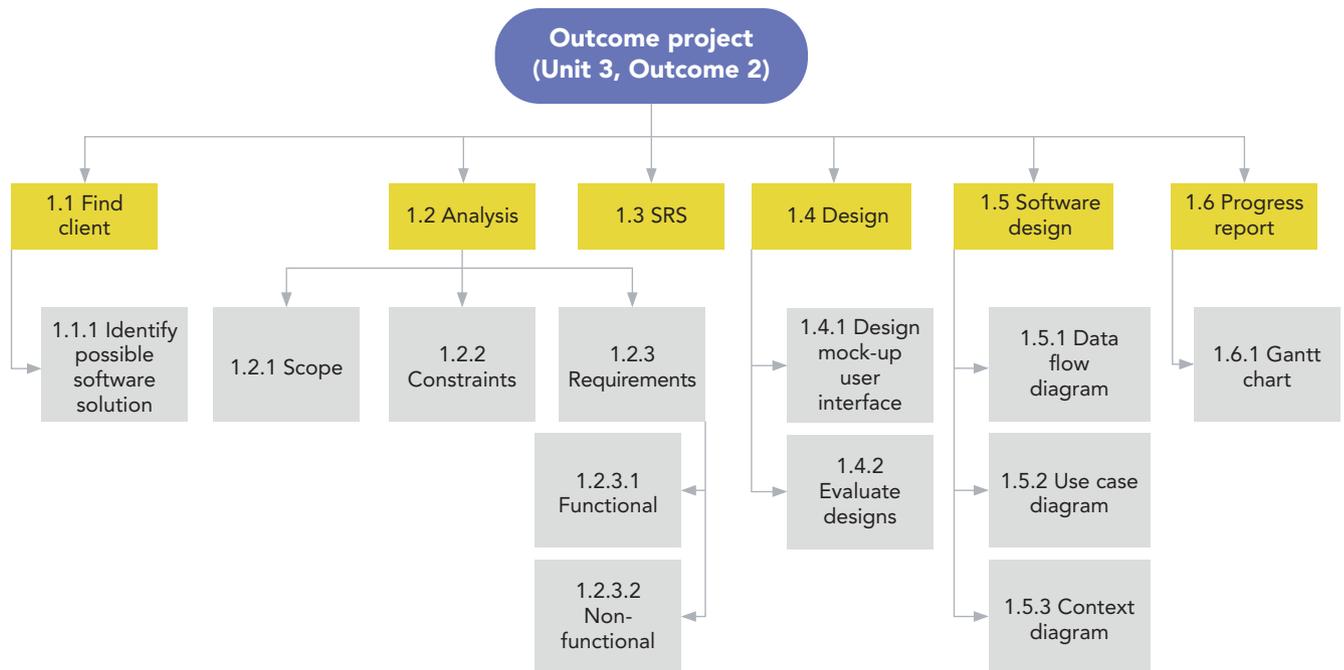
- Identify a problem, need or opportunity.
- Write a design brief outlining the problem, need or opportunity.
- Analyse the client's needs and requirements.
- Create a Gantt chart to identify tasks.
- Create a working title for your software solution.
- Write qualitative questions for interviews.
- Conduct interview(s) with the client and possible users.
- Write quantitative questions for questionnaires.
- Distribute questionnaires.
- Locate other data sets and secondary data.
- Collate data.
- Interpret data and create the SRS.
- Modify your SRS after discussion and agreement with your client.
- Finalise the purpose and function of your software solution.

Note: Not all of the tasks you would undertake for Unit 3, Outcome 2 are included in the list above. You should not use this as your own exhaustive list.

To break down your project into achievable tasks, develop a **work breakdown structure (WBS)** and draw a WBS diagram to accompany it. For large projects, a WBS will often be hierarchical, breaking major tasks into subtasks and even sub-subtasks. Although this may sound confusing, it will actually keep your tasks organised and in context, allowing them to be collapsed or expanded to view overall task progress or fine details about how minor subtasks are proceeding.

Do not leave any tasks out of the WBS. For example, imagine you distributed both a print and online version of a questionnaire, but did not list the printed version in the WBS, and forgot to collect the printed forms from respondents. All the gathered data would be overlooked, or counted too late.

Figure 3.3 shows a sample potential WBS for Unit 3, Outcome 2. Again, the tasks may not be exhaustive and you may find that your own WBS requires additional tasks.



**FIGURE 3.3** Sample work breakdown structure (WBS) diagram for Unit 3, Outcome 2

## Sequencing

When you have identified each individual task, you need to estimate how long each task will take and then put them in a sequence; that is, arrange them in a particular order. As discussed in dependencies, one task often cannot be started before one or more other tasks have been completed.

Decide which tasks can be worked on **concurrently**, but are dependent on other tasks that have already been completed. For example, you could work concurrently on conducting interviews, researching similar applications and writing questions for a survey, but these tasks are all dependent on having a client with a need or opportunity.

Similarly, you cannot write an SRS before you have interpreted the data that you have collected in the analysis stage.

Tasks that must be completed before another task can begin are called **predecessors**. The dependent tasks are called **successors**.

If a predecessor runs overtime, all of its successors will be delayed, causing problems for other tasks and deadlines. This is where a Gantt chart becomes very useful – it helps to monitor tasks and meet deadlines, keeping the project on schedule. It also helps you to visualise the problems that will occur down the line if a predecessor is late.

The amount of time that a task can be delayed without delaying another task, or the project completion date, is called slack time. When workers have **slack time**, you can reassign them to other tasks.

## Time allocation resources

A Gantt chart shows tasks as horizontal bars. Each horizontal bar is of a length proportional to the task's duration. A very short task will have a very short bar, while a very long task will have a very long bar. Figure 3.4 displays a number of features typical of a Gantt chart.

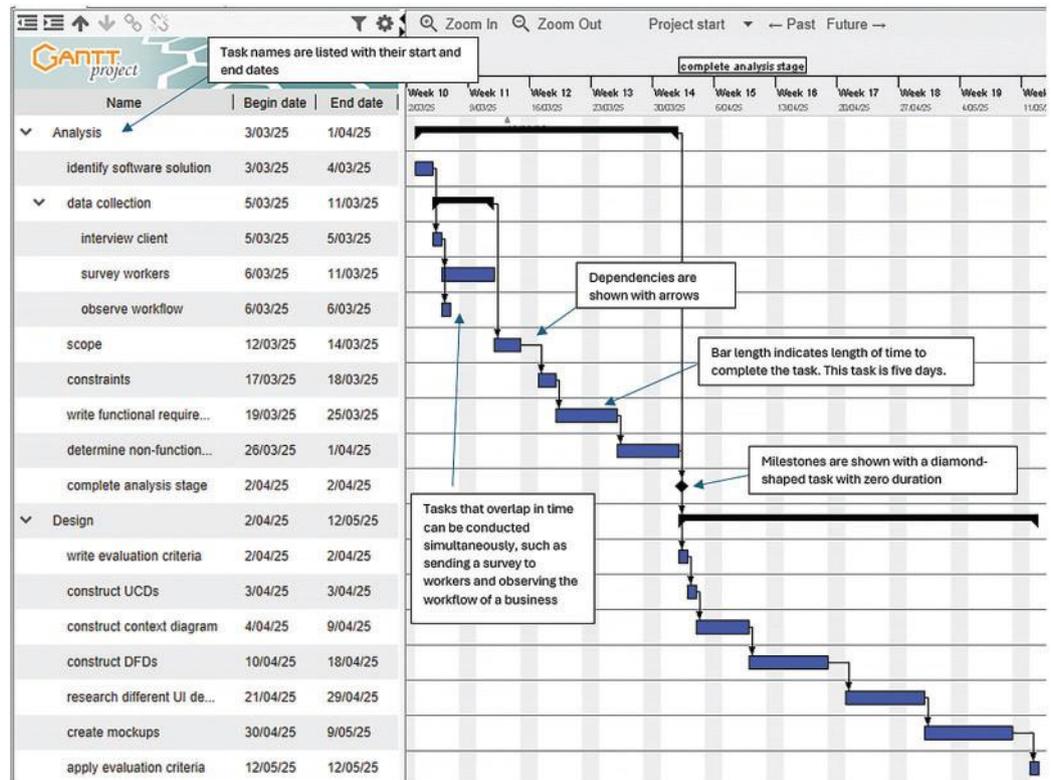
The names of the tasks are shown in the left pane, along with start and end dates, while the right pane shows task timelines. Tasks that overlap in time are concurrent and can be carried out at the same time using different teams.

SCHOOL-ASSESSED TASK TRACKER

Design brief  Project plan  Analysis  Folio of alternative design ideas  Development  Evaluation and assessment  Final submission



Resource  
Sample GanttProject  
chart



**FIGURE 3.4** An annotated Gantt chart. This Gantt chart provides a partial sample model for Unit 3, Outcome 2, with placeholder dates.

Arrows are used to indicate dependency. For example, neither the ‘scope’ nor the ‘constraints’ tasks can be completed until after every task in the ‘data collection’ heading has finished, because they depend on these tasks.

The diamond shape indicates a milestone. Milestones are points of significant progress in a project. They are often the start or end of major stages, and can be used to monitor whether a project is on track. A milestone is an **event** with zero duration and no allocated resources. It is simply shown as a diamond-shaped ‘task’. In this instance, ‘complete analysis stage’ is one such milestone.

An event differs from a task because although something happens (e.g. a major task ends), no resources, work or time are allocated to it because there is nothing that people need to do to make it happen.

A project’s **critical path** is the sequence of tasks from beginning to end, which:

- contains no slack time – any delay in a task on the critical path will affect the end date of the project
- is the longest duration
- is the minimum possible time in which all of the project’s tasks can be completed.

Sometimes, more than one critical path is possible. No task on the critical path can have its duration changed without affecting the end date of the whole project.

### Monitoring and documenting the progress of projects

You can use the Gantt chart you develop to mark your progress throughout this area of study. Including information about the progress of the task and the planned versus actual duration of the task will help keep you on track.

To manage your solution effectively as a complex project, you should also use your Gantt chart to document the resources you have allocated to it, such as any tools and equipment.

#### THINK ABOUT SOFTWARE DEVELOPMENT

3.1

Project management tools are useful to find the perfect number of people needed on a task so it is finished as quickly as possible without anyone being idle. Using software, develop a Gantt chart to plan the baking of a cake. Assume you can use as many cooks as you want.

While a company or organisation may list consultants and buildings as resources, your resources might be computers, particular data sets and software tools.

You should also frequently annotate or modify your Gantt chart to reflect changes in your timeline, such as unforeseen events, incidents or emergencies that have changed the duration of any task. You may find that your client suddenly becomes unavailable or unwilling to continue, or the software language you wish to use is no longer acceptable, or the website hosting your analysis and results crashes and you lose your data. Your Gantt chart should show problems like these and how you reacted to them, such as finding a new client or a different software solution strategy, or switching languages.

You should keep your Gantt chart updated throughout both Unit 3, Outcome 2 and Unit 4, Outcome 1. You will submit an initial project plan, indicating times, resources and tasks, in Unit 3, Outcome 2. After modifying the plan to indicate changes, you will submit an evaluation of the plan in Unit 4, Outcome 1.

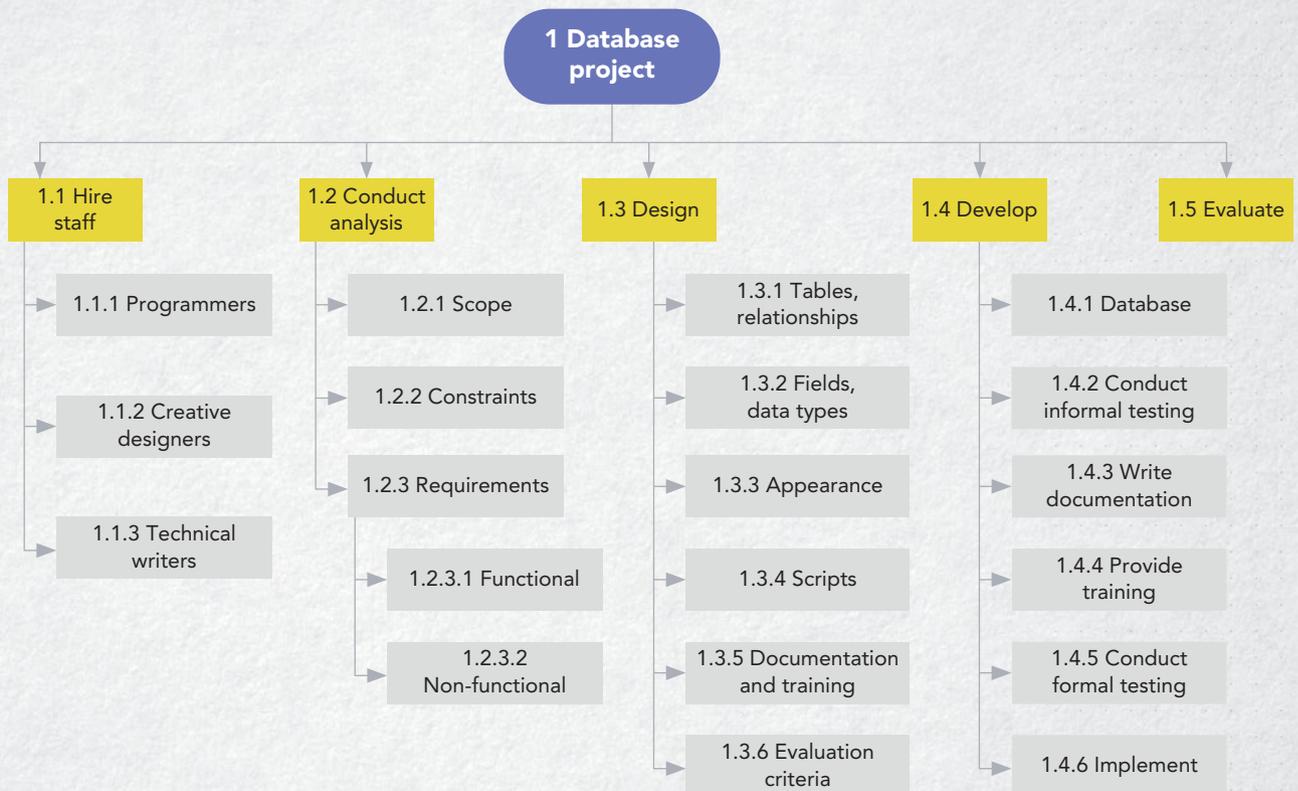
## Gantt chart for creating a database

Several web developers are working on a database project together. As part of their project, they need to build a Gantt chart.

### Task identification

They first identify the tasks they need to complete using a WBS diagram (see Figure 3.5). Next, they enter these tasks into their chosen Gantt software, GanttProject (see Figure 3.6).

**CASE  
STUDY**



**FIGURE 3.5** Task identification



SCHOOL-ASSESSED TASK TRACKER

Design brief

Project plan

Analysis

Folio of alternative design ideas

Development

Evaluation and assessment

Final submission



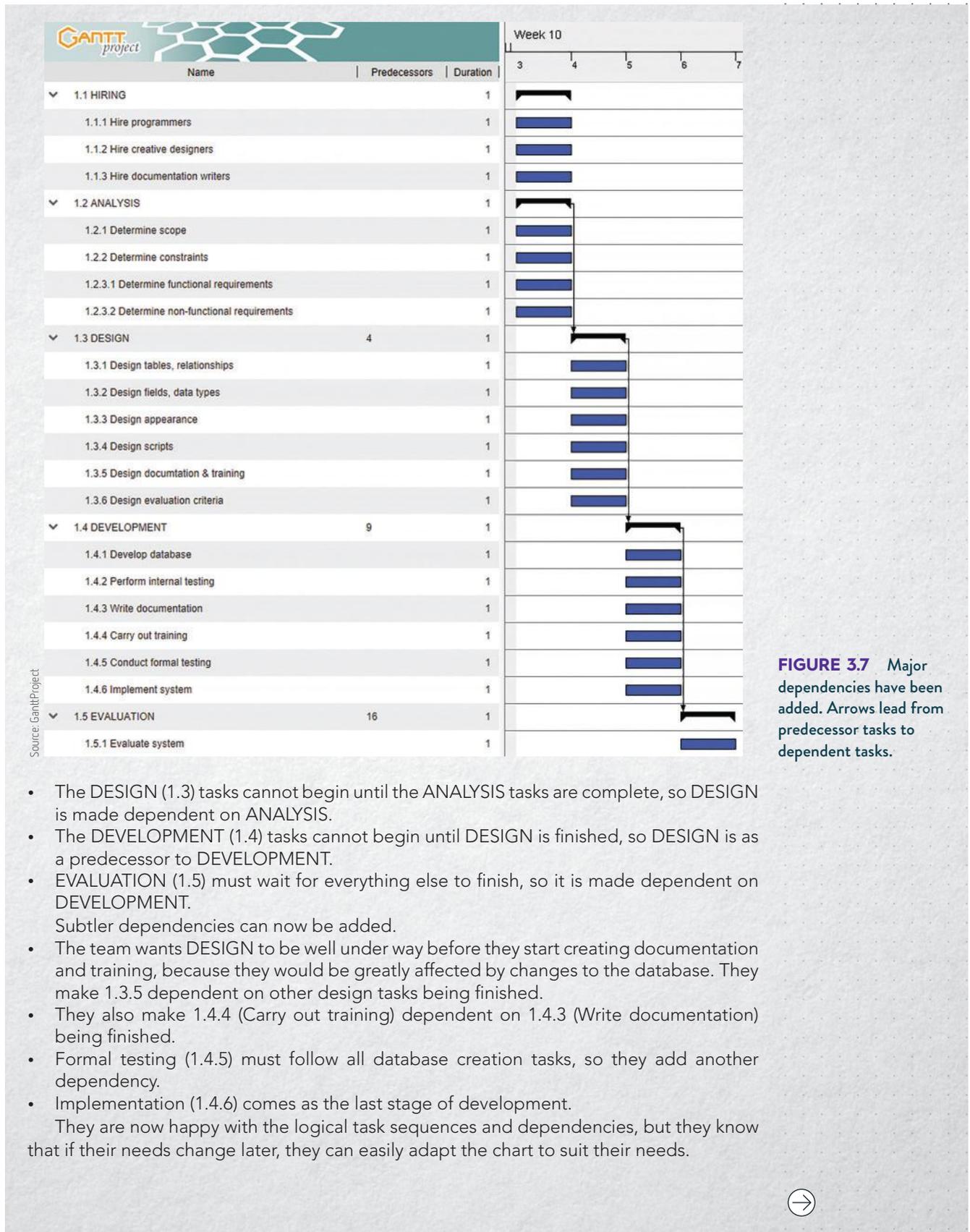
**FIGURE 3.6** Entering tasks into Gantt software

The developers use a hierarchical structure to group tasks under headings, such as HIRING and ANALYSIS, to make task management easier. Groups of tasks can be collapsed or expanded or moved as a group. In GanttProject, multiple levels of sub-tasks can easily be created by just indenting them in the task properties.

### Sequencing

The tasks are sufficiently sequenced, but the order can be shifted easily if needed. The developers start creating dependencies, forcing dependent tasks to wait until their predecessors have finished.

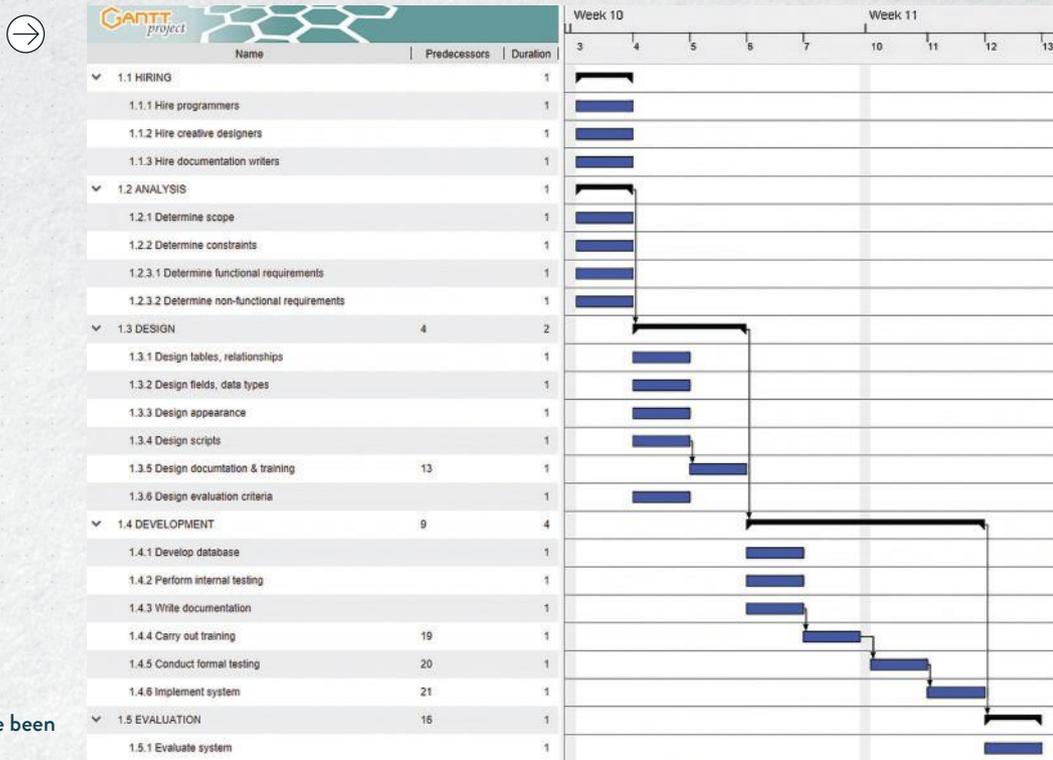
- All of the HIRING (1.1) tasks can start immediately.
- Management will complete the tasks in the ANALYSIS group (1.2), which can also begin immediately and run concurrently with the hiring tasks.



**FIGURE 3.7** Major dependencies have been added. Arrows lead from predecessor tasks to dependent tasks.

- The DESIGN (1.3) tasks cannot begin until the ANALYSIS tasks are complete, so DESIGN is made dependent on ANALYSIS.
  - The DEVELOPMENT (1.4) tasks cannot begin until DESIGN is finished, so DESIGN is as a predecessor to DEVELOPMENT.
  - EVALUATION (1.5) must wait for everything else to finish, so it is made dependent on DEVELOPMENT.
- Subtler dependencies can now be added.
- The team wants DESIGN to be well under way before they start creating documentation and training, because they would be greatly affected by changes to the database. They make 1.3.5 dependent on other design tasks being finished.
  - They also make 1.4.4 (Carry out training) dependent on 1.4.3 (Write documentation) being finished.
  - Formal testing (1.4.5) must follow all database creation tasks, so they add another dependency.
  - Implementation (1.4.6) comes as the last stage of development.
- They are now happy with the logical task sequences and dependencies, but they know that if their needs change later, they can easily adapt the chart to suit their needs.

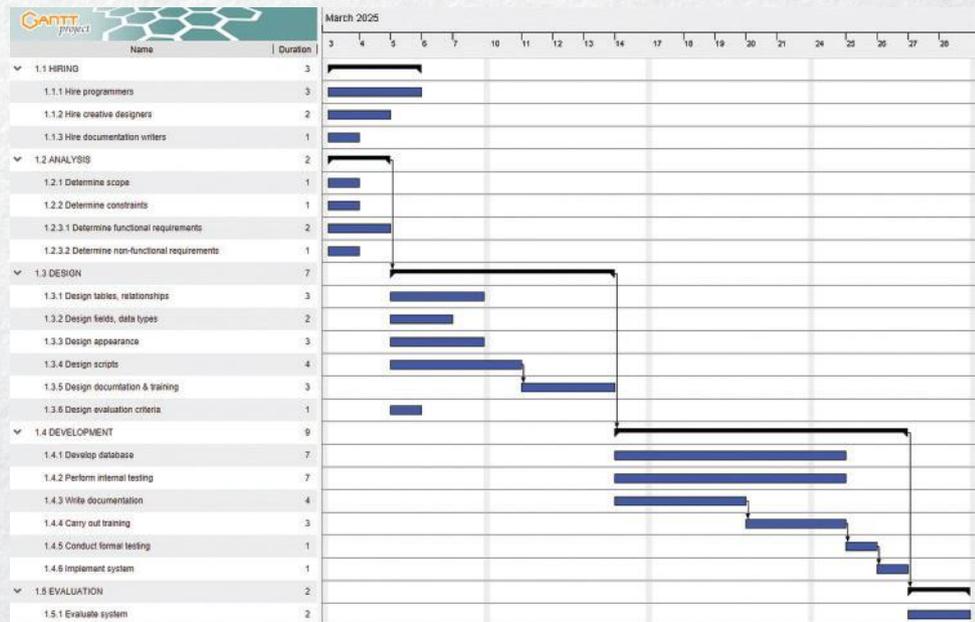




**FIGURE 3.8**  
Dependencies have been created.



**Resource**  
GanttProject chart  
showing task durations



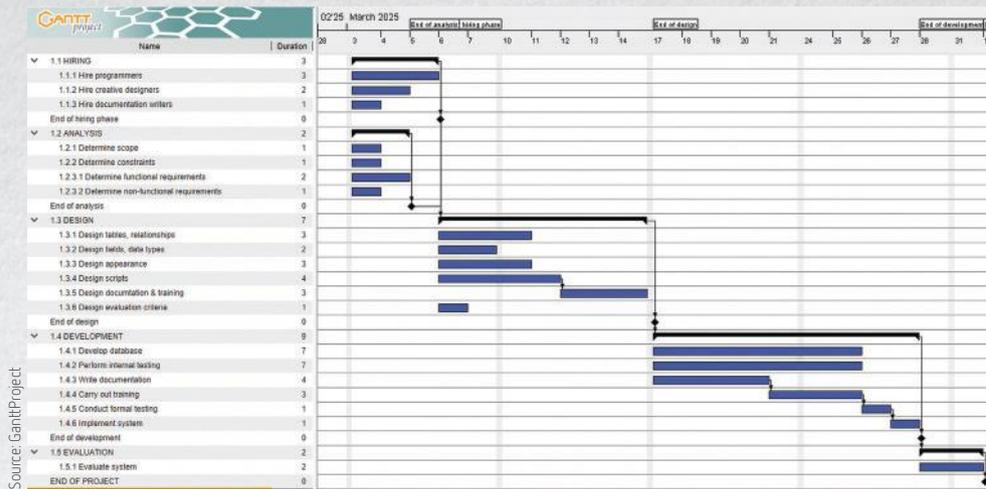
**FIGURE 3.9** Task durations have been estimated.

### Time allocation resources

The project developers now tackle the challenge of the time required for each task. They consult with experts and colleagues, and use their extensive experience and knowledge of past projects to guide their estimates.

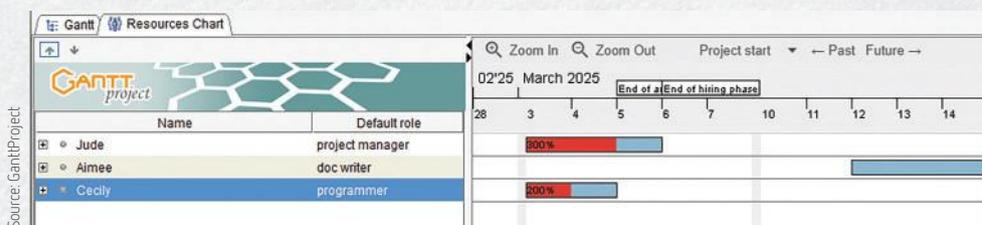
The developers can finish with their Gantt chart by:

- selecting a project start date
- showing a critical path
- adding milestones to mark the end of major stages of the project.



**Resource**  
GanttProject chart  
finished first draft

**FIGURE 3.10**  
Finished first draft of  
the Gantt chart



**FIGURE 3.11** Adding  
resources to the project

## Questions

- 1 The project team has made a mistake with the starting date of the 'Evaluate system' task. Explain why. How could they fix it?
- 2 How much slack does the 'Hire creative designers' task have?
- 3 If designing scripts took a day longer than expected, would it affect the project end date?
- 4 The team discovers that the 'Develop database' task is running over time. How could they keep the project from running past its planned end date?
- 5 Explain the benefits of using Gantt chart software instead of using a pen and paper or a whiteboard.

## Activity

- 6 Obtain a Gantt software tool and create a simple chart of your own.

## Documentation using Gantt charts

To ensure that project workers are not booked to be in two places at once, or idle, and that equipment is ready at the right time, the project manager allocates resources to tasks using the Gantt chart.

Once the project is under way, the team will continue to refer to the Gantt chart to monitor their progress, and they will modify the chart when contingencies force plans to change.

While Gantt charts are one crucial aspect of project management, good file management practices are another. Wise file-naming strategies are easy to learn and useful in many ways. You will find it easier to keep track of the materials you collect for your solution if you learn to manage your files by naming them wisely.

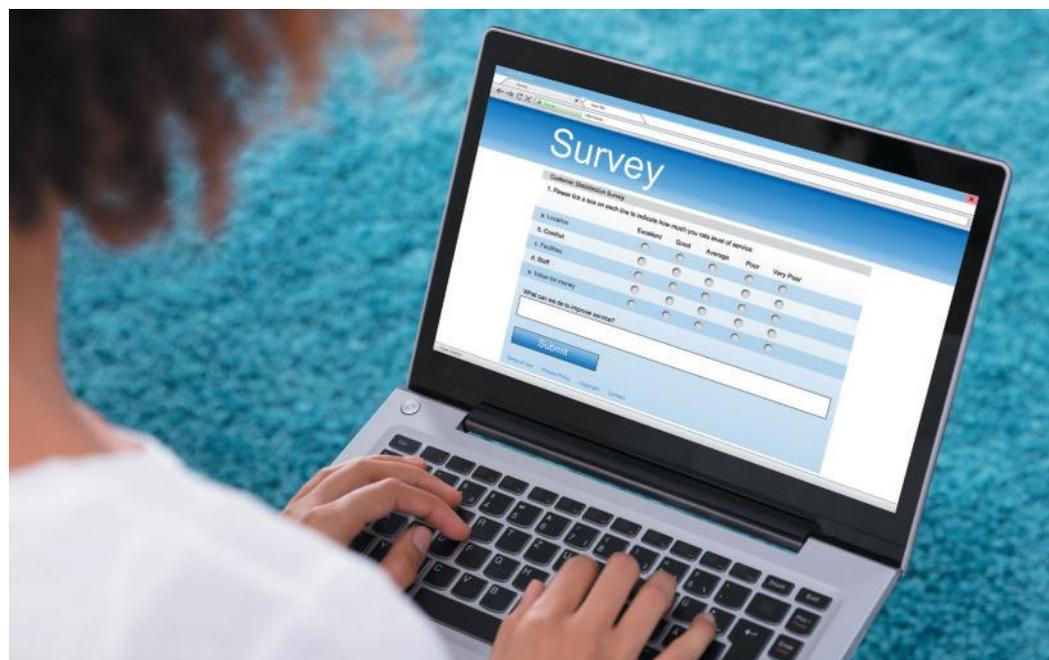
## Collecting data

Data to construct a software requirements specification (SRS) is usually collected through methods such as surveys, interviews, reports and observations. Each method of data collection has its strengths and weaknesses, and it is important that the most relevant methods of data collection are used within a given context. In Software Development, data that is collected typically helps determine the project scope, the functional and non-functional requirements, and the constraints. It can also help determine and/or decide who the end users will be, or how the client is going to use the software.

### Survey

A **survey** is usually a set of questions that ask for a response to be selected from a list of alternatives, such as A, B, C, D; or a range, such as 1 to 5 or very low to very high. This type of survey is also called a questionnaire. Surveys can easily be given to many people, and are quickly processed and analysed using computer-based methods because the answers can be recorded as numbers as long as **close-ended questions** are used. Close-ended questions are questions where the answers are either Boolean (yes or no) or ranked on a fixed scale. These types of questions allow for analysis of **quantitative data** to produce results, which is more efficient than surveys with **open-ended questions**. Open-ended questions are questions that ask for answers in sentences or worded form. This means the number of answers is potentially infinite. These questions tend to ask for opinions, and must undergo analysis of **qualitative data** to interpret the results.

The advantage of administering a survey is that they are relatively inexpensive to conduct. They can also be delivered digitally, so results can be collected immediately, and the survey can easily be given to a large number of users to complete. The disadvantages are that processing of results can take a lot of time if open-ended questions are asked, users may not necessarily be truthful in their responses and, if the survey is quite lengthy, users may lose interest in completing it.



**FIGURE 3.12** A survey with a close-ended question

## Interview

An **interview** is usually conducted face-to-face, sometimes in groups, and can take a substantial amount of time. Interviews give an opportunity for in-depth follow-up and the ability to ask clarifying questions – this cannot be done with a survey, which is often answered in private. Interviews are very useful for eliciting feelings, attitudes and opinions that are too complex to easily record in a survey.

## Report

A **report** is typically a written document providing a summary of findings in relation to the system being analysed. Often when a **software developer** is creating a new system to replace the system that is currently in place, they must investigate the current system as part of their analysis of the requirements of the new system. In this instance, it can be useful to collect data about the current system; often this data has been summarised in report documents. These could include error reports, customer complaint summaries, uptime reports and other related system reports such as system performance reports and monitoring reports. The types of reports collected will vary depending on the system being built and the system being replaced.

Source: Metro Trains

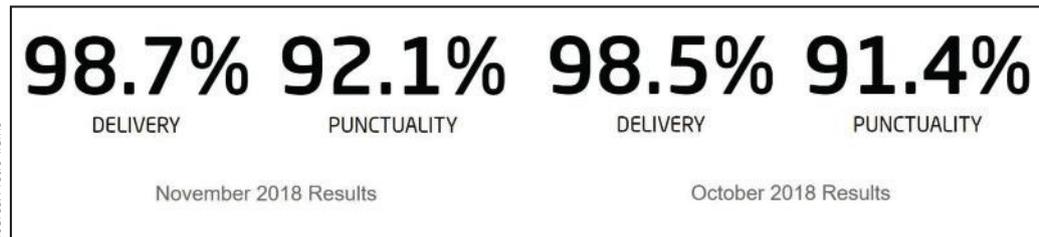


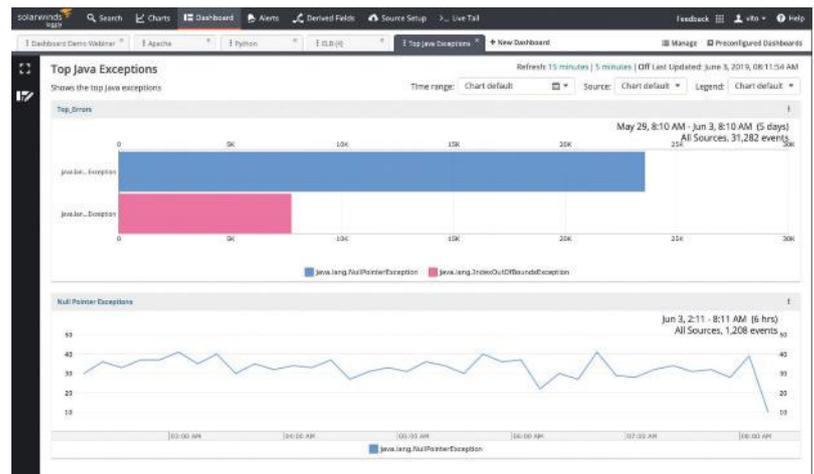
FIGURE 3.13 Metro train performance report

The advantages of using reports as part of data collection is that they are often prepared and quick to obtain, and therefore inexpensive. The data within them has already been collated and interpreted, saving time (Figure 3.13). This can also be a disadvantage, however, as the method of interpretation may not be sound, or may have been deliberately manipulated to present a particular point of view. Using data from reports can be risky if the source of those reports is not reliable (Figure 3.14).

Source: Dotcom-Monitor, Inc.



FIGURE 3.14 Server uptime report



Source: SolarWinds Worldwide, LLC.

FIGURE 3.15 Error log report

## Observation

**Observation** is a method of data collection that involves physically observing how a system operates and how it is used. Observations are typically performed when designing a system that will replace a current system, and are useful when attempting to solve a problem with an existing system. For example, a system may become particularly slow at certain times of the day, but will work well outside of those times. An observation of users at the time when the system slows would be useful in helping determine why the issue has occurred. Similarly, systems may have wildly varying performance depending on the time of year, such as consumer websites during periods of high sales such as Boxing Day or the end of the financial year (EOFY). Observations can also be used when creating a new system that will replace something that is currently being completed manually. In these instances, the manual process is observed and documented.

Advantages of observations are that they can provide an unbiased view of the system; information can be gathered without asking for a user's opinion or relying on their memory. Observations can also occur simultaneously as long as users are located in the same physical space. Disadvantages are that they are quite time-consuming and therefore very expensive if they need to occur more than once or over a long period of time. Time-specific observations can also be very difficult if the project timeline does not allow for it; for example, waiting for the end of the financial year may require waiting 12 months. Similarly, if the observations occur at the wrong time, the results may not be very useful as they may not be representative of how the system is typically used. Another disadvantage is that people can become very self-conscious when they are being observed, so their actions may not necessarily be the same as if they were alone, resulting in inaccurate data.

## Functional and non-functional requirements

Solution requirements are what the client needs from a solution; that is, what the system must do. These can be broken down into functional requirements and non-functional requirements.

### Functional requirements

**Functional requirements** are directly related to what the solution will do. These typically involve calculations; data processing; opening, reading and writing to files; data manipulation such as image editing; and other specific **functionality** required within the system.

Some examples of functional requirements for software solutions are:

- save customer data to a file
- calculate discount values on products
- set an alarm to go off at a particular time
- load a set of jobs into a timetable.

Functional requirements are usually described in terms of the inputs required, the sequence of operations that will be performed, and the output(s) after processing has occurred. These written descriptions often include dot-point descriptions for the purposes of clarity. An example of a functional requirement for a music performance system can be seen in Figure 3.16.

Functional requirements that require user interaction are typically accompanied by a use case; these are described in more detail later in this chapter.

### 3.1.1. Add Performance

This screen will allow the entering of performance information. Information consists of the performance name, date and theme, as well as a unique performance ID (numeric). Also recorded is whether the performance is a concert, and, if so, the theme of that concert.

#### Inputs:

- Performance ID (unique)
- Performance Name, Performance Date
- Whether it is a concert or not
- Concert Theme

#### Sequence of Operations:

- Input will come in from the User Interface, all items except concert theme are required fields
- Themes should be retrieved from a file containing all available themes so a user can select one
- No two concerts should have the same ID
- A theme only needs to be selected if it is a concert
- No concert should be added if a concert with the same theme has occurred in the last six years
- Once data is entered and validated, it should be saved to a performance file

#### Outputs:

A pop-up box should be displayed showing the success or failure of saving the concert information. Once information is saved, the user should be returned to the main menu.

**FIGURE 3.16** A functional requirement for a music performance system

## Non-functional requirements

**Non-functional requirements** are other requirements that the user or client would like the solution to have but that do not affect what the solution does. These tend to be referred to as *quality requirements*, as they typically involve criteria that can be used to ‘judge’ a system, rather than criteria that involve specific behaviour the system is required to have. Non-functional requirements can be categorised in terms of usability, reliability, portability, robustness and maintainability.

Non-functional requirements must be measurable – this means that they must be able to be tested to see if the requirement is met.

Determining non-functional requirements usually involves discussions with a client, such as asking if the software must work on different operating systems, or asking who the users of the system will be and the level of technical experience they have.

### Usability

**Usability** relates to how easy a system is to learn and use. This is typically described in terms of efficiency and effectiveness. Common factors of usability include the **clarity** of the user interface and the intuitiveness of the functions within the system. The success of a system’s usability is often measured in terms of user satisfaction. An example of a non-functional requirement related to usability is ‘users should be able to use all basic functions after one hour of training’.

This non-functional requirement is highly reliant on the skills, expertise and needs of the intended users of the system.

## Reliability

The **reliability** of a software solution relates to how much it can be depended upon to function as designed, and for how long. Typically, this requires that the software is deemed **fit for purpose** over time and that it is resistant to failure.

The reliability of a system is generally expressed as a probability measure, where *reliability* =  $1 - \text{the probability of failure}$ . An example of a reliability measure is the prediction of the **uptime** of a system or solution, such as ‘the system should have a 99.9% uptime over any 12-month period’ or ‘the ability to add a new customer should be available to staff members during working hours on weekdays’.

## Portability

The **portability** of a software solution relates to how easily it can be used in different operating environments. This can involve measuring how easy it is to move from one system to another system with the same architecture; how easy it is to reinstall a program on a new system; and the ability to use the same software on multiple operating systems and platforms, or in multiple languages.

The most time-consuming element of portability is when a developer is required to write software that will work on multiple operating systems and platforms, in particular when multiple languages are involved. This typically means that the user interface must be separated from the core functionality and logic so it is easy to create a new user interface for a new system – for example, creating an application that works on a mobile device as well as a desktop computer, in Mandarin as well as in English.

Portability is increasingly important as mobile computing becomes the norm; users often expect that applications will work on mobile technology as well as desktop technology.

## Robustness

The **robustness** of a system relates to how well a software solution responds to errors that occur when the software is being used; that is, it should perform correctly in every situation encountered by a user. Robustness is therefore an evaluation of the error handling techniques within a software solution. For example, a piece of software would be evaluated according to how it responds to bad input from a user. Robustness is closely linked to the use of validation techniques, as these can help prevent errors from occurring when users enter unexpected or invalid input. An example of a non-functional requirement related to robustness is ‘the system should reject invalid data entered by a user’.

Robustness is measured in terms of the number of failures, crashes and errors that occur while a system is running.

## Maintainability

**Maintainability** is related to how easy it is to look after software once it is being used. This can involve fixing errors in the code, maximising efficiency and reliability, installing the software on new systems, and in some cases expanding on the current functionality with new functionality. Simply put, maintainability is measured in terms of how easy it is to fix, modify or change the software once people are using it.

Often, maintainability is measured in terms of the number of hours a developer or administrator spends to keep the system running after it has been put in place. An example of a non-functional requirement related to maintainability is ‘fewer than 10 hours should be spent per quarter on maintaining the system’.

## Scope

The scope of the software solution must be defined. A project without limits will never be finished, and inevitably disputes will arise over what was and was not included in the expected software solution. A scope statement identifies who will develop the solution and what will be specifically considered, and it may specifically exclude other areas. In its simplest terms, the scope of the project describes the project and explains what the project will and will not do (see Chapter 1, page 9).

**Solution boundaries** in the context of software development and project management refer to the limits or edges of what a project or solution will encompass. These boundaries, the range and limitations of the solution, are crucial for defining the scope of a project and ensuring that the team's efforts are focused and aligned with project goals.

When considering the scope of a solution, it can be useful to think of each cycle of the PSM resulting in a **version** of the software that is fit for use. Each version is characterised by its unique scope, which defines the set of features, improvements and fixes that are included in that particular release.

Determining an accurate scope for a solution is essential for successful project completion. Clarity in the scope helps in managing expectations and preventing scope creep.

The scope defines the boundaries of the project, including technological, resource and time constraints. It also outlines the limitations, such as what will not be addressed in the current phase of the project. Not all features and functionalities can be developed at once due to constraints like time, budget and resources. Prioritising features based on their importance, value to users and feasibility is a critical aspect of scope management.

The solution scope is often influenced by the needs and expectations of various stakeholders, including clients, end users and team members. Maintaining clear documentation of the solution scope and communicating it effectively to all team members and stakeholders is essential for alignment and successful project execution.

## Constraints

**Constraints** play a critical role in shaping the final product. These constraints can be economic, legal, social and technical, and they often overlap in their influence on the direction and scope of a project.

### Economic

**Economic constraints** refer to the financial limitations and considerations that impact the planning and execution of a project. These constraints primarily revolve around budget restrictions, which dictate the scale, scope and resources available for the project. A limited budget may restrict the choice of technology, the size of the development team and the project's overall ambition.

Economic constraints also influence the prioritisation of project features and functionalities, compelling project managers to make strategic decisions about what can be achieved within the available financial resources. Additionally, these constraints necessitate a thorough cost–benefit analysis to ensure that the project's outcomes justify the investment. Economic constraints require careful and efficient resource management, balancing the project's goals with the reality of financial limitations to achieve the best possible outcomes within the set budget.

## Legal

**Legal constraints** refer to the laws, regulations and legal standards that must be adhered to in the planning, development and execution of a project. These constraints can vary widely depending on the industry, the type of project and the geographical locations involved.

For the purposes of VCE Software Development, you must consider the key legal requirements relating to the intellectual property and ownership and privacy of data used, including the *Copyright Act 1968*, the *Privacy Act 1988* and the *Privacy and Data Protection Act 2014*. These are covered in detail in Chapter 4, pages 175–9.

Navigating legal constraints is essential for ensuring that a project is not only compliant with the law but also sustainable in its practices. Failure to adhere to legal constraints can result in penalties, legal action and damage to reputation.

## Social

**Social constraints** refer to the limitations or restrictions imposed by societal norms, cultural practices, ethical considerations and public expectations that can influence the development and implementation of a project, policy or product. These constraints are shaped by the social context in which an organisation or individual operates and can significantly impact decision-making processes and outcomes.

Social constraints might involve considerations such as ensuring inclusivity and accessibility, respecting cultural sensitivities, adhering to ethical standards, and addressing the needs and expectations of various stakeholders and communities.

These constraints often require careful navigation, as they can affect public perception, user acceptance and the overall success of an initiative. Social constraints are particularly important in global or diverse settings, where varying cultural and social norms must be considered to ensure that solutions are appropriate and effective for different groups.

## Technical

**Technical constraints** refer to the limitations and restrictions related to the technology used in a project. These constraints can significantly influence the design, development and implementation of software applications.

Table 3.1 presents some of the technical elements that may be relevant to consider when determining solution constraints.

**TABLE 3.1** Technical constraint considerations

| Key aspect                      | Description  |
|---------------------------------|--|
| Hardware limitations            | Capabilities and limitations of the hardware, affecting performance parameters like processing speed, memory and storage |
| Software dependencies           | Existing software dependencies, such as operating systems, libraries and frameworks, that can limit development options  |
| Programming languages and tools | Choice of programming languages and development tools, influencing capabilities and project suitability                  |
| System integration              | Need to integrate with existing systems, affecting data exchange formats and communication protocols                     |
| Scalability and maintainability | Constraints related to software scalability and ease of maintenance, influencing design and architecture                 |
| Performance requirements        | Performance constraints like load times, response times and throughput, impacting technical choices                      |

**TABLE 3.1** Technical constraint considerations (continued)

| Key aspect                         | Description   |
|------------------------------------|---|
| Security requirements              | Security considerations, including data protection and encryption, imposing constraints on development            |
| User interface and user experience | Technical constraints related to user interface and overall user experience, affecting front-end development      |
| Network and connectivity           | Network capabilities like bandwidth and latency, especially important for cloud-based or distributed applications |
| Resource availability              | Availability of technical resources such as developer expertise, tools and technology infrastructure              |

## Software requirements specifications

A **software requirements specification (SRS)** is a single document that contains the outcomes of the analysis stage of the problem-solving methodology. This document is created after data collection has occurred and before the design stage begins.

An SRS must outline all of the elements considered in the analysis stage. In particular, the *constraints* under which the system must exist, the *scope* of the proposed system (what it will and will not contain) and the *functional* and *non-functional* requirements of the system itself. An SRS may also include an appendix containing additional information needed to interpret the requirements, such as a description of the operating environment of the proposed system (linked to constraints) or descriptions of any third-party tools required (linked to functional or non-functional requirements).

A well-written SRS provides quality assurance that the issue to be solved by the software solution is well documented and understood. It ensures that:

- the problem, need or opportunity is understood, the issues have been identified, and in response a systematic process of addressing each issue has been documented
- the completed SRS will be the basis for the design specifications, and the criteria identified in the SRS will be used to evaluate the success of the final product
- the final evaluation will verify the software product and test that the software performs as expected.

**FIGURE 3.17** Software requirements specification

SCHOOL-ASSESSED TASK TRACKER

Design brief
  Project plan
  Analysis
  Folio of design ideas
  Development
  Evaluation and assessment
  Final submission

Scope creep is when a client changes the scope of a project (by increasing it) during the life cycle of the project. This can be very expensive to a developer in terms of time and money, as it means the project will take longer to complete, often without any extra money being provided by the client.

An SRS can contain performance parameters such as response time, processing time, maintenance requirements, resource demands, and security and backup arrangements.

Diagrammatical tools that can be used to assist in the creation of the SRS include use cases, context diagrams and data flow diagrams.

An SRS is useful because it provides all of the required information about the proposed system in one place. It often also acts as a legally binding contract between a client and a developer, preventing a client from requesting additional features as the project timeline progresses (scope creep).

## Creating a software requirements specification

An SRS should include a cover page that states the name of the project, the author (or authors), the contact details of the author(s) and the version of the proposed system.

The first page after the cover page should contain a table of contents that clearly lists each section of the SRS. Each section inside the SRS should use numbered headings and subheadings so each element of the document can be easily tracked. Page numbers are also included on each page.

The SRS typically contains the following elements:

- technical environment(s): the technical environment in which the software will operate, including hardware, operating systems, network environments and other software products with which it must interface
- functional and non-functional requirements (pages 102–4)
- constraints (pages 105–7)
- scope (page 105).

It may also contain relevant diagrams that depict existing processes and systems.

Any additional documentation should be included at the end of the SRS as part of a set of appendices. This may include reports collected in the analysis stage, evidence of interviews, observations, etc.

Because an SRS is a professional document, consistency in the use of fonts, font sizes and font colours is also important.

## Analytical tools depicting existing processes and systems

There are many useful tools to interpret data collected in the analysis stage of the problem-solving methodology. Three methods of depicting the interfaces between solutions, users and networks are use case diagrams, context diagrams and data flow diagrams.

### Use case diagrams

A **use case diagram (UCD)** is a method of describing how a user interacts with a system. This is created using **Unified Modeling Language (UML)**, which is a general-purpose visual modelling language. Use cases are a diagrammatical representation of the externally visible user interactions, and are often used to complement worded descriptions of those

interactions. Use case diagrams are normally completed prior to the creation of context diagrams or data flow diagrams.

A use case diagram provides a structured view of the functionality of a software solution; it ‘tells a story’ of how the functions within a system work. It is intended to provide a high-level view of how a user actively interacts with the system, and does not show the internal functionality of the system itself. Use case diagrams are not designed to show the sequence or order of interactions a user undertakes within a system, either.

Use case diagrams include actors who have relationships with use cases.

## Actor

In a use case diagram, an **actor** represents an entity that can interact with the software solution. While this typically means a human user, it does not have to be, and can include external systems. Actors are described in terms of the role of a user or external system, rather than as specific users themselves. For example, if Serai was a manager in charge of adding users in a new system, she would be represented by an actor named ‘manager’ rather than by an actor named ‘Serai’.

Actors are represented by stick figures in use case diagrams, even if they are not humans. They can be connected to use cases and other actors through relationships and generalisations, and should always be described using nouns.

## Use case

A use case diagram contains **use cases**, which describe transactions or functions a user (actor) can complete in the system. These represent system functionality. Use cases are shown using an ellipse with the name of the function written inside. Each use case is often quite broad, as a use case diagram is often the first tool used to understand how a system will work. They are generally expanded upon in other diagrams, such as context diagrams and data flow diagrams, or in the functional requirements.

Use cases should begin with strong verbs that describe the action or function being represented in the use case. The phrase inside the use case should be brief, often no more than two or three words – for example, ‘add customer’ or ‘delete user’.

Use cases can be connected to actors and other use cases through relationships and generalisations.

## Relationship

A **relationship** in a use case diagram represents the connections between elements within the use case diagram. Relationships can exist between actor and use case, actor and actor, and use case and use case.

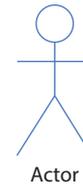
## Association

A typical relationship is represented by a solid line connecting two elements. This type of relationship is referred to as an **association**. This is the default relationship within a use case diagram.

## Generalisation

A second type of relationship is a **generalisation**, which indicates a type of inheritance, or parent–child relationship between the two elements. The ‘child’ in a generalisation gains all of the structure, behaviour and relationships that the parent has. A parent element can have more than one child element. This feature of a use case diagram is often used to show when

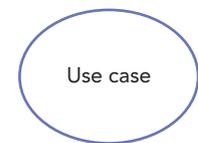
A use case diagram should show non-technical people what the system will do when it is completed. It should show developers what is expected of an application. It does not go into detail about how a system will implement functionality; it only shows what it will do.



Actor

**FIGURE 3.18**

An actor in a use case



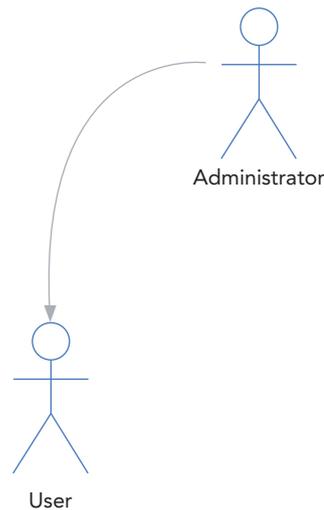
**FIGURE 3.19**

A use case in a use case diagram

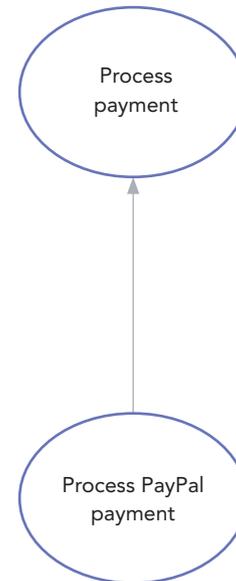
Note that ‘generalisation’ when used in the context of a use case diagram is not the same as ‘generalisation’ used in the context of object-oriented programming.

the same job can be performed differently by different users. It can also be used to show different types of use cases.

Generalisations are represented by a solid, straight line with a closed arrow head, with the child use case pointing to the parent use case. An example of a generalisation is shown in Figure 3.20, where two actors are connected via a generalisation. The 'Administrator' actor in this example can also perform the role of the 'User' actor. Another example is shown in Figure 3.21, where a parent use case 'Process payment' has a child use case 'Process PayPal payment' connected via generalisations.



**FIGURE 3.20** A generalisation of actors where an administrator is a type of user

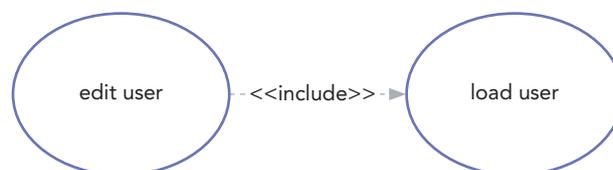


**FIGURE 3.21** A generalisation of use cases

### Include and extend

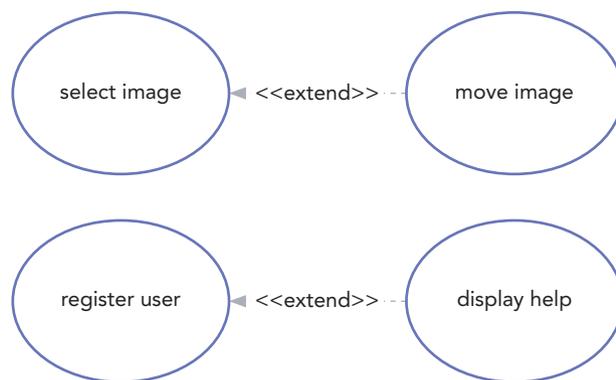
Two other types of relationships are *include* and *extend*. These are shown through the use of dotted or dashed lines with an open arrow head connecting two use cases; they are not used to connect actors. These relationships must be labelled to indicate whether they are an inclusion or an extension.

When **include** is used, it indicates that the entirety of one use case is included in another. This means that the process of completing a use case always requires running the functions in the other use case at least once. For example, in Figure 3.22, the use case shows that 'edit user' includes 'load user'. This is because to edit a user, that user must first be loaded. Similarly, 'update user details' also includes 'load user', as that functionality also requires that a user be loaded first. Notice that the arrow head on the *include* relationship is pointing towards the use case that is included in the other use case. The *include* relationship must always be labelled using two angled brackets on each side of the word 'include', as shown in Figure 3.22.



**FIGURE 3.22** A use case using 'include'

When **extend** is used, this indicates that one use case can sometimes be included in another. It is often used to represent additional or optional functionality within the system. For example, in Figure 3.23, the ‘move image’ use case is an extension of the ‘select image’ use case, as moving the image is an action a user can choose to complete after selecting an image. Similarly, in Figure 3.24, the ‘display help’ use case is an extension of the ‘register user’ use case, meaning that users can choose to display help relevant to registration, but do not necessarily need to in order to complete the registration process. Notice that, in both examples, the arrow head on the extend relationship is pointing towards the use case that is being extended upon. A use case should not be an extension of another use case if it is required as part of the functionality of that use case; in these cases, the relationship should be an ‘include’. The *extend* relationship must always be labelled using two angled brackets on each side of the word ‘extend’, as shown in Figures 3.23 and 3.24.



**FIGURE 3.23** A use case for an image system using ‘extend’

**FIGURE 3.24** A use case for a registration system using ‘extend’

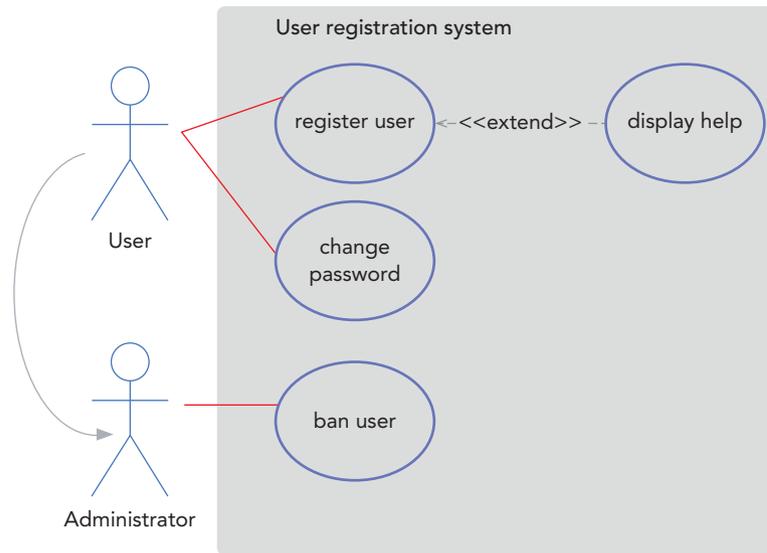
## System boundary

Use case diagrams show the use cases contained within a **system boundary**. This makes it clear what is included in the system being built and what is not. System boundaries are shown by drawing a rectangle around the use cases that are relevant to the proposed system. Actors should remain outside of the system boundary box. Although there is typically only one system boundary, in cases where scope has been reduced, system boundary boxes can be nested to indicate which use cases are within the current scope of the project, and which have been deemed out of scope. This shows which use cases are part of the scope of an SRS and which may be included in later versions of the software.

## Drawing use case diagrams

For consistency, use cases are usually ordered so that all of the main use cases are in the centre of the diagram, typically running vertically, from top to bottom. The order of the use cases does not normally matter, but it can be helpful if similar actions are grouped together. Use cases that are extensions or inclusions of other use cases are typically placed to the right of the use cases they have a relationship with; this often creates two vertical rows of use cases, as shown in Figure 3.25.

Actors are shown on each side of the use cases, and can be repeated (i.e. one on each side) to avoid overlapping relationships; this adds clarity and makes the diagram more readable. System boundaries must be labelled, typically in the top left corner or bottom right corner of the boundary. Use cases should be labelled within the ellipse, and actors should have labels underneath the stick figure.



**FIGURE 3.25** A use case diagram for a user registration system

## Context diagrams

A **context diagram**, sometimes referred to as a Level 0 data flow diagram, is a visualisation of a system in its entirety that indicates the data that is passed into and out of the system. Context diagrams are often created after use case diagrams have been drawn, as all of the interactions a user can have with the system have been documented in the use cases. This allows all of the actions a user can complete to be systematically translated to the data that is needed for those actions to succeed.

Context diagrams do not typically show much detail. They are only intended to focus on the flow of data in and out of the proposed system. This helps establish the context and the boundaries of the software system being created.

There are three primary symbols used in context diagrams, which represent processes, entities and data flows.



**FIGURE 3.26**  
A context diagram process

### Process

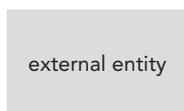
A **process** in a context diagram is an abstract representation of the whole system being created. Unless the proposed system is very complex, a context diagram will typically contain only a single process. This is shown on the diagram using a circle, with the name of the system contained within that circle.

In a context diagram, the system process is connected to external entities by the use of data flows.

### Entity

A context diagram may contain one or more **entities**, which are the users or external systems that interact with the system being created. These are drawn as rectangles within the context diagram, with the name of the entity contained within the rectangle. Much like actors in use case diagrams, entities should not be labelled with the names of real people but instead with the abstract role those people have.

In a context diagram, entities can only interact with the system process and cannot interact with each other.

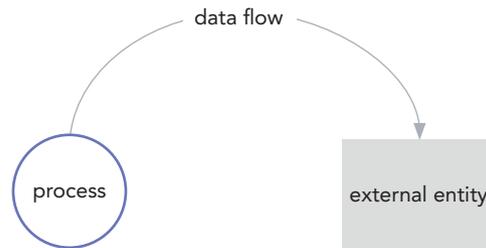


**FIGURE 3.27**  
A context diagram entity

## Data flow

A **data flow** represents a single piece or logical collection of data as it moves into and out of the system being represented in the context diagram. These are represented by solid lines, typically curved, with an open arrow head representing the direction of the data flow. While most data flows are unidirectional, there are instances where data flows in both directions. In these instances, both ends of the data flow should show open arrow heads.

In a context diagram, data flows connect entities to the system process. They must always start or end at a process, and cannot directly connect entities to other entities.

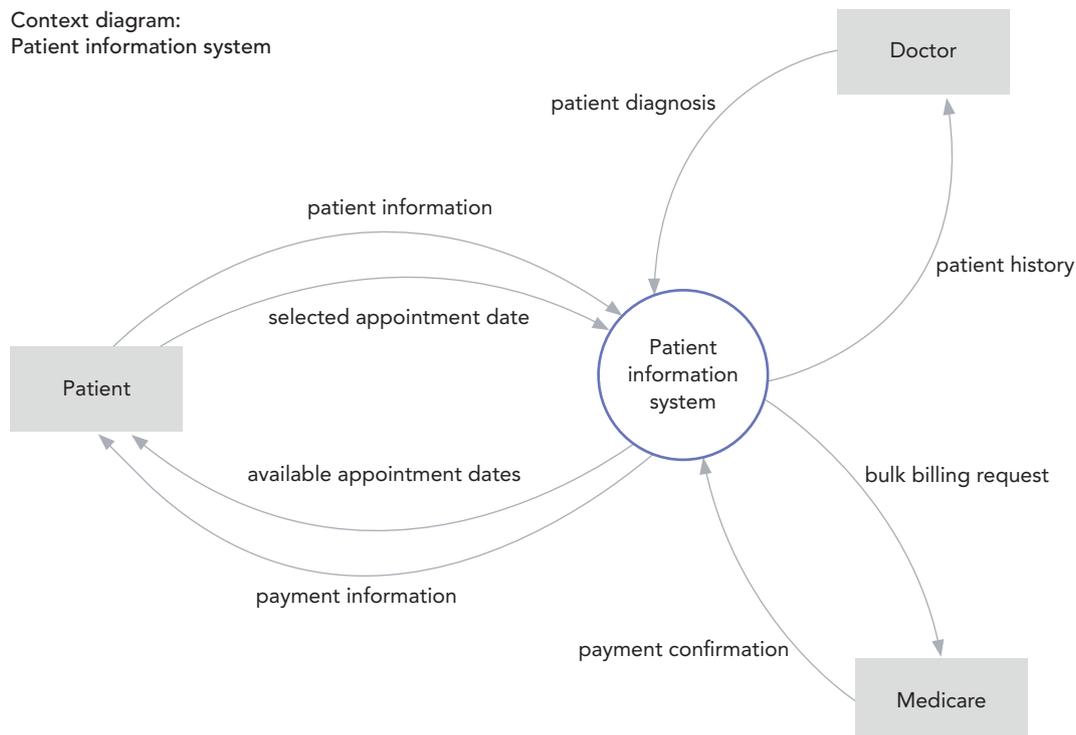


**FIGURE 3.28** A context diagram with a process, entity and data flow

## Drawing context diagrams

There are two main notation styles used to represent context diagrams: Gane-Sarson and Yourdon-DeMarco. In this text, the Yourdon-DeMarco style is used.

Context diagram:  
Patient information system



**FIGURE 3.29** A context diagram for a patient information system

When drawing context diagrams, the single process referring to the proposed system should be displayed in the centre of the diagram, with entities appearing on each side so that they are as balanced as possible. Each data flow should be represented as lines, normally curved, and must clearly indicate the direction the data is moving in. Labels on data flows that indicate which data is transferred should appear close enough to the line that it is clear which data flow the label is attached to.

## Data flow diagrams

A **data flow diagram (DFD)** is a graphical visualisation of the flow of information within a system, including data provided by external entities. DFDs provide more information than context diagrams, and are designed to show the data flowing into and out of every function within the system. For this reason, DFDs are normally drawn with consideration to ‘levels’, where each level of the diagram contains more detailed information than the previous level. Context diagrams are therefore often referred to as ‘Level 0 data flow diagrams’, as they contain the least amount of information about the data flowing into and out of the system.

Level 1 DFD diagrams contain the core processes within the system. There may be more than one of this level of diagram, depending on system complexity. The general rule is that a DFD should contain at least three but no more than seven to nine processes; if there are more, they should be separated into related collections of processes across multiple DFDs. Level 2 DFDs (and higher) contain a more detailed look at the inner workings of the processes that were included in Level 1 DFDs. These are used when processes are quite complex, and more detail is needed to show what the processes are doing. If two or more levels of DFDs are created, the data that flows into and out of Level 1 DFDs must be the same; the only change in the diagrams is the level of detail shown for selected processes. The decision to proceed beyond Level 1 for DFDs can be separately made for each process, with the developer deciding when the level of detail is sufficient.

Much like context diagrams, DFDs contain processes, entities and data flows and they also show data stores, which represent where data is coming from.

### Process

A **process** in a DFD is not the same as a process in a context diagram. While a context diagram process represents the whole system, a process within a DFD represents a whole function. For this reason, DFDs typically include at least as many processes as there are use cases in the system’s use case diagram.

Processes in DFDs are shown using a circle, with the name of the process contained within that circle. Process names are typically short and begin with a strong verb and a singular noun, such as ‘validate PIN’ or ‘print receipt’, and normally represent the functions of the proposed system. Much like use cases, the name of a process should not be longer than two or three words.

In a DFD, processes can be connected to external entities, other processes and data stores. These are all connected via labelled data flows.

A process must have at least one input data flow and one output data flow. Processes must also transform data – the data going in should not be exactly the same as the data coming out.

## Entity

A DFD may contain one or more entities, which, as in context diagrams, are the users or external systems that interact with the system being created. These are drawn as rectangles within the DFD, with the name of the entity contained within the rectangle. Much like actors in use case diagrams, entities should not be labelled with the names of real people, but instead with the abstract role those people have.

In a DFD, entities can only be connected to processes via a data flow. They cannot directly interact with data stores or other entities.

## Data flow

A data flow represents a single piece or logical collection of data as it moves between entities, processes and data stores within the system. Much like in a context diagram, data flows in a DFD are represented by solid lines, typically curved, with an open arrow head representing the direction of the data flow. While most data flows are unidirectional, there are instances where data flows in both directions. In these cases, both ends of the data flow should show open arrow heads.

In a DFD, data flows can connect entities to processes, processes to other processes, and processes to data stores. They must always be coming from or going to a process and cannot directly connect data stores with each other.

## Data store

A **data store** represents a collection of data that is stored in some way. Some examples of data storage include a database, a plain text file and an XML file. Data that flows out of a data store indicates that it is retrieved from it, whereas data flowing into the data store is assumed to be updating or adding to it.

In a DFD, data stores are represented by two parallel lines, with the name of the data store in between them. Data stores can only be connected to processes. Each data store should have at least one input and one output data flow.

## Drawing data flow diagrams

Much like context diagrams, there are two main notation styles used to represent DFDs: Gane-Sarson and Yourdon-DeMarco. In this text, the Yourdon-DeMarco style is used.

DFDs should be drawn to reduce the amount of data flow overlaps. DFDs do not need to show entities if they have already been shown in a preceding level DFD, although they can still be included for clarity.

When creating a DFD, the following questions should be asked:

- Where does the data come from? Does it come in as input from an entity, or does it come from a data store?
- What happens to the data once it enters the system? Which process(es) does it flow into? Which data stores does it flow into?
- Where does data go once it enters a process? Is it stored in a data store? Is it returned to an entity as output?

The first step in constructing a DFD is to identify the processes that are needed to perform the work within the system. These are often closely related to the use cases that were included in the use case diagrams constructed in the analysis stage. At a minimum, each use case should be represented in a DFD as a process. Some use cases will require more than one process.



Data store

**FIGURE 3.30** A data store

Once the processes have been identified, they should first be connected to the relevant entities that provide the data that flows into the process. They should then be connected to other related processes and data stores, with each data flow appropriately labelled. Finally, any outputs from the processes should be included. In general, a process requires at least one data flow as input and at least one as output; if it has less than this, it is not a valid process.

Once all processes and DFDs have been created at Level 1, a decision can be made about whether any process within those diagrams requires further detail. If this is the case, a Level 2 DFD should be created to reflect that detail.

The layout of a DFD follows similar conventions to a context diagram, but as there are more processes involved, it is sometimes difficult to keep them all centred. The general rule is that processes are grouped in the middle of the DFD. Data flows should, as much as possible, be separated so they do not overlap other data flows, and data stores should be beneath the processes that use them.

Once a DFD is complete, it should be checked for accuracy. Verifying that the DFD is accurate involves checking each process, data flow, data store and entity.

Each *process* should be checked to make sure it has a unique name, and that its name is a strong verb phrase and is sufficiently descriptive. Each process must also have at least one input data flow and one output data flow. Output data flows should have different pieces or collections of data from the input data flows. The DFD should not be overcrowded with processes.

Each *data flow* should be checked to make sure it is named after a piece of data or collection of data; they should be noun phrases. Each data flow should connect to at least one process. They should have a minimum amount of overlap with other data flows; it is preferred that there is no overlap.

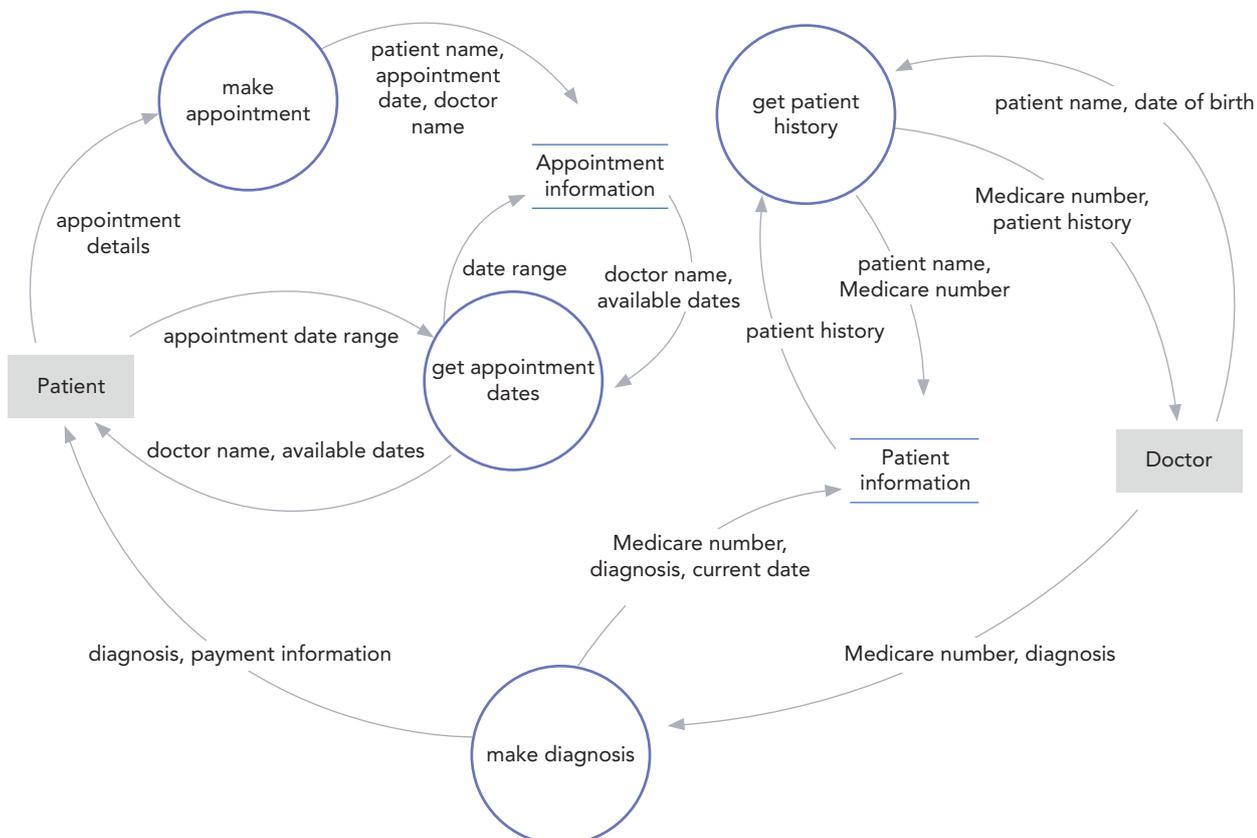


FIGURE 3.31 DFD Level 1 for a patient information system

Each *data store* should be checked to make sure it has a unique name and that it is named appropriately to the data it represents; the names should be noun phrases. Each data store should have at least one data flow connecting it to a process.

Each external *entity*, if shown, should match the name of an entity in the context diagram. The name of the entity should be unique and be a noun or noun phrase. Each entity should be connected to at least one process by at least one data flow.

## Software to track music performances

Susan is a music director who is in charge of running music performances. She would like to replace her current manual system of tracking music performances with a software system so she can track concerts and performances without needing to rely on written notes. She would like the system to run on her computer at home as well as on her mobile phone. Susan is a relatively inexperienced technology user, but needs to be able to learn to use the software quickly; her preference is that it will require no more than an hour of training.

The system Susan needs must enable her to add performances and concerts. If she is adding a concert, she would also like to be able to assign a theme to it. She wants concert themes to come from a saved list that she can add to as required. Susan also needs the software to track the themes of each concert that she runs. She runs these twice a year, and must not use the same theme in any given six-year period; most people will not pay to see a concert again if the theme is the same. She tends to run her concerts in different locations on a six-year rotation.

Susan also needs the ability to edit concerts, performances and themes. In the future she would like the ability to upload recordings of live performances and concerts to the system.

She would like the software to be finished before her next concert in January next year.

Susan has approached a local software developer, Donna, and asked her to create the system for her. Donna began writing a software requirements specification (SRS) that outlined the functional requirements, non-functional requirements, constraints and scope of the system, which are listed below.

The functional requirements are:

- Music performances and concerts can be added (Figure 3.32).
- Music concert themes can only be attached to a concert if no concert has had that theme in the last six years (Figure 3.32).
- Music performances and concerts can be edited (Figure 3.33).
- Concert themes can be added (Figure 3.34).
- Concert themes can be edited (Figure 3.35).

The non-functional requirements are:

- Portability: The software must run on Susan's mobile phone as well as her computer at home.
- Usability: Susan should take less than an hour to learn how to use the software.

The constraints are:

- Technical: Data entry must be supported via keyboard as well as touchscreen.
- Usability: Susan has very little experience using technology.
- Economic: Susan wants the software finished before January next year.

System components that are in scope:

- the ability to add and edit music performances and music concerts
- the ability to add and edit concert themes.

System components that are out of scope:

- the ability to delete music performances, concerts or themes
- the ability to upload live recordings of performances and concerts.

## CASE STUDY





### 3.1.1. Add Performance

This screen will allow the entering of performance information. Information consists of the performance name, date and theme, as well as a unique performance ID (numeric). Also recorded is whether the performance is a concert, and, if so, the theme of that concert.

#### Inputs:

- Performance ID (unique)
- Performance Name, Performance Date
- Whether it is a concert or not
- Concert Theme

#### Sequence of Operations:

- Input will come in from the User Interface, all items except concert theme are required fields
- Themes should be retrieved from a file containing all available themes so a user can select one
- No two concerts should have the same ID
- A theme only needs to be selected if it is a concert
- No concert should be added if a concert with the same theme has occurred in the last six years
- Once data is entered and validated, it should be saved to a performance file

#### Outputs:

A pop-up box should be displayed showing the success or failure of saving the concert information. Once information is saved, the user should be returned to the main menu.

**FIGURE 3.32** An 'add performance' functional requirement from Susan's music performance system

### 3.1.2. Edit Performance

Given a list of all performances, allow a user to change the information of any particular performance.

#### Inputs:

- Performance ID

#### Sequence of Operations:

- The user should be able to select a performance from a list of performances that have been saved to file.
- From this, they should be taken to a screen that looks like **3.1.1. Add Performance**, with the information pre-filled into the screen.
- They should be able to change the performance name, date and theme and whether it is a concert or not, but not the performance ID.
- These changes should be saved to the performance file once the save button is pressed.

#### Outputs:

A pop-up box should be displayed showing success or failure of saving the updated performance information. Once information is saved, the user should be returned to the main menu.

**FIGURE 3.33** An 'edit performance' functional requirement from Susan's music performance system

### 3.1.3. Add Theme

This screen will allow the entering of theme information. Information consists of the theme name as well as a unique theme ID (numeric).

#### Inputs:

- Theme ID (unique)
- Theme Name

#### Sequence of Operations:

- Input will come in from the User Interface, all items are required fields.
- No two themes should have the same ID, so this should be validated before saving.
- Once data is entered and validated, it should be saved to a theme file.

#### Outputs:

A pop-up box should be displayed showing the success or failure of saving the theme information. Once information is saved, the user should be returned to the main menu.

**FIGURE 3.34** An 'add theme' functional requirement from Susan's music performance system

### 3.1.4. Edit Theme

Given a list of all themes, allow a user to change the information of any particular theme.

#### Inputs:

- Theme name

#### Sequence of Operations:

- The user should be able to select a theme from a list of themes that have been saved to file.
- From this, they should be taken to a screen that looks like **3.1.3. Add Theme**, with the information pre-filled into the screen.
- They should be able to change the theme name but not the theme ID.
- These changes should be saved to the theme file once the save button is pressed.

#### Outputs:

A pop-up box should be displayed showing the success or failure of saving the changed theme information. Once information is saved, the user should be returned to the main menu.

**FIGURE 3.35** An 'edit theme' functional requirement from Susan's music performance system

Using the information already contained within the SRS, Donna must complete the SRS by creating a use case diagram, a context diagram and relevant data flow diagrams to represent the system.

