NEA Companion for A level OCR Computer Science

Contents

Teacher's Introduction

IMPORTANT – please read before using this resource

This resource is intended to supplement your teaching only. As with all Non-Exam Assessment (NEA) materials it is the teacher's responsibility to decide what level of support is appropriate for their students and in accordance with the rules from the exam board. For example, you may simply wish to read this material to better inform yourself. Alternatively, you may consider whether it is appropriate to distribute some of the material to students for reference.

The resources here are provided as one experienced teacher's interpretation of the specification. The author does not have any special knowledge of what to expect on any particular assessment.

All exemplar material in this resource is based on entirely original, fictitious scenarios. Any possible resemblances to any future task released by OCR or any other exam board is co-incidental. However, we remind you that it is the teachers' responsibility to decide how this resource can be used to support your students.

This guide has been produced to provide clarity for teachers and students who are embarking on the OCR A Level Computer Science non-exam assessment (NEA), first assessment from 2017. This component is worth 20% of the overall A Level.

The nature of this part of the course is such that there is a great deal of freedom and flexibility in terms of what project students might choose. In my own experience, students who are not used to such breadth of choice can become overwhelmed, resulting in decisions being delayed or not really being made at all. The 'project ideas' section in this guide can be used as a starting point for brainstorming and discussions to allow everyone to see what options they genuinely have.

Although this guide can be used as a reference source, to look up errant facts as they are needed, it would be more useful to use it at the beginning and end of each phase of the project. Before the analysis begins, for example, students could be asked to read through that section and to identify the key requirements or the key pitfalls of that phase. Once the analysis is complete, this guide could be used as the basis for reviewing work, and there are checklists included to that effect. By the time their work is due, each student should be intimately familiar with the mark scheme, and their mark should come as no surprise to them.

However you see fit to use this resource to better equip your students for the NEA, I'm confident it will prove invaluable, and I wish you and your students the very best in this most rewarding part of the course.

R Lee, February 2020

Choosing a Project

Key considerations

Although you have a huge amount of freedom when it comes to choosing a project, there are some constraints to bear in mind. Some of these are built into the qualification and the mark scheme in such a way that ignoring them would cost you marks. Others are simply good advice. Make sure you can say 'yes' to all of the questions in the table below before you make a start.

Question	Things to consider
Is your idea likely to result in a solution that contains a graphical user interface (GUI)?	According to the OCR specification, 'all tasks completed in all languages need to have a suitable graphical interface'. If you have a text-based interface, or no interface at all (for example, with a completely automated control system), you're not meeting the criteria, and a mark of zero could potentially be awarded.
Are you going to be using one of the following languages? • Python • One of the 'C' family (e.g. C# or C++) • Java • Visual Basic • PHP • Delphi	If you are, that's fine. If not, you'll need your teacher to clear your project first with OCR. For details on how to do this, see appendix 5e of the OCR specification. Whichever language or languages are used, the rule on the GUI (above) still applies. If your solution is set to include a combination of languages, each language should be either on the list or cleared with OCR.
Will your solution have scope for validation?	The word 'validation' appears seven times on the NEA mark scheme, and the related word 'robustness' appears once. This means that a project that does *not* include validation will lose marks in more than one place. Validation can exist in one of many ways, such as: • Text boxes that incorporate range, lookup, length, format, type or presence checks • A game board that blocks illegal moves, such as moving a rook diagonally in chess • A sensor or scanner that alerts the user to a value outside an acceptable range
Do similar solutions exist?	There should be a computer-based or manual system already in existence that performs a similar or related task. It doesn't need to fulfil exactly the same role that you're trying to address, but for full marks in the analysis, you need to research solutions to similar problems.
Are you definitely making a computer science project and not an IT project?	The focus of your project needs to be processing. If you create a solution that simply stores, manages and retrieves data, you've made an IT project. The processing also needs to be non-trivial. If your project could be made using Microsoft Excel, without adding any code, it's an IT project and not a computer science project.
Is it interesting to you?	You're going to spend dozens of hours coding the solution and thousands of words documenting what you do. Unless you are working on a project that genuinely engages you, this will be a struggle.

Complexity

When trying to settle on a project, both teachers and students struggle with the question 'is it complex enough?' It's not always a straightforward question to answer, because no part of the OCR mark scheme awards marks for a complex project or removes them for a simple one. That's a point worth highlighting for the skim-readers among us:

> There are no 'complexity' marks on the OCR A Level Computer Science mark scheme. A simple project that is described fully by all top-band descriptors on the mark scheme can achieve full marks.

Usually, a project that is too trivial will be unable to attain all of the marks simply because it does not adhere to a descriptor.

For example:

- In the design section, marks are awarded for defining 'in detail, the structure of the solution to be developed'; a trivial solution is not capable of offering such detail.
- Also in the design section, there are marks awarded for decomposing the problem into pieces and designing algorithms that fully address each of those pieces; a trivial solution will either fail to identify the complexity of a problem or fail to address it.
- Iterative testing requires the implementation of prototypes, which will not be possible for trivial solutions.
- Post-development testing requires candidates to address robustness, function and usability, which will typically not be an option if the solution is too trivial.

In short, as long as it meets the criteria in the previous table (with particular emphasis on creating a computer science project instead of an IT project), it's probably complex enough. Always be guided by the wording of each mark scheme descriptor, and, if in doubt, a centre – but not a student – could contact OCR for advice.

Project ideas

Ideas	Benefits	Potential problems
Games	• The inherent complexity can leave you with plenty of opportunities for detailed development and testing work. • Aside from that, if you have an interest in games development, or artificial intelligence, a project like this is likely to hold your interest.	• If you've never made a game before, you might be surprised by how complex games can be to develop. • Consider how likely you are to put in the extra work needed to learn new skills. • You should also ensure you select a project in which there is scope for data validation, or some marks will be unavailable to you.
Desktop or web-based data-handling applications	• This is probably the type of program with which you are most familiar, meaning fewer new skills will be needed at the outset. • Even if you do not look beyond your school or college, you should have no problems finding an end user. • There is huge scope for validation.	• It can be quite easy to create a front end to a database that doesn't do any substantive processing of its own; a solution needs to do far more than add, edit and retrieve data.

Ideas	Benefits	Potential problems
Interactive learning resources	• You're likely to have plenty of real end users close at hand in the form of teachers and students. • As a student yourself, you probably know about existing interactive learning resources already, giving you a head start on part of the analysis.	• To develop an effective solution, you're likely to need to learn about learning theories or a new subject, which can take time. • There is a potential danger of creating a simple quiz system, which might not attract all of the marks. • If your solution is to involve large amounts of learning material or many questions, this can be time-consuming.
Mobile apps	• Learning to develop mobile apps can be hugely beneficial to your employability. • You can incorporate familiar features such as Google Maps and notifications. • Creating a solution in which multiple devices intercommunicate can be more straightforward, without the problems presented by school or college firewalls and other security systems.	• Not all programming languages lend themselves to app development, so your language may not be appropriate (although plug-ins for additional languages are constantly being developed). • The same mark scheme will apply to an app as a desktop application, so you will still need to consider everything in the 'checklist' table above. • There is a risk that you could rely too much on plug-ins and not develop enough of your own original code.
Control / monitoring systems	• An abundance of low-priced sensors and other components can turn a Raspberry Pi or Arduino into a device that interfaces with the real world. • Much of the complexity will be apparent in setting up the hardware and importing appropriate libraries, so the expectation will be for a somewhat less complex project.	• The more unique your project becomes, the more difficult it will be to seek help from peers. • You still need to pay attention to the requirements for a GUI and for data validation in order to be eligible for full marks.
Simulations	• A wide range of ideas exist in this category, from business modelling to predicting the impact of global warming on different animal species. • You will have plenty of opportunity to combine your computer science knowledge with expertise from a different discipline.	• A great deal of data collection would be needed in order to produce an accurate simulation, which will take time. • The complexity demands of some simulations might be too much to fit into an A Level project, so you should be ready to scale back your objectives if time starts to catch up on you.

These are only broad categories; there are limitless potential computer systems that can be developed. Consider your own interests, talk to friends and family, and have a look online for ideas in other people's projects. Your work should be your own, but you are encouraged to seek ideas wherever they might be found.

1: Analysis (10 marks)

Analysis is focused primarily on the problem rather than the solution. If you are developing a program for a particular situation, you need first to understand the situation. This includes examining prospective users and similar systems, and getting to grips with the high-level ideas around what the solution will ultimately do.

Mark band 1	1–2 marks
Mark band 2	3–5 marks
Mark band 3	6–8 marks
Mark band 4	9–10 marks

1.1: Computational methods

MARK BAND 1	MARK BAND 2	MARK BAND 3	MARK BAND 4
Identify some of the features of your chosen problem that make it a good choice for solving using computational methods (such as abstraction).	**Mark band 1 plus:** Once you have identified these, describe them in sufficient detail that they could be understood by someone with no knowledge of the problem.	**Mark band 2 plus:** Explain why a computational approach is a good one for this problem.	**Mark band 3 plus:** Provide an explanation for *each* feature of the problem, e.g. *why* is abstraction (for example) an applicable technique for *this* particular feature of the problem?

You need to explain why a computational approach is suitable for the problem you're addressing. You don't need to do this first (in fact, it's probably easier if you save it until the end of the analysis, or maybe include it in the design). It only appears first here because it's the first item on the mark scheme.

So, what are 'computational methods'? It's a term that covers a wide range of techniques that you might have encountered elsewhere in computer science. The table below contains details of some of the more common ones.

Abstraction	Will your solution make a complex reality straightforward?
Decomposition	Does the problem (and prospective solution) lend itself to being broken into smaller parts?
Concurrence	Do you require a solution that can perform multiple tasks simultaneously?
Selection	Are decisions needed that will depend on inputs and other data?
Iteration	Will your solution perform the same task, or a similar one, repeatedly?
Modelling	Do you need to represent or simulate some aspect of the real world?
Visualisation	Is there a need for outputs of different forms, such as graphical, sound and text-based?
Data mining	Is there some need to spot patterns in large amounts of data?

