



VCE Applied Computing: Unit 3 and 4 Software Development: Software tools and functions and outcome-specific requirements

The VCE Applied Computing Study Design (From 2025) mandates software tools and functions and outcome-specific requirements that students are to follow in Unit 3 and 4 Software Development. For 2025, schools must use these software tools and functions and outcome-specific requirements in the study.

Government schools are advised to refer to the Department of Education's Generative Artificial Intelligence policy at: [Generative Artificial Intelligence: Policy | education.vic.gov.au](https://www.education.vic.gov.au/generative-artificial-intelligence-policy).

Catholic and independent schools should refer to their sector authorities for advice on generative artificial intelligence.

Students can use code repositories as part of the normal teaching and learning program in order to become familiar with the key knowledge. However, students are not to use code repositories for work to be completed as part of their assessment. This is to ensure teachers can authenticate student work.

Teachers of VCE Applied Computing should note that the Software tools and functions and outcome-specific requirements may be revised for 2026 and notification will be published in the [VCAA Bulletin](#).

Software tools and functions

The following software tools and functions are outlined in the VCE Applied Computing Study Design (From 2025) for Unit 3 and 4 Software Development:

- Unit 3 Area of Study 1 – Software development: programming (Page 53)
 - Students are required to both study and use an appropriate object-oriented programming language.
- Unit 3 Area of Study 2 – Software development: analysis and design (Page 56)
 - Students are required to both study and use Unified Modelling Language (UML) tools to create use case diagrams.
 - Students are required to use, but are not required to study:
 - an appropriate tool for documenting and modifying project plans and
 - appropriate tools for ideation and generating designs.
- Unit 4 Area of Study 1 – Software development: development and evaluation (Page 61)
 - Students are required to both study and use an appropriate object-oriented programming language.
 - Students are required to use, but not required to study:
 - an appropriate tool for documenting and modifying project plans and
 - programming tools and/or integrated development environments to facilitate programming and testing of solutions.

Programming languages

The following is a list of programming languages that can be studied and used:

- Python
- Swift
- Objective-C
- Visual Basic.NET
- Ruby
- Java
- C#
- Golang (Go)
- JavaScript
- PHP
- TypeScript
- Kotlin

It should be noted that this list is not exhaustive, and that any programming language that meets the Programming requirements outlined below may be used in the delivery of the study.

Programming requirements

The following is a list of programming requirements that are studied and used, and that students are expected to be able to apply. Note that this list is not exhaustive; learning does not have to be confined to the requirements listed below.

In the development of working software modules (Unit 3 Area of Study 1) and the software solution (Unit 4 Area of Study 1), the chosen object-oriented programming (OOP) language should provide students with the ability to carry out the development stage of the problem-solving methodology within the three conceptual layers of: interface, logic and data source.

Interface

The chosen OOP language must enable students to develop a graphical user interface (GUI) for use in a digital system through one or more of the following options:

- an Integrated Development Environment (IDE) (drag and drop/WYSIWYG)
- using code (same language)
- using code (a supporting language).

Logic

Programming requirements for the logic layer:

- instructions
- program control structures (sequence, selection and iteration/repetition)
- operators
 - *arithmetic*: addition, subtraction, multiplication, division, integer division, modulus, unary plus/minus, increment/decrement
 - *logical*: AND, OR, NOT
 - *conditional/comparison*: equality, inequality, less than, less than or equal to, greater than, greater than or equal to
- functions and methods
- classes and objects.

Data source

Programming requirements for the data source layer:

- initialise, set and access local and global variables, and constants
- using relevant data types (numeric, text, Boolean)
- read data from external sources, such as files and databases (local or cloud-based)

- write data to external sources, such as files and databases (local and cloud-based).

The following file formats that can be used are:

- delimited (CSV)
- plain text (TXT)
- XML.

Outcome-specific requirements

The outcome-specific requirements for Unit 3 and 4 Software Development provide specifications and scope for the listed key knowledge dot points in this document.

Unit 3: Emerging trends in programming using AI

The following is a list of emerging trends in programming using artificial intelligence:

1. Using prompts to generate code.
 - Based on a single or a series of prompts, generative AI platforms are able to generate code quickly.
 - The benefits of generating code using prompts include, but are not limited to:
 - increased productivity due to reduced development and debugging times
 - reduced development costs
 - enabling non-developers to be able to generate functional code
 - reduced need to develop deep expertise in multiple languages
 - automated code documentation and improved code quality
 - potential support of problem-solving and innovation
 - The disadvantages of generating code using prompts include:
 - potential for bugs and errors to be present
 - reliance on trial/error and specific prompts being used
 - risk of vulnerabilities and threats being present
 - copyright concerns
 - limited context or situational understanding
 - dependence on human oversight
 - An example of a prompt to generate a module that takes two values and outputs the sum and product of the values could be:
Create a program in [chosen OOP language] that takes in two integer values and then outputs the sum, product and average of the values. The output should be labelled.
2. Automated debugging and testing of modules.
 - Automated debugging and testing tools have been a common component of Integrated Development Environments (IDEs) for some time.
 - The benefits of testing modules using automated debugging and testing include, but are not limited to:
 - real-time identification of issues with code
 - suggested fixes to code
 - rapid feedback

