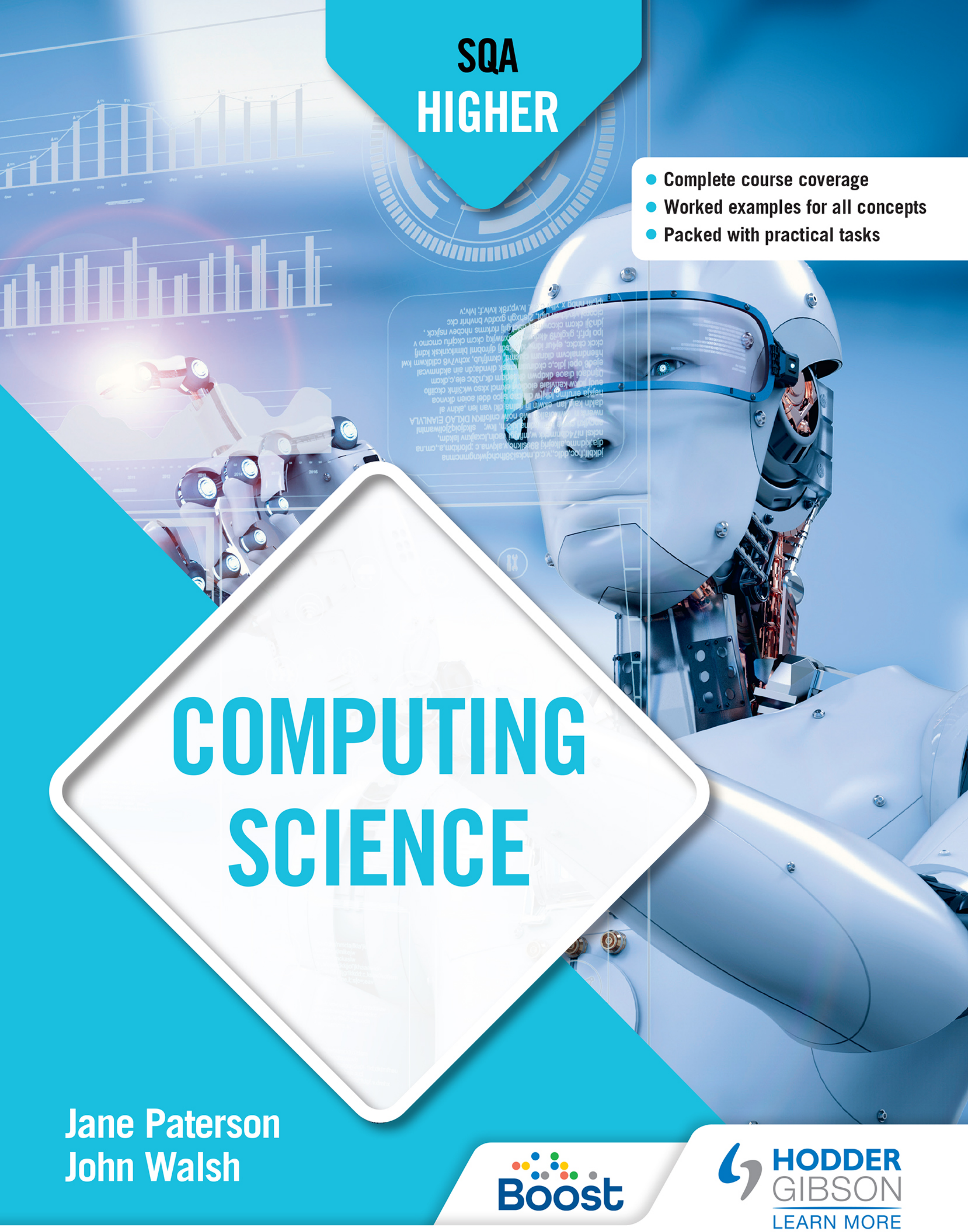
SQA

HIGHER

- Complete course coverage
- Worked examples for all concepts
- Packed with practical tasks



COMPUTING SCIENCE

Jane Paterson
John Walsh

Boost

HODDER GIBSON
LEARN MORE

HIGHER

COMPUTING SCIENCE

Jane Paterson &
John Walsh

DYNAMIC LEARNING

HODDER GIBSON
AN HACHETTE UK COMPANY

The Publishers would like to thank the following for permission to reproduce copyright material.

Photo credits

Images reproduced by permission of: **p.43** © Stefan Szeider, used under a CC-BY-SA 4.0 licence; **p.265** Newscast/Getty Images; **p.295** © Carlos Jones, used under a CC BY 4.0 licence; **p.302** *t* SamJonah/Alamy Stock Photo, *tr* Iconic Cornwall/ Alamy Stock Photo, *b* PhotoEdit/Alamy Stock Photo; **p.303** bl Jeffrey Blackler/Alamy Stock Photo, *br* dpa picture alliance/ Alamy Stock Photo; **p.305** Dawson Images/Alamy Stock Photo; **p.307** Finbarr Webster/Alamy Stock Photo.

Dilbert cartoons on pp. 3, 7, 78, 255, 323 used by permission of ANDREWS MCMEEL SYNDICATION. All rights reserved.

Acknowledgements

To Craig and Laura.