You don't need to collect the set; just choose the ones that apply, then describe exactly where in the problem each approach is needed. You're not likely to need, for instance, visualisation to be a prominent approach throughout, but perhaps the centrepiece of a dashboard form is a line graph.

e.g.	*The dashboard specified by the stakeholder needs to provide an interface that can provide an at-a-glance summary of which student projects show the greatest likelihood of plagiarism. Visualisation* **[IDENTIFY]** *will be at the centre of this part of the solution, with Internet plagiarism and peer plagiarism highlighted in different colours* **[DESCRIBE]**. *There will also be, as stipulated by the stakeholder, a line chart, which shows the percentage of potentially plagiarised content across all work submissions for each student* **[DESCRIBE]**. *Using this approach will allow the user to make a judgement on whether to investigate further, as all necessary information will be visible in a single place* **[JUSTIFY]**.

1.2: Stakeholders

A stakeholder is any individual with an interest in the solution that you are developing. End users are the main stakeholders to consider, but data subjects (i.e. people identifiable by any data processed by your solution) are also stakeholders. In this section, you need to examine how your solution will meet the needs of stakeholders.

MARK BAND 1	MARK BAND 2	MARK BAND 3	MARK BAND 4
Identify the stakeholders as either individuals or groups, describe them and describe what they might want in your solution.	**Mark band 1 plus:** Describe *how* they would use the solution as well as what features they might want.	**Mark band 2 plus:** Describe (rather than identify) the stakeholders and describe *why* your solution will meet their needs.	*The wording in the mark scheme is identical for mark bands 3 and 4, so the emphasis is on providing detailed description and explanation.*

Before you identify the stakeholders, you should first introduce the problem and the organisation to which it pertains. The following questions, in this order, should help you get started on this section:

1. What is the name of the organisation?
2. What does the organisation do?
3. What problem does the organisation have that you could attempt to solve?
4. Who are the stakeholders that would be affected by your solution?

Ideally, stakeholders are real individuals of whom you can ask real questions, so you should try to choose individuals to whom you have some access.

No one will be conducting investigations as to whether your stakeholders are real, and that you actually spoke with them, but it's so much easier if you do. With a real stakeholder, you ask them about the solution they want, make it, and then ask their opinion of it at the end. With an imaginary stakeholder, you have to fabricate an interview, think long and hard about what features to include, then invent a plausible opinion about everything later. Honestly, it's exhausting.

In order to address the needs of your stakeholders, you should first find out what their take is on the problem, as well as what they might want from a solution. There are several ways to this, and you can choose one of these or take more than one in combination. Other approaches may also exist, depending on the nature of the problem.

Interviews are useful for gathering large amounts of data from a small number of stakeholders (including a single stakeholder). Questions can be open-ended, and follow-up questions can be asked. You might, for instance, ask what they think are the problems with the current solution, and be given a list of 20 items. In an interview, you could ask them which are the most significant three, or ask for specifics about how they might want a solution to work – it might differ from your own vision.

A transcript (a record of everything that was said by both interviewer and interviewee) will need to be included for each interview conducted.

	**Questionnaires** lend themselves to collecting a small amount of data from each of a larger number of stakeholders. If you wanted to create a college-wide system to be used by all staff, you might want to collect information from a dozen or so people, but a dozen interviews would be quite time-consuming. Long-answer questions tend not to lend themselves to questionnaires, as people can ignore them more easily than in an interview, and you're less likely to be nearby if they do not understand a question. Multiple-choice questions, ranking questions and those that require short answers are appropriate for questionnaires, and can make subsequent statistical analysis straightforward and meaningful.
	**Observation** involves watching the stakeholder using the current system, and can give you insights beyond what you might find in a questionnaire or interview. On which screen do they spend the most time? How long does it take to perform a particular task (you can time it)? Are there any features that they struggle to find? Since you're looking to create a solution that improves on what's already available, you might simply spot a four-click activity that you could simplify to two clicks.

When planning questions to ask, bear in mind the following:

- Each question should produce an answer that genuinely helps in the development of your system. Asking 'do you like the current system?' yields a 'yes' or 'no', neither of which really helps. Asking 'if you could change one thing about the current system, what would it be?' gives you something to go on, and even leaves the respondent the option of saying 'nothing'.
- Recognise that your technical knowledge might be greater than that of your stakeholder. 'Should this be an integer?' is less likely to be understood than 'will this ever need to store fractions or decimals?'
- Don't make it too easy on your stakeholders by always providing 'I don't know' or 'none of the above' options. Alternative approaches, such as placing features in order of importance, or scoring statements on a scale of 1 to 10, might give you more meaningful data.

Once you have gathered your data, it needs to be analysed. Transcripts of interviews or copies of questionnaires are not enough. Describe how you plan to proceed with your solution as a result of engaging with the stakeholders, and justify any choices you make. This could be done textually or diagrammatically, perhaps by way of flow charts or data flow diagrams (DFD).

Good and bad interview questions

Problematic	Better	Explanation
How bad is the current interface in your opinion?	*How would you rate the quality of the interface on a scale of 1 to 10, 10 being the best?*	The problematic question assumes that the person answering the question believes the interface is bad. That might not be their opinion.
How would you rate the program's interface and performance?	*How would you rate the interface?* *How would you rate the performance?*	The problematic question is really two questions. The interface might be perfect and the performance abysmal, but the question seems to assume that the ratings will be the same.
Do you always keep a printed copy?	*How often do you keep a printed copy?*	Yes/no questions don't really add much to your understanding. If they have kept a printed copy all but one of the thousands of times they have used the system, the honest answer is 'no'.

If you're conducting an interview, don't worry about going off-script. There's nothing wrong with asking a question you weren't planning on asking. Often, an answer you receive might require another question. The respondent might say 'I hate the menu structure', at which point you'd probably want to ask for specifics.

Good and bad questionnaire questions

Problematic	Better	Explanation
How often do you use the system?	*How often do you use the system?* ☐ *More than once a day* ☐ *3–7 times a week* ☐ *1–2 times a week* ☐ *Less than once a week*	Presenting the respondent with options has a few advantages. The question can be answered more quickly, so you're doing them a favour. They're also quicker for you to analyse, as you're simply counting ticks.
When did you last update the software? ☐ *Today* ☐ *This week* ☐ *Last week* ☐ *Last month*	*When did you last update the software?* ☐ *Within the last 24 hours* ☐ *Between 1 and 7 days ago* ☐ *Between 8 and 14 days ago* ☐ *More than 2 weeks ago*	There are gaps as well as overlaps in the problematic options. 'Today' is part of 'this week', there's a gap between last week and last month, and there's no option for further back than last month.
How should the data be stored? ☐ *Cloud* ☐ *USB flash drive* ☐ *Other*	*How should the data be stored?* ☐ *Cloud* ☐ *USB flash drive* ☐ *Other (please state)* _____	'Other' is not very helpful unless you know what 'other' is. Your respondent won't mind, as ticking 'other' is quick and easy, but it doesn't provide you with anything useful.

! You don't have to restrict yourself to lists of options, but you should keep questions short, with any written responses being no more than a few words. An exception to this would be an 'is there anything else...' open-ended question at the end, where people can write what they choose or leave it blank.

Your interview transcripts and/or completed questionnaires should not go in the main body of your work, but should be added at the end, in an appendix. What should be included within your work is your analysis. In your analysis, you will present your findings to the reader and explain the impact these will have on the development of your system.

e.g. *I believe it would be best to incorporate access to the print function in three different places, namely in the 'file' menu, on the toolbar and as a reaction to the CTRL+P key combination* **[DESCRIBE]**. *The reason for this is that the questionnaires showed that people used all three of these options (30% file menu, 50% toolbar, 20% shortcut key)* **[EXPLAIN]**. *While I could have incorporated only a toolbar option – the most popular currently – this would make the solution less intuitive for half of the prospective users* **[JUSTIFY]**.

1.3: Research of existing solutions

MARK BAND 1	MARK BAND 2	MARK BAND 3	MARK BAND 4
List some features that you plan to include in your solution.	**Mark band 1 plus:** Conduct research of similar systems and provide evidence of that research. Clearly show how the features you plan to adopt have arisen from the research.	**Mark band 2 plus:** Describe (not list) the findings of your research, and describe how you might approach your solution in light of this research.	**Mark band 3 plus:** Research a broad and diverse range of similar and related solutions, explaining *why* your approach will be influenced by each of your findings.

Your aim is to create a solution to an existing problem, but you probably won't be the first person to try to solve this problem. In this section, you'll have a look at previous approaches to the problem you're trying to solve. You might examine any or all of the following:

- Computer-based solutions for the problem you're addressing
- Paper-based solutions for the problem you're addressing
- Solutions for different but related problems

(That last one is particularly important if you're working on something new and inventive.)

The following flow chart is something you should work through repeatedly. Ideally, you'll find several systems from which you can gain ideas, and you are likely to find multiple noteworthy features from each system. This process is not about stealing ideas, since you'll adapt your findings to meet the unique needs of your stakeholders.

1. Find an existing system you want to investigate, identify it and include an image if possible
2. Describe the system for the benefit of anyone who is not familiar with it
3. Describe a feature or characteristic you plan to adopt for your own solution
4. Explain how this new feature will fit in within your solution
5. Describe how you will alter this feature in order to customise it for your own solution
6. Justify your choice, including addressing alternative approaches which also might have worked
7. If applicable, repeat steps 3–6 for features you are NOT going to adopt (shows you're being selective)

e.g.	*The system currently used by most kiln operators uploads temperature readings* **[IDENTIFY]** *at regular timed intervals* **[DESCRIBE]**. *I will need to incorporate this into my own system, because there will generally be no one near the kiln when it is in operation, and intervals need to be set as frequently as every five seconds* **[JUSTIFY]**. *Currently, when the user wants to see the temperature history, they have to manually refresh the web interface* **[IDENTIFY]**. *I will develop my own system differently, so that it continually draws the most up-to-date history line graph, without human involvement* **[DESCRIBE]**. *Although this will require more bandwidth, it should not present a problem, as it will be accessed over a wireless network with no data limits* **[JUSTIFY]**.