- consistent approaches to debugging and testing
 - improved code quality
- The integration of AI into automated debugging and testing tools has both improved and introduced challenges to the testing of modules.
- Improvements include:
 - increased productivity
 - enhanced accuracy
 - scalability
 - identification of issues before they occur
- Challenges include:
 - incorrect flagging of code as containing errors or missing subtle issues
 - dependency on training data
 - reduced job opportunities for developers and testers
- Examples of automated debugging and testing tools include, but are not limited to:
 - Python IDEs: Behave, Lettuce, Robot, Pytest, TestProject
 - Visual Studio: Live Unit Testing, IntelliTest,
 - PHP: Codeception, Selenium.

3. Code optimisation.

- AI tools can be used to analyse code performance and suggest improvements to enhance efficiency in relation to runtime, memory usage, security or other technical requirements.
- Types of code optimisation possible using AI include:
 - resource optimisation (CPU/memory)
 - algorithm optimisation
 - code simplification
 - standardisation to organisational or industry rules
 - security optimisation
 - automated comment generation and summarisation
- Examples of tools that can assist developers with code optimisation using AI include, but are not limited to: GitHub Copilot, Tabnine, TensorFlow, Codeium.

4. Responsible and ethical use of artificial intelligence tools.

- Developers should always utilise AI tools responsibly and ethically.
- This includes:
 - understanding the technical limitations of any AI tools being used or considered for use
 - being transparent in their use of AI tools
 - conducting thorough code reviews on all AI-generated code
 - being aware of the potential for bias introduction
 - avoid over-reliance on AI tools
 - understanding the environmental impact of using AI tools
 - respecting intellectual property, copyright and licencing agreements.

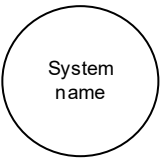
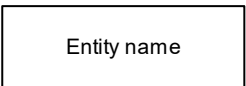
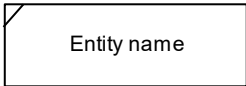
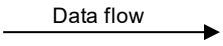
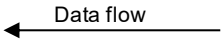
Unit 3: Analytical tools

The following is a list of analytical tools for depicting relationships between users, data and systems:

1. Context diagrams (Level 0) with the components of a system, and entities and data flows.

Context diagrams (Level 0) are used to represent the data flows between a system and external entities.

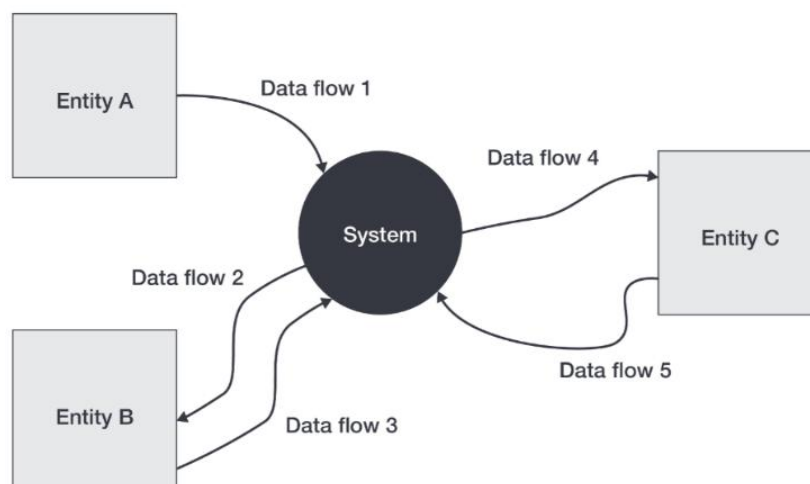
Context diagram components and symbols

System Represented using a circle that contains the name of the system	
External entity Represented using a square or rectangle that is labelled with the name of the entity. Entities are users or other systems that are external to the system being represented. Where necessary, an entity may be duplicated in a context diagram. Where this occurs, the duplicated entities are represented using a square or rectangle with a diagonal line across the top-left corner.	 
Data flow Represented using a labelled, unidirectional line with an arrowhead denoting the flow of data	 

Context diagrams rules:

- Data cannot flow directly between two entities.
- Data flows cannot cross each other.
- Each data flow should only represent a single set of information being transferred.

A sample context diagram is found below:

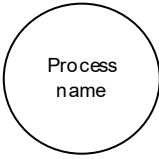
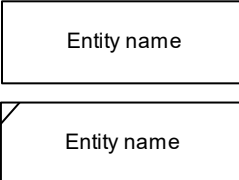
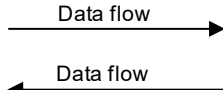
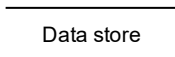


2. Data flow diagrams (Level 1) with the components of processes, entities, data stores and data flows.

Data flow diagrams (Level 1) depict the flow of data within a system, illustrating how data is provided, processed, stored and output.