To Helen, Peter John, Mary, Sarah, Siobhan, Cecilia, Orla, Poppy, Michelle and Erin.

Orders: please contact Hachette UK Distribution, Hely Hutchinson Centre, Milton Road, Didcot, Oxfordshire, OX11 7HH. Telephone: +44 (0)1235 827827. Email education@hachette.co.uk
Lines are open from 9 a.m. to 5 p.m., Monday to Friday. You can also order through our website: www.hoddereducation.co.uk.

ISBN: 978 1 5104 8381 1

© Jane Paterson and John Walsh 2021

First published in 2021 by

Hodder Gibson, an imprint of Hodder Education
An Hachette UK Company
211 St Vincent Street
Glasgow, G2 5QY

www.hoddereducation.co.uk

Impression number 5 4 3 2 1

Year 2025 2024 2023 2022 2021

Cover photo ©phonlamaiphoto - stock.adobe.com

Illustrations by Integra Software Services Pvt. Ltd., Pondicherry, India

Typeset by Integra Software Services Pvt. Ltd., Pondicherry, India

Printed in Italy

A catalogue record for this title is available from the British Library.

Contents

Answers available online at https://www.hoddergibson.co.uk

Software design and development

Chapter 1 Development methodologies

This chapter looks at and compares two different software development methodologies.

The following topics are covered:

- Describe and compare the development methodologies:
  - iterative development process.
  - agile methodologies.

Development methodologies

When creating new programs or applications (apps), software developers follow a development methodology to design the software from start to finish.

The first of these is the iterative development process.

The iterative development process

The iterative development process is better known as the Waterfall development methodology, Waterfall life-cycle or Waterfall model and follows a traditional, linear approach which consists of seven stages as shown in Figure 1.1 on the next page. This sequence of steps, beginning with analysis, is known as the software development process. It is iterative in nature, meaning that steps can be revisited at any point in the life-cycle of the development process (multiple times if necessary) if new information becomes available and changes need to be made or errors discovered.

Looking at and understanding a problem is called ‘analysis’. In the Waterfall model, the software developers need to know up front and in detail exactly what the client wants the software to do. The design phase involves working out a detailed series of steps to solve the problem. Once a solution to a problem has been worked out, it needs to be turned into instructions for the computer (a program). This is implementation. The program must then be tested to make sure that it does not contain any mistakes which would prevent it from working properly. A description of what each part of the program does, i.e. documentation, should also be included. Evaluation is the process which measures how well the solution fulfils the original requirements. Maintenance involves changing the program, often quite some time after it has been written.

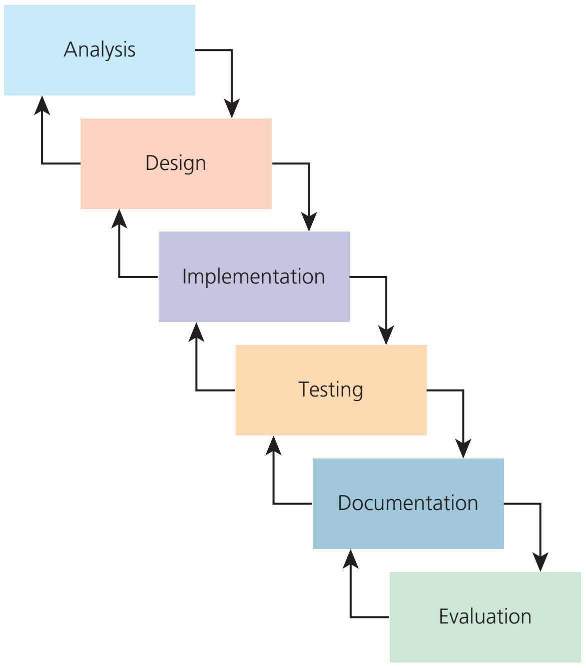


Figure 1.1 An iterative development process (the Waterfall model)

Documentation is needed at each stage of the iterative development process.

- Analysis: the documentation at this stage consists of the **software specification**. This is important because it is the basis of all of the remaining stages of the software development process. It is usually a legally binding document (see Chapter 2 for more information).
- Design: the documentation consists of the description of the program design in an appropriate design notation and the design of the user interface. This description is important because it is the 'bridge' between the software specification and the code.
- Implementation: the documentation at this stage is the program listing(s), complete with internal commentary. This is important because it explains the purpose of each part of the code, and therefore eases the process of maintenance.
- Testing: the documentation at this stage includes the test plan and the results of testing. This is important because it demonstrates whether or not the program does what it was designed to do.
- Documentation: this stage has the technical guide and the user guide. These are important because they explain how to install and operate the software.
- Evaluation: the acceptance test report, the results of evaluation against suitable criteria. This is important because it means that the program has been written to the satisfaction of the client and therefore the software company can be paid for their work.
- Maintenance: documentation at this stage is a log of changes made to the program code, together with the date and the new version number of the program. This is important because it will be updated constantly throughout the life of the software in order to inform programmers about earlier changes that have been made. Maintenance is beyond the scope of Higher Computing Science but is included here for completeness.

Agile methodologies

Agile methodologies belong to a wider category of rapid application development methodologies. The process of designing a program using Agile development is to produce working software quickly so that the client can test it and then give feedback, at which point it can be altered and refined as required. It works on the principal of delivering the software in small increments instead of all at once.