1.4: Essential features

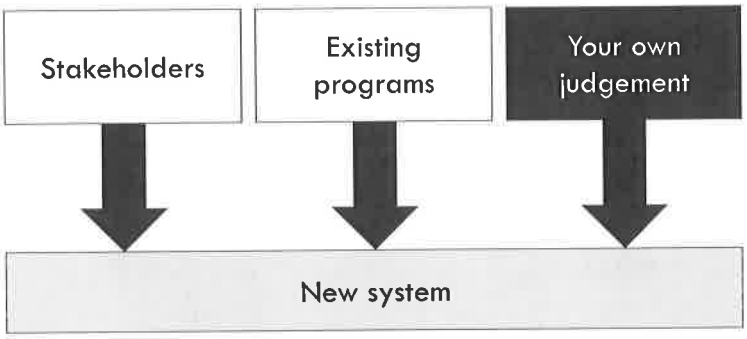
MARK BAND 1	MARK BAND 2	MARK BAND 3	MARK BAND 4
Provide a list of some of the things your solution *must* do in order for it to address your chosen problem.	**Mark band 1 plus:** Ensure you have identified *all* essential features, rather than some.	**Mark band 2 plus:** Describe each feature, treating each listed item as a subtitle under which you will provide some detail.	**Mark band 3 plus:** Explain why each feature is needed and why it is essential, developing your descriptions.

By now, you should have a clear idea of what your system is required to do, so you should be able to provide a list of features that must be included.

You're encouraged to use your own judgment on this, and not simply regurgitate what you learned from stakeholders and existing systems.

For full marks, you need to **state**, **describe** and **explain** each of your essential features.

For example:



e.g.	*My system will require a 'create new account' feature.* **[STATE]** *This should involve the student clicking on a 'new account' button and entering a username and a password. The system will display a message if the username is already chosen, otherwise it will let them proceed.* **[DESCRIBE]** *The feature is needed because 100 new students will join the department each year, and each new student needs their own account. Letting them choose their own username causes less work for staff* **[EXPLAIN]**.

1.5: Limitations

MARK BAND 1	MARK BAND 2	MARK BAND 3	MARK BAND 4
Identify features that your solution will not include, as well as factors that might prevent you from creating the ideal solution.	**Mark band 1 plus:** Describe, rather than identify, offering detail as to the nature of the limitation.	**Mark band 2 plus:** Explain why each of these limitations exists.	**Mark band 3 plus:** Show consideration of alternatives; in your limited time frame, there must be features that are ruled out, and features that were potentially unviable that you'll be including. Walk the reader through the process of deciding.

There are two questions to consider in terms of limitations:

1. What will your program not do that a reasonable person might suspect that it *will* do? For example, is it a game that does not save player progress? Is it a revision tool that does not track test scores over time?
2. What constraints will prevent your program from being perfect? Think about software interoperability, time, limitations in your skill set, unavailability of key stakeholders, lack of network infrastructure...

e.g.	*My system will not be able to work with real-time exchange rates* **[IDENTIFY]**. *One idea was that, at any stage during the transaction being processed, the user would be able to switch between US dollars, euros and pounds sterling* **[DESCRIBE]**. *This will not be an option, as the most straightforward way to accomplish this would be to pay for the data from a provider such as XE, and this is not a commercial project* **[EXPLAIN]**. *While I could spend time writing script that would extract the data from a relevant website* **[DESCRIBE]**, *this would add little to the solution, as my stakeholder has indicated that this would not be an essential feature* **[JUSTIFY]**.

1.6: Hardware and software requirements

MARK BAND 1	MARK BAND 2	MARK BAND 3	MARK BAND 4
List some of both hardware and software that the solution requires, in terms of developing it and running it.	**Mark band 1 plus:** There should be no major omissions in terms of what is needed to develop and run your solution.	**Mark band 2 plus:** All hardware requirements should be described (not listed), and you should consider specifics, such as versions of software and screen resolutions.	**Mark band 3 plus:** Justify any choices you have made, explaining why each piece of hardware or software is needed.

In order to get as far as mark band 2, you simply need a comprehensive list. Let's use the example of a desktop data processing application, developed using Visual Basic, which interfaces with an Access database.
The following would be enough:

Hardware:	**Software:**
• Processor clock speed of at least 1 GHz • 2 GB RAM minimum • 3 GB of free hard disk space • Display capable of 1080 x 768 pixels • Standard UK-layout keyboard • Two-button mouse	• Visual Studio 2017 (for development) • Microsoft Access 2016 • Windows 7 (or later) operating system • .NET framework (minimum version 4.5)

That's it for mark band 2. If there were omissions from this list, that would nudge the score back into mark band 1, but a list that covers everything ticks the box for mark band 2.

!	When it comes to the specifics, don't just make them up. The clock speed, RAM and HDD requirements should come from the most demanding software you're going to run (Access 2017 in this case), and the resolution of the display should come from the on-screen size of the application you're planning to make. If you're not sure about any of these, you can always come back and fill them in once you've started development.

For mark bands 3 and 4, you need to describe, explain and justify each item on the list. There are 10 bullet points above, so that means 10 small paragraphs, each following a similar format. Let's look at the example of the two-button mouse:

e.g.	*A two-button USB mouse will be needed* **[DESCRIBE]**. *USB ports will be available in the target machine, which is a standard-issue college laptop* **[EXPLAIN]**. *Two buttons are important, since the left button triggers the event of on-screen objects, while the right button accesses a help feature for each object* **[EXPLAIN]**. *While the touch pad on a laptop will be adequate for this role, people prefer mice and tend to progress more quickly when using them* **[JUSTIFY]**.

The first point is 'describe' rather than 'identify', because it's a 'two-button mouse' rather than simply a 'mouse'. For the 'justify' point, an alternative approach has been genuinely considered.

1.7: Success criteria

MARK BAND 1	MARK BAND 2	MARK BAND 3	MARK BAND 4
Provide a checklist against which you will be able to evaluate your finished solution.	**Mark band 1 plus:** Ensure that each item on your checklist can be measured in some way, and describe *how* each item will be measured.	**Mark band 2 plus:** Your checklist should cover all aspects of the proposed solution.	**Mark band 3 plus:** Each item should be justified, with reasons given for its inclusion, its threshold and the means of measurement.

Here, you're putting together a list of statements that will be used to measure your solution once you've finished working on it. It's best thought of as your own private mark scheme, in which the perfect solution will score 100%. You should accept that you won't actually score full marks, but it should still be realistic, and you should aim for each criterion. Each one should be achievable independently of the others, and be separately measurable and justified.

Probably the best way to understand a good success criterion is to have a look at some bad ones:

Attempt	Problem
I should have a user-friendly interface	Admirable, but there's no mention of how to measure this
I should have a user-friendly interface, which will be measured by giving stakeholders a questionnaire after they have used it	Better, but we need to know something more about this questionnaire
There will be a question 'rank the usability of this application on a scale of 1 to 10'	Here we have the matter resolved of how this will be measured, but we still need a pass mark of some kind
In order for the solution to be a success in terms of usability, the score should be either 9 or 10	This is now measurable, so all that's needed now is a justification
I should have a user-friendly interface, which will be measured by giving stakeholders a questionnaire after they have used it. There will be a question 'rank the usability of this application on a scale of 1 to 10'. In order for the solution to be a success in terms of usability, the score should be either 9 or 10. I have chosen a questionnaire because there is no objective way of measuring usability, and I have chosen a threshold of 9–10 because the main problem with the current system is a lack of user-friendliness.	Perfect. This is described, explained and justified. It's also measurable, which means there won't be any lack of clarity later on, when you're deciding whether or not you've succeeded in this regard.

Most other success criteria will be easier to define, as there will be less of a 'human factor' about them.
You might aim to have a feature for a new user to create a new account. This could be evidenced simply by testing that the 'new account' part of the solution works.

The success criteria section is important, as it forms the basis of the evaluation at the end of the project.

You will need to provide an assessment, for each criterion, of whether you met it, partially met it or failed to meet it.

As such, you would benefit greatly from numbering your criteria.

Analysis » Checklist

MARK BAND 4:	9–10 MARKS

- ☐ Features that make this problem appropriate to approach using computational methods (such as abstraction and decomposition) are identified, described, explained and justified
- ☐ Analysis of stakeholder requirements is presented, typically as a result of interviews and/or questionnaires, and the way in which a solution is about to be developed is described and explained
- ☐ There is in-depth research covering multiple similar or related solutions
- ☐ Research into similar or related solutions has provided insights on how to proceed, which are described, explained and justified
- ☐ Essential features are identified and described, and there is clear explanation of why they are essential
- ☐ All limitations on the solution are clearly described, explained and justified
- ☐ Every piece of hardware and software required is described in full, and its inclusion within the list for the purposes of the solution is justified
- ☐ Success criteria are measurable, with the means of measurement clear, and each one is justified; the success criteria cover the proposed solution in its entirety

MARK BAND 3:	6–8 MARKS

- ☐ Features that make this problem appropriate to approach using computational methods (such as abstraction and decomposition) are identified, described and explained, but not justified
- ☐ Analysis of stakeholder requirements is presented, typically as a result of interviews and/or questionnaires, and the way in which a solution is about to be developed is described and explained
- ☐ There is in-depth research covering multiple similar or related solutions
- ☐ Research informs descriptions of how (but not why) the problem will be approached
- ☐ Essential features are identified and described, but not explained
- ☐ All limitations on the solution are clearly described and explained
- ☐ Hardware and software requirements are specified in full
- ☐ Measurable success criteria are stated and cover the proposed solution in its entirety

MARK BAND 2:	3–5 MARKS

- ☐ Features that make the problem appropriate for computational methods are described, but not explained
- ☐ Description of how stakeholders will use the system is included, supported by some investigation
- ☐ Features from researched similar solutions that might transfer to your own are identified
- ☐ Essential features, limitations and most hardware/software requirements are identified
- ☐ Success criteria are identified, which must still be measurable

MARK BAND 1:	1–2 MARKS

- ☐ The computational methods section is characterised by identifying applicable features, rather than describing them
- ☐ Stakeholders, and some of their needs, are identified, but not necessarily described
- ☐ Appropriate features are identified for incorporation into your solution, but this may or may not be based on any kind of research
- ☐ Essential features, limitations and some hardware/software requirements are identified
- ☐ Success criteria are identified, although they might not be measurable

2: Design (15 marks)

In the design phase, you plan the development of your solution. This covers data structures, interface features, algorithms and planning out exactly how you will test your solution, both during and after development.