Data flow diagrams should be consistent with context diagrams that depict the same system.

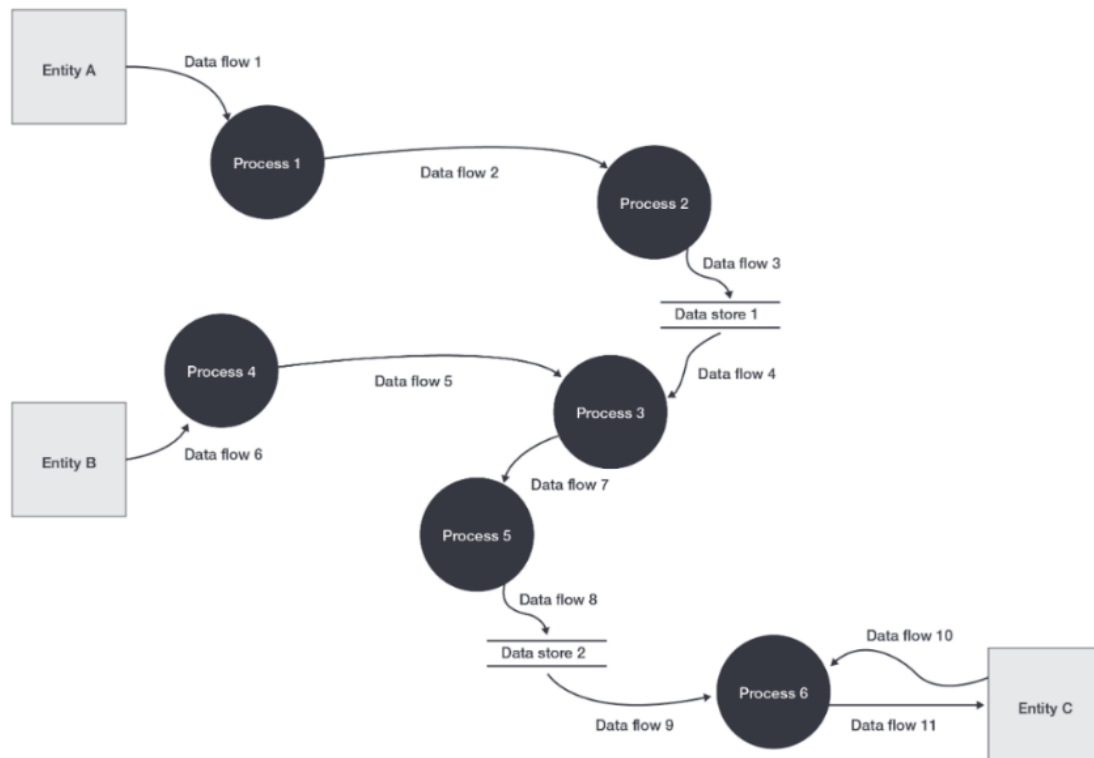
Data flow diagram components and symbols

<p>Process</p> <p>Represented using a circle that contains the name of the process</p>	
<p>External entity</p> <p>Represented using a square or rectangle that is labelled with the name of the entity. Entities are users or other systems that are external to the system being represented.</p> <p>Where necessary, an entity may be duplicated in a context diagram. Where this occurs, the duplicated entities are represented using a square or rectangle with a diagonal line across the top-left corner.</p>	
<p>Data flow</p> <p>Represented using a labelled, unidirectional line with an arrowhead denoting the flow of data</p>	
<p>Data store</p> <p>Represented using horizontal parallel lines, with the name of the data being stored labelled between the two lines.</p>	

Data flow diagram rules:

- Data cannot flow directly between entities.
- Data cannot flow directly between data stores.
- Data cannot flow directly between an entity and data store/s.
- Data flows cannot cross each other.
- Each data flow should only represent a single set of information being transferred.
- A process must have at least one input and one output.
- A data flow that outputs from a process should be different in name to the input received.
- A data store must have at least one input and one output.
- All processes within the system must be connected to at least one data store or process within the system, or an external entity.
- All external entities must, at least, provide one input or receive one output from the system.

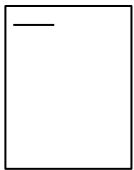
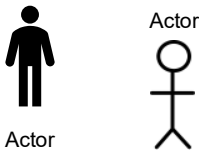
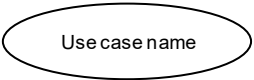
A sample data flow diagram is found below:




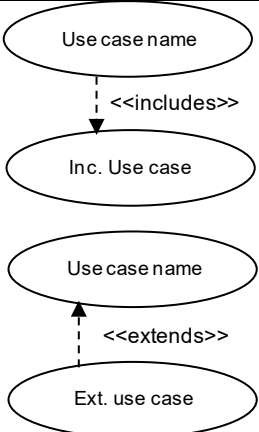
- Use case diagrams with the components of a system boundary, actors, associations, relationships (includes and extends) and use cases.

A use case diagram visually represents the interactions between users and a system.