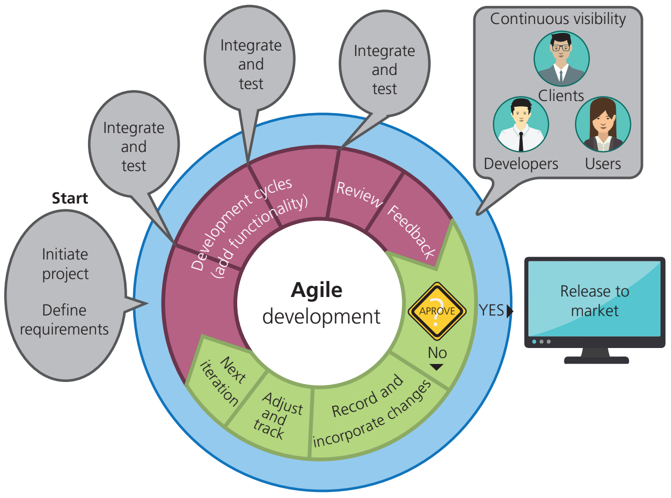


Figure 1.2 Agile methodology

Each project is broken down into what the client needs, otherwise known as '**user stories**'. Each of these user stories is part of the overall Agile plan and encourages the software developers to talk to their clients about what they would like to see in their software. The plan should include an idea of how long each of these stories will take to develop. These are prioritised in conjunction with the client (so that the important requirements are delivered first) and included in the plan for the development of the software.

Teams will normally work in short '**sprints**' to produce working prototypes which can be tested by the client. A sprint is a short, fixed time period (typically two weeks) during which planning, analysis, design, implementation and testing are completed. At the end of the sprint, the prototype is ready to be tested by the client (acceptance testing). The client will then be able to give regular feedback and alterations can be made when required. The plan itself can be updated when and where required.

EXTREME PROGRAMMING
I CAN'T GIVE YOU ALL OF THESE FEATURES IN THE FIRST VERSION.
AND EACH FEATURE NEEDS TO HAVE WHAT WE CALL A "USER STORY."
OKAY, HERE'S A STORY: YOU GIVE ME ALL OF MY FEATURES OR I'LL RUIN YOUR LIFE.
www.dilbert.com
scottadams@aol.com
© 2002 United Feature Syndicate, Inc.

Figure 1.3 User stories

Often by the time the project comes to an end, all the client's requirements may not have been built, and there will be one of two outcomes. Either the client may opt for software which has fewer features or the software development team may ask for more money to complete the software as originally required.

The development of software using Agile methodology is a constant process of analysis, design, implementation and testing. This means it is iterative as each of the stages will be visited multiple times as the software is refined. When what can be delivered differs from was originally requested, Agile allows plans to be changed. This is known as 'adaptive planning'.

Agile methodology vs. Waterfall model

The Waterfall model is an older approach to developing software and is based on the idea that each of the phases of the development life-cycle are discrete parts which should be completed in turn. It gives both client and developer a clear path as to how each project should progress.

The main emphasis of Agile development is speed. Project goals are determined quickly and all phases are iterated continuously rather than individually, so that the software is developed and adapted quickly as shown in Figure 1.4.

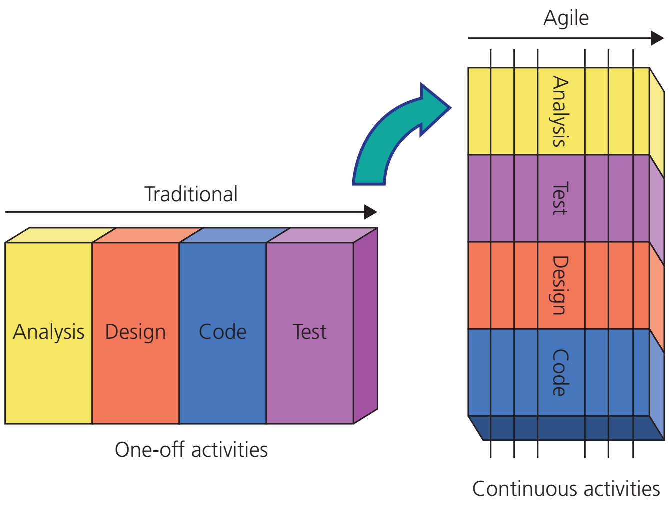


Figure 1.4 Agile methodology vs. Waterfall model

Advantages of the Waterfall model

The Waterfall model is best suited to larger projects and large teams of developers with a long lead time. As the client is usually less involved during development, they need to know exactly what they want at the start of the process because it forms part of the legally binding agreement. Due to its tendency to focus on quality of software over speed of development, the software is tested more fully and the software produced tends to have less bugs. Projects that follow the Waterfall model are generally finished on time and within the set budget. Milestones set at the start of the project give both developer and client an easy way to track its progress.

Disadvantages of the Waterfall model

The linear setup of the Waterfall model is its main failing as there can be no deviation from the plan once it has commenced. Because requirements are only sought at the start of the project, clients are not continuously involved. So, if a client does not have a clear idea about what the software should do and their needs change over the course of the project, there is a strong possibility that the software delivered will not then meet their requirements.