If you have conducted your analysis properly, with meaningful examinations of existing systems, and interactions with real stakeholders, this section should be quite straightforward. The aim is to design a system that the stakeholders want, that builds on the strengths of existing systems.

Mark band 1	1–4 marks
Mark band 2	5–8 marks
Mark band 3	9–12 marks
Mark band 4	13–15 marks

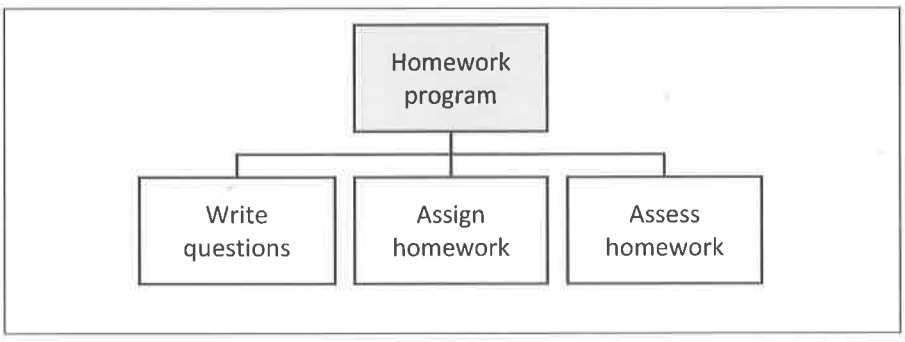
Obviously, your solution will not resemble your design precisely. You will change your mind about how best to proceed, and you're likely to add, remove and change quite a few things. You do not need to revisit your design to make retrospective changes when this happens, but you should keep a log of changes that you make once you're developing. It can lead to a greatly enriched evaluation.

2.1: Problem decomposition

Note than you are decomposing the **problem** here, and not the **solution**. This part is essentially where the analysis gives way to the design, so you're showing that you understand the current situation.

MARK BAND 1	MARK BAND 2	MARK BAND 3	MARK BAND 4
Problem decomposition is not required for mark band 1.	Break the problem into smaller sub-problems, describing what you're doing at each stage.	**Mark band 2 plus:** Add explanation to your descriptions, talking about why you've gone about this process in a particular way.	**Mark band 3 plus:** Offer justifications, by presenting alternative ways in which the problem could have been decomposed, explaining why you favoured your approach.

As an example here, we're going to use the problem of setting and assessing homework questions for an A Level Computer Science class at a sixth form college with a policy of assigning weekly homework.



A hierarchy diagram is optional, but it is a great tool for showing how a large problem can be broken into a series of smaller problems. This has just two layers, but for larger problems, with multiple decomposition phases, there's no upper limit.

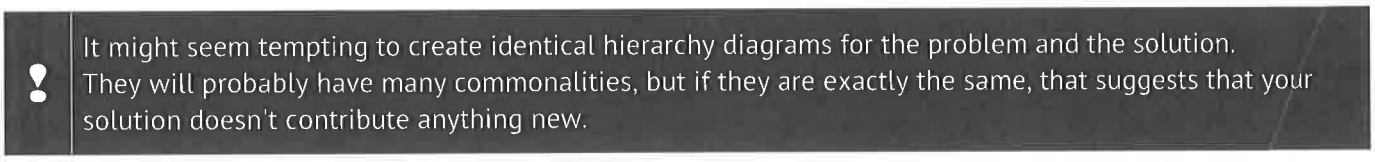
Write questions: Currently, the teacher writes a series of topic-based questions, with each topic in a separate Microsoft Word document (i.e. one document for binary, one for hardware, one for operating systems, etc.). *Assign homework: When homework is assigned during the first class of the week, questions are chosen from all topics covered so far that year, and copied and pasted into a new Microsoft Word document. This is uploaded to the VLE along with an upload link for students.* *Assess homework: Student files are downloaded by the teacher, they are manually marked and grades are uploaded on the VLE so that each student can see only their own.*	This is a detailed description of the problem. More importantly, it's a detailed description of the pieces into which the problem has been decomposed. Each piece has been described clearly.
I have broken the problem down by process rather than by user (student/teacher), as it lends itself better to creating a solution. This is because my solution will be broken down into processes (algorithms) and not by who's using the system.	Here, explanation and justification are demonstrated. The way in which the problem has been decomposed is explained, and a viable alternative (breaking the problem down by user) is also examined but ultimately rejected.

2.2: Structure of the solution

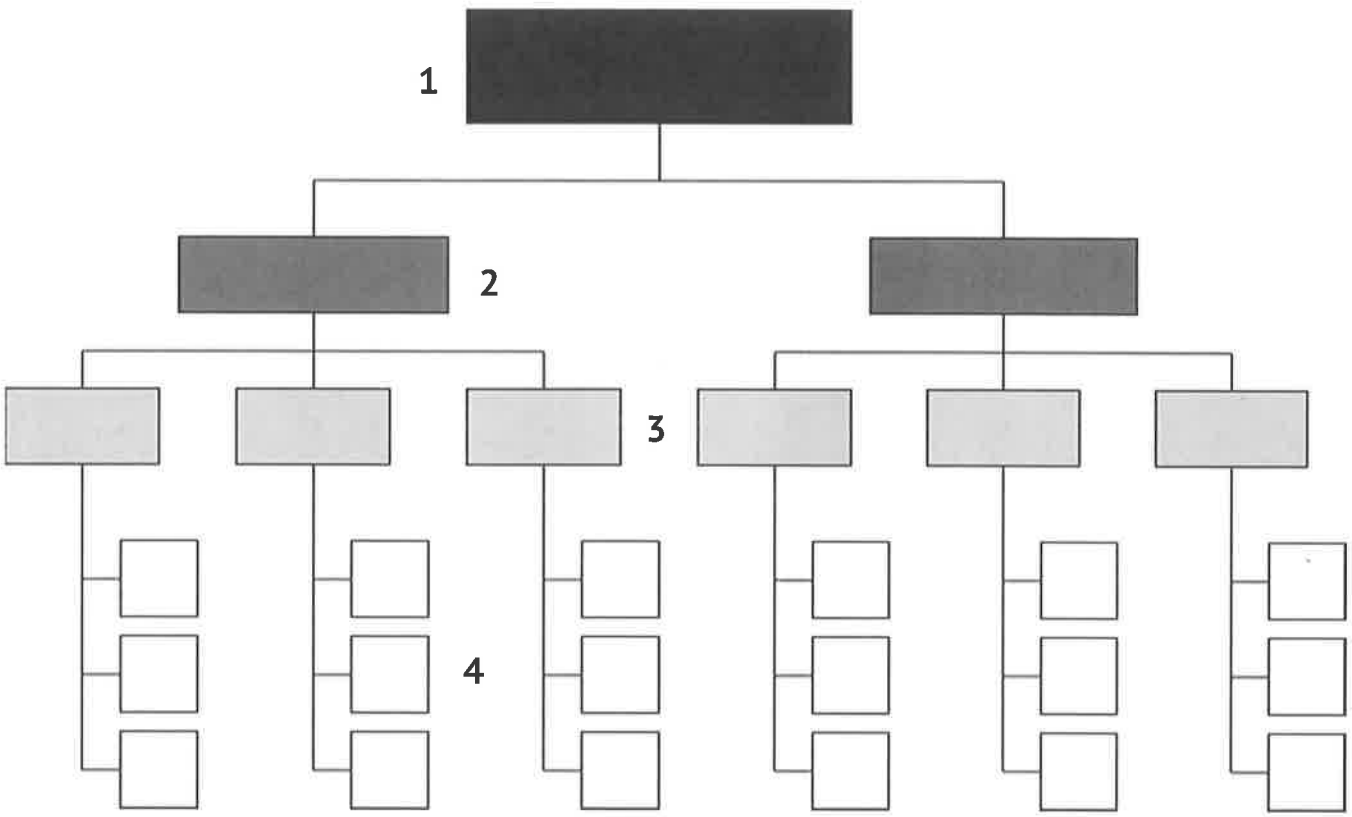
It's easy to confuse part 2.1 (problem decomposition) with this part, and people often approach the two parts at the same time.

This is fine, but you need to bear in mind that in 2.1 we break the **problem** into its constituent parts, whereas in 2.2 we start to showcase the parts of the **solution**. Unless your work addresses both the problem and the solution, you will be unable to gain full marks for the design.