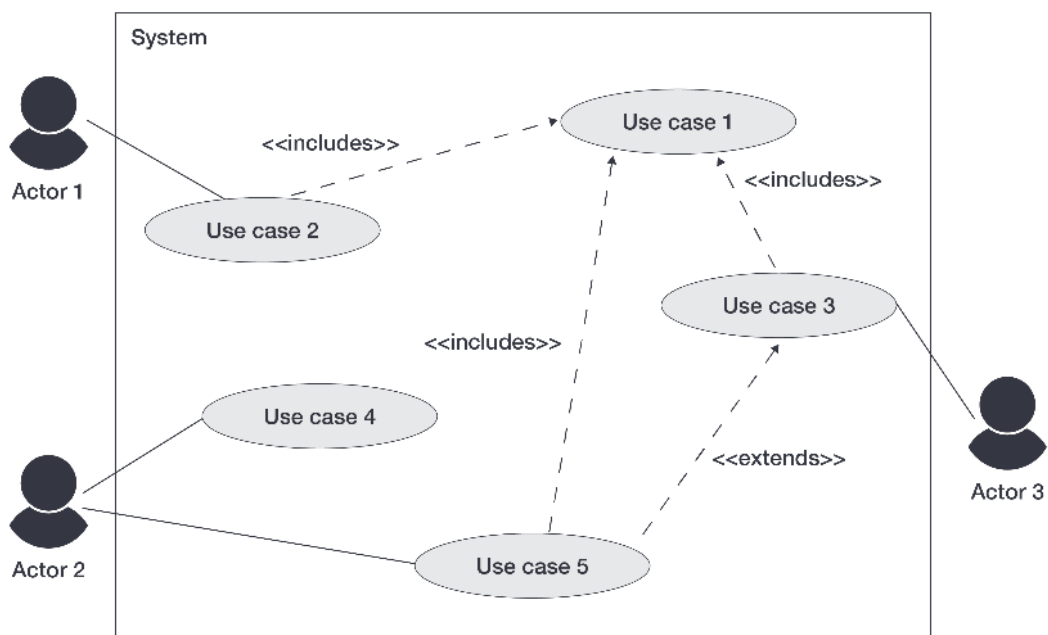
Use case diagram components and symbols

System boundary Represented using a rectangle that contains all of the defined use cases, associations and relationships. The rectangle is labelled with the name of the system in the top left-hand corner.	
Actor Represented using a person icon or stick figure, with the role labelled underneath or above the figure.	
Use cases Represented using an ellipse, with the name of the use case labelled inside the ellipse.	

Association Represented using a line between the use case and the related actor.	
--	---

Relationships Represented using a dashed unidirectional line <<includes>> Used to link use cases, where one use case always incorporates the behaviour of another use case. The arrow points from the base use case to the included use case. <<extends>> Used to link use cases, where one use case extends the behaviour of another use case based on certain conditions being met. The arrow points from the extending use case to the base use case.	
--	---

A sample use case diagram is found below:



Unit 3: Ideation techniques and tools

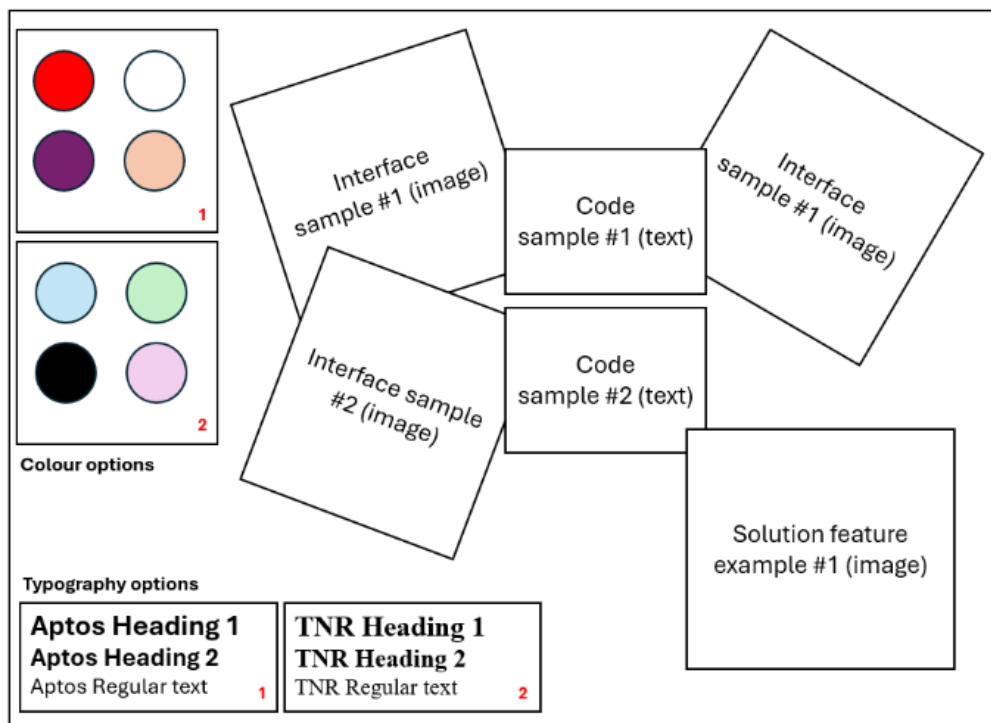
The following is a list of ideation techniques and tools for generating design ideas:

1. Mood boards.

Mood boards are used to collate imagery, colours, typography and code samples to explore and inspire the overall direction of the design of the software solution.