Once the software has reached the testing phase, changes can be difficult to make. If changes then do need to be made, this can take more time and will cost more money.

To try to mitigate this, client feedback can be built in to the phases of the life-cycle so that changes can be made.

Advantages of Agile methodology

Agile methodology is best suited to smaller projects, like creating (or frequently updating) apps, with smaller teams of developers. The client is involved at all stages of the development of the software and feedback is sought constantly. This means that the software developed is more likely to be exactly what the client wants even if they were not completely certain of the requirements at the start of the project. It also allows the client an element of flexibility in that, if they change their minds or would like other features included, adding them does not present too much of a challenge to the developers. Because of the constant iterative nature of Agile methodology, improvements to the software can be incorporated after each cycle.

The main focus of Agile development is to deliver software at speed, which makes it perfect for projects which are required quickly.

Disadvantages of Agile methodology

At the start of the Agile process, there tends not to be a legally binding agreement due to the changing nature of client's requirements. It is also not suited to large projects and large teams. The client needs to be prepared to spend a large amount of time being involved with the project as their involvement is required throughout.

Strict sprint deadlines can be a huge disadvantage and can result in a project not being completed. Either the customer pays more for the project to be completed, or they simply have to accept what has been developed, however much reduced in scope. The frequent updates can be difficult to track and need strict control over the version numbers given to each iteration or update.

CHECK YOUR LEARNING

Now answer questions 1–10 below

QUESTIONS

1 State what is meant by the term 'iteration'.
2 State why a development stage might need to be revisited.
3 Describe each stage in the software development life-cycle.
  a) Analysis
  b) Design
  c) Implementation
  d) Testing
  e) Documentation
  f) Evaluation
4 State the category of methodologies to which Agile belongs.
5 Describe what is meant by the term 'user story'.
6 How does the developer prioritise the order in which to develop the user stories?
7 Describe how working prototypes are produced by the developers.
8 a) Describe a situation where a client's requirements may not have been met.
  b) What two options may the client have if the requirements have not been met?
9 State what is meant by 'adaptive planning'.
10 Complete the table of iterative (Waterfall method) vs. Agile methodology. The first line has been completed for you.

Iterative	Agile
Suited to large software projects	Suited to small software projects

KEY POINTS

- Two types of development methodologies are iterative (Waterfall) and Agile.
- The Waterfall model follows a traditional, linear approach.
- The iterative methodology means that steps can be revisited at any point in the life-cycle.
- Analysis is looking at and understanding a problem.
- Design is working out a series of steps to solve a problem.
- Implementation is turning a design into a computer program.
- Testing makes sure that a computer program does not contain any mistakes.
- Documentation is a description of what each part of the program does.
- Evaluation ensures that the software fulfils the original software specification.
- Agile methodologies belong to a wider category of rapid application development methodologies.
- Agile works on the principal of delivering the software in small increments instead of all at once.
- Each project is broken down into user stories.
- User stories are prioritised by the client and developer, so that the important requirements are delivered first.
- Teams work in short sprints to produce working prototypes which are tested by the client.
- A sprint is a short, fixed time period typically lasting two weeks.
- The development of software using Agile methodology is a constant process of analysis, design, implementation and testing.
- Adaptive planning allows the requirements to be changed on request.
- Iterative is best suited to large projects which require a large team of developers.
- Analysis in the iterative process produces a legally binding contract.
- Software produced using an iterative method is usually high quality and bug free.
- Iterative projects tend to finish on time and within budget.
- The client has limited involvement during the entire iterative process.
- Changes can become more difficult and expensive as the project progresses in iterative projects.
- Agile is best suited to small software projects and requires a small team of developers.
- There is no legally binding contract in the Agile process.
- The client is involved throughout the Agile process.
- Changes can be easily made as the Agile process progresses.
- Agile projects can run over time, are not always completed and can cost more money to finish.
- Version control is important to keep track of updates.

Chapter 2 Analysis

This chapter considers how to analyse a problem to help create the software specification.

The following topics are covered:

- Identify the:
  - purpose
  - scope
  - boundaries and functional requirements of a problem that relates to the design and implementation at this level, in terms of:
    - inputs
    - processes
    - outputs.

Analysis

Analysis first of all involves reading and understanding a problem. If you are set a problem in class, you should read the problem several times and think about it carefully. It often helps to write out the problem in your own words. Sometimes the problem contains parts which are not very clear, and you will have to make some assumptions about what you think is meant by these parts of the problem.

Eventually you will get to the stage where you will be able to create a precise software specification. The software specification should contain what the software is supposed to do but does not indicate how this is to be achieved.

It is very important that the software specification is correct, since mistakes at this stage can be very costly to put right later on in the software development process. The software specification is a clear unambiguous statement of the problem and forms the basis of a legal contract.

Here is an example of what could happen if a software specification is not correct:

A farmer, as can be seen in commissioned a software company to write a database program to store details of his herd of cattle. The maximum size was to be 1000 records. Cow number 1000 had a calf. The farmer entered the new calf's details into the program and the program crashed. Who was to blame? Was it the software company for not anticipating that the herd of cattle would increase in size or was it the farmer's fault for agreeing to the maximum size of 1000 records? In any case, if the program matched the software specification correctly, then the software company would still be entitled to be paid for their work.