MARK BAND 1	MARK BAND 2	MARK BAND 3	MARK BAND 4
Structure of the solution is not required for mark band 1.	Show how the parts of the proposed solution relate to each other.	**Mark band 2 plus:** Your work should be comprehensive and cover all aspects of your solution, whereas mark band 2 allows for some gaps.	*The wording in the mark scheme is identical for mark bands 3 and 4, so the emphasis is on providing detail.*

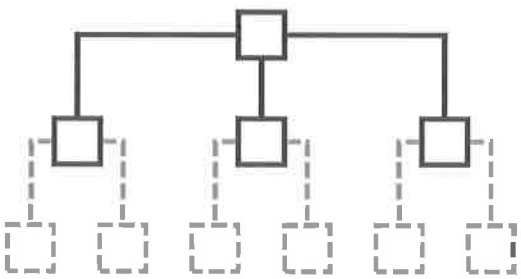


No justification is required by the mark scheme for this part; you simply need to present the structure of the proposed solution. Assuming you can fit the hierarchy diagram for your solution on a single page, it might look like this (of course, each of your shapes will contain text, depending on what they are denoting):



1. This is the name of your entire system. In this instance, it might be 'homework system'.
2. At the first level of decomposition, the solution is divided up into major subsystems. In this case, there might be a student subsystem (to complete homework) and a staff subsystem (to set homework and monitor results).
3. This tier might represent forms or individual web pages (if applicable). For the staff view, this might be divided into 'write questions', 'assign homework' and 'assess homework'.
4. At the bottom are the individual subroutines that make up each form. For example, in assessing homework, there might be a subroutine called '`save_mark`' and another called '`download_next`'.

! With no justification needed in this part of the write-up, top marks are generally awarded to candidat who define their solution in detail. This means breaking down each part into individual subroutin well as creating a design that covers everything mentioned in 'essential features' (1.4) from yo If you need to, split your diagram across multiple pages rather than reducing font size or mi



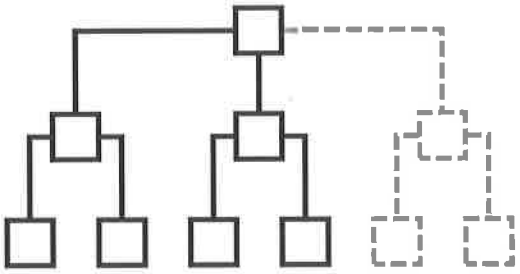
Decomposition has taken place, but enough. Each terminal node (sub any further) should be straight

Either:

- given your stru would progr other an

or:

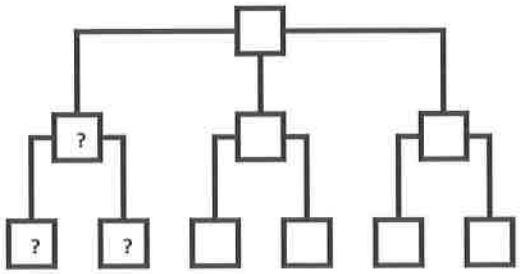
- any c be equa



Some tasks are decomposed in detail, but others are missing. Check your structure diagram against the task.

Does your structure diagram contain the following?

- Everything from your 'essential features' (1.4)
- Everything from your 'success criteria' (1.7)



You have included tasks that cannot be traced back to your analysis. There is no upper limit to how substantial a solution you can develop, and it's not uncommon for a solution to change from design to development. In fact, it probably will change.

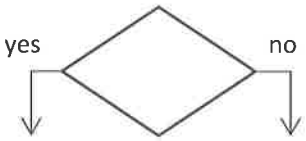
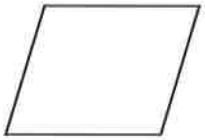
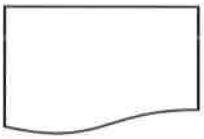
The nature of the solution shouldn't change between analysis and design. If it does, that probably means your essential features section, or your success criteria section (or both) are incomplete.

2.3: Algorithm design

MARK BAND 1	MARK BAND 2	MARK BAND 3	MARK BAND 4
Describe how individual, algorithm-level elements of your solution will work.	**Mark band 1 plus:** Adhere to standard algorithm conventions using flow charts or pseudocode, and ensure *all* algorithms are designed.	**Mark band 2 plus:** Include commentary with each algorithm, explaining how it fits in with the rest of the solution.	**Mark band 3 plus:** Offer alternatives to how individual algorithms are designed, and how they interrelate, explaining why you ultimately made the decisions that you made.

There is no specification in the mark scheme as to the format in which algorithms should be designed. Algorithms are normally presented in the form of flow charts, pseudocode, actual program code and structured English.

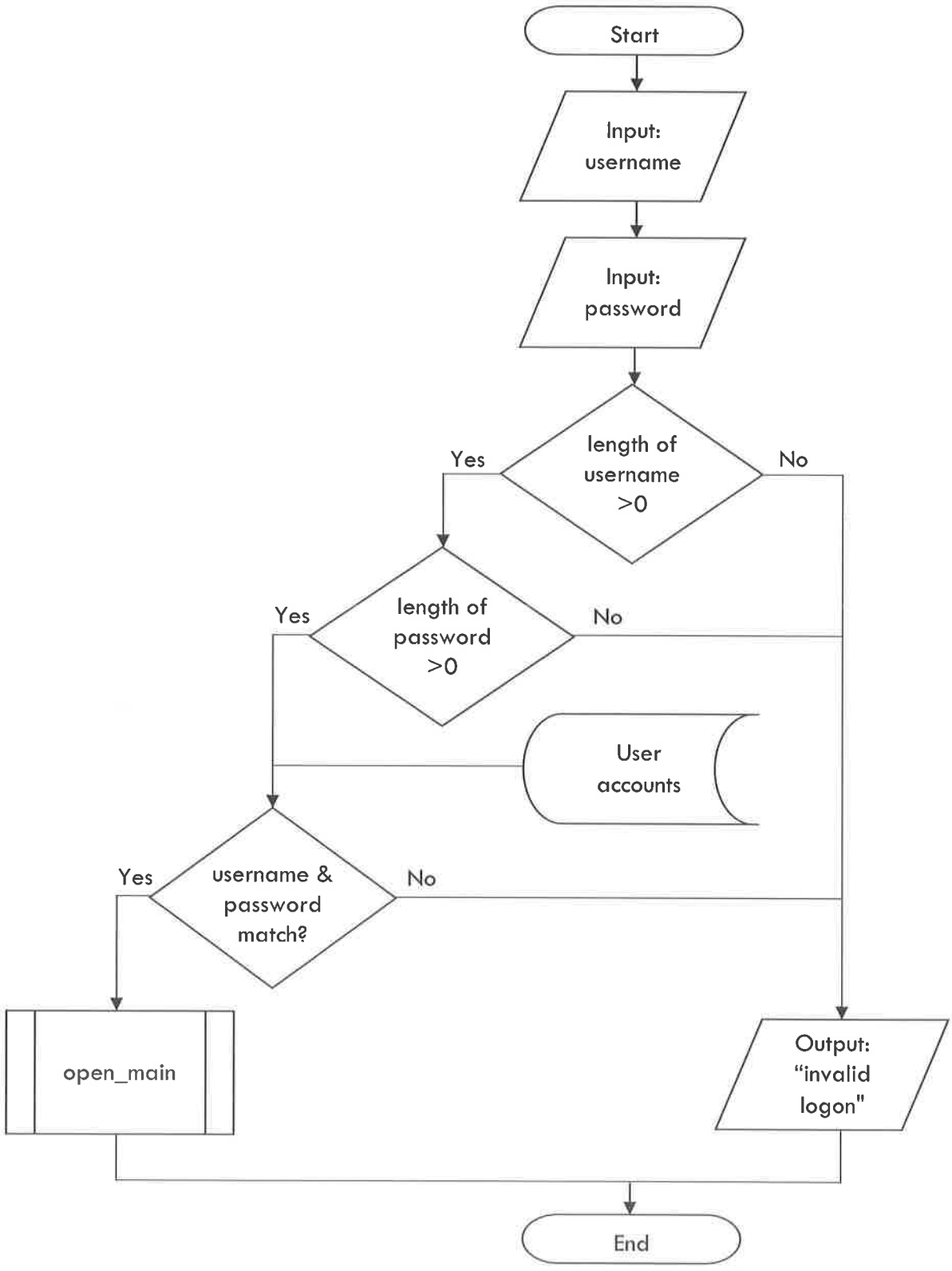
Of these options, the favoured two are pseudocode and flow charts. Structured English, unlike these two, does not follow a standard format, and program code, in this coursework, is assessed elsewhere, so you shouldn't use it here as well.

Flow chart shape	Usage
	**Terminator** Only two of these appear in each algorithm – one saying 'start' and one saying 'end'. If your algorithm splits off into multiple paths, they can all come together at the 'end' terminator.
	**Process** Unless an action involves input, output or storage, it probably belongs in this shape. Process shapes are commonly used for calculations as well as declaring/initialising variables.
	**Decision** Used in place of an IF statement. The text content of a decision should be a yes/no question. The IF part of the algorithm then follows from the 'yes' arrow, with the ELSE part following the 'no' arrow.
	**Input/output** Items input from the user or output to the display are depicted using one of these shapes, which will contain 'input' or 'output' accordingly, as well as the name of the variable.
	**Document** This is most likely to be used if your solution is going to create printed output, but it can also work as an input, particularly if a document is being scanned or transcribed. It would be used in place of the input/output shape.
	**Stored data** This will represent files that are read from or written to, which might be text files, binary files, databases or whatever else your solution will work with.
	**Connector** If you need to split a large flow chart over more than one page, you'll use connectors. At the bottom of page 1, your flow chart will end with a connector containing the letter 'A'. The flow chart would continue from whichever other page begins with another 'A' connector.
	**Predefined process** This is the shape you'll use for one flow chart to 'call' a subroutine defined by another flow chart. The name of the subroutine being called should go into the shape, and an algorithm with a corresponding title should be defined elsewhere.

There are several tools that can help you to draw flow charts, but I would strongly recommend a website called 'draw.io'. Anything you create can be automatically saved in the cloud, and it tends to be far quicker to work with than general purpose software such as PowerPoint.