Samples may be annotated to indicate potential applicability to the solution, specific features to be/not to be included, as well as reasoning why identified elements should/shouldn't be applied within the final design.

A sample mood board is found below:



2. Brainstorming.

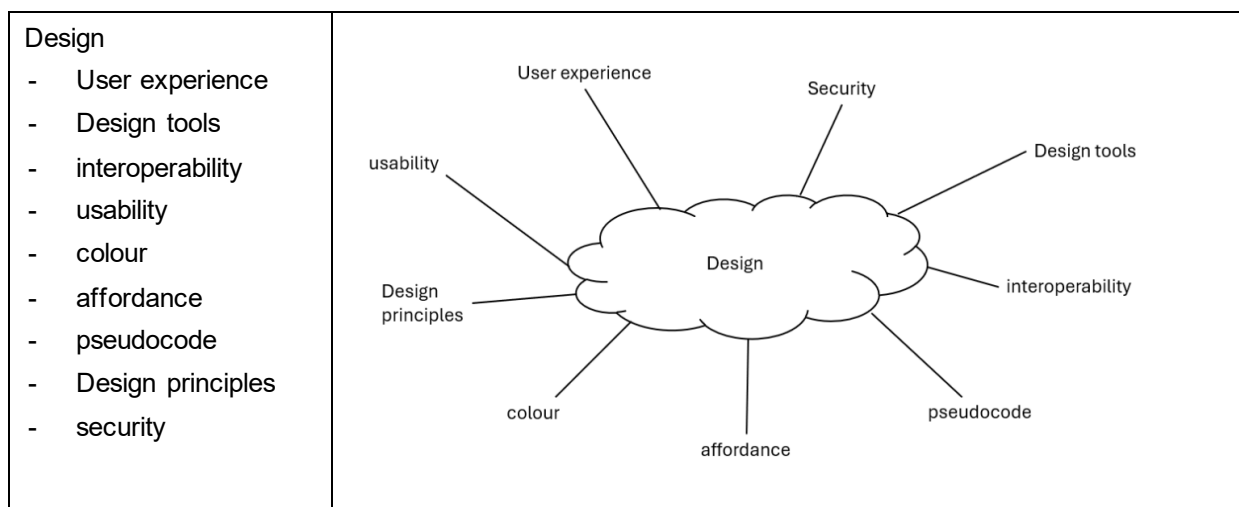
Brainstorming results in an unstructured collation of related ideas, based around a central theme/concept.

Brainstorming can be documented as a list of ideas, or as a diagram.

When documented as a diagram:

- The central theme/concept is represented using a rectangle, cloud-shape or ellipse.
- Related ideas (as words) are connected to the central theme/concept using straight lines.

Examples of brainstorming are found below:



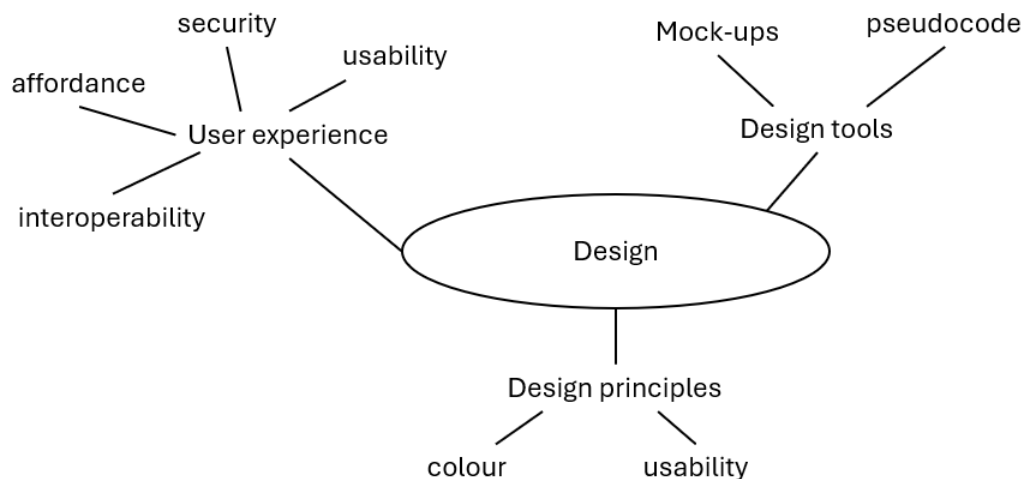
3. Mind maps.

Mind maps are a structured collation of related ideas, based around a central theme/concept, represented as a diagram.