## Step 1: Creating a use case diagram

### Determine the actors

As Susan is the only person who will use this system, there is only one actor. Susan's role in relation to the system is *Music Director*.





### Determine the use cases

To create a use case diagram, Donna looks at the functional requirements of Susan's system:

- Music performances and concerts can be added (Figure 3.32).
- Music concert themes can only be attached to a concert if no concert has had that theme in the last six years (Figure 3.32).
- Music performances and concerts can be edited (Figure 3.33).
- Concert themes can be added (Figure 3.34).
- Concert themes can be edited (Figure 3.35).

Most of these functional requirements can be represented as a use case. These need to have short names that begin with strong verbs:

- Add performance
- Add concert
- Edit performance
- Edit concert
- Add theme
- Select theme
- Edit theme

After writing out the list of potential use cases, Donna realises that concerts are just a particular type of performance, the only difference being that a concert requires that a theme be selected, while a performance does not. She revises her list of use cases as a result:

- Add performance
- Edit performance
- Select theme
- Add theme
- Edit theme

### Determine the relationships

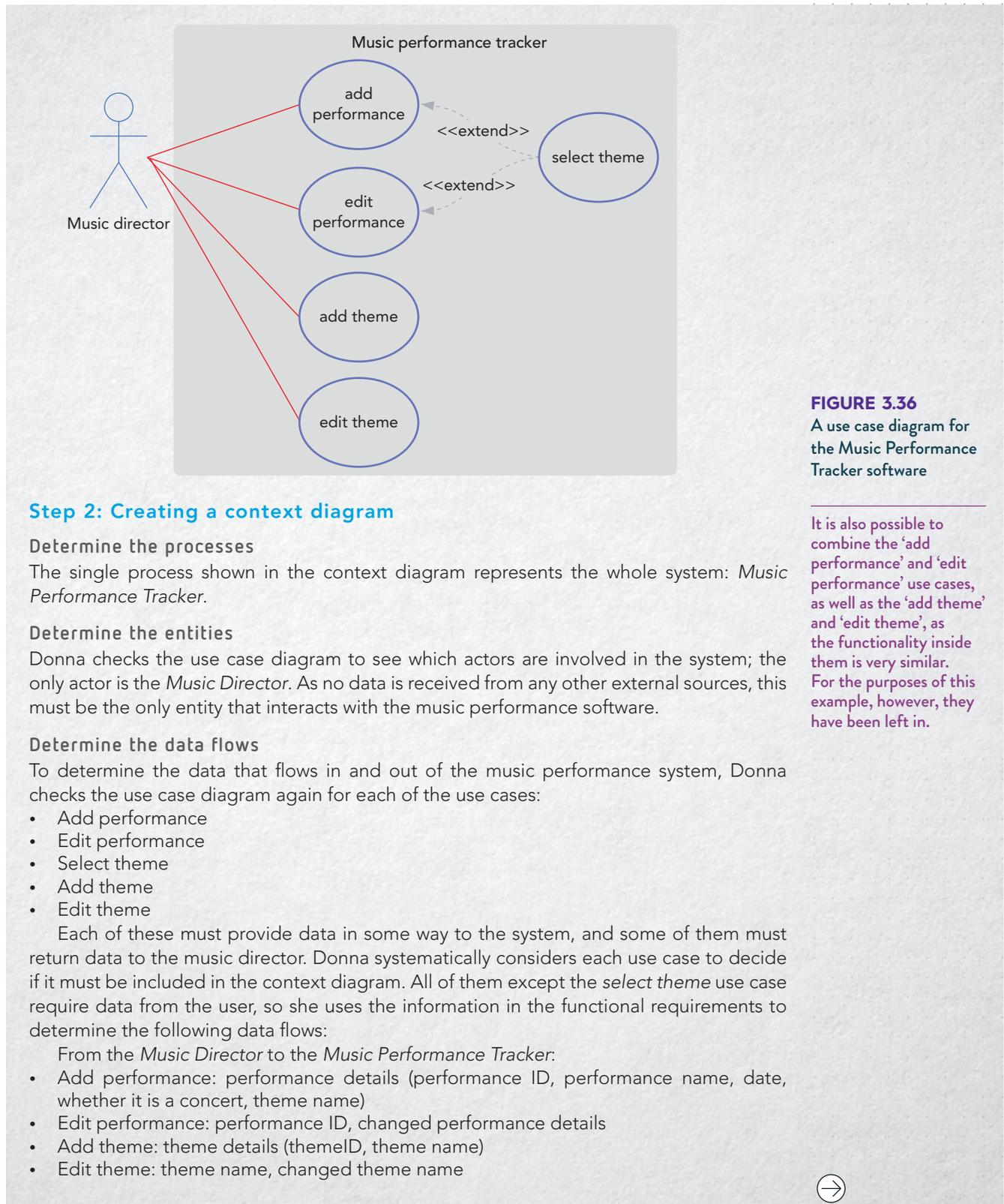
In this scenario, concerts are special types of performances, and not all performances are concerts. This suggests that the relationship between a performance and a concert is an *extension*, where a concert is an extension of a performance. The difference between the two is that a concert requires that a theme is selected, while a performance does not. Therefore, the *select theme* use case has an *extend* relationship with the *add performance* use case.

### Determine the system boundary

All of the listed use cases are within scope of the functional requirements of the new software. This means the system boundary will include all use cases. Donna could also choose to include the functional requirements that are out of scope in the use case diagram, such as the ability to upload live recordings, but this is optional.

### Draw the use case diagram

Given all of the information she now has about the actors, use cases, relationships and system boundary, Donna is able to draw the use case diagram, as shown in Figure 3.36.

**FIGURE 3.36**

A use case diagram for the Music Performance Tracker software

## Step 2: Creating a context diagram

### Determine the processes

The single process shown in the context diagram represents the whole system: *Music Performance Tracker*.

### Determine the entities

Donna checks the use case diagram to see which actors are involved in the system; the only actor is the *Music Director*. As no data is received from any other external sources, this must be the only entity that interacts with the music performance software.

### Determine the data flows

To determine the data that flows in and out of the music performance system, Donna checks the use case diagram again for each of the use cases:

- Add performance
- Edit performance
- Select theme
- Add theme
- Edit theme

Each of these must provide data in some way to the system, and some of them must return data to the music director. Donna systematically considers each use case to decide if it must be included in the context diagram. All of them except the *select theme* use case require data from the user, so she uses the information in the functional requirements to determine the following data flows:

From the *Music Director* to the *Music Performance Tracker*:

- Add performance: performance details (performance ID, performance name, date, whether it is a concert, theme name)
- Edit performance: performance ID, changed performance details
- Add theme: theme details (themeID, theme name)
- Edit theme: theme name, changed theme name



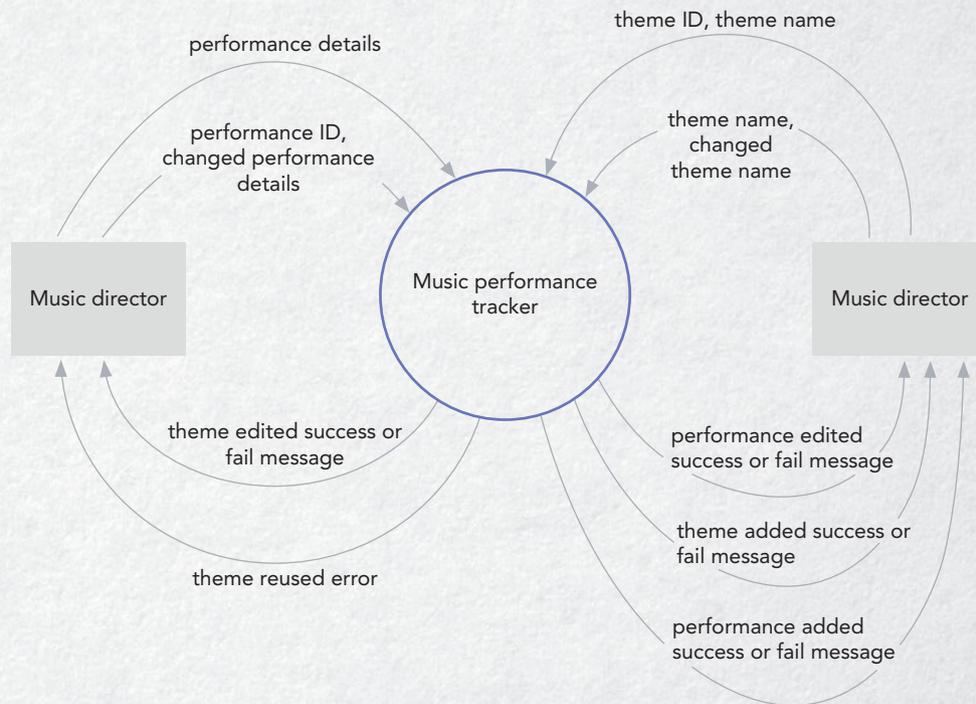


From the *Music Performance Tracker* to the *Music Director*:

- Add performance: Theme reused error
- Add performance: Performance added success or fail message
- Edit performance: Theme reused error
- Edit performance: Performance edited success or fail message
- Add theme: Theme added success or fail message
- Edit theme: Theme edited success or fail message

#### Draw the context diagram

Given all of the information she now has about the process, entities and data flows, Donna is able to draw the context diagram, as shown in Figure 3.37.



**FIGURE 3.37**

A context diagram for the Music Performance Tracker software

### Step 3: Creating data flow diagrams

#### Determine the processes

Donna uses the use case diagram again to determine the processes that must exist in her data flow diagram. Each of the use cases must have a corresponding process in the data flow diagram:

- Add performance
- Edit performance
- Select theme
- Add theme
- Edit theme

#### Determine the entities

Donna checks the context diagram and only a single entity is needed: *Music Director*.

### Determine the data stores

Before determining which data flows are needed, Donna must consider where data will be stored. Performances, including concerts, can be stored in a single data store. Themes should be stored in a different data store, as they are a different collection of data to the performance data.

Donna therefore decides that there will be two data stores: *Performances* and *Themes*.

### Determine the data flows

Each of the processes must now be considered in relation to the data they receive from external entities, the data they pass to other processes, and the data they store and retrieve from the data stores.

Donna checks the context diagram to look at each of the data flows she represented going to the system. This takes quite a while, as she must make sure she shows the movement of every piece of data to and from processes, data stores and entities.

**TABLE 3.2** Data flows going into and out of the music performance system

|    | From             | To                      | Data                                       | Reason   |
|----|------------------|-------------------------|--|--|
| 1  | Add performance  | Music Director          | Performance added success or fail message  | Output to the Music Director when adding a performance                         |
| 2  | Add performance  | Music Director          | Theme reused error                         | Output to the Music Director if reusing a theme before 6 years is up           |
| 3  | Add performance  | Performances data store | Performance details                        | Sending the performance data to be saved                                       |
| 4  | Add performance  | Select theme            | Theme name                                 | Sending the theme name to get a valid theme ID                                 |
| 5  | Add theme        | Music Director          | Theme added success or fail message        | Output to the Music Director when adding a theme                               |
| 6  | Add theme        | Themes data store       | Theme ID, theme name                       | Sending a new theme and theme ID to the data store to be saved                 |
| 7  | Edit performance | Music Director          | Performance edited success or fail message | Output to the Music Director when editing a performance                        |
| 8  | Edit performance | Music Director          | Theme reused error                         | Output to the Music Director if the changed theme was used in the last 6 years |
| 9  | Edit performance | Performances data store | Changed performance details                | Sending the changed performance details to be saved                            |
| 10 | Edit performance | Performances data store | Performance ID                             | To retrieve the performance details when a performance is selected             |

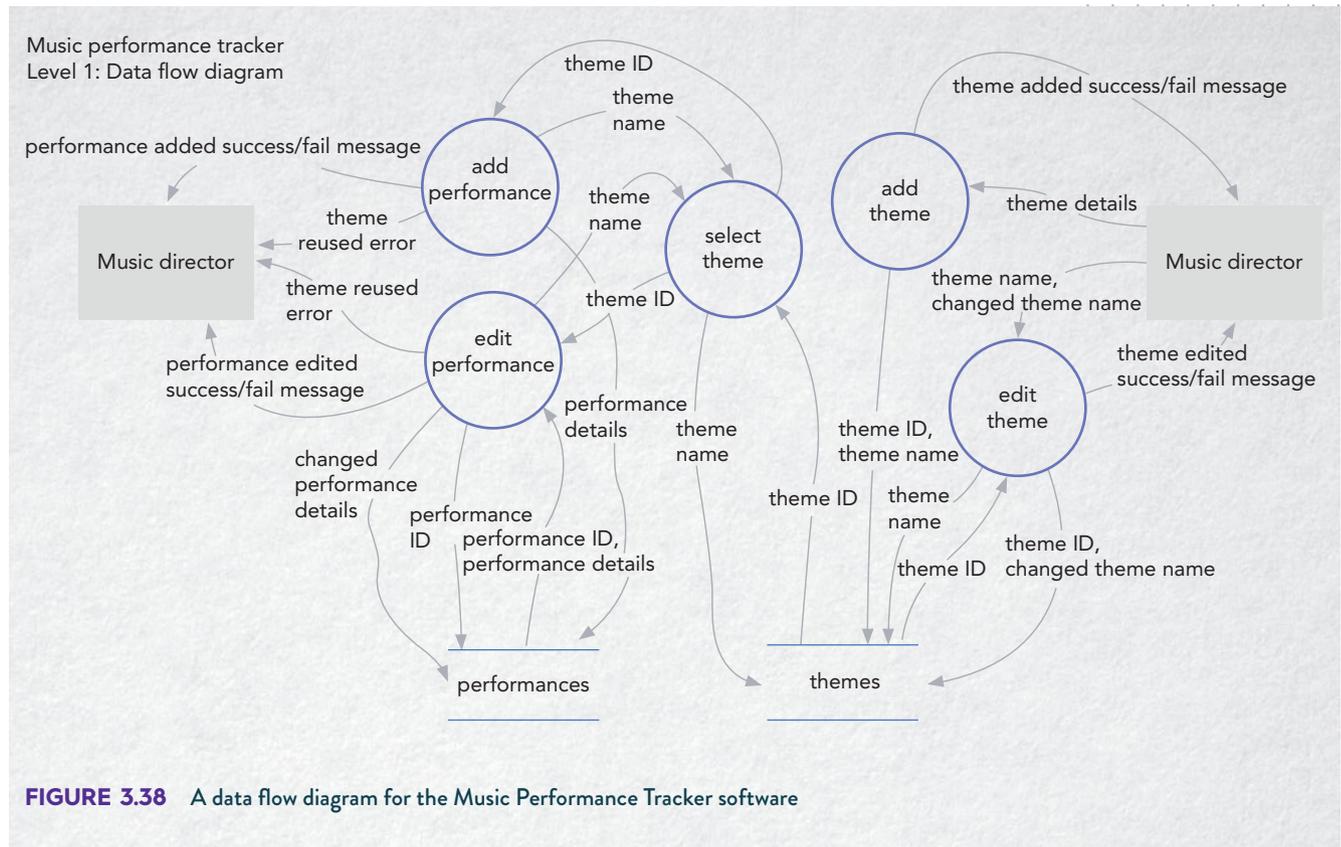


**TABLE 3.2** Data flows going into and out of the music performance system (continued)

|    | From                    | To                | Data  | Reason   |
|----|-------------------------|-------------------|---|--|
| 11 | Edit performance        | Select theme      | Theme name                                  | Sending the theme name to get a valid theme ID                               |
| 12 | Edit theme              | Music Director    | Theme edited success or fail message        | Output to the Music Director when editing a theme                            |
| 13 | Edit theme              | Themes data store | Theme ID, changed theme name                | Sending the changed theme details to be saved                                |
| 14 | Edit theme              | Themes data store | Theme name                                  | To retrieve the theme ID when a theme is selected to be edited               |
| 15 | Music Director          | Add performance   | Performance details                         | Providing performance details to be saved                                    |
| 16 | Music Director          | Add theme         | Theme details                               | Providing theme details to be saved  |
| 17 | Music Director          | Edit performance  | Performance ID, changed performance details | Providing changed performance details to be saved                            |
| 18 | Music Director          | Edit theme        | Theme name, changed theme name              | Providing changed theme details to be saved                                  |
| 19 | Performances data store | Edit performance  | Performance ID, performance details         | Checking to see if a performance exists and returning its details if it does |
| 20 | Select theme            | Themes data store | Theme name                                  | Sending the theme name to get a valid theme ID                               |
| 21 | Themes data store       | Edit theme        | Theme ID, theme name                        | Providing theme details to be edited   |
| 22 | Themes data store       | Select theme      | Theme ID                                    | Providing a theme ID for a given theme name                                  |

#### Step 4: Drawing the data flow diagram(s)

Donna decides that she will only use one data flow diagram, as there are only five processes. Given all of the information she now has about the process, entities, data stores and data flows, she draws this data flow diagram, as shown in Figure 3.38



**FIGURE 3.38** A data flow diagram for the Music Performance Tracker software

## Next steps

In this chapter, we have discussed the features of project management, and the need for regular monitoring and adjustment. You will have gained enough information to make a decision as to how you are going to track changes to your project plan now, before you progress through the Unit 3 and 4 components of the SAT project.

We also discussed the data collection methods available to analyse client requirements, scope of the solution, functional and non-functional requirements and the various constraints that can impact on a project.

Your next step, upon completion of the chapter summary, is to work towards completion and submission of the solution analysis for Unit 3, Outcome 2, according to your teacher's instructions.

As you collect your data, take steps to protect your respondents and subjects. You should begin to think about relevant constraints, the scope of the specification and appropriate analytical diagrams to represent data flow and users. Chapter 4 begins with a discussion of the software solution design requirements for Unit 3, Outcome 2.

# 3

## CHAPTER SUMMARY

### Essential terms

**actor** an entity that can interact with the software solution as shown in a use case diagram

**analysis stage** the stage of the problem-solving methodology where solution requirements, constraints and scope are determined

**association** a relationship between two elements in a use case diagram

**clarity** the extent to which a product is coherent and intelligible

**close-ended questions** questions that can be answered with a finite set of responses

**concepts (project management)** the milestones and dependencies within a project timeline

**concurrently** when a task is carried out at the same time as another task

**constraints** factors that may limit or restrict solution requirements

**context diagram** a visualisation of a system in its entirety that indicates the data that is passed into and out of the system

**critical path** the shortest possible time in which a project can be completed

**data** raw, unprocessed facts and figures

**data flow** the movement of a piece or collection of data within an information system, as shown in context diagrams and data flow diagrams (DFDs)

**data flow diagram (DFD)** a graphical visualisation of the flow of information within a system, including data provided by external entities

**data store** a representation of a collection of data that is stored in some way within a system

**design stage** the stage of the problem-solving methodology where the function and appearance of a solution are planned, and evaluation criteria created

**economic constraints** the limitations on a project or decision imposed by financial factors

**entity** the users or external systems that interact with the system being created

**event** a special type of method that is called when an object's state changes

**extend** a relationship between use cases where one use case has optional or additional functionality, which is represented in a use case diagram as a second use case

**fit for purpose** to be well suited for a role or purpose

**functional requirements** the desired operations of a program that have specified inputs, behaviours and outputs

**functionality** the extent to which a solution is suited to its purpose

**Gantt chart** a type of bar chart or graphic timeline that shows the progress of a project by placing tasks on a timeline, often with comments or annotations

**generalisation** a parent-child relationship between two elements in a use case diagram

**include** a relationship between use cases where one use case is tied to, or relies upon, the functionality contained within another use case

**interview** a face-to-face meeting between people for consultative purposes

**legal constraints** the limitations and requirements imposed on a project or decision by laws, regulations and legal standards

**maintainability** how easy a solution is to look after once it has been put in place

**non-functional requirements** qualitative requirements of a solution, often tied to solution constraints

- observation** a method of data collection that involves physically observing how a system operates and how it is used
- open-ended questions** questions where the number of potential answers is infinite
- portability** how easily a solution is able to be used in different operating environments
- predecessor** a task that must be completed before another one can be performed
- problem-solving methodology (PSM)** an approach that develops the stages involved in solving a problem
- process (context diagram)** an abstract representation of the whole system being created
- process (data flow diagram)** an abstract representation of a function within a system
- processes (project management)** task identification, sequencing and allocation of time and resources within a project timeline
- project management** a method of recording the progress of a project and managing resources to operate within time, resource and cost availability
- qualitative data** data that consists of descriptive details, usually gathered via surveys or interviews
- quantitative data** data that can be easily processed in a statistical manner, usually composed of definite numbers
- relationship** the connections between elements within a use case diagram
- reliability** how much a solution can be depended upon to function as designed, and for how long
- report** a written document providing a summary or finding in relation to the context or system being analysed
- robustness** how well a software solution responds to errors that occur when the software is being used
- slack time** the length of time that a task can run overtime before affecting other tasks
- social constraints** the limitations imposed on a project or decision by societal norms, values and expectations
- software** programs used by a computer
- software developer** a human who participates in design and creation of software programs, typically by writing programming code
- software requirements specification (SRS)** a single document that contains the outcomes of the analysis stage of the problem-solving methodology, including scope, constraints, functional requirements and non-functional requirements
- solution boundaries** the limits or edges of what a project or solution will encompass
- successor** a task that must be completed after another task
- survey** a set of questions that ask for a response to be selected from a list of alternatives
- system boundary** a rectangle around relevant use cases that indicates the use cases that are within the scope of the solution
- technical constraints** the limitations and restrictions related to the technology used in a project
- Unified Modeling Language (UML)** a general-purpose visual modelling language
- uptime** the time during which a machine, solution or application is operational
- usability** the extent to which a system is easy to learn and use
- use case** a representation of the transactions or functions a user (actor) can complete in a system, as shown in a use case diagram
- use case diagram (UCD)** a method of describing how a user interacts with a system, using Unified Modeling Language (UML)
- version** a specific state or release of a software product, used to track and manage changes, improvements and updates made to the software over time
- work breakdown structure (WBS)** an often hierarchical breakdown of a project that organises the work to be done into manageable sections, often displayed as a visual outline or map

## Important facts

- 1** Developing software can help to increase productivity and efficiency, reduce costs, identify opportunities to address gaps in the market and meet organisational objectives and needs.
- 2** Increasing productivity can be achieved through automating tasks, streamlining complex processes and facilitating easier access to information.
- 3** Reducing costs can be achieved through reducing the need for manual labour and implementing automation processes.
- 4** Identifying opportunities can allow an organisation or individual to capitalise on unmet needs or gaps in the market.
- 5** Specific organisational goals or needs can be addressed through software development projects.
- 6** Project management is the practice of applying techniques, processes, tools, knowledge and skills to deliver a solution. Features of project management include identification of tasks, sequencing, time allocation, milestones, dependencies and critical path.
- 7** Critical path indicates the shortest time possible to complete the project.
- 8** If there is any change on the critical path, the timing of the entire project is affected.
- 9** Data collection involves surveys, interviews, reports and observations.
- 10** A survey is a set of questions that ask for a response from a user.
- 11** Close-ended questions in surveys are where the number of responses is finite.
- 12** Open-ended questions in surveys are where the number of responses is infinite.
- 13** Quantitative data contains information that is easily collated, such as values, numbers or counts; typically, this involves numeric variables.
- 14** Qualitative data contains information that is not easily measured, such as opinions and qualities.
- 15** Surveys are relatively inexpensive, and results can be immediately collected.
- 16** Processing survey results can take time if open-ended questions are used.
- 17** People are not always truthful when answering surveys.
- 18** Interviews are usually conducted face-to-face, and can be one-on-one or in groups.
- 19** Interviews can take a lot of time to complete, but allow in-depth data to be collected.
- 20** Interviews are very useful for eliciting feelings, attitudes, judgements and opinions that are too complex to easily record in a survey.
- 21** Reports are written documents providing summaries or findings in relation to a system being analysed.
- 22** Reports include error reports, customer complaint summaries, uptime reports and system performance reports.
- 23** The types of reports collected will vary depending on what is most relevant for the proposed system.
- 24** Reports are useful as they are often pre-prepared, which can save time and money.
- 25** A disadvantage of reports is that the data may have been manipulated to present a particular point of view.
- 26** Using data from reports can be risky if the source of those reports is not reliable.
- 27** Observations involve physically observing how a system operates and is used.
- 28** Observations are considered unbiased, as information can be gathered without asking for an opinion or judgement.
- 29** Observations can be time-consuming, and they may not always provide the information needed due to being performed at an inopportune time.
- 30** Functional requirements are part of the solution required and directly relate to what a solution will do.
- 31** Functional requirements are typically described in terms of required inputs, sequence of operations and expected outputs.
- 32** Non-functional requirements are qualities and aspects of the solution that are desired but do not affect what the solution does.

- 33** Non-functional requirements are often tied to the constraints of the system.
- 34** Non-functional requirements are described in terms of usability, reliability, portability, robustness and maintainability.
- 35** Determining non-functional requirements most often involves client interviews and observations.
- 36** Usability relates to how easy a system is to learn and use.
- 37** Reliability relates to how much, and for how long, a system can be depended upon to function as designed.
- 38** Portability relates to how easily software can be used in different operating environments.
- 39** Robustness relates to how well a software solution responds to errors that occur while the software is being used.
- 40** Maintainability relates to how easy the software is to look after once it is put in place.
- 41** A software requirements specification (SRS) is a single document that outlines all of the elements considered in the analysis stage: constraints, scope, functional requirements and non-functional requirements.
- 42** An SRS provides all of the required information about the proposed system needed in order to design the system.
- 43** An SRS should contain a cover page, table of contents, numbered sections, headings and subheadings, page numbers and relevant appendices.
- 44** Three methods of depicting interfaces between solutions are use case diagrams, context diagrams and data flow diagrams.
- 45** A use case diagram visually describes how a user interacts with a system.
- 46** Use cases include actors who have relationships with use cases.
- 47** An actor represents an entity that can interact with the functionality within software.
- 48** Actors should be described in terms of roles rather than real people.
- 49** Actors are represented as stick figures in use case diagrams.
- 50** Use cases describe transactions or functions an actor can complete on a system.
- 51** Use cases are drawn using an ellipse with the name of the use case written inside.
- 52** Use cases should be described using strong verbs to describe the action or function being represented.
- 53** Relationships represent the connections between elements in a use case diagram.
- 54** Relationships can exist between actors and use cases, actors and other actors, and between two use cases.
- 55** Associations are the standard form of representing a relationship.
- 56** Associations are represented as solid, straight lines.
- 57** Generalisations are a type of relationship that is considered parent–child.
- 58** Generalisations are represented as solid, straight lines with a closed arrow head pointing from child to parent.
- 59** Include and extend are special types of relationships in a use case diagram.
- 60** The include relationship represents a use case that is entirely included in another use case.
- 61** The extend relationship represents a use case that provides additional and/or optional functionality within a system that is connected to a use case, but not always run.
- 62** Include and extend relationships are represented with dashed or dotted lines and open arrow heads pointing to the relevant use case.
- 63** System boundaries make clear what is included within a system and what is not.
- 64** System boundaries are useful to show the scope of a system.
- 65** Context diagrams provide a visualisation of data that is passed into and out of a system.
- 66** Context diagrams are brief and do not show much detail, focusing only on the flow of data.
- 67** Three primary components of a context diagram are processes, entities and data flows.
- 68** A context diagram process is an abstract representation of the whole system.

- 69** A context diagram entity is a user or external system that interacts with the system being described.
- 70** A context diagram data flow represents a single piece or logical collection of data as it moves into and out of the system represented.
- 71** Two notation styles used to represent context diagrams and data flow diagrams are Gane-Sarson and Yourdon-DeMarco.
- 72** Data flow diagrams (DFDs) are visualisations of the flow of information within a system, including data provided by external entities.
- 73** DFDs provide more information than context diagrams, as they show all of the processes that occur within a system.
- 74** Four primary components of DFDs are processes, entities, data flows and data stores.
- 75** A DFD process represents a function or method within the system.
- 76** A DFD entity represents a user or external system that interacts with the system.
- 77** A DFD data flow represents a single piece or logical collection of data as it moves between entities, processes and data stores within the system.
- 78** A DFD data store represents a collection of data that is stored in some way within the system.



## What is a software solution?

- 1 What are the main purposes for software being developed?
- 2 What must be included in a design brief?

## Project management

- 3 What is meant by project management?
- 4 Identify two consequences of a badly managed project.
- 5 Why is a Gantt chart used?
- 6 Differentiate between concepts and processes in project management.
- 7 Differentiate between predecessors and successors on a Gantt chart.

## Collecting data

- 8 How does a survey differ from an interview?
- 9 When would an observation be a preferred method of collecting data?
- 10 What are three advantages of an interview?
- 11 What are two disadvantages of using reports as part of data collection?

## Functional and non-functional requirements

- 12 How is a functional requirement different from a non-functional requirement?
- 13 Categorise each of these requirements as functional or non-functional:
  - a A report must print to a printer.
  - b All font colours must be green.
  - c A discount must be applied to a product.
  - d The drone must be able to navigate a path through a maze.
  - e The body mass index of a person will be calculated.
  - f A typical six-year-old should understand all of the words displayed.
  - g Button sizes must be big enough for touchscreen capability.
  - h Input can be via voice or keyboard.
- 14 What is the difference between reliability and robustness?
- 15 What does it mean for a software solution to be maintainable?
- 16 What does it mean for a software solution to be usable?
- 17 Explain portability in terms of non-functional requirements.



## TEST YOUR KNOWLEDGE

### Software requirements specifications

- 18 Why are software requirements specifications written?
- 19 What is contained within an SRS?

### Interfaces between solutions, users and networks

- 20 What is the purpose of a use case diagram?
- 21 What is the purpose of a context diagram?
- 22 What is the purpose of a data flow diagram?
- 23 Explain the difference between *include* and *extend* in a use case diagram.
- 24 Explain the difference between an *association* and a *generalisation* in a use case diagram.
- 25 How does a *process* differ between context diagrams and data flow diagrams?
- 26 A proposed food ordering system has two types of users: a chef and a server. The server places food orders given to them by customers. A chef confirms orders and flags them as cooked so a server knows they can take the food to the customer. Once a server has delivered the food to the customer, they remove the order from the queue.
  - a Draw a use case diagram to represent this system.
  - b Draw a context diagram to represent this system.
  - c Draw a data flow diagram to represent this system.



- 1** Brainstorm ideas for a problem, opportunity or need for which you are interested in creating a software solution. You may need to ask people in your family or local community for ideas as part of this process. Consider applications relevant to your hobbies and interests as well. Examples of potential projects are:
  - » job scheduling system for a small business
  - » healthy lifestyle app for a mobile system
  - » fitness apps for wearable technology
  - » order management system for a retail store
  - » automated invoice production
  - » competition ladder tournament creation tool.
- 2** Complete your design brief of your proposed software solution. Ensure that it includes:
  - a** a full description of the problem, opportunity or need, including a client if you have one
  - b** proposed users, and the programming language to be used
  - c** a discussion of the feasibility and originality of the proposed solution.
- 3** Consider the problem, opportunity or need you have selected for your School-assessed Task (SAT).
- 4** Construct a Gantt chart with tasks, milestones and dependencies, including predecessors and successors.
- 5** The nature of your problem will determine the data you need to collect, the functional and non-functional requirements of your solution and the constraints and scope of the solution. Begin drafting an overview of the problem, opportunity or need that most appealed to you from question 1. You should check with your teacher to make sure your selection is feasible and appropriate.
- 6** Complete data collection to determine the constraints of your solution. You may need to consult with a client to gather this information. Ensure that you consider technical, economic, social, legal and usability constraints. Document these constraints. It may be useful to list and describe the constraints in table format.
- 7** Complete data collection to determine the scope of your solution. You may need to consult with a client to gather this information. Ensure that you consider your constraints when determining scope. Document what is in and out of scope. It may be useful to list these as dot points.
- 8** Ensure that you have collected all relevant data for your solution. Make sure you have documented the following:
  - a** Constraints: Ensure that you have considered technical, economic, social and legal constraints.
  - b** Scope: Ensure that you have considered the constraints of your system when deciding what is in scope and what is out of scope.
- 9** In your software requirements specification, include all elements related to the constraints and scope. Expand on any element where required.



## APPLY YOUR KNOWLEDGE

- 10** Collect data in order to determine the functional and non-functional requirements of your chosen system. This will likely involve client and/or user interviews, observations, surveys and collecting reports.
- 11** Add the functional and non-functional requirements to your software requirements specification.
- 12** Using the data you have documented in your SRS in regard to constraints, scope, functional requirements and non-functional requirements:
  - a** create a use case diagram for your proposed software solution
  - b** create a context diagram for your proposed software solution
  - c** create a relevant number of data flow diagrams for your proposed software solution.
- 13** Attach appendices containing any additional information that is required to interpret elements of your SRS.

## 4

# Software development: Software design

## KEY KNOWLEDGE

After completing this chapter, you will be able to demonstrate knowledge of:

- KK3.2.10: key legal requirements relating to the intellectual property and ownership and privacy of data used
- KK3.2.11: file management techniques
- KK3.2.12: ideation techniques and tools for generating design ideas
- KK3.2.13: criteria for evaluating design ideas and the efficiency and effectiveness of solutions
- KK3.2.14: design tools for generating solution designs from design ideas, including data dictionaries, mock-ups, object descriptions, IPO charts, pseudocode
- KK3.2.15: characteristics of user experience (UX) and how these affect software design, including affordance, interoperability, security, usability
- KK3.2.16: design principles that influence the appearance and functionality of the user interface/s of the software solution.

Victorian Curriculum and Assessment Authority, Victorian Certificate of Education Applied Computing Study Design 2025–2028, VCAA, 2024, pp. 56–8

## FOR THE STUDENT

This chapter concludes the theory and skills needed for Unit 3, Outcome 2. Having covered identification and analysis of client and software requirements in Chapter 3, you are now introduced to software design, user experience, goals and objectives of information systems, legal requirements and evaluation of design ideas with consideration of efficiency and effectiveness.

By the end of this chapter, you should be equipped to generate two to three designs and choose a preferred software design according to evaluation criteria, be ready to use software development tools, and understand user experience characteristics and legal requirements for user data and information.



## Continuing Unit 3, Outcome 2

In Chapter 3, you learned how to create a software requirements specification (SRS) after consulting widely and analysing the need or opportunity.

The SRS provides the framework for you to consider when designing your solutions. Your design options will address factors identified in the SRS. These factors are all relevant preparation for Unit 3, Outcome 2. For this outcome, you will fully specify one design that has been chosen from the two to three designs you have created and documented.

This chapter begins with a discussion of how to design a software solution and to identify the relevant elements that will lead to a successful project. Next, we will talk about how to determine the characteristics of a positive user experience, both in terms of efficiency and effectiveness of the user interface. We will also talk about types of information system goals and objectives, and important legal requirements for the privacy of **data** and **information** that apply to your outcome.

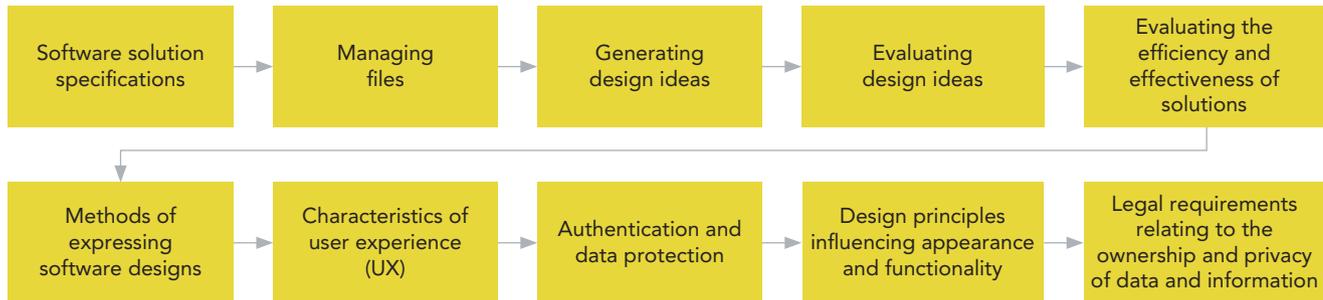


FIGURE 4.1 Chapter map

## Software solution specifications

In order to achieve a successful software design, the software developer must satisfy all design specifications and parameters. The measure for successful delivery of a software product is a combination of meeting user expectations and requirements as detailed in the SRS, followed by an **evaluation** of the client response to the final product. This is usually assessed through client or user opinion surveys, user interviews and by observing users.

If a feature or function is not included in the specifications, it will not be in the design and it will not be an evaluated design criterion. The designed software solution is created as a response to the SRS. Other factors, if discovered after the SRS has been finalised, can create chaos if not well documented. Your project management progress report and Gantt chart should include any changes to the SRS. Any subsequent changes to the software development process have implications for the evaluation of the final product and must be included.

## Managing files

Managing electronic documents forms part of an overall file management strategy. A comprehensive **file management plan** will include all aspects of handling documents, including storage, retrieval, backups, archiving and security.

Computer operating systems and applications have wonderful search functions, but these tools are not as efficient as knowing where files are kept. An easily understood filing system allows you to go directly to the folder or file. Good file management practices always save time, whether used by a single person on one computer or in organisations with a few or hundreds of employees.

## A file management plan

Establishing a file management plan involves following these steps:

- 1 creating the document management plan
- 2 implementing the plan
- 3 following through and establishing consistency.

### Creating documents

A business organisation will have many types of documents, such as sales estimates, email, spreadsheets, invoices and publicity brochures. Is there a process for creating these documents? For example, are there standard templates with logo letterhead for regular business documents? If so, where are they located? Is there a style guide to be followed? How are new documents date- and time-stamped? How will documents be shared?

**TABLE 4.1** Comparison of operating system filename limitations

| Windows OS  | macOS  | Linux  |
|---|--|--|
| <p>Windows operating system (Win32) has a limit, called MAX_PATH, of 260 characters for the file path name (e.g. C:\Program Files\filename.txt). A filename and location may be acceptable until moved or copied to another location where the new path name exceeds 260 characters. Certain characters are reserved by the operating system and are not permitted for general use:</p> <ul style="list-style-type: none"> <li>&lt; (less than)</li> <li>&gt; (greater than)</li> <li>:</li> <li>" (double quote)</li> <li>/ (forward slash)</li> <li>\ (backslash)</li> <li>  (vertical bar or pipe)</li> <li>? (question mark)</li> <li>* (asterisk)</li> </ul> | <p>There is no filename length limit, though after 1024 characters Finder has display issues. If, however, the file is ever to be shared beyond the host computer, then a 255-character limit ensures compatibility with Win32 computers. There are only two characters that are disallowed by OSX:</p> <ul style="list-style-type: none"> <li>\ (backslash)</li> <li>:</li> </ul> <p>Note: Office does not allow : (colon) in filenames. However, for cross-platform compatibility, the Win32 limitations have become the default standard.</p> | <p>Just one character is reserved:</p> <ul style="list-style-type: none"> <li>/ (forward slash)</li> </ul> <p>There is no limit on filename length. However, for cross-platform compatibility, the Win32 limitations have become the default standard.</p> |

### Naming documents

When naming documents, choose meaningful filenames that can be clearly understood. Dates can also be included to ensure that files are displayed in a sorted order. There are some limitations on filenames.

For example, the filename SofDevProjectV1.php has more meaning associated with the name than SDV1.php.

This approach also means subsequent versions can easily be identified. **Version control** is important when projects generate updates of a file. For example:

SofDevProjectV1.php  
SofDevProjectV3.php  
SofDevProjectV4.php

In a similar way, including the date can indicate when the file was created, without having to open the file or inspect Properties. For example:

ElectricityNMIDData20181201  
ElectricityNMIDData20191201  
ElectricityNMIDData20201201

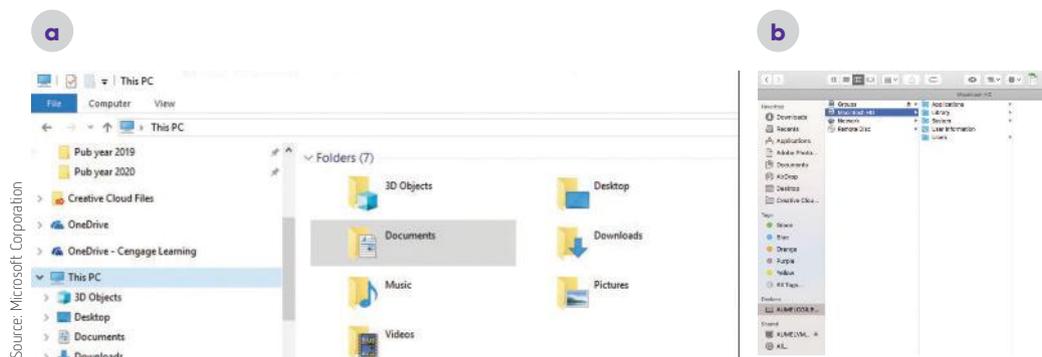
Downloaded on 1 December each year.

### Storing and retrieving files within a directory structure

Managing files on a computer requires a consistent method of allocating locations for those files. Directories are the names given to the hierarchy of folders that can be constructed.

Every new installation of an operating system establishes a standard arrangement. Once a user begins creating folders and allocating files, the certainty of where a file may be found is gone.

Directory structure is the same as folder structure.



**FIGURE 4.2**  
a Windows 10 directory;  
b macOSX directory

Some basic file management approaches that apply to every computer file system are:

- Most user-created files should go into the My Documents folder (WIN10) or Documents (macOSX).
- Brevity promotes clarity.
- Use folder names to specify where files are to be saved.
- Keep folder names short but meaningful.
- Separate work in progress from finished tasks.
- In your regular or monthly clean-up, consider moving files that you no longer need to another drive; you could call that collection 'Archives'.
- Avoid deep and wide folder structures. If there are so many sub-folders that the display is cut off, consider alternatives.
- Keep similar file types together – such as applications, videos, music, pictures, graphics, .docx, .xlsx, .pptx, etc. – as this makes searching simpler.
- Limit the number of files kept. Many files are unnecessary to keep once they have been read and action has been taken.
- Your email inbox is NOT a filing system. Save important messages and attachments to known locations. Choose to delete, move or keep email as soon as you have read the message.

Where there are too many copies of a file, version control can be a problem. While one may have been updated, the others will be out of date, despite retaining the same name.

Consider using shortcuts and aliases instead of creating multiple copies that may later become ‘orphans’ and never be updated. The single source of truth (SSOT) is a management approach that attempts to reduce duplication of files with a similar purpose but different details. For example, a downloaded or printed directory is no longer updated and cannot be changed. Yet that file can be made widely available, putting out inaccurate, misleading or wrong information.

|                       |                                       |                                 |
|-----------------------|---------------------------------------|---------------------------------|
| To create a shortcut: | Windows<br>right-click/ make shortcut | macOS<br>ctrl-click /make alias |
|-----------------------|---------------------------------------|---------------------------------|

Once created, just drag and drop the shortcut to another location.

## Cloud storage

Cloud storage is an option to overcome multiple copies of source files and work in progress. A single online data warehouse storage can hold all data files in one place. The backup strategy is the responsibility of the warehouse provider. Examples of free online storage suitable for student use are Google Drive (15 GB), OneDrive (5 GB), Dropbox (2 GB), Box (10 GB) and iCloud (5 GB).

## Backups

Backing up files and data is necessary in order to make sure that important information is not lost due to hardware failures, software issues or accidental deletions.

A **full backup** involves copying all the data from a system to a backup location. This type of backup is comprehensive and allows for complete restoration of data, but it can be time-consuming and requires significant storage space. It is often done initially and then periodically, such as weekly or monthly, to ensure that a complete set of data is always available.

**Differential backups** only save the data that has changed since the last full backup. This method is faster and requires less storage than a full backup, but more than an incremental backup. Differential backups make restoration quicker than using only incremental backups, as only the last full backup and the most recent differential backup are needed to restore data. Differential backups are typically performed daily to ensure that recent changes are consistently saved.

**Incremental backups** save only the data that has changed since the last backup of any type, whether it was a full or incremental backup. This approach is the fastest and requires the least amount of storage space; however, restoration can be more time-consuming, as it requires the last full backup and all subsequent incremental backups to be applied in sequence to fully restore the data. Incremental backups are often performed multiple times a day, such as hourly, to minimise the amount of data lost between backups.

Backups should be stored in multiple locations to ensure data safety and redundancy. **On-site storage**, such as an external hard drive or a local server, allows for quick access and recovery. **Off-site storage**, including cloud services or remote data centres, protects against physical disasters like fires or floods that could damage on-site backups. A combination of on-site and off-site storage provides a robust backup strategy.



## CASE STUDY

### Backup strategy: HealthyCare Clinic

HealthyCare Clinic is a small health clinic located in Victoria, Australia. The clinic manages and stores sensitive health records for its patients and, as a result, has a growing database of electronic health records (EHRs), including patient histories, treatment plans, prescriptions and billing information. Given the sensitivity of the data, the clinic must comply with the *Health Records Act 2001* and the *Privacy Act 1988*, ensuring that data is securely backed up and protected against loss, unauthorised access and data breaches.

The clinic needs to choose a backup method that considers the need for comprehensive data protection, compliance with legal requirements, cost-effectiveness and ease of implementation. The goal is to ensure that patient records are reliably backed up and can be quickly restored in case of data loss due to hardware failures, software issues or cyber attacks.

The clinic decides to implement the following backup strategy.

**TABLE 4.2** Backup strategy

| Type                | Frequency                  | Location  | Reason   |
|---------------------|----------------------------|---|--|
| Full backup         | Weekly, every Sunday       | On-site server as main storage and an off-site cloud storage service as a secondary storage point | Provides a comprehensive backup that ensures quick and complete restoration. Storing backups off-site mitigates risks associated with physical damage to the on-site location. |
| Differential backup | Daily (Monday to Saturday) | On-site server as main storage and off-site cloud storage service as a secondary storage point    | Balances the need for up-to-date backups with storage efficiency. Easier to restore than incremental backups while ensuring that daily changes are saved.                      |
| Incremental backup  | Hourly                     | On-site service with regular uploads throughout the day to off-site cloud storage service         | Provides the most up-to-date backup of records, minimising potential data loss. Hourly backups ensure that even recent changes are preserved.                                  |

The clinic uses automated backup software to schedule and perform these backups, ensuring consistency and reducing the chance of human error. The software is configured to encrypt data before transmission to the cloud storage service, complying with data protection regulations. Additionally, regular tests of the backup and restore processes are conducted to ensure data integrity and readiness in case of an emergency.

### Disposing of files

Placing a file in the recycle or trash marks the file to be deleted. The deletion takes place once the recycle or trash is emptied. The space on the drive is flagged as being available. The next time a file is saved, this space may be overwritten.

Deleting a file does not remove the data from the drive or memory. When a file is 'deleted', the data usually remains untouched and the file allocation table (FAT) or table of contents (TOC) entry is removed. This will allow a 'file recovery' application to reassemble the data from the volume so the file can be read. The only change is that the name of the file may have been lost. Many file recovery applications regularly record the Directory structure so that a complete re-build can be achieved.

To permanently delete a file, the data must be made unreadable, also known as being scrubbed or wiped. This will require the data to be overwritten with 0s and 1s. There are different levels of erasing data. A single overwrite of 1s and 0s is insufficient to remove the magnetic pattern of the original data. File recovery applications can recover data that has been overwritten. Multiple overwrites are necessary to prevent recovery. Three overwrites are considered to be sufficient scrubbing. This requires a cycle of writing all locations to read 1, then overwriting with 0, then repeating this twice more, for three cycles.

**Windows:** SDelete is a Microsoft command level utility that will securely clean the space on a logical disk. The number of passes can be specified.

**macOS:** Disk utility application that is integrated within macOS X has a function to erase one, seven or 35 times.

## Disposing of a drive

When a computer is being passed on to be used by someone else, the existing data should be removed or deleted. The simplest strategy is to replace the drives with new, unused drives. If the drives are to be erased, then a secure erase process of overwriting at least three times must be completed. If the removed drives contain sensitive data, then the drives should be physically disabled or destroyed.

## Archiving

It is important to understand that backups are not archives. Remember, the purpose of a backup is to allow a recovery after damage or loss. Archives, however, preserve a record of events and are often required by regulatory authorities. Archives are placed in medium- to long-term storage and are usually compressed to preserve storage space. An index of files archived would help locate the necessary document. For example, the Australian Tax Office requires all records to be kept for seven years, although prosecutions can go back 10 or more years.

The archived data may not be easily accessible to a regular user who is logged in daily. While backups may occur hourly or daily, archiving usually occurs when files are no longer needed, such as each year following the tax reporting season, after a major project is completed and signed off, when a semester unit is finished, or at the end of year.

## Factors affecting access of data

File access can be affected by the method of file organisation and the storage media onto which the files are placed. There are different technologies available for storage, either mechanical or solid state.

### File organisation and storage media

Frequently accessed files need to be placed in an easily accessible location on fast and reliable media. The choice of media for storage is often about capacity, but other important attributes are:

- latency, or speed of access
- reliability
- cost
- ease of use.

The fastest access can be provided by **SSD**. When placed on SSD, the operating system offers a typical start-up time of 10 seconds, compared to several minutes for **HDD**. Modern operating systems require at least 10 per cent ‘scratch space’ for saving frequently accessed system files. A hybrid solution is a ‘fusion drive’, which has a small SSD attached to an HDD.

## Generating design ideas

Generating **design ideas** requires a logical approach as well as a creative mind. These design thinking skills are referred to as convergent thinking and divergent thinking, respectively. There are several aspects to design that need to be considered. The estimated experience of the expected users may be influenced by the previous experience of the software designers and developers. Generally, there is low to very low awareness of accessibility issues when designing software applications. Designing products that can be used by people with a wide range of abilities and disabilities is called **universal design**.

Some individuals use specialised software and hardware called **assistive technology** to operate software products. For example, a person who is blind might use a screen reader program with a speech synthesiser to access the content and functionality of a program.

### SCHOOL-ASSESSED TASK TRACKER

- Design brief
- Project plan
- Analysis
- Folio of alternative design ideas
- Development
- Evaluation and assessment
- Final submission

## Convergent thinking

**Convergent thinking** involves coming up with a single, well-established answer to a problem. All avenues are explored when considering possible solutions, and the best solution is found, ignoring all constraints. This type of thinking involves completing research, such as looking at other software programs that achieve similar purposes, or visiting other companies in the same field to interview or observe their employees. Convergent thinking results in design ideas that are based on other, proven ideas. Data is used from interviews, reports, observations and surveys (see Chapter 3), and this data is extrapolated to provide an optimal solution to the problem.

## Divergent thinking

**Divergent thinking** is more creative than convergent thinking. It involves exploring many possible solutions using spontaneous, free-flowing techniques, such as mind mapping, brainstorming, meditation and role-playing. Divergent thinking involves considering as many possible solutions as you can in a given amount of time. Problems are often explored using stream-of-consciousness techniques. These techniques typically produce unexpected solutions that may not necessarily have been considered using convergent thinking techniques.

Divergent thinking is sometimes described as finding solutions 'outside the box'.

## Combining thinking skills

On their own, neither convergent thinking nor divergent thinking are likely to produce the most efficient and effective design idea. Convergent thinking ignores constraints that are likely non-negotiable, and divergent thinking may never produce an optimal result. The most effective method of generating design ideas is to use a combination of these thinking skills to produce design ideas that are worth further exploration.

## Techniques for design ideation

There are several techniques for generating a range of creative and appropriate design ideas. They all aim to find the most effective and efficient software solution to solve a client problem. Your techniques should take into account the functional and non-functional requirements of your solution.



**FIGURE 4.3** Ideation can be performed using a variety of techniques.

## Mood boards

A mood board in user interface design is a visual tool that helps designers communicate the general 'feeling' or 'atmosphere' of a project. It is a collection of images, text, colours, patterns and other design elements that together convey a particular style or concept. Mood boards are used in the early stages of the UI design process to establish the creative direction and potential look and feel of a software solution.

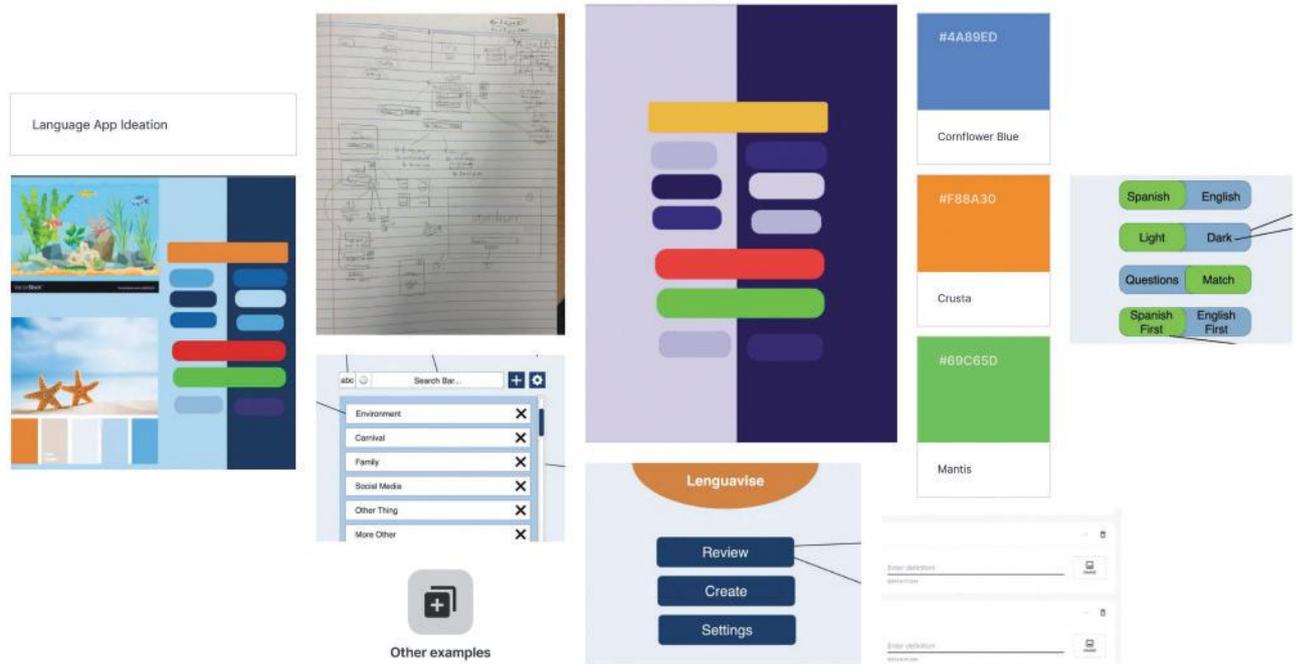
Mood boards help in gathering and organising inspiration. They allow designers to explore different visual directions and conceptualise the project's overall design language without requiring them to spend a lot of time creating the designs themselves. They serve as a communication bridge between designers, clients and other stakeholders. By presenting a mood board, designers can ensure that everyone involved has a shared understanding of the visual direction.

Mood boards are not rigid blueprints. They allow for flexibility and iteration, enabling designers to refine and evolve the visual style as the project progresses.

There are a number of online tools that you can use to create a mood board. One free tool is Milanote.



Weblink  
Milanote



Source: Selina Dennis

**FIGURE 4.4** An example mood board for a language learning application

## Brainstorming

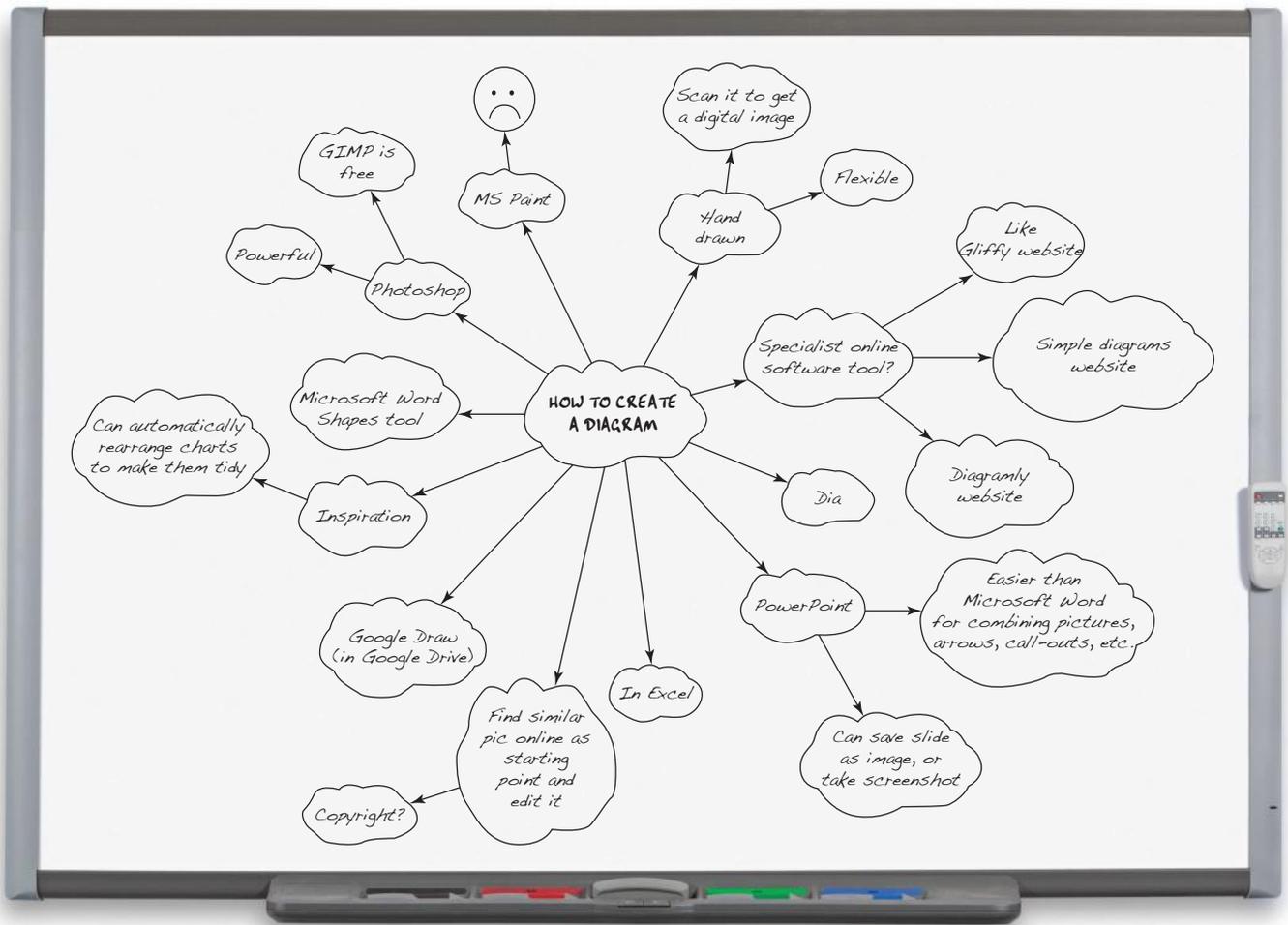
Brainstorming is a process where ideas are presented in a non-judgemental, spontaneous, unstructured and admittedly somewhat haphazard process. The only rule is that no idea is criticised or rejected; every idea, no matter how outrageous or silly, goes onto a list of possible solutions. Try not to hamper your imagination by rejecting ideas too soon.

Participants must have no fear of being judged, making mistakes or breaking rules. While some ideas may turn out to be ridiculous, sometimes a half-baked, half-comical concept may in fact turn out to be creative genius, or it may stimulate a related idea that turns out to be perfect.

There are certain rules that you need to follow when running a brainstorming session. The most important one is that no one judges any contribution – all suggestions must be accepted. An idea that may seem slightly crazy to begin with can sometimes be workshopped into something great. In the 1970s, a brainstorm produced the idea of a pet rock. The idea was workshopped, and before long you could buy not only a pet rock, but also a pet rock house and a training manual. The idea made the company millions of dollars. It was the pre-technological version of the 1990s Tamagotchi.

SCHOOL-ASSESSED TASK TRACKER

Design brief
  Project plan
  Analysis
  Folio of alternative design ideas
  Development
  Evaluation and assessment
  Final submission



**FIGURE 4.5** Whiteboards are popular brainstorming aids because they are visible to all and are easy to edit.

Make sure that everyone listens to everyone else's ideas. Make sure that only one person talks at a time, and that there is only one idea at a time. This ensures that even the most shy member of the group will contribute, and also makes it easier to record the ideas. Using these rules will help you gather a large number of ideas to work with.

Brainstorming example: NASA faces the problem of lifting people and equipment into space. It is extremely difficult, expensive, loud and dangerous. How can it be improved?

Brainstorms for this project include:

- helium balloons: float up, take off
- fire the rockets from the tops of mountains: reduce the distance to space
- a very, very tall ladder
- a giant catapult
- a jet airliner carries the rocket ship as high as it can, then the rocket takes off from there
- antigravity.

While antigravity has no foundation in real science, some of the other design ideas could work, and warrant more research. The team chuckles at the funny 'very, very tall ladder' idea until one person pauses and says:

‘Wait ... I wonder if we could somehow get a super strong cable from the ground to low Earth orbit and anchor it in space, like a space elevator. You would ride up the cable to the end. The rocket can take off from there. You wouldn’t need all the fuel to achieve escape velocity ... no need to launch from the ground.’

‘And ... and re-entry,’ says someone else. ‘You could ride down the cable to get home. Simple. And low cost.’

From thinking that was whimsical, impromptu, unconventional and unconcerned with constraints came a serious concept that was further investigated by scientists at the Shizuoka University in Japan, with deployment of a prototype in October 2018. They aim to have a fully functional space elevator by 2050.

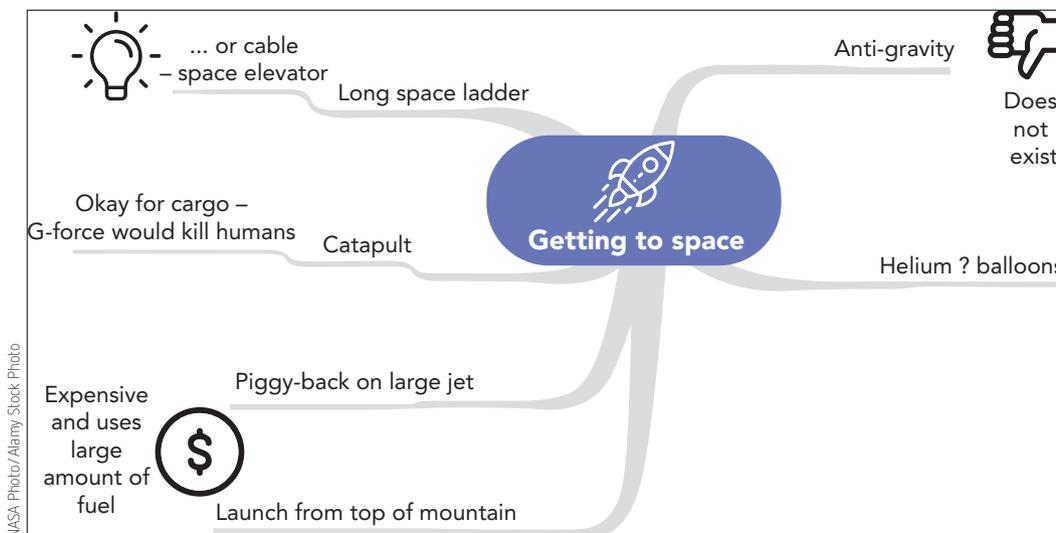
Brainstorming is helped by including people with different skills, experiences and areas of expertise. Sometimes, a group of specialists struggling for a solution may be inspired by an idea from someone who is not constrained by their shared assumptions, preconceptions and modes of thought.

### Mind mapping

Mind mapping is an ideal technique to complement the process of brainstorming. Mind mapping involves quickly generating and linking ideas. It is a creative and flexible tool that enables you to add, connect, organise and reorganise ideas (see Figure 4.5). Mind-mapping software is generally flexible enough that you will not need to stop very often to learn how it works while mapping; in other words, your creative flow will not be interrupted.

Unlike whiteboards or physical sheets of butcher’s paper, electronic mind maps can stretch endlessly in any direction. It is easy to add or remove links between items, or move entire branches of thought to new locations. Also, you will not have to copy out all of the scribbled ideas at the end of the session. The mind map can be saved for later development, printed, saved as an image, or transferred to a word processor.

Using the ‘getting to space’ problem, an example of a mind map of the design process is shown in Figure 4.6.



**FIGURE 4.6**

Mind-mapping a project about how to get into space, using Inspiration software

### Graphic organisers

Graphic organisers are visual methods of organising ideas. One popular type of graphic organiser is a PMI. A PMI involves organising ideas into three columns: what has been successful (Plus), what was unsuccessful (Minus) and what needs more thought (Interesting). You can use a PMI to reflect and evaluate, or to brainstorm new ideas (see Table 4.3).

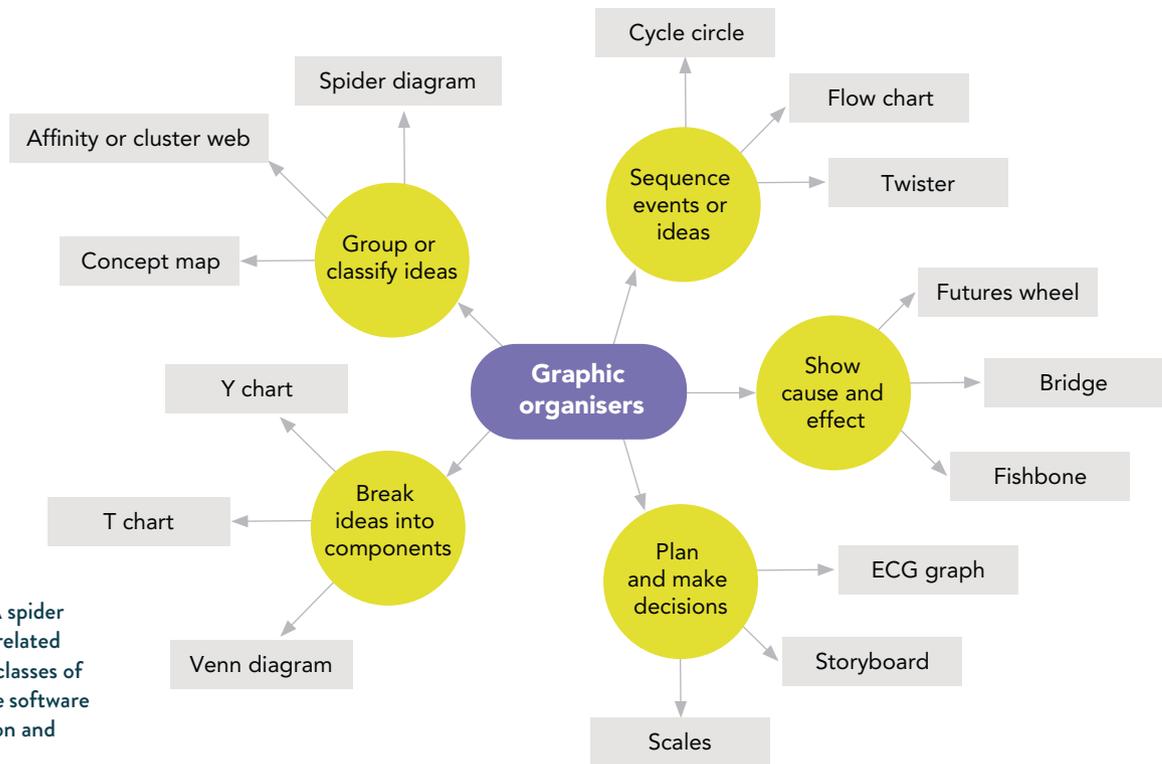
**TABLE 4.3** Example of a PMI

| <i>Using helium balloons to reach space</i> |                            |                             |
|---|----------------------------|-----------------------------|
| P   | M                          | I                           |
| Quiet                                       | Crashes if gas leaks       | Can balloon go high enough? |
| Relatively cheap                            | Slow to reach stratosphere | How much does helium cost?  |
| Limited payload weight                      |                            |                             |

A spider diagram (Figure 4.7) is a powerful tool that gives an overview of a central idea. The body of the spider is the central idea and the branching legs radiate out to related ideas and sub-ideas.

There are dozens of variations of such visual tools to help organise and clarify ideas. Others include character maps, concept webs, POOCh (Problem – Options – Outcomes – Choice), ranking ladders (to prioritise or rank ideas, information or tasks), stair steps (to organise a process step by step), a chain of events, sequence charts (to put sequential factors in order), pie charts (to represent the relative sizes of components in a whole), bone charts, organising trees and even Gantt charts for managing project timelines. Gantt charts were discussed in Chapter 3 (see page 91).

Edward deBono became well known for 'lateral thinking'. Mind mapping, rubrics, POOCh and SCAMPER are some of his approaches worth considering.



**FIGURE 4.7** A spider diagram showing related concepts and subclasses of concepts. Suitable software includes Inspiration and LucidChart.

## Sketches and annotations

Sketches and annotations are valuable tools in the software design process, often used for brainstorming, communicating ideas and detailing software architecture. In the early stages of software development, they can be very useful in helping turn abstract concepts into tangible and feasible designs that can be implemented into final products.

Sketches provide a visual platform for brainstorming. They help software designers and developers to conceptualise and visualise different aspects of the software, such as user interfaces, system architecture or data flow. Sketching is a quick way to explore multiple ideas without the need for detailed or technical drawings. They are often drawn by hand, either physically or digitally.

Sketches can be used to demonstrate how a software feature will work or how different parts of the system will interact. They help in ensuring that everyone has a shared understanding of the proposed design or solution.

For designing user interfaces, sketches can illustrate the layout of screens, placement of buttons, menus and other interactive elements. This is particularly useful in the early stages of designing user interfaces, allowing for rapid iteration and experimentation.

Annotations added to sketches provide additional detail and context. This might include descriptions, labels or notes about functionality, behaviour or design rationale. Annotations can also specify technical requirements or constraints that need to be considered during development.

Sketches allow for easy modifications and iterations. As the design process progresses and feedback is received, sketches can be quickly updated to reflect changes or improvements.

Initial sketches often serve as the foundation for more detailed designs or prototypes. Once a sketch is agreed upon, it can be elaborated into a detailed design specification using more formal tools, such as pseudocode, mock-ups or object descriptions, or be used to help develop a proof of concept or prototype.

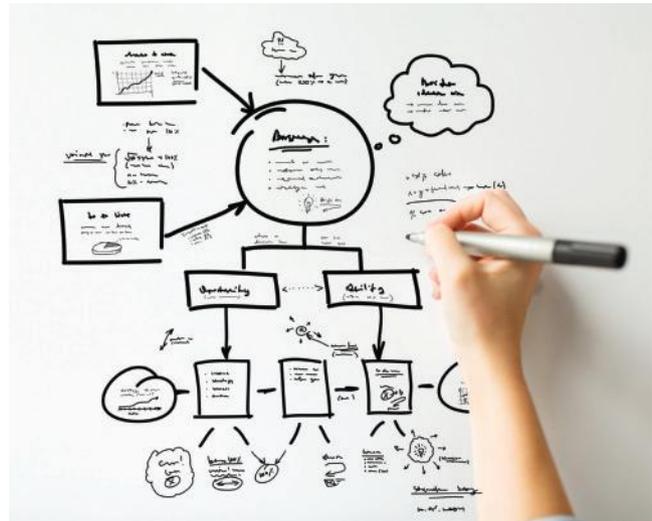


Baitoey CS/Shutterstock.com

**FIGURE 4.8** Sketches such as wire frames are often drawn quickly by hand.



**FIGURE 4.9** Early designs of user interfaces are often rough sketches.



**FIGURE 4.10** Annotations to sketches help clarify design concepts.

## Tips for critical and creative thinking

Creative design can be learned. You do not need to be born with the talent. There are techniques that anyone can use to improve their design creativity.

### Substitute

Replace part of the problem with something else. For example, if you are producing hundreds of certificates, do not use mail merge to take data from a spreadsheet and insert it into a word processor. Use a database instead.

### Combine

Join unconnected things together, such as reducing the weight of camping supplies by combining a spoon and fork into a single utensil – the spork.

### Adapt

Use an existing component in a different way. For example, use presentation software such as PowerPoint or Keynote to create a poster. The first spreadsheet was created using the concept of paper-based accounting books.

### Strip

Strip the problem right back to its most basic parts and see what is left. For example, the tiny and cheap computer, the Raspberry Pi, is a stripped-down Linux PC with minimal components. Inspecting the basics may reveal the nature of a problem more clearly.



**FIGURE 4.11**  
A Raspberry Pi

pengpeng/Getty Images

## Compare

Ask yourself, ‘What other thing do I know that resembles this problem, and how does that other thing work?’ For example, when sending a number of print jobs to a single printer, how can they be handled? Like a group of people waiting at a gate, you could organise them into a queue and process them in the order of their arrival.

## Sleep on it

Creators often reach a point where they can make no further progress. Rather than dwelling on the same failed ideas, it is often better to let them go and think of something else. While the front of your brain is enjoying a wrestling match with your sibling, or an episode of *Australia’s Got Talent*, the back of your brain will busily be pulling ideas together to create a solution.

## Research

Thomas Edison, developer of the light bulb, phonograph and electric power, said, ‘Through all the years of experimenting and research, I never once made a discovery. I start where the last man left off. It is important to learn from your predecessors so you do not waste time ‘reinventing the wheel’.

You are unlikely to be the first person to have faced a particular problem. How have other people solved problems similar to the one you face? How have others coped? Their successes may lead you in the right direction, and their failures may prevent you wasting time. Take care when using Google. Make sure you acknowledge your sources, and watch out for false information.

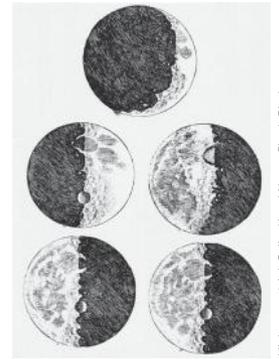
## Visualisation

Geniuses often represent their thoughts visually because words cannot adequately convey their ideas. Einstein was famous for his non-verbal thought experiments. He visualised travel at the speed of light as travelling on a train. He said that written words and numbers did not play a significant role in his thinking process.

Leonardo da Vinci is renowned for his sketches of his inventions. Galileo Galilei drew diagrams and maps of planetary orbits and phases of the Moon (Figure 4.12). Sigmund Freud, Alfred Hitchcock, Isaac Asimov, Beethoven and Mozart all reported the use of mental imagery in their creative processes. In *Thinking in Pictures* (Vintage Press, 1996), Dr Temple Grandin, famous for her work with livestock, wrote:

I think in pictures. Words are like a second language to me ... Language-based thinkers often find this phenomenon difficult to understand, but in my job as an equipment designer for the livestock industry, visual thinking is a tremendous advantage.

You may choose to use software simulations or models to help structure your thinking and construct knowledge (Figure 4.13).

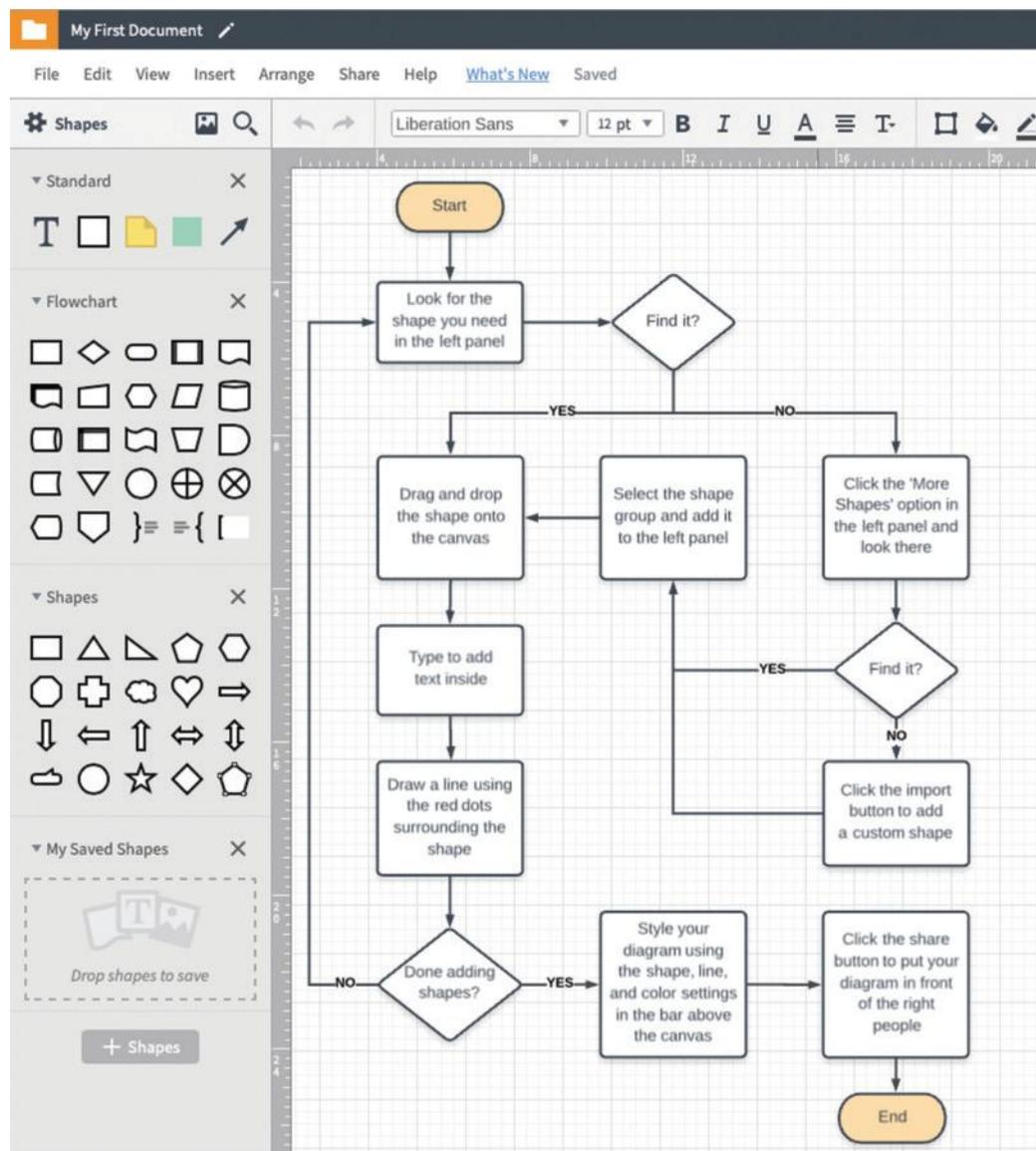


**FIGURE 4.12** Galileo's drawings of phases of the Moon, based on observations through his telescope, from his 1610 manuscript, *Sidereus Nuncius*

History and Art Collection/Alamy Stock Photo

### SCHOOL-ASSESSED TASK TRACKER

Design brief
  Project plan
  Analysis
  Folio of alternative design ideas
  Development
  Evaluation and assessment
  Final submission



Source: LucidChart

**FIGURE 4.13** Creating a flow diagram with LucidChart

### Be observant and prepared

Many inventions have arisen from people seeing things that were similar to the problem they were trying to solve. Can a blockage in a canal be similar to solving a blockage in blood vessels? How can thousands of ants travel safely and quickly through a small gap, while a crowd of human spectators takes nearly an hour to leave a football stadium?