Figure 2.1 The software specification forms the basis of a legal contract between the client and the software company

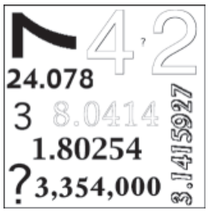


Figure 2.2 Numbers

The analysis phase can be broken down into a series of discrete sections.

Purpose

The purpose of the problem is stating what the software should do once completed. The detail for this is written in the scope, boundaries and functional requirements.

Scope

The scope should state clearly and concisely what the software must do, i.e. specific project goals. It should also state the start and end dates, cost, deliverables (including but not limited to, the design of the software, the software itself, results of testing and the test plan), milestones and deadlines. A milestone is a completed step in a software development project. It means that developers know what work is due and by which date.

If the project is poorly managed and changes are constantly made, then **scope creep** can occur. This usually happens if the scope is not properly defined or documented. It is also known as 'requirement creep', 'function creep' or 'kitchen sink syndrome'.

The scope should also define the boundaries of the software.

OUR PROJECT SCOPE HAS VASTLY EXPANDED, SO WE'LL ALL NEED TO PULL TOGETHER AS A TEAM.
IS IT TOO LATE FOR ME TO BAIL OUT BEFORE THIS PROJECT BECOMES A BLIGHT ON MY CAREER?
I ALREADY PUT THE TEAM PHOTO ON OUR COFFEE CUPS.
GAAA!!!
www.dilbert.com
scottadams@aol.com
11-17-06 ©2006 Scott Adams, Inc./Dist. by UFS, Inc.

Figure 2.3 Scope creep

Boundaries

Boundaries help to clarify what the software should and should *not* do. They should also state any assumptions that are being made about what the client requires.

WORKED EXAMPLE

Consider the following simple problem outline:

Average problem: Write a program which calculates and displays the average of a set of numbers.

This could hardly be described as a precise software specification. If you, as a programmer, were given this task you would need to ask some questions before you could begin to design a solution.

Questions you may ask about the Average problem:

- How many numbers are in the set?
- What is the maximum value of a number?
- Are numbers to be whole numbers (integers) or numbers with a fractional part (real numbers)?
- To how many significant figures should the average value be displayed?
- How are numbers to be obtained as input to the program (i.e should the program ask the user to enter each number every time the program is run)?
- Are any of the numbers entered to be stored after the program is complete?
- What output device(s) are to be used (i.e. display on screen or hard copy to printer)?

If there is no one available to ask for clarification of a problem, then you should examine the problem carefully and write down some assumptions.

For the Average problem, these assumptions might be:

- The maximum amount of numbers is 10.
- The minimum value of a number is 1.
- The maximum value of a number is 100.
- All the numbers are whole numbers, apart from the average, which has to be displayed correct to two decimal places.

Putting these assumptions together with the original Average problem, would give a more precise description of the scope and boundaries.

Functional requirements

The functional requirements describe how the software should function or perform. They involve identifying the problem **inputs**, the **process**(es) and the problem **outputs**. The best way to do this is to create a table with the three headings: Input, Process and Output and use the information given in the problem as shown in Table 2.1.

Input	Process	Output
A maximum of 10 numbers (integers), within a range of 0 to 100.	Calculate the average (sum the numbers and divide the total by the amount of numbers).	Display the average value (a real number, correct to two decimal places).

Table 2.1 Example table for the Average problem

CHECK YOUR LEARNING

Now answer questions 1–3 below

QUESTIONS

1 Why does the software specification have the status of a legal contract?

2 Give an example of what could happen if the software specification was not correct.

3 Analyse the following problem outlines and produce a precise software specification by reporting on:
- the purpose of the software.
- scope and boundaries of the software.
- functional requirements of the software.

You should ensure that you include any assumptions that need to be made and identify the problem inputs, process(es) and outputs as part of the functional requirements.

a) Write a program which will ask the user for two numbers and give their sum (+), product (*) and quotient (/).

b) Write a program which will take in a word of up to 15 letters and display it on the screen backwards.

c) Write a program which will only allow the user to enter words consisting of five letters and display them on the screen.

d) Write a program which will calculate how fast a cyclist is travelling if you input their time taken to travel 100 metres.

e) Write a program which will accept a temperature in degrees Celsius and output the temperature in either degrees Fahrenheit or degrees Kelvin as required by the user. (Kelvin = Celsius + 273.15; deg F = 9/5 Celsius + 32)

f) Write a program which will capitalise the first character of a word entered by the user (i.e. john to John).

g) Write a program which will calculate the length of one side of a right-angled triangle if the lengths of the other two sides is input (Pythagoras).

h) Write a program which will change a student's percentage test mark into a letter grade (A–E).

i) Write a program which will calculate the area of a triangle if you enter its base and height. (area = 1/2 base × height)

j) Write a program which will take in a message and display it on the centre of the screen.

KEY POINTS

- Analysis involves reading and understanding a problem.
- The purpose of the problem is what the software should do once completed.
- The software specification should contain what the software is supposed to do but does not indicate how this is to be achieved.
- The scope should state clearly and concisely what the software must do.
- Boundaries help to clarify what the software should and should *not* do.
- Boundaries should also state any assumptions that are being made about what the client requires.
- The functional requirements describe how the software should function or perform.
- The functional requirements should define inputs, processes and outputs to the program.
- Inputs should clearly state what data must be provided for the program to function.
- Processes should determine what has to be done with the data entered.
- Outputs should show the results of the program when it is run.