The central theme/concept is represented using a rectangle, cloud-shape or ellipse.

Related ideas (as words) are connected to the central theme/concept using straight lines. Ideas branch out further as required.

A sample mind map is found below:

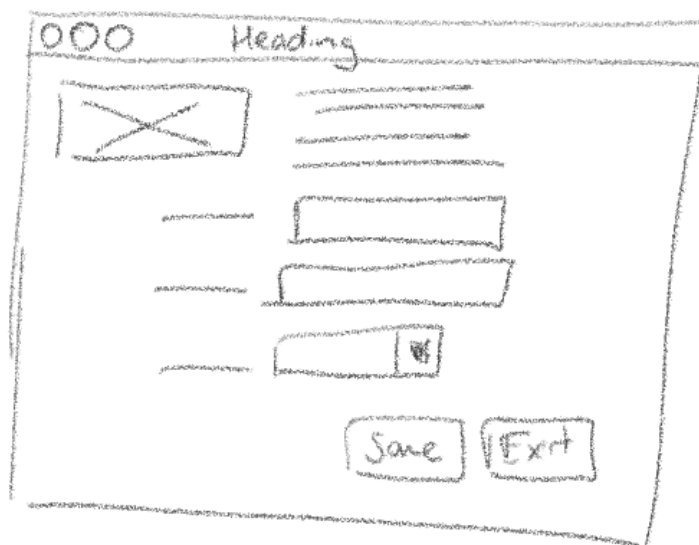


4. Sketches.

Sketches are unrefined illustrations that briefly depict an interface or indicate how different components of the software solution could be linked or interact.

Sketches may be drawn either by hand or using digital drawing tools.

A sample sketch is found below:



5. Annotations.

Annotations can be used to organise thoughts and highlight or justify design considerations within design ideas, as well as propose or establish connections between design ideas.

Unit 3: Design tools

The following is a list of design tools for generating solution designs from design ideas:

1. Data dictionaries.

Data dictionaries are used to design the data requirements of a software module. This includes, but is not limited to, variables, constants, objects and interface controls.

Represented using a table with the following fields included as a minimum:

- Name
- Data type

Additional fields may be included, as required, such as:

- Description
- Field size
- Validation rules
- Data sample

A sample data dictionary is found below:

Name	Data type	Description
strGivenName	String	User's given name
strFamilyName	String	User's family name
intAge	Integer	User's age (calculated based on the provided date of birth and rounded down)

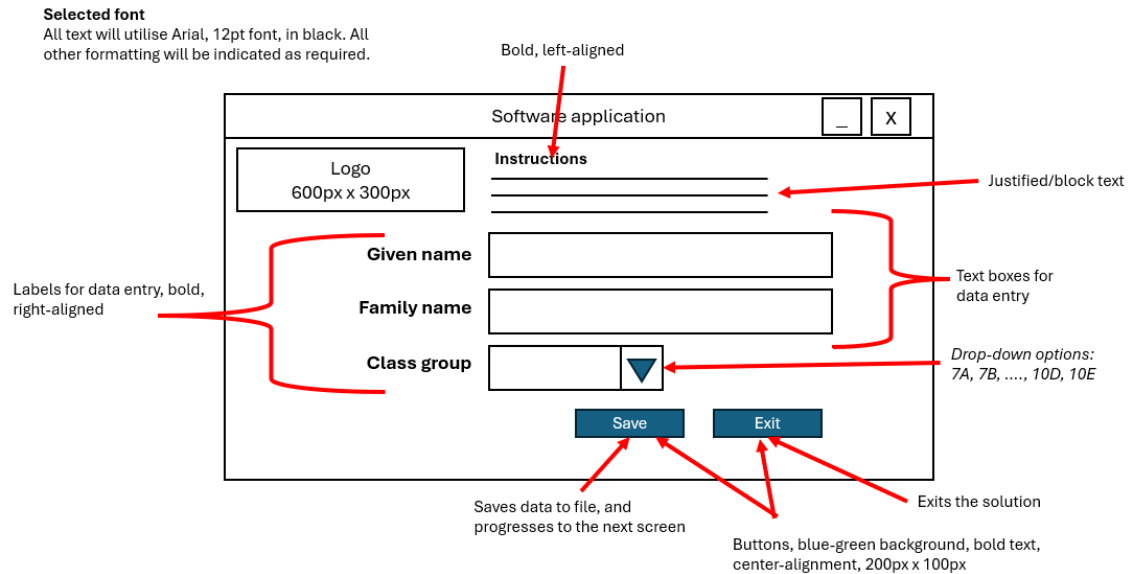
2. Mock-ups.

A mock-up is a detailed diagram that illustrates how a software module's interface will look.

Annotations are used to describe the appearance (font, text colour, text size, alignment/justification, images to be sourced, control properties) and the structure (required controls, functionality, default values, control size, control properties).

Annotations may also be used to explain how design principles and the characteristics of user experience have been incorporated, or to justify design choices that have been made.