Play Doh, Post-It Notes, potato chips, Velcro, Teflon, Cellophane, insulin, dynamite, stainless steel, Super Glue, cornflakes and vulcanised rubber were all found by observant people after accidents or failed attempts to invent something else.

Research also suggests that creative people are typically hoarders – they keep lots of knick-knacks, photos and articles around, and revisit these for stimulus.

Keep your eyes open for connections between apparently dissimilar things. Revolutionary ideas often come from ‘ridiculous’ connections that no one previously considered. Physicists argued whether light was a wave or a particle, until someone innocently (and correctly) proposed that it could be both.

Someone with a solid knowledge of a topic and an ongoing curiosity about new ideas is receptive, and this will help them recognise the importance of an observation to an existing idea. The uncreative observer will either not notice the idea, or will fail to see its relevance to a developing design.

### Take risks, persist and be brave

Someone with a creative design idea often needs to take the risk of being dismissed, mocked or rejected. Many of the greatest breakthroughs were rejected at first, and took a lot of time and effort to be accepted.

The germ theory, that diseases are caused by microorganisms, was put forward by Louis Pasteur in the 1860s. It superseded the miasma theory, which suggested that a poisonous vapour in the air caused diseases. This theory had endured for several centuries. Pasteur's theory was initially mocked, until further experimentation showed it to be most likely correct.

More recently, Steve Wozniak combined the concepts of a typewriter, a calculator and a display. He was envisioning a whole new technological paradigm: the personal computer. His employer at the time, Hewlett-Packard, rejected Wozniak's concept five times. This led Wozniak to team up with Steve Jobs, which in turn led to the creation of Apple Computer Inc. The idea of a tablet computing device had been tried by Apple and Microsoft and ended in failure. Jobs tried again when the technology was mature, following the development of the iPod and iPhone, and the resultant iPad was this time successful.

James Dyson (of Dyson vacuum cleaner fame) is believed to have created 5000 prototypes of his vacuum cleaner over five years before he got it right.

These examples show that persistence, not genius, is the greatest contributor to success.

Quotes by Thomas Edison include the following:

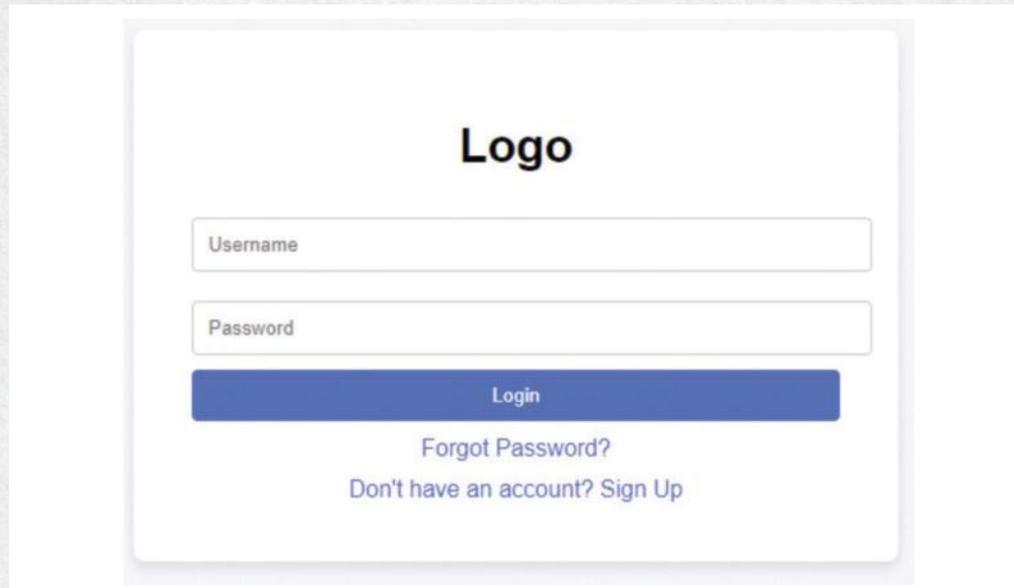
- 'Genius is one per cent inspiration; 99 per cent perspiration.'
- 'Many of life's failures are people who did not realise how close they were to success when they gave up.'
- 'I have constructed 3000 different theories in connection with the electric light, each one of them reasonable and apparently likely to be true. Yet only in two cases did my experiments prove the truth of my theory.'

## Evaluating design ideas

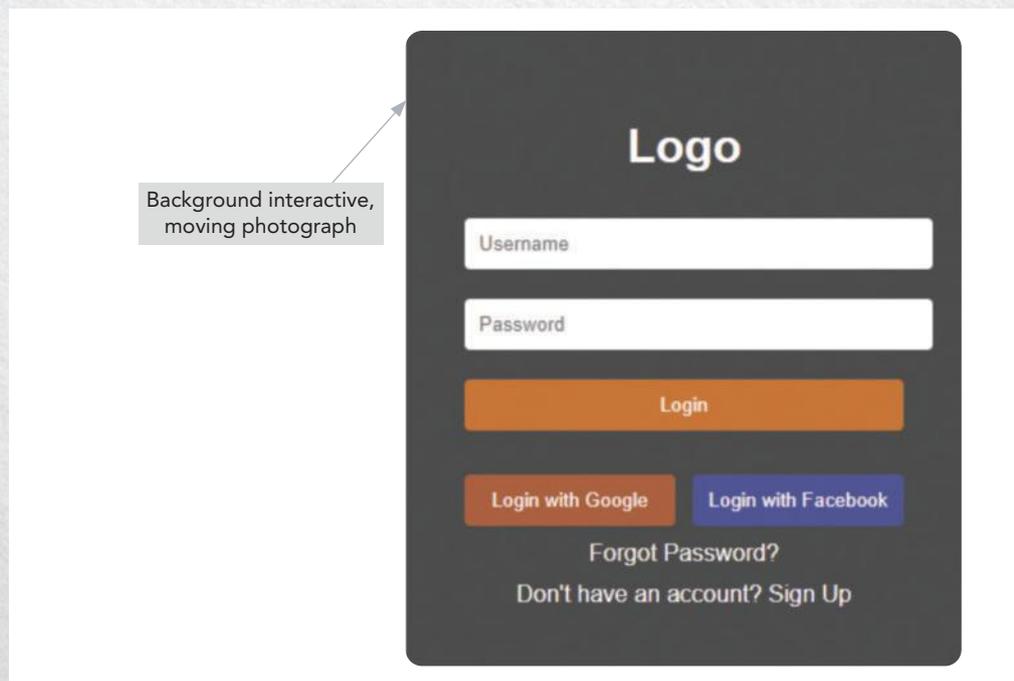
The criteria that should be used to evaluate design ideas must be based on the software requirements specification (SRS) produced at the end of the analysis stage. Elements such as the constraints, functional requirements and non-functional requirements should be considered carefully, and each design should be evaluated in relation to these components before a preferred design is selected. For example, a decision to be made in relation to functional requirements would be the file format in which to save data: plain text, CSV or XML (see Chapter 1, pages 13–16). There are advantages and disadvantages to each, which would then need to be considered against the requirements listed in the SRS before making a final decision. Similarly, the non-functional requirement of usability may result in alternative user interface designs, which must then be evaluated against the needs of the users before a decision is made as to which design is preferred.

**CASE STUDY****Dani's design ideas**

Dani is designing her user interface for her School-assessed Task (SAT), which needs to have a username and password interface before users can access the main functionality of her software. She has completed some brainstorming and narrowed her design ideas down to two – one being a minimalist and modern design (Design 1), the other interactive and engaging (Design 2) – and has sketched out a rough design. She completes a comparison table so she can focus on the differences between the two designs.

**Design ideas****Design 1**

**FIGURE 4.14** Image of username and password interface in a minimalist and modern design

**Design 2**

**FIGURE 4.15** Screenshot of username and password interface in an interactive and engaging design

**TABLE 4.4** Design features

| Feature              | Design 1  | Design 2   |
|----------------------|---|--|
| Background           | Plain background  | Full screen background with moving image   |
| Layout               | Centred login form  | Semi-transparent overlay for the login form  |
| Colours              | Soft, neutral colours   | Vibrant colours with high contrast   |
| Fonts                | Simple sans serif fonts   | Stylish, modern fonts  |
| Logo position        | Top centre  | Top left   |
| Input fields         | Standard input fields for username and password   | Floating input fields for username and password  |
| Buttons and links    | Prominent 'Login' button, smaller 'Forgot Password?' link below the login button, centred | Large, colourful 'Login' button, 'Forgot Password' and 'Sign Up' links below the button.<br>Links to login via Facebook and Google |
| User-experience goal | Focused, minimalist, distraction-free, functional   | Engaging, visually appealing, interactive  |

### Comparing the designs

Dani then creates another table to help her decide between the two designs, outlining the advantages and disadvantages of each.

**TABLE 4.5** Design advantages and disadvantages

| Criteria      | Design 1  | Design 2   |
|---------------|---|--|
| Usability     | Advantages: <ul style="list-style-type: none"> <li>• Simple and straightforward</li> <li>• Focuses on essential elements</li> </ul> Disadvantages: <ul style="list-style-type: none"> <li>• Less engaging than Design 2</li> <li>• Limited flexibility for additional features such as login methods</li> </ul> | Advantages: <ul style="list-style-type: none"> <li>• Adds options for other logins such as Google and Facebook, increasing flexibility</li> <li>• Visual cues can aid navigation</li> </ul> Disadvantages: <ul style="list-style-type: none"> <li>• Can be overwhelming for some users, a little bit busy</li> <li>• Potentially slower load times because of the rich media content</li> </ul>  |
| Accessibility | Advantages: <ul style="list-style-type: none"> <li>• High contrast and clean layout improve accessibility</li> <li>• Simple design ensures compatibility with screen readers</li> </ul> Disadvantages: <ul style="list-style-type: none"> <li>• Limited dynamic feedback</li> </ul>                             | Advantages: <ul style="list-style-type: none"> <li>• Dynamic elements can provide more feedback and assistance</li> <li>• Larger buttons and text can aid users with visual impairments</li> </ul> Disadvantages: <ul style="list-style-type: none"> <li>• Background images/videos may reduce contrast</li> <li>• May require more effort to ensure that all elements are accessible</li> </ul> |
| Readability   | Advantages: <ul style="list-style-type: none"> <li>• Clear, simple fonts</li> <li>• High contrast between text and background</li> </ul> Disadvantages: <ul style="list-style-type: none"> <li>• Less visually stimulating</li> <li>• Basic design might appear too plain</li> </ul>                            | Advantages: <ul style="list-style-type: none"> <li>• Modern, stylish fonts can be engaging</li> <li>• Well-designed overlays can maintain readability despite background complexity</li> </ul> Disadvantages: <ul style="list-style-type: none"> <li>• Busy backgrounds can hinder text readability</li> <li>• Stylish fonts may be harder to read for some users</li> </ul>                     |



**TABLE 4.5** Design advantages and disadvantages (*continued*)

| Criteria       | Design 1  | Design 2   |
|----------------|---|--|
| Attractiveness | Advantages: <ul style="list-style-type: none"> <li>• Clean and simple look</li> <li>• Professional appearance</li> </ul> Disadvantages: <ul style="list-style-type: none"> <li>• May seem too plain or boring</li> </ul>  | Advantages: <ul style="list-style-type: none"> <li>• Visually engaging</li> <li>• Can create a memorable user experience</li> </ul> Disadvantages: <ul style="list-style-type: none"> <li>• Risk of being overly flashy or distracting</li> </ul>  |
| Clarity        | Advantages: <ul style="list-style-type: none"> <li>• Direct focus on login process</li> <li>• Clear call-to-action buttons</li> </ul> Disadvantages: <ul style="list-style-type: none"> <li>• Minimalist approach may lack detailed guidance for new users</li> </ul> | Advantages: <ul style="list-style-type: none"> <li>• Clear visual hierarchy with attractive design elements</li> <li>• Engaging design can guide user flow</li> </ul> Disadvantages: <ul style="list-style-type: none"> <li>• Complex design elements can potentially confuse users</li> <li>• More elements may dilute the main action focus</li> </ul> |

### Selecting the preferred design

Dani considers her intended users, who have high vision needs and are also not very confident in using technology. She consults with her client, who states that while attractiveness was important to her, it was accessibility, usability and clarity that were the most important measures for the solution. Dani constructs the following questions to allow her to select between the two designs, ranking each design idea on a scale of 1 to 5.

**TABLE 4.6** Selection criteria

| Criteria      | Question  | Design 1 | Design 2 |
|---------------|---|----------|----------|
| Usability     | Is the software easy to navigate for people who are not tech-savvy?                       | 5        | 2        |
|               | Does the design focus on essential elements without overwhelming users?                   | 5        | 3        |
|               | Can users quickly and easily complete the login process without confusion?                | 5        | 4        |
| Accessibility | Does the design provide high contrast between text and background for better readability? | 5        | 5        |
|               | Is the interface compatible with screen readers and other assistive technologies?         | 5        | 2        |
|               | Are the buttons and text clear and easy to read for users with visual impairments?        | 5        | 5        |
| Clarity       | Does the design have a clear and direct focus on the login process?                       | 5        | 4        |
|               | Are the call-to-action buttons prominent and easy to identify?                            | 5        | 5        |
|               | Is the overall layout straightforward and free of unnecessary distractions?               | 5        | 2        |
| Total         |   | 45       | 32       |

As a result, Dani decides to go with Design 1, as its simplicity, high accessibility and clear focus on essential functions are its strengths. This design ensures that users can easily understand and navigate the login process, reducing frustration and improving the overall user experience.

When designing the solution to a problem, the first design idea you have will rarely be the best one. A different strategy might be cheaper, easier, faster or more effective, or may better meet the client's needs. While one design idea may be attractive to the developer, the client may have non-technical constraints or priorities that will make another strategy more attractive. Providing a range of design ideas means the client can choose the solution that best suits them.

Although previously proven strategies can be useful, you need to be willing to think outside the box. You may have used a design idea successfully in the past, but it may not be appropriate in the current circumstances. Old strategies will not work for you in every situation.

A successful problem-solver will consider current functional and non-functional requirements and relevant constraints in order to develop an imaginative range of options. The best design idea can then be chosen and developed into a detailed design.

The criteria for choosing the best design idea may include effectiveness criteria such as accessibility, accuracy, attractiveness, clarity, communication of message, completeness, maintainability, readability, relevance, timeliness and usability. They may also include efficiency criteria such as cost of data and file manipulation, functionality and speed of processing.

When using these criteria, it is also helpful to consider the following:

- ease of use
- how long it will take to implement
- scalability (how easily the product can be increased in capacity)
- its scope for future modification and enhancement (e.g. one design idea may include plug-ins so extra functionality can easily be added)
- the degree to which it satisfies all requirements
- the degree to which it copes with constraints
- ease of implementation
- development cost, future running and maintenance costs
- the amount of labour required to create it and keep it working
- the amount of disruption likely to be caused to the organisation
- compatibility with existing hardware, software, data and procedures
- the amount of training required for staff.

To make an evaluation, the criteria used must ask a question. The answer to that question will indicate whether the criteria has been achieved or satisfied.

For example, some of the previously listed criteria would be evaluated by asking:

- Is the software easy to use?
- Can the software be implemented quickly?
- Can the software be scaled for more users?
- Can the software be simply modified?
- Can the software be implemented easily?

Some design decisions can be very difficult, and require careful balancing of competing needs – usually cost and time against quality. A design that is cheap and quick to produce may quickly wear out, be barely competent or unpleasant to use. A superior design that would lead to a solution with a long life and happy users will probably take longer to produce and cost more.

Compare the likely differences in design philosophies and criteria between the pairs shown in Figure 4.16.



FIGURE 4.16 Pairs

**Evaluation criteria** should be documented, ideally in a table format, and as each design is evaluated the outcome of each criteria should be recorded.

## Evaluating the efficiency and effectiveness of solutions

Testing and evaluation are not the same thing! Testing can be done at any time, and may use dummy or test data. Testing looks for specific results and expected behaviour by the known capabilities of programs or equipment. Evaluation is conducted after the software solution has been implemented. Users are consulted and criteria are considered.

Evaluation typically involves checking to see how well a software solution has met its stated requirements. This post-implementation review evaluation is normally performed at a set time period after the solution has been put into place, where the time frame selected is relevant to the context in which the software operates. However, such an evaluation would not generally occur in the first six months of software being put in place.

In order to conduct an evaluation, an evaluation strategy needs to be created. This normally occurs at the end of the design stage of the problem-solving methodology. The evaluation strategy specifies the time frame in which an evaluation will occur and outlines the data that will need to be collected to complete the evaluation, including a description of

the methods and techniques that will be used to collect that data. It is also made clear how the data collected relates to the criteria that was written in the design stage.

The software solution is then evaluated in terms of **efficiency** and **effectiveness**.

## Efficiency

The **efficiency of a solution** concerns how much time, cost and effort has been applied to achieve the intended results. This could include measurements against the speed of processing, the functionality of the software, or the cost of file manipulation.

## Effectiveness

The **effectiveness of a solution** relates to how well a solution achieves its intended results. This typically requires measurements of the quality of the solution in relation to its **accessibility, accuracy, attractiveness, clarity, communication of message, completeness, maintainability, readability, relevance, timeliness** and **usability**. These are explained further in Chapter 5, pages 220–1.

Some examples of the criteria that could be used to evaluate the effectiveness of a solution are included in Table 4.7.

**TABLE 4.7** Criteria for evaluating the effectiveness of a solution

|                                 |   |
|---------------------------------|---|
| <b>Accessibility</b>            | How well can the system be accessed by someone who is hearing impaired or vision impaired? For example, does your solution use a colourblind safe palette? Do all images have an ALT-tag? Will a screen reader work with your solution?                                     |
| <b>Accuracy</b>                 | Is all of the data stored accurate in relation to how it was entered?<br>Are all calculations accurate 100% of the time?<br>Are all reports produced within the correct date ranges, including boundary values?   |
| <b>Attractiveness</b>           | Are the colours used throughout the software complementary?<br>Are the colour choices appropriate to the context?   |
| <b>Clarity</b>                  | Is the language used in the software age-appropriate?<br>Are headings, labels and buttons consistently used throughout the software?  |
| <b>Communication of message</b> | Have appropriate and region-specific conventions been used for all displayed data (e.g. currency conventions, date/time format, alignment)?   |
| <b>Completeness</b>             | Were all of the functional and non-functional requirements that were required by the client implemented in the software system?   |
| <b>Maintainability</b>          | Are there clear and well-documented processes for logging and tracking bugs reported by users?<br>Is there an established schedule for monitoring and evaluating system performance?  |
| <b>Readability</b>              | Can every part of the software program be easily read by its users?<br>Are the fonts chosen appropriate in size and face to the system on which the software is installed?<br>Are contrast ratios acceptable?<br>Is the text colour readable against the background colour? |
| <b>Relevance</b>                | Is all of the information produced and shown by the software system relevant to its intended use?<br>Are there any parts of the system that are not often used (or not at all) by users?  |
| <b>Timeliness</b>               | Does the software respond to requests within an acceptable time frame?  |
| <b>Usability</b>                | Are all of the elements within the software easy to use?<br>Are there sections of the program where users are more likely to make errors?   |

'Cost' in relation to the efficiency of a solution does not necessarily mean monetary cost. It can also refer to time, as manipulating files can be quite slow depending on the amount of data being manipulated and the selected algorithms that have been implemented to handle that data.

## Methods of expressing software designs

Once you have decided on a design architecture where:

- the use case diagram identifies each entity and accurately shows all relationships
- the context diagram has all aspects of your software solution recorded
- the data flow diagram has been mapped out and your model is consistent

you are ready to begin the software design documentation. Documentation is essential to record definitions, decision details and assumptions that underlie the final software solution. For example, if a date of birth input allowed an unrestricted age range, but a further calculation assumed only adults would be included, then unexpected results would be generated if an adolescent used the application. This assumption, if documented, would be shown to cause problems in the evaluation stage of the PSM.

There are several methods available to document software designs, including:

- data dictionaries
- object descriptions
- mock-ups
- input–process–output (IPO) charts
- pseudocode.

### Data dictionaries

In 2015, the International Organization for Standardization recommended specific ways of naming files and variables: ‘singular nouns, present tense verbs, uppercase 1st letter for second and subsequent words. DO NOT prefix names with type or table name’.

ISO 11179-5:2015

There are several types of data dictionary used in the computing industry. In Software Development, the data dictionary has a particular meaning, which is different from the definition used in VCE Data Analytics. Both types are included here so you can compare the different purposes.

### Data dictionary used as a database design tool

Data dictionaries are used when designing databases to explain how to set up the properties of each field in database tables. Table 4.8 is an example of a data dictionary that can be used in database design.

**TABLE 4.8** Example of a data dictionary used by database developers

| Modern naming style | Traditional naming style (old school) | Type             | Format       | Size  | Purpose                          | Example     |
|---------------------|---------------------------------------|------------------|--------------|-------|----------------------------------|-------------|
| id                  | txtCustomerID                         | Text             | XXX99        | 5     | Customer ID                      | SMI40       |
| firstName           | txtFirstName                          | Text             | Xxxxxxxxxxxx | 15    | Customer given name              | Jane        |
| lastName            | txtFamilyName                         | Text             | Xxxxxxxxxxxx | 25    | Customer family name             | Smith       |
| birthDate           | dateDOB                               | Date             | YYYY-MM-DD   | Fixed | Date of birth                    | 2001-07-19  |
| clubMember          | boolClubMember                        | Boolean          | Yes/No       | Fixed | Is a member of the buyer’s club? | Yes         |
| memberYears         | intMemYears                           | Integer          | 99           | Fixed | Years a member                   | 12          |
| sales               | sngSales                              | Single precision | ###,###.##   | Fixed | Total amount spent               | \$12,456.78 |

## Data dictionary used as a software design tool

A **data dictionary** in software design is used to plan data elements in the software, including definitions, formats and explanation of usage. Its main purpose is to ensure consistency and clarity in the management and use of data within the software system.

It usually lists the names of the variables, their type, size (in characters), scope in regard to the code and purpose of the variable's function. In some cases, it may also list the format and example for the variable.

Data dictionaries are most commonly used when code needs to be modified at a later date by programmers. It allows them to understand the purpose of a variable or array that may be unclear without it. Data dictionaries may need to be kept up to date during the development stage, when changes are introduced to the initial design.

**TABLE 4.9** Example of a data dictionary used by software developers

| Modern naming style | Traditional naming style (old school) | Type          | Size | Scope | Format         | Purpose  | Example     |
|---------------------|---------------------------------------|---------------|------|-------|----------------|--|-------------|
| id or ID            | intCustomerID                         | Integer       | 6    | Local | 999999         | Customer ID  | 201940      |
| firstName           | strFirstName                          | Text          | 50   | Local | Xxxxxxxxxxxxxx | Client first name                                  | Jane        |
| lastName            | strLastName                           | Text          | 50   | Local | Xxxxxxxxxxxxxx | Client last name                                   | Smith       |
| DOB                 | dtDOB                                 | Date/time     | 8    | Local | YYYY-MM-DD     | Client date of birth                               | 2001-07-19  |
| clubMember          | bolClubMember                         | Boolean       | 1    | Local | true/false     | Is the client a club member?                       | true        |
| memberYears         | intMemYears                           | Integer       | 2    | Local | NN             | Loyalty bonus paid to 5, 10, 15, etc. year members | 06          |
| sales               | fpSales                               | FloatingPoint | 8    | Local | NN,NNN.NN      | Total Amount spent                                 | \$12,543.76 |

## Differences between the two styles of data dictionary

For VCE Software Development, you will only be required to use and answer questions about the software design data dictionary. It is good to know about data dictionaries for database design, but it is not within the scope of this course (it is, however, important for students enrolled in VCE Data Analytics).

The major differences between the data dictionaries are as follows.

- The data dictionary related to software design concentrates on *variables* and *data structures* used in programming, while data dictionaries related to database design focus on data being stored in a database.
- There are a number of field heading differences between each template table design (see Tables 4.8 and 4.9).
- One is used by a computer programmer, the other by a database developer.

Data dictionaries are valuable when code needs to be modified later by other programmers, and the purpose of a variable is not clearly understood.

## Object descriptions

There are many different ways of representing objects that are to be incorporated into the design of the software solution. The basic capabilities for any computer software object are identity, properties and attributes. Object-oriented languages have the added capability of inheritance.

Object descriptions are similar to data dictionaries, but can contain information about methods and events that the object contains. Additionally, object descriptions differ between computer languages.

Figure 4.17 shows an example of ways to define objects. There is not one standard for describing this type of design tool. It is important to understand the purpose and function of each design tool.

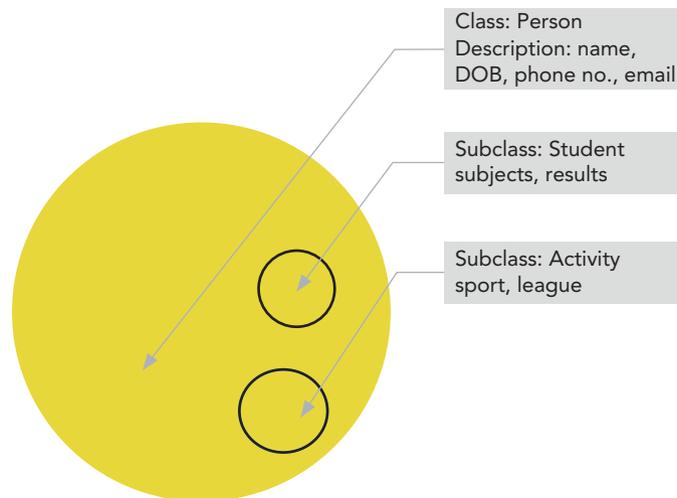
### Examples of an object description

Example 1:

A class Person has name, date of birth, phone number, email address.

A subclass of Person called Student has enrolled subjects, assessment results.

Another subclass of Person called Activity has sport, league.



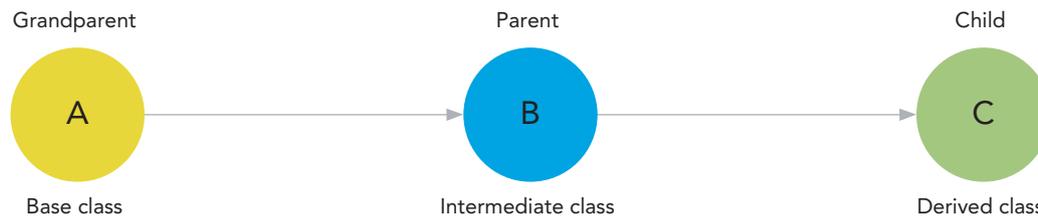
**FIGURE 4.17** Example of an object description

Example 2:

| Object  | Event                                   | Method                                |
|---------|---|---------------------------------------|
| Windows | The user clicks on the tick button      | Void windowClosing(WindowEvent e)     |
|         | The window is opened for the first time | Void windowOpened(WindowEvent e)      |
|         | The window is activated                 | Void windowActivated(WindowEvent e)   |
|         | The window is deactivated               | Void windowDeactivated(WindowEvent e) |
|         | The window is closed                    | Void windowClosed(WindowEvent e)      |
|         | The window is minimised                 | Void windowIconified(WindowEvent e)   |
|         | The window is maximised                 | Void windowDeiconified(WindowEvent e) |

Example 3:

| Object Name: Book |         |   |
|-------------------|---------|---|
| Name              | Type    | Description   |
| title             | String  | The title of the book   |
| author            | String  | The author of the book  |
| isBorrowed        | Boolean | Indicates whether the book is borrowed or not   |
| getTitle()        | Method  | Returns the title of the book   |
| getAuthor()       | Method  | Returns the author of the book  |
| isBorrowed()      | Method  | Returns 'true' if the book is borrowed, 'false' if it is not  |
| borrowBook()      | Method  | Sets the 'isBorrowed' attribute to 'true' if the book is not already borrowed, otherwise returns an error |
| returnBook()      | Method  | Sets the 'isBorrowed' attribute to 'false' if the book is currently borrowed, otherwise returns an error  |



Inheritance is when an object that is derived from an existing base class, a subclass or 'child', acquires all the properties and behaviours of the super-class or 'parent'.

In some languages (e.g. JAVA), the child can only inherit parent behaviours, not grandparent behaviours – that is, no super-super set of behaviours. C++ can inherit both.

**FIGURE 4.18**  
Multi-level inheritance

## Mock-ups and annotated diagrams

As we saw in Chapter 1, mock-ups and **annotated** diagrams show the intended appearance of printed output, on-screen information and interfaces.

Mock-ups are visual representations of a software application's UI, allowing clients and stakeholders to visualise the final design, provide feedback and ensure a shared understanding among the design team.

Annotated diagrams are visual representations with added notes, clarifying design decisions, constraints, and relationships, serving as comprehensive documentation and guidance for developers.

Some useful tools to construct mock-ups and annotated diagrams can be viewed in the weblinks to the right.

## Input-process-output (IPO) charts

As we saw in Chapter 1 (page 11), IPO charts are particularly helpful when designing and developing software, particularly when used in conjunction with context diagrams and data flow diagrams.

A completed IPO chart serves as a blueprint for software development. It helps in coding, testing and debugging by providing a clear overview of the software's functionality.

It also aids in communication among team members and stakeholders, ensuring that everyone has a clear understanding of how the system should work.

In VCE Software Development, you do not have to use software to create your mock-ups. You may use software if you wish, but you may also create your mock-ups by hand using pen and paper. See the following weblinks for free software tools.



**Weblinks**  
Mockplus

Balsamiq Mockups  
(Flash based)

Wireframe CC

Adobe XD CC

SCHOOL-ASSESSED TASK TRACKER

Design brief

Project plan

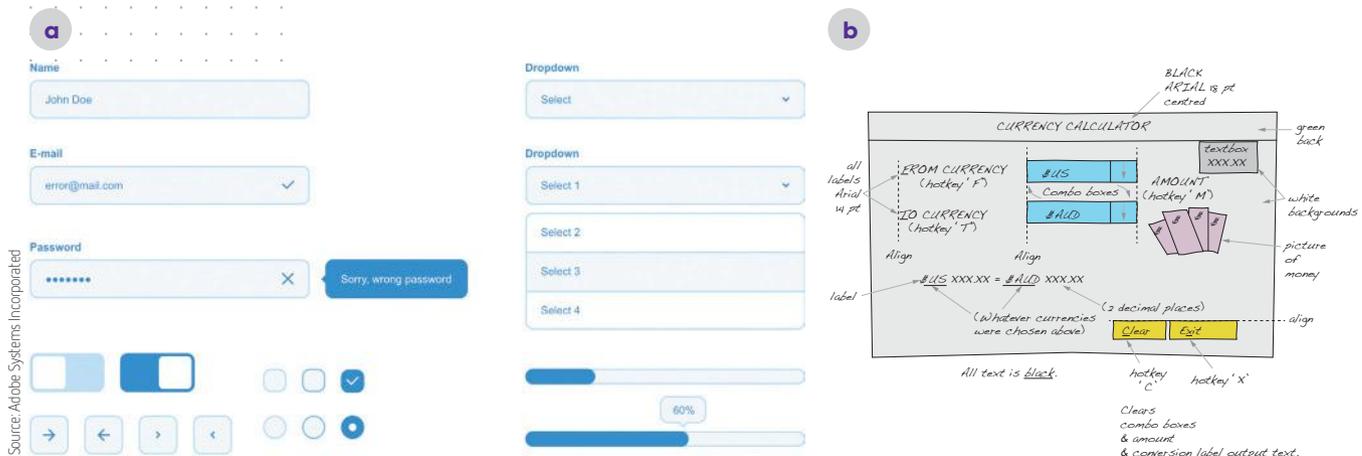
Analysis

Folio of alternative  
design ideas

Development

Evaluation and  
assessment

Final submission



**FIGURE 4.19** Input menu options: **a** Mock-up for an app in Adobe XD CC; **b** Hand-drawn screen interface

The steps to take to create IPO charts in the design stage involve:

- selecting the functional requirement
- determining the input
- describing the process(es)
- describing the output
- creating the chart
- refining and revising the chart.

### Selecting the functional requirement

Referring to the software requirements specification, select a relevant functional requirement that the IPO chart will relate to.

### Determining the input

Identify all the inputs required for the functional requirement. Inputs can be received from external sources, user interactions or other systems. These should have been described in at least one context diagram or data flow diagram relating to the functional requirement.

Inputs should be clearly defined and specific. For example, in a file handling functional requirement, the inputs might include the filename and the data to be written.

### Describing the process(es)

Processes are the actions taken on the inputs to produce the desired output. They can be simple or complex, depending on the functional requirement.

List all the steps or operations that the functional requirement will perform. This includes algorithms, calculations, data processing, decision-making, etc.

In the context of the file handling example, the processes would include checking if the file exists, opening or creating the file, writing data to the file and closing the file.

### Describing the output

Determine what outputs the functional requirement will produce. Outputs can be final products, messages to the user, signals to other systems, changes in user interface, etc.

Outputs should be the result of processing the inputs through the defined processes.

In the file handling example, the output is the status message indicating whether the file has been successfully saved or not.

## Creating the chart

Once all inputs, processes and outputs are identified, they are arranged in a tabular format to create the IPO chart.

The chart typically has three columns, one for each component (input, process, output) and rows for each specific element within these components.

Table 4.10 shows the IPO chart that would be created for the file handling functional requirement.

**TABLE 4.10** IPO chart for giving a bonus for sales over \$10 000

| Input                            | Process   | Output       |
|----------------------------------|---|--------------|
| Sales amount<br>Bonus percentage | <ol style="list-style-type: none"> <li>1. Check if the sales amount is greater than 10,000.</li> <li>2. If true, calculate the bonus by multiplying the sales amount by the bonus percentage.</li> <li>3. If false, the bonus amount is 0.</li> </ol> | Bonus amount |

## Refining and revising the chart

Review the chart for completeness and accuracy. Ensure that all necessary elements are included and correctly described. Check this against the software requirements specification and all previously created diagrams.

The IPO chart may be refined based on feedback or as a better understanding of the system's full set of requirements is developed.

## Pseudocode

**Pseudocode** is intended for human reading rather than machine reading. It appears as informal high-level descriptions of a computer program or other algorithm. A combination of programming terminology and plain English describes algorithms, or instructions, which are easier to understand than programming language code.

Pseudocode describes the logic of the program or algorithm. As you learned in Chapter 1, pseudocode is written in Structured English and contains control syntax such as:

```
IF-THEN-ELSE a choice is made between two alternatives when a
condition is satisfied (this is also known as Boolean)
REPEAT is a loop with a conditional test at the start
FOR-NEXT is a loop with a conditional test at the end
SEQUENCE is when one task is followed by another task
```

Common keywords are used when writing pseudocode. For example:

```
START, END, BEGIN, STOP, DO, WHILE, FOR, UNTIL, REPEAT, IF, THEN,
ELSE, EQUAL, CASE, LESS THAN, GREATER THAN, NOT, OR, TRUE, FALSE,
GET, OPEN, CLOSE, READ, WRITE, END OF FILE, RETURN
```

Combinations of keywords can extend the terms. For example:

```
DO UNTIL, DO WHILE, ELSEIF, END WHILE, END UNTIL, END REPEAT, END IF,
FOREACH, NOT EQUAL, NOT OR, XOR
```

Symbols often carry keyword meaning; there are several that have currency. See Table 4.11 for examples. However, there is no single definition of pseudocode.

**TABLE 4.11** Common symbols used in pseudocode

|                                |                          |
|--------------------------------|--------------------------|
| Assign value to variable       | ←                        |
| Equivalence (exactly equal to) | = =                      |
| Comparison                     | =, ≠, <, >, ≤, ≥, <=, >= |

Other ideas that can be incorporated into your pseudocode include the following:

- sensible variable names.
- comments (where necessary, no need to explain the obvious).
- indentation to assist readability and pairing of keywords.

Examples of pseudocode were provided in Chapter 1, page 12.

#### Preparation for the VCE examination:

While there are no 'rules' with pseudocode, only conventions, previous VCE examinations have established some specific expectations:

- the assign symbol (left-pointing arrow)
- indenting of code.

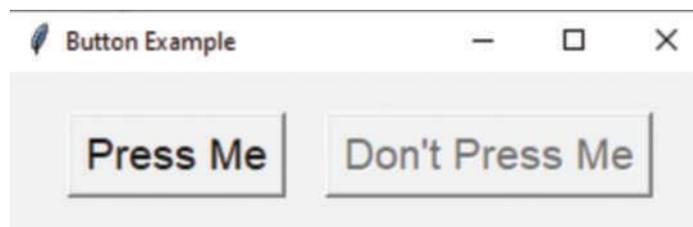
## Characteristics of user experience (UX)

Factors influencing user experience include affordance, interoperability, security (authentication and data protection) and usability. There are design options that can affect each of these factors. The degree to which each factor is implemented depends on the intended purposes of the client. Some useful links to use to consider UX experience are displayed on the left.

There is an interdependence between these factors, so the development of the software product is often a cycle. Each cycle will improve the user experience, or UX, to meet user expectations.

### Affordance

In user experience (UX) design, affordance refers to the design elements that indicate to users how they can interact with an interface or object. Effective affordance uses visual cues and familiar patterns to make actions intuitive, reducing the need for explicit instructions. For instance, a button with a raised appearance suggests it can be pressed, and a scroll bar indicates that content can be moved up or down. These visual signals help users understand the function of UI elements quickly and intuitively.



**FIGURE 4.20** Buttons provide visual cues to afford pressing or not pressing.

Clear affordance guides users towards the correct interactions, enhancing their overall experience. For example, a trash bin icon conveys that dragging a file to it will delete the file, while a floppy disk icon suggests saving. Misleading affordance, however, can lead to user confusion and frustration, highlighting the importance of consistent and well-thought-out design.

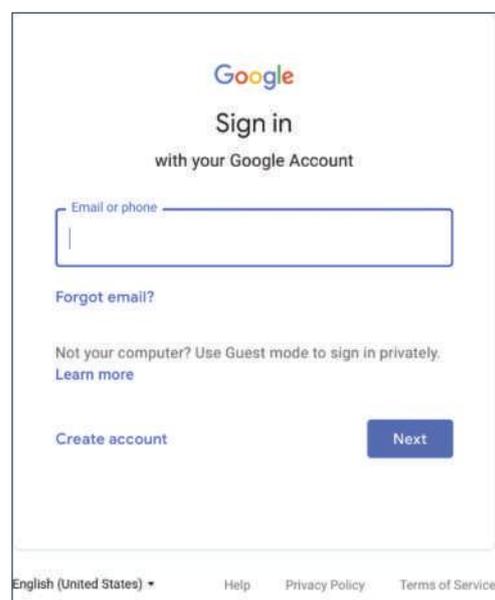
Designers employ affordance by carefully considering the shape, size, colour and placement of UI elements to communicate their purpose effectively. Proper affordance ensures that the interface is self-explanatory, making it easier for users to navigate and interact with the system efficiently, ultimately leading to higher user satisfaction. When designing a system, it is therefore important to consider industry standards and social expectations of how the UI will look and feel.

## Interoperability

Interoperability refers to the capability of different systems, applications and devices to seamlessly work together and exchange information effectively. When systems are interoperable, users can switch between applications or platforms smoothly, leading to increased productivity and reduced cognitive load. It also ensures data consistency across systems, preventing discrepancies and confusion.

Interoperability enhances user flexibility by allowing them to integrate preferred tools and platforms, leading to a more personalised and satisfying experience. For organisations, it reduces costs related to training, support and maintenance, as users do not need to learn new systems from scratch. It also future-proofs applications, making them adaptable to new technologies and standards as they evolve.

In collaborative environments, interoperability facilitates smoother communication and data sharing among team members using different tools, improving teamwork and project outcomes. Examples include cross-platform applications that maintain consistency across different operating systems, API integrations like Google Maps in third-party apps and Single Sign-On (SSO) solutions that streamline user access across multiple systems.



**FIGURE 4.21**  
Single sign-on maintains consistency across different operating systems.

Overall, consideration of interoperability increases the opportunities for developers to create user-friendly, efficient and cohesive digital systems that work together to provide intuitive and effective user experiences.

## Usability

Usability refers to how easily and effectively users can interact with a product to achieve their goals. It includes how quickly users can learn to use the system (learnability), how efficiently and effectively they can complete tasks (efficiency and effectiveness), how easy it is for them to remember how to use the system after a break (memorability) and how pleasant the overall experience is (satisfaction). Usability also considers how well the system helps users recover from errors. High usability ensures a positive user experience, increasing user satisfaction and product adoption.

**TABLE 4.12** Usability factors that improve UX

| Factor          | Description  |
|-----------------|--|
| Learnability    | Learnability assesses how easy it is for new users to accomplish basic tasks the first time they interact with the system. For example, a mobile app with clear icons and straightforward navigation makes it easy for first-time users to understand its functionality. |
| Efficiency      | Efficiency evaluates how quickly users can complete tasks once they have learned the system. For example, a streamlined checkout process on an e-commerce site enhances efficiency by reducing the number of steps needed to complete a purchase.                        |
| Effectiveness   | Effectiveness measures whether users can successfully achieve their objectives using the system. For example, a well-designed search function on a website could help users find relevant information easily using auto-completion or drop-down filters.                 |
| Memorability    | Memorability allows users to return to software after a break and quickly regain familiarity. For example, consistent designs, high levels of affordance and logical organisation in software help users remember how to use it without needing to relearn everything.   |
| Satisfaction    | Satisfaction reflects how pleasant and enjoyable the user experience is. This includes aspects like aesthetic design, interactive feedback and overall user engagement.  |
| Error tolerance | A usable product anticipates common mistakes and provides clear, constructive feedback to guide users back on track. For example, helpful error messages that suggest corrective actions can significantly improve usability.  |

Usability factors also need to be considered when designing the appearance and functionality of a system, as outlined later in this chapter.

## Authentication and data protection

In an increasingly connected world, security considerations for software have become paramount for any business, large or small. The risk of a software solution being compromised is very high, with data breaches having widespread repercussions ranging from loss of reputation to financial loss and possible violation of legal regulations. For example, some countries hold businesses legally responsible for a system that does not comply with laws regarding the storage

and communication of electronic data. Programmers therefore have an obligation to protect the security of data within a system as much as possible. This can involve implementing security features such as encryption for data storage and transfer, as well as putting in place authentication protocols to access elements of the software or the software solution as a whole.

## Encryption

**Encryption** is the process by which **plain text data** is encoded – scrambled – so that it is unreadable by unauthorised applications or people. Once encrypted, this data is referred to as **cipher text data**. Plain text data is encrypted using a key, and the resulting cipher text data can only be decrypted by a person or application that has a decryption key. Encryption is typically used to protect data when it is stored on a computer system, as well as to protect data as it is transferred over unsecured networks.

There are two main algorithms used to **encrypt** data. The first involves using **symmetric key encryption**, where the key used to encrypt the data is the key that is also used to **decrypt** that data. An analogy to describe this is when a code is needed to open a safe, and anyone who has that code can open the safe and access its contents.

Examples of popular symmetric key encryption algorithms are AES, Twofish, Blowfish, 3DES and RC4.

A second type of encryption algorithm is **public key encryption**, also known as **asymmetric key encryption**, where the key used to encrypt data is not the same as the key that is used to decrypt that data. In this algorithm, a **public key** is used to encrypt the data. This key can be used by any person or application and is generally widely known. The data can only be

A common application of encryption is to encrypt messages using a digital signature. Digital signatures give people the confidence that the message they receive is authentic.

The AES encryption algorithm is one of the most popular symmetric encryption algorithms used in the computing industry. Many financial and government institutions use this type of encryption. AES encryption typically uses 128-bit keys, but stronger versions can also use 192-bit and 256-bit.



Rawpixel.com/Shutterstock.com

### THINK ABOUT SOFTWARE DEVELOPMENT 4.1

Bluetooth is commonly used to transfer data between systems, particularly mobile phones. Conduct research on which type of encryption algorithm is typically used for Bluetooth data encryption.

**FIGURE 4.22**

Encryption is a common method of securing data.

#### SCHOOL-ASSESSED TASK TRACKER

Design brief

Project plan

Analysis

Folio of alternative design ideas

Development

Evaluation and assessment

Final submission

decrypted using a **private key**, which is known only to the recipient person or application. Public key encryption is used widely in the computing industry, such as with **Transport Layer Security (TLS)** and **Secure Sockets Layer (SSL)**. This type of encryption is typically used when data is transferred over an open networked environment, such as the internet. A popular public key encryption algorithm is RSA, which is the standard used for data transfers via the internet. Other public key encryption algorithms are Diffie-Hellman, ECC and DSA.

### Implementing encryption algorithms

Encryption algorithms are quite complex to implement and it is critically important that the implementation is accurate and bug-free. A faulty algorithm could, at best, result in data not being able to be decrypted and, at worst, allow data to be decrypted easily by unauthorised people or applications.

Most programming languages have built-in functions or third-party packages that provide the ability to implement encryption without needing to write the encryption algorithm yourself. For example, Python has libraries that include implementations of encryption algorithms such as AES and RSA. Similarly, Visual Basic contains classes that implement 3DES and AES.

An example of how to represent encryption in pseudocode is shown in Figures 4.23 and 4.24.

```

ALGORITHM encryptData(plainTextData)
BEGIN
    ENCRYPT (plainTextData) using 3DES
END

ALGORITHM decryptData(encryptedCipherText)
BEGIN
    plainTextData ← DECRYPT(encryptedCipherText) using 3DES
    RETURN plainTextData
END

```

**FIGURE 4.23** Representing symmetric key encryption in pseudocode

```

ALGORITHM encryptData(plainTextData, publicKey)
BEGIN
    ENCRYPT(plainTextData, publicKey) using RSA
END

ALGORITHM decryptData(encryptedCipherText, privateKey)
BEGIN
    plainTextData ← DECRYPT(encryptedCipherText, privateKey) using RSA
    RETURN plainTextData
END

```

**FIGURE 4.24** Representing public key encryption in pseudocode

## Authentication

Aside from encryption, another method of securing data to reduce the risk of data breaches is to integrate authentication functionality into a software solution. Authentication in computing can have two meanings: one is to prove an identity, the other is to prove that a user has a right to access a software system. For the purposes of VCE Software Development, authentication should be interpreted as the latter: authorisation.

TLS and SSL are security protocols that are used over computer networks. These protocols are typically used by web browsers and email programs to provide encryption over the internet. SSL is the predecessor to TLS; TLS was based on SSL 3.0. This means that TLS should be used over SSL for relevant applications unless the use of SSL is a constraint.

Authentication methods can range from the simplest, single-factor authentication to more complex multi-factor authentication.

### Single-factor authentication

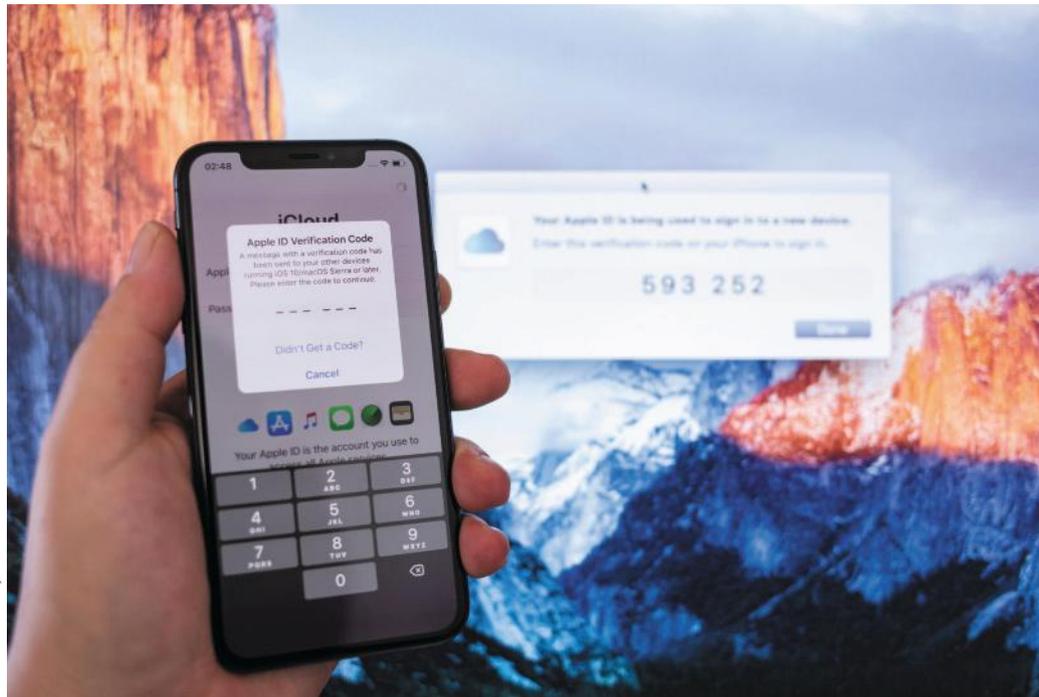
This is the simplest method of authentication, where *something you know*, typically a **username and password**, is required to log in to a software solution. Once logged in, users can have **full access** or **restricted access** to the functionality within the software, depending on the level of access they have been granted. This method of authentication is increasingly considered inadequate due to the prevalence of password-cracking tools available on the internet.

### Two-factor authentication

This type of authentication involves *something you know* as well as *something you have*. **Two-factor authentication**, also known as two-step verification or 2FA, typically involves the user possessing some physical item alongside a password that they must use to access a particular piece of software. One example of two-factor authentication is the process you take to withdraw money from an ATM. To do this, you traditionally must have a physical card in your possession as well as a password, which in this instance is your PIN. Another common example of two-factor authentication is the use of a secondary application to provide a single-use verification code to the user that changes frequently, such as every 60 seconds. The user must provide this alongside a username and password to gain access to the relevant software.

The ease of implementing two-factor authentication is highly dependent on the programming language selected. Some languages have third-party packages and libraries that allow for ‘plug-in’ style inclusion into a software solution, such as Swift and Python, but this is not necessarily the case in all programming languages.

Google Authenticator is an application that implements two-factor authentication for users. It uses a time-based one-time password (TOTP) algorithm for authenticating users of mobile applications.



Cristian Dina/Alamy Stock Photo

#### THINK ABOUT SOFTWARE DEVELOPMENT

4.2

Research time-based one-time password (TOTP) algorithms. How do they work? Aside from Google, what other companies use this algorithm to authenticate users? How secure is this method of authentication?

**FIGURE 4.25** Verification codes are required for two-factor authentication.

#### SCHOOL-ASSESSED TASK TRACKER

- Design brief
  Project plan
  Analysis
  Folio of alternative design ideas
  Development
  Evaluation and assessment
  Final submission

**THINK ABOUT  
SOFTWARE  
DEVELOPMENT**

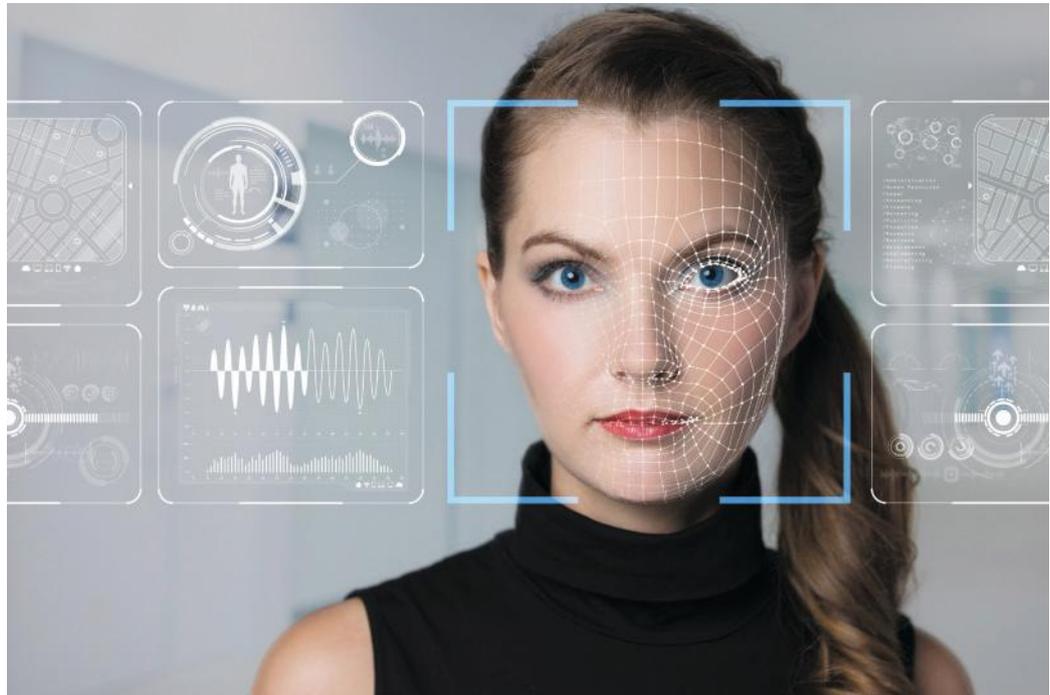
4.3

What are some of the risks of two-factor and multi-factor authentication processes?

## Multi-factor authentication

Multi-factor authentication typically involves a user providing three or more pieces of evidence to prove that they are who they say they are. Typically, these involve *something you know*, *something you have* and *something you are*. It can also involve *somewhere you are*. As in single-factor and two-factor authentication, *something you know* is typically a username and password and *something you have* is typically a physical or digital authenticator that acts as a secondary device providing single-use passwords. The third element, *something you are*, typically uses one or more physical characteristics of the user to authenticate them on the system, such as checking against **biometric data**. Biometric data is data that is obtained from humans, which can include fingerprints, iris scanning, facial recognition, palm prints, hand geometry and DNA matching. It can also involve behavioural characteristics, such as typing speed, key-press patterns, gait patterns and voice recognition. The fourth element, *somewhere you are*, involves location-based factors including the physical location of the user. For example, a user may only be able to access a system if they are on a hard-wired network within an organisation, or within a range of GPS coordinates.

The complexity of implementing multi-factor authentication is tied to the complexity of the biometric data that is required for authentication to be successful. These methods are typically very expensive, and tend to be implemented on systems that are highly sensitive or require tight security.

**FIGURE 4.26**

Biometric data can involve facial or voice recognition.

Metamorworks/Shutterstock.com

# Design principles influencing appearance and functionality

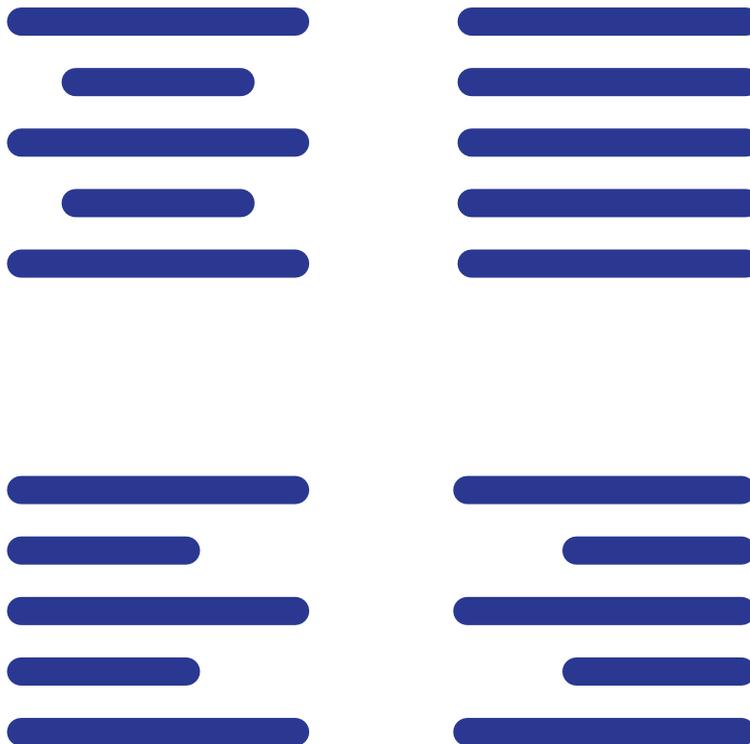
UX design provides relevant and meaningful experiences for all users. This incorporates design principles that influence the appearance and functionality of the user interfaces of a software solution. These principles include alignment, balance, contrast, space, text formatting, usability and navigation.

## Alignment

Alignment is about creating a clean and orderly layout in a design. Proper alignment ensures that elements on the screen are organised and visually connected, which helps users process information more efficiently.

It contributes to a seamless flow, guiding the user's eye across the design in a predictable and coherent way.

The most typical decisions to make in terms of alignment are to do with the justification of texts – left-justified, right-justified, fully justified or centred.



beansky23/Shutterstock.com

**FIGURE 4.27** Different types of text justification

**TABLE 4.13** Text alignment types

| Alignment       | Explanation   |
|-----------------|---|
| Left-justified  | In left justification, the text is aligned to the left margin or edge, creating a straight line on the left side, while the right side remains uneven. This is the most common form of text alignment in Western languages, as it aligns with the natural reading direction (left to right). Left-justified text is easy to read because the starting point of each line is consistent, which helps in eye movement and scanning.   |
| Right-justified | Right justification aligns text with the right margin or edge of the page or container, creating a straight line on the right while the left side is uneven. This type of alignment is less common in English but may be used for stylistic reasons or in languages that read from right to left, such as Arabic, Hebrew or Urdu. For languages that read from left to right, this type of alignment can be harder to read in longer passages as the starting point of each line changes, which may interrupt the natural flow of reading.  |
| Fully justified | Fully justified text aligns with both the left and right margins, creating a clean, block-like appearance. The spacing between words and sometimes letters is adjusted so that each line stretches from edge to edge. This type of alignment can create a neat and orderly look, especially when columns are narrow. In graphic design elements within a website, such as banners or infographics, full justification can be used for text elements to create a balanced, uniform look. However, in software applications, the use of full justification for text alignment is less common compared to left or right alignment, primarily due to considerations of readability and aesthetic consistency across diverse screen sizes and resolutions. |
| Centred         | Centered alignment places text in the middle of the page or container, with an equal amount of space on either side. This creates a symmetrical appearance. This is often used for titles, headings or short pieces of text where emphasis or a formal appearance is desired. It is not ideal for large bodies of text as the irregular starting and ending points of each line can make it challenging to follow and read efficiently.   |

## Balance

Balance refers to the distribution of elements within a design, which can be symmetrical or asymmetrical. Symmetrical balance is when elements are mirrored on either side of a central line, while asymmetrical balance uses different elements that still achieve a visually satisfying effect.

It is important for ensuring that no part of the design overpowers other parts, making the user interface (UI) more comfortable to interact with.

## Contrast

Contrast is used to highlight differences between elements, making them stand out. It can be achieved through colour, size, shape or choice of fonts.

High contrast is essential for readability and accessibility, particularly in text content, but it is also used to draw attention to key elements such as buttons or alerts.

## Space

Space, or white space, refers to the areas of a design that are left empty. This is not just blank space – proper use of space can help reduce clutter, increase readability and create a focal point. It also contributes to a design's balance and helps in grouping or separating elements logically.

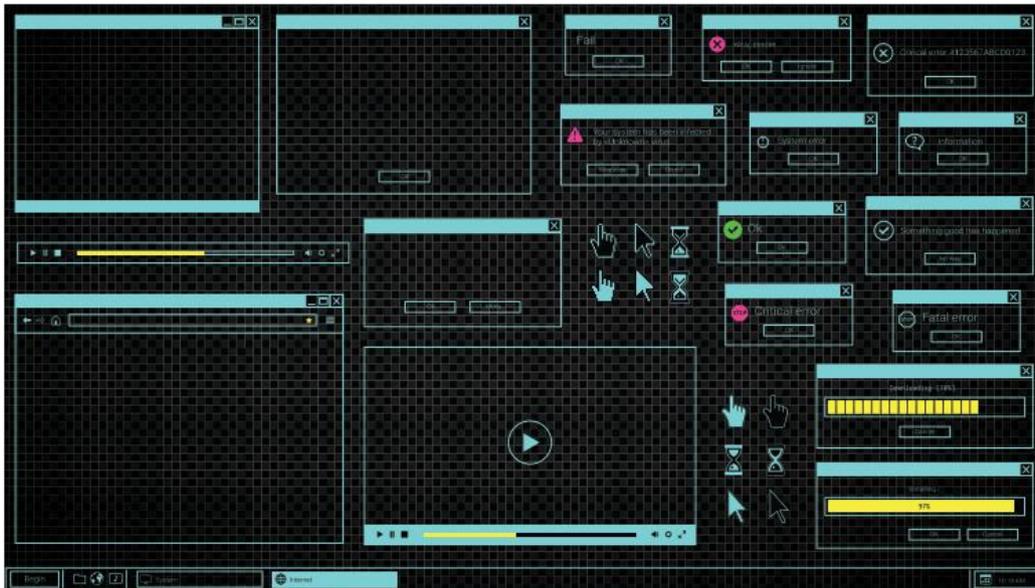
You can check the contrast of colours in a user interface through websites such as [Colors](#).



Weblink  
Colors



**FIGURE 4.28** Balanced user interfaces are often mirrored using columns.



**FIGURE 4.29** High-contrast user interfaces increase readability and accessibility.

## Text formatting

Text formatting involves choosing appropriate fonts, sizes, colours and spacing for text. This helps to increase the readability and accessibility of text.

Good text formatting ensures that content is easy to scan and understand, with hierarchy used to guide the user through the content (e.g. using larger fonts for headings).

## Usability

Usability means making sure that something works well and as efficiently as possible for the user. It is about creating products that are easy to use and understand.

### SCHOOL-ASSESSED TASK TRACKER

|  |  |  |   |                                      |  |   |
|--|--|--|---|--------------------------------------|--|---|
| <input checked="" type="checkbox"/> Design brief | <input checked="" type="checkbox"/> Project plan | <input checked="" type="checkbox"/> Analysis | <input checked="" type="checkbox"/> Folio of alternative design ideas | <input type="checkbox"/> Development | <input type="checkbox"/> Evaluation and assessment | <input type="checkbox"/> Final submission |
|--|--|--|---|--------------------------------------|--|---|

Key aspects include ease of learning, efficiency of use, memorability, error frequency and severity, and user satisfaction.

**TABLE 4.14** Usability aspects explained

| Usability aspects            | Description   |
|------------------------------|---|
| Ease of learning             | How quickly and easily a new user can start effectively using a software product or system. A software application with high ease of learning has intuitive design, clear instructions and a shallow learning curve, allowing users to accomplish basic tasks without extensive training or reference to user manuals.  |
| Efficiency of use            | How quickly users can perform tasks once they have learned to use the software. An efficient system minimises the steps needed to complete a task, has responsive interfaces and allows experienced users to use shortcuts or advanced features to speed up their workflow.   |
| Memorability                 | How easily a user can remember how to use the software after a period of not using it. Good memorability means that returning users can quickly reestablish proficiency with the software without needing to relearn everything from scratch.   |
| Error frequency and severity | How often users make errors while using the software and how severe those errors are. A well-designed software application minimises the likelihood of user errors and ensures that any errors that do occur are not critical and can be easily recovered from. It also involves providing helpful error messages and intuitive mechanisms for users to correct their mistakes. |
| User satisfaction            | How pleasing and satisfying the software is to use. Factors influencing user satisfaction include aesthetic design, the joy of using the software, the extent to which it meets users' needs and expectations, and the absence of frustration during use.   |

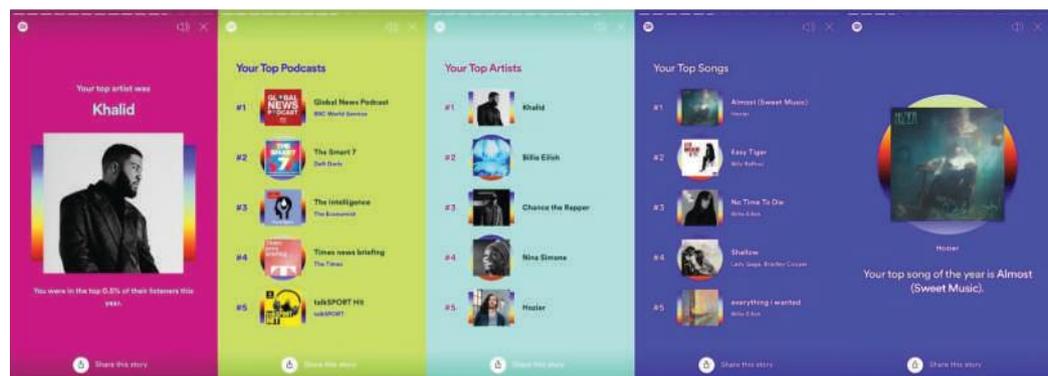
When generating ideas for your software solution, it can be useful to consider successful UX/UI designs that have been created by professional designers. You can read about some of them at the Maze website.

## Navigation

Navigation design deals with how users move through the application or website. It includes the design of menus, links, buttons and any other element that helps users find what they are looking for.

Good navigation should be intuitive, consistent and accessible, enabling users to easily find their way and understand how to get from one point to another.

Navigation should be simple and predictable, and use patterns that software users are already familiar with – for example, having menus at the top of the page or window, breadcrumbs to navigate back to previous screens, and consistent naming and colour-coding of links and buttons.



**FIGURE 4.30** Predictable navigation with familiar elements

Maze.co

# Legal requirements relating to the ownership and privacy of data and information

Data and information can be very valuable, both to you and to others. Decisions and actions can be made in response to knowing such details, but only if you are authorised to use that information. Unauthorised access to personal and sensitive business information is an increasing problem. Since 1998, in Australia, legislation has been developed to manage information effectively. At the federal level, the *Privacy Act 1988* and, at state level in Victoria, the *Privacy and Data Protection Act 2014* and the *Health Records Act 2001* have been enacted. Periodically, this legislation is updated to reflect the technological advances that impact on information. The *Copyright Act 1968* (which includes the *Copyright Amendment (Digital Agenda) Act 2000*) also limits who can use certain information.

Information is the most valuable asset that an organisation owns. In many cases, the time and resources that have been used to collect the data and assemble the information could not be replicated. If the data and information were lost or damaged, it may not be recoverable. An information systems manager would plan to prevent loss from occurring.

There are several key laws relating to the information systems and telecommunications industries. At a federal level, the law concerned with how information about people can be used is the *Privacy Act 1988*. In Victoria, we are especially concerned with the *Privacy and Data Protection Act 2014* and the *Health Records Act 2001*. Combined, these laws govern the collection and use of private information by both government and non-government organisations at both state and federal levels. Employers and government agencies have a legal responsibility to ensure that these laws are implemented within their organisations. In addition, organisations must make employees and customers aware of their rights, as well as their responsibilities, in relation to these laws.

## Privacy Act 1988

In the mid-1980s, the federal government attempted to introduce an 'Australia Card'. This proposal was met with overwhelming resistance from the public, and eventually dropped. In its place, the pre-existing tax file number (TFN) system was updated. There were many concerns about how the federal government might use tax file numbers, especially regarding the release of confidential tax information, or matching data from different government departments.

Originally, the *Privacy Act 1988* only dealt with the handling of data by government agencies. Many people criticised these limitations because it seemed that private organisations were not required to apply even the most basic of safeguards regarding the data they collected. Even worse, there were no regulations preventing non-government organisations from collecting data by any method and using it for any purpose without consent. The rapid growth of electronic transactions, especially over the internet, led many people to demand some sort of legal protection from those who might gather data about their browsing habits. The government was keen to encourage the development of electronic commerce, while

'These are the most significant changes to privacy laws in over 25 years and affect a large section of the community. The world has changed remarkably since the late 1980s when the Privacy Act was first introduced, and so the changes were required to bring our laws up-to-date with contemporary information handling practices, including global data flows,' said then Australian Privacy Commissioner Timothy Pilgrim in 2014 when the Privacy Amendment (Enhancing Privacy Protection) Bill came into effect.

Australian Government, Office of the Australian Information Commissioner

### SCHOOL-ASSESSED TASK TRACKER

 Design brief

 Project plan

 Analysis

 Folio of alternative design ideas

 Development

 Evaluation and assessment

 Final submission

**THINK ABOUT  
SOFTWARE  
DEVELOPMENT**

4.4

Many organisations have a privacy policy listed on their website. Find out what is covered by your school's privacy policy. What information might the school have about students that should not be made publicly accessible?

protecting the confidentiality of consumers and increasing public confidence in electronic transactions. As a result, several amendments were incorporated into the *Privacy Act 1988*. These were the most significant changes to have been made to privacy laws since the inception of the legislation.

There have been several additional powers included within this Act since 1988, but its main purpose has remained unchanged. The *Privacy Act 1988* was amended by the Privacy Amendment (Enhancing Privacy Protection) Bill 2012, which came into effect on 12 March 2014.

### Application of the Privacy Act

The Privacy Act applies to both electronic and manual or conventional forms of data gathering and handling by private organisations. The Act also has provisions specifically addressing the use of personal data for direct marketing via email, which can only be used with the consent of the individual concerned. It also extends to general privacy issues regarding workplace email. The Act encompasses businesses with an annual turnover of \$3 million, all private health services that store health records, businesses that trade in personal information, and those organisations that choose to opt in.

Individuals also have rights under the Act, which makes for provisions on how their personal information is collected. The Act defines personal information as being:

→ information or an opinion about an identified individual (or an individual who is reasonably identifiable), whether the information or opinion is true or not, and whether the information or opinion is recorded in a material form or not.

The amended Act defines personal information as including an individual's:

- name and address
- signature
- telephone number
- date of birth
- medical records and health information
- bank account details
- photos and videos
- biometric and genetic information
- philosophical beliefs
- likes and dislikes
- opinions or commentary about a person
- racial or ethnic origin
- memberships of political associations
- professional or trade associations or trade unions
- religious beliefs or affiliations
- criminal record
- sexual orientation or practices.

Since the introduction of the updated *Privacy Act 1988*, organisations have had to review the way they handle customer information and have had to update their technologies and their security processes to ensure that they comply with the new legislation.

The Act prescribes severe penalties for serious and repeated interferences with privacy, which can result in criminal prosecution and/or fines of up to \$340 000 for individuals and \$1 700 000 for public and private organisations.

## Australian Privacy Principles

There are 13 Australian Privacy Principles (APPs), described in Table 4.15. For the purposes of Software Development, the focus is on APPs 1, 3, 6, 8, 9 and 11. These are marked with an asterisk.

The amount specified in the Privacy Act is not variable. That is, the fine of \$340 000 for individuals and \$1 700 000 for public and private organisations does not change. The amount will only change if the Act is changed. In Victoria, however, the fines are variable and tied to penalty units with an amount that is adjusted on 1 July each year.

**TABLE 4.15** Australian Privacy Principles

| Principle  | Description  |
|--|--|
| *APP 1 – Open and transparent management of personal information | Organisations must have a clearly articulated and accessible privacy policy. This policy should be freely available and easy for individuals to find. It should explain how and why the organisation collects, holds, uses and discloses personal information. The policy must also outline how an individual can access and correct their personal information held by the organisation.  |
| APP 2 – Anonymity and pseudonymity                               | If an individual chooses not to provide their personal information, the organisation should not enforce identification unless it is necessary for the specific service. Where practicable and lawful, individuals have the option to use a pseudonym instead of their actual name. A pseudonym is a name or other identifier that an individual uses instead of their actual name.         |
| *APP 3 – Collection of solicited personal information            | Organisations can only collect personal information that is reasonably necessary for their functions or activities. They must also notify individuals about the purpose of collection and other relevant details. Individuals should be aware of why their information is being collected and how it will be used.   |
| APP 4 – Dealing with unsolicited personal information            | Organisations must destroy or de-identify the information as soon as practicable if the information was not requested or asked for. There should be no use or disclosure of unsolicited personal information unless it is permitted by law.  |
| APP 5 – Notification of the collection of personal information   | This requires organisations to inform individuals at or before the time of collecting their personal information about key details, including that their information is being requested, why it is being requested, potential recipients of the information, and the legal requirements or consequences if the information is not provided.  |
| *APP 6 – Use or disclosure of personal information               | Organisations are allowed to use or disclose personal information for the primary purpose it was collected. They may also use or disclose personal information for secondary purposes if the individual has consented or if it is otherwise permitted by law.  |
| APP 7 – Direct marketing   | Organisations must not use or disclose personal information for direct marketing purposes without the individual's consent. They should provide a clear and easy way for individuals to opt out of direct marketing communications.  |
| *APP 8 – Cross-border disclosure of personal information         | Before disclosing personal information to an overseas recipient, organisations must take reasonable steps to ensure that the recipient will protect the information in a manner consistent with the Australian Privacy Principles. Individuals should be informed about the likelihood of their information being sent overseas and the countries or regions where recipients are located. |



**TABLE 4.15** Australian Privacy Principles (continued)

| Principle  | Description  |
|--|--|
| *APP 9 – Adoption, use or disclosure of government related identifiers | Organisations are generally prohibited from adopting, using or disclosing government related identifiers as their own. A government related identifier is a unique identifier assigned by a government agency, such as a tax file number or driver's licence number.   |
| APP 10 – Quality of personal information                               | Organisations are required to take reasonable steps to ensure that the personal information they collect is accurate, up to date and complete. This includes verifying the information at the time of collection and regularly reviewing and updating it if necessary. Individuals have the right to request correction of their personal information if they believe it is inaccurate, incomplete or out of date, and organisations must respond to such requests promptly.                                       |
| *APP 11 – Security of personal information                             | Organisations are required to take reasonable steps to protect the personal information they hold from misuse, interference, loss, unauthorised access, modification or disclosure. This includes implementing security measures such as encryption, access controls and secure storage to prevent unauthorised access and data breaches. Organisations must also have processes in place to detect and respond to data breaches, including notifying affected individuals and relevant authorities when required. |
| APP 12 – Access to personal information                                | Individuals have the right to request access to their personal information held by an organisation. Organisations must provide access to the requested information within a reasonable time and at no cost.  |
| APP 13 – Correction of personal information                            | Individuals have the right to request corrections to their personal information if they believe it is inaccurate, out of date, incomplete or misleading. Organisations must take reasonable steps to correct the information upon receiving such a request, unless there are valid reasons not to do so. If an organisation refuses to correct the information, they must provide reasons for the refusal.   |

## Privacy and Data Protection Act 2014

The *Privacy and Data Protection Act 2014* (PDPA) was introduced by the Victorian Government. It replaced the *Information Privacy Act 2000* and the *Commissioner for Law Enforcement Security Act 2005*. The PDPA is intended to strengthen the protection of personal information and other data held by Victorian government agencies, including local councils and contractors working for the state government.

Under the PDPA there is a single privacy and data protection framework. The PDPA uses its own Information Privacy Principles (IPPs), and organisations are obliged to act in accordance with the IPPs. As a result of the PDPA, a Privacy and Data Protection Commissioner has been established.

### Information Privacy Principles

As discussed in the previous section, the amendments to the *Privacy Act 1988* in 2014 introduced new Australian Privacy Principles (APPs). It was anticipated that the current Victorian Information Privacy Principles (IPPs) would be replaced with new principles based on the Australian Privacy Principles (APPs). However, this has not happened, so the *Victorian Privacy and Data Protection Act 2014* continues to use the IPPs.

APPs are discussed further on pages 295–6.

**TABLE 4.16** The 10 Information Privacy Principles (IPPs)

| IPP | Description   |
|-----|---|
| 1   | <b>Collection of personal information</b><br>When an organisation collects information, it should only collect the information it needs. The organisation should inform people that their information is being collected.             |
| 2   | <b>Use and disclosure of personal information</b><br>When an organisation uses and discloses personal information, it is only for the purpose that it was collected for, or for a secondary purpose that you would reasonably expect. |
| 3   | <b>Data quality</b><br>Ensure that the information collected is accurate, complete and up to date.  |
| 4   | <b>Data security</b><br>Information must be protected from misuse, loss, unauthorised access, modification or disclosure. Reasonable steps must be taken to destroy or de-identify personal information that is no longer needed.     |
| 5   | <b>Openness</b><br>The organisation needs to be transparent about what it does with information. Non-compliance will result in a maximum penalty for a body corporate of 3000 penalty units and 600 penalty units for an individual.  |
| 6   | <b>Access and correction</b><br>When an organisation collects information, it should allow people to see the information it collects about them and provide them with the opportunity to correct it if it is inaccurate.              |
| 7   | <b>Unique identifiers</b><br>Use of unique identifiers, usually a number, is only allowed where an organisation can demonstrate that the assignment is necessary to carry out its functions efficiently.                              |
| 8   | <b>Anonymity</b><br>Where possible, people supplying information should be given the option of not identifying themselves.  |
| 9   | <b>Transborder data flows</b><br>If your personal information travels outside Victoria, your privacy protections must travel with it.   |
| 10  | <b>Sensitive information</b><br>Organisations need to ensure that they do not collect sensitive information about people, such as their religion, political views or criminal record, without checking the applicable laws.           |

In VCE Software Development, the relevant IPPs that are examinable are: 1, 2, 4, 5, 7, 9 and 10.

Penalty units define the amount that needs to be paid for offences in Victoria. Generally, the legislation does not specify the monetary amount, but does specify the penalty unit. Each year, the penalty unit is specified. For example, from 1 July 2023 to 30 June 2024, one penalty unit was worth \$192.31. The rate for penalty units is indexed each financial year so that it is raised in line with inflation. Changes to the value of a penalty unit take effect on 1 July each year.

#### THINK ABOUT SOFTWARE DEVELOPMENT

4.5

Stevie is a student support officer at a Victorian government school. He has access to student and parent personal details. He has been approached by an external organisation to 'sell' these details in exchange for new computer equipment for the school.

- 1 Identify key legislation that Stevie should consider before deciding whether to provide the information to the external company.
- 2 What are Stevie's ethical responsibilities to the students, parents and the school?