Here is an example of a flow chart that deals with an attempt to log into a system:

Algorithm: logon



In order for this flow chart to be valid, several other pieces would need to be in place throughout the rest of the work:

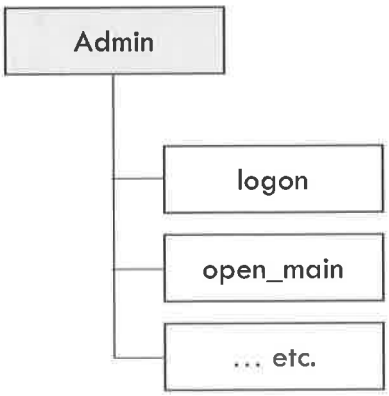
- Another algorithm, called 'open_main', would need to be designed
- A file called 'User accounts' would need to have been designed in section 2.5
- Variables called 'username' and 'password' should also be designed in section 1

An alternative to the flow chart would be pseudocode (for OCR's pseudocode syntax, see their specification on their website):

```
public procedure logon()
      username = input("Username: ")
      password = input("Password: ")
      if username.length < 1 then
            print("invalid logon")
      elseif password.length < 1 then
            print("invalid logon")
      else
            results = SQL.run("SELECT * from user_accounts WHERE U_NAME = '" +
            username + "' AND P_WORD = '" + password + "'"
            if results.length > 0 then
                  open_main()
            else
                  print("invalid logon")
            endif
      endif
endprocedure
```

! The flow chart and pseudocode shown here perform the same task. You are not required to do both, and are unlikely to gain any marks by doing both, though you will lose time.

Every terminal node in your hierarchy diagram from part 2.2 (assuming you approached 2.2 with a hierarchy diagram) should have an algorithm to correspond to it. The diagram in this guide has 18 terminal nodes, so there would be either 18 flow charts or 18 sets of pseudocode (or a combination of the two). The names of the algorithms should correspond with the names written or typed onto the terminal nodes.



For full marks in this subsection, you need to justify how these algorithms fit together to make the complete solution. To put it another way, why did you choose to create these particular algorithms when other combinations might have been suitable?

e.g.	*I decided to create a separate, self-contained logon subroutine* **[DESCRIBE]**, *since this aids modularity and maintainability* **[EXPLAIN]**. *If a new logon system were to be developed, such as using RFID cards, only this subroutine would need to be rewritten* **[DESCRIBE]**. *Had I incorporated this into a larger 'admin' subroutine instead, subsequent maintenance would have been more difficult, as any code that needs to be changed would first need to be located within a larger body of code* **[JUSTIFY]**.

Providing a justification for each individual algorithm should see you safely into the top mark band.

2.4: Usability features

MARK BAND 1	MARK BAND 2	MARK BAND 3	MARK BAND 4
List some of the features your solution will employ to make your program accessible to all and user-friendly.	**Mark band 1 plus:** Provide a full list of such features, and add descriptions to each of them.	**Mark band 2 plus:** Explain specifically why each feature has been included; what is its purpose?	**Mark band 3 plus:** Offer a justification of the chosen features, to include explanations of alternative approaches which you ultimately did not adopt.

You are not required to design screen layouts, but you might find it a useful preliminary step to help you to identify the usability features you intend to use. They can also reduce your word count, since fewer words are needed to show, for example, a search utility than to describe it.

Usability features include the following:

- Use of colour, including text, background, buttons, etc.
- Use of a consistent layout, in which you specify the positions of each element on the screen (e.g. the 'OK' button always goes in the bottom right, a help button is available on all screens in the top right)
- Use of icons
- Accessibility features, such as settings to change font size, colour combinations and language
- Any other approaches you might take to make the solution user-friendly

For full marks, you need to **identify**, **describe**, **explain** and **justify** all of the features you plan to use:

e.g.	*On my main menu bar, I intend to use tooltips on each of the icons* **[IDENTIFY]**. *When you hover over any of the 'new', 'open', 'save', 'print', 'close' or 'help' icons, the word that denotes that button's purpose will appear in a temporary box that disappears when the cursor moves on* **[DESCRIBE]**. *This feature will be used so that users can be sure of the purpose of the button they're about to click before they click it* **[EXPLAIN]**. *I did consider using text buttons, but they take up more space than is available, besides which most people are familiar with the standard icons for each of these processes* **[JUSTIFY]**.

2.5: Variables and validation

MARK BAND 1	MARK BAND 2	MARK BAND 3	MARK BAND 4
Provide names and, where applicable, data types of the most important variables and data structures (such as lists, arrays, classes and database tables and text files).	**Mark band 1 plus:** Identify any validation for any variables or data types.	**Mark band 2 plus:** Justify the existence and nature of each variable / data structure, e.g. 'why is it necessary?', 'why is it an integer?', 'why is it an array?'	**Mark band 3 plus:** Include justifications for all data validation as well, including why a data item needed any validation at all, and why a particular validation type was chosen.

It's not just variables that need to be designed in this section, but anything that can store one or more pieces of data. This includes variables, data structures (such as arrays), classes and external files. Given the somewhat formulaic way in which you will describe each variable in turn, a table is probably the most sensible approach:

Variables

Name	Data type	Scope		Purpose
password	String	Local to the 'logon' subroutine of the User class	N/A – local variable	The password of the current user
userID	String	Attribute of User class	Private, non-static	The ID of the current user
userCount	Integer	Attribute of the Log module	Public, static	Total number of users

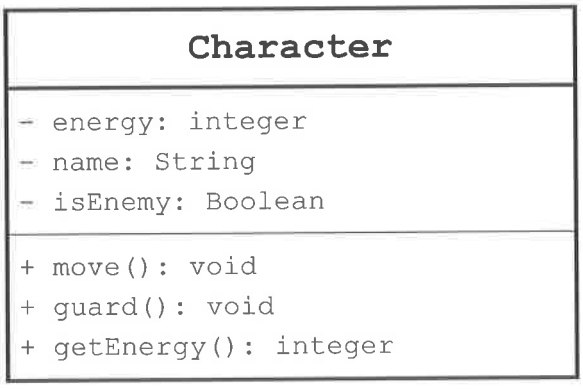
When it comes to justifying decisions here, you can justify the very existence of a variable (why do we need a userID at all?), and you can justify some property of a variable (why is it a string, or why is it private?). Given the large number of variables you're likely to have, you should not aim to justify every property of every variable. Instead, you should offer justifications whenever you were presented with a genuine choice. You should also divide your variables into separate tables, with one table per class, module or form.

Data structures

Name	Data type	Scope	Purpose
arrScores	Integer array	An attribute of the 'game' class	A 10-element array to store top 10 scores
dtbUsers	Access database	N/A: this is outside the bounds of the program	Stores login names and passwords of all current users

In this instance, the Access database would also require a table of its own, since it would contain fields of various purposes and data types. 'Scope' would not be required in such a table, as it is not part of the program, but an external file.

Class diagrams



Again, there is plenty here to justify. Why is 'energy' an integer? Why is it private instead of public? Why does the 'move' method have a void data type? Why do we even need a 'character' class? How else could this have been done?

Data validation

!	It is critically important that your solution involves some form of data validation. Without it, marks can be lost in each of the design, implementation and testing stages.

You're not expected to provide validation routines for every individual variable. In fact, if you were to validate unnecessarily, you'd be quite likely to lose marks, as the top-band mark scheme says you should be 'justifying and explaining any necessary validation'. The inclusion of any unnecessary validation would prevent you from getting the top mark here.

You might make use of any of the following:

- Presence check – ensuring that the user has entered something
- Range check – ensuring that a date or number is above a lower bound, below an upper bound, or between an upper and a lower bound
- Length check – ensuring that an acceptable number of characters has been entered
- Type check – ensuring that only data of the correct data type (e.g. integer) is entered
- Format check – ensuring that input matches an acceptable sequence of character types, such as for a National Insurance number
- Lookup check – ensuring that any data item entered exists on a list of valid data items
- Any combination of these, as multiple validation checks can be assigned to a single data item

As in other sections, you need to identify, describe, explain and justify your selection:

e.g.	*For the user's email address, I will apply a format check* **[IDENTIFY]**. *There will need to be an @ symbol within the entry, but this should not be at the start or the end. There should also be at least one dot after the @ symbol, which should also not be at the end* **[DESCRIBE]**. *This is a suitable check because all email addresses follow this convention, with a domain following the @ sign consisting of at least one dot* **[EXPLAIN]**. *I could have also implemented a length check, but the format check will catch any email addresses that are too short, and I am not aware of any upper limit to the number of characters in an email address* **[JUSTIFY]**.

2.6: Iterative test data

MARK BAND 1	MARK BAND 2	MARK BAND 3	MARK BAND 4
Identify data that will later be used for testing the solution during development (i.e. not after development).	**Mark band 1 plus:** Expand the range of test data so that the entire solution can be tested in a way that examines all interactivity, functionality and validation.	**Mark band 2 plus:** For each piece of test data, explain why it is needed; what exactly is the purpose of each test?	*The wording in the mark scheme is identical for mark bands 3 and 4, so the emphasis is on providing comprehensive test coverage and justification.*

You need to plan out your test data in advance, and there should be test data for each subroutine. Test data should fit into categories of **typical**, **boundary** and **erroneous**. To illustrate each of these, let's take the example of a system that allows estate agents to list houses for sale that have between 1 and 10 bedrooms:

Typical	Boundary	Erroneous

`3, 4, 6`
Each of these is an acceptable number of bedrooms for this system, and there is no need to test every possible input value.
	`0, 1, 10, 11`
	The values of 1 and 10 are valid, but just barely. These are being tested to ensure that the system accepts them.
	The values of 0 and 11 are invalid, but again just barely. These are used to test that the correct error messages appear.
		`-4`
		Negative numbers should not be accepted by this system.
		`Five`
		This is a string, so should also not be accepted.
		`" "`
		Entering nothing at all is a sensible test, as it can often crash a program.
		`3.4`
		While this is within the acceptable range, it is not an integer, so should be rejected.