A sample mock-up is found below:



3. Object descriptions.

An object description is used to design an object to be used within a software module.

Object descriptions include the name of the object, a list of the required attributes/properties (data) and a list of the required methods (behaviours) that are required to maximise the potential future use of the object.

The data type of each property/attribute may or may not be listed with the property/attribute.

Where required, parameters should be included in the description of methods.

A sample object description is found below:

Name	Dog
Properties/Attributes	Name (string) Breed (string) Age (integer) Colour (string)
Methods	bark() eat(food) wag_tail()

4. Input-process-output (IPO) charts.

IPO charts provide a brief overview of the inputs and outputs of a task, and outline the transformations (processes) required to generate the output/s from the input/s.

IPO charts are represented using a table, with columns for Input, Processes and Outputs.

Separate IPO charts should be used when multiple tasks are being represented.

A sample IPO chart is found below:

Input	Processes	Output
User uploads file containing a series of numbers	Calculate sum and average of the numbers	Display sum and average of numbers Output values to file

5. Pseudocode.

Pseudocode is a design tool for representing algorithms using a combination of structured English and common symbols.

Pseudocode utilises reserved words within algorithms.

Pseudocode conventions

Pseudocode blocks must commence with **BEGIN** and conclude with **END**.

Reserved words are represented within algorithms using bold text and capitalised.

Indentation is used within pseudocode to indicate hierarchy within an algorithm to enhance human readability. Control structures can be nested using appropriate indentation.

Values can be assigned to variables using the left-arrow (\leftarrow) symbol or the **SET** keyword.

Values can be input into the algorithm using the **READ** or **INPUT** keywords.

Values can be output from the algorithm using the **PRINT**, **RETURN** or **OUTPUT** keywords.

Selection control structures are represented in pseudocode using **IF-THEN**, **IF-THEN-ELSE**, **IF-ELSEIF** and **CASE/SWITCH** blocks.

Iterative/Repetition control structures are represented using **DO-WHILE**, **REPEAT-UNTIL**, **WHILE** and **FOR** loops.

Conditions can be joined using the **AND** and **OR** keywords.

Lines may or may not be numbered.

A sample block of pseudocode is found below:

BEGIN

$a \leftarrow 7$

$b \leftarrow 1$

REPEAT

$c \leftarrow a * b$

$a \leftarrow a - 1$

$b \leftarrow b + 1$

UNTIL $a < 4$

PRINT c

END

Unit 3: Principles of OOP

The following is a list of principles of object-oriented programming:

1. Abstraction.

Abstraction involves hiding the complex implementation details of a class and exposing only the necessary and relevant parts to users.

It simplifies interaction with classes/objects allowing users to focus on what a class/object does rather than how it does it.

An example of abstraction is found below:

Applying the process of abstraction to a Dog

Identify the most relevant properties/attributes of a dog. Ignore/omit any that are not immediately relevant to the situation. In this case, Name, Breed, Age and Colour are required.

Define common Dog behaviours. In this case, barking, eating and wagging it's tail are required to begin with.

Name	Dog
Properties/ Attributes	strName strBreed intAge strColour boolTrained
Methods	bark() eat() wag_tail()

2. Encapsulation.

Encapsulation is the practice of bundling the data (attributes) and methods (functions) that operate on the data into a single class.

It restricts direct access to some of a class/object's components, which helps to protect the integrity of the data and prevent unintended interference and misuse.

An example of encapsulation is found below:

Applying the process of encapsulation to the Dog class

Keep properties/attributes private (using access modifiers)

Define public methods to safely read or modify the properties/attributes

In this case, creating a get and set method for each property/attribute.

Ensure that the properties/attributes and methods that use the data are together in the same class.

Name	Dog
Properties/ Attributes	<code>private strName</code> <code>private strBreed</code> <code>private intAge</code> <code>private strColour</code> <code>private boolTrained</code>
Methods	<code>bark()</code> <code>eat()</code> <code>wag_tail()</code> <code>getName() / setName(name)</code> <code>getBreed() / setBreed(breed)</code> <code>getAge() / setAge(age)</code> <code>getColour() / setColour(colour)</code> <code>getTrained() / setTrained()</code>

3. Generalisation.

Generalisation is a process where shared attributes/properties are extracted from two or more classes and combining them into a generalised superclass.

This allows for the creation of a more abstract representation of common features, promoting code reuse and reducing redundancy.

A sample of generalisation is found below:

A developer has created a *Dog* class and a *Cat* class.

- A *dog* object has the same attributes and methods as the previous examples.
- A *cat* object will have a name, breed, age and colour, as well as purr, eat and catch mouse as methods.

After identifying the shared attributes and similar methods, the developer decides to create a *Pet* superclass.