#### SCHOOL-ASSESSED TASK TRACKER

Design brief
  Project plan
  Analysis
  Folio of alternative design ideas
  Development
  Evaluation and assessment
  Final submission

# 4

## CHAPTER SUMMARY

### Essential terms

**accessibility** how easily the software can be used by those who experience disabilities

**accuracy** the absence of mistakes or errors

**annotate** add comments to a document or diagram

**assistive technology** any device or system that is designed for individuals who would otherwise find the task difficult or impossible

**asymmetric key encryption** a cryptographic system that uses a pair of keys – a public key and a private key – for secure data encryption and decryption

**attractiveness** how pleasing something is to the viewer

**biometric data** information derived from the unique physical characteristics of an individual, such as fingerprints, facial recognition or iris patterns, used for identification and authentication

**cipher text data** data that has been transformed from plain text into an unreadable format using encryption

**clarity** ease of understanding

**communication of message** the process through which meaning is transferred

**completeness** nothing left out

**convergent thinking** involves coming up with a single, well-established answer to a problem

**data** raw, unorganised facts, figures and symbols fed to a computer during the input process; data can also refer to ideas or concepts before they have been refined

**data dictionary (software design)** used to plan storage structure; provides specifications of variables, arrays and GUI objects

**decrypt** the process of converting encrypted data back into its original, readable form using a decryption key

**design ideas** conceptual solutions or creative approaches intended to address a specific problem or requirement in a design process

**differential backup** a type of backup that saves only the data that has changed since the last full backup

**divergent thinking** involves exploring many possible solutions using spontaneous, free-flowing techniques

**effectiveness** produces the expected result

**effectiveness of a solution** how well the software works to produce the desired result

**efficiency** economic use of resources with minimum waste

**efficiency of a solution** whether the result is produced quickly and simply

**encrypt** to convert data into a coded format to prevent unauthorised access

**encryption** the process of converting plain text data into an unreadable format to protect it from unauthorised access

**evaluation** an assessment of whether a solution achieves the goals for which it was originally designed; not the same as testing

**evaluation criteria** rules set out during design that include effectiveness and efficiency criteria; based on the solution's requirements, which were defined during analysis

**file management plan** a plan that includes all aspects of handling documents, including storage, retrieval, backups, archiving and security

**full access** unrestricted permission to view, modify and manage all resources and information in a system

**full backup** a complete copy of all data in a system at a given point in time

**HDD** hard disk drive; a data storage device that uses spinning disks to read and write data

**incremental backup** a backup that saves only the data that has changed since the last backup of any type

**information** knowledge about a person, place, event or thing

**maintainability** the ease with which a system or component can be maintained to correct defects, improve performance or adapt to a changed environment

**off-site storage** physical or cloud-based storage of data and backups located outside an organisation's premises

**on-site storage** physical storage of data and backups located within the premises of an organisation

**plain text data** data in its original, readable form, without encryption

**private key** a secret key used in asymmetric encryption that is kept confidential and used for decrypting data encrypted with the corresponding public key

**pseudocode** code that designs algorithms in a clear, human-readable, language-independent format

**public key** a key used in asymmetric encryption that is publicly available and used to encrypt data or verify a digital signature

**public key encryption** an encryption method that uses a pair of keys: a public key for encryption and a private key for decryption

**readability** the ease of understanding the text

**relevance** appropriate meaning in the context of the discussion

**restricted access** limited permission to view, modify or manage resources and information in a system

**Secure Sockets Layer (SSL)** a protocol for establishing secure, encrypted connections over the internet

**SSD** solid state drive; a data storage device that uses flash memory to store data, offering faster access speeds than HDDs

**symmetric key encryption** a cryptographic system that uses the same key for both encryption and decryption

**timeliness** occurring at the right time

**Transport Layer Security (TLS)** a protocol that provides secure communication over a computer network, succeeding SSL

**two-factor authentication** a security process that requires two forms of identification before granting access

**universal design** designing products that can be used by people with a wide range of abilities and disabilities

**usability** ease of use to achieve specified goals in terms of efficiency, effectiveness and satisfaction

**username and password** a combination of a user identifier and a secret set of characters used to authenticate and grant access to a system or service

**version control** a system for managing changes to files, tracking modifications and maintaining a history of versions

## Important facts

- 1 Putting in place file management strategies allows for efficiency in software development.
- 2 Documents should be sensibly and meaningfully named.
- 3 Filenames should include dates that allow for sorted order, such as YYYY-MM-DD.
- 4 Cloud storage allows for shared work to occur on projects.
- 5 Full backups copy all data from the system to a backup location.
- 6 Differential backups only copy data that has changed since the last full backup.
- 7 Incremental backups only copy data that has changed since the last backup, of any type.
- 8 On-site storage allows for quick access and recovery.
- 9 Off-site storage protects against physical disasters such as fires or floods.
- 10 A combination of on-site and off-site storage provides a robust backup strategy.
- 11 Disposing of files requires file scrubbing to make the data unreadable.

- 12 There are different levels of disposing/erasing data.
- 13 The simplest method of disposing of a drive is to physically disable or destroy it.
- 14 Archiving preserves a record of the file in medium- to long-term storage.
- 15 Archived data may not be easily accessible to a regular user.
- 16 File organisation and storage media can affect access to data.
- 17 Frequently accessed files need to be placed in an easily accessible location.
- 18 Convergent thinking involves coming up with a single, well-established answer to a problem.
- 19 Convergent thinking results in design ideas that are based on other, proven ideas.
- 20 Divergent thinking is more creative than convergent thinking.
- 21 Many possible solutions are explored using divergent thinking, using spontaneous, free-flowing techniques.
- 22 Combining convergent and divergent thinking skills is likely to produce the most efficient and effective design ideas.
- 23 Mood boards allow designers to communicate the general feeling of a solution.
- 24 Mood boards help to establish the creative direction and potential look and feel of a software solution.
- 25 Brainstorming allows for non-judgemental, spontaneous design ideation.
- 26 Ideas should not be rejected too soon when brainstorming.
- 27 Mind mapping involves quickly generating and linking ideas together.
- 28 Mind mapping complements brainstorming.
- 29 Graphic organisers provide visual methods of organising ideas.
- 30 Graphic organisers can extend upon mind-mapping techniques.
- 31 Sketches and annotations are fast tools to brainstorm and communicate ideas.
- 32 Evaluating design ideas should use criteria based on the software requirements specification.
- 33 Evaluation is a systematic method used to select preferred designs.
- 34 Evaluation should be considered in terms of efficiency and effectiveness.
- 35 Criteria should be measurable.
- 36 Methods to express software designs include data dictionaries, mock-ups, object descriptions, IPO charts and pseudocode.
- 37 No single method is superior to another when expressing design ideas.
- 38 User experience (UX) factors include affordance, interoperability, security and usability.
- 39 Design principles that influence appearance and functionality include alignment, balance, contrast, space, text formatting, usability and navigation.
- 40 At a federal level, the law concerned with how information about people can be used is the *Privacy Act 1988*.
- 41 As part of the Privacy Act, the Australian Privacy Principles (APPs) were devised to set out the standards, rights and obligations for collecting, handling, holding, accessing, using, disclosing and correcting personal information.
- 42 In VCE Software Development, relevant APPs that are examinable are: 1, 3, 6, 8, 9 and 11.
- 43 The *Privacy and Data Protection Act 2014* (PDPA) was introduced by the Victorian Government. It replaced the *Information Privacy Act 2000* and the *Commissioner for Law Enforcement Security Act 2005*.
- 44 In VCE Software Development, relevant IPPs from the PDPA that are examinable are: 1, 2, 4, 5, 7, 9 and 10.



## Managing files

- 1 Why do files need to be organised? Explain how a directory hierarchy might work.
- 2 You are asked for advice on how to keep your software application and data safe and secure. What would you recommend?
- 3 How can an HDD be securely wiped? How is this different from just deleting a file?

## Generating design ideas

- 4 What is the difference between convergent thinking and divergent thinking?
- 5 What is the most important rule to follow when you are conducting a brainstorming session?
- 6 Why is it important to show your solution to the end user?
- 7 Explain how you would use mind mapping to follow up a brainstorming session.
- 8 Provide one example of a historical breakthrough. What technique was used to get to that breakthrough?

## Evaluating efficiency and effectiveness

- 9 What is the difference between effectiveness and efficiency?
- 10 Explain how an improvement in efficiency may cause a decrease in effectiveness.
- 11 Describe how efficiency and effectiveness might be measured. How might these quantities be measured?
- 12 The statement 'Testing is not evaluation' suggests that evaluation is not about test tables and input-output expected results. Identify some of the quantities that evaluation will measure.
- 13 When should evaluation take place?
- 14 Explain the concept of affordance in UX design and provide examples of how it can enhance the intuitiveness of an interface.
- 15 Discuss the importance of interoperability in UX design and describe how it can improve user efficiency and satisfaction.
- 16 Define usability in the context of UX and explain its key attributes. How do these attributes contribute to a positive user experience?
- 17 Explain how contrast and space can be utilised in UX design to enhance readability and visual hierarchy. Provide examples of how improper use of these principles can impact user experience.

## Legal requirements

- 18 Explain why an organisation must comply with legal requirements.
- 19 Briefly summarise the role and scope of the three key laws affecting privacy of information.
- 20 Why have these laws been introduced?



## TEST YOUR KNOWLEDGE

- 21 If you believe that the privacy of your information has been breached by the Australian Taxation Office, to whom can you complain?
- 22 What are the penalties for breaches of the *Privacy and Data Protection Act 2014*?
- 23 For each of the following breaches of privacy, suggest which privacy law would apply.
- a You find that your employer has published your tax file number on the internet.
  - b Medical records are found at the tip.
  - c A bank refuses to give you a loan because the manager claims your credit record is poor, when it should actually be very good.
  - d A consultant working for the Victorian Government passes on your VCE results to a friend without your permission.
  - e A website you visit asks for personal information from you, but does not display its privacy policy.



- Riley is an IT manager responsible for the data backup strategy of her company's servers. Her company performs a full backup every Sunday night. From Monday to Saturday, she needs to choose between incremental and differential backups to ensure data integrity and minimise storage usage and backup time.

Monday: Riley creates a backup of the data changed since the last full backup.

Tuesday: Riley wants to back up only the data changed since Monday.

Wednesday: Riley decides to back up all data changed since Sunday.

Based on the scenario above, which type of backups (incremental or differential) should Riley choose for Monday, Tuesday and Wednesday? Explain your choices and the advantages or disadvantages of each backup method in this context.

- Kestrel is designing a system to manage an online store where customers can purchase products. Each product in the store has a name, a price, a unique product ID and a stock quantity indicating how many units are available.

Create an object description Kestrel could use to represent the Product class. The table should include properties and methods necessary to manage the product's information and stock levels.

- Match the evaluation criteria with the method of evaluation. Identify which criterion is efficiency and which is effectiveness.

| Evaluation criteria                                       | Method of evaluation  |
|---|---|
| The system should have over 95% availability              | Opinion interviews of 500 users at random                       |
| User survey should achieve a very high satisfaction level | Operators record time on task and other duties                  |
| 20% less operator time with adoption of the new system    | Online survey of users  |
| Usability of the system should be rated highly            | A log of faults records the time and duration of system failure |



## APPLY YOUR KNOWLEDGE

- 4 A large Australian financial institution sends a policy document to all clients on how the company handles personal information. The document includes the following statements:

### Your personal information

Personal information is information or opinions that allows others to identify you. It includes your name, age, gender, contact details, as well as your health and financial information. We will act to protect your personal information in accordance with the National Privacy Principles or an industry privacy code. We collect personal information to provide you with the products and services you request as well as information on other products and services offered by or through us.

### How we use your personal information

Personal information may be used and disclosed within the company to administer our products and services, as well as for prudential and risk management purposes and, unless you tell us otherwise, to provide you with related marketing information. We also use the information we hold to help detect and prevent illegal activity. We cooperate with police and other enforcement bodies as required or allowed by law.

- a Explain why the document is sent to clients.
- b Provide reasons for including these statements within the document.

Also included in the document is the following statement:

We disclose relevant personal information to external organisations that help us provide services. These organisations are bound by confidentiality arrangements. They may include overseas organisations.

- c How should the organisation ensure that a client's privacy is protected if they need to send personal details overseas?
- 5 The Children's Singing and Dancing Academy is an organisation that runs singing, dancing and acting classes specifically aimed at school-age children. The company offers classes after school on most weeknights and on weekends in various locations around Melbourne. Children from many suburbs participate in this extracurricular activity. The company is a not-for-profit organisation set up specifically to broaden children's interest in the performing arts. The company has a website and advertises its classes and locations through this medium. Pictures taken of students during their end-of-semester performances are used for advertising the company. The company relies on technology usage by creating brochures, updating its website and storing clients' data. Although the data stored is primarily about the children, data on the parents/guardians of the children is also stored.
- a What particular data on children is stored at the Children's Singing and Dancing Academy?
  - b What particular data on the parents/guardians is stored at the Children's Singing and Dancing Academy?
  - c Why does the company need to store data on its clients?
  - d What does the Children's Singing and Dancing Academy need to do to ensure that it is compliant with the *Privacy Act 1988*?

## PREPARING FOR

# Unit

# 3

### OUTCOME 2

On completion of this unit, the student should be able to document a problem, need or opportunity, formulate a project plan, document an analysis, and generate design ideas and a preferred design for creating a software solution.

To achieve this outcome you will draw on key knowledge and key skills outlined in Area of Study 2. This outcome concludes the analysis and design stage of the problem-solving methodology for Unit 3, Outcome 2.

## Outcome milestones

- 1 Develop a project plan (Gantt chart). This plan includes all known dates and tasks for Unit 3 and Unit 4. (SAC and SAT dates and tasks will be available from your teacher.) The project plan outlines tasks to be completed, which will include duration, sequencing, milestones and dependencies.
- 2 Create a design brief to describe the problem and its proposed users and the feasibility and originality of the proposed solution.
- 3 Compile an SRS report. The SRS provides an analysis that defines details of the software solution purpose, requirements, constraints and scope, functional and non-functional features, and analytical tools diagrams (data dictionary, context diagram, use case diagram, data flow diagram, flow chart).
- 4 Design the software solution folio. The design folio includes two or three feasible alternative design ideas, which are rough sketches or mock-ups of development strategies, without much detail. Choose one design idea using criteria derived from the SRS, then add further details to create a complete preferred design of your software solution.

## Steps to follow

- 1 Develop the SRS, including:
  - a purpose and audience of the SRS
  - b user characteristics (general characteristics of eventual users)
  - c environment characteristics (technical description of environment within which the solution will operate)
  - d scope of the solution
  - e functional requirements
  - f non-functional requirements
  - g constraints
  - h analytical tools diagrams
    - i use case diagrams (UCDs)
    - ii context diagrams
    - iii data flow diagrams (DFDs).

- 2** Use appropriate software or manual design techniques to design the software solution. Develop two or three feasible alternative designs, and add further details of your preferred design with a total of three or four design strategies. Your design folio should include information about:
  - a** mock-ups of each alternative design, using manual or software methods
  - b** relevant and appropriate evaluation criteria to select the preferred design; these criteria are derived from the SRS
  - c** how the selection of the nominated preferred design from an evaluation of each alternative design was achieved
  - d** the final preferred design, with further developed complete details; this may include sketches and annotations, mock-ups with colours and fonts, layout specifications for input and output screens, screen sizing, screen layout alternatives (notebook, smartphone, tablet), pseudocode and data dictionary descriptions.
- 3** Develop a proposed project plan (Gantt chart). Note: This will be used as a progress report and will be resubmitted with changes for project evaluation in Unit 4, Outcome 1.
  - a** The Gantt chart will include all known dates and tasks, duration, sequencing, milestones and dependencies. A critical path will be indicated. Unit 3 and 4 dates will be provided by your teacher.
  - b** Problem-solving methodology stages will be included over both parts of the project. Analysis, design, development and evaluation stages must be explicitly stated and breakdown tasks included with estimates of duration, sequence and resources.
  - c** The project duration and tasks must include both Unit 3, Outcome 2 and Unit 4, Outcome 1.

---

## Documents required for assessment

- 1** A design brief that describes the problem and its proposed users as well as the feasibility and originality of the proposed solution.
- 2** An analysis of the problem, need or opportunity in the form of a software requirements specification (SRS) that defines the purpose, requirements, constraints, scope, functional and non-functional features, and analytical tools diagrams as detailed in Unit 3, Outcome 2
- 3** A folio of two or three design ideas and evaluation criteria and the detailed design specifications of the preferred design
- 4** A proposed project plan (Gantt chart) for monitoring project progress in a written report or an annotated visual plan

---

## Assessment

Your teacher will provide you with a more detailed set of assessment criteria before you begin this assessment.

The SAT (comprising Unit 3, Outcome 2 and Unit 4, Outcome 1) will contribute 30 per cent to your Software Development study score.



## Unit 3 exam practice questions

For Unit 3, you need to ensure that you have a sound understanding of the required key knowledge as well as the ability to demonstrate the key skills. When revising, it is useful to relate the key knowledge to real-world applications to enhance comprehension. Real-world applications provide concrete examples of abstract concepts, making them more tangible and easier to understand. For example, demonstrating how a specific sorting algorithm is used in a widely used application like a search engine can clarify its utility and efficiency.

Familiarise yourself with the syntax and structure of pseudocode used in the study. Pseudocode often varies slightly between sources, so understanding the conventions used in past VCE exams is important.

Thoroughly read and understand all questions, including the case study if you are provided with one. Underline command terms and key phrases in the question to help you understand what is being asked, especially for theory and short-answer sections.

For multiple-choice questions, learn to use the process of elimination in questions where you are unsure of the answer. Often, you can narrow down the choices to two, increasing your chances of selecting the correct answer.

Short-answer questions require clear and concise responses. Focus on directly answering the question, avoiding unnecessary details. Demonstrate your knowledge by using appropriate technical terms. However, ensure that you fully understand the terms you use.

Practise managing your time effectively. Allocate time to each section based on its weight in the overall grade and your proficiency in that area. If time permits, review your answers, focusing on sections you found challenging. This can help catch and correct any mistakes or oversights.

### Multiple-choice questions

#### Question 1 (1 mark) KK3.1.2, KK3.2.6

A startup is developing a mobile app to facilitate online tutoring. The app must support video calls and have a response time of less than two seconds under normal usage conditions. Additionally, the project must be completed within a six-month time frame due to funding restrictions. Which of the following represents a non-functional requirement?

- A. Support for video calls
- B. Online tutoring facilitation
- C. Completion within six months
- D. Response time of less than two seconds

#### Question 2 (1 mark) KK3.1.3

Daniel is tasked with designing a complex inventory management system. As part of this task, he needs to show the relationships between various inventory objects, their attributes and their methods. To effectively represent this information, Daniel should use:

- A. data dictionaries and object descriptions.
- B. object descriptions and IPO charts.
- C. mock-ups and data dictionaries.
- D. mock-ups and IPO charts.



## EXAM PRACTICE

### Question 3 (1 mark) KK3.2.3

For a sports league management application, the developers need to store detailed information about each player, including name, team, position and statistics, and also maintain a schedule of matches where each match includes details such as date, teams involved and scores. Which data structures would be most appropriate for storing this information?

- A. Records for player information and a two-dimensional array for match schedules
- B. A one-dimensional array for player information and records for match schedules
- C. A two-dimensional array for both player information and match schedules
- D. Records for both player information and match schedules

### Question 4 (1 mark) KK3.1.13, KK3.2.16

During the user testing phase of a new photo editing software, users report that certain filters cause the application to crash unexpectedly, and some icons in the toolbar are misleading, leading to frequent misclicks. The development team categorises the crash as one type of error and the misleading icons as an issue related to a specific design principle. What type of error is the crash, and which design principle is being violated by the misleading icons?

- A. Runtime error; affordance
- B. Runtime error; feedback
- C. Logic error; consistency
- D. Syntax error; visibility

### Question 5 (1 mark) KK3.1.7

A software engineer is designing a class structure for a new video game that includes characters, weapons and levels. When creating a subclass for special weapons using the main weapon class, what OOP principle is being applied?

- A. Encapsulation
- B. Inheritance
- C. Polymorphism
- D. Abstraction



## Short-answer questions

### Sample case study: GreenGrocer inventory management system

GreenGrocer is a chain of grocery stores that has decided to develop a new inventory management system (IMS) to streamline its operations. The system will track inventory levels, sales and deliveries in real time. It will feature a user-friendly interface for employees to update inventory data, generate sales reports and manage orders. The system must ensure data accuracy, be scalable to accommodate future growth and have minimal downtime. For ease of use, the interface will display information in a clear, intuitive format, and the system will provide alerts when inventory levels for popular items are low.

To design the system, the development team plans to use various tools, including data dictionaries to define the structure of inventory data, mock-ups for the user interface and IPO charts to outline the process flows. The system will handle various data types, including text for item descriptions, numeric for prices and quantities and Boolean for stock availability. The inventory data will be stored using records to represent individual items, with fields for item ID, name, price, quantity and restock level. The team will adhere to clear naming conventions to ensure code readability and maintainability.

#### Question 1 (2 marks) KK3.1.2

Describe one functional requirement and one non-functional requirement for the GreenGrocer IMS based on the case study.

#### Question 2 (4 marks) KK3.1.2, KK3.2.4, KK3.2.5

Describe two methods GreenGrocer might have used to collect data for determining the IMS needs and how collecting this data would have helped in defining the functional and non-functional requirements.

#### Question 3 (2 marks) KK3.2.8, KK3.2.5

How might use case diagrams have been used in the IMS project, and what insights into the functional requirements could they provide?

#### Question 4 (2 marks) KK3.2.12, KK3.2.14

Describe an ideation technique that could be used in developing the IMS and how a specific design tool would translate this idea into a solution design.

#### Question 5 (2 marks) KK3.1.6

Given that sales reports need to be shared with suppliers in a format that is easy to process, which data source format would be most suitable and why?

#### Question 6 (4 marks) KK3.2.10, KK3.2.15

Discuss the importance of considering legal requirements related to data privacy in the IMS and how this consideration affects the UX design, specifically in terms of security and usability.



## EXAM PRACTICE

**Question 7** (3 marks) *KK3.2.13, KK3.2.14*

How might the team evaluate the efficiency and effectiveness of a proposed design idea for the IMS, and which design tool would be instrumental in this process?

**Question 8** (2 marks) *KK3.1.1, KK3.1.14*

As part of the development of the IMS, the team is considering integrating emerging programming trends such as AI-generated code snippets to automate certain tasks like data entry and report generation. The system is being designed using object-oriented programming (OOP) principles to ensure modularity and ease of maintenance. The programming language selected offers robust support for OOP, including features necessary for building a user-friendly graphical user interface.

How could AI-generated code snippets aid in the debugging and testing of the GreenGrocer IMS, and what ethical considerations should the team keep in mind?

**Question 9** (2 marks) *KK3.1.10, KK3.1.13*

Explain how type checking during data validation can prevent logic errors in the IMS when processing inventory updates.

**Question 10** (4 marks) *KK3.1.8, KK3.1.12*

Discuss the choice between implementing a binary search and a linear search algorithm for searching inventory items in the IMS, considering the data structure and control structures used.

# Unit 4

## INTRODUCTION

In Unit 3, Outcome 2, you wrote a design brief identifying a problem, need or opportunity in your community that required a software solution. The software requirements specification (SRS) provided details about data relationships and functional and non-functional requirements. A design folio outlined a preferred design, chosen from several alternatives, with mock-up, data dictionary and pseudocode descriptions in preparation for further development of a software solution.

In Unit 4, Area of Study 1, you will use the development and evaluation stages of the problem-solving methodology to complete the second part of the School-assessed Task (the SAT). You will develop the preferred design, identified in part 1, into a software solution using an appropriate programming language.

You will especially focus on good design (chosen from several alternative design ideas) and communicating your message clearly to the intended audience.

You will also evaluate how effectively your software solution achieves its goals.

Throughout the development of your solution, you will use and evaluate the project management plan you began in Unit 3, Outcome 2.

In Unit 4, Area of Study 2, you will focus on the security risks to software and data during the software development process. You will focus on the security risks to organisations as a result of insecure software development environments and practices. You will analyse and evaluate the security of current software development practices, examine the vulnerabilities and risks using threat-modelling principles, and consider the consequences of identified issues.

### Area of Study 1 – Software development: development and evaluation

**OUTCOME 1** In this outcome, you will develop and evaluate a software solution that meets and assesses the effectiveness of the project plan.

### Area of Study 2 – Cyber security: secure software development practices

**OUTCOME 2** In this outcome, you will respond to a teacher-provided case study to analyse an organisation’s software development practices, identify and evaluate current security controls and threats to software development practices, and make recommendations to improve practices.

Victorian Curriculum and Assessment Authority, Victorian Certificate of Education  
Applied Computing Study Design 2025–2028, VCAA, 2024, pp. 61–5



# Software development and project evaluation

## KEY KNOWLEDGE

After completing this chapter, you will be able to demonstrate knowledge of:

- KK4.1.1: characteristics of efficient and effective solutions
- KK4.1.2: characteristics of data types, data structures and data sources for input, storage and output
- KK4.1.3: features of a programming language: access modifiers (public, protected and private)
- KK4.1.4: established and innovative approaches to software development, including the use of code repositories, APIs and AI assistants
- KK4.1.5: validation techniques
- KK4.1.6: debugging and alpha testing techniques for checking solutions meet requirements and function correctly: commenting out code
- KK4.1.7: strategies for conducting beta testing
- KK4.1.8: features of evaluation strategies
- KK4.1.9: techniques for applying evaluation criteria
- KK4.1.10: factors that influence the effectiveness of project plans, including scope creep, personnel changes and technical issues
- KK4.1.11: techniques for recording the progress of projects
- KK4.1.12: techniques for assessing the effectiveness of a project plan.

Victorian Curriculum and Assessment Authority, Victorian Certificate of Education Applied Computing Study Design 2025–2028, VCAA, 2024, pp. 62–3

## FOR THE STUDENT

This chapter will cover the theory and skills required for Unit 4, Outcome 1. We will discuss access modifiers as a feature of your chosen programming language as well as characteristics of effective and efficient solutions leading to methods of debugging, validation techniques, and alpha and beta testing. Finally, the chapter will consider evaluation of the efficiency and effectiveness of the project management plan.

By the end of this chapter, you will be ready to develop and evaluate your software solution, and to report on the effectiveness of the Gantt chart in monitoring the progress of your software solution.

This chapter provides the theory and skills needed for Unit 4, Outcome 1. Having covered analysis and design requirements of the problem-solving methodology (PSM) in Chapters 3 and 4, you are now introduced to the development and evaluation stages. Programming languages, processing features, characteristics of efficient and effective solutions, and techniques for checking that SRS design specifications are verified will all be reviewed. Testing, validation and testing techniques will be applied to the software solution. The chapter concludes by considering project evaluation of the software solution and the efficiency and effectiveness of project plans.

By the end of this chapter, you should be able to complete development of your software solution using features of the chosen object-oriented programming language; document the progress of the project using an updated Gantt chart with annotations; evaluate the efficiency and effectiveness of the software solution with testing, validation and usability testing; and evaluate the effectiveness of the project management plan.

Note: You will develop your own software product for the SAT, including identifying the software solution.



## Efficient and effective solutions

Efficiency refers to the ability to achieve a desired outcome or goal with the least amount of resources, time or effort. When considering solutions, efficiency optimises processes and minimises resource consumption and costs while still achieving the desired result.

Effectiveness is achieving the intended objectives or goals successfully. It measures how well a solution accomplishes its intended purpose. Effectiveness emphasises doing the right things to achieve the desired outcomes, even if it may require more resources or effort than an efficient approach.

Efficient and effective solutions in software development possess several key characteristics, including: **user-centred design**, **clear and concise code** and detailed **internal documentation**.

### User-centred design

User-centred design is a design approach that places the needs, preferences and behaviours of end users at the forefront of the design process. It is a human-centred approach that focuses on creating solutions that are functional, intuitive, usable and enjoyable for users. It helps ensure that software solutions are technically sound and meet user expectations and needs.

When undertaking user-centred design, designers conduct user research to learn about the users' goals, preferences and pain points. Software designers use data collection techniques such as surveys, interviews and user observations to gather insights into user needs and ideation techniques to generate ideas and concepts for the design of the software. Software designers create **personas**, which are fictional representations of different user types (see Figure 5.1). These personas are based on research findings and represent the diverse characteristics of actual users. Personas, which include demographics, needs, goals and behaviours, help designers to empathise with users and make design decisions that cater to the needs of specific user groups.

User-centred design is an iterative process. Designers create prototypes or design concepts and gather feedback from users at various stages of development. Feedback drives refinements and improvements, ensuring that the final product aligns with user expectations. As such, prototyping is a key element of user-centred design. **Low-fidelity prototypes** and **high-fidelity prototypes** are created to test concepts and gather feedback. These prototypes allow for quick iterations and refinements before investing in full-scale development.

In user-centred design, the user interface (UI) is designed to be visually appealing, intuitive and consistent. Designers consider factors such as layout, colour, font selection and interactive elements to enhance user experience. This includes consideration of accessibility and inclusivity. Designers ensure that software interfaces are designed to be usable by individuals with disabilities as well as cater to diverse user demographics.

Ultimately, the goal of user-centred design is to create software solutions that meet functional requirements and establish satisfied users. Positive user experiences lead to user loyalty, word-of-mouth recommendations and business success.

Low-fidelity prototypes are basic and created to quickly explore design ideas and concepts with limited detail and interactivity, often in the form of sketches or mock-ups. They are primarily used for initial concept validation and idea generation. In contrast, high-fidelity prototypes are advanced, detailed simulations that closely resemble the final product, with realistic visuals, interactive elements and often actual content. High-fidelity prototypes are valuable for user feedback and demonstrating the final design to stakeholders. The choice between low and high fidelity depends on the design phase and specific objectives of the prototyping process.

## Jane



GoodStudio/Shutterstock.com

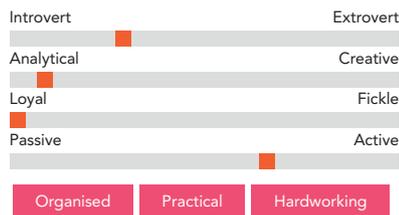
*'I'm looking for software that will simplify the planning of my bookings'*

Age: 30  
 Work: Motivational Speaker  
 Family: Married, 1 child  
 Location: Melbourne, Australia  
 Character: The Frequent Flyer

## Bio

Jane is a Motivational Speaker who visits schools across Australia to present motivational talks to students about mental health and wellbeing. She speaks at many schools and spends a lot of time manually scheduling her bookings for the year ahead, no matter how many times she has visited particular schools. She wants her diary to be as organised as she is, without spending the amount of time she is spending now.

## Personality



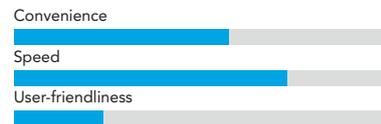
## Goals

- To spend less time booking schools
- To organise her schedule quickly

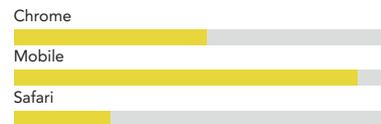
## Frustrations

- Too much time spent doing data entry – she's busy!
- Not terribly tech-savvy – doesn't like the process

## Motivation



## Preferred Tools



**FIGURE 5.1**

A sample persona, used in user-centred design

## Clear and concise code

Clear and concise code plays a crucial role in efficiency and effectiveness in software development. When code is clear and concise, it is easier to understand, work with and maintain. This contributes to efficiency by reducing development time, debugging effort and the likelihood of introducing errors. It also enhances effectiveness by promoting collaboration among developers, reducing miscommunication and ensuring that the code's logic and intent are easily comprehensible.

Clear code reduces the learning curve for new developers, improves error handling and ultimately leads to more successful software development and maintenance, aligning with efficient and effective software practices.

## Detailed internal documentation

Internal documentation plays an important role in increasing efficiency and effectiveness. It accelerates efficiency by expediting induction training for new developers, reducing debugging time through detailed explanations of code logic, and streamlining maintenance efforts by providing insights into code structure and interactions.

Simultaneously, internal documentation enhances effectiveness by promoting collaboration and shared knowledge, reducing miscommunication among team members, guiding developers in adhering to coding standards and facilitating code reviews. Furthermore, it ensures that critical information remains accessible for knowledge transfer and long-term organisational effectiveness, making internal documentation an indispensable asset in the software development process.

Internal documentation is complementary to clear and concise code. While clear code focuses on making the code readable without the need for comments, internal documentation provides detailed insights and context that may not be clear just from reading the code. Achieving a balance between the two is essential for undertaking efficient and effective software development.

## Data types, structures and sources

Data types and structures are characterised by how data is handled when considering input, storage and output.

### Input

This relates to how data is received or entered into the system. For different data types and structures, input mechanisms vary. For example, simple data types such as integers and strings might be directly entered by the user or read from a file. Complex data structures such as arrays or records may require more elaborate methods of input, such as filling in each element or field individually from a user interface or reading structured data from databases or files.

### Storage

Storage is how data is represented and organised in memory. For basic data types, storage allocates memory space based on the type's size and format (such as binary for integers). In data structures, storage becomes more complex. Arrays store elements in **contiguous memory** locations, while structures like linked lists or trees use pointers to connect elements that may be scattered in memory (**non-contiguous**). The efficiency of storage greatly affects the performance of data manipulations and access.

### Output

Output is how data is retrieved and presented from the system. Output for basic data types converts the stored binary data into a human-readable format, such as displaying numbers or text. For data structures, output can be more complex, often requiring traversal or iteration over the structure. For instance, outputting a 2D array might involve formatting the data in a tabular form, while a tree structure may need a traversal algorithm to output all of its elements in a specific order.

Each data type and structure has unique characteristics as to how it handles input, storage and output, which guides programmers in choosing the most suitable ones for their specific needs. For example, when deciding between an array and a record, the choice hinges on the need for uniform data handling (preferencing arrays) against the requirement for a composite data type that can encapsulate diverse data types (preferencing records). Understanding these characteristics ensures efficient data handling and processing in software applications.

Table 5.1 outlines the input, storage and output considerations of the data types and structures studied in VCE Software Development.

**TABLE 5.1** Characteristics of data types and structures

| Data type/structure   | Input   | Storage   | Output   |
|-----------------------|---|---|--|
| Character             | Single symbols via keyboard or data files           | Numeric codes (e.g. ASCII, Unicode) in memory   | Visual symbol representation on screens or printed formats     |
| String                | Sequences of symbols via user input or data files   | Sequence of character codes in contiguous memory, possibly with length or null terminator | Readable format of characters maintaining original order       |
| Integer               | Decimal representation from users or data sources   | Binary format in a fixed number of bits   | Decimal or other numeral systems for human readability         |
| Floating point        | Decimal input with fractional parts                 | Stored as a base, exponent and sign (e.g. IEEE 754 format)                                | Decimal or scientific notation respecting precision            |
| Date/time             | Various string formats or graphical UI selection    | Large integer (timestamp) or separate components (year, month, day, etc.)                 | Formatted strings according to desired date/time format        |
| One-dimensional array | Sequential or individual inputs at specific indices | Contiguous memory locations maintaining order by index                                    | List or sequence format in a readable layout                   |
| Two-dimensional array | Row by row or column by column with two indices     | Row-major or column-major order in contiguous memory                                      | Tabular format respecting row and column structure             |
| Record                | Separate input for each field with varying types    | Aggregates of fields in memory, sequential or aligned                                     | Key-value formatted listing of fields and corresponding values |

As well as writing the code, you will need to check other important details for your program to operate correctly. Testing of your code must be documented in a test table. Validation of data and data types is to be thoroughly carried out. Remember, testing is not validation. One checks that the code executes as expected. The other checks boundary conditions for data so the limitations are observed. Be sure to know the difference, as a program can execute correctly with the wrong input data.

## Features of a programming language

Data types were first introduced in Chapter 1 (see page 3) and programming features were first introduced in Chapter 2 (see page 31). This section will review and extend on that knowledge for inclusion in your SAT Unit 4, Outcome 1. Stage 3 of the problem-solving methodology (PSM) is development. Your development task will implement your previously chosen design, with details documented in your client's software requirements specification (SRS).

### Choice of programming language

Your actual programming language will be chosen at your school in accordance with Software Development programming requirements. These requirements are considered each year, and any changes are announced annually in the VCAA Bulletin.

The discussion about languages presented in this text provides development within three conceptual layers:

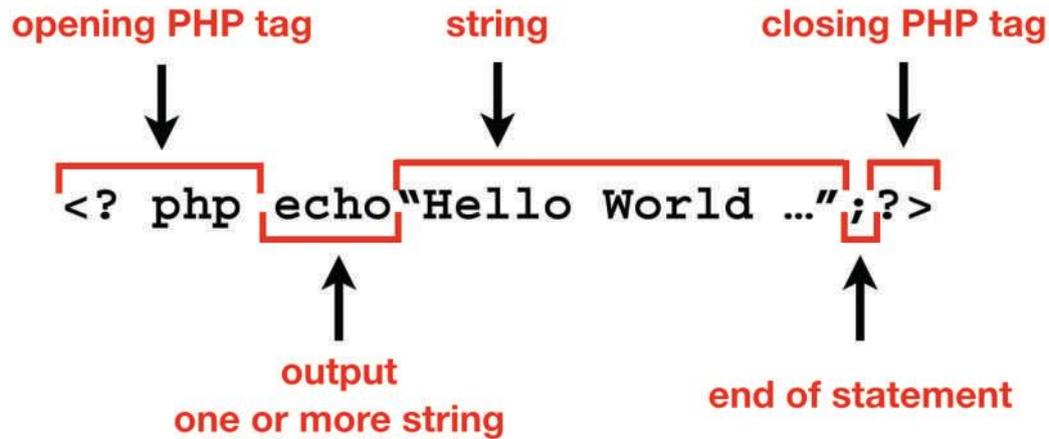
- *Interface* includes graphical user interface (GUI) for digital systems such as notebooks, tablets, smartphones, gaming consoles, and microprocessors including robots.
- *Logic* is applied through data structures, data validation techniques, and the control structures of selection, iteration and sequencing. Modular organisation and code optimisation is supported. The programming language contains objects, methods and event-driven programming functions.
- *Data sources* can allow retrieval of data from external storage or external access technologies.

There are many programming languages used within education, in industry and for personal use. No two languages are the same. Programming languages come with their own syntax and style, but there are elements in a computer language that can be found in all languages.

The following are features that every programming language contains.

## Instructions and syntax

Every language has its own syntax. Syntax is the set of rules that are used to create instructions. An instruction is something you want the computer to execute. A language reference guide is usually available to assist in finding the syntax and reserved words for the language.



**FIGURE 5.2**  
An example of PHP syntax

The following shows syntax with a couple of instructions:

```
if ($my_name == "someone")
{
    echo "Your name is someone! <br/>";
}
echo "Welcome to my homepage!";
```

The words in bold are the built-in commands for this particular language, while the whole lines (and there are three lines) are instructions and follow a specific syntax.

**TABLE 5.2** Examples of Hello World! in several computer languages

|        |  |
|--------|--|
| JAVA   | <pre>public class Main {     public static void main(String[] args)     {         System.out.println("Hello, World!");     } }</pre> |
| PYTHON | <pre>Print ("Hello, World!")</pre>   |
| C#     | <pre>public class Hello {     public static void Main()     {         Console.WriteLine("Hello, World!");     } }</pre>              |




**TABLE 5.2** Examples of Hello World! in several computer languages (continued)

|   |  |
|---|--|
| VB – Visual Basic   | <code>Console.WriteLine("Hello, World!");</code>   |
| HTML<br>(note: HTML is NOT object oriented and does NOT meet VCAA requirements) | <pre>&lt;!DOCTYPE html&gt; &lt;html&gt;   &lt;head&gt;     &lt;title&gt;Hello, World!&lt;/title&gt;   &lt;/head&gt;   &lt;body&gt;     &lt;p&gt;Hello, World!&lt;/p&gt;   &lt;/body&gt; &lt;/html&gt;</pre>  |
| PHP   | <pre>&lt;?php echo "Hello, World!"; ?&gt;</pre>  |
| RUBY  | <code>puts 'Hello, World!'</code>  |
| Mathematica   | <pre>CloudDeploy[   ExportForm[     Style[       Framed["Hello, World", ImageMargins -&gt; 60],       80,       Orange,       FontFamily -&gt; "Verdana"],       "GIF"],     Permissions -&gt; "Public"]</pre> <p>Once written, Wolfram language uses CloudObject, which does not require a 'host computer', just a browser.</p> |



Weblink  
CloudObject

There are many other programming languages, but any language chosen in VCE Software Development must meet the VCAA programming requirements.

## Variables and constants

Local variables, global variables and constants help manage and store data in a program.

- *Local variables* are declared inside a function or a block and can only be accessed within that function or block. Their scope is limited to the function or block in which they are declared.
- *Global variables* are declared outside any function or block and can be accessed from any part of the program. Their scope is the entire program, making them accessible across different functions or blocks.
- *Constants* are values that are defined once and cannot be changed once set. They are defined with a specific value at the time of declaration and are used to store values that remain the same throughout the program's execution.

The implementation of these concepts can vary across high-level programming languages. Table 5.3 shows how local variables, global variables and constants are implemented in Python, Java and C#.

**TABLE 5.3** Local, global and constant variables in Python, Java and C#

| Type of variable | Python   | Java  | C#  |
|------------------|--|---|---|
| Local variable   | Declared inside a function or block of code:<br><pre>def func():     l_var = 10</pre>  | Declared within a method or block of code:<br><pre>void method() {     int lVar = 10; }</pre>           | Declared within a method or block of code:<br><pre>void method() {     int lVar = 10; }</pre> |
| Global variable  | Outside functions:<br><pre>g_var = 20</pre> To modify inside a function:<br><pre>def func():     global g_var     g_var = 30</pre> | Outside any method but within a class:<br><pre>public class MyClass {     static int gVar = 20; }</pre> | At the class level:<br><pre>class MyClass {     public static int gVar = 20; }</pre>          |
| Constant         | Conventionally uppercase:<br><pre>CONST_VALUE = 100</pre>  | With final keyword:<br><pre>final int ConstValue = 100;</pre>   | With const keyword:<br><pre>const int ConstValue = 100;</pre>                                 |

## Data types

- *Numeric*: A broad category that includes any type of data that represents a number. Numeric types can be further divided into integer, floating point and date/time types, among others, depending on the language and the precision required.
- *Integer*: Represents whole numbers without fractional components. Integers can be signed (including negative values) or unsigned (only positive values and zero). They typically come in various sizes (e.g. 8-bit, 16-bit, 32-bit and 64-bit) depending on the range of values they can store.
- *Floating point*: Used to represent real numbers that can have fractional parts. They are called floating point because the decimal point can 'float'; that is, the number can have a varying number of digits after the decimal point. They are required when precision in calculations requires more than just whole numbers.
 

This data type can encode dates (year, month, day), times (hour, minute, second) or both. Operations on date/time types often include calculating differences between dates, formatting dates for display and manipulating individual components of a date or time.
- *Text*: Designed to store and manipulate textual data, encompassing both single characters and strings of characters. This data type is used for handling human-readable content, allowing programs to process words, sentences, paragraphs and even entire documents of textual information.
- *Character*: Represents a single character, which can be a letter, number, punctuation mark, space or symbol. Characters are typically stored in a character encoding standard such as ASCII or Unicode, allowing for a wide range of characters from various languages.
- *String*: A sequence of characters treated as a single entity. Strings are used to represent text and can include letters, numbers, spaces and symbols. Operations on strings can include concatenation (joining), slicing (separating) and manipulation of individual characters.
- *Boolean*: Represents a logical entity that can have only two values: true or false. Booleans are used in control structures (such as if-else statements and loops) and allow for decision-making and flow control in programming.

How data types are declared and used varies across high-level programming languages. Table 5.4 shows how to define these data types in Python, VB.Net and Swift.

**TABLE 5.4** Data types in Python, VB.Net and Swift

| Data type      | Python  | VB.Net  | Swift                                      |
|----------------|---|---|--|
| Integer        | <code>intVar = 42</code>  | <code>Dim intVar As Integer = 42</code>           | <code>var intVar: Int = 42</code>          |
| Floating point | <code>floatVar = 3.14</code>  | <code>Dim floatVar As Double = 3.14</code>        | <code>var floatVar Double = 3.14</code>    |
| Date/time      | <code>from datetime import datetime<br/>dtVar = datetime.now()</code> | <code>Dim dtVar As DateTime = DateTime.Now</code> | <code>let dtVar = Date()</code>            |
| Character      | <code>chrVar = 'A'</code>   | <code>Dim chrVar As Char = "A"c</code>            | <code>var chrVar: Character = "A"</code>   |
| String         | <code>strVar = "Hello!"</code>  | <code>Dim strVar As String = "Hello!"</code>      | <code>var strVar: String = "Hello!"</code> |
| Boolean        | <code>boolVar = True</code>   | <code>Dim boolVar As Boolean = True</code>        | <code>var boolVar: Bool = true</code>      |

## Control structures

Control structures were introduced in Chapter 2 (page 33) and several detailed examples were provided. Here is a brief review.

Control flow is the execution order of statements, instructions or functions that are evaluated in a program. Flow of control is implemented with three basic types of control structures:

- sequential
- selection, which chooses between two or more alternatives
- iteration or repetition, where looping determines how many times the instruction is executed.

### Sequence or sequential structure

Sequential is the default mode, where instructions are executed line by line in the sequence they were written.

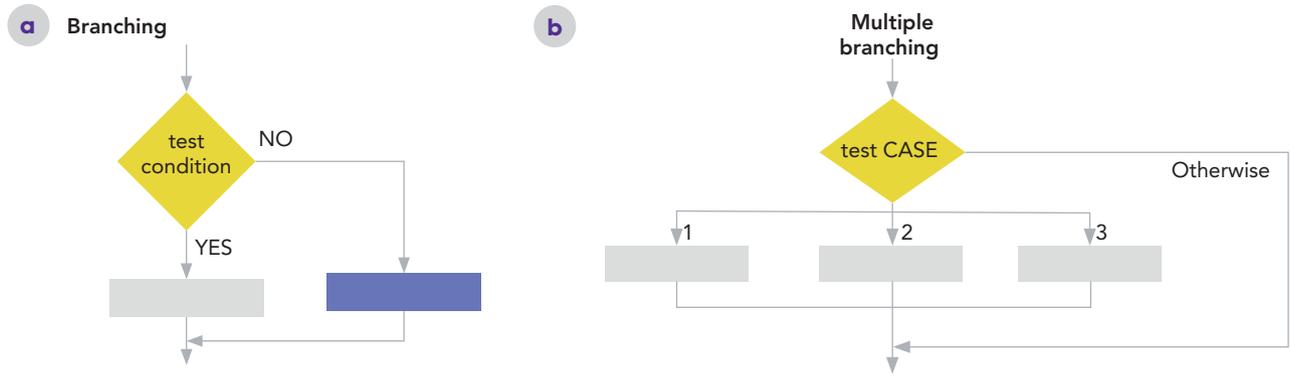
#### Sequence



**FIGURE 5.3** Start at the beginning and follow the steps, one by one.

### Selection structure

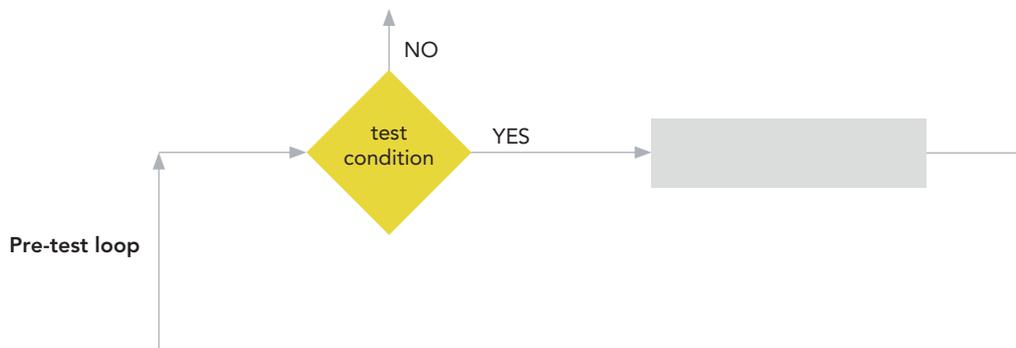
Selection is used for branching when a choice is made between two or more alternative paths. All the instructions for every program can be constructed from just seven control structure statements (see Table 5.5).



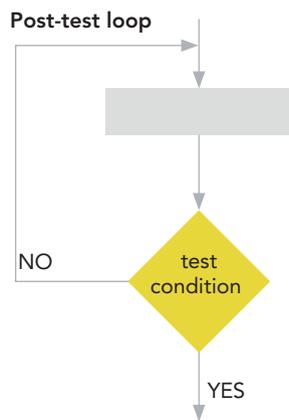
**FIGURE 5.4** Branching selection: **a** Binary (e.g. If/Then); **b** Multiple branches (e.g. Switch or Select CASE)

### Iteration structure

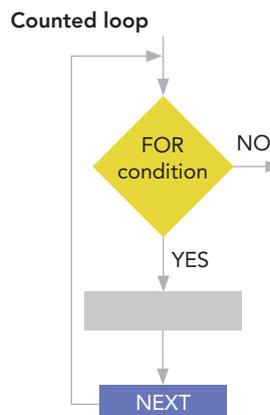
Repetition allows loops to execute a section of code multiple times. There are three types of loops.



**FIGURE 5.5** Pre-test: the condition is met at the beginning of the loop (e.g. While-Do).



**FIGURE 5.6** Post-test: the condition is met at the end of the loop (e.g. Repeat Until).



**FIGURE 5.7** Counted where the number of times can be specified

**TABLE 5.5** Control structure statements

| Control structure statement | Action   |
|-----------------------------|--|
| If                          | When condition is met, flow is directed to the alternative instruction                                 |
| Switch or Case              | Choose between listed alternatives   |
| If ... Then ... Else        | When condition is met, flow can be directed multiple times   |
| While ...                   | May not execute to satisfy condition   |
| Do ... While                | May not execute if condition is not satisfied once   |
| Repeat ... Until            | Executes at least once<br>If conditions are not met, there is a possibility of an <b>infinite loop</b> |
| For ... Next                | A counting loop to be executed a set number of times   |

These control statements are combined in only two ways: stacking and nesting. The stacking method for control statements has the exit point of one control statement connecting to the entry point of the next control statement. These control statements are simply lined up one after the other, or stacked. The only other method for connecting control statements is to place control statements inside other control statements. This method is known as nesting.

A ternary operator is a concise way to perform conditional operations in many programming languages. It is called 'ternary' because it involves three operands: a condition, a result for when the condition is true, and a result for when the condition is false. The basic syntax of a ternary operator is:

```
condition ?
result_if_true :
result_if_false
```

## Arithmetic, logical and conditional operators

Arithmetic operators are used to perform common mathematical operations, such as addition (+), subtraction (-), multiplication (\*), division (/) and modulus (%). They are used when processing numerical data and performing calculations.

Logical operators include AND, OR and NOT. They are used in decision-making structures to combine or invert Boolean conditions. For example, AND returns true if both operands are true, OR returns true if at least one operand is true and NOT inverts the truth value of its operand.

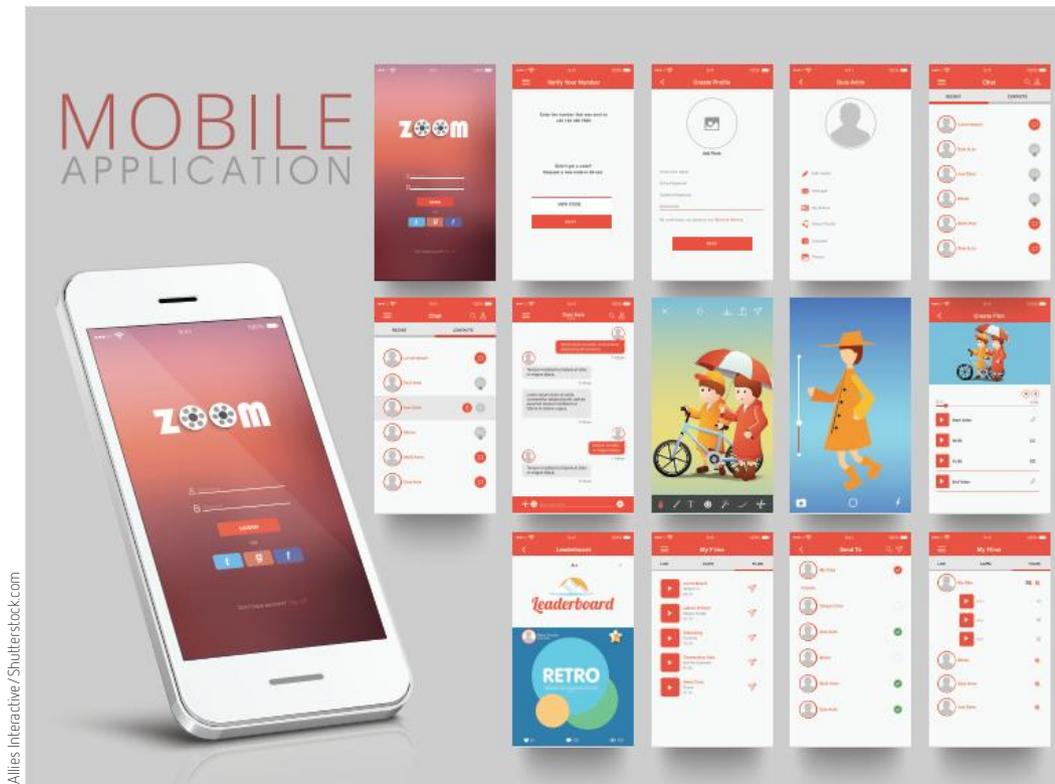
Conditional operators (often part of conditional statements) evaluate expressions to produce a Boolean result (true or false). The most common conditional operator is the 'if' statement, which executes a block of code based on a condition. The ternary operator '?' is a shorthand for if-else statements in many languages, allowing a simple conditional assignment in a single line.

## Graphical user interfaces (GUIs)

Graphical user interfaces (GUIs) are interactive interfaces that allow users to interact with electronic devices using graphical icons and visual indicators, as opposed to text-based interfaces, typed command labels or text navigation. GUIs make software applications more intuitive and user-friendly by providing windows, buttons, menus, icons and other visual elements that users can interact with using devices such as mice, keyboards and touchscreens.

## Functions and methods

Functions are reusable blocks of code designed to perform a specific task. Functions take in data as input, process it and return a result. They help in making code more organised, readable and maintainable. In functional programming languages, functions can be passed as arguments to other functions, returned as values and assigned to variables.



**FIGURE 5.8** Some examples of graphical user interfaces

Methods are similar to functions but are associated with objects or classes. A method is a function that ‘belongs to’ an object, and it can access and modify the data of the object it belongs to. Methods are used in object-oriented programming, enabling the encapsulation and manipulation of an object’s state.

## Classes and objects

Classes are blueprints for creating objects (a particular data structure). A class defines a set of attributes and methods that characterise any object of the class. It encapsulates data for the object and methods to manipulate that data.

Objects are instances of classes created using the class blueprint. Each object can have different values for the attributes defined by its class, allowing for the creation of a diverse range of entities.

## Access modifiers

Security is an important aspect to consider when writing source code for modern applications. Aside from encryption, there are other ways a software developer can protect access to data within variables. The most common method is to use access modifiers to restrict access to functions and variables within applications. Access modifiers can also be used on variables.

## Public visibility

Public visibility of a function or variable means that it is visible both inside the source code or class in which it exists, and also via external source files, classes and applications. While a programmer can explicitly refer to a function or variable as public, there is no need to do this in most programming languages, as public visibility is the default visibility.

### SCHOOL-ASSESSED TASK TRACKER

 Design brief

 Project plan

 Analysis

 Folio of alternative design ideas

 Development

 Evaluation and assessment

 Final submission

## Protected visibility

Protected visibility means that it is visible only to a class or extensions of that class. This means that the functions, variables and methods defined as protected within a class can only be used by that class as well as by any classes that inherit the class that contains the protected element.

## Private visibility

Private visibility means that it is visible only to a particular class. Unlike protected elements, a private element cannot be used by a class that inherits the class that contains it.

Python uses a convention-based approach to signal the intended level of access for methods and attributes, relying on naming patterns rather than language-enforced access controls.

**FIGURE 5.9** Example use of public, private and protected variables in Python

```
1 class Car:
2     def __init__(self):
3         print("Engine started")
4         self.name = "corolla"
5         self.__make = "toyota"
6         self._model = 1999
```

Source: Python Software Foundation

Figure 5.9 shows an example of how to use public, protected and private variables in Python. No underscores on a variable (`self.name`) is public, one underscore (`self._model`) is protected and two underscores (`self.__make`) is private.

# Approaches to software development

Software development has evolved significantly over the years, with both established practices and innovative approaches shaping the way software is designed, developed and maintained. Among these is the use of **code repositories**, **application programming interfaces (APIs) and libraries** and **artificial intelligence (AI)-based assistants**.

## Code repositories

Code repositories, such as Git, have become a fundamental part of modern software development, offering **version control** and collaboration features. They allow developers to track changes, revert to previous versions of the code and manage multiple development branches simultaneously. Platforms such as GitHub, GitLab and Bitbucket enhance this further by providing a web-based interface for code hosting, review and collaboration, facilitating open-source projects and team-based development.

Innovation in code repositories involves integrating continuous integration/continuous deployment (CI/CD) pipelines, automated testing, and deployment tools directly within these platforms. This integration supports **DevOps** practices, enabling faster and more reliable software releases by automating build, test and deployment processes.

## Continuous integration/continuous deployment

Continuous integration (CI) is the practice of automatically testing and merging code changes into a shared repository frequently, ideally multiple times a day. The goal is to catch and fix integration errors quickly, to maintain a high quality of code and to reduce the time it takes to validate and release new software updates. In CI, every time a developer checks in changes to the version control system, an automated build and test process is triggered. This ensures that the new code integrates seamlessly with the existing codebase and that any issues are identified and addressed immediately.

Continuous deployment (CD) extends the CI process by automatically deploying all code changes to a testing or production environment after the build stage. This means that every successful build can be released to customers without manual intervention, allowing for a much faster and more efficient release process. The CD process typically includes automated testing in production-like environments to ensure that changes do not adversely affect the live application.

## APIs and libraries

APIs and libraries have long been used in software development, allowing developers to reuse code, connect with external services and extend functionality. Libraries provide pre-written code snippets, functions and frameworks that developers can incorporate into their projects, speeding up the development process and ensuring reliability.

In today's software world, APIs and libraries have evolved. They now include smaller, more flexible building blocks called microservices and cloud-based APIs. These make it easier to create software that can grow and adapt as needed. Developers can now access a vast range of cloud services (such as AWS, Azure and Google Cloud) through APIs for functionalities such as machine learning, data analytics and Internet of Things (IoT), enabling more powerful and complex applications to be built more efficiently.

## AI-based assistants

AI-based assistants in software development are a relatively new and rapidly evolving area. Tools such as GitHub Copilot, powered by OpenAI's Codex, offer AI-driven code completion, suggesting entire lines or blocks of code as developers type. These assistants can significantly speed up coding, help explore new APIs and libraries by providing usage examples and even assist in debugging by suggesting potential fixes.

Beyond code completion, AI in software development is expanding into areas such as automated code review, where AI tools analyse code for bugs, vulnerabilities and performance issues. AI is also being explored for automating more mundane tasks, such as writing documentation and even generating test cases, further streamlining the development process.

## Software testing

There are some basic and essential software testing processes that every software developer should be performing as they go, before the code is shown to anyone else.

- Use functionality testing, so every feature is tested at each stage and after any changes.
- Keep versions of working code, and save and back up files regularly with date (and time if necessary). This may be automated in some development environments.

- Undertake one change at a time, and ensure it is tested. When the change is working with no errors, the next change/improvement can be introduced.
- Undertake module testing, where independent modules can be modified without affecting other parts of the software solution.
- Identify edge condition or boundary values for testing while coding, and place them into the test table for later.

## Validation techniques

**Validation** checks that input data is reasonable. Validation does not and cannot check that inputs are accurate. For example, it cannot tell whether a person is being honest when entering their age. However, validation can detect problems when a person enters their age as 174 years, or 'banana', or nothing at all. You can perform validation manually (yourself) or allow software to do it for you.

Computers are particularly good at conducting validation checks.

- 1 *Existence checks* ensure that a value has been entered and the field is not blank, or <null>.
- 2 *Type checks* ensure data is of the right type; for example, the age that has been entered is actually a number.
- 3 *Range checks* ensure that data is within acceptable limits (e.g. children enrolling in kindergarten must be 3–6 years old) or comes from a list of acceptable values (e.g. small, medium or large).

### Verification or validation?

Verification checks that the specific requirements are being developed. Validation checks that the final product meets expectations.

### THINK ABOUT SOFTWARE DEVELOPMENT

5.1

If a user is expected to enter their age, what would be a reasonable range check?

- a 5–50 years
- b 15–80 years
- c 0–100 years
- d 1–200 years

### THINK ABOUT SOFTWARE DEVELOPMENT

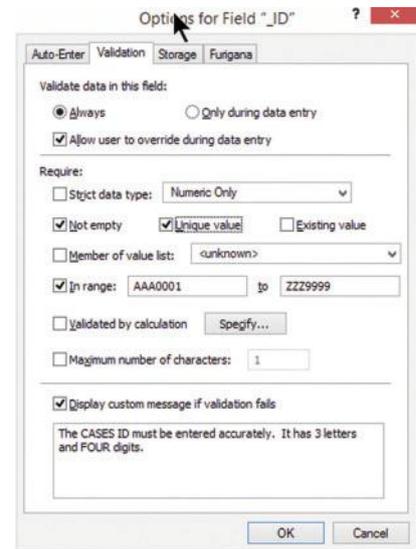
5.2

In Unit 4, Outcome 1, which data will you need to validate, and how will you achieve this?

Remember:

- **validation** checks whether the data inputs are reasonable
- **testing** checks the accuracy of information outputs.

In your SAT, ensure that all of your data is thoroughly and appropriately validated.



**FIGURE 5.10** Validation rules in FileMaker database. Here an ID field is made compulsory ('Not Empty') and unique, and within a defined range of values. The database is also told what error message to display if validation fails.

Source: FileMaker

## Debugging and alpha testing

### Debugging

**Debugging** is the process of identifying and resolving errors or bugs in software code to ensure that the program functions correctly. The most common methods of debugging involve setting **breakpoints** and commenting out code.

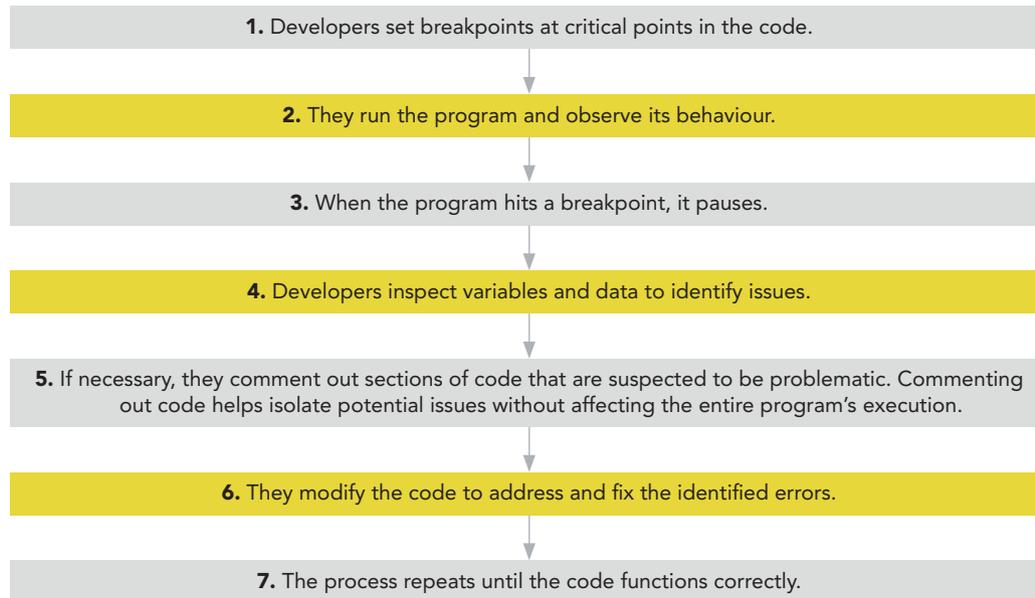
### Breakpoints

We looked at breakpoints in Chapter 2, page 65. Developers can set breakpoints in their code at specific lines where they suspect issues might occur. When the code reaches a breakpoint during execution, it pauses, allowing developers to inspect variables, values and the program's state to identify and fix errors.

## Commenting out code

Temporarily disabling sections of code by commenting them out (adding `//` or `/* */` around the code) is a technique used to isolate problematic code. This helps developers test the rest of the program to ensure it works correctly before addressing the disabled code.

### The debugging process typically involves the following steps:



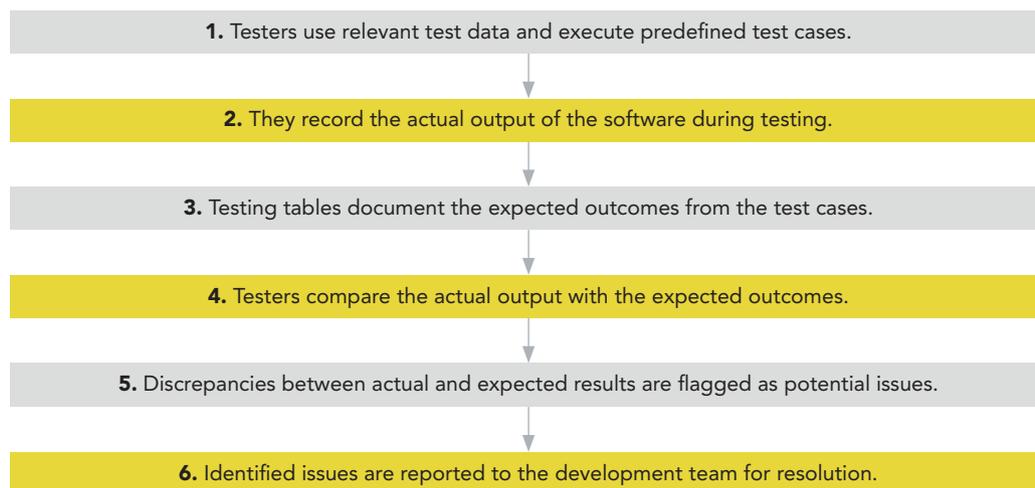
**FIGURE 5.11** Steps in the debugging process

## Alpha testing

**Alpha testing** is an early stage of software testing conducted by the software development team within the development environment. It focuses on identifying issues and ensuring the software meets initial requirements.

The alpha testing process typically involves using relevant test data and running test cases to compare expected and actual output as documented in testing tables.

### The alpha testing process typically involves the following steps:



**FIGURE 5.12** Steps in the alpha testing process

## Relevant test data

Alpha testing involves using real or simulated data that represents actual usage scenarios. This ensures that the software behaves realistically and helps uncover any data-related issues.

## Test cases and testing tables

Test cases are specific scenarios or inputs that testers use to verify the software's behaviour. Testing tables are often used to record and compare expected outcomes with actual results during testing.

## Expected and actual output

Test cases define expected outcomes based on the software's requirements. During alpha testing, testers execute these test cases and compare the actual results with the expected outcomes recorded in testing tables.

**TABLE 5.6** Sample testing table

| Test ID | Test description   | Expected output  | Actual output                                   | Result |
|---------|--|--|---|--------|
| TC001   | User login with valid credentials.<br>Username: saleh<br>Password: Ooei09! | Access granted, redirected to menu screen                    | Access granted, redirected to menu screen       | Pass   |
| TC002   | User login with invalid password<br>Username: saleh<br>Password: saleh     | Error message displayed                                      | Access granted, redirected to menu screen       | Fail   |
| TC003   | Create a new user profile, user does not exist in the database             | Success message displayed. User added to system              | Success message displayed. User added to system | Pass   |
| TC004   | Update user profile information, user exists in the database               | Profile updated successfully, user redirected to menu screen | Profile did not update                          | Fail   |
| TC005   | Submit a support ticket, all form fields contain data                      | Ticket submitted and assigned to helpdesk                    | Error message displayed                         | Fail   |

Thorough and careful **testing** is necessary, whether the software solution is a game, a website shopping cart or an airliner autopilot. If a software solution fails, this could disadvantage users.

If your solution fails because of undiscovered faults, it may become difficult to use, or completely unreadable.

Testing checks that a solution will produce the correct output and will do what it should do. Testing is not simple, quick or cheap – especially for a product such as an operating system with megabytes of code in thousands of files created by hundreds of people.

The typical steps involved in testing are as follows:

- 1 Decide which tests will be conducted.
- 2 Create suitable test data.
- 3 Determine expected results.
- 4 Conduct the test.
- 5 Record the actual results.
- 6 Correct any errors.
- 7 Document all steps in a summary table.

There are many test types, each intended to uncover different kinds of errors at different times during development. The types of testing relevant to your solution are listed in Table 5.7.

**TABLE 5.7** Testing types

| Name of test             | What is tested?   |
|--------------------------|---|
| Alpha testing (informal) | The completed initial software solution is tested internally by developers or in-house testers                              |
| Beta testing             | Typical end users use their own equipment to check that the finished solution is acceptable under different user conditions |

## Test data

To prove the accuracy of the output of a software solution, provide some test data to work with, and compare the actual output answer with expected output that is known to be correct.

Good test data includes:

- valid data – data that is perfectly acceptable, reasonable and fit to be processed.
- valid but unusual data – data that should not be rejected even though it seems odd. For example, a 10-year-old might, very occasionally, enrol in university. Validation that rejected the young genius’s enrolment would cause embarrassment.
- invalid data – to test the code’s validation routines. For example, if people must be 18 years old to be given a credit card, test data should include people under 18 so they can be seen to be rejected.
- boundary condition data – data that is on the borderline of some critical value where the behaviour of the code should change. These ‘tipping point’ errors are a frequent cause of logical errors in programming.
- wrong data – data with an inappropriate format would be expected to generate an error.
- absent data – a blank field entry will test how the system handles a ‘no entry’ entry.

Refer to Table 5.8 for test data and boundary test data.

**TABLE 5.8** A data test table with suitable (a) dummy data (b) boundary test

|     | Data name    | Data type | Range             | Below range          | Within range  | Above range          |
|-----|--------------|-----------|-------------------|----------------------|---------------|----------------------|
| (a) | Postcode     | Numeric   | 3000–3999         | 2200                 | 3500          | 4500                 |
|     | ID number    | Numeric   | 500–2000          | 200                  | 1000          | 3000                 |
|     | Order number | Numeric   | 00100–10000       | 50                   | 500           | 11000                |
|     | Data name    | Data type | Normal data range | Below lower boundary | At boundary   | Above upper boundary |
| (b) | Postcode     | Numeric   | 3000–3999         | 2999                 | 3000 and 3999 | 4000                 |
|     | ID number    | Numeric   | 500–2000          | 499                  | 500 and 2000  | 2001                 |
|     | Order number | Numeric   | 00100–10000       | 99                   | 100 and 10000 | 10001                |

## Documenting test results

A **testing table** is a commonly used way to record evidence of functionality testing. A testing table for a software solution may look like the example in Table 5.9.

**TABLE 5.9** A testing table

| What was tested   | How it was tested  | Expected result   | Actual result   | How it was fixed, if relevant                        |
|---|--|---|---|--|
| Age < 18  | 17 y 364 d   | 'under age'   | 'under age'   | -  |
| Age = 18  | 18 y 0 d   | 'happy birthday'  | 'under age'   | Boundary condition test was fixed                    |
| Age > 18  | 18 y 1 d   | 'adult'   | 'adult'   | -  |
| Readability   | Asked two volunteers to read sample pages and report on text size, contrast, alignment, spelling, vocabulary, offensiveness and headings | Reports each page was easy to read, accurate and inoffensive    | One reader suggested a button label in italics was hard to read | Changed button label text style from italics to bold |
| Code calculation of number of days since the last event | Set computer's clock to 7 days after the event   | Index page should display 'The last event occurred 7 days ago.' | Displayed '... created 6.89586 days ago ...'                    | Rounded up the age calculation in the code           |

## How to document your testing

- Use a testing table such as the one shown in Table 5.9.
- Seek a subjective report from your client or a fellow student who tried out your solution's readability and usability.
- Capture screenshots of features that are not normally visible, such as drop-down menus and warning messages, showing that they work when needed.
- Make handwritten calculations, annotating printed screenshots of your solution's calculations to verify that the output has been checked for accuracy.
- Capture screenshots of the solution's validation rules, responding properly to invalid data.

## Testing your solution

After designing and building your software solution, you need to demonstrate that it has been thoroughly tested. You need to know what to test in your solution. These will be discussed in the following sections.

### Inputs

Every input must be inspected to check that it is displaying in the right place, at the right time, and at the right speed and volume in a variety of common environments (meaning different browsers and devices).

### Buttons and links

Every internal and external button or link in the solution needs to be manually clicked and the result noted. Create a list of buttons and links and tick off each one as it passes testing.

## Links to external services

You should be able to completely test all parts of the solution under your control. You need to test the operation of any external connections to your product to ensure that data updates the function as expected.

## Readability

Use the checklist provided in Table 5.10 to test readability of your solution.

**TABLE 5.10** Readability checklist

| Checklist   |
|---|
| Is the text large enough to read comfortably on a small device? |
| Is contrast optimal, or at least satisfactory?                  |
| Is the typeface a readable size?                                |
| Are all buttons labelled and identifiable as buttons?           |
| Is text alignment attractive and readable on the page?          |
| Are the spelling, punctuation and grammar correct?              |
| Is expression clear and unambiguous?                            |

## Calculations

If your solution calculates any information, its answers need to be verified by manual recalculation in a testing table. For example, you might create a screen containing code to display a countdown timer to the next Software Development examination. To prove that you have tested the accuracy of its output, take a screenshot. Annotate the screenshot with whatever manual calculations will demonstrate that it is true, based on the time the screenshot was taken and the time of the exam.

## Loading times

If the software solution is online, clear any cache to remove pre-loaded copies of files and media and try loading the site via cable and Wi-Fi. Any page that takes more than a few seconds to load should be inspected and optimised. Another method is to use one of many online services that can measure the loading times for your pages. Online data repositories may have varying access times due to user demand.

## Accessibility

Does your solution create unnecessary difficulty for users with common disabilities such as poor eyesight or muscular control, or other challenges such as weak language skills? Is alt text applied to images? Are colour combinations considerate of colourblind people? Many colourblind-safe palettes are documented online.

There are several places online to test the accessibility of your solution. Try the AccessiBe website.



Weblink  
AccessiBe

## Dynamic features

Every selection option item must be checked and its behaviour documented in a testing table (see Table 5.9). If data entry forms are expected to work, data should be entered and its successful arrival at its destination should be documented. Any simulated functionality, such as a faked login box, should, as far as is practical, appear to work genuinely. Any coding should be run using a variety of test data and the behaviour of the code recorded.

## Beta testing

**Beta testing** is when a nearly complete version of the product is evaluated by a group of end users in real-world conditions. This stage follows alpha testing, which is typically conducted in a controlled environment by internal teams and precedes the final release of the product to the general public.

The primary goal of beta testing is to uncover technical issues, usability issues and other concerns that may not have been identified during earlier testing phases. It allows developers to gather feedback from actual users regarding the software's functionality, performance and overall user experience. This feedback is invaluable for making final adjustments to ensure the software meets user needs and expectations.

Beta testers are usually selected from the target user base and are often not part of the organisation developing the software. These testers use the software in their own environments, which helps identify issues related to diverse hardware configurations, operating systems and usage scenarios that the development team may not have tested.

Beta testing can be either 'closed', where a specific set of users are invited to test the software, or 'open', where the testing phase is made public and a broader audience can participate. The feedback from beta testers is collected, analysed and used to make final tweaks to the software, fix any critical issues and improve the overall stability and user experience. By involving real users in the testing process, developers can ensure the software is more aligned with user expectations, ultimately leading to a more successful and well-received product launch.

## Strategies for beta testing

A well-structured approach to beta testing involves several key strategies, including the construction of a testing plan and test scenarios, observation of these scenarios, and thorough documentation of test results.

The construction of a testing plan and test scenarios begins with defining the objectives of the beta test, such as identifying technical issues, assessing user satisfaction or understanding user interactions with the product. The plan outlines the scope of testing, the target demographic for testers, the duration of the beta test and the key features or functions to be tested. Test scenarios are then developed to simulate a wide range of user interactions with the product. These scenarios should cover common and uncommon use cases to ensure a comprehensive evaluation. For example, a software company might create scenarios that include installing the software on different operating systems, using the software in various network conditions or integrating the software with other applications.

Observation of testing scenarios helps in understanding how actual users interact with the software and in identifying any issues they encounter. This involves not just passive observation but also active engagement with testers to gather qualitative feedback. Techniques such as screen recording, real-time monitoring tools and direct communication channels such as forums or chat groups can be employed. For example, a video game developer might use live play sessions streamed by beta testers to observe gameplay, identify potential glitches and gauge user engagement and enjoyment levels.

Documentation of test results involves systematically recording the outcomes of each test scenario, including any technical issues identified, user feedback, performance metrics and any deviations from expected behaviour. This documentation should be organised and accessible, allowing for easy analysis and reference. The documentation process can be supported by tools such as bug tracking systems and feedback forms. For example, a mobile app developer who has had users beta testing a new fitness app might document issues reported by users, such as problems with tracking workout data, discrepancies in calorie counts or feedback on the app's user interface and experience.

**TABLE 5.11** Testing plan for a fitness app

| Component         | Description  |
|-------------------|--|
| Objective         | Evaluate app functionality, identify technical issues and assess user satisfaction.                              |
| Test scenarios    | Test the app's registration process, workout tracking, goal setting, device compatibility and nutrition logging. |
| Methodology       | Recruit users from the target audience, provide an in-app feedback system and hold weekly feedback sessions.     |
| Expected outcomes | Collect feedback on usability and technical issues and gather suggestions for improvements.                      |
| Timeline          | Three weeks: initial set-up in Week 1, ongoing testing in Weeks 2–3, final feedback at end of Week 3.            |
| Resources         | Beta version of the app, feedback tools.   |
| Success criteria  | Successful set-up by most participants, positive user feedback and useful suggestions for enhancements.          |

## Test scenario: tracking a morning jog

### Objective

To evaluate the app's performance in tracking the distance, route, time and calories burned during a morning jog.



CASE STUDY



### Steps for beta tester

- 1 Preparation:** The tester ensures the app is installed on their smartphone or connected to a compatible wearable device. The tester also ensures the device's GPS and any other required sensors are active and functioning.
- 2 Starting the activity:** The tester opens the app and selects the option to start a new outdoor running activity. The app should display a countdown or a 'start' button to begin tracking.
- 3 During the jog:** While jogging, the tester occasionally checks the app (if safely possible) to ensure it is displaying real-time data such as current pace, distance covered and time elapsed. The app should also maintain a stable connection with any wearable device being used.



SCHOOL-ASSESSED TASK TRACKER

Design brief

Project plan

Analysis

Folio of alternative design ideas

Development

Evaluation and assessment

Final submission



- 4 *Ending the activity:* Upon completing the jog, the tester stops the activity tracking on the app. The app should then display a summary of the jog, including total distance, total time, average pace and calories burned.

#### Reviewing the data

The tester reviews the accuracy of the recorded data by comparing it with known distances or with data from a reliable fitness device. The tester also checks the app for a visual representation of the route taken, if available.

#### Expected outcomes

- The app accurately tracks and records the duration, distance and route of the jog.
- The app provides a summary that closely matches the tester's expectations and external measurements.
- The user interface during the activity is clear, providing useful information at a glance.

#### Feedback points

- Any discrepancies between the app's data and known measurements.
- The responsiveness and accuracy of real-time data display.
- The ease of starting, pausing and stopping the activity tracking.
- The clarity and usefulness of the summary data provided after the jog.
- Any issues with the app's integration with wearable devices.