Chapter 3 Design

This chapter considers how to turn the software specification created at the analysis stage into a design for a program.

The following topics are covered:

- Identify the data types and structures required for a problem that relates to the implementation at this level.
- Read and understand designs of solutions to problems at this level, using the following design techniques:
  - pseudocode
  - structure diagrams.
- Exemplify and implement efficient design solutions to a problem, using a recognised design technique, showing:
  - top-level design
  - the data flow
  - refinements.
- Describe, exemplify and implement user-interface design, in terms of input and output, using a wireframe.

Simple data types and structures

When designing the solution to a problem, the data types and structures to be used in the solution should be identified so that the software developer knows the type of data each variable will store.

The data types stored by a program may be a number, a character, a string, a date, an array, a sound sample, a video clip or indeed, any kind of data. Some high-level languages, such as C++, allow programmers to specify their own data types; these are called user-defined data types.

Some of the more important data types are listed below:

- **Alphanumeric data**: may include letters, digits and punctuation. It includes both the character and string data types. **Character data** is a single character represented by the character set code, e.g. ASCII (American Standard Code for Information Interchange). **String data** is a list of characters, e.g. a word in a sentence.
- **Numeric data**: may consist of **real data** or **integer data**. Real or float data includes *all* numbers, both whole and fractional. Integer data is a subset of real data which includes only whole numbers, either positive or negative.
- **Date data**: is data in a form representing a valid date, e.g. 29/2/2020 is valid date data, 30/2/2020 is not.
- **Boolean** or **logical data**: may only have two values, *true* or *false*. Boolean data is used in a program's control structures.
- **Sample data**: consists of digitally recorded **sound data** (e.g. MP3) and **video data** (e.g. a video clip MPEG). These are complex data types which contain enough data to allow a subprogram or application to reproduce the original data.

One factor which may influence a programmer's choice of software development environment is the range of data types available. For example, C++ has at least six different numeric data types, whereas some versions of the BASIC language may only have the two numeric data types described above.

Data structures include arrays, records and arrays of records. More detailed information on data structures can be found in Chapter 4.

WORKED EXAMPLE 1

Take the Average problem we discussed in Chapter 2 and Table 2.1 from the Functional Requirements. We would need to consider which data types and structures would be suitable for storing the data in this problem.

We know from the input part of the functional requirements that 10 numbers are required. If we assume that those 10 numbers are whole numbers, then we could say that the data type for input is integer.

A running total would be required total the numbers and since the input is an integer so too must be the total.

Output for the program is a little easier as it would always be a real (float) number unless the output is a multiple of 10.

Data types and structures

number, total: integer

average: real (float)

Designing the solution to a problem

Once you have a precise software specification, then you can begin to design your solution to the problem.

In a normal problem-solving situation, you should always ask these questions:

- Is writing a program the best way to solve this problem?
- Can it be solved more easily another way?

Returning to the Average problem again, if the average is to be found for only *one* set of numbers, i.e. as a one-off, then it would probably be much more efficient to use a calculator and work out the average that way, rather than entering the numbers into a computer. However, if you have to work out many different averages for lots of sets of numbers, then it would be worthwhile using a computer software solution.

If you decide to use a computer to solve the problem, then you should begin by looking at the highest level of software you have available.

For instance, would it be possible to solve this problem using a general-purpose application package such as a spreadsheet or a database rather than by writing a program in a high-level language?

How could you use a spreadsheet to solve the *Average Problem*?

Answer: create a new spreadsheet document; enter the set of numbers into a column or a row and enter the formula =AVERAGE(cell range) into an unused cell.

However, in this unit on software design and development, you are concerned with producing a computer solution to a problem using some kind of programming language. Your approach to problem solving should therefore take account of this.

Program **design** is the process of planning the solution. The design of the program is very important for its success. Time spent at this stage is very worthwhile and can reduce the chance of errors appearing later on in the solution.

Consider the many home improvement, DIY and gardening make-over programs which appear on television. Their success appears magical as a team of 'experts' descends on a person's home and completes the conversion in two or three days. Despite how casual it may appear, all these transformations are planned out to the last detail. The 'experts' have all visited the homes weeks, if not months, in advance, and have gone away and drawn up detailed plans for the conversion.

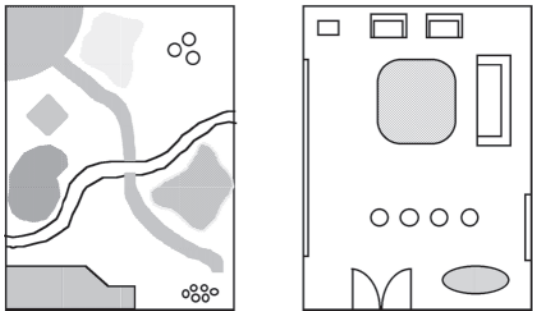


Figure 3.1 Garden plans and room plans

It is just the same with programming: the more time you spend thinking about and planning the design of the program, the less time you will spend wondering why your program does not work as it should.