As you describe each piece of test data for each subroutine, you should explain why it's needed. You might find it useful to organise your test data into a table (as below). This can help to ensure you don't miss anything out.

Subroutine	Field	Data	Explanation
`sell_house`	`number_of_bedrooms`	`1`	This is boundary data. It is just about valid, and should be accepted.
`sell_house`	`number_of_bedrooms`	`11`	Also boundary data; this is barely outside the valid range, but should trigger an error message.
`sell_house`	`number_of_bedrooms`	`3.4`	While in the acceptable range, this is the wrong data type, so should trigger a different error message from the previous test.

2.7: Post-development test data

MARK BAND 1	MARK BAND 2	MARK BAND 3	MARK BAND 4
Post-development test data is not required for mark band 1, provided that it has been provided for 2.6 – iterative development. If not, see mark band 2 →	Identify data to be used in testing after the solution has been completed.	**Mark band 2 plus:** For each piece of test data, explain why it is needed; what exactly is the purpose of each test?	*The wording in the mark scheme is identical for mark bands 3 and 4, so the emphasis is on providing comprehensive test coverage and justification.*

This is test data that will be applied after development has completed, unlike the previous section which is used to test the solution as it is being developed. You can use exactly the same approach, and should justify each set of test data.

The main difference is that here you're testing the whole program, whereas in section 2.6 you're testing each part in turn.

Testing is not something that is considered only when coding is complete. You need to design your tests, conduct testing as you develop, conduct testing post-development and finally evaluate the outcome of testing. You are advised to look ahead to sections 3 to 6 in order to better plan your tests.

Design » Checklist

MARK BAND 4:	13–15 MARKS

- ☐ The problem has been broken down in a systematic manner, such as using a hierarchy chart, and there is a clear explanation of why the problem was broken down in the way that it was broken down
- ☐ The structure of the solution is defined down to the level of each algorithm, with clarity in terms of how the algorithms relate to each other
- ☐ Every individual algorithm, as identified by the point above, is defined using either pseudocode or a flow chart
- ☐ The presence of each algorithm is explained and justified, to make it clear why it is incorporated into the solution in the way it has been incorporated
- ☐ All usability features are identified, described and explained, with choices justified
- ☐ All key variables and data structures are identified, described and explained, with choices justified
- ☐ All validation is identified, described and explained, with choices justified
- ☐ Test data that will be used during iterative development is outlined in detail and justified
- ☐ Test data that will be used after iterative development, as a basis for evaluation, is outlined in detail and justified

MARK BAND 3:	9–12 MARKS

- ☐ The process of breaking down the problem is explained, and is still expected to be systematic, but without the justification
- ☐ The structure of the solution is specified down to the level of each algorithm
- ☐ All subroutines need to be defined using either pseudocode or a flow chart, and their place in the solution will be explained, but there is not a justification as to how they fit in with the rest of the solution
- ☐ All usability features are described and explained, but not justified
- ☐ All key variables are described and explained, but not justified
- ☐ All validation is described and explained, but not justified
- ☐ Test data that will be used during iterative development is outlined in detail and justified
- ☐ Test data that will be used after iterative development, as a basis for evaluation, is outlined in detail and justified

MARK BAND 2:	5–8 MARKS

- ☐ The problem is broken down into smaller problems, with commentary on this process as it happens
- ☐ The structure of the solution is defined, although there may be gaps or errors, and it may not be broken down to algorithm level
- ☐ All algorithms are defined in pseudocode or flow chart form, but there is no additional work provided to explain or describe these algorithms
- ☐ The variables are identified, as are necessary validation routines
- ☐ Usability features are described
- ☐ A range of iterative test data is defined
- ☐ A range of post-iterative test data is defined

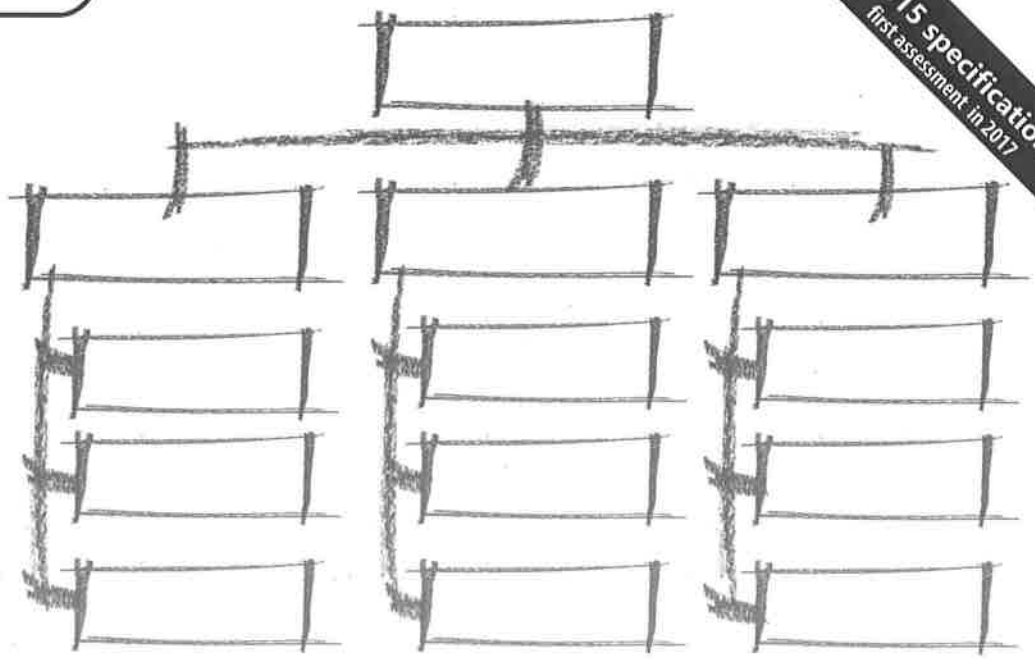
MARK BAND 1:	1–4 MARKS

- ☐ There are some algorithms defined using at least an attempt at a standard convention
- ☐ Some usability features are described, but there could be gaps
- ☐ The variables are identified
- ☐ Some test data is presented for **either** the iterative phase **or** the post-iterative phase

Zig Zag Education

Computer Science A Level | OCR | H446

2015 specification
first assessment in 2017



NEA Companion

for A Level OCR Computer Science

R Lee

zigzageducation.co.uk

POD 10096

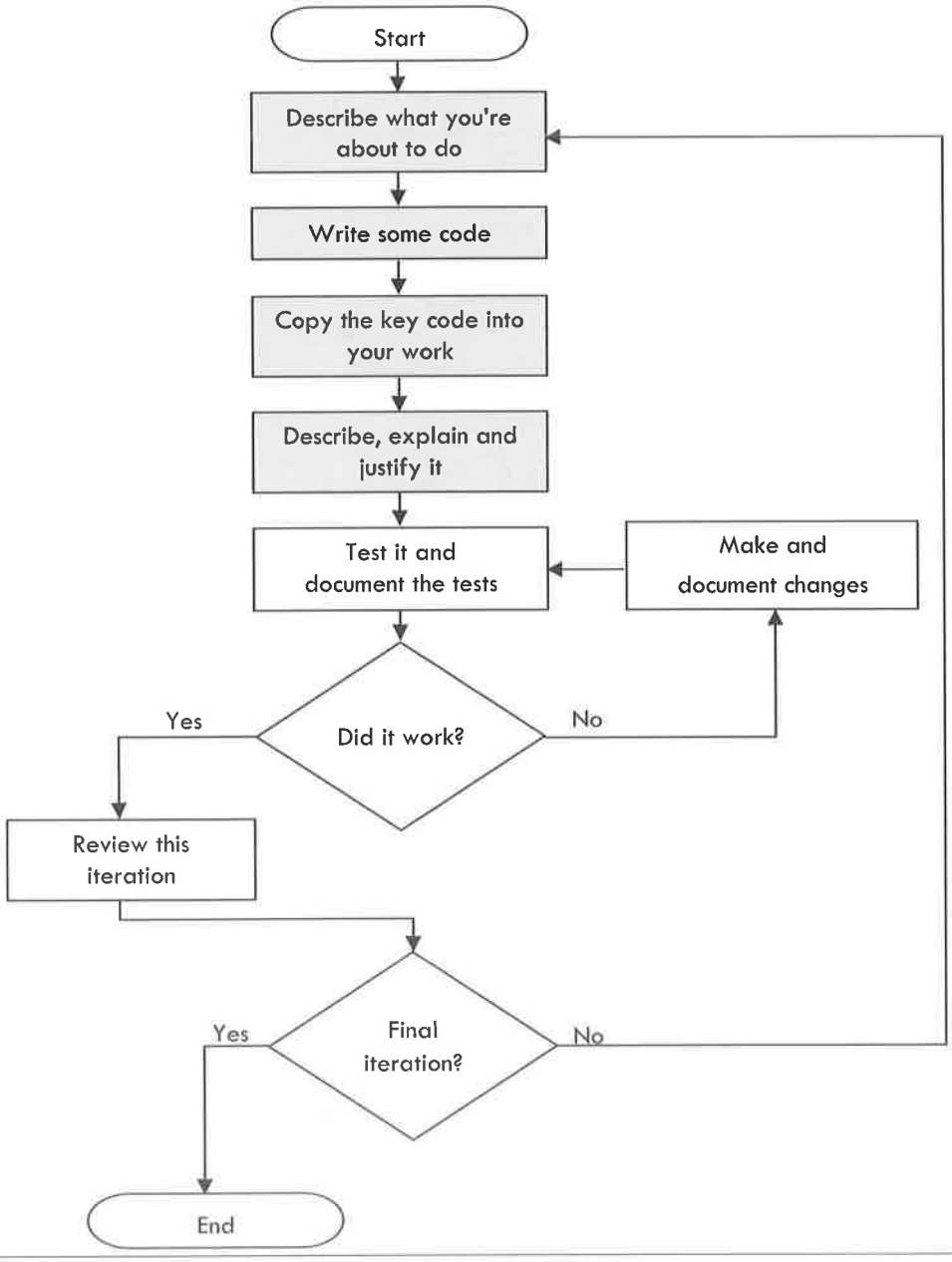
3: Iterative Development (15 marks)

One of the dangers, especially for talented programmers, is the temptation to keep on developing until the solution is complete. While that might result in a good-quality program, it could cost you some marks, because you can only get credit for what you document.