## Evaluating solutions

When evaluating a solution, efficiency and effectiveness can often be seen as competing goals. A process may be effective if it is thorough and checks all possible options. A similar task may be considered efficient if it is quick, uses few resources and is completed with minimal interactions with other activities. For a task to be more effective, it will often become less efficient.

Efficiency is gauged in terms of measurable quantities. Each part of your software solution will need data collected relating to these quantities:

- time taken
- the effort required
- resources accessed.

Once this data is collected, you can evaluate how efficient your solution may be.

Effectiveness relates to accuracy. What characteristics would a software solution need in order to be effective? The output has to suit the client's purpose and be in a format that follows appropriate conventions as described in the SRS. The information produced by the system needs to be accurate and comprehensive, without glaring omissions or superfluous information.

## Evaluation strategies

Strategies to evaluate solutions in terms of their efficiency and effectiveness hinge on several core features that ensure the evaluation is structured, objective and actionable. It is therefore important that the strategy include evaluation criteria, a time frame for the evaluation to occur, and identification of the person or people responsible for the evaluation to occur.

## Evaluation criteria

Evaluation criteria are the standards or benchmarks against which the solution's success or performance is measured. These criteria should be clearly defined, measurable and directly related to the goals and objectives the solution aims to achieve. In the context of software development, for example, appropriate evaluation criteria measure the efficiency as well as the effectiveness of the solution.

### Efficiency

The efficiency of a software solution refers to how well it utilises resources and performs tasks while minimising waste, cost and time. Efficiency directly impacts the software's performance, resource consumption and user experience. Three key components that contribute to the efficiency of software solutions are: the **cost of data and file manipulation**, its **functionality** and the **speed of processing**.

#### Cost of data and file manipulation

Efficiency in handling data and files within a software solution relates to how efficiently the software manages, processes and stores data and files, with a focus on minimising resource usage and associated costs. This ensures effective utilisation of memory and storage and reduces data management expenses.

You can consider the features of evaluation strategies used on solutions as 'What/When/Who'. 'What' is being used (evaluation criteria), 'when' is it going to be used (time frame) and 'by who' (responsibility).

## Cloud storage and data transfer efficiency in a cloud-based file-sharing service

### Scenario

Imagine you are a software engineer working on a cloud-based file-sharing service similar to Dropbox or Google Drive. Users of your service can upload, store and share files in the cloud. You want to optimise data and file manipulation to ensure efficiency in terms of both resource usage and cost.

### Efficiency measures

#### Data storage efficiency

Efficiently managing data storage will help to minimise costs in cloud-based services. You can implement data deduplication, where redundant data is identified and stored only once, reducing storage requirements and costs. Additionally, using data compression algorithms can further optimise storage utilisation.

#### Data transfer efficiency

Implementing efficient data transfer protocols and optimising network bandwidth usage can reduce the cost of data transfer. This includes using techniques such as differential syncing, where only changes to files are transmitted rather than the entire file.

#### File retrieval efficiency

Caching files that are accessed by users frequently can reduce data transfer and storage costs. Implementing content delivery networks (CDNs) can also help in serving files from geographically closer locations, reducing latency and improving efficiency.

## CASE STUDY



### SCHOOL-ASSESSED TASK TRACKER

 Design brief

 Project plan

 Analysis

 Folio of alternative design ideas

 Development

 Evaluation and assessment

 Final submission



### Application of efficiency measures

You design the cloud-based file-sharing service to use advanced data deduplication techniques, efficient data compression and a CDN for file distribution. When a user uploads a file, the system checks if it already exists in the storage. If it does, only metadata is updated, and redundant storage is avoided. When the user shares the file, the CDN serves it from the nearest location, reducing data transfer costs and improving download speed. This approach optimises both storage and data transfer costs, resulting in a more cost-efficient service.

## Functionality

Efficient functionality involves providing features and capabilities that align closely with user requirements without adding unnecessary complexity. This ensures that the software meets user needs effectively, without overloading it with unnecessary features.



## CASE STUDY

### E-commerce website checkout process optimisation

Imagine you are a software developer working on an e-commerce website. One of the critical aspects of your website is the checkout process, where users complete their purchases. You want to optimise functionality to ensure efficiency by offering essential features that avoid unnecessary complexity.

#### Efficiency measures

##### Streamlined user interface (UI)

The UI should be intuitive and straightforward, guiding users through the checkout process with minimal steps and distractions. Reducing unnecessary clicks and form fields is the focus of the UI design ideation process.

##### Configurable options

Instead of overwhelming users with too many choices, offer configurable options for products. For example, if selling a laptop, allow users to select specifications such as RAM and storage capacity. This simplifies product selection while maintaining flexibility.

##### Avoiding feature bloat

Resist the temptation to add numerous features that users may not find valuable. Focus on core functionality, such as adding products to the cart, providing shipping options, and secure payment processing. Avoid adding redundant or rarely used features that can complicate the process.

#### Application of efficiency measures

You design an e-commerce website that sells consumer electronics. To optimise functionality efficiency, you design the checkout process with the following features:

- 1 *One-page checkout:* Users complete their entire purchase on a single page, eliminating the need for multiple steps.
- 2 *Product customisation:* For laptops, users can customise RAM and storage capacity through simple drop-down menus, making the selection process quick and clear.
- 3 *Secure payment:* The checkout process includes secure payment processing with multiple payment options, ensuring a seamless and secure transaction.

By streamlining the user interface, providing configurable options and avoiding unnecessary features, your e-commerce website achieves functionality efficiency. Users can complete their purchases quickly and easily, reducing cart abandonment rates and improving the overall user experience.

## Speed of processing

The speed of processing refers to the software's ability to perform operations and respond to user input or system events quickly. Efficient processing enhances user satisfaction, reduces wait times and contributes to overall software performance.

## Real-time analytics for a financial trading platform

Imagine you are a software engineer working on a financial trading platform used by traders to execute high-frequency trades. In this context, optimising the speed of processing is absolutely critical to provide traders with real-time market data and execute orders swiftly.

### Efficiency measures

#### High-performance algorithms

Implement high-performance trading algorithms that can analyse market data and execute trades with minimal latency. These algorithms are designed to process large volumes of data quickly and make split-second decisions.

#### Data streaming and event processing

Utilise data streaming technologies to import and process market data in real time. Event processing systems can identify critical events, such as price changes or order updates, and trigger immediate actions.

#### Hardware acceleration

Consider hardware acceleration techniques, such as using GPUs (graphics processing units), to perform complex calculations rapidly. Hardware acceleration can significantly speed up computational tasks, improving processing speed.

### Application of efficiency measures

If this design is implemented, the platform will employ advanced trading algorithms designed to analyse market data, identify trading opportunities and execute orders within milliseconds. These algorithms are optimised for speed and efficiency, allowing traders to capitalise on market fluctuations.

Real-time market data is streamed into the platform, where event processing engines identify critical events, such as price movements or news releases. When a significant event is detected, the platform can automatically execute predefined trading strategies without delay.

To accelerate complex mathematical calculations required for risk assessment and order execution, the platform leverages GPUs. This hardware acceleration significantly reduces the time required to process trading-related computations.

By optimising the speed of processing through high-performance algorithms, real-time data streaming, event processing and hardware acceleration, the financial trading platform ensures that traders have access to the latest market information and can execute orders swiftly.

CASE  
STUDY



SCHOOL-ASSESSED TASK TRACKER

Design brief

Project plan

Analysis

Folio of alternative  
design ideas

Development

Evaluation and  
assessment

Final submission

## Effectiveness

The effectiveness of a solution, whether it is a software application, a document or a product, refers to its ability to achieve its intended goals and objectives. This involves designing for and considering attributes that collectively determine how well the solution fulfils its purpose and meets the needs of its users. The key factors contributing to the effectiveness of solutions are **accessibility**, **accuracy**, **attractiveness**, **clarity**, **communication of message**, **completeness**, **maintainability**, **readability**, **relevance**, **timeliness** and **usability**.

### Accessibility

Accessibility refers to the ease with which users can access and interact with the solution. An effective solution ensures that it is readily available to all users, including those with disabilities, by providing features such as screen readers, keyboard navigation and compatibility with assistive technologies.

### Accuracy

Accuracy reflects how well the solution produces correct and reliable results. An effective solution minimises errors, inconsistencies and inaccuracies in its output, data or content.

### Attractiveness

Attractiveness is the visual and aesthetic appeal of the solution. An effective solution considers user preferences, design principles and user interface elements to make the solution visually appealing and engaging.

### Clarity

Clarity ensures information, instructions or content within the solution are easily understandable. An effective solution uses clear language, logical organisation and user-friendly layouts to convey its message or purpose.

### Communication of message

Effective communication ensures that the message is conveyed clearly, persuasively and in a manner that resonates with the target audience.

### Completeness

Completeness refers to the inclusion of all necessary elements or components within the solution. An effective solution is comprehensive and includes all relevant information, features or functionalities required to fulfil its purpose.

### Maintainability

Maintainability involves the ease with which the solution can be updated, modified or extended. An effective solution is designed to facilitate future changes or enhancements, reducing the cost and effort of maintenance.

### Readability

Readability is crucial for written content, documents or user interfaces. An effective solution ensures that text and content are easy to read, with appropriate fonts, spacing and formatting.

### Relevance

Relevance ensures that the solution remains aligned with the evolving needs and expectations of its users. An effective solution adapts to changing circumstances and remains pertinent over time.

## Timeliness

Timeliness involves delivering the solution or its content at the right time. An effective solution ensures that information, updates or actions are provided in a timely manner to meet user requirements.

## Usability

Usability is perhaps one of the most significant factors in effectiveness. It assesses how easily users can interact with the solution to accomplish their goals. An effective solution prioritises user-friendly interfaces, intuitive navigation and efficient workflows.

**TABLE 5.12** Examples of effectiveness features

| Effectiveness feature    | Example  |
|--------------------------|--|
| Accessibility            | A website provides alternative text for images and supports keyboard navigation, ensuring it is accessible to users with disabilities.                                   |
| Accuracy                 | A weather forecasting app consistently provides precise weather predictions with minimal errors.   |
| Attractiveness           | Graphic design software features a visually appealing user interface with modern design elements and customisable themes.  |
| Clarity                  | An online document editor uses clear and concise formatting options to help users create well-organised and readable documents.  |
| Communication of message | Marketing automation software effectively delivers personalised email campaigns to engage and convert customers based on their preferences.                              |
| Completeness             | An accounting software includes all necessary financial tools, such as income tracking, expense management and tax calculations, for comprehensive financial management. |
| Maintainability          | An integrated development environment (IDE) provides extensive documentation and version control features to facilitate software project maintenance.                    |
| Readability              | A code editor uses legible fonts, syntax highlighting and code folding options to enhance the readability of programming code.   |
| Relevance                | A news aggregator app constantly updates its content to display the latest news articles relevant to the user's selected topics and interests.                           |
| Timeliness               | A stock trading platform offers real-time stock price updates and timely execution of buy/sell orders to help traders make quick decisions.                              |
| Usability                | A mobile banking app offers an intuitive user interface, straightforward account management and fingerprint authentication for ease of use by customers.                 |

## Time frames

The time frame for evaluation refers to the specific periods at which the evaluation of the solution will take place. This could be a one-time review after the software has been used for a particular amount of time, or a series of evaluations at different time periods after the software has been deployed.

Time frames should be realistic and allow enough time for the solution to have an impact, but also be timely enough to make necessary adjustments or decisions. For ongoing projects, formative evaluations might occur at key milestones, while a summative evaluation might take place at the project's conclusion.

Formative evaluations are diagnostic and supportive, focusing on identifying strengths and weaknesses to refine and enhance the software. Summative evaluations aim to provide a conclusive assessment of the solution's success in meeting its objectives and are often used for accountability purposes or to inform decisions about the future of the software.

### SCHOOL-ASSESSED TASK TRACKER

 Design brief

 Project plan

 Analysis

 Folio of alternative design ideas

 Development

 Evaluation and assessment

 Final submission

## Responsibility

Assigning responsibility involves determining who will be accountable for conducting the evaluation and ensuring that the evaluation criteria are met. This could be an internal team or external evaluators, depending on the nature of the solution and the required expertise.

Responsibilities should be clearly defined to include data collection, analysis and reporting. In collaborative environments, responsibilities might be distributed among different stakeholders, with a lead evaluator overseeing the process.

Typically, the following people have responsibilities during an evaluation:

- the project team – team members are often responsible for ongoing monitoring and intermediate evaluations
- external evaluators – specialists or consultants who might be brought in for objective, unbiased final evaluations, particularly for large-scale or high-stakes projects
- stakeholders – end users or clients might also play a role in the evaluation process, providing feedback or participating in surveys and interviews.

## Applying evaluation criteria

Applying evaluation criteria effectively to solutions involves a structured approach to assess how well a solution meets its intended objectives and performance standards. Techniques that can be used to do this include rubrics, checklists, benchmarks, user feedback and expert reviews (refer to Table 5.13).

**TABLE 5.13** Evaluation criteria techniques and examples

| Technique      | Definition  | Example   |
|----------------|---|---|
| Rubrics        | Structured scoring guides used to assess the quality of a software solution by breaking down its components into specific criteria and levels of performance  | A rubric for evaluating a web application might include criteria such as user interface design, functionality, security and performance. Each criterion would be scored on a scale, allowing for a comprehensive assessment.                        |
| Checklists     | Lists of items or criteria used to determine if a software solution meets specific requirements or is complete  | For a mobile app release, a checklist could include items such as feature implementation, bug fixes, compatibility testing and documentation. Each item must be checked off before the app is considered ready for launch.                          |
| Benchmarks     | Reference points or industry standards used to measure the performance or quality of a software solution against established norms or competitors in the software development field                                   | When developing a new software product, developers may benchmark its performance against similar applications in the market, comparing factors like response time, resource usage and user satisfaction.  |
| User feedback  | Gathering opinions, comments and insights from users or potential users of a software solution to identify areas for improvement in terms of user experience, functionality and overall quality                       | In the development of a mobile game, conducting focus groups with gamers can provide valuable user feedback on gameplay, graphics, controls and in-app purchases, helping developers refine the gaming experience.                                  |
| Expert reviews | Involving knowledgeable individuals, such as software architects or experienced developers, to critically assess a software solution's quality, code structure, adherence to best practices and overall effectiveness | An expert review of a software application may involve a senior software engineer evaluating the codebase, looking for code maintainability, architectural flaws and adherence to coding standards, and providing recommendations for improvements. |

# Recording the progress of projects

## Adjustments to tasks and time frames

In the first half of this SAT (Unit 3, Outcome 2), we discussed how to manage a project, and why project management is important. In Unit 4, you will continue to use these project management techniques to complete development and evaluation of your software solution.

Very few projects ever proceed perfectly in line with the project plan. One unexpected rainy day, a hard disk crash or a sick day for a key worker can slow down a work team enough to affect tasks, other teams and deadlines. Project plans are not fixed, and Gantt charts should be regularly modified to reflect reality.

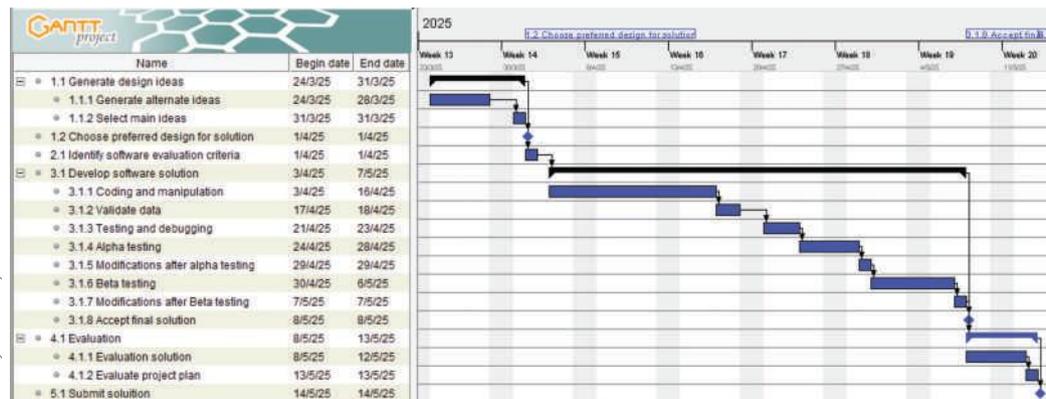
Project plans are living documents. Tasks that run overtime may have resources added to them or be modified so they finish earlier. Bad weather may force changes to scheduling, so indoor tasks may be completed instead of outdoor work. Late deliveries of equipment may cause a project manager to move people off one task and assign them to another one that can proceed without the deliveries.

## Annotations

A project plan should be annotated to give reasons for changes to task schedules or resourcing priorities. When the project is later assessed, these annotations will serve as valuable lessons before undertaking the next project. Annotations might also be added by other project leaders to advise the team of significant news or concerns. Annotations could be handwritten or added as notes in the Gantt chart itself.



**Resource**  
GanttProject chart: a project plan for the chosen solution



Courtesy of GanttProject

**FIGURE 5.13** Gantt chart: a project plan for the chosen solution

### Gantt chart

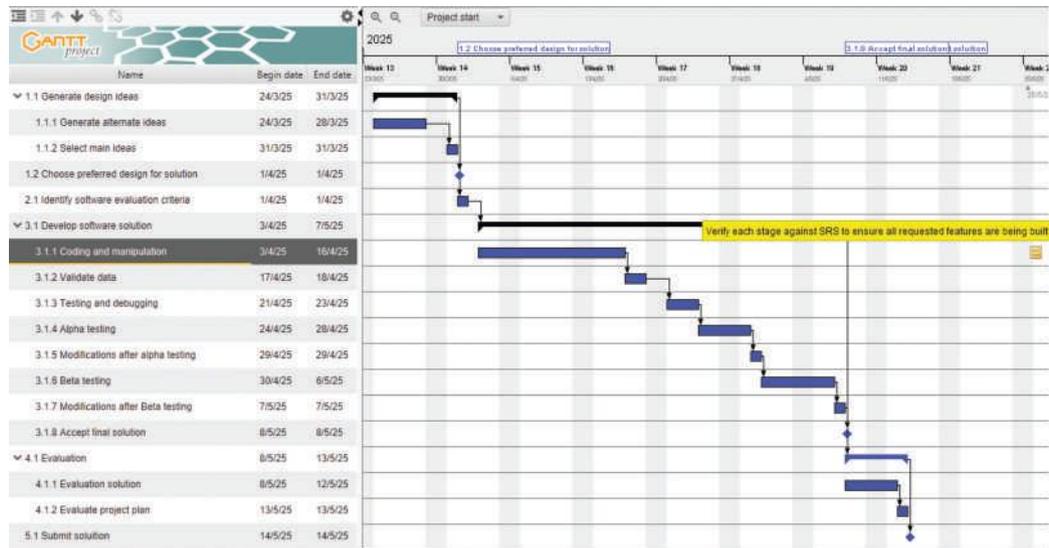
- 1 Modify the project plan in Figure 5.13 by adding two days to the '2.1 Identify software evaluation criteria' stage.  
Redraw Figure 5.13, either by hand or electronically, or modify the electronic file on student Nelson MindTap.
- 2 Do you need to adjust the rest of the project so that the final date for Submit solution does not move forward by more than one day?
- 3 Justify how you have maintained the project end date.



**RESEARCH**

SCHOOL-ASSESSED TASK TRACKER

Design brief  Project plan  Analysis  Folio of alternative design ideas  Development  Evaluation and assessment  Final submission



Courtesy of GanttProject

**FIGURE 5.14** Gantt chart with notes added

**Automated Excel data validation**

Select the cell for the drop-down menu. Go to Data tab on the ribbon, then Data validation. On the Setting tab, in the Allow box, click List. Click in the Source box, then select your list range, or a comma-separated list will provide option choices for that cell.

Handwritten annotations are also acceptable. One strategy for Unit 4, Outcome 1 is to print out the final version of the Gantt chart and complete the printed copy. Laminating before annotation adds extra presentation finesse. Make a printout of the PDF and annotate this copy or use PowerPoint and add callouts to a screenshot. Annotation needs to be quick, simple and effective. Where do the comments and observations, insights and suggestions come from?

| Issue # | Date       | Reported by  | Item/Functionality                   | Description   | Issue type | Priority | Impact | Suggested fix   | Status  |
|---------|------------|--------------|--------------------------------------|---|------------|----------|--------|---|---|
| 1       | 2025-05-09 | Developer    | input box submit button              | spacing too close to text box   | Usability  | Medium   | High   | increase spacing, introduce colour, add label with meaning  | Resolved, adjusted size, colour and labelling                               |
| 2       | 2025-05-09 | Developer    | title                                | the font size of the main input screen is too small, hard to read   | Design     | High     | High   | adjust the size and colour contrast   | Resolved, larger font, darker colour  |
| 3       | 2025-05-09 | Developer    | what to do is not clear              | after reading the introduction screen the section headings are ok but items that are required and not optional are not clearly marked | Usability  | Medium   | Medium | adjust the size and colour shading of the button. Add a label   | Partly resolved, need to work out how to include a label inside             |
| 4       | 2025-05-09 | Developer    | quality of image                     | image is of poor quality  | Data       | Medium   | Low    | locate better quality logo file. Re-capture screenshot or create brand new logo if required.          | Still looking for better quality image. Recreating original yet to be done. |
| 5       | 2025-05-10 | Alpha tester | sources and privacy statement issues | there is no acknowledgement of sources, destination of data or privacy statement.   | Data       | Low      | Low    | Add source citation link to Privacy terms and conditions. Put in lower left corner of opening screen. | Yet to be completed.  |

Used with permission from Microsoft

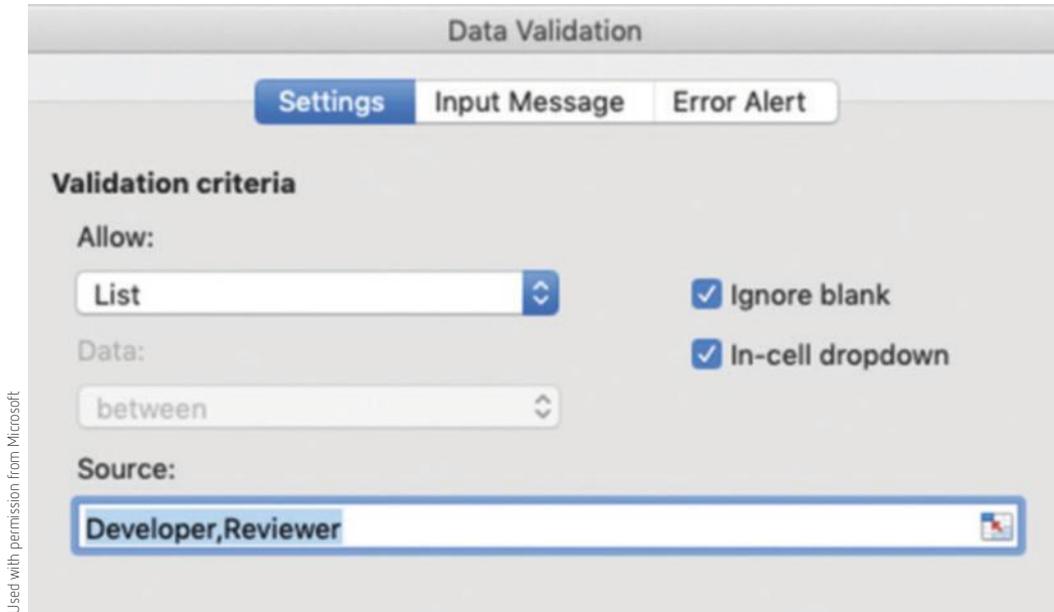
**FIGURE 5.15** Project log template created in Microsoft Excel. Columns C, F, G and H are data validation drop-down lists. Columns G, H and J also have conditional formatting applied to indicate priority and impact by colour: low, medium, high, critical and showstopper. This template is available for download through your NelsonNet login.



Resource  
Project log template

**Keeping logs**

Project logs are a record of all the small and large steps a project takes on its way to completion. The log is usually in electronic form and is shared with all project leaders. It may be created using specialist log software, in a weblog ('blog') for example, or with Microsoft Word or Excel. It could be shared online using Google Drive or similar technology.

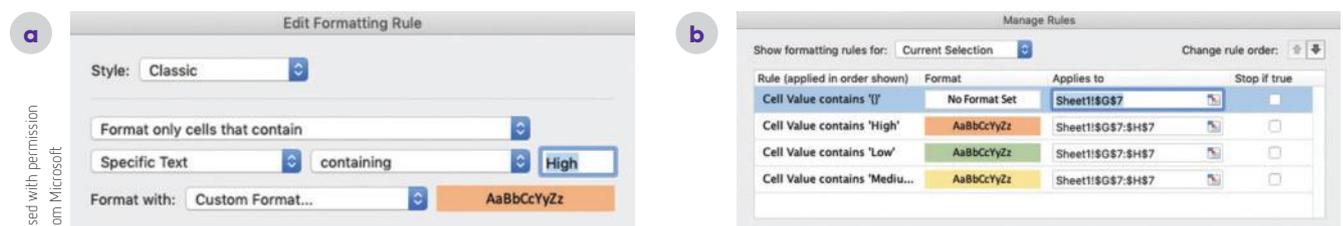


**FIGURE 5.16** Excel data validation dialog

### Conditional formatting

The spreadsheet has many inbuilt functions that automate the formatting of the appearance. Once initial settings have been chosen and tested, the conditionally formatted cells can be copied down the column, ready for the data to be placed.

As shown in Figure 5.17, the priority cells can be conditionally formatted to give you a drop-down menu choice of options. High, Medium and Low data will appear once you begin typing, but the formatting of the cells relies on you manually formatting each individual cell. Alternatively, format the first cell, then copy format and pasting across all similar cells. In this example, the first row is conditionally formatted for Column C: Reported by [Developer, Reviewer], then column F: Issue type [Usability, Design, Data,], column G: Priority [High, Medium, Low], column H: Impact [High, Medium, Low], column J: Status [Resolved, Partly resolved, Not resolved].

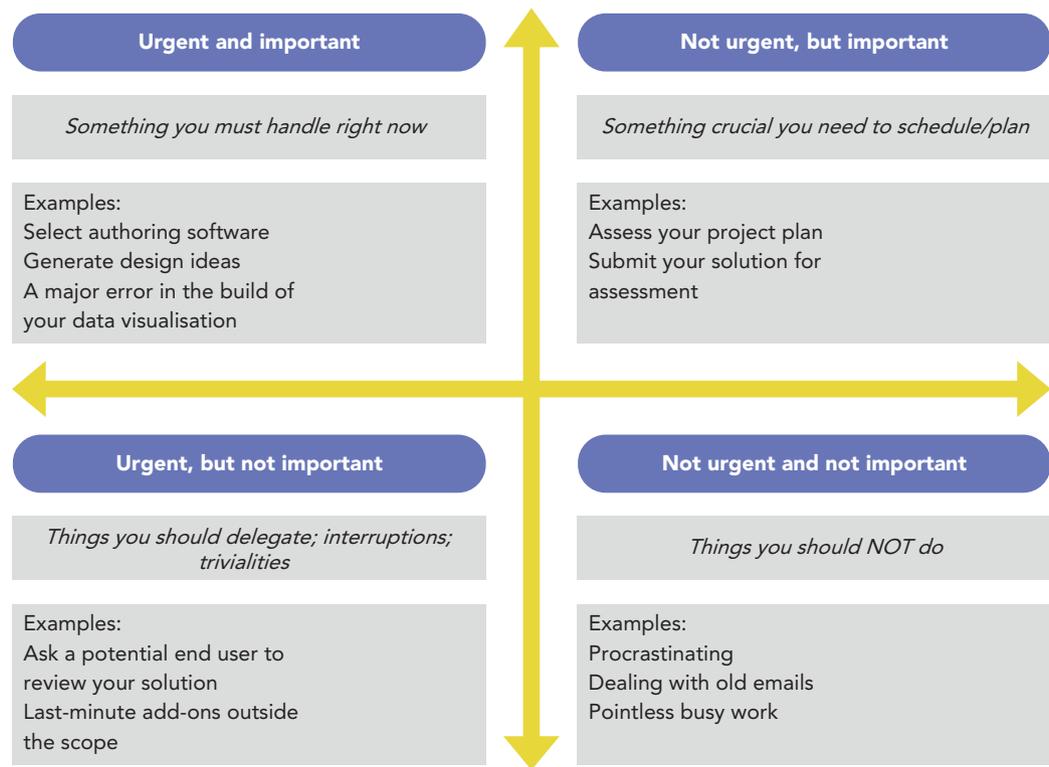


**FIGURE 5.17**  
An example of automated cell formatting in Excel:  
**a** New rule; **b** Managing rules

A project log helps a team keep track of the status of project tasks, which team member is responsible for the tasks, and when deadlines and milestones are due. It can also help you keep track of the work you are doing on your solution. In conjunction with your Gantt chart, a project log can be a valuable tool to help you manage your project efficiently. It can include time and date stamps, comments on progress, project risks, issues that arise, ideas for solutions, actual solutions, explanations of decisions, results of testing, forecasts and warnings, task changes, photos and future action needed. A project log is like a diary that records the complete history of a project. It can be recorded online using a blog (weblog). Keeping a

weekly blog that records daily achievements is a very effective method of gathering evidence. Your final Gantt chart can be annotated with entries from your blog, with references to the URL.

If you find yourself struggling to divide up the tasks by priority, creating a priorities quadrant may also be of use (see Figure 5.18).



**FIGURE 5.18** Example of a priorities quadrant

## Efficiency tips

You have a limited amount of time to complete the project, so you need to ensure that you use your time as efficiently as possible. Your first priority is to make your proposed solution work. While the attractiveness of your solution is an assessment criterion, appearance is worth much less than the quality of its information. You need to make sure that you allocate your time appropriately to reflect this priority.

One way of doing this is to use a plain text heading and then insert all of the key information. If you have time later, you can go back and design a graphic heading and modify the fonts. Remember, a beautiful solution with incomplete content is a waste of time.

It is easy to get sidetracked ‘fiddling’ with small details. Spending half an hour in Photoshop cutting a figure from its background is an inefficient use of your time, and this will be reflected in your assessment. You need to complete the less exciting jobs, such as typing data and testing that all links work.

It may help to make an agreement with yourself to do just a little of the task. It will not be finished, but at least you will have made some progress. This will boost your self-confidence, and you may even decide to do a little bit more while you are at it. The hardest part of most jobs is taking the first step. Once the first step has been taken, it is usually much easier to continue.

# Evaluating the efficiency and effectiveness of solutions and project plans

## Evaluation

**Evaluation** is the final stage of the problem-solving methodology. It checks how well the solution is satisfying the needs of the user it was originally created for.

Remember, you will be evaluating your programming solution and your project plan as part of your assessment in Unit 4, Outcome 1. When evaluating your solution, you need to refer to the evaluation criteria you developed during the design phase. For each criterion, you will choose a method to evaluate it.

Evaluation is not the same as testing; its purpose is distinctly different. By the time evaluation begins, the solution has already been proved to work properly and its functionality is no longer in question.

Evaluation can best be understood by saying what it does not do:

- Evaluation does not test that a solution is working properly. That should have been done during testing.
- Evaluation does not enter test data to check that output is accurate. That should have been done during testing.
- Evaluation does not use a stopwatch to time how long a process takes. That should have been done during testing.
- Evaluation does not perform checks with immediate results, such as pulling out the power plug to see if a system loses data. That should have been done during testing.

Evaluation looks at a solution's performance over time in terms of the evaluation criteria.

## What to evaluate

**Evaluation criteria** are determined during the design phase of the problem-solving methodology, and are based on the most important qualities that the solution is expected to have when it is designed. For example, for your solution, essential criteria include the ability to quickly enter and record data. You should evaluate the features that would, if they were not achieved, render your solution unsatisfactory.

As mentioned earlier in this chapter, evaluation criteria fall under two headings: efficiency and effectiveness.

- 1 Efficiency can be measured in terms of how much time, cost or effort has been spent on the solution. This includes consideration of cost of data and file manipulation, cost of functionality and the speed of processing.
- 2 Effectiveness includes accessibility, accuracy, attractiveness, clarity, communication of message, completeness, maintainability, readability, relevance, timeliness and usability (see page 220).

You may conduct interviews or create questionnaires to seek feedback from your peers regarding the criteria you were expecting to achieve. This feedback could be offered as evidence of evaluation.

Evaluation is not the same as testing; its purpose is very different.

For your software solution, you may decide to use annotated screenshots to demonstrate how key criteria were satisfied, such as readability being assisted by consistency of formatting. If recordings of animated screen activity are needed to prove that you *satisfied* an evaluation criterion, you could use screen recording software such as Camtasia.

### Evaluation methods

For each evaluation criterion for your solution, there must be a corresponding evaluation method that can measure the degree to which the criterion has been achieved.

- Objective (fact-based, measurable) results are solid facts that are hard to argue with. Measure whenever you can.
- Subjective results (emotions, opinions, personal judgements) can be gained from interviews, questionnaires and surveys. These should only be used when objective measurement is not possible or practical, such as when evaluating how comfortable users feel when using an unfamiliar programming solution.

Remember: Evaluation assesses your solution's performance over time. It is not instantaneous, like testing is. Emotional or judgemental feedback is only gathered on appropriate criteria. For example, asking questions such as, 'Is the new system faster than the old one?' is not helpful as you would not be able to trust the accuracy of the answer.

### When to evaluate

Evaluation occurs after the solution has been in regular use for some time, when it is well 'bedded in' and its users are familiar and comfortable with it. A few months of regular, daily use is typical.

**TABLE 5.14** Typical evaluation methods

| Criterion   | Method  |
|---|---|
| Accuracy (effectiveness)                                      | Check the complaints log and count the complaints from staff or customers about inaccurate information received from the system over the past three months.   |
| Reliability (effectiveness)                                   | Record the number of faults in the system's error log.  |
| Security (effectiveness)                                      | Record the number of successful and blocked attempts made to penetrate system security.   |
| Attractiveness, pleasure, comfort, confidence (effectiveness) | Interview users.  |
| Ease of use, usability (effectiveness)                        | Record the number of times the help file was accessed (indicating that the solution may not have been intuitive).<br>Record the number of user errors. (A solution that is hard to use tends to cause users to make mistakes.)<br>Check the help desk records to see how often users asked for help or complained about the solution.<br>Ask users to complete a questionnaire about their feelings regarding the system's usability. |
| Speed of processing (efficiency)                              | Refer to system logs and the number of transactions the system handled over three months compared to the previous system.   |
| Cost in terms of profit (efficiency)                          | Ask the accountants to tally the new system's running costs over time. Check organisational profit figures and see if profit has increased.   |
| Cost of labour (efficiency)                                   | Record the number of staff hours spent operating and maintaining the system compared with the previous system.  |

Evaluating a solution too soon can lead to negative feedback, because users may not be used to it, and so may be slow and prone to making errors. When users are comfortable and skilled with the solution, their feedback is likely to be more positive.

If a system is used infrequently but its success is critical to the organisation (such as creating school reports, or managing a flood of tax returns at the end of the financial year), evaluation may be done immediately after the system is used.

You will not have a long period of time for users to explore your solution. Instead, ensure that you evaluate your own solution especially thoroughly during beta testing, which should have included asking a variety of potential end users (such as classmates, friends, teachers and family members) to evaluate your solution.

## Solutions

Evaluation of your software solution began with the SRS. That document recorded all the client's expectations and specifications. Those performance statements can be converted into evaluation criteria. It is against each of those criteria that the evaluation will be performed.

For example:

→ Specification (SRS) statement: The menu will be easy to understand.

→ Evaluation criterion: Is the menu easy to understand?

A Likert (pronounced lick-ert) scale would be the most appropriate conversion of opinion to a rating scale. To extend the authority of such a user survey, several users need to be interviewed or questioned about the usability of the software solution.

There may be too many specifications contained within the SRS, so you will need to prioritise the most significant.

## Project plans

Organisations invest a great deal of time, money and labour into projects, so they tend to look back at their project plans to evaluate how the planning went. This is important not only to reflect on what has occurred, but also to better inform the planning of future projects, as they can avoid repeating mistakes.

## Factors influencing project plan effectiveness

The effectiveness of project plans is an important contributor to the success of a project, particularly in terms of achieving the objectives of the project on time and within budget. Various factors can impact this effectiveness, with **scope creep**, **personnel changes** and **technical issues** being particularly significant.

### Scope creep

Scope creep involves the gradual expansion of project scope through the addition of features, requirements or tasks not initially planned for, without corresponding increases in resources. This can lead to delays, budget overruns and a strain on resources, making it challenging to adhere to original plans and deadlines. Mitigating scope creep requires a clear, well-documented project scope agreed upon by all stakeholders, alongside regular reviews and strict change control processes to manage unauthorised expansions.

## Personnel changes

Personnel changes, such as turnover or reassignment of team members, can disrupt team dynamics and lead to a loss of knowledge, requiring time for new members to acclimatise. These changes can affect the project's momentum and the effectiveness of the original plan. To alleviate the impact of personnel changes, cross-training, thorough documentation and a flexible project plan are essential, along with a well-defined induction and training process for new team members.

## Technical issues

Technical issues can range from unforeseen software bugs and hardware failures to integration challenges or limitations in the technology used that were not identified during planning. Such issues can halt or delay progress and increase costs, potentially requiring a re-evaluation of the project's technical feasibility. Addressing technical issues effectively involves conducting comprehensive technical assessments and feasibility studies early in the planning phase, implementing a robust risk management process, and maintaining a portion of the project budget for unforeseen technical challenges.

## Assessing the effectiveness of a project plan

Assessing the effectiveness of a project plan allows a project manager to understand how well the project adapted to challenges and met its objectives. One key technique involves a thorough review of the changes made to the project plan during the project's life cycle. This review process not only considers the sheer number of adjustments but also considers the reasons behind these changes and their subsequent impact on project completion.

When reviewing the number of changes made to the project plan, it is important to differentiate between minor tweaks and major alterations. Minor adjustments might be expected in the normal course of project execution, reflecting a healthy adaptability within the project management process. However, a high number of significant changes could indicate initial planning oversights or a lack of understanding of project requirements and constraints. Each change, whether large or small, should be documented, noting the date, nature and a brief explanation as to why the decision to make the change was made.

Understanding the reasons behind changes is important. Changes could be necessitated by external factors such as legal changes, stakeholder/client demands or feedback from users. Alternatively, they might stem from internal factors such as technical challenges, resource constraints or personnel changes. Examining the causes of these changes helps in identifying patterns or recurring issues, which can offer insights into the project planning and execution process's robustness. It can also highlight areas of the project plan that were particularly vulnerable to assumptions or uncertainties.

Discussing the impact of changes on project completion is required when assessing the effectiveness of a project plan. Each change has the potential to affect the project's scope, timeline, budget and quality. Evaluating the impact involves looking at how changes affected the project's critical path, whether they led to scope creep, how they influenced project costs, and if they had any effects on the quality of the final solution. This evaluation helps in understanding the plan's flexibility and the project team's responsiveness to unforeseen events.

Assessing the effectiveness of a project plan through a detailed review of changes provides valuable insights into the strengths and weaknesses of the planning and execution of the project plan. It highlights areas where initial assumptions may have been inaccurate, where

more robust risk management could have been beneficial and where the project team excelled in adapting to new information. This approach ensures that lessons learned are integrated into future projects, enhancing an organisation's project management capabilities and increasing the likelihood of success in subsequent endeavours.

## Data to collect

When you are assessing the effectiveness of your project plan, you can collect various types of data to provide a comprehensive evaluation. Some of these types are listed in Table 5.15.

**TABLE 5.15** Data to consider when writing a project plan evaluation

| Data collected              | What to evaluate/consider  |
|-----------------------------|--|
| Change log data             | <p>Number of changes: Total number of changes made to the project plan.</p> <p>Type of changes: Categorise changes into minor adjustments (e.g. timeline tweaks, minor resource allocation changes) and major alterations (e.g. scope changes, significant adjustments).</p> <p>Date of changes: Document the dates when changes were made to track their frequency and timing.</p> <p>Nature of changes: Detailed descriptions of each change, specifying what aspect of the project plan was altered.</p> <p>See Figure 5.19 for a sample project plan change log.</p> |
| Reasons for changes         | <p>External factors: Changes due to external influences such as legal requirements, stakeholder demands or user feedback.</p> <p>Internal factors: Changes driven by internal factors such as technical challenges or resource constraints.</p> <p>Explanation: Brief explanations for why each change was made, highlighting the driving factors behind the adjustments.</p>  |
| Impact on project           | <p>Scope: How changes affected the project scope, such as with the addition or removal of features.</p> <p>Timeline: Impact on the project schedule, including any delays or accelerations in the timeline.</p> <p>Quality: Effects on the quality of the final product, including any compromises or improvements.</p>  |
| Critical path analysis      | <p>Changes to the critical path: How changes impacted the project's critical path, including any shifts in critical tasks.</p> <p>New critical tasks: Identification of any new critical tasks that emerged due to changes.</p>  |
| Client/stakeholder feedback | Client/stakeholder feedback: Responses and feedback from stakeholders regarding changes and their satisfaction with how the project adapted.   |
| Lessons learned             | <p>Successes: Areas where the project plan was effective in adapting to changes and meeting objectives.</p> <p>Areas for improvement: Identified weaknesses in the initial planning or execution process that could be improved in future projects.</p>  |

|   | A         | B          | C              | D                                   | E                   | F               | G                  | H                 | I                    | J                    |
|---|-----------|------------|----------------|-------------------------------------|---------------------|-----------------|--------------------|-------------------|----------------------|----------------------|
| 1 | Change ID | Date       | Type of Change | Description                         | Reason for Change   | Impact on Scope | Impact on Timeline | Impact on Quality | Critical Path Change | Stakeholder Feedback |
| 2 | 1         | 10/04/2025 | Minor          | Extended deadline by 2 days         | Resource constraint | None            | +2 days            | None              | No                   | Accepted             |
| 3 | 2         | 15/07/2025 | Major          | Added new feature X                 | Client demand       | Increased       | +10 days           | Enhanced          | Yes                  | Positive             |
| 4 | 3         | 5/08/2025  | Minor          | Reallocated team member to Task Y   | Technical challenge | None            | None               | None              | No                   | Neutral              |
| 5 | 4         | 10/08/2025 | Major          | Reduced scope by removing feature Z | Time constraint     | Decreased       | -5 days            | None              | Yes                  | Negative             |

**FIGURE 5.19** Sample project plan change log

## Next steps

Complete the chapter summary material, including 'Preparing for Unit 4, Outcome 1', then, after consultation with your teacher, begin to prepare your solution for submission.

# 5

## CHAPTER SUMMARY

### Essential terms

**accessibility** ensures that software can be used by people with a wide range of abilities and disabilities

**accuracy** the degree to which software correctly performs its intended functions without errors

**alpha testing** an early stage of testing conducted by the development team within the development environment

**application programming interfaces (APIs) and libraries** sets of routines, protocols and tools for building software and applications, enabling different software components to communicate

**artificial intelligence (AI)-based assistants** software entities designed to perform tasks or services for an individual based on commands or questions

**attractiveness** the aesthetic appeal of the software's user interface, contributing to user satisfaction

**beta testing** the phase of software testing where a sample of the intended audience tests the software in a real environment

**breakpoints** markers set within a development environment to pause program execution for debugging purposes

**clarity** the quality of being easily understood, particularly in the context of code and user interfaces

**clear and concise code** code that is written in a straightforward and easily understandable manner, avoiding unnecessary complexity

**code repositories** central locations where code is stored and managed, allowing for version control and collaboration

**communication of message** the effectiveness with which a software conveys information or instructions to its users

**completeness** the extent to which all necessary features and functionality are included in the software

**contiguous memory** memory locations sequentially aligned in address space, allowing for efficient data storage and access

**cost of data and file manipulation** the resources required to perform operations on data and files, including time, computational power and storage

**debugging** the process of identifying, analysing and removing errors or bugs from software

**DevOps** a set of practices that combines software development (Dev) and IT operations (Ops) to shorten the development life cycle and provide continuous delivery

**evaluation** the final stage of the problem-solving methodology. It checks how well the solution is satisfying the needs of the user for which it was originally created.

**evaluation criteria** performance criteria made from the expectations and specification

**functionality** the range of operations that can be performed by a software system as defined by its requirements

**high-fidelity prototypes** advanced prototypes that closely mimic the final product, offering a detailed and interactive representation of the user interface and functionality

**internal documentation** written text or annotations within code intended to explain how the code works and how to use it

**low-fidelity prototypes** basic versions of a product, often simplistic and non-interactive, used to explore ideas and concepts in the early stages of design

**maintainability** the ease with which software can be modified to correct faults, improve performance or adapt to a changed environment

**non-contiguous** memory or data storage that is not sequentially aligned, potentially leading to inefficiencies in access and management

**personas** fictional characters created based on user research to represent the different user types that might use a service, application, product, site or brand

**personnel changes** modifications or transitions in the team composition that can impact the development process and timelines

**readability** the ease with which text (including code) can be read and understood

**relevance** the degree to which software meets the current needs and requirements of its users or market

**scope creep** the tendency for a project's requirements to increase over time, often leading to delays and budget overruns

**speed of processing** the rate at which a software system can execute commands and perform operations

**technical issues** challenges or problems related to the technology being used in software development, potentially impacting project progress

**testing** checks the accuracy of information outputs

**testing table** a commonly used way to record evidence of functionality testing

**timeliness** the ability of software to provide information or functionality when it is needed, without undue delay

**usability** the ease with which a user can interact with a software system to achieve their goals efficiently and satisfactorily

**user-centred design** a design approach that focuses on understanding the needs, wants and limitations of end users at each stage of the design process

**validation** checks the reasonableness of data inputs

**version control** the method that keeps track of the current, most up-to-date document through a drafting process

## Important facts

- 1 Characteristics of efficient and effective solutions involve user-centred design, clear and concise code and detailed internal documentation.
- 2 Characteristics of data types, structures and sources need to consider input, storage and output.
- 3 Features of a programming language include local and global variables, constant variables, data types, instructions and control structures, arithmetic, logical and conditional operators, graphical user interfaces, functions, methods, objects, classes and access modifiers.
- 4 Control structures are sequence, selection and iteration/repetition.
- 5 Access modifiers are public, protected and private.
- 6 Approaches to software development include using code repositories, APIs, libraries and AI assistants.
- 7 Validation techniques include existence checking, type checking and range checking.
- 8 Debugging and alpha testing involves using breakpoints, commenting out code, using relevant test data and applying test cases.
- 9 Beta testing involves constructing test plans and scenarios, observing test scenarios and documenting test results.
- 10 Features of evaluation strategies focus on evaluation criteria, time frames and who is responsible.
- 11 Factors that influence the effectiveness of a project plan involve how much scope creep has occurred, personnel changes and technical issues.
- 12 Recording changes to the progress of a project involves documenting adjustments to tasks and time frames, making annotations on project plans, and monitoring and documenting progress using a log or journal.
- 13 Techniques for assessing the effectiveness of a project plan include reviewing the number of changes and modifications made to the project plan, considering the reasons those changes were necessary and considering the impact those changes had on the completion of the project.



## TEST YOUR KNOWLEDGE

### Efficient and effective solutions

- 1 Effectiveness is often subjective. Who will you involve when determining the effectiveness of your software solution? What questions will you ask so you know they didn't just make up their response?
- 2 Efficiency is usually measured. What can be measured to demonstrate the efficiency of a software solution?
- 3 Explain what is involved when following a user-centred design process.
- 4 Why is it important to consider accessibility and inclusivity early in the design process?
- 5 Create a persona to represent a user of the e-commerce website outlined in the case study on page 218.
- 6 Explain how internal documentation assists the efficiency and effectiveness of a solution.

### Data types, structures and sources

- 7 What type of relationship does a data type or structure have with the method of input? Explain using an example.
- 8 What is the difference between contiguous and non-contiguous memory?
- 9 Using examples, explain the difference between the output of an array and the output of a record.
- 10 When is a data dictionary useful for programming? Explain how you would convince someone to use a data dictionary when they think the code itself is enough to keep track of variable names and data types.
- 11 In what data type or structure are date/time values typically stored? Why?

### Features of a programming language

- 12 Briefly explain how any programming language works. Include reference to control structures.
- 13 What is iteration? Why is it useful? Give an example using an array with 10 elements.

### Approaches to software development

- 14 Aside from version control, what are two benefits of using a code repository?
- 15 Discuss how APIs and libraries can assist a software developer. Use an example as part of your discussion.
- 16 Select and use one AI-based tool that can assist in software development. Use the tool to generate internal documentation for some of the code you have written.
- 17 Use an AI-based tool such as GitHub Copilot to assist you in writing code that scrapes data from a website and stores it in a CSV file. After doing so, answer the following:
  - a What suggestions were helpful? Were there any challenges?
  - b Review the CSV file created to ensure the data was scraped and stored correctly.
  - c (Optional) Extend the scraper to handle pagination or scrape additional details.



## Software testing

- 18** How is validation different from testing?
- 19** Create a table listing what validation does and does not do.
- 20** List three types of validation.
- 21** What are some techniques for checking code?
- 22** Explain how and when alpha testing occurs. How is that different from beta testing?
- 23** If you cannot ‘test everything’, what do you test? Explain why.
- 24** What is a ‘testing table’? What does it record?

## Evaluating solutions

- 25** When collecting data to measure the efficiency of a solution, what are the three measurable quantities the data should provide information about?
- 26** Explain how measuring the cost of data and file manipulation assists in evaluating a solution’s efficiency.
- 27** Suggest one method of data collection that could help demonstrate the efficiency of a solution in terms of speed of processing.
- 28** State two elements of a solution that could help to support the effectiveness factor of accessibility.
- 29** State two elements of a solution that could be checked when considering the effectiveness factor of readability.

## Evaluating project plans

- 30** Why are developers strongly advised to keep an accurate daily log journal?
- 31** What are some of the reasons for delays to a project plan?
- 32** How does a critical path assist with modifying a project plan? Could you have arranged your plan to make better use of the available time?
- 33** What advice would you give to someone who is thinking of starting their Unit 3 Gantt chart?



## APPLY YOUR KNOWLEDGE

- 1** How would you apply validation to the following user input?
  - a** Country of birth
  - b** Age
  - c** Agreement to terms and conditions
  - d** Email address
  - e** Victorian residence address (give two methods)
  - f** Date of birth (give two methods)
- 2** Write a pseudocode algorithm for a new member to enter a username and a new password. Include verification of the password, without displaying the characters. Provide a 'Re-enter password' message if the passwords do not match, and an acknowledgement message if there is a match.
- 3** How would you adjust any sort algorithm to sort from largest to smallest?
- 4** Write the following as pseudocode, then select and apply suitable test data for the following data entry:
  - a** a person is an adult if their age is 18 years or older
  - b** data entry of daily maximum and minimum temperatures in Victoria
  - c** ages of clients for a local dental surgery
  - d** car registration number plates
  - e** a system accepts values between 1 and 20 entered as input.

## PREPARING FOR

# Unit

# 4

### OUTCOME 1

On completion of this unit, the student should be able to develop and evaluate a software solution that meets requirements and assess the effectiveness of the project plan.

This outcome (U4O1) will be assessed according to the school-based assessment advice issued annually by the VCAA. All details should be verified each year prior to beginning your submission. Your teacher will provide specific details of the assessment rubric that will be used in that year.

The methods for the alpha and beta testing, evaluation of the efficiency and effectiveness of the software solution and the project plan evaluation have been developed throughout Chapter 5. Long reports must follow a sequence and have internal consistency. The examples and suggestions in Chapter 5 provide some guidance on how to accomplish the PSM development and evaluation stages to complete Unit 4, Outcome 1.

There are three components assessed in Unit 4, Outcome 1. Teachers will provide you with a schedule for submission of these assessments.

The following completed assessment tasks are to be submitted for the SAT:

- 1 A software solution that meets the software requirements specification
- 2 Evidence of preparation and conduction of beta testing
- 3 An evaluation of the efficiency and effectiveness of the software solution
- 4 An assessment of the effectiveness of the project plan (Gantt chart) in monitoring project progress.

The third and fourth tasks will be submitted as a written report or an annotated visual plan.

## 1. Submission: Software solution

(Final delivery of your finished software will be negotiated with your teacher.)

Your finished 'product' should be made available for assessment as a compressed (zipped) folder containing a directory of all files required for the software to function with a plain ReadMe.txt file, with instructions for installation and operation.

Also included in the ReadMe.txt file would be any passwords to allow entry past security.

Ensure your software solution meets the SRS criteria before submission. This is achieved by continual verification during the PSM development stage.

## 2. Document: Beta testing

This report will contain a comprehensive range of test data that provides:

- testing plan and test scenarios
- evidence of observation(s) of testing scenarios
- documentation of test results.

---

### 3. Document: Evaluation of the software solution

Evaluate the quality of the software solution using criteria from the SRS and Design folio developed for Unit 3.

- Evaluate efficiency and effectiveness of the solution based on the evaluation criteria created in Unit 3.
  - Evaluate efficiency and effectiveness of how the software solution meets functional and non-functional requirements.
- 

### 4. Document: Project plan evaluation

Evaluate the effectiveness of the project plan and the PSM in developing the software solution. The annotated visual plan or written report will include:

- the annotated project plan with notes and comments explaining which tasks needed to be modified and what the changes were
- your change log, which corresponds with the changes on your project plan
- the assessment and analysis of the effectiveness of the project plan.

When completing your project plan evaluation written report or annotated visual plan, consider using headings and dot points where possible. This is not an essay. Refer to the latest VCAA assessment criteria for further information.

## KEY KNOWLEDGE

After completing this chapter, you will be able to demonstrate knowledge of:

- KK4.2.3: types of vulnerabilities and risks within insecure development environments, including:
  - » use of application programming interfaces (APIs)
  - » malware
  - » unpatched software
  - » poor identity and access management practices
  - » man-in-the-middle attacks
  - » insider threats
  - » cyber security incidents
  - » risks present from software acquired by third parties
  - » ineffective code review practices
  - » combined development, testing and production environments
- KK4.2.4: security controls used to protect software development practices and data stored within applications, including:
  - » version control and code repositories
  - » robust identity and access management
  - » encryption
  - » code review
  - » regular updates and patches to software
  - » separated development, testing and production environments
- KK4.2.5: threat modelling principles, including:
  - » defining security requirements
  - » identifying and mitigating threats
  - » confirming threats have been mitigated.

## FOR THE STUDENT

In this chapter, you will focus on the security risks to software and data during the software development process. This chapter introduces the theory and skills necessary for completing Unit 4, Outcome 2. You will also consider the ongoing security risks for the use of the software solution by an organisation. During this chapter, you will analyse and evaluate current software development security practice, including the risks to software and data and the consequences of deploying software with ineffective security strategies. You will also investigate types of security vulnerabilities, including software controls, application programming interface (API) risks and third-party software risks.

This chapter begins with a consideration of cyber security risks during the software development process and deployment of the software solution software and data. You will analyse and evaluate a range of threats, including physical and software controls to protect software and data including version control, user authentication, encryption and software updates, software auditing, types of software security and data breaches, types of web application risks and, finally, managing risks offered by software through a third party.

By the end of this chapter, you should be able to identify and discuss potential risks to software and data security, and propose and apply criteria to evaluate the effectiveness of security practices for an organisation.

The theory and skills for Unit 4, Outcome 2 will be applied to a teacher-provided case study. The assessment task will be determined as either structured questions, a written report or a multimedia report.



## Security controls

Software development may take months or years before a product is released. The product is then maintained through various updates and eventually retired. Over that entire period of time, the product is expected to withstand many potentially hostile acts and still preserve the integrity, confidentiality and availability of its underlying functions, database and customer/consumer information. The process of securing the integrity of the software begins in the design and development stages.

Often vulnerabilities are caused unintentionally. Software developers insert sections of code as a temporary measure, with every intention of returning to fix this short-term measure – for example, an insecure database with live data taken from an active data set for testing, or a **back door** that avoids login security to allow quick access to modify settings. Too frequently, the motivation behind that decision to use insecure methods was to meet a deadline. Project management and version control software has removed the opportunity for insecure code that is not ready or does not meet production requirements. The new strategy does not allow the creation of a potential weakness or vulnerability. Instead, only compliant code is permitted to be added to the project; any code that is not yet ready is simply not permitted to be added to the project. (See the case studies on Atlassian project tracking, GitHub and Apache Subversion on pages 240–2.)

### How to protect software and data

Data must be protected in line with legal requirements (see Chapter 7 for legislation implications) during the stages of development and after release of the product. Frequently, ‘real data’ may be used for testing purposes, when protections may not have been put in place. Software may be a ‘work in progress’, but it can be an attractive target for those interested in gaining commercial advantage, early release for bootleg copies, or with other motivations.

Software security in this instance may be more vulnerable and more important than after the release of the gold master final version. There may be large numbers of employees with proximity and access to the master file storage. The security protocols outlined in the security plan will specify when and who has access, and this needs to be consistently applied during the development and release stages.

### Version control and code repositories

As all software developers quickly discover, keeping track of changes and the working up-to-date ‘master’ file is a challenging task. Frequently, soon after changes are made, a reversion to the previous unmodified file is necessary. Should the developer choose to keep one master file, all changes will not be recorded. A log of all changes is necessary. This **version control** records each stage of the development so that it is possible to ‘roll back’ to any previous point in the development.

When teams of developers work on several streams of the software, versions and stages become important, and the issue of merging the improvements becomes a new challenge. Software control of versions has been developed with ‘Subversion’, a popular open source option. Atlassian provides a commercial ecosystem for software development, and is free for teams of less than five. Version control may also be integrated into the programming language IDE.

#### CASE STUDY

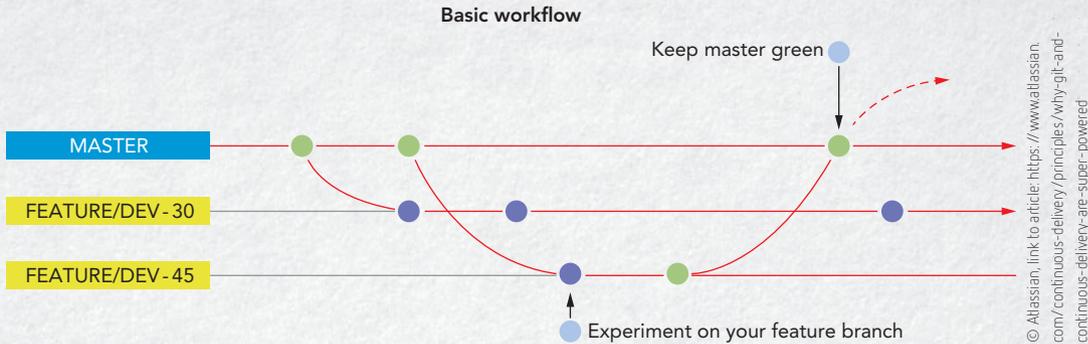


#### Atlassian

Creating a branch for each issue makes it easy to hand-pick which changes to ship out to production or bundle into a release. Since you are not dog-piling changes onto master, you get to select what comes into master – and when. You can ship an epic’s MVP plus one

nice-to-have, rather than wait until all the nice-to-haves are fully baked. Or ship a single bug fix and do it within the framework of a regular ol' release. Even if the fix is urgent, you won't have to deal with the three-ring circus of backing out other changes that aren't ready to ship yet just to get that one change out the door.

And that ease of shipping a single code change is the essence of continuous integration/continuous delivery (CI/CD).



**FIGURE 6.1** A branch-per-issue workflow developed by Atlassian

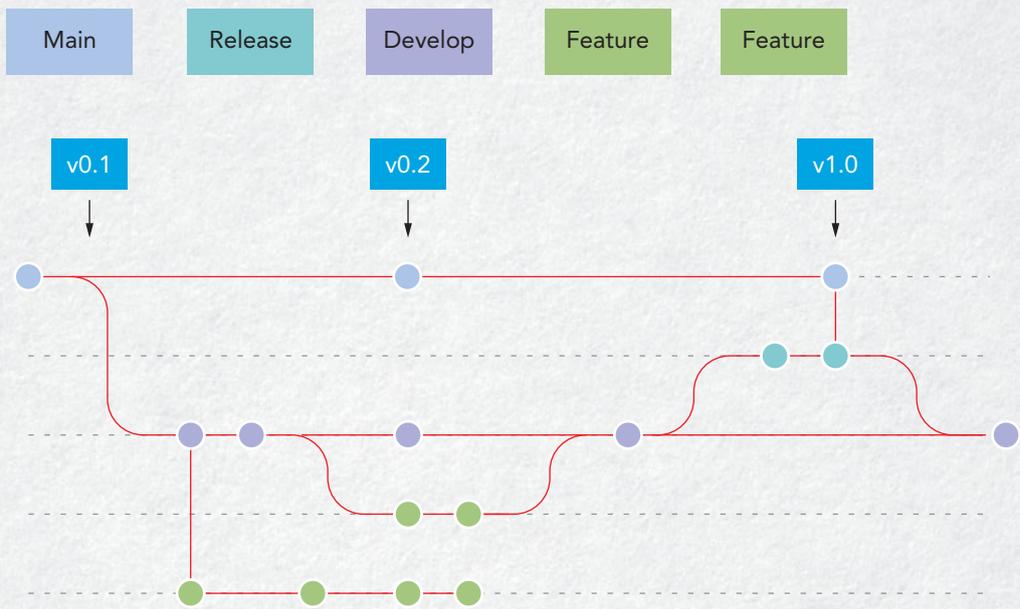
Text extract from 'Super-powered continuous delivery with Git', by Sarah Goff-Dupont, accessed from <https://es.atlassian.com/continuous-delivery/principles/why-git-and-continuous-delivery-are-super-powered>

### GitHub and Bitbucket both use Git

Git provides access to a repository ('repo') and version control system (VCS) located on your computer.

Online or distributed version control have web interfaces to manage the development of software by simple commands, which include clone, pull, put and merge.

Clone creates a developer version in a separate repository. Changes can be made and tested and merged back to the main branch where consistency checks are made. Authorisation and permissions are strictly controlled by assigning roles: Read, Write, Maintain and Admin.



**FIGURE 6.2** GitHub file development

**CASE STUDY**

**GitHub Copilot**  
AI is already reducing tedious, repetitive tasks and is expected to provide augmented thinking for software development. Reliable problem solving and clear workflows remain a challenge.

SCHOOL-ASSESSED TASK TRACKER

- Design brief
- Project plan
- Analysis
- Folio of alternative design ideas
- Development
- Evaluation and assessment
- Final submission

## CASE STUDY



## Apache Subversion

Apache Subversion (SVN) is a system of software versioning and revision control. Subversion was created by CollabNet Inc. in 2000, and is now a top-level Apache project being built and used by a global community of contributors.

Subversion uses branches and tagging to map versions. A branch represents a separate line of development. Tagging refers to labelling the repository at a certain point in time so that it can be easily found in the future. In Subversion, the only difference between branches and tags is how they are used.

A new branch or tag is set up by using the 'svn copy' command, which should be used in place of the native operating system mechanism. The copied directory is linked to the original in the repository to preserve its history, and the copy takes very little extra space in the repository.

All the versions in each branch maintain the history of the file up to the point of the copy, plus any changes made since. One can 'merge' changes back into the trunk or between branches.

Subversion is distributed as open source under the Apache License. Subversion is used by software developers to maintain current and historical versions of files such as source code, web pages and documentation. Its goal is to be a mostly compatible successor to the widely used Concurrent Versions System (CVS). The open-source community has used Subversion widely in projects such as Apache Software Foundation, FreePascal, FreeBSD, GCC and SourceForge.

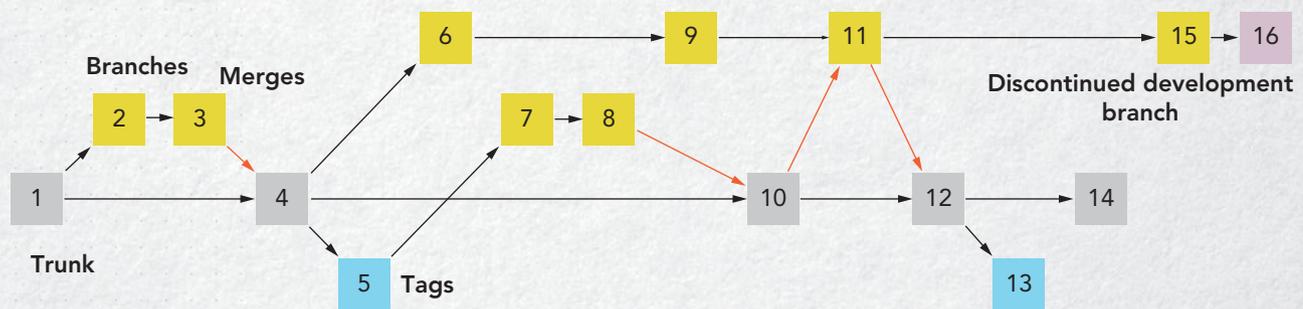


FIGURE 6.3 A project plan using Subversion

To prevent or lessen the impact of data spills and breaches and other cyber security incidents, the ACSC advises the implementation of the following steps.

- 1 Require all users to periodically reset passwords to reduce the ongoing risk of credential compromises.
- 2 Consider increasing password length and complexity requirements to mitigate the risk of brute-force attacks being successful.
- 3 Implement a lockout for multiple failed login attempts.
- 4 If credentials have been compromised, reset passwords as soon as possible.
- 5 Discourage users from reusing the same password across critical services such as banking and social media sites, or sharing passwords for a critical service with a non-critical service.
- 6 Recommend the use of passphrases that are not based on simple dictionary words or a combination of personal information. This reduces the risk of password guessing and simple brute-forcing.

- 7 Advise users to ensure new passwords do not follow a recognisable pattern. This reduces the risk of intelligent brute-forcing based on previously stolen credentials.
- 8 Use multi-factor authentication for all remote access to business systems and for all users when they perform a privileged action or access an important (sensitive/high-availability) data repository.
- 9 Look out for unusual account activity or suspicious logins. This may help detect when a service such as email has been compromised and needs a password reset.
- 10 Encourage users to think carefully before entering credentials.
  - » Ask if this is normal.
  - » Don't enter credentials into a form loaded from a link sent in email, chat or other means open to receiving communications from an unknown party.
  - » Even if the page looks like the service being reset, think twice.
  - » Do not click the link. Instead, browse to the website and reset the password from there.
  - » Be aware that friends' or other contacts' accounts could be compromised and controlled by a third party to also send a link.
- 11 If some credentials have been compromised, try to identify a specific cause. Were the credentials entered in an untrusted place? Were they recently reset? What were the credentials for? Were the credentials used elsewhere?
- 12 Keep operating systems, browsers and plugins up-to-date with patches and fixes.
- 13 Enable anti-virus protections to help guard against malware that steals credentials.

© Office of the Australian Information Commissioner – [www.oaic.gov.au](http://www.oaic.gov.au). Extract from 'Information from the Australian Cyber Security Centre about preventing and mitigating data breaches'. Released under CC BY 3.0 AU, link to license: <https://creativecommons.org/licenses/by/3.0/au/deed.en>

## Robust identity and access management

The Australian Cyber Security Centre (ACSC) recommends using certain techniques to prevent cyber security threats. These techniques include clearly documenting and training employees in cyber security systems and plans, and designing and implementing cyber security awareness programs for all employees.

Robust identity and access management (IAM) improves security by ensuring that only authorised individuals have access to resources and systems, reducing the risk of unauthorised access and potential security breaches.

## Encryption

The encryption strategy recognises that data will be exposed to unauthorised access, so the data is rendered useless by encoding it to make it unintelligible to all but those properly authorised and accredited. There are several forms of encryption readily available. Pretty Good Privacy (PGP) has been available since 1991.

## Code review

Code review acts as a security control by systematically examining source code to identify and eliminate vulnerabilities before deployment. It ensures adherence to security standards and best practices and reduces the risk of security flaws reaching production. By involving multiple reviewers, code reviews leverage diverse expertise to catch subtle issues, fostering a culture of security awareness among developers and enhancing the software's overall security.

The Australian Cyber Security Centre (ACSC) is an organisation within the Australian Signals Directorate (ASD) responsible for improving Australia's cyber security posture. The ACSC provides advice and information on cyber security threats and vulnerabilities to government, businesses and individuals, and coordinates responses to cyber incidents across the nation. Its mission is to enhance Australia's defence against cyber threats and to foster a secure and resilient digital environment.

SCHOOL-ASSESSED TASK TRACKER

Design brief
  Project plan
  Analysis
  Folio of alternative design ideas
  Development
  Evaluation and assessment
  Final submission



**FIGURE 6.4** ‘Patch Tuesday’ (every second Tuesday of every month) is when Microsoft releases updates and patches.

Source: Microsoft Corporation

## Regular updates and patches to software

### Software updates and patches

Vulnerabilities to existing systems are constantly identified and a fix is created by the distributors of the software. These fixed versions are released by software companies as a ‘patch’ to be inserted in the operational software wherever it may be installed. The obligation for installing and updating the vulnerable software is entirely on the user or operator of the software. The software company has a responsibility to make the patch available, but it is up to the software operator to update and ensure that the software and system are as secure as they can be. (See the WannaCry case study below. WannaCry **exploited** a well-known vulnerability on systems that had not been updated.)

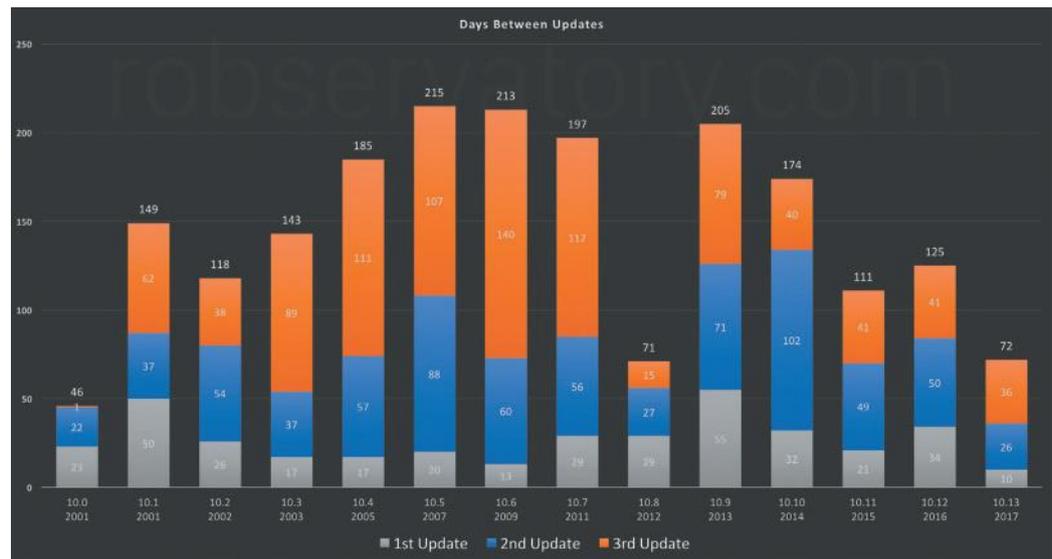
There are severe implications for operating with compromised systems. For example, compensation is likely to be due to affected users of the system if all reasonable steps have not been taken to secure users’ personal data and information.

Software updates are a regular step in the maintenance of deployed software products. Operating systems are frequently updated according to a published timetable or as the need arises.



**Weblink**  
The Robservatory history of macOS release dates and rates

**FIGURE 6.5** macOS updates for OSX. There is a major release each year; dot updates have been irregular or as the need demands. View the most current update data on the Robservatory website.



Robservatory



### CASE STUDY

## WannaCry is a ransomware cryptoworm

In 2017, the WannaCry cryptoworm, also known as WannaCrypt, targeted Microsoft Windows operating systems by encrypting data and demanding ransom payments in Bitcoin cryptocurrency.

The worm automatically spread itself by using a vulnerability that was first identified by the US National Security Agency but was withheld to allow the exploit to be used for NSA purposes. WannaCry has been estimated to affect 230 000 computers in 150 countries over a period of two weeks.

Ninety-eight per cent of the affected systems were running Windows 7 and 8, with the remainder running Windows XP. Microsoft had released a patch several months earlier that would have prevented the attack, but only if Windows updates were enabled. The affected computers had not received an update since the last patch issued by Microsoft in 2014.

To avoid such exploit attacks:

- 1 Maintain the operating system by installing all security updates immediately. Microsoft issued a patch in March; WannaCry did the damage in May.
- 2 Only run supported operating systems. Both Windows XP and Windows 7 are no longer receiving updates.
- 3 Software developers rely upon notification of vulnerabilities as and when they are found. The vulnerability was known, but had been stockpiled rather than disclosed.



**FIGURE 6.6** Impact of the WannaCry cryptoworm attack

## Separated environments

Separated development, testing and production environments act as a security control by isolating different stages of the PSM to prevent untested or insecure code from affecting the live (production) environment. This separation ensures that development code and testing activities do not have access to sensitive production data, reducing the risk of data breaches and unauthorised access.

Separated environments also help enforce stricter access controls and audit trails, making it easier to track changes and identify the source of issues. By replicating production-like conditions in testing environments without exposing actual production data, this practice allows thorough testing and validation of code, thereby minimising the likelihood of vulnerabilities reaching the production environment.

## Vulnerabilities and risks

Security vulnerabilities and risks go hand in hand with software development, and it is important that these are considered within any development environment. Ensuring security helps comply with regulations, avoiding legal penalties and fines. Preventing security issues early also helps to reduce potential financial losses from security breaches and avoid costly fixes after the implementation stage of the PSM.

The types of vulnerabilities and risks within development environments are outlined below. Addressing these vulnerabilities and risks during development encourages a security-first culture, enhancing user experience and ensuring long-term sustainability.

SCHOOL-ASSESSED TASK TRACKER

Design brief

Project plan

Analysis

Folio of alternative design ideas

Development

Evaluation and assessment

Final submission

## Use of application programming interfaces (APIs)

The use of application programming interfaces (APIs) can create vulnerabilities and risks in software development by exposing endpoints that can be exploited if not properly secured. APIs often handle sensitive data and critical functions, making them attractive targets for attackers. Common vulnerabilities include insufficient authentication and authorisation, lack of input validation and improper error handling. These issues can lead to data breaches, unauthorised access and other security incidents if not addressed during development and deployment.



### RESEARCH

#### Facebook-Cambridge Analytica

A good demonstration of API vulnerabilities is the Facebook-Cambridge Analytica scandal. In this case, a third-party application exploited Facebook's API to collect personal data from millions of users without their explicit consent. The API allowed extensive data access, and the lack of proper controls and oversight led to a significant data privacy breach, highlighting the importance of securing APIs to protect user data and maintain trust.

Read more about the Facebook-Cambridge Analytica scandal via the weblink.



Weblink  
Facebook-Cambridge  
Analytica data scandal

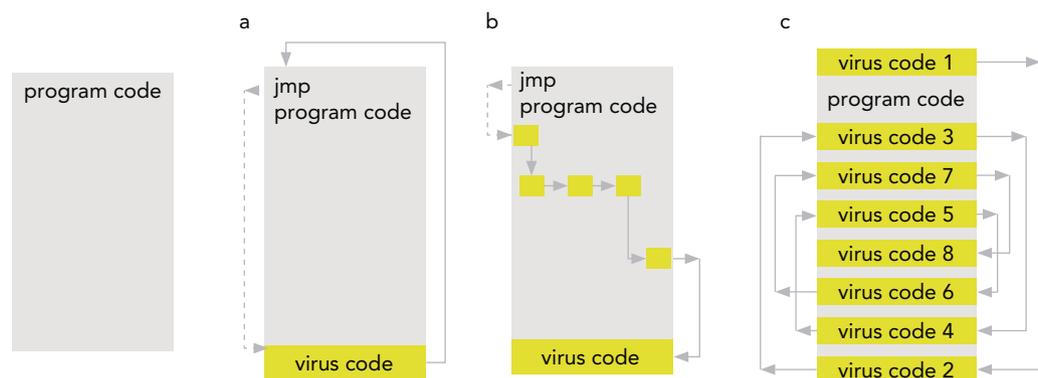
## Malware

A variety of suspicious code or **malware** (malicious software) exists that can take advantage of weaknesses in the host computer software or operating system. The main objective of viruses and worms is to spread infection, while others aim to remain concealed and undetected. All, however, enter the computer system without the user's knowledge and perform unwanted and sometimes harmful actions.

### Viruses

A computer **virus** is malicious code that reproduces itself on the same computer. The code inserts itself into files and modifies, corrupts or destroys the ability of the file to operate as normal. There are many ways the virus can infect the data file or program, including:

- appender infection
- Swiss cheese infection
- split infection.



**FIGURE 6.7** Types of virus infection that can overwhelm a file collection on local storage media:

- a** Appender virus;
- b** Swiss cheese virus;
- c** Split infection virus

## Worms

A **worm** is a self-replicating malicious code that can spread across computer systems and can modify, corrupt or delete data or information. Other programs can also be run undetected by the legitimate user and operated remotely as ‘zombie bots’.

## Trojan horses

This code is named after the story from Greek mythology where the Trojan Horse appears to be one thing, but is actually another. **Trojan horses** often take the form of ‘free virus-checking software’, when in fact they are malware and can be virus propagators. Permission must be granted to allow the software to be installed on the computer. The user must be convinced to enter the appropriate password to activate installation.

This attack is difficult to counter, as the weakness is the human user who has administrative permission to install software. If that person installs the software, the computer can be infected. Preventative measures are mostly defensive, such as maintaining full incremental backups of system and user files and applying anti-virus software that scans for known characteristics. The disadvantage with both of these strategies is that the newer malware will not be detected and the computer hard drive will need to be wiped and a full system reinstalled.

## Logic bombs

Malicious software will lie dormant until a timer or conditional trigger goes off, which will activate at a predetermined time or when certain circumstances are satisfied. Logic bombs are placed by programmers or developers who have access to the source code.

Ransomware attacks (2018 to 2024)



Source: Comparitech Limited

**FIGURE 6.8** Ransomware attacks are increasing.

### THINK ABOUT SOFTWARE DEVELOPMENT

6.1

Explore the latest Scamwatch statistics at the ACCC website.

- 1 What are the limitations on the information displayed?
- 2 If these statistics are an estimate, will the actual numbers be higher or lower than those displayed? Explain your reasoning.
- 3 For the current year, which category has:
  - a the most reported scam?
  - b the greatest amount lost?
  - c the most frequent delivery method?
  - d the most affected age group?
- 4 Why is the connection between ‘Amount lost’ and ‘Number of reports’ inconsistent?