Modular design

NOTE

It is not within the scope of this textbook (nor Higher Level Unit) to cover all the possible methods of program analysis and design which may be used as part of the software development process.

Modular design is a method of organising a large computer program into self-contained parts called **modules**, which can be developed simultaneously by different programmers or programming teams. Modules are specially constructed so that they can be designed, coded and maintained independently of one another. Some modules may be linked to other modules and some may be separate programs. **Top-down design** and **bottom-up design** are both forms of modular design.

Top-down design

Top-down design involves looking at the whole problem (top) and breaking it down into smaller, easier to solve, sub-problems. Each sub-problem may be further sub-divided into smaller and simpler sub-problems (modules). This process of breaking down sub-problems into yet smaller steps is called **stepwise refinement**. Eventually the stage is reached where each sub-problem can no longer be broken down and the refinement process comes to a halt. At this point each small step can be translated into a single line of program code.

Figure 3.2 Breaking down a problem …

When stepwise refinement is complete, then you have created an **algorithm**, which is a sequence of instructions that can be used to solve a problem.

Bottom-up design

The bottom-up method of designing a solution to a problem begins with the lowest levels of detail and works upwards to the highest level of the idea. It seems strange to think that it is sensible to work towards something without really knowing what that something is before you begin. However, using a bottom-up design approach means writing modules or procedures first. This approach is sometimes called '**prototyping**', where you construct a procedure separately before joining it together with the rest of a program.

Design techniques

The way of representing the program design or algorithm is called the **design technique** (or **design notation**). The programmer has a choice of design techniques. Common design techniques include drawing a **flow chart**, a **structure diagram** or writing **pseudocode**. Some design techniques use graphical objects such as icons to represent the design of a program. However, in Higher Computing Science, we look only at pseudocode and structure diagrams.

Pseudocode

Pseudocode is the name given to the language used to define problems and sub-problems before they are changed into code in a high-level computer language. Pseudocode uses ordinary English terms rather than the special keywords used in high-level languages. Pseudocode is therefore language independent.

Here is some pseudocode showing part of the design of one possible solution to the Average problem. This pseudocode shows the top-level design with stepwise refinements.

Algorithm

```
1 initialise
2 take in numbers
3 calculate average value
4 display average value

Refine sub-problem 2:

2.1 loop REPEAT
2.2   add one to counter
2.3   ask user for a number
2.4   take in a number
2.5 UNTIL counter equals amount required

Refine sub-problem 2.4:

2.4.1 loop REPEAT
2.4.2   get number from user
2.4.3   IF number is outwith range THEN
        display error message
2.4.4 UNTIL number is within range
```

Pseudocode is very useful when you are programming in languages like LiveCode or Python, because it fits in neatly with the structure of the code. The main steps in the algorithm relate directly to (in fact, become) the main program, the refinements of each sub-problem become the code in the procedures. (See Chapter 6 for more examples of pseudocode.)

WORKED EXAMPLE 2

Implementation

Suppose that your chosen software development environment is a high-level programming language such as Python. Let's look at how you might choose to implement part of the solution to the Average problem in Python.

Design (Pseudocode)

```
2.1 loop WHILE not equal to amount
    required
2.2   add one to counter
2.3   ask user for a number between 1
      and 100
2.4   take in a number between
      1 and 100
```

Python (actual program code)

```
def takeInNumbers():
  while counter != amount_required:
    counter= counter+1
    print('Please enter a number
    between 1 and 100')
    number=checkInput(number)
```

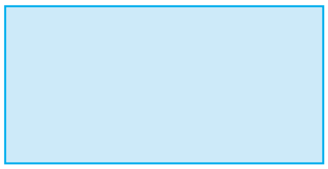
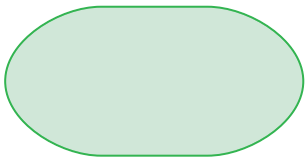
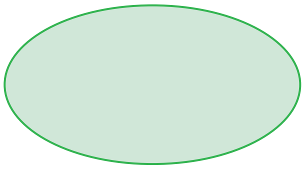
Pseudocode is useful for representing the design of the types of problem that you are likely to face in this unit.

In a more complex professional programming situation, or in a much larger project, structure diagrams may be more appropriate.

Structure diagrams

Structure diagrams use linked boxes to represent the different sub-problems within a program. The boxes in a structure diagram are organised to show the level or hierarchy of each sub-problem within the solution. In general, structure diagrams follow a left to right sequence.

Table 3.1 shows the function of each symbol in a structure diagram.

Symbol	Name	Description
	Process	This represents an action to be taken, a function to run, or a process to be carried out, e.g. a calculation.
 or 	Loop	The loop symbol indicates that a process has to be repeated either a fixed number of times or until a condition is met.

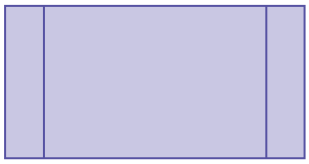
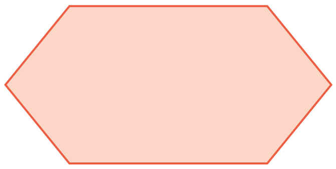
Symbol	Name	Description
	Pre-defined process	This symbol describes a process that contains a series of steps. It is most commonly used to indicate a sub-process or subroutine but could also indicate a pre-defined function like the random number function.
	Selection	This symbol shows that there may be different outcomes depending on user input or the result of an earlier process.