The result is below:

Name	Cat (extends Pet)	Dog (extends Pet)	Pet
Properties/ Attributes		<code>private boolTrained</code>	<code>private strName</code> <code>private strBreed</code> <code>private intAge</code> <code>private strColour</code>
Methods	<code>purr()</code> <code>overrides</code> <code>make_noise()</code> <code>catch_mouse()</code>	<code>bark()</code> <code>overrides make_noise()</code> <code>wag_tail()</code> <code>getTrained()</code> <code>setTrained()</code>	<code>make_noise()</code> <code>eat()</code> <code>getName() / setName(name)</code> <code>getBreed() / setBreed(breed)</code> <code>getAge() / setAge(age)</code> <code>getColour() / setColour(colour)</code>

4. Inheritance.

Inheritance is a mechanism where a new class (subclass) inherits attributes and methods from an existing class (which in turn becomes a superclass). This allows the subclass to reuse code from the superclass, extend its functionality, and promote a hierarchical relationship between classes.

A sample of inheritance is found below:

Name	Shape	Circle (extends Shape)	Rectangle (extends Shape)
Properties/Attributes	private colour(integer) private bThick(integer)	private radius (floating point)	private length (floating point) private width (floating point)
Methods	draw() calcPerimeter() calcArea()	drawCircle() overrides draw() getDiameter() calcCircumference() overrides calcPerimeter() calcArea() overrides calcArea()	drawRectangle() overrides draw() calcPerimeter() overrides calcPerimeter() calcArea() overrides calcArea()

Units 3 and 4: Key legislation and frameworks

The following is a list of key legislation and frameworks.

1. Copyright Act 1968 (Cwlth)

For more information go to: Federal Register of Legislation:

<https://www.legislation.gov.au/C1968A00063/2019-01-01/text>.

2. Privacy Act 1988 (Cwlth)

- Unit 3 Area of Study 2: APP 1, 3, 6, 8, 9, 11
- Unit 4 Area of Study 2: APP 1, 6, 8, 9, 11

For more information go to: Federal Register of Legislation:

<https://www.legislation.gov.au/C2004A03712/2019-08-13/text>.

3. Privacy and Data Protection Act 2014

- Unit 3 Area of Study 2: IPP 1, 2, 4, 5, 7, 9, 10
- Unit 4 Area of Study 2: IPP 1, 2, 4, 5, 9

For more information go to: Your privacy rights:

<https://ovic.vic.gov.au/privacy/for-the-public/your-privacy-rights/>.

Additionally, the following key industry frameworks are to be studied by students in Unit 4:

4. Essential Eight

For more information go to: Essential Eight:

<https://www.cyber.gov.au/resources-business-and-government/essential-cyber-security/essential-eight/essential-eight-explained>.

5. Information Security Manual (ISM) Guidelines for Software Development

- Development, testing and production environments
- Secure software design and development
- Application security testing

For more information go to: Information Security Manual (ISM):

<https://www.cyber.gov.au/resources-business-and-government/essential-cyber-security/ism/cyber-security-guidelines/guidelines-software-development>.

Unit 4: Established and innovative approaches to software development

The following are a list of established and innovative approaches to software development:

1. The use of code repositories.

Code repositories play a key role in software development, facilitating version control, and fostering collaboration and transparency.

Code repositories also allow developers to further develop existing functionality or include new functionality or features, without affecting the main codebase. Changes to code are then merged after testing and review.

Examples of code repositories include, but are not limited to:

- Git
- GitHub
- BitBucket
- SourceForge

2. Application programming interfaces (APIs) and libraries.

APIs and libraries are key tools within software development.

APIs

APIs are interfaces that allow different systems and solutions to interact and communicate with each other seamlessly.

Examples of established and innovative approaches to software development with APIs include, but are not limited to, REST, SOAP, Operating system APIs, GraphQL, webhooks, serverless APIs and API gateways.

Libraries

Libraries are collections of pre-written code that developers can reuse and reference to perform specific tasks and functions. They enable developers to focus on developing their solution, whilst relying on libraries for common tasks.

Most programming languages come with standard libraries for input/output and data manipulation, and there are also open-source third-party libraries (such as NumPy) available.

Examples of established and innovative approaches to software development with libraries include, but are not limited to, modularisation, micro-libraries, AI-generated libraries, and community-driven libraries.

Challenges that arise from the use of both APIs and libraries include:

- complexity
- dependencies on API or libraries
- security risks that arise from APIs and outdated libraries
- versioning and compatibility.

3. Artificial intelligence-based (AI) assistants.

AI-based assistants are increasingly becoming integrated into development environments. They utilise a combination of artificial intelligence, natural language processing and machine learning to observe developer actions and behaviours, as well as respond to prompts from developers.

The benefits of using AI-based assistants include:

- increased productivity and efficiency
- potential to support problem-solving and innovation
- reduced development costs
- code generation and suggestions
- automated debugging and testing, including automated generation of test data and test cases
- automated generation, maintenance and improvement of internal documentation.

The challenges that developers should be aware of when using AI-based assistants include:

- lack of accuracy and context awareness
- over-reliance on AI-assistants
- cost of integration
- accidental introduction of security vulnerabilities
- copyright and intellectual property theft
- bias in training data
- environmental impact.

Examples of AI-based assistants include, but are not limited to GitHub Copilot, Tabnine, Synk, TensorFlow, Codeium.