Weblinks  
Scamwatch  
Comparitech

### SCHOOL-ASSESSED TASK TRACKER

- |  |  |  |   |   |   |  |
|--|--|--|---|---|---|--|
| <input checked="" type="checkbox"/> Design brief | <input checked="" type="checkbox"/> Project plan | <input checked="" type="checkbox"/> Analysis | <input checked="" type="checkbox"/> Folio of alternative design ideas | <input checked="" type="checkbox"/> Development | <input checked="" type="checkbox"/> Evaluation and assessment | <input checked="" type="checkbox"/> Final submission |
|--|--|--|---|---|---|--|



## ISSUE

### '... there was a 37% increase in ransomware attacks in 2023' – Jay Chaudhry

The reason is that cyberattacks are getting more sophisticated and firewalls and VPNs were built for a time when most people were working within the confines of an office. Today, with the cloud, mobility and the rise of hybrid work, an organization's attack surface has exponentially expanded. In other words, the landscape has fundamentally changed, and so the ways to secure it must change as well.

'Don't break the bank stopping ransomware from getting the best of your business', by Jay Chaudhry on Forbes website, 1 February 2024. Source: <https://www.forbes.com/sites/forbestechcouncil/2024/02/01/dont-break-the-bank-stopping-ransomware-from-getting-the-best-of-your-business/?sh=49e2441954f3>

WannaCry ransomware was released in May 2017. Within two weeks, an estimated 230 000 computers in 150 countries were infected. Security experts released decryption tools quickly to limit the damage.



WebLink

Wannacry ransomware impact animation

## Unpatched software

Unpatched software can create vulnerabilities and risks in developing software by leaving known security flaws unaddressed, which attackers can exploit. When software vendors discover vulnerabilities, they release patches to fix these issues. However, if these patches are not applied quickly, the software remains susceptible to attacks that target these known weaknesses. This negligence can lead to unauthorised access, data breaches and other malicious activities, compromising the security and integrity of the entire system.

A real-world example of the risks posed by unpatched software is the WannaCry ransomware attack in 2017, as described on page 244. This attack exploited a vulnerability in Microsoft Windows that had been previously identified and patched by Microsoft. However, many organisations had not applied the patch, leaving their systems vulnerable. WannaCry spread rapidly, encrypting data on affected computers and demanding ransom payments. The attack caused widespread disruption, affecting critical services like health care and transportation, and underscored the importance of timely patching to mitigate security risks.

## Poor identity and access management practices

Poor identity and access management (IAM) practices can create vulnerabilities and risks in software development by allowing unauthorised users to access sensitive data and critical systems. Weak IAM practices, such as inadequate password policies, lack of multi-factor authentication (MFA) and insufficient monitoring of access logs, can lead to security breaches.



## RESEARCH

### Twitter (X) hack

One of the risks associated with poor identity and access management practices is the 2020 X (formerly known as Twitter) hack. In this incident, attackers used social engineering techniques to target X employees, gaining access to internal tools and systems. Due to inadequate access controls and insufficient use of multi-factor authentication (MFA), the attackers were able to take over high-profile accounts, including those of prominent individuals and companies. They used these accounts to post a cryptocurrency scam, causing significant reputational damage and financial losses.

This breach underscores the importance of implementing strong IAM practices, such as enforcing the principle of least privilege, using MFA and conducting regular security training to prevent social engineering attacks.

To find out more, read the blog post, 'What the Twitter Hack Revealed About Social Engineering' on the Terranova Security website.

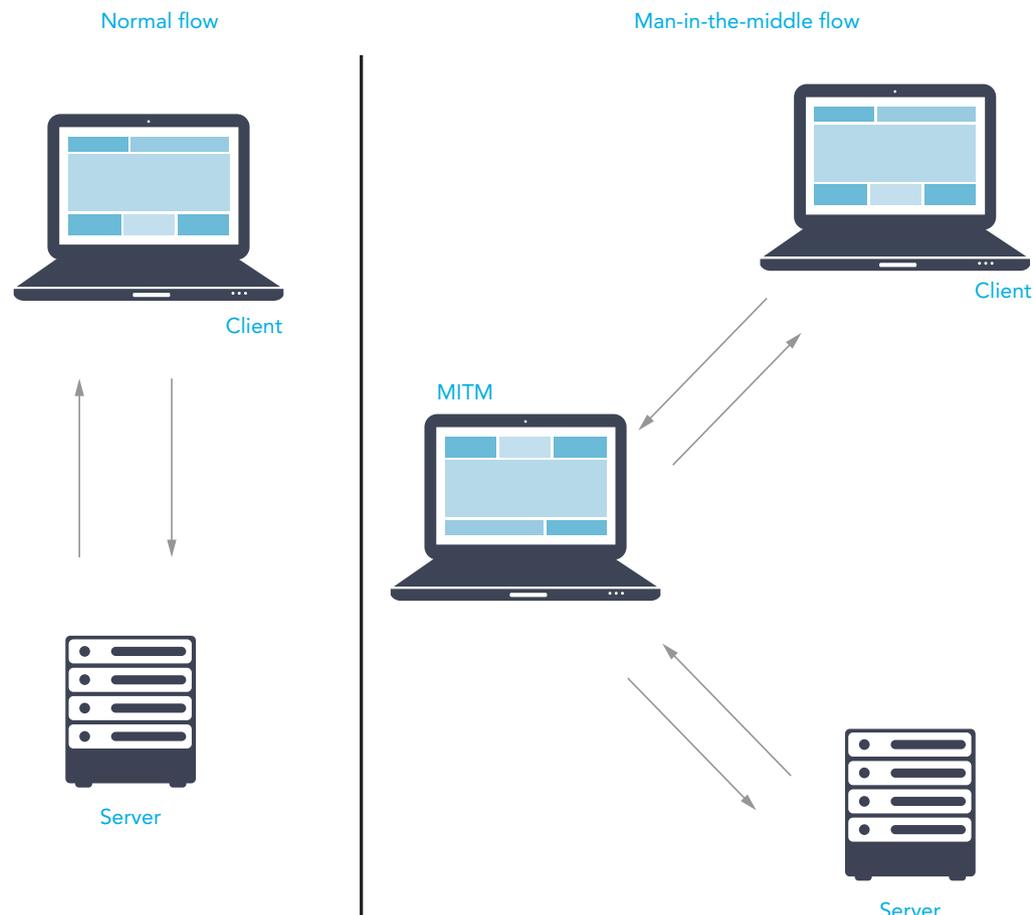


WebLink  
X hack

## Man-in-the-middle attacks

**Man-in-the-middle (MITM) attacks** are a type of eavesdropping attack that is difficult to detect. The threat is present whenever a transmission takes place and communications and data are exposed to an unauthorised third party. While unsecured Wi-Fi is a commonplace opportunity for MITM attacks to occur, many other situations also permit unauthorised interception of the transmission.

In its simplest form, an instance of an MITM attack requires a transmission to be redirected through a third computer system, which the attacker controls, then relayed to either side of a conversation. Often data and information are harvested for valuable or sensitive items, or modified or substituted – for example, bank account login details, company secrets or other confidential material.

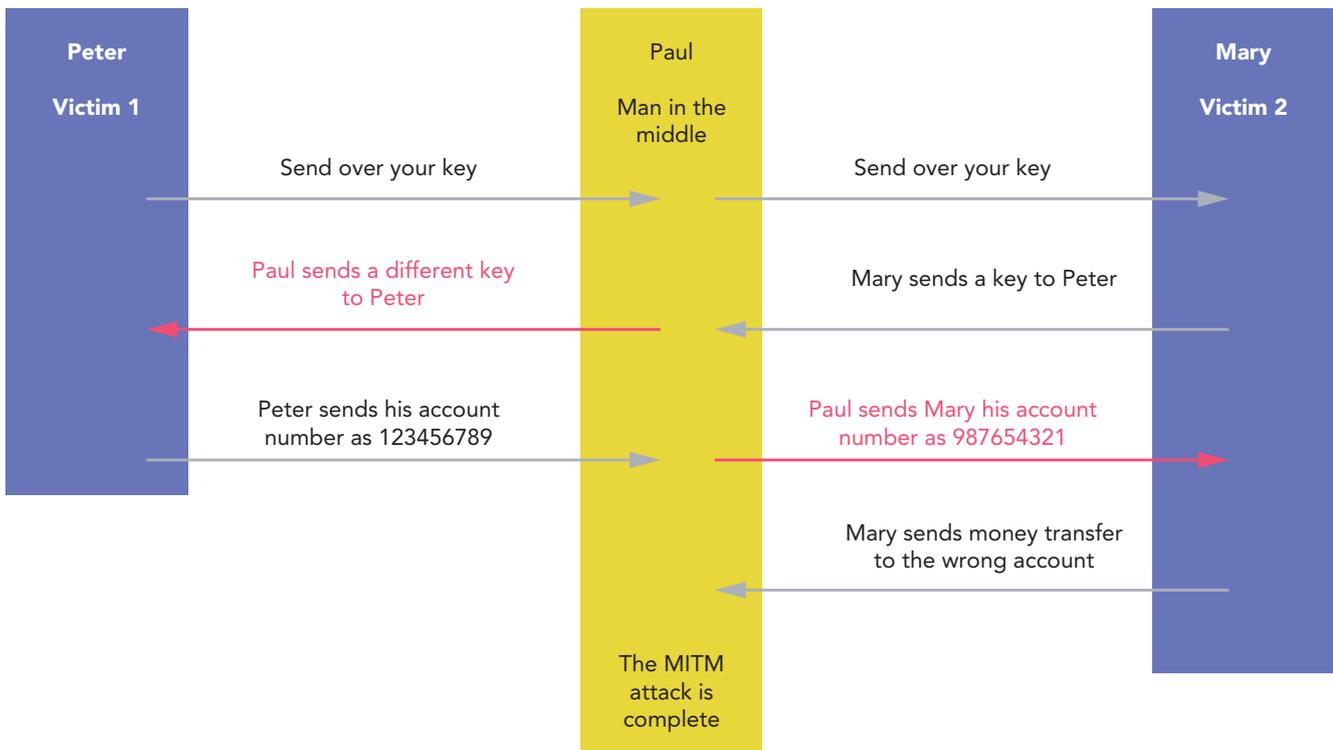


**FIGURE 6.9** Man-in-the-middle attacks are very difficult to detect.

### THINK ABOUT SOFTWARE DEVELOPMENT

6.2

Describe the steps of a simple MITM attack during an online banking transaction.



**FIGURE 6.10** A banking scam using MITM on unsecured Wi-Fi. Often free Wi-Fi in shopping centres and airports can be impersonated (faked) and users fooled (spoofed) into using the attacker's Wi-Fi server.

## Mitigating risk of MITM attacks

There are a number of strategies that can be put in place to mitigate the risk of man-in-the-middle attacks.

**TABLE 6.1** Strategies to mitigate the risk of MITM attacks

| Strategies                             | Description  |
|--|--|
| Strong encryption                      | Utilise TLS/SSL and end-to-end encryption to protect data while it is being transferred across networks.   |
| Server certificate verification        | Put in place security certificates and ensure that they are issued by trusted certificate authorities. Ensure that applications verify the authenticity of these certificates. |
| Use of secure networks                 | Avoid public Wi-Fi and use VPNs for secure, encrypted connections.   |
| Multi-factor authentication            | Add extra layers of authentication to prevent unauthorised access.   |
| Ensure up-to-date software and systems | Regularly update and patch software to fix security vulnerabilities.   |

## Insider threats

### Insiders

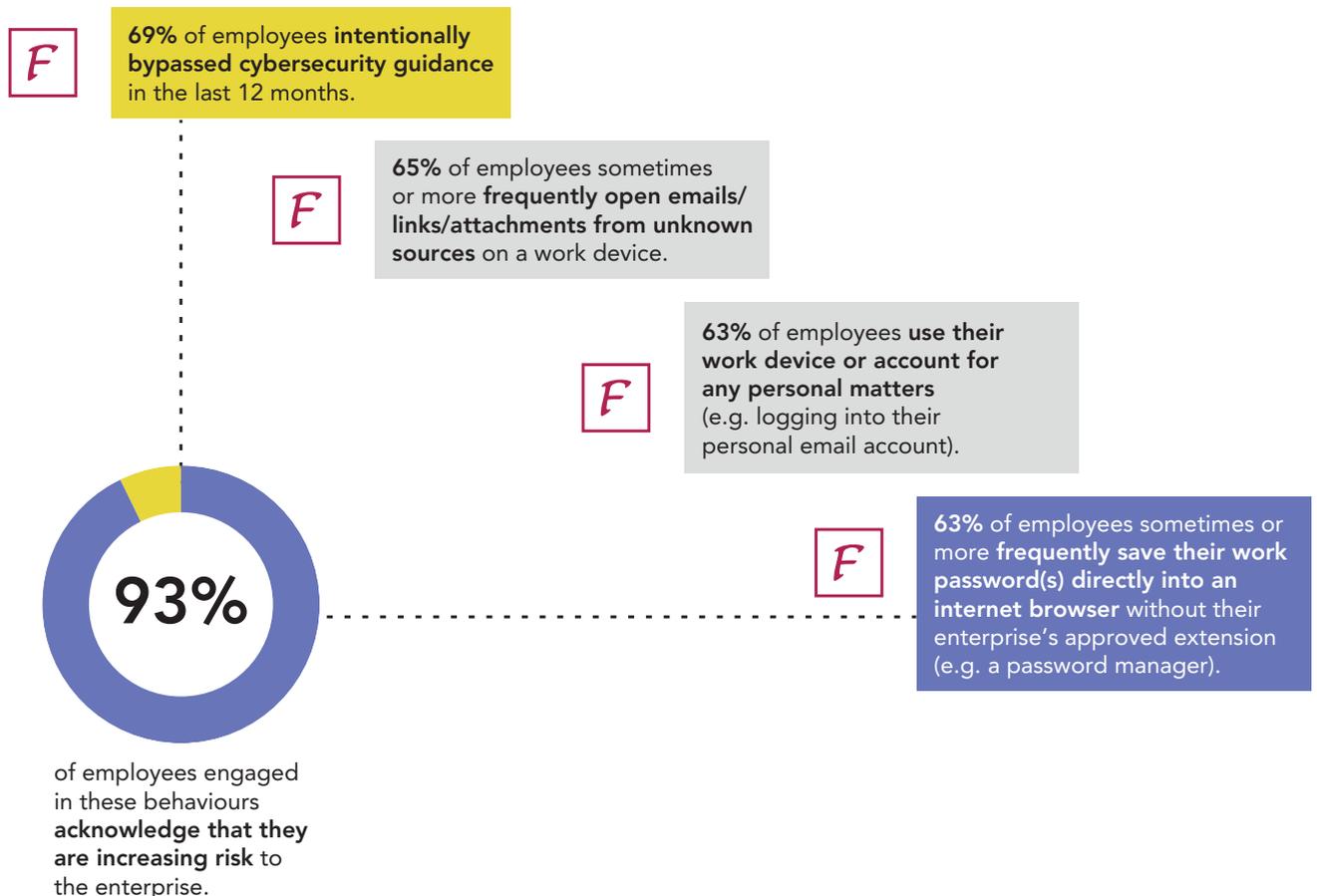
The most effective attack on an organisation comes from an unlikely source known as insiders (a person a business trusts, such as current employees, former employees, contractors, vendors or business partners). Insiders can abuse that privileged access to company information and

systems. Attacks range from unintended vulnerabilities to malicious sabotage. There are four types of insider threats:

- data breach
- fraud
- theft of intellectual property
- sabotage (to damage a system or destroy data).

Insider threats are the cause of most data breaches in organisations. There are also situations when insider attacks are accidental, such as human error, poor judgement, malware, phishing and convenience. Examples include an email attachment where sensitive data is inadvertently included; email addresses placed in the CC field instead of the BCC field; and blacked-out sections in a PDF that can be uncovered using software. These gaps in knowledge may be due to inadequate familiarity with process or technology. Therefore, it is essential that organisations offer appropriate training and support.

### Where Most Employees Fail: Secure Behaviour Scorecard



n = 1310 Employees  
Source: Data from 2022 Gartner Drivers of Secure Behavior Survey. Source: Mindflow

**FIGURE 6.11** Awareness is not the issue, human behaviour is.

Compromised accounts are not controlled by insiders; rather, others have gained control and allow actions to go undetected. Phishing or another tactic has stolen the credentials to take advantage of access.

Malicious insiders are a third type of insider threat that are intent on causing damage or making a personal gain. Predicting who will cause deliberate disruptions is very difficult; however, the scale of harm to the business can be catastrophic. Data and information critical to the operation of the business can be altered, destroyed or sold to competitors. This sort of exposure of company secrets can be very harmful to a business.

It is important to respond promptly and appropriately to insider threats. If all employees are considered malicious, the workforce will quickly choose to go elsewhere. Similarly, if everyone is trusted and there is no monitoring, the company could be irretrievably affected. The intention must be to identify the type of insider threat quickly and provide an appropriate response.

## CASE STUDY

### Internal Breach at Tesla Exposes Valuable Company Data



JOHN THYS/Getty Images

A former software engineer at Tesla, Alex Khatilov, exploited his two-week employment to steal 6,000 scripts or code files that automate various company functions. Despite the files being unrelated to his job, Khatilov had been granted access to them. Tesla discovered that Khatilov had transferred the sensitive information to his personal Dropbox, which could be accessed from his personal computer. Although investigators found thousands of confidential files in Khatilov's storage, which he attempted to destroy before the investigation, there is no evidence of whether he shared the data with others.

Tesla emphasised the immense value of these scripts, which the company uses to automate various business tasks including the ordering of parts and delivering vehicles. Unauthorised access to these scripts could provide competitors with a roadmap to copy

Tesla's innovation. Apart from the competitive disadvantage this could create, such incidents erode trust among customers and other stakeholders, posing reputational risks to the organisation.

The internal breach at Tesla serves as a stark reminder of the critical need for enhanced security measures and proactive approaches to data protection.

## Cyber security incidents

Cyber security incidents are a significant security vulnerability and risk because they exploit weaknesses in systems, networks and applications, leading to unauthorised access, data breaches, financial loss and reputational damage.

### Social engineering

In many ways, social engineering attacks are not about computing. **Social engineering attacks** rely on the manipulation of human nature to persuade the victim to provide personal information or to initiate a transaction. Once the attacker convinces the victim that they can be trusted, they gain the management and control needed to achieve the illegal activity.

The basic tricks of the 'con' artist are employed, using flattery and friendliness to gain trust. Another tactic is to create the illusion that 'everyone is doing it' in order to encourage the victim to join in.

Types of social engineering attacks include the following.

- **Phishing** steals personal information such as usernames, passwords, phone numbers, tax file numbers, driver's licence numbers, and credit card and banking details. The phishing methods deceive victims into providing personal details by using fake websites or urgent messages requiring a response, or malware that searches the computer and extracts documents and files from the documents directory.
- Spear phishing targets wealthy individuals or victims with valuable access or information.
- Pretexting fabricates a plausible scenario that fools a victim into revealing sensitive information or providing access to restricted systems.
- Baiting entices users to provide their password in exchange for free music or movie downloads. This frequently takes the form of a USB stick that installs malware or a keylogger once connected to a computer or network.
- **Pharming** redirects users to false websites that imitate the legitimate URL. Pharming can affect large numbers of users simultaneously by a 'poisoned' DNS server that redirects to the wrong website.
- Quid pro quo attacks offer a benefit in exchange for information. These are often in the form of IT service assistance that requires the installation of a 'fix', which is actually malware.
- **Tailgating** or **piggyback entry** allows unauthorised people into restricted areas. This may take place at security doors or by observing password entry at login screens.

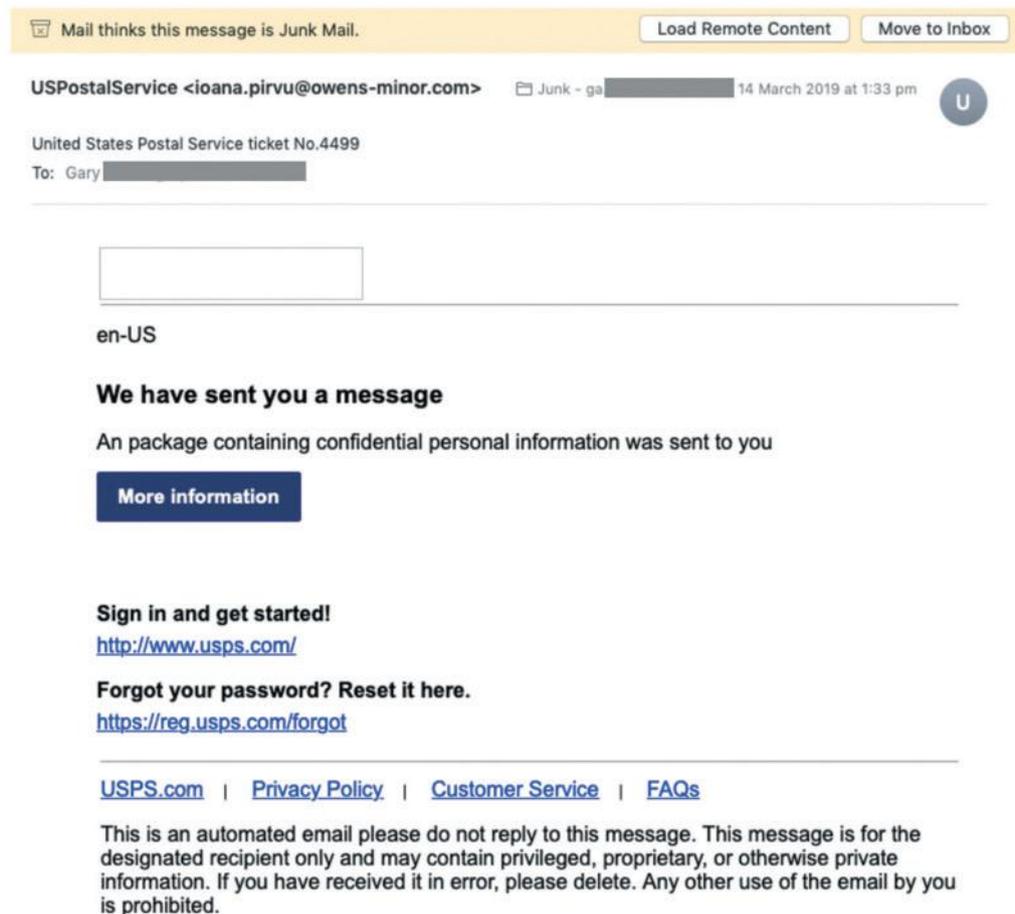
#### SCHOOL-ASSESSED TASK TRACKER

Design brief
  Project plan
  Analysis
  Folio of alternative design ideas
  Development
  Evaluation and assessment
  Final submission

### Is there protection against social engineering attacks?

Users must take every online request seriously and treat each one as a potential threat from someone trying to compromise their data. Most social engineering attacks rely on vulnerabilities in human behaviour and poor security habits. In an attempt to overcome the limitations of humans, certain procedures can be adopted.

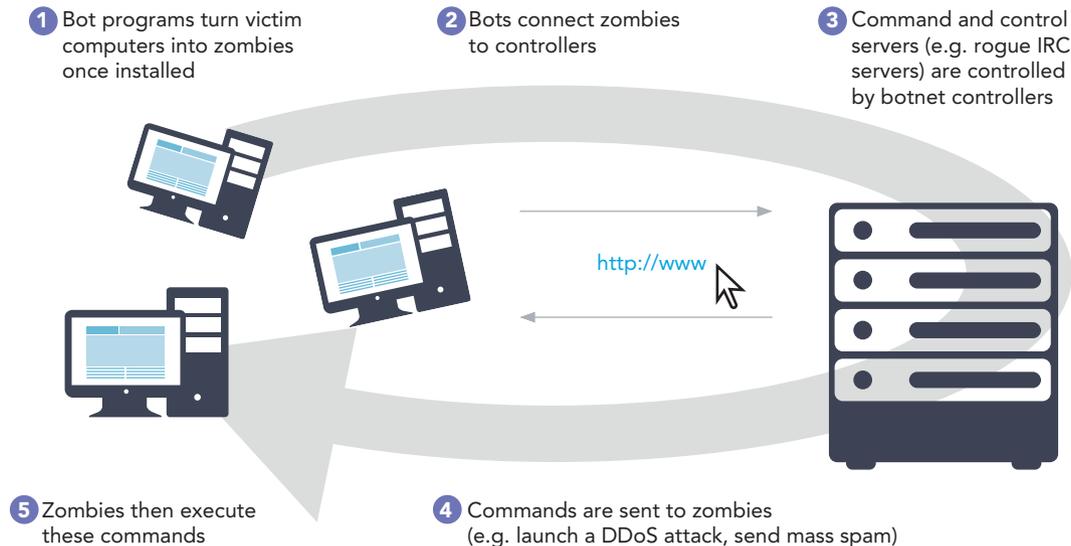
- Train all users so there is awareness of the risks and an ability to recognise an attack, and the indications of an attack.
- Make sure users know how to report an incident and who to report it to.
- Keep anti-virus software up to date, and activated.
- Use **spyware** detection software.
- Maintain operating systems and applications with updates and patches, after receiving the advice of system administrators.
- Do not open email (or SMS) attachments when the source is unknown.
- Avoid social engineering attacks by refusing to disclose information by phone or email.
- Only download, and install, authorised software from approved download sites.
- Use complex two-factor authentication and protect passwords and usernames.
- Employ zero trust methods that use biometric devices to authenticate and verify at every stage of movement inside the firewall.



**FIGURE 6.12** Phishing for your details. Spelling and grammar errors are the first indication of a scam. The unusual domain name in the email is also a warning.

## Distributed denial-of-service attacks

A **distributed denial-of-service (DDoS) attack**, or the lesser version, a denial-of-service (DoS) attack, involves an overwhelming number of access requests being put to a web server. The attacker sometimes makes use of **botnet zombies**, which have been commandeered unknowingly and are marshalled in a coordinated attack against a specified target. Zombies in a botnet are instructed to send IP requests to a targeted web server simultaneously. The server becomes overwhelmed by the huge number of access requests, and this slows all server operations or causes the server to stop altogether, or crash.



**FIGURE 6.13**  
Denial-of-service attacks are increasingly from cloud-based data centres.

DDoS attacks do not attempt to steal or corrupt data; the purpose is simply to overload web servers and cause them to be unstable. This loss of service can be very expensive to a company or institution. One possible solution for a single server is to change the IP address. It can take up to 72 hours to propagate a new IP. However, this may not stop the problem, as the attackers can simply redirect the attack on the new IP. Other options are to operate several servers with different IP addresses; to increase bandwidth to accommodate ‘spikes’ in traffic; and configuring firewall/router settings to reject certain DNS and ping-based attacks by blocking certain ports. Automatic active solutions monitor the number of incomplete connections and cancel them when a predetermined threshold has been reached. This will avoid the server being overwhelmed with connection requests.

## Bots and web scraping

Automated internet robots, or netbots, are responsible for approximately 47 per cent of all web traffic. While there are good and bad bots, harmful bots are responsible for 30 per cent of web traffic. These bad bots include spambots and unauthorised data scrapers. **Spam** generated automatically accounts for over 80 per cent of all email messages circulating each day.

Good bots are essential for the orderly conduct of many web functions. For example, Google and Facebook operate many millions of web crawlers or data-extracting spiders that check the connections and update details every day. Feed fetchers are helper bots that update a Facebook feed on a mobile app and cause about 5 per cent of all web traffic. Retail and entertainment bots gather data to compare offerings and aggregate listings for comparison websites.

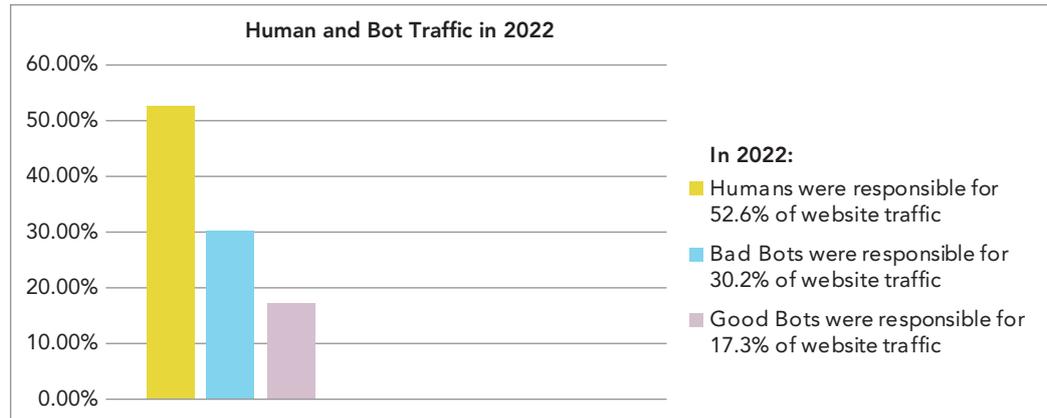
There are examples where thousands of bots every day send millions of unauthorised spam emails to make up over half of all internet traffic:

- 54 billion spam messages per day
- 156 million phishing emails per day.

The most common spam email is healthcare or dating spam.

### SCHOOL-ASSESSED TASK TRACKER

- |  |  |  |   |   |   |  |
|--|--|--|---|---|---|--|
| <input checked="" type="checkbox"/> Design brief | <input checked="" type="checkbox"/> Project plan | <input checked="" type="checkbox"/> Analysis | <input checked="" type="checkbox"/> Folio of alternative design ideas | <input checked="" type="checkbox"/> Development | <input checked="" type="checkbox"/> Evaluation and assessment | <input checked="" type="checkbox"/> Final submission |
|--|--|--|---|---|---|--|



**FIGURE 6.14** Bot traffic in 2022



Weblink  
Imperva Bad Bot  
Report

Impersonator bots, also known as zombies, are controlled by attackers to perpetrate DDoS attacks against third parties. An estimated 25 per cent of web traffic is used in this way. The bot may operate without the knowledge of its owner host. It may be created by malicious code, which creates undetected activities to be carried out by the captured computer. The only way to detect this presence is to monitor computer activity for unexpected online traffic or CPU loads.

### Denial of inventory

Bots hold items in shopping carts, which prevent access to items for 'real' customers. The shopping cart is later abandoned as the 'hold' times out. Retailers notice increases in the number of abandoned shopping carts from certain 'zombie' accounts with no purchases.

### Gift card balance checking

Bots steal money from gift card accounts with a remaining balance. This is a variation on brute-force or dictionary attacks, and can be detected by spikes in login requests to gift card balances, and resultant complaints from customers who have 'lost' the balance of their account.

### Price scraping

Web scraping is the act of taking data and information without authority or permission. Often a website display relies on the application of copyright laws to protect ownership access and use of the information and data. Data scraping aggregates enormous volumes of data and applies the knowledge gained for commercial advantage. For example, price scraping of competitors' websites ensures that up-to-date knowledge can be used to undercut competitors, and the perpetrator can emerge with the lowest prices in the marketplace. Airlines, gambling odds, real estate, used-car prices, petrol and other supply retailers are all business categories where price scraping occurs every minute of every day.

### Content scraping

Content scraping is when web page content is stolen and reused without permission. Although copyright laws apply, the fact that the internet is international means it is unlikely that any action will be taken. Another aspect of content scraping is to save on the cost of gathering data

#### THINK ABOUT SOFTWARE DEVELOPMENT

6.3

Research the latest Imperva Bad Bot Report via the weblink.

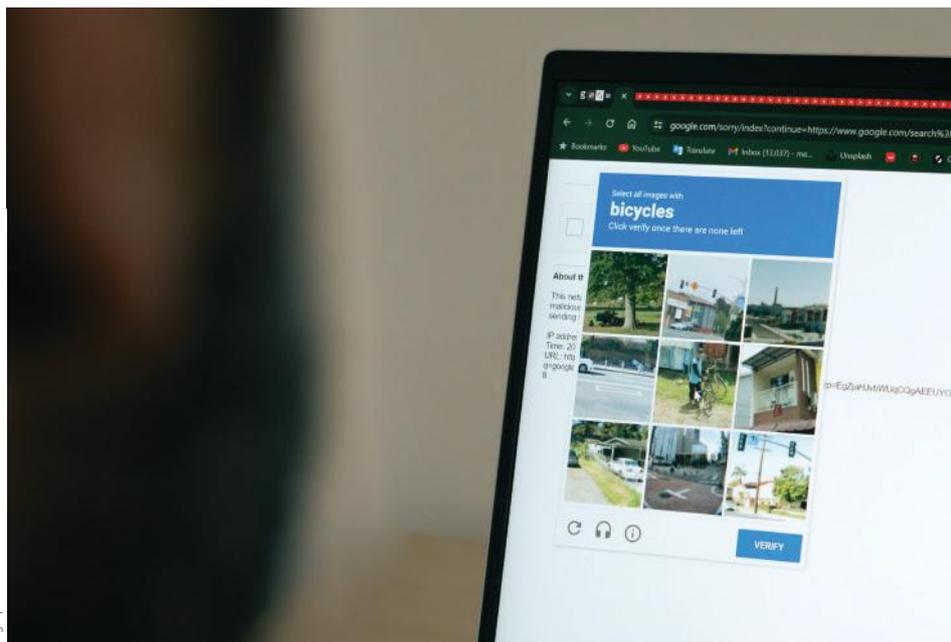
- 1 Identify the most recent numbers of Humans vs Good bots vs Bad bots.
- 2 Have any categories of attack changed over time?
- 3 Has the source of bot attacks changed over time?

and information and to take advantage of the infrastructure of another business. Examples are news gathering, textbook and subject reference websites, sporting results websites and stock market 'ticker' websites where summary reports of activity are available for a fee. Paywalls and login requirements offer little defence against aggressive web scraping. The web bots can mimic human behaviour, create login accounts and extract large amounts of content. Previously, displaying the web page as an image was an effective defence against bots that would record text; however, AI and OCR converting text within an image have made this a trivial conversion task.

An example of a web scraper application is when the application can record a specific stock price every five minutes and the volumes traded without the need to subscribe to a stock market monitoring service. Similar monitoring of airline prices for ticketing can be automated. Images, email addresses, phone contacts and addresses can all be harvested from websites automatically.

### Strategies to reduce the occurrence of bot attacks

- Identify out-of-date browser agents, which often disguise legacy bots. Access to the website can be denied by blocking superseded versions of common browsers – for example, versions of Firefox, Chrome, Internet Explorer and Safari that are more than two years out of date. An alternative is to insist on human logic for login access, such as using the CAPTCHA process.
- Monitor website traffic. Spikes in traffic can be assessed and the sources identified. If a single specific source is indicated, then a bot is responsible for the increase in apparent business. Failed logins may also indicate bad bot activity.
- Be alert for public data breaches. Whenever credit card details are stolen, they are quickly deployed to run the credentials into your website. Be aware of financial authorities' warnings and updates to ensure that credit cards are genuine.



**FIGURE 6.15**  
A CAPTCHA process can reduce the likelihood of a bot attack.

#### SCHOOL-ASSESSED TASK TRACKER

- Design brief
- Project plan
- Analysis
- Folio of alternative design ideas
- Development
- Evaluation and assessment
- Final submission

## Session hijacking

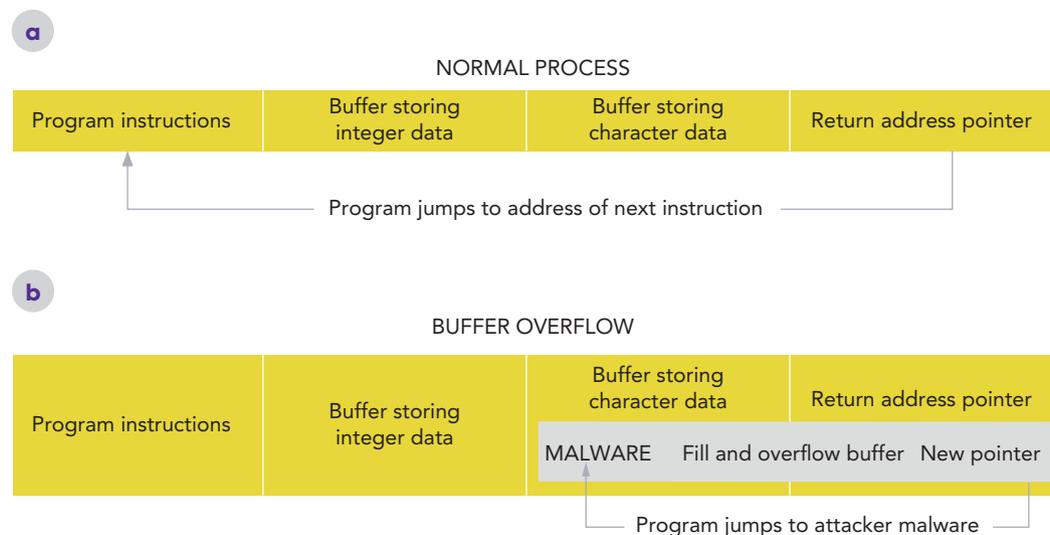
The integrity of transactions across the web depends on certainty of identity – if users are who they say they are. Usually when an account holder logs in, a session token is issued. All transactions that take place in that login session exchange the token to verify the identity of the user. Session hacking is a variation on eavesdropping, where the attacker attempts to impersonate the legitimate user. Benefits of session hijacking depend on what is dealt with in the session, which typically ranges from shopping carts to credit card details.

The vulnerability is caused when session identifier tokens are stored or transmitted insecurely. A simple solution is to ensure that the internet connection is secure and uses the encrypted protocols HTTPS and SSL.

## Buffer overflow

When a process attempts to store data in RAM beyond the assigned fixed length storage buffer, an overflow into neighbouring memory locations may occur. Under certain conditions, the overflow data may corrupt data, or it can stop functioning by crashing the computer or being replaced with attacker code to provide alternative instructions for execution once the program restarts.

Buffer overflow attacks are well known, but legacy and newly developed applications continue to be vulnerable. The basis of the success is violation of a programmer's assumptions.



**FIGURE 6.16**

**a** The normal process;  
**b** Buffer overflow.  
The simplest buffer overflow attack is the most common.

Protection against buffer overflow attacks can be provided by ensuring that all code that accepts input from users via an HTTP request is reviewed to make sure it provides appropriate size checking on all such user inputs.

Java and Python are interpreted programming languages, and are immune to these attacks.

## Restrict URL access

Undertake access control checks each time a page is accessed. This will prevent attackers entering URLs directly into browsers, meaning hidden or orphan pages cannot be accessed, as permission will be denied to unauthorised requests.

## Risks from software acquired by third parties

Many software products are offered as a subscription rather than as a 'purchase'. Traditionally, a software purchase was a payment for a licence to use the software on a 'as is' basis. The software was not permitted to be modified or re-engineered.

One direct result of the subscription model is to maintain the distributed user base of software and ensure that software is up to date. Automatic software updates prompt the user to ensure that the latest security patches will be installed.

Other software purchased and installed may have been registered with the developer. This registration usually provides update notifications or access to a web page that provides information about security issues and fixes.

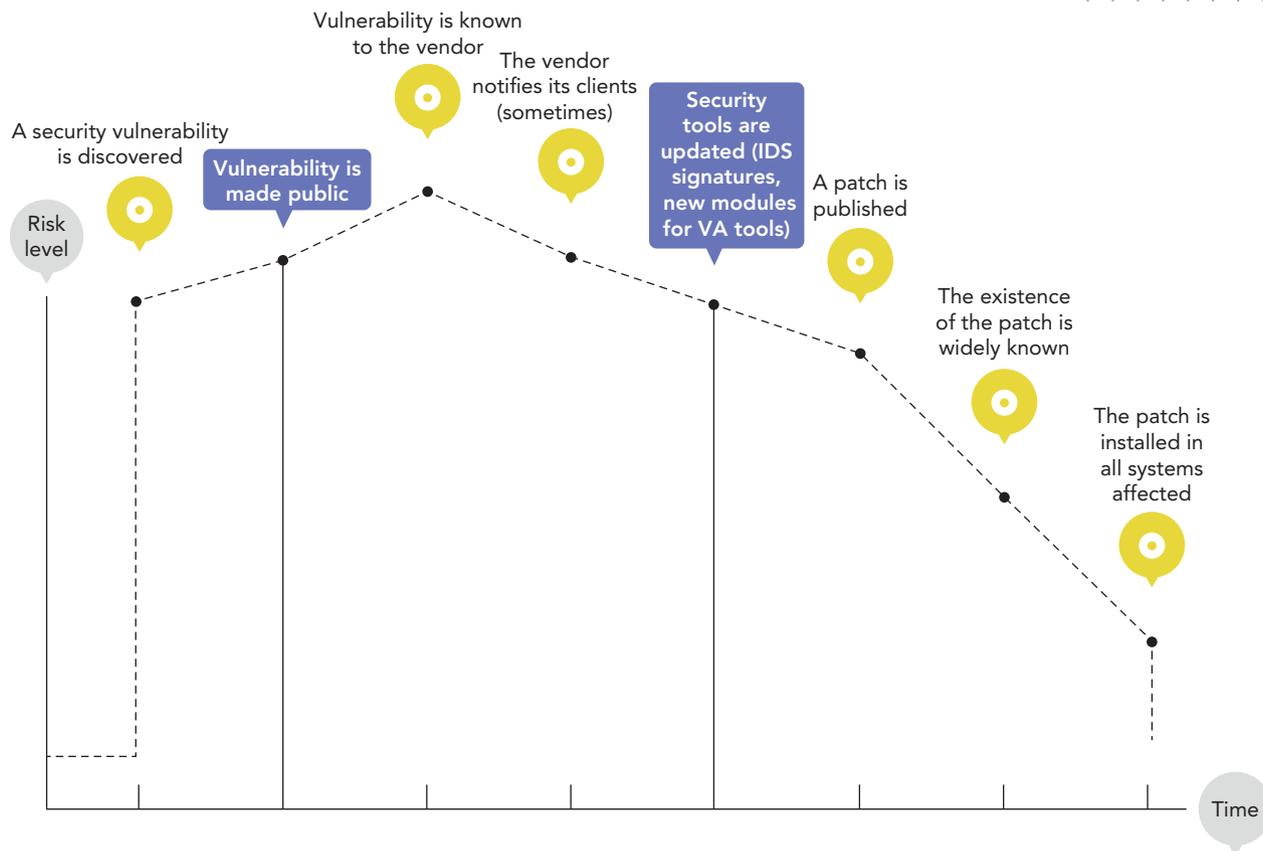
Overall, the user is reliant on the software developer or distributor for notification about necessary improvements and modifications to the original product.

Use of third-party libraries and components (APIs) may introduce security weaknesses that can be exploited. Integrated code that has not been tested is an increasing threat. Contents of libraries may be restricted from view and unable to be inspected. All components should be tested and approved prior to incorporating into the production code.



Source: Adobe Systems Incorporated

**FIGURE 6.17** Adobe Creative Cloud is a subscription service.



© OWASP. Accessed from <https://www.owasp.org/images/1/19/OTGW.pdf>. Released under CC BY Attribution-ShareAlike 4.0 International, link to license <https://creativecommons.org/licenses/by-sa/4.0/>

**FIGURE 6.18** Window of vulnerability for software. Patches, once announced, are not applied immediately. Many applications may continue unpatched for years, until discovered by malware or ransomware.

### SCHOOL-ASSESSED TASK TRACKER

- Design brief
- Project plan
- Analysis
- Folio of alternative design ideas
- Development
- Evaluation and assessment
- Final submission

From Google Terms of service: 'Software ... You may not copy, modify, distribute, sell, or lease any part of our Services or included software, nor may you reverse engineer or attempt to extract the source code of that software, unless laws prohibit those restrictions or you have our written permission ...'.

Extract from Google terms of service: Software, accessed from <https://policies.google.com/terms?hl=en>

The attraction of APIs for hackers is the automatic nature of the processing. Once credentials are accepted, the API proceeds uncritically to process the input.

Web applications often rely on third-party services and application interfaces. For example, Google Forms is often used to host a survey activity that is used for purposes unrelated to Google. Disclaimers warn users that their details are not secure.

## Ineffective code review practices

Ineffective code review practices can allow security flaws and coding errors to go undetected. When code reviews are inconsistent, too brief or sloppy, critical issues such as buffer overflows, SQL injection vulnerabilities and other security weaknesses can slip through. This can lead to the deployment of insecure software that is susceptible to attacks, potentially resulting in data breaches, system outages and compromised user data. Comprehensive and systematic code reviews are essential for identifying and mitigating these risks before software is released.



### RESEARCH



Weblink  
Microsoft  
article

#### Microsoft Exchange zero-day

Microsoft Exchange Server vulnerabilities (CVE-2022-41040 and CVE-2022-41082) were discovered in 2022. These zero-day vulnerabilities, collectively known as ProxyNotShell, were exploited in targeted attacks due to weaknesses in the code review and patch management processes. Attackers were able to leverage these vulnerabilities to execute remote code and establish persistent access to affected systems. The lack of effective code reviews and timely patch application highlighted the importance of robust code review practices in identifying and mitigating security flaws before they can be exploited in the wild.

Read more about it in the article, 'Analyzing attacks using the Exchange vulnerabilities CVE-2022-41040 and CVE-2022-41082' on the Microsoft website via the weblink.

## Combined environments

Combining development, testing and production environments allows untested or insecure code to potentially make its way into the production environment, making it vulnerable to exploitation. Sensitive production data may be exposed to developers and testers who do not have the same security clearances, increasing the risk of data breaches. Moreover, the lack of environment separation complicates the enforcement of access controls and audit logs, making it harder to track changes and identify security issues. This integration also means that testing conditions may not accurately replicate the production environment, leading to missed vulnerabilities and security flaws.

### Capital One data breach

The 2019 Capital One data breach occurred because a vulnerability in a web application firewall allowed an attacker to access sensitive customer data stored in the cloud. One contributing factor was the use of combined environments, which enabled the attacker to exploit configuration weaknesses across development and production environments. This incident exposed personal information of over 100 million customers, including bank account details, highlighting the need for maintaining strict separation between development, testing and production environments to enhance security and protect sensitive data.

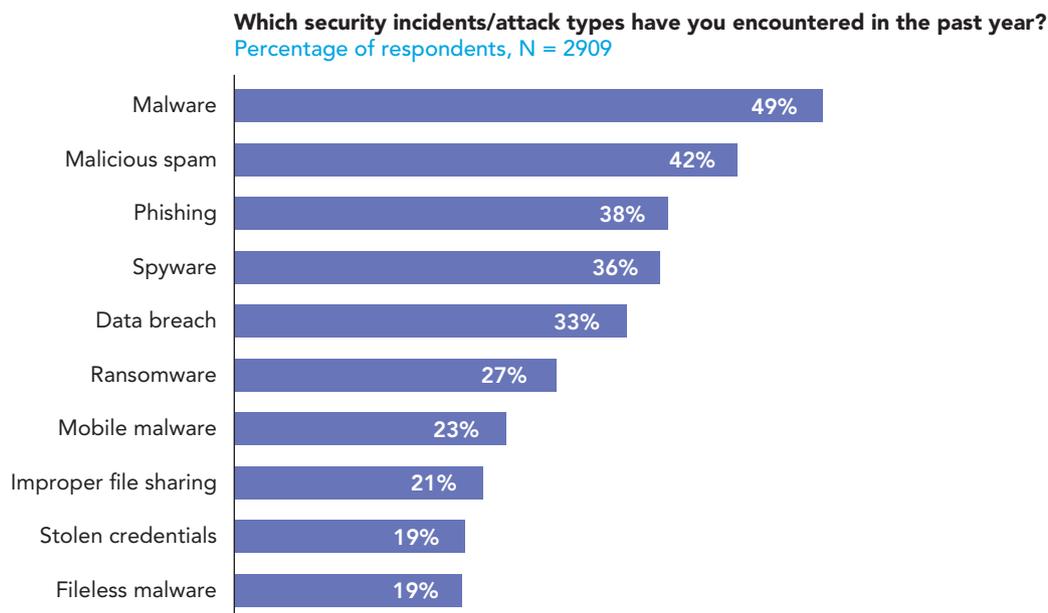
### RESEARCH

## Software security

There is no single way of securing information.

To develop secure applications, begin by assuming that all data received by the application is from an untrusted source. This applies to all data received – data, cookies, emails, files or images. This includes users who have logged in to their account and authenticated themselves. There is no preferential treatment to any user; all should be subject to the entire range of security measures.

Not trusting user input means always validating it for type, length, format and range. The data is challenged each time it is processed – for example, when data is entered through a web form to an application script, and encoded before being displayed on a dynamic page. Logistically, this means that any values accepted from the client side are checked, filtered and encoded before being passed back to the user. Any user-supplied data handling and processing must be certified as secure.



**FIGURE 6.19** A survey across many businesses and several countries identified the most frequent risk of attacks. Notably, the first three all use email to gain access to a computing device.

Based on data from Cisco's Cybersecurity Series 2019: 'Anticipating the Unknowns: Chief Information Security Officer (CISO) Benchmark Study, March 2019.'

### SCHOOL-ASSESSED TASK TRACKER

- Design brief
- Project plan
- Analysis
- Folio of alternative design ideas
- Development
- Evaluation and assessment
- Final submission

**THINK ABOUT  
SOFTWARE  
DEVELOPMENT**

6.4

'The Australian Government will spend \$586 million on the 2023–2030 Australian Cyber Security Strategy. Some of these funds are allocated to defending against cyber attacks as part of the government's Cyber Security Strategy. Quick responses are required in the event of a targeted attack against government IT systems. Ransomware is an increasing threat, and globally, there is an estimated attack on a business every 11 seconds.'

Adapted from ABC News article 'Australian Cyber Security Strategy outlines how government plans to tackle cyber crime' by Evelyn Manfield, 21 November 2023

- 1 Use the weblink below to research the details of the Cyber Security Strategy.
- 2 What is Australia's 2030 vision?
- 3 What are the six cyber shields to achieve this vision?
- 4 How might these cyber shields affect you, directly or indirectly?



**Weblink**  
Australia's Cyber  
Security Strategy

## Strategies for minimising potential risks

A recommended attitude to security would be that risks can be reduced rather than eliminated. When new threats and variations on existing threats are revealed, quite often the basics need to be checked, and well-known, established solutions implemented.

### Software auditing and testing strategies

During software development there are many opportunities to test the software for compliance with expected outputs. An end-to-end strategy ensures that testing takes place at every stage of the software development cycle. The Open Worldwide Application Security Project (OWASP) has published a recommended testing framework that may be considered to form the basis of the development workflow. OWASP provides software tools that can perform tests for known vulnerabilities. Sample code is provided which will 'break' the web application.

A cautionary comment by Michael Howard, a software security expert at Microsoft, at an OWASP AppSec Conference held in Seattle (2006) was:

Tools do not make software secure! They help scale the process and help enforce policy.

The Foreword to the OWASP Testing Guide also warns that companies should not rely on OWASP's security software alone:

Most importantly, these tools are generic - meaning that they are not designed for your custom code, but for applications in general. That means that while they can find some generic problems, they do not have enough knowledge of your application to allow them to detect most flaws. In my experience, the most serious security issues are the ones that are not generic, but deeply intertwined in your business logic and custom application design.

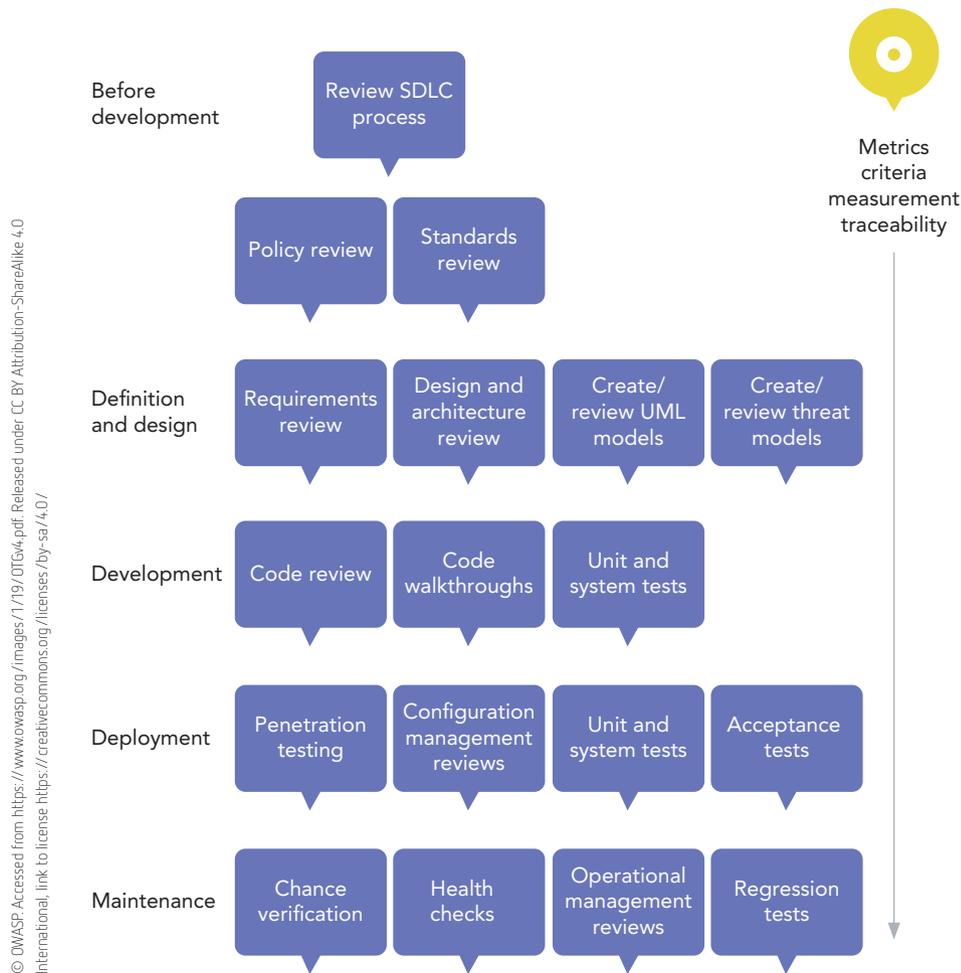
Quote from Michael Howard at the 2006 OWASP AppSec Conference in Seattle and extract from the Foreword to the OWASP Testing Guide accessed from [https://www.owasp.org/index.php/Testing\\_Guide\\_Foreword](https://www.owasp.org/index.php/Testing_Guide_Foreword). Released by CC BY-SA 4.0 International, link to license: <https://creativecommons.org/licenses/by-sa/4.0/>

### Software auditing

A software security audit may be conducted separately or as part of a larger overall software audit. Auditing of software may have several meanings. One meaning looks at how the software works. Another meaning is to review the software for compliance with specified standards. Does the construction of the software follow the rules? Has any licensing been noted and recorded? For example, the costs of licensing should be looked at when a site licence is held for 10 seats but only six people are ever using the software, or when a licence for 10 is being paid for but 30 people are using the software every day. This issue is not about how the software works; rather, it is a consideration of the risk of legal action and cost–benefit of a particular software solution.

The purpose of a software review is to:

- uncover any issues or problems early; it is simpler and costs less to fix an issue earlier in the project
- improve performance, scalability and reliability



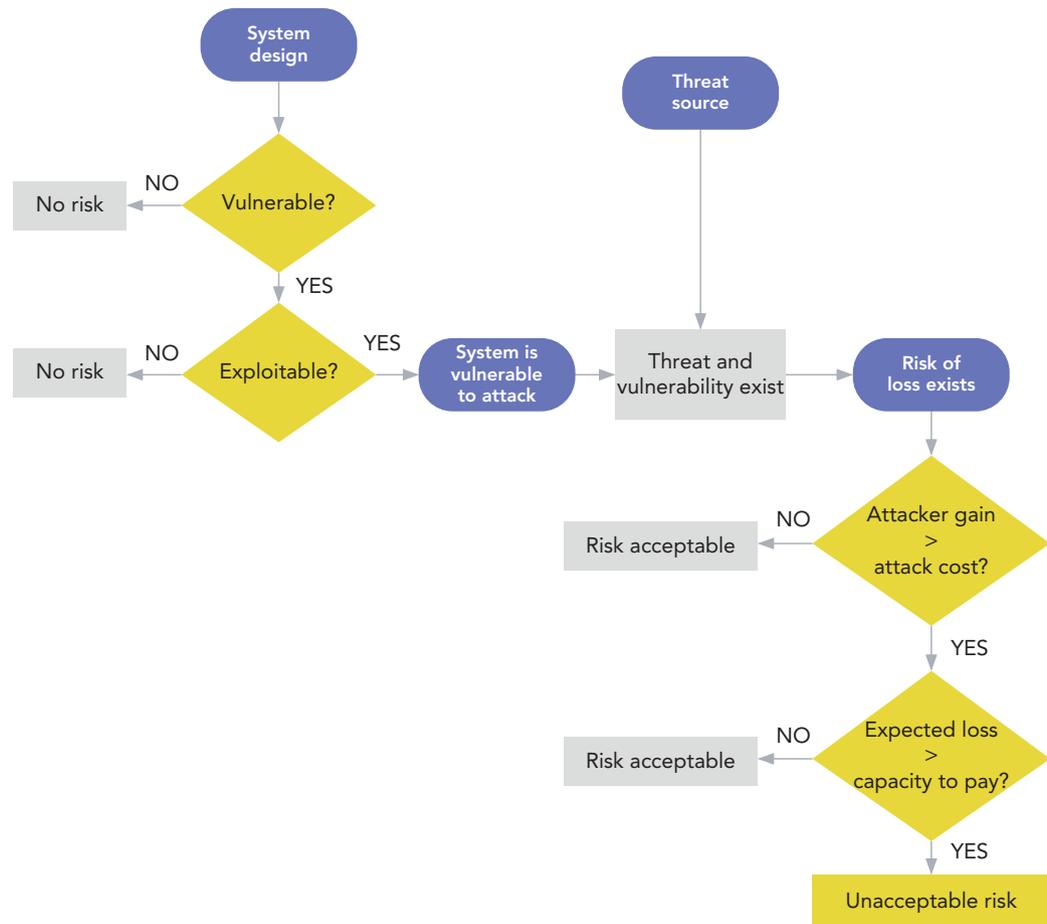
**FIGURE 6.20** The OWASP Testing Guide outlines a comprehensive security strategy for the software development life cycle.

- comply with specific standards
- review any necessary or unnecessary testing
- ensure that the application can be maintained and extended in the future
- make sure you use the appropriate technology for the job
- satisfy legal and licensing requirements.

The software audit may be an internal or external review; that is, the review may be carried out by someone independent of the developer team, or by the developer team itself. The software audit may use analysis tools to gather data on the performance of the software for security or for functionality.

A typical software audit process will usually:

- document all uses and demands of the software
- test the software for standards compliance, also known as pen testing
- identify assets necessary for the operation of the software, including extreme case limits
- identify security configurations and compare with acceptable security settings
- consider levels of user training necessary for the operation of the software.



**FIGURE 6.21**  
Determining unacceptable security risk

These findings would be documented and a report prepared. A management group would consider the findings and authorise any further action. Part of the consideration of the report would be to look at risk management and cost–benefit analysis. The directors would consider the legal obligations and, once satisfied that all legal compliance had been met, gauge the level of risk that the organisation would be prepared to accept.

If it is not possible to eliminate all risks, the question then becomes, ‘What level of risk is tolerable?’

**Risk tolerance**, also known as **risk appetite**, is determined by balancing the expense in terms of financial resources and usability of information assets against financial liability, loss of information assets and reputational damage if the risk is exploited.

The steps in determining unacceptable risk are as follows.

- When a flaw or weakness exists, reduce the chances of the vulnerability being exploited by implementing security controls.
- When the vulnerability can be exploited, prevent the opportunity for attack by applying changes in design or administrative controls, or increase protections.
- When an attacker’s gain is greater than the costs of attack, apply measures to increase the attacker’s costs or reduce the attacker’s gains so the gain is not worth the effort.
- When the predicted losses are beyond the ability to absorb the costs, use technical and non-technical protection to limit the impact of an attack and the amount of losses. For example, at the first sign of a DDoS attack, switch the IP address of the server to maintain connections.

## Penetration testing

Penetration testing (or pen testing) identifies security vulnerabilities in web applications. This is achieved by challenging every page and line of code in the application for known weaknesses. This could be a very time-consuming process if attempted manually. Fortunately, there are automated security tools that will perform continuous web application penetration tests. These automated tests rely on up-to-date descriptions of the parameters that can cause a vulnerability. Each application must be scanned, or crawled, and each parameter must be tested for compliance with OWASP security standards.

The OWASP guide states:

the most important thing to remember when performing security testing is to continuously re-prioritize. There are an infinite number of possible ways that an application could fail, and organisations always have limited testing time and resources. Be sure time and resources are spent wisely. Try to focus on the security holes that are a real risk to your business.

Extract from OWASP 'Testing Guide Foreword', accessed from [https://www.owasp.org/index.php/Testing\\_Guide\\_Foreword](https://www.owasp.org/index.php/Testing_Guide_Foreword). Released by CC BY-SA 4.0 International, link to license: <https://creativecommons.org/licenses/by-sa/4.0/>

## Identifying software and data vulnerabilities

The best defence is to maintain awareness of the possibilities as they may apply to your immediate and future situation. Many of the vulnerabilities are adapted from analogue frauds, scams and confidence tricks that have existed for more than a century. Deception and misrepresentation are common factors in a large number of the schemes that have been successful in gaining unauthorised access and performing illegal or corrupting actions with data and information.

The largest single common factor is loss of trust with employees or insiders. More than 40 per cent of all criminal activities are perpetrated by people who have inside knowledge or access to the IT systems.

## Data breaches

The number of users of the internet has increased such that:

- commercial company activities are now online
- organisations use the cloud for storage
- big data is collected routinely across the internet and social media
- greater amounts of information can be created from data due to improved processing speeds and increased storage
- routine retail transactions of consumer banking and purchasing are carried out online.

### THINK ABOUT SOFTWARE DEVELOPMENT

6.5

The Australian Notifiable Data Breaches (NDB) scheme was applied from 22 February 2018.

The *Privacy Amendment (Notifiable Breaches) Act 2017* applies to all agencies and organisations regulated under the *Australian Privacy Act 1988*.

'The NDB scheme includes an obligation to notify individuals whose personal information is involved in a data breach that is likely to result in serious harm. The notification must include recommendations about the steps individuals should take in response to the breach. The Australian Information Commissioner (Commissioner) must also be notified of eligible data breaches.'

Office of the Australian Information Commissioner – [www.oaic.gov.au](http://www.oaic.gov.au).  
Extract from 'Notifiable Data Breaches scheme'. Released under CC BY 3.0 AU, link to license: <https://creativecommons.org/licenses/by/3.0/au/deed.en>

Examples of a data breach include the following incidents:

- a device containing customers' personal information is lost or stolen
- a database containing personal information is hacked
- personal information is mistakenly provided to the wrong person.

Research the latest (known) data breach locally and globally.

- a Identify the nature of the breach.
- b How many people were affected?
- c What was the impact of the breach?

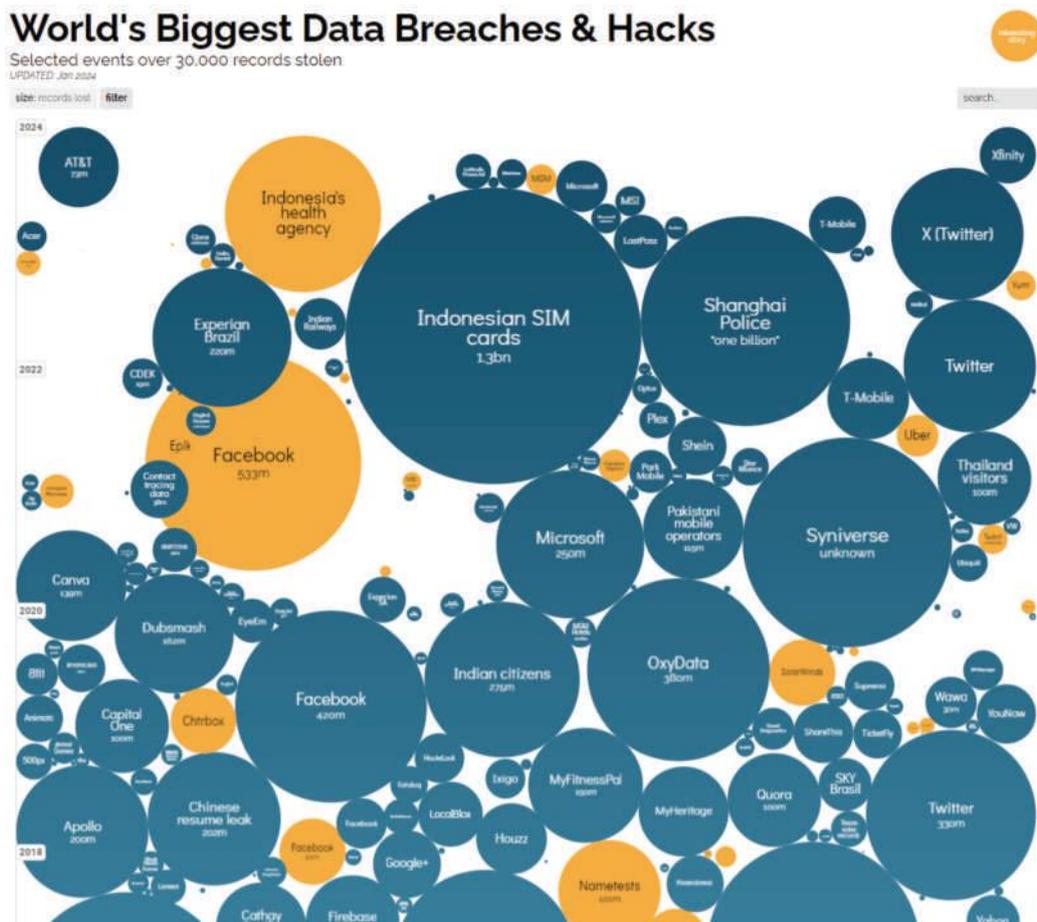
### SCHOOL-ASSESSED TASK TRACKER

- Design brief
- Project plan
- Analysis
- Folio of alternative design ideas
- Development
- Evaluation and assessment
- Final submission

Accompanying these increases are announcements of data breaches that have become commonplace. This is partly due to new regulations that require anyone affected to be notified (see Think About Software Development 6.5 on the previous page), and partly because there are more breaches being discovered. There are many examples of huge numbers of usernames and passwords being accessed by unauthorised actors. This may come about because:

- a web page looks like a legitimate site and tricks a user into entering their credentials
- there is an automated brute-force attack against a service that does not prevent such an attack
- a service is accessed by an outside party and credentials are stolen, which are then used to access social media and email
- credentials are stolen by malware.

Breach categories can be filtered on the interactive infographic in Figure 6.23.



**FIGURE 6.22** A selection of data breaches; only those with over 30 000 records are shown.

## Threat modelling

Threats represent a constant danger to an asset, and while the details will vary according to the specific situation, threats have been well researched and are well understood.

### Threat modelling principles

In the current threat environment, cyber attacks have become expected.



# THREAT MODELING MANIFESTO

## What is threat modeling?

Threat modeling is analyzing representations of a system to highlight concerns about security and privacy characteristics.

At the highest levels, when we threat model, we ask four key questions:

1. What are we working on?
2. What can go wrong?
3. What are we going to do about it?
4. Did we do a good enough job?

## Why threat model?

When you perform threat modeling, you begin to recognize what can go wrong in a system. It also allows you to pinpoint design and implementation issues that require mitigation, whether it is early in or throughout the lifetime of the system. The output of the threat model, which are known as threats, informs decisions that you might make in subsequent design, development, testing, and post-deployment phases.

## Who should threat model?

You. Everyone. Anyone who is concerned about the privacy, safety, and security of their system.



Weblink  
Threat Modeling  
Manifesto

**FIGURE 6.23** The Threat Modeling Manifesto was published in 2001.

Source: Threat Modeling Manifesto

Modelling exercises attempt to identify all possible threats and plan to prevent the worst possibilities from occurring.

An estimation and rating of what is the worst that could happen must be made. These value judgements need to involve all levels of management and workers, as different perspectives and priorities inevitably exist across an organisation. Another motivation to involve a cross-section of the business is to develop a shared understanding of security risks and set priorities to build a fix to potential problems. One ‘ultimate catastrophe’ is to be fined and jailed, though that would not be a realistic option for many who work for the company and would not be under consideration. More likely, a suspension of operations and lay-offs would occur due to lack of income to pay wages. Very few companies have the capacity to withstand lengthy periods without income.

There are several **threat modelling principles**. For example:

- OWASP
- Microsoft STRIDE (general context for any software)
- VerSprite PASTA (process for attack simulation and threat analysis – business context).

All have common themes, which typically include:

- description of the security requirements
- assumptions that can be checked over time as the threats change
- identification of potential threats
- actions that can mitigate each threat
- validation of the model and threats, and confirmation/verification of successful action taken.

Updating of threat modelling is advised after:

- a new feature is released
- a security incident occurs
- organisational or infrastructure changes.



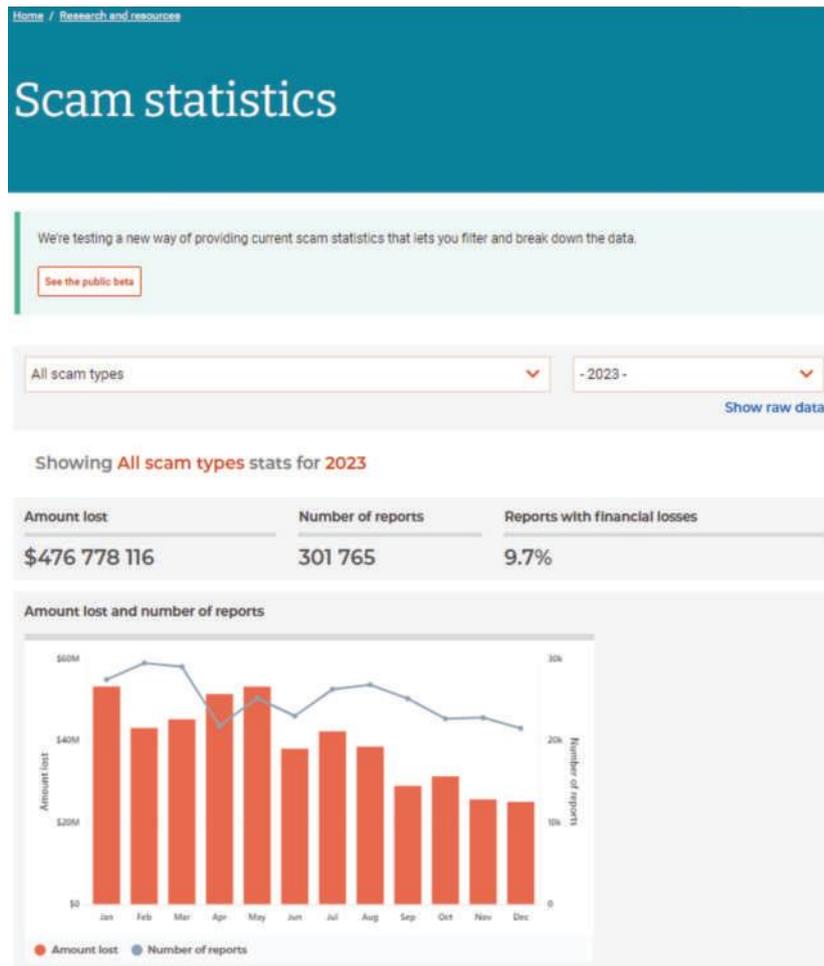
Weblinks  
OWASP  
Microsoft STRIDE  
VerSprite PASTA

## Detection and prevention

In Australia, the Australian Competition and Consumer Commission (ACCC) maintains a Scamwatch website, where instances of the frequency and estimated costs of various scams are presented.

The Australian Cyber Security Centre (ACSC) provides:

- monitoring of cyber threats across the globe 24 hours a day, seven days a week so that Australians can be alerted early on about what to do
- advice and information about how to protect individuals and businesses online
- clear and timely advice to individuals, small to medium business, big business and critical infrastructure operators when there is a cyber security incident
- investigation and solution development to cyber security threats
- services to fight **cybercrime**.



**FIGURE 6.24** ACCC scam statistics for 2023

Source: Commonwealth of Australia



**FIGURE 6.25** The Australian Cyber Security Centre is part of the Australian Signals Directorate, which works in matters of intelligence, cyber security and offensive operations.

Security procedures related to malicious software are shown in Table 6.2.

**TABLE 6.2** Security procedures for malicious software or malware

| Detection  | Prevention equipment   | Prevention policies   | Countermeasures  |
|--|--|---|--|
| <p>Malicious software (malware) may cause:</p> <ul style="list-style-type: none"> <li>changes in file sizes or date/time stamps</li> <li>computer slow starting or slow running</li> <li>unexpected or frequent system failures</li> <li>change of system date/time</li> <li>low computer memory or increased bad blocks on disks</li> <li>unexpected data loss or file corruption.</li> </ul> <p>Performing a virus scan may uncover potential threats.</p> | <p><b>Firewall:</b> A firewall is a hardware device or software that blocks unauthorised access from particular locations or to particular parts of a computer.</p> <p>For example, web traffic is normally sent and received on port 80 of your modem; a firewall can be configured to block access from all other ports.</p> <p>Firewalls are very good at stopping the spread of worms but ineffective with other types of malware.</p> <p>A network firewall resides between two networks (usually the internet and the organisation's network).</p> <p>A client firewall is software that runs on the end user's computer.</p> <p><b>Antivirus software:</b> Antivirus software can scan your system for known viruses or detect system behaviour that may indicate an infection.</p> <p><b>Spam filter:</b> Worms often present themselves as an email attachment. Having an effective spam filter with up-to-date definitions can prevent this.</p> | <p>Limit user privileges. Limiting user privileges will limit the damage that can be done.</p> <p>Limit user connectivity and user downloads.</p> <p>Limit executable files and limit media that can be used for loading data and software.</p> <p>Provide thorough user training. Many threats, including Trojan horses, rely on inadvertent user cooperation. User training is the best prevention.</p> <p>Perform security upgrades. Upgrade via security patches as soon as they become available.</p> <p>Keep all virus and spam definition files up to date.</p> <p>Perform regular backups. In the event that infection cannot be prevented, the data needs to be able to be efficiently restored.</p> | <p>Quarantine affected equipment and remove from the network. Infected computers may need to be completely reformatted and all data destroyed.</p> <p>Run antivirus software and attempt to locate and destroy known viruses.</p> <p>Determine source of infection and issue alert. Depending on the nature of the threat, the alert may need to be issued not only to other system admins, but to all network users.</p> <p>Restore any lost data with most recent backups and replace any damaged equipment.</p> <p>Enact business continuity plan in order to limit the amount of downtime and minimise the negative consequences of the infection.</p> |

## Defining security requirements

Defining security requirements involves identifying and establishing the necessary security controls and measures to protect a system or application against potential threats. This is a foundational step in the threat modelling process, ensuring that security is integrated into the system design from the beginning.

### Who and what needs protecting?

The first step in defining security requirements is to identify the assets that need protection and the stakeholders involved. Assets can include data, applications, systems and infrastructure. Stakeholders may include users, administrators, developers and business owners.

Understanding what needs protection and who is involved helps in tailoring the security requirements to meet specific needs.

SCHOOL-ASSESSED TASK TRACKER

Design brief
  Project plan
  Analysis
  Folio of alternative design ideas
  Development
  Evaluation and assessment
  Final submission

## Determining threats

It is important to identify potential threats and vulnerabilities that could impact the system. Common threats include unauthorised access, data breaches, malware attacks and denial-of-service attacks. By analysing these threats, organisations can determine what kind of security controls are necessary to mitigate them.

## Reacting to threats

Based on the identified threats, specific security requirements are developed. These requirements are detailed descriptions of the security controls needed to protect the system. They can include technical measures, such as encryption, access controls and intrusion detection systems, as well as procedural measures, such as security policies and training.

Not all security requirements can be implemented simultaneously, especially if resources are limited. Therefore, it is important to prioritise the requirements based on the potential impact of threats and the likelihood of their occurrence. High-priority requirements should address the most critical threats that could cause significant harm to the organisation.

## Continuous review

Threats to security are constantly evolving and so should the security requirements. Regular reviews and updates are necessary to adapt to new threats, changes in the system and emerging security technologies. This continuous improvement process ensures that the security requirements remain relevant and effective over time.

## Identifying and mitigating threats

Threats to data integrity can be accidental, event-based or deliberate.

### Accidental threats

There are several ways users can cause accidental damage to information systems. It is important to make a distinction between inadvertent errors, which may be unlikely or unforeseen, and those errors that occur due to poor training or poor design of user interfaces (UI), or an increasing non-compliance with recommended protocols. Mistakes rarely 'just happen'.

Mitigating circumstances or contributing factors often include:

- user inattention or carelessness
- confusing screen design with lookalike interface
- lack of confirmation before execution; all changes are 'live' with no simple backup process
- inappropriate permissions, which allow untrained users access to modify or delete strategic files or settings.

### Event-based threats

Event-based threats are usually external to the organisation and beyond the immediate control or influence of security planning to prevent. The only course of action is to prepare contingency plans in the event that the threat occurs. Well-considered recovery plans can minimise the impact on the organisation.

Event-based threats include:

- failure of storage – for example, HDD or SSD
- power failure

- file corruption
- using cloud storage where power outage, software malfunction or data breach occurs
- using third-party software where software malfunction or data breach occurs
- acts of nature – for example, fire, flood, earthquake and lightning strike/power surge.

Similar strategies can be used to prevent accidental loss or event-based damage.

A **hot site** maintains a full or partial duplicate for a primary IT operation, including complete computer systems and near-real-time backups for systems, applications and data. In its most expensive form, mirroring software is used to keep a hot backup site and a primary site synchronised. A hot site is used when an organisation can tolerate little or no downtime, and is where a switchover may be achieved in a few hours.

A **cold site** by contrast may take a few weeks to be established following a catastrophic systems failure, though the business may be able to operate at a reduced level without full IT services. Increasingly, businesses are more dependent on IT, so **warm sites** (also known as mobile sites) become attractive. The time for startup depends on how quickly access to the restored backup can be made operational.

### Deliberate threats

Many of the details of deliberate threats have been discussed earlier in this chapter. Table 6.4 provides a summary of those threats, prevention strategies and actions to be taken. One particular security strategy, known as a **honeypot**, has been widely adopted to detect, deflect or counteract unauthorised access to computer systems. The honeypot acts as a decoy for trapping hackers or tracking new **hacking** methods, or simply as a delay by wasting hackers' time. Honeypots are established solely to take attention away from the main company servers. They have no other purpose, and legitimate users will not connect with the honeypot. Attackers can be identified and traced if a honeypot has logged the attacker IP details. Multiple honeypots across a network may form a **honeynet**. Honeypots can also be used on an internal network to prevent authorised users browsing beyond their immediate permissions.