Table 3.1 Structure diagram symbols

Figure 3.3 shows one possible design for the Average problem from Chapter 2.

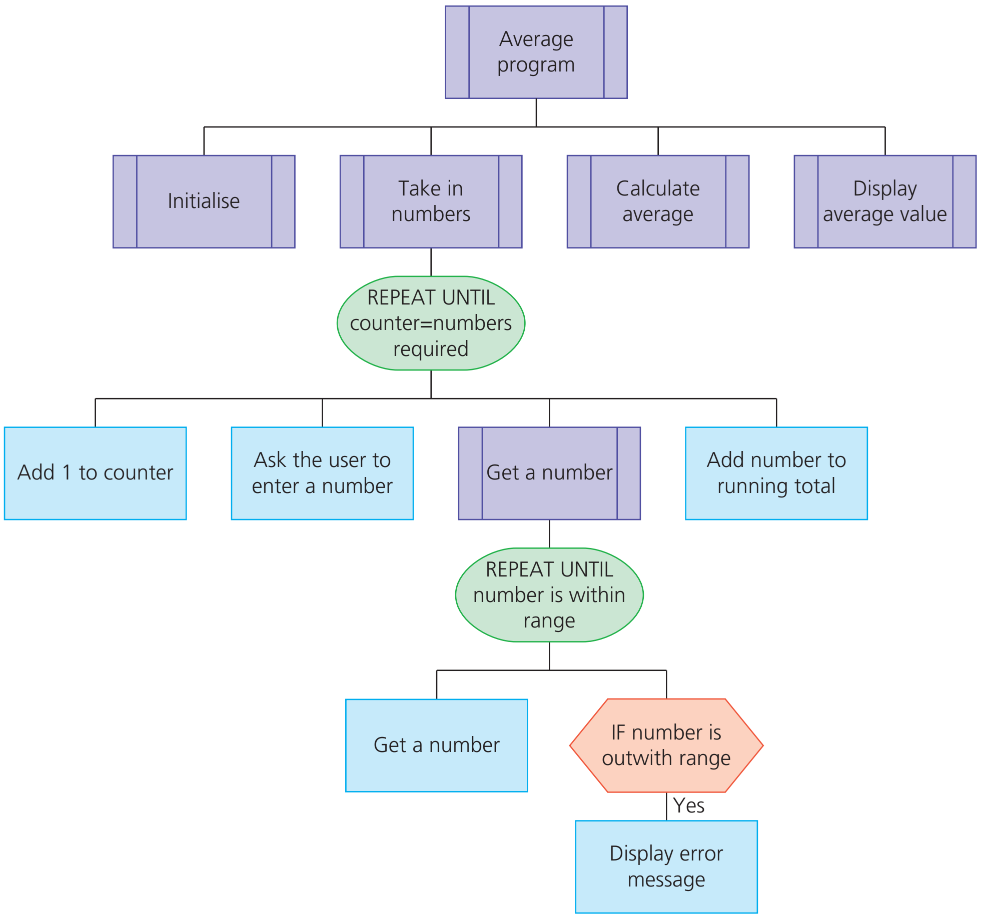

Average program
Initialise
Take in numbers
Calculate average
Display average value
REPEAT UNTIL counter=numbers required
Add 1 to counter
Ask the user to enter a number
Get a number
Add number to running total
REPEAT UNTIL number is within range
Get a number
IF number is outwith range
Yes
Display error message

Figure 3.3 Structure diagram

Data flow

Giving some indication of the flow of data between modules of your program is important. Some design techniques allow data flow to be shown clearly. Pseudocode uses the terms in:, out: and in-out: to represent the flow of data used in subprograms. For Higher, we will only use in: and out:. Structure diagrams use up and down arrows to indicate data flow into and out of subprograms.

Trust highly experienced teachers and authors Jane Paterson and John Walsh to guide you to success in Higher Computing Science.

This is the most comprehensive resource available for this course, brought to you by Scotland's No. 1 textbook publisher.

▶ Gain in-depth knowledge of the four areas of study (Software Design and Development, Database Design and Development, Web Design and Development, Computer Systems) with clear explanations of every concept and topic.

▶ Understand advanced concepts and processes as hundreds of examples throughout the book show the theory in action.

▶ Build the skills of analysis, design, implementation, testing and evaluation that are required for success in both the exam and the assignment.

▶ Apply the knowledge and skills developed through the course to a variety of practical tasks and end-of-chapter 'check your learning' questions.

▶ Use computing terminology confidently and accurately by consulting a detailed glossary of all key terms and acronyms.

About the authors

Jane Paterson began her teaching career in 1991 and has been an SQA marker for over 25 years. She has written a variety of support materials for Computing Science courses and has worked with Hodder Gibson since 2012.

John Walsh taught in Ayrshire schools for 38 years and has held a wide variety of SQA appointments. He has written nine textbooks and has been a Hodder Gibson author since 1994.

HODDER GIBSON

t: 01235 827827

e: education@hachette.co.uk

w: hoddergibson.co.uk

ISBN 978-1-5104-8381-1

9 781510 483811

FSC