**TABLE 6.3** Summary of event-based threat detection, prevention and recovery actions

| Detection   | Prevention  | Prevention action  | Recovery  |
|---|---|--|---|
| Unexpected computer system behaviour: <ul style="list-style-type: none"> <li>• data loss</li> <li>• slow performance</li> </ul> | Ensure that diagnostic monitoring operates continuously, regularly checking condition of storage media.<br><br>Periodic full backups are enabled. Incremental backups with short intervals are enabled, dependent on activity and cost effectiveness. | Maintain RAID and incremental backup protocols. Diagnostic tools constantly monitor hardware and software. Alerts are issued when anomalous events occur.<br><br>Uninterruptable power supply (UPS) provides sufficient time to save all data and settings before controlled shutdown. | Restore data with most recent backups.<br><br>Replace damaged, lost or stolen equipment.<br><br>Activate disaster recovery plan (DRP) to limit losses and damage. |
| Disaster alert from emergency services and smoke alarms   | Ensure that regular backups are recorded.<br><br>Establish off-site duplicate of IT services when the base operation is unavailable.  | Fire services deployed appropriate to the circumstances. Fire detection/smoke alarms. Halon gas reduces potential for flame and water damage.<br><br>Uninterruptable power supply (UPS) provides sufficient time to save all data and settings before controlled shutdown.             | Consider hot or cold site recovery plan solution.<br><br>Restore data with most recent backups.   |

**TABLE 6.4** Summary of deliberate threat detection, prevention and recovery actions

| Detection   | Prevention  | Prevention action   | Recovery   |
|---|---|---|--|
| Malicious software or malware: viruses, worms, Trojan horses, logic bombs, spyware, back doors and <b>keyloggers</b> may cause: <ul style="list-style-type: none"> <li>unexpected file losses or corruption</li> <li>slowly running computer</li> <li>unusual file sizes or date/time stamps</li> </ul> | Permissions are restricted for all users.<br>Establish download limits and email attachment limits.<br>Ensure that all users are trained in recognising the dangers of 'unsafe sources' of software and untrusted sources of documents.<br>Install and run anti-virus software and SPAM filters.<br>Run anti-virus scans on incoming email.   | Firewall settings block unauthorised access to the computer or the network.<br>Spam filters are constantly available to scan all email.<br>Set anti-virus software to scan files introduced on USB drives.  | Quarantine affected files and any infected computer from the network. Reformat the drives and wipe all data.<br>Restore lost data from latest incremental backup.  |
| Cyber attackers: hackers, spies, insiders, criminals and terrorists; activity is sometimes difficult to detect  | Intrusion detection system (IDS) monitors system and file activity to identify and signal an alert whenever anomaly or violations occur.  | Constant vigilance is necessary, as detection is slow and false alarms are more frequent than actual attacks.   | Data breaches require notification of those affected (see Chapter 7).<br>Loss of usernames and passwords will necessitate re-establishing authenticity for users. Suspend all account activity until all passwords are reset through <b>two-factor recovery</b> process. |
| Social engineering attacks: phishing, hoaxes, <b>credential stuffing</b> , fraudulent offers, <b>impersonation</b> , pharming, <b>shoulder surfing</b> , tailgating, spear phishing, vishing, <b>smishing</b> ; often difficult to detect.  | Ensure that all users are trained in recognising the characteristics of social engineering and receive regular updates on the most recent versions.<br>Restrict the use of company SMS and email accounts to strictly business purposes.<br>Ensure that user email subscriptions are limited to essential contacts to minimise SPAM.<br>Require unique passwords that are not shared across different purposes. | Social engineering is, at its most basic, about confidence tricks, or the 'con'. Vigilance is necessary at all times with email messages, SMS and websites.<br>Limit financial authority within the organisation to ensure that only legitimate orders are taken and payments made.<br>Eliminate passwords and substitute biometric login and verification of ID. | Depends on the instance, and whether the violation is detected, or reported. Often embarrassment prevents the reporting of these criminal activities.  |

## Confirming threats have been mitigated

Several processes can be put in place to ensure a thorough and effective approach to cyber security to confirm that security threats have been mitigated.

### Security assessments

Regular security assessments are essential, starting with vulnerability scanning and penetration testing. These assessments help identify and address vulnerabilities before they can be exploited. By conducting these tests periodically, organisations can ensure that their defences remain robust against evolving threats.

### Monitoring and logging

Implementing continuous monitoring and logging is another critical process. By constantly monitoring network traffic, system activity and user behaviour, organisations can detect and respond to suspicious activities in real time. Logging all activities allows for detailed analysis and auditing, helping to trace and understand security incidents, which is essential for both mitigation and compliance purposes.

## Patch management

Patch management processes ensure that all systems and applications are up to date with the latest security patches. This reduces the risk of vulnerabilities being exploited. This process involves not only applying patches promptly but also verifying that they are correctly implemented and do not introduce new issues.

## Access control reviews

Access control reviews should be conducted periodically to verify that access permissions are appropriate and aligned with the principle of least privilege. This process involves reviewing user access rights, ensuring that they match current job roles and revoking unnecessary privileges. Such reviews help prevent unauthorised access and potential data breaches.

## Training

Security awareness training for employees is another important process. Educating staff about the latest threats and best practices for cyber security helps create a security-conscious culture within the organisation. Regular training sessions and simulations can help employees recognise and respond appropriately to potential threats, reducing the risk of human error leading to security incidents.

## Intrusion detection and prevention

Deploying and maintaining advanced security technologies, such as intrusion detection and prevention systems, firewalls and antivirus software, is also important. These technologies provide multiple layers of defence against a wide range of threats, and regular updates ensure that they remain effective against new and emerging threats.

## Security audits

Finally, conducting regular security audits and compliance checks helps ensure that all security measures are in place and functioning as intended. These audits, performed by internal teams or external experts, assess the overall security posture of the organisation and identify any areas that require improvement. Compliance checks ensure that the organisation meets industry standards and regulatory requirements, which often include specific security measures and practices.

## Next steps

This chapter has discussed cyber security and some of the main threats to physical and logical (software) security. Software development processes continually prepare for maintenance of confidentiality, integrity and availability. Those entrusted with preparing security plans regularly update their understanding of social engineering, web application threats and the countermeasures required to prevent such attacks. In the next chapter, you will consider the legal requirements and obligations for software providers, and those who provide customer services involving personal data and information.

The final assessment for Unit 4 will be Outcome 2, which requires you to consider a case study and provide recommendations on modifications after exploring security and legal implications.

# 6

## CHAPTER SUMMARY

### Essential terms

**back door** allows remote access to a computer system, circumventing all security processes

**botnet zombies** a coordinated network of compromised or 'zombie' computers used to transmit malware or spam or to launch DDoS attacks

**cold site** prepared external recovery IT infrastructure; when the duplicate is not immediately ready to replace operations, and may take a few weeks to resume normal operations

**credential stuffing** passwords are reused across different applications with similar ID; for example, work login and Facebook have the same password, user IDs are different but are linked by birthday

**cybercrime** targeted attacks against financial networks, gaining unauthorised access to information and stealing sensitive information, extortion and blackmail use of exfiltrated data and information

**distributed denial-of-service (DDoS) attack** an attack that overwhelms a web server by targeting requests

**exploit** take advantage of a vulnerability

**hacking** unauthorised access to a computer or network or the modification of programs, systems or security for unapproved purposes

**honeynet** multiple honeypots across a network

**honeypot** a decoy server that imitates the 'real' website to lure attackers away from production systems to waste time and dissipate their energies harmlessly

**hot site** prepared external recovery IT infrastructure; where the duplicate is ready immediately to replace operations by restoring backups and resuming normal operations with little or no downtime

**impersonation** an attack that creates a fictional persona and then plays the role of that person in order to defraud or deceive

**keyloggers** record keystrokes from a computer and send passwords or sensitive information to unauthorised attackers

**malware** 'malicious software' designed to infiltrate and damage computer systems without authorisation

**man-in-the-middle (MITM) attack** a type of eavesdropping attack where communications and data are exposed to an unauthorised third party

**pharming** redirecting users to false websites that imitate the legitimate URL

**phishing** pretending to be a reputable person or entity in order to induce the disclosure of sensitive information

**piggyback entry** see **tailgating**

**ransomware** malicious software that will lock or encrypt a user's computer until a ransom fee is paid

**risk appetite** see **risk tolerance**

**risk tolerance** the quantity and nature of risk that is acceptable

**shoulder surfing** when an attacker observes password entry or security codes on a keypad, sometimes with a camera

**smishing** using SMS for phishing attacks

**social engineering attack** tricking the victim into clicking 'accept' with admin permissions or into giving the attacker physical access to a device

**spam** unsolicited bulk messages and advertising

**spyware** a program that collects information without the users' knowledge; it arrives as a free download and is automatically installed

**tailgating** when an authorised person swipes their entry card and enters, and an unauthorised person quickly enters behind them before the door closes

**threat modelling principles** a set of principles that outline how to prepare for dealing with cyber security threats

**Trojan horse** a program that appears to be safe and reliable but creates a back door into the computer

**two-factor recovery** using a recovery email and a phone or SMS message in order to verify or re-establish the identity of the user

**version control** records each stage of the development so that it is possible to 'roll back' to any previous point in the development

**virus** often arrives as an email attachment or download, and infects the active host computer and those on any contact list with spam and unwanted ads; hijacks the web browser and disables security access settings

**warm site** a recovery plan that is in between hot and cold, and limited in scope and scale compared with normal operations; it relies on restored backups being brought online before systems and services can be made operational

**worm** a standalone self-replicating program that exploits operating system vulnerabilities

## Important facts

- 1 Data must be protected to keep in line with legal requirements.
- 2 Version control can help track changes and allow for reversion or splitting from master files.
- 3 Robust identity and access management helps improve security by ensuring only authorised individuals have access to resources and systems.
- 4 Encryption strategies recognise that data will ultimately be exposed, so it should be rendered unintelligible to anyone who is not authorised or accredited to access it.
- 5 Code review acts as a security control as examining source code can identify, and therefore eliminate, vulnerabilities before deployment.
- 6 Deploying software updates and patches is a critical component of software security.
- 7 Separating development environments from testing and production environments acts as a security control because each stage is isolated from the other.
- 8 Using APIs can create vulnerabilities and risks by increasing the potential for exploitation.
- 9 Malware can take advantage of weaknesses in a host computer or operating system.
- 10 Viruses are malicious code that reproduce themselves on a computer.
- 11 Worms are self-replicating malicious code that can spread across computer systems.
- 12 Trojan horses masquerade as, or hide in, useful software and require a user to activate installation.
- 13 Logic bombs lie dormant until a timer or conditional trigger goes off.
- 14 Unpatched software leaves known security flaws unaddressed, which attackers can exploit.
- 15 Poor identity and access management practices can create vulnerabilities and risks by allowing unauthorised users to access sensitive data and critical systems.
- 16 Man-in-the-middle attacks are a type of eavesdropping attack.
- 17 Insider threats come from people a person or business trusts, such as current employees, former employees, contractors, vendors or business partners.
- 18 Social engineering relies on the manipulation of human nature to persuade victims to provide personal or sensitive information.
- 19 Denial-of-service attacks involve overwhelming servers with access requests. The server slows operations and may stop altogether.
- 20 Software from third parties creates risk as users are reliant on the developer or distributor to provide necessary patches, improvements or modifications.
- 21 Ineffective code review practices allow security flaws and coding errors to go undetected.
- 22 Software auditing and testing strategies can help mitigate risk.
- 23 Threat modelling attempts to identify all possible threats and allow for planning to occur to prevent the worst possibilities from coming true.



## TEST YOUR KNOWLEDGE

### Security controls

- 1 Explain the importance of security controls in software development. Give an example as part of your explanation.
- 2 Discuss the role of version control systems and code repositories in managing software development projects.
- 3 What are some best practices for managing user identities and controlling access to sensitive data?
- 4 How does encryption help to protect data? Explain using an example.
- 5 How does code review help in identifying and mitigating potential security vulnerabilities?
- 6 What are the risks associated with not keeping software up to date, and how can regular updates help mitigate these risks?

### Vulnerabilities and risks

- 7 What security considerations should be taken into account when using APIs?
- 8 How do worms, Trojan horses and logic bombs differ in their methods of infection and damage to computer systems?
- 9 What risks are associated with using unpatched software and how can regular patching mitigate these risks?
- 10 What are the consequences of poor identity and access management practices on organisational security?
- 11 How do man-in-the-middle (MITM) attacks compromise communication security and what measures can be implemented to prevent them?
- 12 Describe two ways organisations can protect themselves from insider threats.
- 13 What techniques do social engineers use to manipulate individuals into divulging confidential information and how can such attacks be prevented?
- 14 What is the difference between DoS and DDoS attacks and how do they disrupt the availability of online services?
- 15 What are the potential risks of using third-party software and how can these risks be mitigated?
- 16 Why is it risky to combine development, testing and production environments, and what best practices should be followed to separate them?

### Software security

- 17 How do auditing and testing strategies contribute to identifying and mitigating vulnerabilities in software systems?
- 18 What is penetration testing, and how does it help organisations uncover security weaknesses before they can be exploited by malicious actors?



## Software and data vulnerabilities

- 19 What are the primary causes of data breaches and how can organisations implement effective measures to prevent them?

## Threat modelling

- 20 How does threat modelling help in identifying and mitigating potential security threats in a software system?
- 21 What are the key differences between threat detection and threat prevention?
- 22 Why is it important to define security requirements during the early stages of software development?
- 23 How do accidental, event-based and deliberate threats differ?
- 24 How do monitoring and logging practices support the detection and response to security incidents?
- 25 What is patch management and why is it critical for maintaining software security?
- 26 Why is ongoing security training important for employees? What key topics should be included in such training programs?
- 27 What are the functions of intrusion detection systems (IDS) and intrusion prevention systems (IPS) and how do they differ?
- 28 What is the role of security audits in ensuring compliance with security policies and standards?



## APPLY YOUR KNOWLEDGE

### Software development case study

HealthTech Solutions is a growing software development company specialising in healthcare management systems. The company is developing HealthConnect, a comprehensive mobile and web application designed to streamline patient management, appointment scheduling and medical record keeping for healthcare providers.

HealthConnect will collect and store sensitive patient information, including medical histories, appointment details and billing information. It will integrate with various healthcare systems and databases using APIs and will be accessible via both mobile devices and web browsers.

Patients will be able to schedule appointments, access their medical records and communicate securely with healthcare providers through the app. Healthcare providers will have tools for managing patient data, tracking appointments and processing payments. The application will use cloud storage for data management and employ machine learning algorithms to provide predictive analytics and personalised healthcare recommendations.

#### Current security controls and threats

HealthTech Solutions employs several security controls to protect sensitive patient data, but there are areas that require attention and improvement.

- Data is encrypted in transit using HTTPS.
- The company uses biometric access control for physical entry to secure areas.
- Third-party APIs are used for integration with various healthcare systems, facilitating essential functions and data exchange.
- The application utilises cloud storage for managing sensitive patient data.
- There is a focus on maintaining software and systems, although specific strategies are not detailed.
- Employee training on security best practices is provided once every two years to ensure that staff are aware of and can implement necessary security measures.

#### Questions

- 1 How effective is HealthTech Solutions' current data encryption strategy for the whole system?
- 2 What are the weaknesses in HealthTech Solutions' current access control mechanisms, and how might these impact the security of patient data?
- 3 Evaluate the potential vulnerabilities in HealthTech Solutions' use of third-party APIs. What specific measures should be considered to enhance API protection?
- 4 What best practices should HealthTech Solutions implement given it is using cloud storage for patient data?
- 5 What should HealthTech Solutions implement in terms of patch management strategies?
- 6 Assess the current employee training program at HealthTech Solutions. Is it sufficient? Why/why not?

## APPLY YOUR KNOWLEDGE



- 7 What key areas should be included in HealthTech Solutions' employee training to improve security awareness and reduce the risk of accidental breaches?
- 8 Given the sensitive nature of the data handled by HealthConnect, what additional authentication methods could be implemented to enhance user security?



## KEY KNOWLEDGE

After completing this chapter, you will be able to demonstrate knowledge of:

- KK4.2.1: goals and objectives of medium and large organisations
- KK4.2.2: advantages and disadvantages of developing software in-house or externally
- KK4.2.6: criteria for evaluating the security of software development practices within an organisation
- KK4.2.7: key legislation and industry frameworks that affect how organisations develop software and control the security and communication of data, including the:
  - » *Copyright Act 1968* (Cwlth)
  - » Essential Eight
  - » Information Security Manual (ISM) (Guidelines for Software Development: Development, testing and production environments; Secure software design and development; Application security testing)
  - » *Privacy Act 1988* (Cwlth) (APP 1, 6, 8, 9, 11)
  - » *Privacy and Data Protection Act 2014* (IPP 1, 2, 4, 5, 9)
- KK4.2.8: ethical issues that arise when developing software, including:
  - » ineffective security practices
  - » use of artificial intelligence during development
  - » intellectual property
  - » copyright issues
- KK4.2.9: mitigation measures to reduce or eliminate threats, vulnerabilities and risks within organisations and development environments
- KK4.2.10: strategies for improving the security of software development practices, including:
  - » onboarding/induction practices and developer training focused on secure development
  - » development of risk management plans.

Victorian Curriculum and Assessment Authority, Victorian Certificate of Education Applied Computing Study Design 2025–2028, VCAA, 2024, pp. 64–5

## FOR THE STUDENT

By the end of this chapter, you should be able to analyse an organisation's software development practices, evaluate the current security controls and threats, and provide recommendations to improve practices.

This chapter is based on Unit 4, Area of Study 2 and, together with Chapter 6, provides the key knowledge required to complete Unit 4, Outcome 2. At the end of Chapters 6 and 7, your performance will be assessed by responding to a teacher-provided case study.

The case study scenario will enable:

- an analysis of the organisation's software development practices
- an evaluation of the current security controls and threats
- recommendations to improve practices.

Task time allocated should be 100–120 minutes.

## Why develop software?

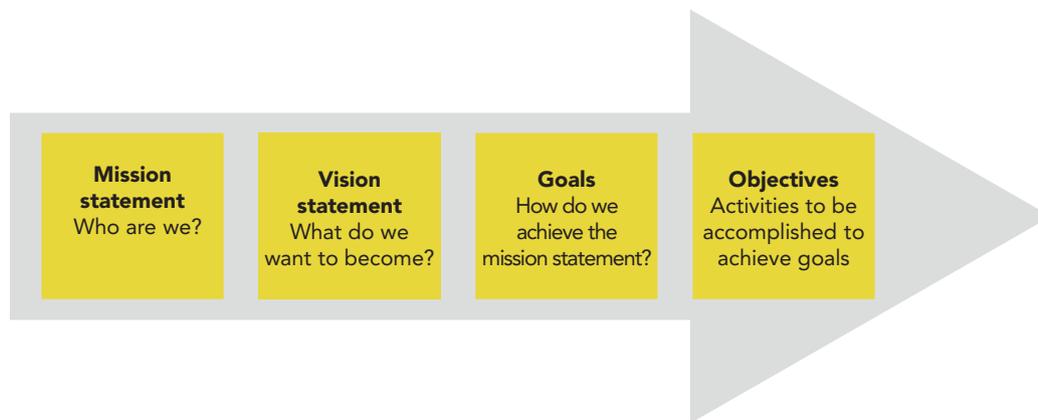
Sometimes, software needs to be written to specific criteria rather than being purchased off the shelf. Although open-source software is freely available, a developer is often needed to customise the program to a particular purpose. Sometimes software is needed to perform specific tasks that off-the-shelf software cannot do. This can affect both individuals and organisations. Sometimes the goals and objectives of organisations are not being met, and software needs to be developed to meet these.

### Goals and objectives of organisations

Organisations tend to go through many changes over time. These changes are generally the result of a strategic plan. A strategic plan is a process for identifying long-term goals within an organisation. For example, a school is an organisation. A school needs to establish a strategic plan outlining how it intends to maintain or increase enrolments, introduce new courses and perhaps erect a new building. This type of planning looks beyond the day-to-day running of the organisation and concentrates on future developments. These plans could range anywhere from two to 25 years.

Once an organisation has developed a strategic plan, a mission statement is developed based on the organisation's purpose, visions and values. The **mission statement** is the basis for establishing a set of common goals that will help accomplish the organisation's aims. These are known as **organisational goals**.

For example, a school may establish targeted goals, such as the introduction of a vocational education course within two years, or the completion of a new science and technology centre in the next five years. To achieve these goals, the organisation needs to develop a list of **objectives**. Objectives are small, achievable tasks that need to be undertaken in order to accomplish a big task. Objectives are measurable. For example, an objective may be to increase student enrolments by 10 per cent within 12 months.



**FIGURE 7.1** The relationship between mission statement, vision statement, goals and objectives

Organisations are increasingly dependent on the use of software to achieve their goals and objectives. When planning the software, the systems analyst will identify a **goal**. This explains the specific role of the software in achieving the organisational goal and, ultimately, the company's mission. Setting up the right type of software can help an organisation make improvements in efficiency, effectiveness and decision-making.

Identifying organisational goals is important to understanding how and why organisations operate.

#### THINK ABOUT SOFTWARE DEVELOPMENT

7.1

- 1 Can you think of any type of organisation that would need a strategic plan? For what purpose?
- 2 Using the internet, search for two examples of a mission statement.

A mission statement concentrates on the present, whereas a **vision statement** focuses on the future. A mission statement explains the company's reason for existence. It describes the company, what it does and its overall intention. A vision statement describes the organisation as it would appear in a future successful position.

Organisational goals and objectives will often relate to improving the efficiency or effectiveness of operations.

A systems analyst performs both the analysis and the design of the software. As well as being a good planner, a systems analyst must be a good communicator. As part of the analysis, the systems analyst is responsible for talking with users and translating their needs into a design, which is then passed on to programmers to build.

#### SCHOOL-ASSESSED TASK TRACKER

- |  |  |  |   |   |   |  |
|--|--|--|---|---|---|--|
| <input checked="" type="checkbox"/> Design brief | <input checked="" type="checkbox"/> Project plan | <input checked="" type="checkbox"/> Analysis | <input checked="" type="checkbox"/> Folio of alternative design ideas | <input checked="" type="checkbox"/> Development | <input checked="" type="checkbox"/> Evaluation and assessment | <input checked="" type="checkbox"/> Final submission |
|--|--|--|---|---|---|--|

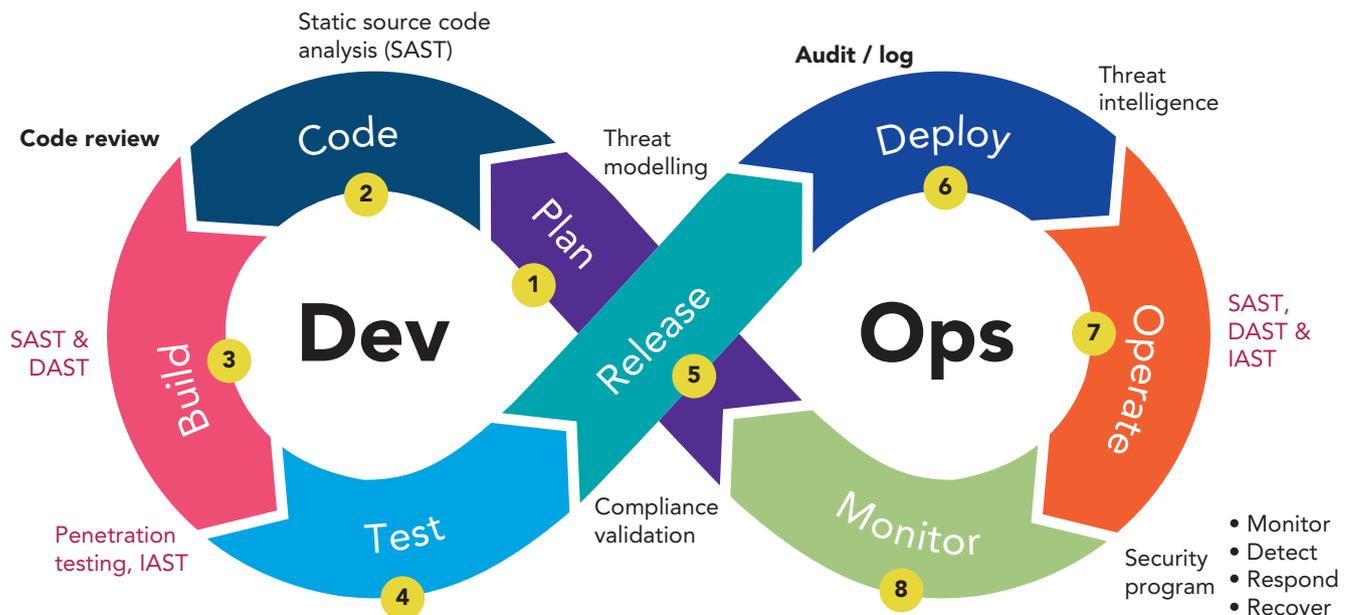
## Minimising risk

Threats to data and information occur every day. Data can be compromised through activities such as theft (such as by disgruntled workers or criminals who will make money selling the data), loss of devices (such as accidentally leaving laptops somewhere), neglect (not erasing data when recycling computer hardware) and not following appropriate data-handling procedures and policies. Collecting, storing, sending, encrypting, gathering and disposing of data all involve risk. To protect data and information, organisations need to manage this risk by putting security measures into place, such as:

- securing computers, servers and wireless networks
- utilising anti-virus and anti-spyware protection, and firewalls
- storing data backups off-site and ensuring that backups occur routinely
- securing passwords
- ensuring that staff are familiar with digital systems policies and procedures
- becoming familiar with their legal obligations.

## Development and operations

Often referred to as 'DevOps', development and operations describes the process of creating an application: coding, testing and implementing, then maintaining ongoing operation. Recently, a new term, 'DevSecOps', has been used to emphasise the integration of security into the planning, development and ongoing operation of the application. Development–security–operations has checks and validation throughout the entire life cycle. An additional step is the decommissioning and disposal of the application and data and information once the application is no longer in active service.



**FIGURE 7.2** DevSecOps has security embedded through the entire life cycle.

## Key components of DevSecOps

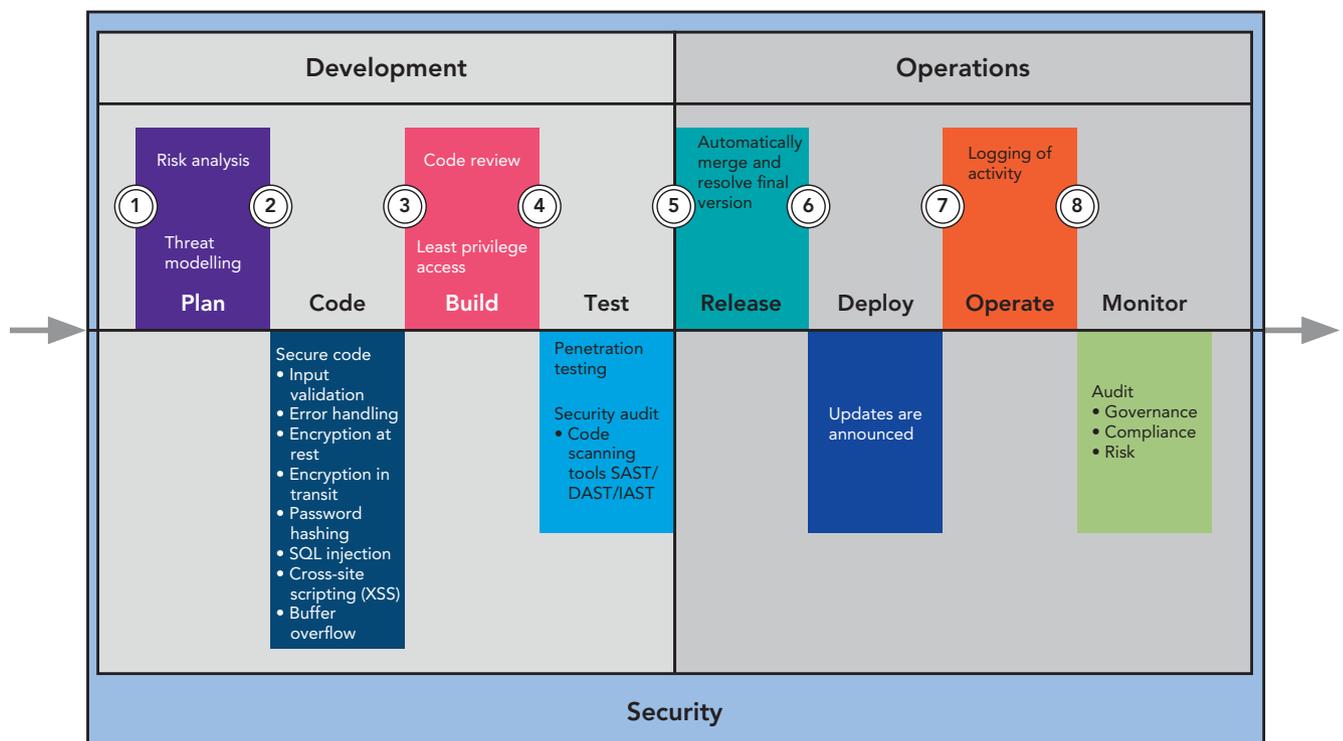
- Continuous security: Starting with the developer environment, all software undergoes regular security testing. The principle of least privilege (PoLP), also referred to as the principle of minimal privilege or the principle of least authority, is the minimum permissions required to accomplish a specific task. Higher-level privileges give access to high-value data and can bypass security features. Back doors can be seen as both convenient and highly risky.
- To mitigate the risks, it is crucial to monitor all activities to keep track of access and facilitate adjustments to privileges. If permissions are not used, they should be removed to maintain a secure environment.
- **Continuous integration (CI)**: A central repository holds all updates to resolve and merge versions for a single master certified against bugs and vulnerabilities.
- **Continuous delivery (CD)**: Shifts code from development to implementation. Code is thoroughly tested and integrated. APIs are verified before release.
- Collaboration: Communication between individuals and teams quickly addresses any coding issues to achieve agreed goals and objectives. Pair programming is where coding is done by two programmers who exchange technical and social skills. As code is created, it is checked against standards and expectations.



**Weblink**  
Microsoft: What is  
DevSecOps?

‘... most defects end up costing more to fix than it would have cost to prevent them’

Kent Beck, *Extreme Programming Explained*, Addison-Wesley Professional, 2nd edition, 2004



**FIGURE 7.3** Development is followed by operations and maintenance with integrated security at every stage. Despite being presented as a linear or waterfall process, there may be several iterations or repeated passes through each stage.

SCHOOL-ASSESSED TASK TRACKER

Design brief
  Project plan
  Analysis
  Folio of alternative design ideas
  Development
  Evaluation and assessment
  Final submission

## Components of security testing

- **Static application security testing (SAST)**: SAST is also known as ‘white-box’ testing. Applied during development stage to 100 per cent of the codebase, it may be automated to identify vulnerabilities. Test data needs to be comprehensive to avoid false positives.
- **Dynamic application security testing (DAST)**: DAST, or ‘black-box’ testing, is applied while software is operating during the testing and quality assurance stages of the PSM. Reports on compliance and security vulnerabilities across the whole application, including authentication, memory resource issues, permissions, API vulnerabilities and resilience of encryption algorithms. Many false positives are generated at this stage that take time to dismiss or to fix.
- **Interactive application security testing (IAST)**: IAST is a combination of SAST and DAST to evaluate specific aspects of the application. IAST is run from inside the application server and evaluates code similar to SAST to dynamically identify issues during operation, like DAST. IAST is a real-time analysis tool and is compatible with continuous integration (CI) and continuous delivery (CD) processes.
- **Runtime application self-protection (RASP)**: RASP provides ongoing security checks and monitoring of all activity and can respond to cyber attacks by raising an alert or terminating the session. RASP is integrated with the application and is not dependent on network protections or connectivity.

## Minimising costs

Business objectives commonly revolve around minimising costs and maximising benefits. Business leaders always aim for a successful project. However, a project can be deemed a failure if it is abandoned, significantly altered, or withdrawn from the market shortly after its release.

A software project failure may be attributed to several factors, including the following.

- **Time delays**: Delays in project development can affect stakeholders awaiting the completed software. Additionally, the project environment may have changed significantly since the development began, making the software less relevant or viable.
- **Cost overruns**: When the project exceeds its budget, requiring additional financial commitments, it can lead to further delays or even suspension of the project.
- **Quality issues**: A rushed project may result in compromises that impact the effectiveness and quality of the software. Insufficient time for thorough testing and refinement can lead to a product that fails to meet expectations.
- **Market misfit**: If the final software does not meet user expectations or proves to be difficult to use, it may not be well received by reviewers and fail to be used by many.

A software project can be deemed a failure when a combination of any of these factors comes into play. For instance, limited time and staff resources may force a team to bypass comprehensive testing, including user acceptance testing. As a result, the state of the software and user experience may be negatively impacted, leading to a decline in overall quality. Sales and marketing efforts may reveal that consumers are not enthusiastic about the new software product.

Software development can be limited by the availability of skilled workers. Outsourcing is a strategy where specifications are developed in-house, and external agents are contracted to complete the task satisfactorily.

## Advantages of developing software externally

Advantages of developing software externally include the following.

- Up-to-date security tools: Access to the latest techniques for software development. Experienced and well-regarded specialists will have the latest up-to-date privacy, data security and risk management techniques.
- Continuous security: Secure development of software through the life cycle. Vulnerabilities are patched as they are found during development using agile methods.
- Responsibility: Software team is well skilled in secure development and takes responsibility for data breach prevention.
- Compliance with regulations: Outsourcing providers audit the development process for compliance and misconfigurations. Due diligence assessment provides an estimate of liabilities and degree of risk.

Often a contract may have a fixed time or a fixed cost. However, open-ended consulting contracts may last for many years and the final cost could increase substantially over the original agreement.

Cost and time ‘blow-outs’ are not the only risks associated with outsourcing software development.

## Disadvantages of developing software externally

Disadvantages of developing software externally include the following.

- Protect confidentiality: **Non-disclosure agreements (NDA)** will require any intellectual property details to be kept secret.
- Increased friction: Encrypt all files and communications outside the organisation. Any shared resources must be kept secure. Assume all documents will be seen by an unauthorised third party; encryption would render such documents useless. File transfer protocols, VPNs and encryption are all easily available means of preventing access.
- Security best practice is required: Unique passwords, at least two-factor authentication for every account, least privilege access and limited use of personal devices (e.g. smartphone).
- Programming best practices are enforced: There are established standards for cloud services, programming frameworks and risk management that must be followed, or exceptions documented so known vulnerabilities are acknowledged. Security by design is well known, just not well implemented.
- Operates within the law: Legislative requirements must be met. However, who is required to meet them? The outsourcing agreement must be clear who does what, and someone must check that it has been done.
- Data and access restrictions: The external developers need to be given ‘just enough’ access to systems and real data. It is strongly recommended that real data with personal details never be provided unless an exceptional situation has been accepted. Fictional test data can be used to satisfy any user acceptance of testing scenario.

## Essential Eight

A strategy recommended by the Australian Cyber Security Centre (ACSC) for all business activities advises adopting a risk-based approach. Each organisation establishes minimum expectations for preventative measures at a certain level of maturity to mitigate the impact of cyber security incidents. The **Essential Eight** is an industry framework designed to assist all businesses to become cyber secure and resistant to cyber attack. Additional security mitigation strategies can be considered from the **Information Security Manual**.



Weblinks  
Essential Eight

Strategies to mitigate cyber  
security incidents

A malicious actor refers to any individual or group that intentionally exploits vulnerabilities in computer systems, networks or software to carry out harmful activities.

### Essential Eight maturity levels

In this framework, there are four maturity levels, ranging from Level Zero to Level Three. Businesses are encouraged to assess their current level and identify the level they want to reach. The process of improving the self-assessed maturity level is at the discretion of the business. This is not an externally accredited assessment.

Level Zero indicates there are weaknesses.

Level One has a focus on prevention of access by 'malicious actors' and mitigation of simple exploits.

Level Two has prepared for weak authentication and phishing exploits by malicious actors intent on stealing or destroying data.

Level Three has in place preparation for organised malicious actors who could exploit unpatched software and systems, circumvent authentication controls and disguise their activity within the organisation. Read about Essential Eight via the weblink.



### CASE STUDY

#### AI assistance

##### Using EDGE with BING user prompt <essential eight>

BING output:

**Essential Eight** – a set of **cyber security** measures recommended by the Australian Cyber Security Centre (ACSC). These strategies are designed to protect internet-connected information technology networks from cyber threats. Implementation of the Essential Eight makes it significantly harder for adversaries to compromise their systems.

Here are the eight essential mitigation strategies:

- 1 **Application whitelisting:** Control which applications can run on your systems by allowing only approved executables.
- 2 **Patch applications:** Regularly update software applications to address known vulnerabilities.
- 3 **Configure Microsoft Office macro settings:** Adjust settings to prevent malicious macros from running.
- 4 **User application hardening:** Limit the functionality of web browsers and other applications to reduce attack surface.
- 5 **Restrict administrative privileges:** Limit the number of users with administrative access to critical systems.
- 6 **Patch operating systems:** Keep operating systems up to date with security patches.
- 7 **Multi-factor authentication:** Require additional verification beyond passwords for user authentication.
- 8 **Daily backups:** Regularly back up critical data to prevent loss after an incident.

These strategies serve as a baseline for cyber security defences, helping organisations safeguard their systems against common threats.

Source: Australian Signals Directorate



**FIGURE 7.4** The Australian Cyber Security Centre has published the Information Security Manual, which is regularly updated.

## The cyber security principles

The purpose of the cyber security principles is to provide strategic guidance on how an organisation can protect their systems and data from cyber threats. These cyber security principles are grouped into four key activities: govern, protect, detect and respond.

**Govern:** Identifying and managing security risks

- Security risks are identified, documented and managed before and during deployment.

**Protect:** Implementing controls to reduce security risks

- Only trusted users, systems, applications and code are permitted.
- Multiple systems establish trust.
- Access to systems is restricted.
- Data is encrypted, controlled and backed up.

**Detect:** Detecting and understanding cyber security events to identify cyber security incidents

- Event logs are collected and analysed.
- Cyber security events are analysed to identify cyber security incidents.

**Respond:** Responding to and recovering from cyber security incidents

- Incidents are reported internally and externally.
- Incidents are contained and recovery enacted.

Australian Cyber Security Centre, [www.cyber.gov.au/resources-business-and-government/essential-cyber-security/ism/cyber-security-principles](http://www.cyber.gov.au/resources-business-and-government/essential-cyber-security/ism/cyber-security-principles)

The full description of all principles can be viewed via the weblink.

You can download the Information Security Manual (ISM), which has several detailed sections.

Recommended criteria for software development environments include:

- segregated areas for development, testing and production to limit the spread of malicious code
- protection of software source
- secure-by-design and secure-by-default development principles
- use of memory-safe programming languages, such as C#, Go, Java, Ruby, Rust and Swift



**Weblinks**  
Cyber security  
principles

Guidelines for software  
development

SCHOOL-ASSESSED TASK TRACKER

Design brief
  Project plan
  Analysis
  Folio of alternative design ideas
  Development
  Evaluation and assessment
  Final submission

**What is memory-safe?**

There are known limitations with certain languages due to the way errors are handled.

A write or read error is not always generated when the index is exceeded (out-of-bounds errors, or OOB). When an array is deleted and memory released, that memory can be used by an adjacent address to extend that range (called use-after-free, or UAF).

Ninety per cent of Android-confirmed exploited vulnerabilities were memory-safe vulnerabilities.

**'Living off the land' (LOTL) exploit identified**

'Many organizations do not implement security best practice capabilities that support detection of living off the land (LOTL), so this technique continues to be effective with little to no investment in tooling by malicious cyber actors. This guidance provides several observed network defense weaknesses that make it difficult for IT administrators to distinguish malicious activity from legitimate behavior, even for those organizations with more mature cyber postures.'

Cybersecurity & Infrastructure Security Agency, 'Identifying and Mitigating Living Off the Land Techniques', 7 February 2024, <https://www.cisa.gov/resources-tools/resources/identifying-and-mitigating-living-land-techniques>



**Webblink**  
Living off the land

- vulnerability disclosure program, which allows for the discovery and reporting by internal and external parties, public disclosure, and rewards (if any) for finders of vulnerabilities
- resolving vulnerabilities by providing patches or updates, or, failing to remedy, then providing advice on how to reduce the impact of vulnerabilities being exploited.

## Key legislation for storage and disposal of data and information

There are several key laws relating to the information systems and telecommunications industries. At a federal level, the *Copyright Act 1968* protects the rights of creators of creative and artistic works. Other key legislation includes the *Privacy Act 1988* (see Chapter 4), which governs how information about people can be used. In Victoria, we are especially concerned with the *Privacy and Data Protection Act 2014* (see Chapter 4) and the *Health Records Act 2001* (see Chapter 4). Combined, these laws govern the collection and use of private information by both government and non-government organisations, at both state and federal levels. Employers and government agencies have a legal responsibility to ensure that these laws are implemented within their organisations. Organisations must also make employees and customers aware of their rights and their responsibilities in relation to these laws.

This section examines the key laws affecting the storage and disposal of data and information held by organisations.

### Copyright Act 1968

The *Copyright Act 1968* is a federal law that recognises that any original creative or artistic work is the property of the person who created it. Any person wishing to use another person's work must obtain permission and/or pay for a licence. The Copyright Act protects the creator of an original work from unauthorised reproduction, conversion, adaptation, transmission or publication of their intellectual property (IP), which includes:

- original literary, dramatic, musical and artistic works
- websites
- software
- electronically recorded music, films and books.

The *Copyright Act 1968* was amended by the *Copyright Amendment (Digital Agenda) Act 2000*, the *Copyright Amendment Act 2006* and the Australia–United States Free Trade Agreement (AUSFTA).

**Copyright** protection exists automatically, as soon as intellectual property is created and recorded in a way that can be seen or heard – for example, written, recorded, filmed or put online. However, the Copyright Act does not cover ideas, concepts, styles, techniques, information, names, titles, slogans, people or images of people. You do not have to register for copyright as you do for patents or trademarks. You do not need to use a copyright symbol or statement, although they are recommended.

You have the right to protect your own original works using technological devices such as encryption or copy protection.

Without permission from the copyright holder, it is illegal to:

- digitise a non-digital work, such as ripping a CD to MP3, or converting a DVD to an MKV file
- make or import devices or software to bypass copy protection
- remove or tamper with a copyright notice
- share copyrighted material online
- keep or share programs recorded from TV
- publish unauthorised screenshots from some web pages or software.

In Australia, penalties for copyright violations are calculated based on a system of penalty units. A penalty unit is a set monetary value that is periodically adjusted. As of 1 July 2023, the value of a penalty unit is \$313.

Fines for copyright infringement are determined by multiplying the number of penalty units assigned to the specific offence by the current value of a penalty unit (rounded to the nearest dollar).

Fines under the Copyright Act are substantial, with a maximum of 850 penalty units for an individual. A corporation can be fined five times the amount of the maximum fine for an individual.

This means that an individual violating copyright could be fined up to  $850 \times \$313$ , or \$266 050 (rounded). For a corporation, this would total \$1 330 250.

Additionally, serious copyright infringements can also lead to criminal penalties, which may include further fines and/or imprisonment for up to five years, depending on the severity and commercial nature of the infringement.

These penalties aim to compensate copyright owners and deter future violations by making infringement costly for the violators.

It is important to note that employees who infringe copyright by pirating software on work computers are liable, but their employers can also be required to share some of that liability.

## THINK ABOUT SOFTWARE DEVELOPMENT

7.2

Why is this legislation contentious? Why is privacy protection important?

## INSIDE LAW: Dallas Buyers Club and Village cases explained

28 April 2016 by Sonia Borella

Two of Australia's top media and entertainment lawyers – Sonia Borella and Dan Pearce from national law firm Holding Redlich – explore two recent developments in combating piracy.

Rights holders have been active recently in taking steps to counter unauthorised downloading of their films. We catch up on the final outcome in the Dallas Buyer's Club claim and review the first claim under the recently introduced anti-piracy legislation.

Enforcing rights in copyright material has its obstacles but compensation is not unattainable – lessons from the Dallas Buyers Club case.

The owners of the film 'Dallas Buyers Club' have not proceeded with their claim for payments in connection with the unauthorised downloading of the program, but the action is still an important case for rights owners.

By way of background, in October 2014, a preliminary discovery application was filed in the Federal Court of Australia (Court) by the rights holders in the film Dallas Buyers Club (Film), naming six Australian ISPs as respondents.

The purpose of the application was to obtain the details of customers associated with 4726 IP addresses that were used to download and share copies of the Film using the peer-to-peer file-sharing service BitTorrent. The intention was to then issue letters of demand to those customers for the infringement of the copyright in the Film, seeking compensation.

## CASE STUDY



SCHOOL-ASSESSED TASK TRACKER

Design brief

Project plan

Analysis

Folio of alternative design ideas

Development

Evaluation and assessment

Final submission



On 7 April 2015, the Court delivered its decision on preliminary discovery in favour of the rights holders in the Film, and ordered that various ISPs, including members of the iiNet Group and Dodo, provide to Dallas Buyers Club LLC (or parent company Voltage Pictures LLC) the names and addresses of customers who are alleged to have used BitTorrent to share the Film.

But the order was conditional – the communications to be sent to alleged infringers needed to first be approved by the Court in order to prevent ‘speculative invoicing’ against alleged infringers without a proper legal basis.

Due to the nature of the preliminary proceedings, there was no decision as to whether copyright in the Film had been infringed, but the Court did ‘not regard as fanciful the proposition that end-users sharing movies on-line using BitTorrent are infringing the copyright in those movies’.

The ISPs claimed in opposing the discovery that naming individual users would be economically pointless, as the compensation for the infringement would be in the order of \$15.00. Representatives of Voltage Pictures LLC reportedly said that bringing proceedings against alleged infringers has value beyond the compensation recovered.

Further, in the case of multiple downloaders, the Court said it must be considered at least plausible that a copyright owner may be able to obtain aggravated damages under section 115(4) of the *Copyright Act 1968* (Cth). The ability to obtain aggravated damages exists partly because of the need to provide deterrence.

The rights holders in the Film proposed to demand of the alleged infringers a sum including:

- a** the cost of legitimately purchasing the Film
- b** a licensing fee relating to each infringer’s uploading activities
- c** punitive damages, based on the alleged infringers other illegal downloading activity, and
- d** a claim for damages, for the costs incurred by the rights holders in the Film in bringing the application.

On 14 August 2015 the Court handed down its decision, which imposed certain limitations on and requirements of the rights holders in the Film.

Of particular concern to the Court was whether the payment demands by the rights holders were reasonable in a copyright infringement context.

The Court found that it was reasonable to demand the sum equal to the cost of purchasing the Film, given that sharing the Film online was undoubtedly copyright infringement, and reasonable compensation would be equivalent to the cost for the alleged infringer to have seen the Film legitimately.

The Court also found that the rights holders in the Film had expended significant resources and legal costs in obtaining the infringing IP addresses, and therefore that the claim for damages was also reasonable.

The Court did not regard the demands for sums equal to a licensing fee and punitive damages as reasonable.

The Court concluded that it would only order the ISPs to hand over the customer details sought if the rights holders in the Film provided the Court with a written undertaking that they will restrict the demands to those which the Court has ruled as permissible.

The Court also ordered that this undertaking be secured by the lodging of a \$600 000 bond.

The rights holders in the Film then said the bond amount they would pay would only be in the amount of \$60 000 for access to 472 names initially, and that the costs sought would only be for an individual licence fee as well as damages for legal costs incurred.

The Court then gave the rights holders until February 2016 to appeal the Decision, after which time the proceedings would be terminated.

The drawn-out battle came to an anti-climactic end when the rights holders in the Film decided not to proceed any further, and failed to lodge an appeal within the time limit imposed by the Court.

The rights owners in the Film had to overcome several obstacles but fell at the final hurdle. They had the capacity to claim compensation for the permissible demands, including the purchase price of the Film and the costs expended in obtaining the IP addresses.

However, at their election, the matter did not progress any further, which begs the question as to whether, for copyright owners, these sorts of applications can be brought cost effectively.

This case has a lesson for infringers in putting them on notice and acting as a potential deterrent.

It also provides a roadmap of sorts for copyright owners as to the best way to approach these sorts of situations, including an indication as to the types of restrictive undertakings a court may impose to prevent things like speculative invoicing taking place.

It would seem that if copyright owners are prepared to act in accordance with any condition the court may impose, that their enforcing of their intellectual property rights and the process involved in holding infringers to account for online piracy is now more transparent following this decision.

### Australian Government passes Bill aimed at reducing instances of online piracy, but is legislation the answer?

In June 2015, the Australian Government passed the *Copyright Amendment (Online Infringement) Bill 2015 (Bill)*, the purpose of which is to prevent unauthorised access to certain websites, including the BitTorrent website, The Pirate Bay, and to ultimately reduce instances of copyright infringement.

For rights holders such as film and television production companies, the Bill provides a legal avenue to protect their rights, through an application being made to the Court for an order requiring ISPs to block access to websites.

Importantly, the website in question accessed by alleged infringers must have the principal purpose of copyright infringement.

While this may act as a deterrent for online piracy, it will not of itself prevent illegal downloaders from continuing to download from the websites in question. Further, in practice, the individual activity of the downloader will only be changed once the Court orders that the ISP take reasonable steps to block the website in question, following an initial application to the Court by the rights holder.

Although not necessarily a complete solution to the problem of online piracy in Australia, the Bill is a step in the right direction, and provides rights holders with a legal avenue available to protect their original copyright material.

Village Roadshow (Village) is amongst the first to utilise the newly introduced anti-piracy legislation, in recently filing an application with the Court for an order to be made to compel the ISP in question to block piracy website, SolarMovie. The basis of the claim is that this website's primary purpose is the facilitation of copyright infringement. Although these types of claims have been successful in other overseas jurisdictions, this is a test case for the Australian anti-piracy laws, and it will be interesting to see whether Village will be successful in holding the operators of the infringing website accountable for the damage they are causing to its industry.

– Dan Pearce

### Where to from here?

The combination of increased avenues for Australian consumers to access television and film content legally, together with the implementation of new deterrent legislation, and possibly more cases like Dallas Buyers Club tackling this issue, will hopefully go some way to reducing the number of instances of online piracy in Australia.

© Intermedia, IF Magazine. 'Inside law: Dallas Buyers Club and Village cases explained', 28 April 2016 by Sonia Borella. Accessed from <https://www.if.com.au/inside-law-dallas-buyers-club-and-village-cases-explained/>

SCHOOL-ASSESSED TASK TRACKER

Design brief
  Project plan
  Analysis
  Folio of alternative design ideas
  Development
  Evaluation and assessment
  Final submission

Since the AUSFTA was implemented on 1 January 2005, copyright generally applies for the life of the creator plus 70 years. The copyright holder may not necessarily be the author, performer or director if someone else (e.g. a recording company) paid for these works to be produced. Employers usually hold copyright over material that their employees create, as do film, game and music producers. Sometimes the performer may own a share in the copyright held by these organisations. Copyright can even be sold.

According to the Australian Copyright Council, ‘copyright is infringed if copyright material is used without permission, in one of the ways exclusively reserved to the copyright owner’. This means that someone may not use a whole or a part of a work, including changing or adding to it, without seeking permission from its copyright owners. For example, a student producing a video for their local sporting club must seek permission to use any music or video clips if they are not the student’s own original work. Similarly, someone who imports then sells copyrighted items from overseas without permission is considered to be in breach of copyright.

There have always been some provisions for ‘fair use’ of copyright material, such as by schools (for research or study purposes), libraries, reviewers (e.g. book and film critics) and government bodies.

When the *Copyright Amendment (Digital Agenda) Act 2000* came into effect in early 2001, the Copyright Act was updated to cover work published electronically. The main purpose of the amendment was to extend the Copyright Act to cover works that were produced, stored or transmitted digitally. This includes the use of web-based materials, digital sound and video recordings (including free-to-air broadcasts, CDs, DVDs and MP3s) and circumvention of technologically based copyright protection measures. These provisions were further extended by the AUSFTA in 2005.

In 2006, the Copyright Act was further updated to provide more direction for users, such as strengthening the owner’s rights to their digital material. In addition, it makes provision for users to access some legitimate copyright material without breaking the law, such as time-shift recordings, although users are not permitted to store their recordings.

New exemptions relating to personal use of recorded works have allowed consumers the right to make copies of works they have purchased and transfer them into other formats for personal use. This means that it is legal to copy music from CDs that you own into an MP3 format to be used on a personal music player. People are also able to transfer tapes and vinyl records to an electronic format. Consumers are also permitted to record television and radio programs to watch or listen to at a later time (time shifted). Again, this is only for personal use, and these recordings cannot be distributed to others. Enforcement measures include on-the-spot fines and proceeds of crime remedies.

In 2015, further amendments were made to the Copyright Act to incorporate online infringements. The Amendment to the Copyright Act was intended to disrupt large-scale websites that operate outside Australia and distribute (or facilitate the distribution of) infringing material to Australian consumers. It enables copyright owners to apply to the Federal Court of Australia (the Federal Court) to block access to an online location that meets certain conditions.

Making and distributing copies of games, music and software, even if it is not for any personal financial gain, is illegal in Australia. These acts are commonly referred to as **piracy**.

Further information on Australian copyright laws, including information sheets on a wide variety of copyright-related questions, can be obtained from the Australian Copyright Council.



Weblink  
Australian  
Copyright Council

## THINK ABOUT SOFTWARE DEVELOPMENT

7.3

- 1 Explain what is meant by ‘fair use’.
- 2 What do you and your classmates see as ‘fair use’ for purchased music?
- 3 What are the arguments for and against a ‘fair use’ amendment to the Copyright Act 1968?

## Copyright and music, computer games and computer software

Copyright legislation means that what you can copy or reproduce in other formats is highly restricted. In terms of music, as noted previously, when you buy a CD you have the right to make a backup copy, or even to rip a copy to MP3 format to play on your personal music player. You are permitted to download music from the internet via peer-to-peer transfers, but only if you have the permission of the copyright holder. There may be specific terms and conditions for online music stores that allow you to make a certain number of copies for personal use, but these vary between distributors.

Computer games fall into several categories because they incorporate the program code, as well as audio and video works that may be licensed from other copyright holders. Generally, you are permitted to make a backup copy of the game itself, but not any of the artistic works that may also be on the media, such as video or audio, without seeking the permission of the relevant copyright holders. You may lend a legitimate copy of a game to someone to play, but it is illegal to play an infringing copy. Under the AUSFTA, any devices specifically designed to bypass copyright protection measures are considered to be illegal.

Computer software is treated as a 'literary work' under the Copyright Act. As with games, you are permitted to make a backup copy of the software only, but not of any associated video or audio works unless the licence allows this. You are not, however, permitted to bypass copy protection features in order to make your backup. If your original media are destroyed, the Copyright Act allows you to make another backup from the first backup. Naturally, it is illegal to make multiple backup copies and distribute them to other people. The specific software licence will tell you how many times you are allowed to have the program installed simultaneously.

### When is copyright infringed?

Copyright is infringed when a person other than the copyright owner uses a 'substantial part' of the material in any of the ways reserved for the copyright owner without their permission. For example, if software users make copies, install software on multiple devices or make software available for downloading, this can infringe copyright. A person may infringe copyright by importing, selling or otherwise commercially dealing an infringing copy of computer software.

Software developers can take measures to protect their software, such as copy protection, password access and other types of technically based restrictions. There are two main types of technology prevention measures: those that confine access to the material, and those that limit or prevent copying of the material.

There are also provisions in the Copyright Act that give copyright owners the right to take legal action against people who make, supply, distribute or import devices to circumvent technology prevention measures.

Software developers can also protect their work by embedding details about the material, the copyright owner and related data. For example, details can be embedded in the metadata of a sound file, or watermarking and other data embedded into an image or video file. If someone tries to remove this data, a case can be made that the copyright was intentionally infringed.

Software developers should also consider the type of licence they will release alongside their software. The use of the software can range from restrictive uses to **open source**. The type of licence will depend on whether it is a commercial venture or if the software developer is comfortable freely distributing the software.

Software can be distributed through various app stores. Each store will have its own terms and conditions. The software developer will need to agree to these before their software can be sold on that platform. The licence will extend to revenue, acceptable software content and function, and how many copies of the app can be installed at a particular time, how many devices an app can be installed on and similar restrictions. Conversely, it is important for software users to be aware of the terms and conditions that outline how the software can be used, as well as consumers' rights in relation to factors such as making backup copies, re-downloading, and installing the software on multiple devices.

### Copyright and cloud computing

Many copyright owners are using **cloud** computing services to deliver copyright material to users, such as iTunes, Spotify, Netflix and Stan. These subscriptions or pay-per-use services provide on-demand access to large libraries of properly licensed music, films, books and other content.



**FIGURE 7.5** Copyright material can be delivered to users using services such as Spotify, Netflix and Stan.

Conversely, individuals tend to engage with cloud computing services to store copyright material they have copied or 'ripped' themselves, such as music files copied from a CD. The advantages of storing these copies on remote servers is that it means content from multiple computers and devices, including mobile devices, can be accessed easily. The issue, however, is that the copyright holders of the material may object to this use.

### Privacy Act 1988

The *Privacy Act 1988* was introduced in Chapter 4 (see page 175). This chapter will investigate how the Privacy Act applies to the collection, storage and communication of data.

#### What is included in the Privacy Act?

The Privacy Act includes:

- 13 Australian Privacy Principles (APPs) that apply to the handling of personal information by most Australian and Norfolk Island government agencies and some private sector organisations
- credit reporting provisions that apply to the handling of credit-related personal information that credit providers are permitted to disclose to credit reporting bodies
- the collection, storage, use, disclosure, security and disposal of individuals' tax file numbers
- the handling of health information for health and medical research purposes in certain circumstances, where researchers are unable to seek individuals' consent
- an Information Commissioner who approves and registers enforceable APP codes that have been developed
- the option for a small business operator, who would otherwise not be subject to the Australian Privacy Principles (APPs), to opt in to being covered by the APPs.

## Who is covered under the Privacy Act?

The Privacy Act gives an individual control over the way their personal information is handled. The Privacy Act allows individuals to:

- know why their personal information is being collected, how it will be used and who it will be disclosed to
- have the option of not being identified, or of using a pseudonym in certain circumstances
- ask for access to their own personal information (including health information)
- discontinue receiving unwanted direct marketing
- ask for incorrect personal information to be corrected
- make a complaint about an entity covered by the Privacy Act, if personal information has been mishandled.

## Australian Privacy Principles

As part of the Privacy Act, the Australian Privacy Principles (APPs) were devised to set out the standards, rights and obligations for collecting, handling, holding, accessing, using, disclosing and correcting personal information.

The APPs generally apply to federal government agencies. They do not apply to local councils, or state or territory governments. Some states have their own privacy laws, such as Victoria's *Privacy and Data Protection Act 2014*.

The APPs oversee the handling of personal information by:

- Australian and Norfolk Island government agencies
- all private health service providers
- businesses that have an annual turnover of \$3 million or those that trade personal information.

**TABLE 7.1** The Australian Privacy Principles

|               |   |
|---------------|---|
| <b>*APP 1</b> | <b>Open and transparent management of personal information</b><br>Ensures that APP entities manage personal information in an open and transparent way. This includes having a clearly expressed and up-to-date APP privacy policy. |
| <b>APP 2</b>  | <b>Anonymity and pseudonymity</b><br>Requires APP entities to give individuals the option of not identifying themselves, or of using a pseudonym. Limited exceptions apply.   |
| <b>APP 3</b>  | <b>Collection of solicited personal information</b><br>Outlines when an APP entity can collect personal information that is solicited. It applies higher standards to the collection of sensitive information.                      |
| <b>APP 4</b>  | <b>Dealing with unsolicited personal information</b><br>Outlines how APP entities must deal with unsolicited personal information.  |
| <b>APP 5</b>  | <b>Notification of the collection of personal information</b><br>Outlines when and in what circumstances an APP entity that collects personal information must tell an individual about certain matters.                            |
| <b>*APP 6</b> | <b>Use or disclosure of personal information</b><br>Outlines the circumstances in which an APP entity may use or disclose personal information that it holds.   |
| <b>APP 7</b>  | <b>Direct marketing</b><br>An organisation may only use or disclose personal information for direct marketing purposes if certain conditions are met.   |

In Software Development, you will focus on APPs 1, 6, 8, 9 and 11.



SCHOOL-ASSESSED TASK TRACKER

Design brief
  Project plan
  Analysis
  Folio of alternative design ideas
  Development
  Evaluation and assessment
  Final submission


**TABLE 7.1** The Australian Privacy Principles (*continued*)

|                |  |
|----------------|--|
| <b>*APP 8</b>  | <b>Cross-border disclosure of personal information</b><br>Outlines the steps an APP entity must take to protect personal information before it is disclosed overseas.  |
| <b>*APP 9</b>  | <b>Adoption, use or disclosure of government related identifiers</b><br>Outlines the limited circumstances when an organisation may adopt a government related identifier of an individual as its own identifier, or use or disclose a government related identifier of an individual.   |
| <b>APP 10</b>  | <b>Quality of personal information</b><br>An APP entity must take reasonable steps to ensure the personal information it collects is accurate, up to date and complete. An entity must also take reasonable steps to ensure the personal information it uses or discloses is accurate, up to date, complete and relevant, having regard to the purpose of the use or disclosure. |
| <b>*APP 11</b> | <b>Security of personal information</b><br>An APP entity must take reasonable steps to protect personal information it holds from misuse, interference and loss, and from unauthorised access, modification or disclosure. An entity has obligations to destroy or de-identify personal information in certain circumstances.  |
| <b>APP 12</b>  | <b>Access to personal information</b><br>Outlines an APP entity's obligations when an individual requests to be given access to personal information held about them by the entity. This includes a requirement to provide access unless a specific exception applies.   |
| <b>APP 13</b>  | <b>Correction of personal information</b><br>Outlines an APP entity's obligations in relation to correcting the personal information it holds about individuals.   |

© Office of the Australian Information Commissioner – [www.oaic.gov.au](http://www.oaic.gov.au). Summary of the Australian Privacy Principles accessed from <https://www.oaic.gov.au/agencies-and-organisations/guides/app-quick-reference-tool>. Released under CC BY 3.0 AU, link to license: <https://creativecommons.org/licenses/by/3.0/au/deed.en>

## Credit reporting provisions

There have been changes to the credit-reporting provisions of the *Privacy Act 1988* and to the way credit-related personal information is collected. The Privacy Act also encompasses a code of practice for credit reporting. The credit-reporting provisions for consumer credit include the simplification of the language used in reports, and improved privacy protections. The process to lodge a complaint has also been simplified.

## Privacy and Data Protection Act 2014

The *Privacy and Data Protection Act 2014* (PDPA) was covered in Chapter 4, pages 178–9.

## Ethical issues

Although software provides numerous benefits, negative effects, both intended and unintended, can impinge upon people's rights. Therefore, those who design, control and use software must consider the real and potential negative effects of the software, and eliminate or lessen them as much as possible. Sometimes even this may not be enough to justify the proposed collection or creation of data. It is important to take into account legal objections and ethical considerations when creating or acquiring data. The purpose for collection needs to be clear. It also needs to be articulated in the participant information statements and the consent forms provided to the people from whom information will be sought.

**Ethics** refers to behaving in ways that are based on our morals and accepted standards. These standards may be common in a particular society or specific to a single organisation. They apply to questionable activities over and above any legal requirements. Ethics often provide us with a set of guidelines for appropriate behaviour. Choosing to ignore these guidelines is not necessarily a crime, but it can lead to being sacked by an employer or being shunned by society. For example, the impact of violent video games on children has long been debated. Some people have voiced their concerns that video game writers should not include animated violence in their games because it can have a negative impact on children. A system of classification exists for games, similar to television and film classifications, which outlines the content that is permissible within each of the classification categories, and what content is prohibited.

These examples demonstrate how ethics hinge on society's values and standards. Some people would argue that protecting children from possibly harmful video games is the right thing to do. Others would argue that it is more important to maintain freedom of expression. Often, questions of ethics become debates over which of the two arguments is more important. This kind of conflict can be problematic, especially when the consequences of action are open to debate or interpretation.



Source: Commonwealth of Australia

**FIGURE 7.6** Classification markings ensure that consumers and parents can make informed choices.

Ethics are the moral guidelines that govern, among other things, the use of data collection. Often ethical principles/guidelines have an accompanying law, but the ethical principle is usually broader, and the law applies only to certain circumstances or applications of the principle.

For example, it is ethical to obtain permission to publish photos of people on websites or in promotional material. It is important to state the purpose for taking the photo and how it is intended to be used. Ethically, it is wrong to use a photo for a different purpose from that for which it was originally collected. However, the copyright for any photo is owned by the photographer. Legally, there is an issue about images taken in publicly accessible locations and published on the internet. In Australia, the 'right to privacy' is often cited, but there is no specific law, just a conditional implied entitlement. This often means that a costly court case would be required to resolve the issue. Similarly, when using data-collection tools such as surveys, interviews and questionnaires, it is important to reassure participants that the data provided, within the limits of the law, will remain anonymous, and that their individual comments will not be able to be identified by others. This is not just to put participants' minds at rest, it is also to ensure that their privacy is in fact protected (and that non-participants in the larger group that the sample is supposed to represent are not put at unacceptable risk of suffering as a result of mistaken identification).

This may be of particular concern when deciding what must be removed to sufficiently de-identify data to protect all its potential users. It is also important when reporting personal information anonymously or using pseudonyms in a newspaper report, for example.

#### SCHOOL-ASSESSED TASK TRACKER

Design brief
  Project plan
  Analysis
  Folio of alternative design ideas
  Development
  Evaluation and assessment
  Final submission

## Ethical issues when developing software

Often outrage is expressed when certain behaviour is made public and the act is declared ‘wrong’. A common defence statement counters with, ‘I have done nothing illegal’. Both are correct.

Legal remedies are not always possible as the legislation does not provide clear guidance on what is and is not acceptable. The cost and time of getting an enforceable judgment may work against taking any legal action. The spread and speed of social media publicity, and the ability to cross legal jurisdictions render many previous era laws ineffectual and impotent.

Many of the *Privacy Act 1988* provisions are no longer effective in 2024. The changes in the digital environment have exposed many shortcomings, and individual protections are not addressed.

The *Privacy Act 1988* provides consumer protections for large companies, with exemptions for journalists and political parties. Individuals have an implied right to privacy, which is limited by the Copyright Act. For example, an individual does not own their image in a photograph; the photographer owns the copyright. When using AI tools to create images, the prompt author ‘owns’ the copyright of the image.

Ethical issues will arise during the development of software. Occasionally, the issue will be familiar and may be dealt with and resolved by reference to precedents. However, sometimes it may be a ‘new’ issue, or an old issue with a new emphasis.

### Ineffective security practices

Ineffective security practices in software development create ethical issues by exposing users to risks such as data breaches, identity theft and loss of privacy, which can lead to financial and emotional harm. Regardless of what is legal and what is not, developers have an ethical responsibility to protect users’ sensitive information and ensure the integrity and confidentiality of their data. Failing to implement robust security measures not only violates users’ trust but also can result in significant reputational consequences for a company, undermining the ethical principle of doing no harm and eroding public confidence in technology.

### Use of artificial intelligence

Using artificial intelligence during software development can create ethical issues by embedding biases present in training data into the software, leading to unfair or discriminatory outcomes. AI systems can inadvertently reinforce stereotypes or make decisions that lack transparency, accountability and fairness, affecting users’ rights and experiences.

Additionally, the use of AI raises concerns about job displacement, privacy and consent, as AI can process vast amounts of personal data without explicit user approval. Developers must address these ethical concerns by ensuring that AI systems are trained on diverse, representative data and by implementing mechanisms for transparency, fairness and user consent throughout the development process.



### CASE STUDY

#### Bias in recruitment algorithms

In 2018, Amazon discontinued an AI recruitment tool that exhibited bias against women. The algorithm, designed to streamline hiring, was trained on résumés submitted over a decade, predominantly from men, reflecting the male-dominated tech industry. Consequently, the algorithm favoured résumés resembling those of male applicants, downgrading those that included the word ‘women’ and penalising résumés with experiences and skills typically associated with female candidates, such as working in women’s advocacy groups.

The biased algorithm resulted in discrimination against women and the perpetuation of gender stereotypes, causing Amazon to potentially miss out on qualified talent.

This case underscores the need for ethical considerations in the development and deployment of AI systems, particularly in areas like recruitment where biases can significantly impact individuals' career opportunities.

### Takeaway points

- Ensuring that training data is diverse and representative of the desired candidate pool is essential to avoid reinforcing existing biases.
- Regular audits of algorithms for bias and discrimination are crucial. These audits should involve diverse stakeholders to provide a comprehensive evaluation.
- While automation can improve efficiency, human oversight remains vital to catch and correct biases that algorithms might introduce.

## Intellectual property

Intellectual property (IP) rights can cause ethical problems in software development by limiting access to important technologies and ideas, which can slow down creativity and progress. If IP rights are enforced too strictly, it can stop developers from working together and sharing ideas, leading to monopolies where only a few large companies control the software market. This makes it harder for smaller companies or individuals to compete and access useful technologies. Balancing the protection of inventors' rights with the need for open access and sharing is important to solve these ethical issues and support a fair and innovative technology environment.

## Copyright issues

Ethical concerns can arise from copyright issues, especially considering the limitations of the *Copyright Act 1968*. The Act protects original works such as literature, music and software, giving creators exclusive rights to their use and distribution; however, it does not cover ideas, procedures or facts, which can lead to ethical dilemmas. For example, if a developer uses an uncopyrighted idea from another source and creates new software using that idea, they might gain significant financial benefit while the original idea's creator receives no recognition or compensation. Additionally, strict enforcement of copyright can restrict access to educational and cultural materials, raising concerns about fair use, spread of knowledge and social equity. These gaps and rigidities in the Act can create ethical tensions between protecting creators' rights and ensuring public access to information and innovation.

## 'Safe' programs

Programmers are legally obliged to produce programs that work properly. If a user can link a problem or consequence resulting from the use of a program to the work of a programmer, then the programmer will be liable for the damage caused. Program licences often state that the producer of the program is not liable if the user has used the program in a way that was not intended. However, if the program had bugs that caused a problem, the programmer may be liable.

Ethically, computer programmers must ensure that no damage or harm can come from the use of the program. This means the programmer must take into account who the users are, and must take no shortcuts in testing the program. This is an important consideration, given that programmers can write programs for all types of occupations and workplaces, including nuclear power plants, chemical factories, medical facilities and financial institutions.

### THINK ABOUT SOFTWARE DEVELOPMENT

7.4

Conduct an online search for 'history's worst software bugs' to find more examples.

#### SCHOOL-ASSESSED TASK TRACKER

- Design brief
- Project plan
- Analysis
- Folio of alternative design ideas
- Development
- Evaluation and assessment
- Final submission

Programmers have an ethical obligation to design, build and test programs that will benefit the user. Programming computers to have a beneficial impact is not easy, and is not always a conscious developmental goal. Programmers are often bound by financial and time constraints. The obligations to produce the best program possible and to remove all bugs are important. There have been cases where bridges have fallen down and planes have crashed due to programming bugs such as faulty code or badly designed features.

## 'Bad', hidden and malicious software

There are several illegal or unethical features that programmers can design into programs. For example, they can program the software to:

- have a *back door* so that people can bypass security features to gain access to a system
- contain hidden functions that can monitor the use of a program or a computer
- gain access to data on a computer system
- make connections to the internet and report back to the author.

Some programs will install spyware or adware – these are usually attached to software installed via the internet. The user is generally unaware of the spyware or adware, as it is attached to a program or file the user wanted to download.

Writing viruses and other such programs can also be classified as unethical. Other situations that might be considered problematic include:

- websites being programmed to store 'cookies' on computers, which record the activities of the person who visits the site
- programming games with features that challenge normal behaviour and values, such as excessive violence and other features that are discriminatory to particular genders or races
- installing camera surveillance and email monitoring, which can raise some ethical problems, especially if the employees are not informed.



### CASE STUDY

#### Programming bugs with BIG effects

- 1 Multiple crashes involving Tesla vehicles using the Autopilot feature were linked to software issues and limitations in the vehicle's self-driving capabilities. Some of these crashes resulted in fatalities, raising concerns about the safety and reliability of autonomous driving technologies.
- 2 In the 1980s, a radiation therapy device malfunctioned and delivered lethal doses of radiation at some installations. Based upon an existing design, the Therac-25 was an 'improved' therapy system that could deliver two different kinds of radiation: either a low-power electron beam (beta particles) or X-rays. One 'improvement' was the replacement of the older Therac-20's electromechanical safety interlocks with more reliable software control. But the operating system had been written by a programmer with no formal training. Because of a subtle bug based on the speed of response of the equipment, a quick-fingered typist could accidentally trick the Therac-25 so the electron beam would fire in high-power mode but with the metal X-ray target out of position. At least five patients died and others were seriously injured as a result of this malfunction.
- 3 The first internet worm infected thousands of computers in less than a day by taking advantage of a buffer overflow. The faulty code was part of the input/output network library routine called `gets()` in the Unix operating system, which is designed to get

a line of text. Unfortunately, it has no limit to its input, and a large input allows the worm to take over any machine it can connect to by inputting a full program and then running it. Programmers responded by attempting to remove the `gets()` function from all working code.

- 4 A *Mariner 1* rocket deviated from its designated path on launch, and mission control had to destroy the rocket over the Atlantic Ocean. The investigation discovered that a formula written on paper in pencil had not been accurately copied into computer code, causing the computer to miscalculate the rocket's path. The bug had not been discovered during testing of the software.

## Mitigating threats, vulnerabilities and risks

Mitigation measures are actions or strategies that organisations and developers use to reduce or eliminate potential problems that could harm their systems or data. Think of them as protective steps to make sure that software and computer systems stay safe and work properly.

For example, if there is a risk that a computer virus could infect your system, a mitigation measure might be installing antivirus software. If there is a chance that someone could hack into your system, using strong passwords and setting up firewalls would be mitigation measures. In software development, this could also mean writing secure code to prevent hackers from exploiting weaknesses in the software. These measures help protect the organisation and ensure that the development environment is secure and reliable.

Mitigation of risks in development environments can be addressed by putting in place security controls that were outlined in Chapter 6.

Mitigation of risks within an organisation can be addressed through consideration of hardware, software, physical equipment, procedures and electronic measures. These are outlined in Table 7.2.

**TABLE 7.2** Mitigating threats, vulnerabilities and risks within organisations

|                     |   |   |
|---------------------|---|---|
| Hardware            | Physical devices with built-in security features to prevent unauthorised access | Firewalls, secure servers, routers                            |
| Software            | Security programs to detect and prevent threats                                 | Antivirus, anti-malware, intrusion detection (Chapter 6)      |
| Physical equipment  | Devices and systems to control physical access to hardware                      | Locked server rooms, security cameras, biometric scanners     |
| Procedures          | Policies and protocols to ensure best practices for security                    | Employee training, strong password policies, access protocols |
| Electronic measures | Techniques to protect data and ensure secure access                             | Encryption, multi-factor authentication (MFA)                 |

### Firewalls, secure servers and routers

→ Firewalls act as a barrier between internal networks and external threats by filtering incoming and outgoing traffic based on predetermined security rules. For example, a firewall can block unauthorised access from suspicious IP addresses or prevent specific types of data from leaving the network, thus protecting sensitive information.

- Secure servers store sensitive data in protected environments with advanced security measures. These can include encryption, regular security patches and strict access controls. For instance, a secure server might use SSL/TLS encryption to protect data during transmission and restrict access to authorised personnel only.
- Routers can be configured to prioritise secure connections and manage network traffic efficiently, helping to prevent attacks such as denial of service (DoS). For example, routers can be set up to allow only specific devices to connect to the network.

## Antivirus, anti-malware and intrusion detection

- Antivirus software scans for and removes malicious programs that could harm systems. For example, if a computer is infected with a virus, antivirus software can detect and quarantine or delete the virus to prevent further damage and spread.
- Anti-malware software protects against a broader range of malicious software, including spyware, adware and ransomware. For instance, it can detect and remove a spyware program that records user keystrokes, protecting sensitive information.
- Intrusion detection systems (IDS) monitor network traffic for suspicious activity and alert administrators to potential breaches. For example, an IDS can detect an unusual number of failed login attempts, indicating a possible brute-force attack, and alert the security team to take preventive action.

## Locked server rooms, security cameras and biometric scanners

- Locked server rooms prevent unauthorised physical access to critical hardware by requiring keys or access cards. For instance, only IT staff with clearance can enter a locked server room, reducing the risk of physical tampering or theft.
- Security cameras monitor and record activity in sensitive areas, deterring intrusions and providing evidence if a breach occurs. For example, cameras installed in a server room can capture footage of unauthorised access attempts, which can be reviewed and used for security purposes.
- Biometric scanners ensure that only authorised personnel can access sensitive areas by using unique biological traits like fingerprints or retinal scans. For instance, a biometric scanner at the entrance of a data centre can ensure that only staff with the appropriate clearance level can enter.

## Employee training, strong password policies and access protocols

- Employee training educates staff about recognising and responding to security threats such as phishing emails or suspicious downloads. For instance, training sessions can teach employees to identify and report phishing attempts, reducing the likelihood of successful attacks.
- Strong password policies require the use of complex, unique passwords, reducing the risk of unauthorised access. For example, policies might mandate passwords with a mix of letters, numbers and symbols, and regular password changes to enhance security.

→ Access protocols define who can access certain data and systems, ensuring that only qualified personnel can reach sensitive information. For instance, a protocol might restrict access to financial records to only the accounting department, while other employees are denied access.

## Encryption and multi-factor authentication

- Encryption protects data by converting it into a coded format that can only be read with the proper decryption key. For instance, encrypting emails ensures that even if they are intercepted, the content remains unreadable to unauthorised parties.
- Multi-factor authentication (MFA) requires multiple forms of verification before granting access, significantly reducing the likelihood of unauthorised entry. For instance, a user might need to enter a password and then confirm their identity with a code sent to their mobile phone, adding an extra layer of security.

## Security and access factors

In some organisations, sensitive data and information of a personal, financial, military, industrial, legal or governmental nature may be stored and processed. The need for security and control of access may play a large part in the design of a system, especially the communication of the information across a network. For example, special consideration may need to be focused on the encryption of data. There have been problems with the security of credit card numbers used for online shopping because the security arrangements have been inadequate. Banks have been very careful moving to online banking because they have not been fully satisfied with the security of the financial information flowing online between the bank and the customer.

Some companies hire skilled hackers to attempt to break into their systems in order to identify security problems. Better monitoring and security hardware and software may be required as a result.

Security of and access to data need to be protected, and this should be addressed at the design phase. One technique is to make some files ‘read only’, so that users can access them but cannot change the contents of the data. The access permissions required by various persons in the organisation need to be determined early, and should eventually be put under the control of a system manager or data administrator.

Other forms of protecting data include using:

- usernames and passwords to access different levels of the information system, and to keep unauthorised people from having access to data when there is no need for it
- encryption, which is the coding of data, so those who are unauthorised cannot read it
- biometric security methods, such as fingerprint and retina scanning
- keeping logs of activities – which files were used, when and how, and on which computer.

## In case of attack

The starting point for every organisation is to assume they will be attacked, and work backwards from there. Threat modelling establishes a worst-case scenario, or two. Going out of business due to being unable to function and recover, or being forced out of business with overwhelming debts and liabilities are situations to be avoided.



Weblink  
Australian Cyber  
Security Centre

The cyber security principles indicate what to do in the event of an attack, assuming the earlier steps were in place and have failed. The Australian Cyber Security Centre provides advice on dealing with threats. To find out more, visit the website.

**Cyber Security Principles**

Content complexity  
Moderate

First published: 01 Dec 2023  
Last updated: 01 Dec 2023

Content written for

- Large organisations & infrastructure
- Government

Attachments

- Cyber Security Principles (December 2023) 912KB .pdf

### The cyber security principles

#### Purpose of the cyber security principles

The purpose of the cyber security principles is to provide strategic guidance on how an organisation can protect their systems and data from cyber threats. These cyber security principles are grouped into four key activities: govern, protect, detect and respond.

- Govern:** Identifying and managing security risks.
- Protect:** Implementing controls to reduce security risks.
- Detect:** Detecting and understanding cyber security events to identify cyber security incidents.
- Respond:** Responding to and recovering from cyber security incidents.

**FIGURE 7.7** The cyber security principles provide guidance to prevent and respond to the inevitable attack.

Source: The Australian Government

## Threat responses are different from risk responses

‘What action to take?’ has a different emphasis from ‘What preparations are to be made?’

Once threats are identified, actions taken could include the following.

- Mitigate: Reduce the possibility for the threat to occur.
- Eliminate: Remove the feature that is causing the threat.
- Transfer: Cloud-based applications have transferred responsibility for account security to subscribers and now 80 per cent of breaches are ‘user-associated errors’.

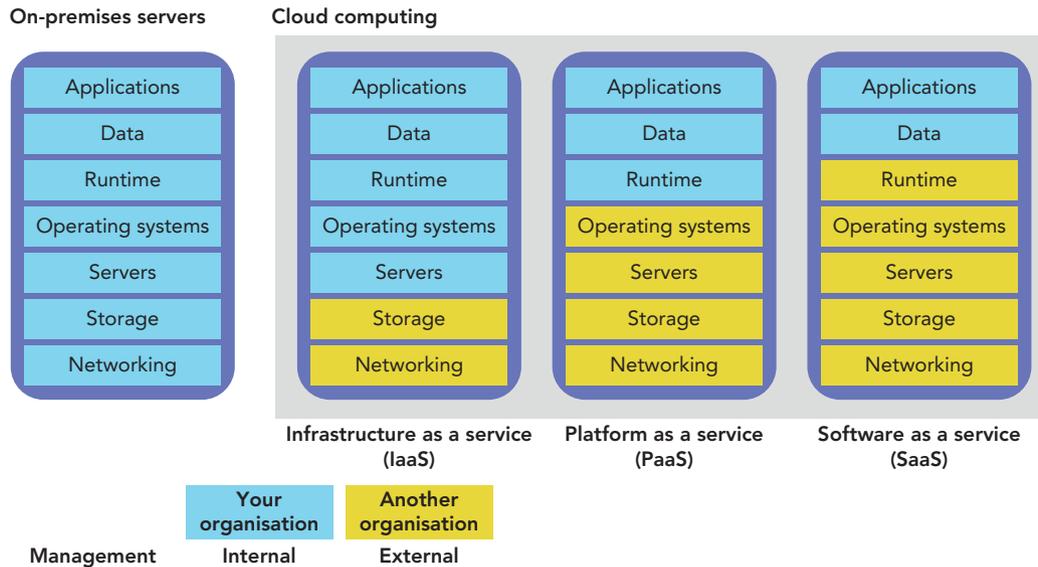
There are four ways to provide network services:

- in-house servers
- **infrastructure as a service (IaaS)**
- **platform as a service (PaaS)**
- **software as a service (SaaS).**

Each configuration utilises more cloud services and reduces reliance on hardware. However, many security protocols have not kept up with these developments, leading to confusion over the responsibility for security and configurations. Assumptions about responsibility and default settings are widespread.

The basic error is where the provider of services configures a default setting, and the application developer does not change those settings and publishes their production version with plainly accessible data.

Authentication processes that satisfy one protocol do not necessarily satisfy another. For example, application programming interfaces (APIs) designed to be used in SaaS were a responsibility of the cloud services provider. However, when used in the IaaS environment, the application security was inadequate, leaving gaps that were easy to exploit.



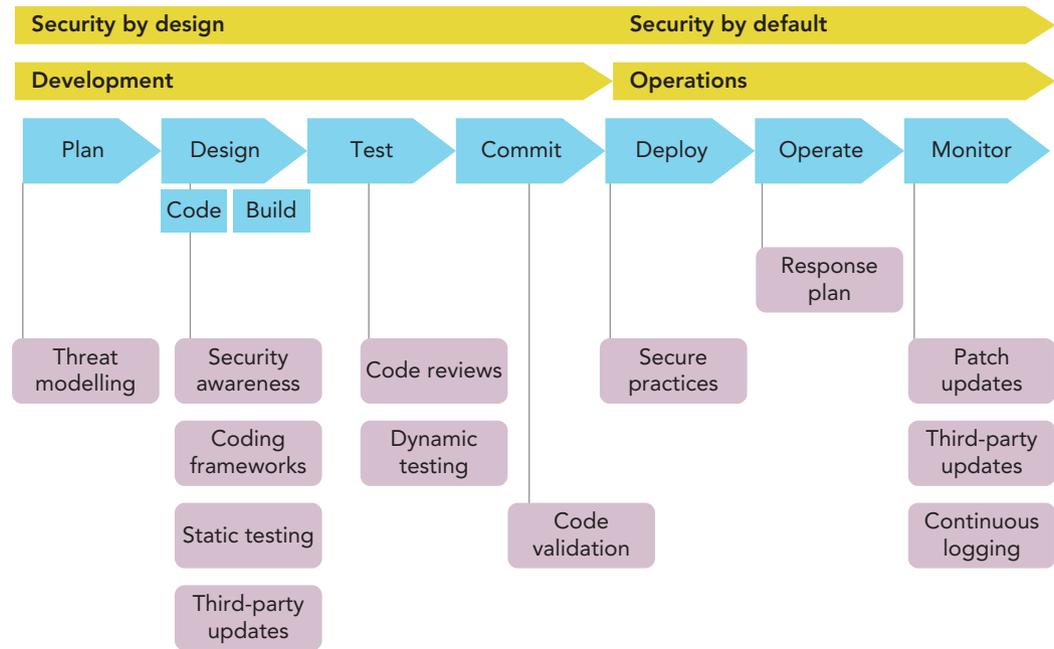
**FIGURE 7.8** Cloud computing configurations

### All risk can be reduced

All chief information security officers (CISOs) have a desire to eliminate risks; however, it is challenging as many situations only allow risk reduction due to the presence of unknown factors.

Industry frameworks, Essential Eight and the Information Security Manual offer clear guidance on measures to be put in place to establish a secure environment for development and production. Some general best practices are encouraged to be adopted and routinely put in place for all organisations.

- 1 Begin by assuming that ‘anything can happen’ and develop plans to prevent and recover from incidents.
- 2 Conduct regular risk assessments to prioritise potential threats and vulnerabilities.
- 3 Implement security controls, such as firewalls, intrusion detection systems, encryption, access controls (**MFA and 2FA**) and antivirus software, to mitigate known threats.
- 4 Continuously educate and reassure staff about phishing attempts, password management hygiene and safe internet habits to reduce human errors that could lead to security breaches.
- 5 Maintain effective software update management to install recommended patches.
- 6 Follow secure development practices throughout the software development life cycle, including secure code reviews, static (white box) and dynamic (black box) testing, penetration testing and secure version control.
- 7 Establish a disaster recovery plan (DRP) to restore operations after data loss or damage, with routine data backup as a crucial component.
- 8 Manage third-party risk by implementing contracted obligations and requirements, and conducting regular risk audits.
- 9 Develop an incident response plan that outlines the actions to be taken upon detection, including detection, response, recovery and review.
- 10 Continuously monitor ‘normal’ activity to quickly detect and respond to any unusual activity.



**FIGURE 7.9** Strategies for improving security during software development

During the process of software development, there are several strategies to improve security to ensure that vulnerabilities within the software are minimised and that the project is completed safely.

- 1 Security awareness of developers requires regular opportunities to reinforce coding practices, discover common vulnerabilities and emphasise the importance of security.
- 2 Coding frameworks provide guidance on acceptable standards for secure practice. Essential Eight maturity levels encourage continuous improvement.
- 3 Code reviews ensure that vulnerabilities are identified and neutralised early and often in development. Automated analysis can identify memory overflows, setting misconfigurations and injection errors.
- 4 Static (white box) analysis and dynamic (black box) analysis scan for vulnerabilities, and penetration testing identifies weaknesses in applications while running.
- 5 Threat modelling identifies potential security threats and vulnerabilities early in the development process and provides the opportunity to prioritise security controls that will mitigate risks.
- 6 Configuration management requires patches to be applied safely and consistently to minimise known vulnerabilities. Libraries and tools need to be updated regularly to minimise weaknesses as they become known.
- 7 Deployment processes need to be robust and resilient against potential attack. Least privilege access and secure network configurations ensure that only authorised entry is permitted.
- 8 Continuous monitoring of all activity allows security to be extended to the production environment. Real-time responses are possible to detected incidents. Activity logging allows post-incident analysis and improvement of security systems.

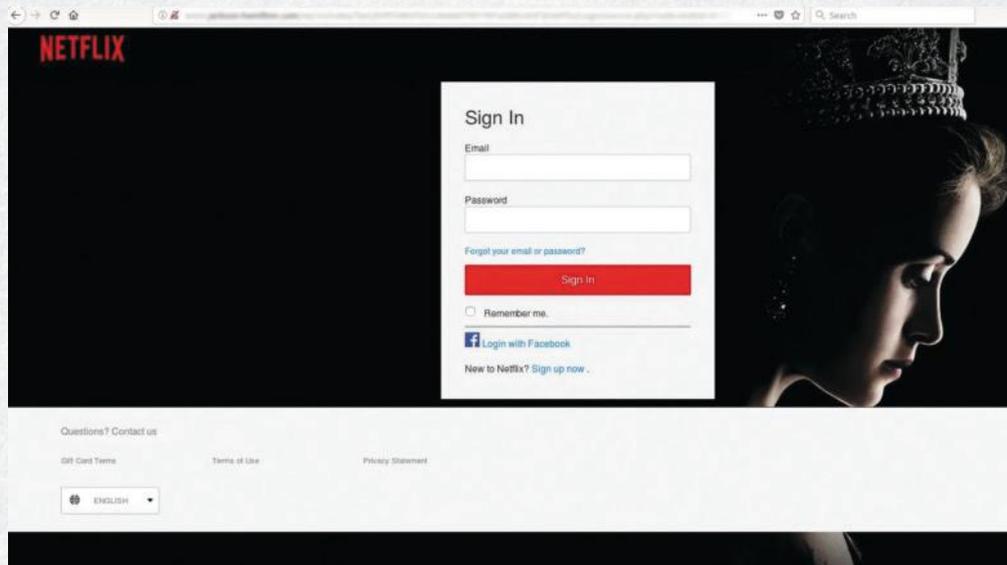
Organisations can strengthen software development security and reduce the impact of security breaches and other application vulnerabilities by implementing these strategies.

## Netflix customers urged to be vigilant as 'high quality' email scam circulates

By Jenny Noyes, 29 January 2019, Jenny Noyes /SMH

Customers of streaming service Netflix are being urged to exercise caution when checking and responding to emails from the company, as a scam email that looks almost identical to a legitimate customer service message circulates.

The email purports to come from Netflix – with the spelling 'NETFLIX' using a lower-case L instead of an I, and the subject line 'We've temporarily suspended your account until you verify your details.'



Source: Netflix, Inc

CASE  
STUDY



**FIGURE 7.10**  
Screenshot of a Netflix  
email phishing scam  
picked up in January  
2019

In the body of the email, the customer is informed their account has been 'suspended [sic]' due to 'issues in the automatic verification process' and warned that if they don't update their details their account will be deleted.

After the customer clicks on the 'update your details' button, they are redirected to a site that looks just like a real Netflix login page, where they can 'log in' and then enter their full credit card details.

The scam was picked up by mail security service MailGuard, which detects abnormalities in email messages and blocks fraudulent emails from reaching clients' inboxes.

People could easily go through the entire process without realising they have been swindled, said MailGuard CEO Craig McDonald, who described the scam as 'very well executed with high quality graphical elements in the email message and phishing page'.

'It's easy to imagine that it could potentially trick a lot of unsuspecting people.'

While the scam is well executed, Mr McDonald said it is not necessarily unusual – and it's not the first scam to target Netflix users in the past month.

A similar scam targeting Netflix customers with the claim their account had been 'suspended' due to issues with their billing information was detected by MailGuard's email filters in December.

The *Sydney Morning Herald* has approached Netflix for comment.



SCHOOL-ASSESSED TASK TRACKER

Design brief
  Project plan
  Analysis
  Folio of alternative design ideas
  Development
  Evaluation and assessment
  Final submission



Phishing scams are the most prevalent fraud reported to the ACCC's Scamwatch, which has received more than 20 reports of Netflix-related scam emails in January so far.

'We frequently detect similar email scams that are near-perfect clones of email templates from other reputable organisations,' Mr McDonald said.

'By targeting popular brands, recipients are more likely to have a relationship with the company being impersonated. That's an instant foot in the door.'

The increasing sophistication of such email scams requires a high level of vigilance from email users and organisations alike, he said.

'Prevention has now become critical for anyone dealing with cybercrime, and organisations must implement a multi-layered defence to help minimise the risks associated with cyber security attacks.'

As for individuals, the following red flags can help to identify and avoid being sucked in by a potential scam email:

- Generic greetings, such as 'Dear customer'
- A sense of urgency: 'Ensure your invoice is paid by the due date to avoid unnecessary fees'
- Bad grammar or misuse of punctuation, incorrect spelling and poor-quality or distorted graphics
- An instruction to click a link to perform an action
- Obscure sending addresses that don't match the real company's domain URL
- If in doubt, type the web address (URL) directly into your browser rather than clicking the link, or better still, phone the company
- If you've identified a scam email, report it to Scamwatch

## Improving the security of software development practices

Improving security in software development involves a range of best practices and tools to ensure that software is resistant to vulnerabilities and attacks.

- Code reviews are essential, as they involve regularly reviewing code to identify and fix security vulnerabilities. For example, a team might conduct peer reviews of new code to catch security flaws and vulnerabilities early.
- Static code analysis is another critical practice, where automated tools are used to analyse source code for security issues without executing the code. Tools like SonarQube or Checkmarx can scan the codebase for security issues, ensuring that potential vulnerabilities are addressed.
- Dynamic code analysis, which involves testing the running application to find security vulnerabilities that occur during execution, is also important. For instance, dynamic analysis tools like Detectify's OWASP ZAP can identify over 2000 security issues in websites during runtime.
- Penetration testing, which simulates cyber attacks to identify and address vulnerabilities, is vital. A cyber security firm might conduct penetration testing on an application to find and fix security weaknesses.
- Integrating security practices throughout the life cycle of the problem-solving methodology (PSM) ensures that security is considered at every stage of development.

SonarQube can be used to analyse source code for security issues.



Weblink  
SonarQube

Further to this, it is important to consider **onboarding and induction practices** and **developer training** as part of larger strategies to improve the security of software development practices. It is also important that **risk management plans** are put in place and followed when required.

## Onboarding and induction practices

Effective onboarding and induction practices help new employees quickly become productive and integrated into the company culture while understanding security protocols and expectations. Comprehensive orientation programs provide new hires with an overview of the company and its culture, policies and security protocols.

For example, a tech company might conduct a week-long orientation program where new employees learn about the company's goals, objectives, values and security policies, including data protection standards.

Role-specific training tailors induction training to the specific roles and responsibilities of new employees, emphasising security practices relevant to their duties. A software development firm might provide new developers with specialised training on secure coding practices, version control and tools used in their development environment, for example.

Mentorship programs are also beneficial, pairing new employees with experienced mentors who can guide them through their initial period and provide support and knowledge transfer. For instance, a company might assign each new hire a mentor who helps them understand their role, navigate the workplace and adhere to security protocols, improving integration and productivity.

## Developer training

It is important that developers participate in ongoing training and professional development to keep up to date on the latest security practices, technologies and threats. Regular workshops and training sessions focused on the latest security threats and best practices are important.

A tech company might organise monthly workshops on topics like secure coding, threat modelling and incident response, ensuring that developers stay current with evolving security threats.

Encouraging developers to pursue online courses and certifications in cyber security and secure software development is also beneficial. Developers at a company might be encouraged to complete courses on platforms like Coursera and obtain certifications like Certified Secure Software Lifecycle Professional (CSSLP).

Implementing security awareness programs raises awareness about security issues and promotes a security-first mindset among developers. For instance, a company might run a security awareness program that includes regular updates on security trends, phishing simulation exercises and recognition for developers who demonstrate strong security practices.

## Risk management plans

Risk management plans identify, assess and mitigate risks to software development and operations, ensuring continuity and security.



**Weblink**  
Certified Secure Software  
Lifecycle Professional

A systematic risk assessment involves:

- Identifying and evaluating risks to determine their potential impact and likelihood
  - » For example, a software company might conduct a risk assessment to identify potential threats to their new product, including cyber security threats, legal compliance issues and operational risks.
- Developing and implementing mitigation strategies to reduce or eliminate identified risks
  - » For example, a company might create a mitigation plan that includes regular software updates, employee training and improved access controls to address identified security risks.
- Preparing detailed incident response plans for responding to security incidents, with the goal to minimise damage and recover quickly
  - » For example, a financial services company might have an incident response plan that outlines steps to take in case of a data breach, including communication protocols, containment measures and recovery procedures.
- Regularly monitoring and reviewing the risk management plan to ensure currency as well as relevance
  - » For example, a company might check to ensure that new threats and vulnerabilities are considered, or make sure that relevant and current employees are aware of their roles and responsibilities in case of incidents.

As part of a risk management plan, risks are assessed against a probability matrix (Figure 7.11) as well as assigned a category (Table 7.3). The results of these assessments are then used to help identify priority areas for management.

|             |            | Impact        |       |          |       |              |
|-------------|------------|---------------|-------|----------|-------|--------------|
|             |            | Insignificant | Minor | Moderate | Major | Catastrophic |
| Probability | Frequent   | 5             | 6     | 7        | 8     | 9            |
|             | Probable   | 4             | 5     | 6        | 7     | 8            |
|             | Occasional | 3             | 4     | 5        | 6     | 7            |
|             | Possible   | 2             | 3     | 4        | 5     | 6            |
|             | Improbable | 1             | 2     | 3        | 4     | 5            |

**FIGURE 7.11** Risk assessment matrix

**TABLE 7.3** Risk assessment categories

| Risk category | Risk rating  | Status        | Action              |
|---------------|--------------|---------------|---------------------|
| 1             | Low          | Acceptable    | None                |
| 2             | Low          | Acceptable    | None                |
| 3             | Low          | Acceptable    | Routine audit       |
| 4             | Medium       | Tolerable     | As soon as possible |
| 5             | Medium       | Tolerable     | As soon as possible |
| 6             | High         | Manageable    | Priority            |
| 7             | Serious      | Manageable    | Priority            |
| 8             | Extreme      | To be avoided | Urgent priority     |
| 9             | Catastrophic | To be avoided | Immediate action    |

## CASE STUDY



## Risk management: Shane's project management application

Shane, the owner of a small software development company, is launching a new desktop application for project management. To ensure that the application is secure and reliable, Shane decides to implement a comprehensive risk management plan.

### Identify risks

Shane identifies the following risks:

- 1 Data breach: unauthorised access to user data stored in and by the application
- 2 Crashes: software crashes due to bugs or compatibility issues
- 3 Insider threats: malicious actions by employees or contractors
- 4 Malware infection: possibility of infection, spreading to users' systems.

### Evaluate risks

Shane then categorises and assesses each of the identified risks to rate their importance.

**TABLE 7.4** Risk evaluation

| Risk                | Probability | Impact       | Rating        |
|---------------------|-------------|--------------|---------------|
| Data breach         | Probable    | Catastrophic | 8: Extreme    |
| Application crashes | Occasional  | Major        | 6: Manageable |
| Insider threats     | Possible    | Moderate     | 4: Tolerable  |
| Malware infection   | Frequent    | Major        | 8: Extreme    |

Shane now needs to consider his priority actions.

**TABLE 7.5** Prioritise risks

| Risk                | Rating        | Action              |
|---------------------|---------------|---------------------|
| Data breach         | 8: Extreme    | Urgent priority     |
| Application crashes | 6: Manageable | Priority            |
| Insider threats     | 4: Tolerable  | As soon as possible |
| Malware infection   | 8: Extreme    | Urgent priority     |

### Develop and implement

Shane develops a list of strategies he can put in place to mitigate the identified risks.

**TABLE 7.6** Mitigation strategies

| Risk                | Strategy  |
|---------------------|---|
| Data breach         | Implement strong encryption for data storage; use multi-factor authentication for accessing sensitive data  |
| Application crashes | Conduct thorough alpha and beta testing; implement automated testing and bug tracking systems   |
| Insider threats     | Restrict access to sensitive parts of the codebase; conduct background checks of employees and contractors and monitor the application and related systems for unusual activity |
| Malware infection   | Use trusted antivirus and anti-malware tools; implement regular security updates and patches  |





## Create incident response plans (IRPs)

Each of the identified risks now needs an incident response plan that any person could use if an incident occurs.

### Data breach IRP

- Identification: A data breach has been detected and confirmed through monitoring tools.
- Containment: The IT team isolates affected systems to prevent further data loss.
- Eradication: Remove the cause of the breach (e.g. malware) and patch vulnerabilities.
- Recovery: Restore data from backups and resume normal operations.
- Notification: Inform affected users and relevant authorities about the breach.
- Review: Analyse the incident and update security measures to prevent future breaches.

### Application crashes IRP

- Identification: The support team receives reports of application crashes from users.
- Containment: Investigate and identify the cause of the crashes.
- Eradication: Developers fix the identified bugs and compatibility issues.
- Recovery: Release a software update/patch to address the crashes and restore normal functionality.
- Notification: Inform users about the issue and the availability of the update/patch.
- Review: Analyse the incident to improve testing and quality assurance processes.

### Insider threats IRP

- Identification: Suspicious activities indicating an insider threat have been detected and confirmed through monitoring tools.
- Containment: The IT team restricts the insider's access to sensitive systems and isolates affected systems to prevent further unauthorised actions.
- Eradication: Investigate and address the root cause of the insider threat, including removing any malicious actions performed by the insider and reinforcing security measures.
- Recovery: Restore affected systems and data from backups if necessary, and resume normal operations with enhanced monitoring.
- Notification: Inform senior management, relevant employees and, if applicable, external authorities about the insider threat incident.
- Review: Analyse the incident to identify lessons learned and update security measures and protocols to prevent future insider threats.

### Malware infection IRP

- Identification: Detection of malware infection through antivirus alerts or user reports.
- Containment: Isolate affected systems and prevent the malware from spreading.
- Eradication: Use anti-malware tools to remove the infection and clean the system.
- Recovery: Restore affected systems from clean backups and verify integrity.
- Notification: Inform users about the malware incident and provide guidance on protection.
- Review: Analyse the incident and enhance security measures to prevent future infections.

## Monitor and review

Shane knows it is important to regularly monitor and review the risk management plan. These are the steps he has put in place to do so:

- Regular audits: Shane ensures periodic security and compliance audits to assess the effectiveness of mitigation strategies. He prioritises his audit based around his priority actions.
- Update plan: The risk management plan is regularly updated based on new threats and vulnerabilities.
- Training: Continuous education for employees on security best practices and emerging threats is maintained.

# Evaluating the security of software development practices

Using criteria for evaluating the security of software development practices within an organisation is essential for several reasons. Firstly, it ensures consistency and standardisation. Clear criteria provide a consistent and repeatable approach to evaluating security practices across different projects and teams within the organisation. This consistency ensures that all aspects of security are assessed uniformly, reducing the risk of oversight. For example, a standardised criterion for code review ensures that every team follows the same rigorous process, regardless of the specific project they are working on.

Specific criteria also help to identify potential security risks and vulnerabilities that can occur during software development. By systematically evaluating practices against these criteria, organisations can proactively address and mitigate risks before they lead to security incidents. This proactive approach not only enhances the overall security of software development practices but also prevents costly breaches and data losses.

Criteria-based evaluations provide clear benchmarks for compliance with industry standards and regulations, ensuring that the organisation meets its legal and ethical obligations. This can be particularly important for organisations handling sensitive data, where compliance is critical to maintaining trust and avoiding legal penalties.

Using criteria for security evaluations supports continuous improvement. By regularly reviewing and updating the criteria to reflect emerging threats and best practices, organisations can ensure that their security measures remain effective and up to date. This iterative process helps in fostering a culture of security awareness and vigilance, ultimately leading to more robust and resilient software development practices.

When writing criteria, it is important to consider what it is that the criterion will help measure, to determine the data that needs to be collected and then to explain how that data will be used to measure against the criterion.

Table 7.7 provides a sample criterion for each of the software development practices that were covered in Chapter 6:

- version control and code repositories
- robust identity and access management
- encryption
- code review
- regular updates and patches to software
- separated development, testing and production environments.

**TABLE 7.7** Sample criteria for software development practices

| Practice                              | Criterion  | Details  |
|---------------------------------------|--|--|
| Version control and code repositories | Are only authorised personnel allowed access to code repositories?               | <p><b>Description:</b> Ensure that only authorised personnel have access to code repositories.</p> <p><b>Data to collect:</b> Audit logs from the version control system (e.g. GitHub) showing access permissions and activities.</p> <p><b>Measurement:</b> Review audit logs to verify that access is limited to authorised personnel and that access permissions are appropriately assigned and managed.</p>                |
| Version control and code repositories | Are secure communication protocols used for accessing code repositories?         | <p><b>Description:</b> Use secure communication protocols (e.g. SSH, HTTPS) for accessing code repositories.</p> <p><b>Data to collect:</b> Configuration settings from the version control system confirming the use of secure protocols.</p> <p><b>Measurement:</b> Inspect configuration settings to ensure that secure protocols are in use and compare against best practices for secure communication.</p>               |
| Version control and code repositories | Are code commits cryptographically signed to verify their integrity?             | <p><b>Description:</b> Verify the integrity of code commits by using cryptographic signing of commits.</p> <p><b>Data to collect:</b> Logs and records showing the use of GPG or similar cryptographic tools for commit signing.</p> <p><b>Measurement:</b> Analyse logs to ensure that all code commits are cryptographically signed, confirming the integrity and authenticity of the code.</p>                              |
| Robust identity and access management | Is MFA implemented for accessing development tools and systems?                  | <p><b>Description:</b> Implement MFA for accessing development tools and systems.</p> <p><b>Data to collect:</b> Audit logs confirming MFA implementation for all users.</p> <p><b>Measurement:</b> Verify through audit logs that MFA is enabled and enforced for all user accounts, ensuring an additional layer of security.</p>  |
| Robust identity and access management | Are access permissions based on roles and responsibilities?                      | <p><b>Description:</b> Ensure that access to resources is granted based on roles and responsibilities, following the principle of least privilege.</p> <p><b>Data to collect:</b> Access control lists and permission settings documentation.</p> <p><b>Measurement:</b> Review access control lists to ensure that permissions are assigned based on roles and responsibilities and that unnecessary access is minimised.</p> |
| Robust identity and access management | Are user activities monitored and logged to detect unauthorised access attempts? | <p><b>Description:</b> Monitor and log user activities to detect unauthorised access attempts.</p> <p><b>Data to collect:</b> Security logs and reports from user activity monitoring systems.</p> <p><b>Measurement:</b> Analyse security logs to identify and respond to any suspicious activities or unauthorised access attempts promptly.</p>   |
| Encryption                            | Is sensitive data stored in databases and file systems encrypted?                | <p><b>Description:</b> Encrypt sensitive data stored in databases and file systems.</p> <p><b>Data to collect:</b> Configuration files and system documentation showing encryption settings for stored data.</p> <p><b>Measurement:</b> Review configuration files to confirm that encryption is enabled for all sensitive data that is stored, ensuring data protection.</p>  |
| Encryption                            | Is data transmitted over networks encrypted using secure protocols?              | <p><b>Description:</b> Encrypt data transmitted over networks using secure protocols such as TLS.</p> <p><b>Data to collect:</b> Network configuration and logs showing the use of encrypted communication protocols.</p> <p><b>Measurement:</b> Inspect network configuration and logs to verify that data in transit is encrypted, protecting it from interception.</p>  |
| Encryption                            | Are secure key management practices implemented for encryption keys?             | <p><b>Description:</b> Implement secure key management practices for encryption keys.</p> <p><b>Data to collect:</b> Documentation and records of key management procedures, including key generation, distribution and rotation.</p> <p><b>Measurement:</b> Evaluate key management documentation to ensure that secure practices are followed, maintaining the integrity and security of encryption keys.</p>                |
| Code review                           | Does all code undergo peer review before merging into the main codebase?         | <p><b>Description:</b> Implement a peer review process for all code changes before merging into the main codebase.</p> <p><b>Data to collect:</b> Review logs showing peer approval for each code merge.</p> <p><b>Measurement:</b> Examine review logs to ensure that all code changes undergo peer review, enhancing code quality and security.</p>  |

**TABLE 7.7** Sample criteria for software development practices (continued)

| Practice   | Criterion   | Details   |
|--|---|---|
| Code review  | Are automated tools used for static code analysis to identify security vulnerabilities?       | <b>Description:</b> Use automated tools to perform static code analysis to identify potential security vulnerabilities.<br><b>Data to collect:</b> Reports from static analysis tools detailing identified issues and their resolution.<br><b>Measurement:</b> Assess static analysis reports to verify that vulnerabilities are identified and addressed, improving code security.   |
| Code review  | Are dedicated security reviews conducted for critical code sections?                          | <b>Description:</b> Conduct dedicated security reviews for critical code sections, focusing on potential vulnerabilities and threats.<br><b>Data to collect:</b> Records of security reviews and any actions taken to address identified issues.<br><b>Measurement:</b> Review security-specific review records to ensure thorough examination and mitigation of critical vulnerabilities.  |
| Regular updates and patches to software                    | Is there a documented policy for timely application of security patches?                      | <b>Description:</b> Implement a documented policy for timely application of security patches.<br><b>Data to collect:</b> Patch management policy document and records of patch application timelines.<br><b>Measurement:</b> Evaluate patch management documentation and timelines to ensure that patches are applied promptly, reducing security risks.  |
| Regular updates and patches to software                    | Are all software components regularly updated to address known vulnerabilities?               | <b>Description:</b> Ensure that all software components are regularly updated to address known vulnerabilities.<br><b>Data to collect:</b> Update logs showing the implementation of software updates.<br><b>Measurement:</b> Review update logs to confirm that all software components are kept up to date with the latest security patches.  |
| Regular updates and patches to software                    | Are regular vulnerability scans conducted on the software and infrastructure?                 | <b>Description:</b> Regularly scan the software and infrastructure for known vulnerabilities.<br><b>Data to collect:</b> Vulnerability scan reports and records of remediation actions taken.<br><b>Measurement:</b> Analyse vulnerability scan reports to ensure that identified issues are promptly addressed, maintaining a secure environment.  |
| Separated development, testing and production environments | Are development, testing and production environments physically or logically separated?       | <b>Description:</b> Ensure that development, testing and production environments are physically or logically separated.<br><b>Data to collect:</b> Network diagrams and access control policies showing environment segregation.<br><b>Measurement:</b> Inspect network diagrams and access control policies to verify that environments are appropriately segregated, preventing cross-contamination.                                  |
| Separated development, testing and production environments | Is access to production environments restricted to authorised personnel only?                 | <b>Description:</b> Restrict access to production environments to authorised personnel only.<br><b>Data to collect:</b> Access logs and access control lists showing restricted access to production environments.<br><b>Measurement:</b> Review access logs and control lists to confirm that only authorised personnel can access production environments, enhancing security.  |
| Separated development, testing and production environments | Are formal procedures followed for deploying code from development to testing and production? | <b>Description:</b> Implement formal procedures for deploying code from development to testing and finally to production.<br><b>Data to collect:</b> Deployment records and change management logs showing adherence to deployment procedures.<br><b>Measurement:</b> Evaluate deployment records and change management logs to ensure that code deployment follows formal procedures, reducing the risk of errors and vulnerabilities. |

# 7

## CHAPTER SUMMARY

### Essential terms

**cloud** internet-based data storage and/or processing

**continuous delivery (CD)** automatically testing code prior to production; manual approval deploys the final product

**continuous integration (CI)** frequently merging tested improvements to software in a shared repository

**copyright** the legal right of the creator of a work to determine how that work is used

**developer training** educational programs to enhance developers' skills and knowledge on programming, tools and security

**dynamic application security testing (DAST)** or 'black box' testing; testing for outputs with known inputs

**Essential Eight** an industry framework designed to assist all businesses to become cyber secure and resistant to cyber attack

**ethics** principles of right and wrong that are accepted by an individual or a social group; ethical behaviour often guides policymakers within organisations

**goal** an anticipated result or aim, which is specific, measurable, achievable and timely

**Information Security Manual** advice for businesses published by the Australian Cyber Security Centre

**infrastructure as a service (IaaS)** a fully hosted system with servers, storage and networking virtualised over the internet in a cloud computing model, typically on a pay-as-you-go basis

**interactive application security testing (IAST)** testing that monitors running applications with sensors to detect vulnerabilities in real time

**MFA and 2FA** multi-factor authentication and two-factor authentication; a password and a verification code sent via an alternative device or channel, such as SMS or email

**mission statement** statement setting out an organisation's purpose or what it is trying to achieve; the mission of most companies is to make a profit, while non-profit organisations tend to define their key mission as providing a service to their members

**non-disclosure agreement (NDA)** a contractual agreement restricting the ability to use or share information

**objectives** small, achievable tasks undertaken to accomplish a big task

**onboarding and induction practices** processes to integrate and train new employees in company culture and job responsibilities

**open source** source code that is available free and can be modified

**organisational goal** how an organisation intends to go about achieving its mission

**piracy** the illegal making and distributing of copies of games, music and software

**platform as a service (PaaS)** a fully supported operating system environment to run a virtual network

**risk management plans** strategies and processes for identifying, assessing and mitigating project or organisational risks

**runtime application self-protection (RASP)** an embedded security technology that detects and prevents attacks during application runtime

**software as a service (SaaS)** applications with all other support maintained by the cloud provider

**static application security testing (SAST)** or 'white box' testing; testing that analyses source, byte or binary code for vulnerabilities without executing the application

**vision statement** a statement describing the organisation as it would appear in a future successful position

## Important facts

- 1 Organisational goals are a set of common goals that help to accomplish an organisation's aims. They are typically long-term.
- 2 Organisational goals can be achieved through a variety of shorter-term objectives, which are small, achievable tasks.
- 3 Development and operations (DevOps) is the process of creating an application from start to finish – coding, testing, implementing and maintaining. It is also a process that emphasises the integration of security into the planning, development and ongoing operation of an application.
- 4 Components of security testing include static application security testing (SAST), dynamic application security testing (DAST), interactive application security testing (IAST) and runtime application self-protection (RASP).
- 5 Minimising costs and maximising benefits means taking into account, and therefore minimising, time delays, cost overruns, quality issues and marketing expectations.
- 6 Essential Eight is published by the Australian Cyber Security Centre to advise businesses on steps to mitigate cyber security threats. The approach is risk minimisation.
- 7 The Information Security Manual is also published by the ACSC and provides detailed advice for each maturity level on mitigating specific cyber threats and overcoming vulnerabilities that occur during the software development life cycle.
- 8 Security by design is an approach to software development in which security is designed into each stage of development and not outsourced to an external specialist expert.
- 9 Secure by default assumes all software has vulnerabilities, and to minimise harm the default promotes data security. Software runs with least necessary privileges.
- 10 The *Privacy Act 1988* (Cwlth) was amended by the Privacy Amendment (enhancing Privacy Protection) Bill in 2012. This came into effect in 2014. As part of this Act, the Australian Privacy Principles (APPs) replaced the National Privacy Principles and the Information Privacy Principles.
- 11 As part of the Privacy Act, the Australian Privacy Principles (APPs) were devised to set out the standards, rights and obligations for collecting, handling, holding, accessing, using, disclosing and correcting personal information. There are 13 APPs.
- 12 The *Privacy and Data Protection Act 2014* (PDPA) was introduced by the Victorian Government. It replaced the *Information Privacy Act 2000* and the *Commissioner for Law Enforcement Security Act 2005*.
- 13 Open source refers to programs where the source code is available free for use and can be modified. Open-source code is usually created as a collaborative and combined effort by many programmers who improve upon the code and share the changes with people in the open-source community.
- 14 The *Copyright Act 1968* (Cwlth) outlines the laws related to copyright. A breach of the Copyright Act could result in fines or imprisonment. Computer software is treated as a 'literary work' under the Copyright Act.
- 15 Most programmers adhere to copyright laws. Using code developed by other programmers without paying for it or without obtaining permission, or modifying it but still keeping the major features, can leave the programmer open to prosecution under copyright laws. This legal obligation also applies to the content of a program, such as images, sounds and text, and the interface design.
- 16 Copyright is infringed when a person uses or sells a program or a substantial part of a program without obtaining permission. Licensing agreements usually specify what the licensee can do with the program.
- 17 Program and information system developers have to ensure that they take into account a variety of legal obligations and ethical considerations. Some have an effect on the development of the program and others on the way the program is used. Even if there are no laws already in place, there can be other considerations that developers need to be aware of that could result in harm for users and others.
- 18 Mitigating threats involves putting in place firewalls, secure servers and routers. It also involves installing and keeping updated antivirus and anti-malware software and intrusion detection systems.
- 19 Physical measures that help to mitigate threats involve locked server rooms, security cameras and biometric scanners. Social measures involve employee training, strong password policies and consistent access protocols.

- 20** Encryption, multi-factor authentication and other security measures can also help to mitigate threats.
- 21** The security of software development practices can be improved by consistent onboarding and induction practices being implemented.
- 22** Developer training is an important element of increasing security, as developers can then be kept up to date on the latest security practices, technologies and threats.
- 23** Risk management plans are an important component in managing risk, as they help to identify, assess and mitigate risks to software development and operations, ensuring continuity and security.
- 24** Evaluating the security of software development practices involves using criteria-based measures to support continuous improvement.



## Goals and objectives of organisations and information systems

- 1 Explain the difference between an organisational goal and a mission statement.
- 2 What is a vision statement?
- 3 How are values important to the content of mission and vision statements?
- 4 Where does the purpose of mission and vision overlap?
- 5 Why are mission and vision statements important for organisational goals and objectives?
- 6 Explain why an organisation must comply with legal requirements.

## Legal requirements

- 7 How does the Essential Eight approach cyber security threats?
- 8 How does the Information Security Manual help in mitigating cyber threats?
- 9 What are the main purposes of the Australian Privacy Principles (APPs)?
- 10 What are the penalties for breaches of the *Privacy and Data Protection Act 2014*?
- 11 Identify a situation where downloading a file may be legal.
- 12 Name the Act outlining laws about copyright in Australia.
- 13 What does copyright protect and how can it be infringed?
- 14 What copying is allowed in the Copyright Act?
- 15 What constitutes copyright infringement in the context of software development?
- 16 Name the Act that outlines the laws relating to the storage of medical records in Victoria.
- 17 What legal obligations do programmers have concerning the content and interface design of their programs?



## TEST YOUR KNOWLEDGE

### Ethical issues

- 18** What are the legal and ethical consequences of releasing software with known security vulnerabilities?
- 19** Describe some of the legal and ethical implications for (i) the user, (ii) the supplying company and (iii) the financial institution paying for the transaction, when users reuse their passwords and credential stuffing results in unauthorised transactions taking place.
- 20** In a widely publicised incident involving well-known retailers in Melbourne in January 2024, why did the banks pay when several retailers supplied goods for several hundred transactions as a result of credential stuffing?
- 21** How can both of these comments be correct? ‘There is no right to privacy’ and ‘we are over-regulated on requirements for keeping personal information identifying details.’  
Who does the *Privacy Act 1988* apply to? Who is not covered? Which groups have exemptions?
- 22** ‘All AI has bias.’ It has been argued that the use of AI will be fairer than existing bias, which is well hidden and rarely explained. AI is based on data sets and can be instructed to ‘be fair’. Unstated discrimination is no longer acceptable.  
Find an example where the use of AI may be (i) an improvement and (ii) a worse situation compared to the current practice.



The Melbourne Robotics Centre (MRC) is an organisation that runs professional designing, building and programming classes specifically aimed at school-aged children. The company offers classes after school on most weeknights and on weekends in various locations around Melbourne. Children from many suburbs participate in this extracurricular activity. The MRC is a not-for-profit organisation set up specifically to broaden children's interest in science, technology and engineering, and has approximately five full-time staff and 30 casual staff. The MRC has a website where it advertises its classes and locations. Pictures taken of students during the classes are used in the company's advertising. The website and its associated back-end database is developed by a programmer who has modified open-source software for the MRC. The MRC collects vast amounts of data on the students, such as their date of birth, home address, medical conditions, living arrangements in relation to custody, and their parents' names and occupations. The MRC relies on all its staff to update the data and its website, even though staff are not aware of all the legislation.

- 1 Can you identify the goals and objectives of the organisation?
- 2 If you were to write an organisational goal for the Melbourne Robotics Centre (MRC), what would it be?
- 3 Why did this organisation need to develop software specific to its needs?
- 4 Why does the company need to store data on its clients?
- 5 What key legislation should the staff be aware of specifically related to collecting, storing and communicating data?
- 6 What does the MRC need to do to ensure that it is compliant with the *Privacy Act 1988*?
- 7 What measures are needed for the MRC to protect the integrity of data and information?
- 8 Can you identify the possible legal and ethical consequences for ineffective security practices?
- 9 Can you recommend a backup solution for the MRC?
- 10 How does the MRC protect itself from security threats?
- 11 Does the MRC require all the data it collects?
- 12 What data should be collected on the children and their parents?

## PREPARING FOR

# Unit

# 4

### OUTCOME 2

Respond to a teacher-provided case study to examine the current software development security strategies of an organisation, identify the risks and the consequences of ineffective strategies and recommend a risk management plan to improve current security practices.

To achieve this outcome, you will draw on knowledge and skills outlined in Unit 4, Area of Study 2 – Cyber security: secure software development practices. This area of study is covered in Chapters 6 and 7 of this textbook.

### Steps to follow

- 1 Read the teacher-provided case study.
- 2 Analyse and discuss the current security controls to protect software development practices.
- 3 Propose and apply criteria to evaluate the effectiveness of current software development security practices.
- 4 Identify and evaluate threats to the security of software development.
- 5 Identify and discuss possible legal and ethical consequences of ineffective security strategies.
- 6 Recommend a risk management plan and justify strategies to improve current software and data security practices.

### Documents required for assessment

Submit your responses to the provided case study to your teacher. This could be in the format of written responses to structured questions, a written report or a multimedia report. (This will be determined by the requirements provided by your teacher.)

### Assessment

This task is marked out of 100 and is worth 10 per cent of your study score. Your performance will be assessed using one of the following.

- A case study with structured questions
- A report in written format
- A report in multimedia format



## Unit 4 exam practice questions

For Unit 4, you need to ensure that you have a sound understanding of the required key knowledge as well as the ability to demonstrate the key skills. When revising, it is useful to relate the key knowledge to real-world applications to enhance comprehension. Real-world applications provide concrete examples of abstract concepts, making them more tangible and easier to understand – for example, explaining how ethics is important during the software development stage.

You need to be familiar with the difference between the law, covered by federal and state legislation, and ethics.

Thoroughly read and understand all questions, including the case study when provided. Underline command terms and key phrases in the question to help you understand the intention of the question, especially for theory and short-answer sections.

For multiple-choice questions, learn to use the process of elimination in questions where you are unsure of the answer. Often, you can narrow down the choices to two, increasing your chances of selecting the correct answer.

Short-answer questions require clear and concise responses. Focus on directly answering the question, avoiding unnecessary details. Demonstrate your knowledge by using appropriate technical terms. However, ensure that you fully understand the terms you use.

Practise managing your time effectively. Allocate time to each section based on its weight in the overall grade and your proficiency in that area. If time permits, review your answers, focusing on sections you found challenging. This can help catch and correct any mistakes or oversights.

### Multiple-choice questions

#### Question 1 (1 mark) KK4.1.3

In programming languages, what are used to store and manipulate data?

- A. Data types
- B. Classes and objects
- C. Functions and methods
- D. Instructions and control structures

#### Question 2 (1 mark) KK4.1.5

When validating data, the most effective test order would be:

- A. type, existence then range.
- B. range, existence then type.
- C. range, type then existence.
- D. existence, type then range.



**Question 3** (1 mark) *KK4.1.11*

What factor can significantly impact the effectiveness of project plans by introducing additional requirements beyond the original solution requirement specifications (SRS)?

- A. Scope creep
- B. Technical issues
- C. Budget constraints
- D. Personnel changes

**Question 4** (1 mark) *KK4.2.8*

Which one of the following best represents a legal ethical issue that may arise during the software development process? (Choose all that may apply.)

- A. Gathering customer data details for personal advantage
- B. Using customer images to populate your website
- C. Collecting sensitive data without the participants knowing what the data will be used for
- D. Translating code from a piece of software without the permission of the original developer

**Question 5** (1 mark) *KK4.2.10*

Which strategy aims to enhance the security of software development practices by providing comprehensive training to developers in secure development techniques?

- A. Code review processes
- B. Risk management plans
- C. Incident response procedures
- D. Onboarding/induction practices

## Short-answer questions

### Case study 1: Problem-solving methodology – development and evaluation

#### Software development life cycle

While a software solution is under development, there may be several teams involved in refining modules for the final working prototype.

**Question 1** (2 marks) *KK4.1.1*

Describe the characteristics of an efficient and effective solution.

**Question 2** (3 marks) *KK4.1.3*

Before development and coding of the software solution can commence, the choice of programming language (PL) must be made.

List three features that must be considered in making a choice of appropriate programming language.



**Question 3** (3 marks) *KK4.1.6*

Once the programming language has been chosen and different teams begin coding, there will inevitably be unintended consequences when various modules are assembled.

Describe how the different modules are modified and made to work together.

**Question 4** (3 marks) *KK4.1.10, KK4.1.11, KK4.1.12*

Once the software solution has been tested and accepted, the development manager may review how the project had progressed. Such a review would be to inform and assist planning for their next software development project.

Describe how a development manager might assess the effectiveness of a project plan.

**Case study 2: Cyber security software development**

A startup software company intends to develop an online platform.

A small business software development company creates software for a variety of clients. There are 10 developers contributing to a variety of projects for different clients.

The lead developer for the software solution disagrees with the managing director of the company. During a design meeting, her manager suggests that the data stored on the cloud will not need to be encrypted because the software solution will require users to log in using multi-factor authentication.

**Question 1** (2 marks) *KK4.2.4*

Explain why data protection and user authentication are both important during software development.

**Question 2** (2 marks) *KK4.2.5*

During the same design meeting, the issue of security was raised. The managing director put forward the view that the team of developers could cope with any issues as they occurred. Any developer was welcome to raise any issue at the weekly sprint meetings. The lead developer put forward an alternative view about identifying threats systematically rather than in an ad hoc, or as they happened, manner.

Using threat modelling principles, identify the steps required to safeguard the development of a software solution. Explain why a systematic threat modelling audit is more effective than an ad hoc approach.

**Question 3** (4 marks) *KK4.2.3*

Identify three vulnerabilities or risks that may occur within an insecure environment.

Provide one action that could prevent or reduce the impact of the risk or vulnerability being exploited.

**Question 4** (2 marks) *KK4.2.10*

Recommend a strategy for improvement of software development security practices.



# Index

- 32-bit computer systems 4
- 64-bit computer systems 4
- abstraction 44, 46
  - benefits of 47
- access control reviews 273
- AccessiBe website 213
- accessibility/accessibility testing 50, 157, 180, 220
- access management 314
- access modifiers 47, 205–6
- access protocols 302–3
- accidental threats 270
- accuracy (data) 157
- actors 109
- adapt (creative design) 148
- Adem's gym 90
- advanced machine learning models 27
- adware 300, 302
- AES encryption algorithm 167
- affordance 164–5
- AI algorithms 28
- AI-driven testing 28
- algebraic expressions 69
- algorithms
  - efficiency of 60–1
  - for searching 57–60
  - for sorting 51–7
- alpha testing 209–14
- alternative execution 34
- analysis phase (SRS) 107
- analytical tools, processes and systems 108–25
  - data flow diagrams (DFD) 114–25
  - use case diagrams (UCD) 108–12
- 'AND' operator 35
- annotated diagrams 161
- anti-malware software 302
- antivirus software 254, 269, 302
- Apache Software Foundation 242
- Apache Subversion (SVN), case study 242
- appender virus 246
- application programming interfaces (APIs) 206, 207, 246, 304
- arguments 42–4
- arithmetic operators 32, 204
- array 5–7
- artificial intelligence (AI) 298–9
  - future of programming with 30
  - and human intelligence 30
  - responsible and ethical use 30
- artificial intelligence (AI)-based assistants 206
- assistive technology 141
- association (relationship) 109
- asymmetric key encryption 167
- attractiveness (interface) 157
- Australian Copyright Council 292
- Australian copyright laws 292
- Australian Cyber Security Centre (ACSC) 243, 268, 287, 304
- Australian Privacy Principles (APPs) 177–8, 295–6
- Australian Signals Directorate (ASD) 243
- Australia–United States Free Trade Agreement (AUSFTA) 288, 292
- authentication
  - encryption 167–8
  - multi-factor authentication 170
  - processes 304
  - single-factor authentication 169
  - two-factor authentication 169, 285
- automated debugging and testing of modules 28
- automated invoicing system 87
- automation 87
- average-case scenario 60–1
- back door 240, 300
- backups 140, 305
- bad bots 255, 256
- 'bad' software 300
- baiting 253
- BankAccount 46
- best-case scenario 60–1
- beta testing
  - defined 214
  - strategies for 214–16
- big data 57
- Big O notation 60
- binary files 13
- binary search algorithms 58–60
- binary selection 203
- biometric data 170, 302
- biometric scanners 302
- Bitbucket 241
- bits 4
- blogs 226
- Boolean 34
- Boolean data types 5
- Boolean values 5
- bot attacks
  - source by country 256
  - strategies to reduce the occurrence of 258
- botnet zombies 255
- bots 255, 256
- boundary values 67, 208, 211
- Box (10 GB) 139
- brainstorming 143–5
- branching selection 202
- breakpoints 65, 208–9
- buffer overflow 258–9
- built-in library functions 42–3
- buttons and links, testing 212
- bytes 4
- C# 200
- C (C++) 32, 40, 42
- calculations, checking 213
- camel caps 18
- camel case 18
- casting 61
- chained selection 36–7
- character data type 4–5
- character encoding 4
- Chaudhry, Jay 247
- checking 62–72
  - errors types 62–6
  - test data and test cases 66–70
  - trace tables 70–2
- chief information security officers (CISOs) 305
- child elements 14, 109, 161
- cipher text data 167
- clarity 157
- clarity (usability) 102, 157
- classes 44–5, 46, 161, 205
- classification markings for games 297
- clear and concise code 196
- close-ended questions 100
- cloud computing 294
- cloud storage 217
- code optimisation 29
  - using ChatGPT 30
- code repositories 206, 314
- code reusability 48
- code review 243, 314–15
- cold site recovery plan 271
- collaboration 283
- collecting data 100–2
- combine (creative design) 148
- combining thinking skills 142

- commonalities 48
- communication of message 157
- compare (creative design) 149
- compiler 32
- compile time 32
- completeness (software solution) 157
- computer games
  - classification markings 297
  - and copyright 293
- computer software *see* software
- concepts (project management) 91–2
  - dependencies 92
  - milestones 91–2
- concurrent tasks 93
- Concurrent Versions System (CVS) 242
- conditional formatting 225
- conditional operators 33, 204
- constants 31–2, 200, 201
- constraints 8–9, 94, 105
  - economic constraints 105
  - legal constraints 106
  - social constraints 106
  - technical constraints 106–7
- content scraping 257
- context diagrams 108, 112–14
- control structures 202–4
  - data flows 115
  - drawing 113–14
  - entities 115
  - for patient information system 113
  - processes 114
  - statements 204
  - Susan’s music performance system 121–2
- contiguous memory 197
- continuous delivery (CD) 283
- continuous deployment (CD) 207
- continuous integration (CI) 207, 283
- continuous security 283
- control structures 33–41
  - selection statement 34–8
  - sequence 33–4
- convergent thinking 141, 142
- copyright 288–92
  - and cloud computing 294
  - and computer software 293
  - infringed 293–4
  - and intellectual property 288
  - issues 299
  - and music and computer games 293
  - personal use exemptions 292
  - piracy case studies 288–9
  - when is copyright infringed? 293–4
  - when is copyright not infringed? 293–4
- Copyright Act 1968* (Cth) 106, 175, 288–94
- Copyright Amendment Act 2006* (Cth) 288
- Copyright Amendment (Digital Agenda) Act 2000* (Cth) 175, 261, 288, 292
- Copyright Amendment (Online Infringement) Bill 2015* (Cth) 291
- counted loop 203
- creating 108
- creative design, tips for 148–50
- credit reporting provisions 296
- criteria-based evaluations 313
- critical path 94
- cross-platform GUI development 50
- CSV file 13–14
- customer interactions 87
- customer relationship management (CRM) system 87
- custom project management tool 87
- cyber attackers 272
- cybercrime 268
- cyber security principles 287
- cybersecurity risks 240–79
- Dallas Buyers Club (Film), piracy case study 289–91
- data 136
  - and access restrictions 285
  - factors affecting access of 141
  - ownership and privacy of 135
- data accuracy 157
- databases
  - design, data dictionary use 158
  - Gantt chart for creating, case study 95–9
- data breaches 265–6, 298
- data collection 94, 100–2, 231
- data dictionaries 10, 158, 159
  - as a database design tool 158
  - differences between styles 159–60
  - as a software design tool 159
- data encapsulation 46
- data flow diagrams (DFD) 88, 108, 114–16
  - data flows 114
  - data store 115
  - drawing 115
  - entities 115
  - level 0 DFD 112, 114
  - level 1 DFD 114–15, 116
  - level 2 DFD 114
  - processes 114
  - Susan’s music performance system 122–4
- data flows 112
  - context diagrams 115
- data hiding 46
- data integrity 270
  - threats to 270–2
- data source 198
- data stores (data flow diagrams) 115
- data structures 5–7, 159, 197–8
  - array 5–7
  - record and field 7
  - types 197–8
- data types 3–5
  - Boolean 5, 201, 202
  - character 201, 202
  - comparison operations 3
  - floating point 201, 202
  - fundamental operations 3
  - incompatible 63
  - integer 201, 202
  - numeric 3–4, 201
  - string 201, 202
  - text 201, 202
  - text and character 4–5
- data types and structures
  - characteristics of 198
  - input 197
  - output 197–8
  - storage 197
- data vulnerabilities 265–6
- debugging 62, 208–9
  - code 28–9
  - statements 65–6
- decrease and conquer algorithm 58
- deliberate threats 271
  - detection, prevention and recovery actions 272
- delimited files (CSV) 13–14
- delimiter 13
- denial of inventory 256
- denial-of-service attack (DoS) 255
- dependencies 92
- deposit() 46
- design brief 136
  - feasibility 89
  - originality 90
  - problem, need or opportunity outline 89
  - programming languages used to be 89
  - proposed users 89
- design features 153
- design ideas/ideation
  - brainstorming 143–5
  - critical and creative thinking 148–50
  - effectiveness criteria 155
  - evaluating 151–6
  - generating 141–51
  - graphic organisers 146
  - mind mapping 145
  - mood boards 142–3
  - observations and prepared 150–1
  - sketches and annotations 147–8
- design phase (SRS) 86
- design principles, for appearance and functionality
  - alignment 171–2
  - balance 172
  - contrast 172
  - navigation 174
  - space 172–3
  - text formatting 173
  - usability 173–4

- desk checking 70
  - developer expertise 89
  - DevOps 206
  - differential backups 139
  - dimensions 6
  - disaster recovery plan (DRP) 271
  - distributed denial of service attack (DDoS) 255
  - divergent thinking 141, 142
  - divide and conquer algorithm 54
  - divide by zero error 64
  - documentation
    - software design 158–64
    - using Gantt charts 99
  - documenting
    - problems 88
    - test results 212
    - your testing 212
  - DO/WHILE iteration 39–40
  - Dropbox (2 GB) 139
  - dynamic application security testing (DAST) 284
  - dynamic features, testing 214
- 
- ease of maintenance 48
  - economic constraints 8, 105
  - effectiveness, evaluation of 228
  - effectiveness of solution 157, 216
    - criteria for evaluating 157
    - factors influencing 229–30
  - efficiency, evaluation of 228
  - efficiency of algorithms 60–1
  - efficiency of solution 157, 216
    - clear and concise code 196
    - internal documentation 196–7
    - user-centred design 195–6
  - ELSE condition 37
  - ‘ELSEIF’ 36
  - emerging trends, in programming
    - languages 27–31
  - employee training 302–3
  - encapsulation 44, 46–7
    - with Thermostat class 47
  - encryption 167, 243, 303, 314
  - entities (context diagrams) 115
  - entities (data flow diagrams) 115
  - errors types 62–6
  - Essential Eight 286
  - ethical issues 298–9
    - artificial intelligence, use of 298–9
    - ‘bad,’ hidden and malicious software 300–1
    - copyright issues 299
    - ineffective security practices 298
    - intellectual property 299
    - ‘safe’ programs 299–300
    - in software development and use 299–300
  - evaluating design ideas 151–6
    - evaluating solutions 216
    - evaluating the efficiency and effectiveness of solutions 156–7
    - evaluation 227–9
      - of the client response 136
      - of project plans 229
      - vs testing 156
      - what it does not do 227
      - what to evaluate? 227–8
      - when to evaluate? 228–9
      - see also* software requirements specification (SRS)
    - evaluation criteria 156, 217, 222
    - evaluation methods 228
    - evaluation strategies
      - evaluation criteria 217–21
      - responsibility 222
      - time frames 221
    - event-based threats 271
      - detection, prevention and recovery actions 271, 272
    - event-driven programming 50
    - events 45, 94
    - existence checks 61, 208
    - expected and actual output 210
    - expected results 62
    - exploit 244
    - extend (relationship) 110–11
    - Extensible Markup Language (XML) file 14–16
    - external buttons, testing 212
    - external entities 112, 114
    - external links to your product, testing 212
    - feasibility 89, 90
    - feed fetchers 255
    - field 7
    - file access 141
    - file allocation table (FAT) 140
    - file management plan
      - archiving 141
      - backups 139
      - Cloud storage 139
      - creating documents 137–8
      - disposing of files 140–1
      - storing and retrieving files within directory structure 138–9
    - file organisation 141
    - files 13–16
      - delimited files 13–14
      - Extensible Markup Language file 14–16
      - naming conventions 158
      - operations 13
      - plain text file 13
    - financial feasibility 90
    - firewalls 269, 301–2
    - fit for purpose 104
    - floating points 3, 4
    - flow diagrams 150
    - flow of execution 70
    - FOR iteration 40–1, 58, 60
    - FOR Loop 7
    - FreeBSD 242
    - FreePascal 242
    - full access 169
    - full backup 139
    - functionality 102, 157
    - functional requirements 8, 102–3
      - Susan’s music performance system 118
    - function call 41, 42
    - function declaration 42
    - function definition 42
    - functions 41–4, 204–5
      - built-in library 42–3
      - and methods 204–5
      - using pseudocode 43
    - fundamental data type operations 3
    - Gane-Sarson notation style 113, 115
    - Gantt charts 92–94
      - adjustments and annotations 223
      - for creating a database, case study 95–9
      - documentation using 99
      - features 93–4
      - time allocation resources 93–4
    - GCC 242
    - generalisations 44, 47–8
      - relationship 109–10
    - generating design ideas 141–51
      - techniques for design ideation 142–51
    - getWet() function 41
    - gift card balance checking 256
    - GitHub 241
    - GitHub Copilot 207, 241
    - global variables 31, 200, 201
    - goals of organisations 281
    - goals of software 281
    - good bots 255
    - Google Drive (15 GB) 139
    - Google Maps 165
    - graphical user interfaces (GUIs) 50, 198, 204
    - graphic organisers 146
    - hacking 271
    - handwritten annotations 224
    - hard-coding variables 45
    - hardware limitations 106
    - header comment 16
    - Health Records Act 2001* (Vic) 140, 175, 288
    - HealthyCare Clinic 140
    - hidden software 300
    - high-fidelity prototypes 196
    - high-level language 27
    - honeynets 271
    - honeypots 271
    - hot site recovery plan 271

- Howard, Michael 262
- human error 305
- human intelligence 30
- human languages 27
- Hungarian notation 18, 19
- .....
- iCloud (5 GB) 139
- identity and access management (IAM) 243
- IF condition tests 36
- impersonation 272
- include (relationship) 110
- incremental backups 139
- index out of range error 63
- infinite loop 39, 204
- informal testing 211
- Information Privacy Principles (IPPs) 178–9
- Information Security Manual (ISM) 286, 287
- infrastructure as a service (IaaS) 304
- inheritance 44, 48–9, 109, 161
- input 11
- input-process-output (IPO) charts 11
  - creating chart 163
  - describing output 162
  - determining the input 162
  - process(es) 162
  - refining and revising 163
  - selecting functional requirement 162
- inputs, inspecting 212
- insider threats 250–3
- instantiations 45
- instructions 32
  - and syntax 199–200
- integer overflow 4
- integers 3, 4
- integrated developer environments (IDEs) 72
- integration with back-end code 50
- integrity of data 270–2
- intellectual property (IP) 9, 299
  - and copyright 288
- Interactive application security testing (IAST) 284
- interface 10, 198
  - mirrored using columns 173
  - readability and accessibility 173
  - username and password interface 152
- interfaces between solutions, users and network
  - context diagrams 112–14
  - data flow diagrams (DFD) 88
  - networks 88
  - use case diagrams (UCD) 88
- internal buttons, testing 212
- internal documentation 16–18, 196–7
  - conventions 16–18
  - writing 17
- International Standards Organisation, recommended specific ways of naming
  - files and variables 158
- interoperability 165–6
- interpreter 32
- interviews 101
- intrusion detection and prevention 273
- intrusion detection systems (IDS) 302
- iteration 38–41
  - DO/WHILE 39–40
  - FOR 40–1, 58, 60
  - REPEAT/UNTIL 41
  - WHILE 39, 66
- iteration structure 203–4
- .....
- Java 89, 200
- JavaScript 50, 90
- .....
- keyloggers 272
- Khatilov 252–3
- .....
- legal constraints 106
- legal requirements
  - relating to ownership and privacy of data information 175–9
  - for storage and disposal of data and information 288–96
- libraries 206, 207
- licensing agreements 293
- linear search algorithms 58
- linked lists 197
- links, internal and external, testing 212
- ‘living off the land’ (LOTL) 288
- loading times 213
- local variables 31, 200, 201
- locked server rooms 302
- logic 198
- logical operators 33, 204
- logic bombs 247
- logic errors 64, 72
- lowerCamelCase 18
- low-fidelity prototypes 196
- low-level languages 27
- LucidChart 146, 150
- .....
- machine learning models 29
- maintainability 103, 157
- malfunction 271
- malware/malicious software 246–7, 261, 272
  - security procedures 269
- man-in-the-middle (MITM) attacks 249–50
- mathematical constants 32
- mathematical notation 60
- Melbourne Robotics Centre (MRC) 321
- memory leak 63
- memory-safe 288
- mental imagery 149
- methods 45, 204–5
- Microsoft Exchange zero-day 260
- milestones 91–2
- mind mapping 145
- mission statement 281
- mobile site recovery plan 271
- mock-ups 10–11, 161
- monitoring and logging 272
- mood boards 142–3
- multi-factor authentication (MFA) 170, 248, 303
- multi-level inheritance 161
- multi-line comments 17
- multiple branching selection 203
- music, and copyright 293
- MediaPlayer class 46
- .....
- naming conventions 18–19
  - for files and variables 158
- nested selection 37–8
- nesting 204
- Netflix customers, email scam, case study 307–8
- Netflix email phishing scam 307
- network and connectivity 107
- non-disclosure agreements (NDA) 285
- non-functional requirements 8, 103–4, 128–9
  - Susan’s music performance system 117–18
- non-technical constraints 8–9
- Notifiable Data Breaches (NDB) scheme 265
- numeric data type 3–4
- .....
- object description 10, 160–1
- objective results (evaluation) 228
- objectives 136
  - of organisations 281
- object-oriented languages 160
- object-oriented programming (OOP) 44–9
  - abstraction in 46
  - classes 44–5
  - encapsulation 46–7
  - generalisation 47–8
  - inheritance 48–9
  - languages 7
  - methods 45
- objects 45, 205
- observations 102, 150–1
- off-site storage 139
- one change at a time, tested 208
- one-dimensional arrays 5, 6
- OneDrive (5 GB) 139
- online piracy 289–91
- on-site storage 139
- OpenAI’s Codex 207
- open-ended questions 100

- open source 293
- Open Web Application Security Project (OWASP) Testing Guide 262, 263, 265
- operating system 137, 165
- operational costs 87
- operational feasibility 90
- organisational goals 281
- originality 90
- ‘OR’ operator 35
- out-of-bounds errors (OOB) 288
- output 11
- overflow error 63
- OWASP Testing Guide 262, 263, 265
- ownership of data 9
  - information, legal requirements 175–9
- parameters *see* arguments
- parent element 14, 109, 161
- pass by reference 43
- pass by value 43
- patches 63
- patch management 273
- patient information system
  - DFD level 1 116
- patient information system context diagram 113
- penetration testing (pen test) 265
- performance requirements 89, 106
- personas 196
- personnel changes, project plan effectiveness 229
- pharming 253
- phishing 253, 254, 261, 307
- piggyback entry 253
- piracy 292
  - case studies 289–91
  - Government passes bill aimed at reducing online piracy 291
- pivot 54
- plain text data 167
- plain text (TXT) file 13
- platform as a service (PaaS) 304
- PMI 146
- pointer 31
- portability 104
- post-test iteration 39
- post-test loop 203
- PowerPoint 148
- predecessor tasks 97
- preferred design 154
- pre-test iteration 39
- pre-test loop 203
- pretexting 253
- price scraping 257
- primary logical operators 33
- priorities quadrant 226
- Privacy Act 1988* (Cth) 106, 140, 176, 178, 265, 288, 294, 298
  - application 176–7
  - Australian Privacy Principles 177–8, 294
  - credit reporting provisions 296
  - penalties 177
  - what is included? 294
  - who is covered? 295
- Privacy Amendment (Enhancing Privacy Protection) Bill 2012 176
- Privacy Amendment (Notifiable Breaches) Act 2017* (Cth) 265
- Privacy and Data Protection Act 2014* (Vic) 106, 175, 178–9, 296–301
  - Information Privacy Principles 178–9
  - penalties 179
- privacy laws 9
- privacy of data and information, legal requirements 175–9
- private key 168
- private visibility 206
- problem-solving methodology (PSM) 88
- process (context diagrams) 114
- process (data flow diagrams) 114
- process (project management) 91–9
  - documentation using Gantt charts 99
  - sequencing 93
  - task identification 92
  - time allocation resources 93–4
- programmers 300
- programmer-selected character 13
- programming bugs, case study 300–1
- programming languages 2, 27–30, 88
  - access modifiers 205–6
  - arithmetic, logical and conditional operators 204
  - arithmetic operators 32
  - choice of 198–9
  - classes and objects 205
  - commenting conventions 17
  - conditional operators 33
  - constants 31–2, 200–1
  - control structures 33–41, 202–4
  - data types 201–2
  - efficiency of algorithms 60–1
  - emerging trends in 27–31
  - features of 31–44
  - functions 41–4
  - functions and methods 204–5
  - graphical user interfaces (GUIs) 50, 204
  - instructions 32
  - instructions and syntax 199–200
  - logical operators in 33
  - modules function correctly 62–72
  - object-oriented programming 44–9
  - processing features 198–206
  - searching, algorithms for 57–60
  - sorting, algorithms for 51–7
  - and tools 106
  - validation techniques 61–2
  - variables 31, 200–1
- project logs 224
  - conditional formatting 225–6
  - template 224
  - value of 225
- project management 91–9
  - concepts 91–3
  - processes 91
- project plans
  - adjustments to tasks and timeframes 223
  - annotations 223–4
  - effectiveness of 230–1
  - efficiency tips 226
  - factors influencing 229–30
  - keeping logs 224–6
- prolog 14
- prompts 27
- proposed users 89, 90
- protected visibility 206
- pseudocode 5, 6, 12, 163–4
  - binary search 59
  - to check an age range 67
  - common keywords and symbols used in 163
  - completed test data for 72
  - FOR iteration in 40
  - linear searches 58
  - logic error 71
  - object and method 45
  - quick sort 57
  - representation of a function in 44
  - rules of 12
  - selection sort 53
  - switch/case control structure 38
  - test data for 71
  - trace table for 70
  - WHILE loop 70
- public key 167
- public key encryption 12, 167
- public visibility 205
- PyCharm IDE 65
- PyQt 50
- Python 27, 42, 89, 200
  - breakpoints in 65
  - built-in functions 43
  - selection sort implemented using 54
- qualitative data 100
- quantitative data 100
- quick sort algorithm 54–7
- quid pro quo attacks 253
- RAM 45
- range checks 61, 208
- ransomware 244–5, 248, 302
- React framework 90
- readability 157
  - checklist 213
- record 7

- recording the progress of projects 223–6
  - adjustments to tasks and timeframes 223
  - annotations 223–4
  - efficiency tips 226
  - evaluating project plans 229
  - keeping logs 224–6
- recursive algorithms 56
- relationship (use case diagram) 109
  - association 109
  - generalisation 109–10
  - include and extend 110–11
- relevance (data) 157
- relevant test data 210
- reliability 103
- REPEAT/UNTIL iterations 41
- repetition *see* iteration
- reports, as part of data collection 101
- representing designs 10–12
  - data dictionary 10
  - input–process–output (IPO) charts 11
  - mock-up 10–11
  - object descriptions 10
- research (creative design) 149
- resource availability 107
- responsible and ethical use of AI 30
- restricted access 169
- return value 41
- risk appetite 264
- risk minimisation 282–8
  - development and operations 282–4
  - essential eight 286–8
  - minimising costs 284–5
- risk taking, persistence and bravery (creative design) 151
- risk tolerance 264
- robust identity 314
- robustness 104
- root element 14
- Runtime application self-protection (RASP) 284
- runtime errors 63
- .....
- ‘safe’ programs 299–300
- SAT project 125
- scalability 48, 89
- scalability and maintainability 106
- scam statistics 268
- Scamwatch website 268
- School-assessed Task (SAT) 152
- scope 9–10
  - of software solution 105
- scope creep 108, 229
- searching, algorithms for 57–60
  - binary search 58–60
  - linear search 58
- Secure Sockets Layer (SSL) 168
- security
  - and access factors 303
  - assessments 272
  - audits 273
  - cameras 302
  - data integrity 270–2
  - development practices 313–15
  - requirements 107
  - restricting URL access 259
  - software and data vulnerabilities 265–73
  - software security 261
  - strategies for minimising potential risks 262–5
  - threat modelling 266–73
  - vulnerabilities and risks 245–61
- security attacks
  - detection and prevention 268–9
  - malware 246–8
- security controls
  - code review 243
  - encryption 243
  - robust identity and access management 243
  - separated environments 245
  - software and data protection 240
  - software updates and patches 244–5
  - version control and code repositories 240–3
- selection sort algorithm 51–4
- selection statement 34–8
- selection structure 202
- semantic errors 64
- separated environments 245
- sequencing (processes) 93
- session hijacking 258
- shoulder surfing 272
- single-factor authentication 169
- Single Sign-On (SSO) solutions 165
- sketches and annotations 147–8
- slack time 93
- sleep on it (creative design) 149
- smishing 272
- snake case 18
- social and legal requirements 8–9
- social constraints 106
- social engineering attacks 253–4, 272
  - protection against 254
- SofDevProjectV1.php 137
- software 87
  - acquired from third parties 259–60
  - copyright 293
  - and data protection 240
  - and data vulnerabilities 265–6
  - distribution with app stores 294
  - goals of 281
  - open-source 261
  - reasons for developing 281
  - ‘safe’ programs 299
  - software acquired from third parties 259–60
- software as a service (SaaS) 304
- software auditing 262–5
- software dependencies 106
- software design documentation 158–64
- software developers 101, 293
  - ethical issues 299
  - protection of their work 293
- software development 284
  - AI-based assistants 207
  - APIs and libraries 207
  - code repositories 206–7
  - costs reduction 87
  - gaps, in market 87
  - organisational objectives or needs 87
  - productivity and efficiency 87
- Software Development algorithms 60–1
- software development life cycle (SDLC) 263
  - security testing strategies 262–5
- software development practices
  - how to protect software and data? 240
  - strategies for minimising potential risks 262–5
- software externally
  - advantages of 285
  - disadvantages of 285
- software licence 293
- software requirements specification (SRS) 88, 100, 107–8, 136, 151
  - analysis phase 107
  - analytical tools depicting existing processes and systems 108–12
  - collecting data 100–2
  - constraints 105–7
  - design phase 86
  - functional and non-functional requirements 102–4
  - scope 105
  - software to track music performances, case study 117–25
- software security 261
  - incident/attack types across businesses 261
- software solution
  - efficiency and effectiveness 156–7, 216
  - evaluation 229
  - project management 91–9
  - specifications 136
  - testing your 212–14
  - usability 210
  - what is it? 87
- software testing 207–9
- software to track music performances, case study 117–25
- software updates 244–5
- software vulnerabilities 259, 265–73
- solution boundaries 105
- solution requirements 8
  - functional requirements 8
  - non-functional requirements 8
- sorting algorithm 51–7, 60

- SourceForge 242
- spam 255, 272
- spam filters 269, 272
- spear phishing 253
- specific program values 32
- spider diagrams 146
- split infection virus 246
- spyware 254, 300
- stacking 204
- Static application security testing (SAST) 284
- storage media 141
- streamlining 87
- strip the problem right back to its most
  - basic parts (creative design) 148
- strong password policies 302–3
- Structured English 12
- subjective results (evaluation) 228
- subscription software 259
- substitute (creative design) 148
- Subversion, case study 242
- successor tasks 93
- superclass 48
- surveys 100
- Susan's music performance system, case study 117–25
- Swift 27
- swiss cheese virus 246
- switch/case conditional 203
- switch/case statement 38
- switch statement 38
- symmetric key encryption 167
- syntax 199–200
- syntax errors 62–3
- system boundary 111
- system integration 106
- systems analyst 281
- Systems Hungarian 19
- system testing 211
- .....
- table of contents (TOC) 140
- tailgating/piggyback entry 253
- task identification 92
- technical constraints 8, 106–7
- technical feasibility 90
- technical issues, project plan effectiveness 229
- Tesla 252–3
- test cases 66–70, 210
- test data 66–70, 211
  - boundary values 208, 211
- testing
  - defined 210
  - document 212
  - vs evaluation 156, 227–9
  - usability of solutions 210–11
  - vs validation 198
- testing table 210, 212
- testing your software solution 212–14
- .....
- test results, documenting 212
- text alignment types 172
- text data type 4–5
- text files 13
- Thermostat class 47
- Thinking in Pictures* 149
- third-party apps 165
- threat modelling
  - access control reviews 273
  - detection and prevention 268–9
  - identifying and mitigating threats 270–3
  - intrusion detection and prevention 273
  - monitoring and logging 272
  - patch management 273
  - principles 266–7
  - security assessments 272
  - security audits 273
  - security requirements 269–70
  - training 273
- threats, to data integrity 270–2
- time allocation resources 93–4
- timeliness 157
- Tkinter 50
- trace tables 70–2
- training, for employees 273
- Transport Layer Security (TLS) 168
- tree 14
- Trojan horses 247
- trojans 247
- truth table 35–6, 69
- Twitter (X) hack 248
- two-dimensional arrays 5, 7
- two-factor authentication 169, 285
- two-factor recovery process 272
- type checks 61, 208
- type mismatch error 63–4
- .....
- unacceptable security risks, determining 264
- Unified Modeling Language (UML) 88, 108
- Unit 4 Outcome 1 (U4O1), preparing for 237–8
- universal design 141
- Unix timestamp 4
- unsigned integers 4
- updateInternalState() method 47
- uptime 104
- URL access, restricting 259
- usability 50, 103, 157, 166
- use-after-free (UAF) 288
- use case 102, 109
- use case diagrams (UCD) 88, 108–12
  - actors 109
  - drawing 111–12
  - relationship 109–11
- Susan's music performance system 119–20
- system boundary 111
- use cases 102
- .....
- user-centred design 195–6
- user experience (UX) 107, 135
  - affordance 164–5
  - interoperability 165–6
  - usability 166
- user interface 107
- username and password 169
- useUmbrella() function 41
- .....
- validation, vs testing 198
- validation checks 208
- validation techniques 61–2, 208
- validation test data 66
- variables 3, 18, 31, 158, 159
  - local and global 31
- variables and constants 200–1
- VB.Net 42
- VCAA programming requirements 200
- VCE Software Development 62, 106, 159, 200
- version control 206, 240–3, 314
  - and code repositories 240–3
- version control system (VCS) 17, 241
- Village Roadshow, piracy case study 291
- viruses 246, 300
- vision statement 281
- visualisation 149
- vulnerabilities and risks
  - application programming interfaces (APIs) 246
  - combined environments 260–1
  - cyber security incidents 253–9
  - ineffective code review practices 260
  - insider threats 250–3
  - logic bombs 247
  - malware 246
  - man-in-the-middle (MITM) attacks 249–50
  - poor identity and access management practices 248
  - risks from software acquired by third parties 259–60
  - Trojan horses 247
  - unpatched software 248
  - viruses 246
  - worms 247
- .....
- WannaCry cryptoworm 244–5
- WannaCry ransomware 248
- warm site recovery plan 271
- WBS diagram 92, 93, 95
- weblogs 224, 225
- web scraping 255–8
- well-written internal documentation 16
- WHILE iteration 39, 66
- whiteboards 144
- withdraw() 46

work breakdown structure (WBS) 92  
worms 247  
worst-case scenario 58, 60–1  
writing internal documentation 17

XML element types and characteristics  
14  
XML files 14–16  
XML tree 14

Yourdon-DeMarco notation style 113  
zero-day vulnerabilities 260  
zombies 255, 256





# Flexible online learning designed to support you

## Nelson MindTap puts **you** at the centre

Access resources and content that make learning simpler yet smarter to help you go beyond drill and practice.



**Develop** your understanding through the wide range of weblinks and digital resources.

**Revise** using quizzes to practice your skills and build your confidence.

**Navigate** your own path, accessing the content and support you need whenever you need it in your learning journey.

Find everything you need to access your Nelson MindTap course at **[cengage.com.au/nelsonmindtap](http://cengage.com.au/nelsonmindtap)**

 Nelson MindTap

ISBN 978-0170483964



9 780170 483964