Iterative development entails making part of the solution, or a version of it, then testing it, documenting the tests, reflecting on what you have done, then moving on to the next part or the next version. It's a cycle, and it needs to take place in tandem with section 4, iterative testing. For this reason, you should aim to complete section 3 and section 4 at the same time, rather than as separate sections within your coursework.

Mark band 1	1–4 marks
Mark band 2	5–8 marks
Mark band 3	9–12 marks
Mark band 4	13–15 marks

Overview:



3.1: Iterative development stages

It cannot be overstated that you should not simply create a program, from start to finish, and submit it. As this is not an iterative approach, you would not even get into mark band 1. Instead, you should aim to produce the program piece by piece, documenting as you go.

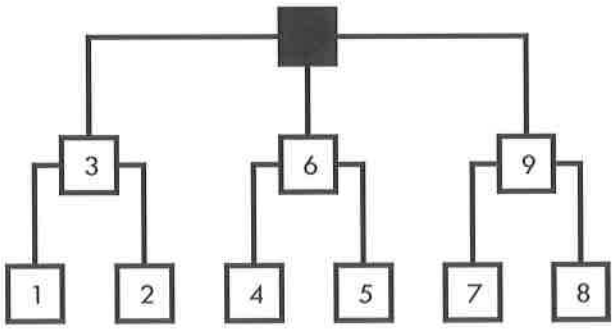
MARK BAND 1	MARK BAND 2	MARK BAND 3	MARK BAND 4
Provide some evidence that development of the solution was iterative.	**Mark band 1 plus:** Add descriptions of each iterative stage, and make sure most or all stages are covered by your evidence.	**Mark band 2 plus:** Ensure coverage of every stage of development, providing a link to the breakdown provided in sections 2.1 and 2.2. Include evidence of more than one prototype at intermediate stages.	**Mark band 3 plus:** Whenever your evidence shows that a decision was made, describe the necessary decision and justify your choice. Evidence of prototypes should be submitted for each iterative stage.

The flow chart on the previous page shows how each iterative development stage should be addressed. In the design phase, you will have broken your solution (the problem too, but this is about the solution now) into pieces, and each of those pieces is one iterative development stage.

> **!** There is no straightforward answer to the question 'how many iterative stages do I need?' It depends on the pieces that make up your solution. Even then, it's possible to do a large number of small cycles, each focusing on a smaller part of the solution, or a smaller number of large cycles. Either is valid.

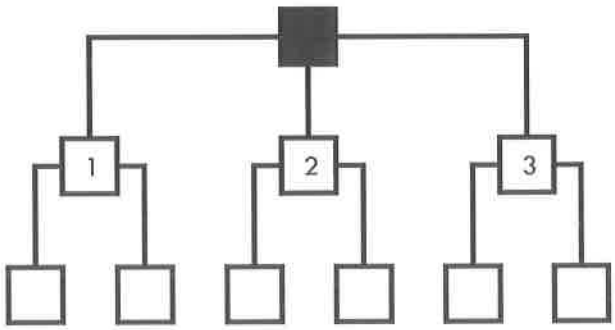
A. Here, the first iterations would be '1', then '2', followed by combining them together in '3'.

The rest of the subprograms would be similarly tested and developed, giving 10 cycles (including a final one that brings all of the parts together).

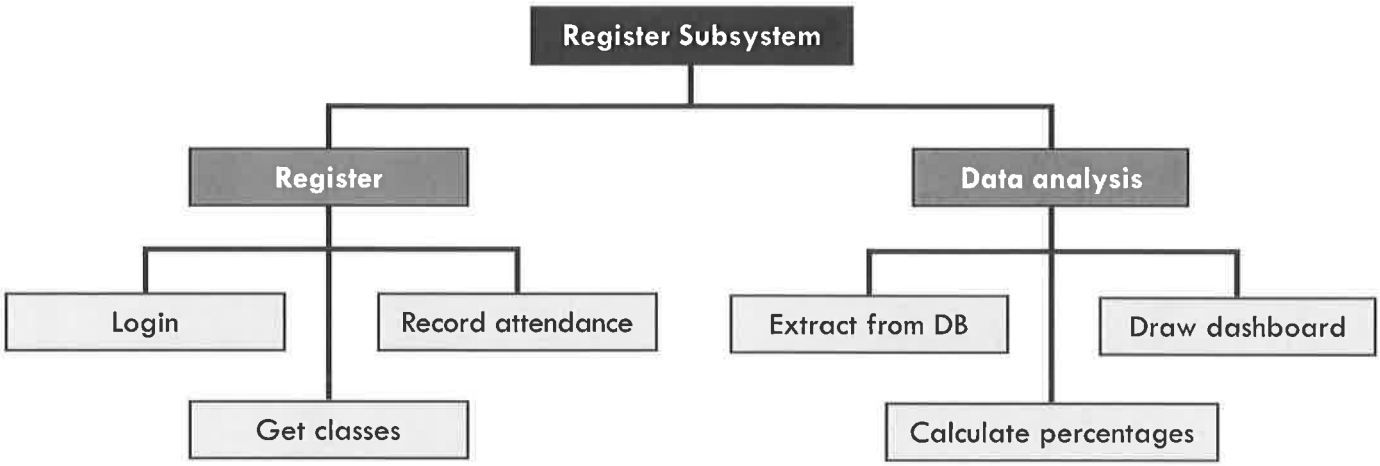


B. In this approach, cycle '1' covers that node as well as the associated child nodes.

This would result in four cycles (again, including a final one that brings the pieces together), but each cycle would include more commentary, more code and more testing.



Let's consider this with the specific example of a register subsystem:



A. We could develop and test the 'login' subsystem, then the 'get classes' subsystem, then the 'record attendance' subsystem, before finally assembling the 'register' subsystem, which contains all of these other parts.

B. Alternatively, we could spend a longer time developing the 'register' subsystem as a single element, without first developing the subparts.

There is no single 'best' way to conduct or document your development, but the following pointers should be observed:

- Describe everything you're doing and explain why you're doing it – there should never be a screenshot in your work without associated text
- Provide evidence in the form of code and screenshots throughout each stage
- Ensure that any code you include is fully commented

3.2: Modularity

MARK BAND 1	MARK BAND 2	MARK BAND 3	MARK BAND 4
Modularity is not required for mark band 1.	Show clear evidence of structure, along the lines of subroutines, loops and/or classes.	**Mark band 2 plus:** Ensure that your program is modular and clearly signpost the features that make it modular.	**Mark band 3 plus:** Evidence of good-quality program structure, which is also well signposted.

You are not required to have a separate 'modularity' section of your work; you are simply expected to demonstrate modularity in your solution, with evidence of it provided in each iterative cycle.

It's beyond the remit of this resource to teach you how to incorporate good structure and modularity into your work (especially as you have a choice of languages), but here are some examples of modular, well-structured code:

Modular and well structured	Not modular and poorly structured
Breaking your program into separate, self-contained pieces (called, generically, modules), which can be independently tested.	A single, typically quite long, body of code, which cannot be tested until the whole solution is complete.
Code that is frequently needed is placed into subroutines and called whenever required.	Completely identical sections of code can be found within the solution.
Iteration is used whenever a single activity is to be repeated or when a very similar sequence of activities are required in sequence.	Solution is characterised by sections of code that have been copied and pasted, then slightly changed.
Arrays, trees, other data structures and your own custom-written classes are used to store data.	Large numbers of variables are used in situations in which they each have a very similar role to one another. NB Variables are not 'bad' – in countless cases, they're the right tool for the job, but if 20 variables, similarly named, perform a very similar task, a 20-element array might have been a better choice.
Attributes and constants are located together within the code, giving any subsequent developer easy access to them. Those attributes and constants are then used in calculations throughout the code, rather than the string/numeric literals.	String and numeric literals are used throughout the code, which makes maintenance very difficult. If your solution were to, for example, calculate VAT, there should be a constant of that name, set to 0.2 (VAT is, at time of writing, 20% in the UK). There should not be countless instances of 0.2 throughout the code.
Classes have private attributes and variables are local, passed as parameters whenever required.	Classes have public attributes and global variables are used in place of parameter passing.
Classes demonstrate high cohesion – each class models the behaviour of a single real-world equivalent fully, without including code that's not relevant to the class.	Classes may not be well defined. A 'student' class, for instance, might represent aspects of teachers, classes or rooms, rather than those pieces of functionality belonging to separate 'teacher', 'class' or 'room' classes.
Classes demonstrate loose coupling – data shared between classes is kept to the minimum of what is necessary.	Classes demonstrate tight coupling, with parameters, return values and public attributes sharing more than is needed.

Ultimately, modular programming works in your favour. Not only does it gain you marks, it also means you have to produce fewer lines of code. That means less time spent documenting your work and fewer opportunities for mistakes.

Even though there's no discrete 'modularity' chapter in your work, you should make a big deal of the features that make your work modular. In the iterative development section, whenever you have produced code that shows characteristics of modularity, draw the reader's attention to it. Describe the feature you have used, describe alternative approaches you might have taken, and justify yourself – why did you do it this way?

3.3: Annotation

MARK BAND 1	MARK BAND 2	MARK BAND 3	MARK BAND 4
Annotation is not required for mark band 1.	Code must include some comments, which will focus on the most important elements.	**Mark band 2 plus:** Comments must extend to at least one per subroutine and one per variable.	**Mark band 3 plus:** Comments must be included throughout the code, and must be of use to any subsequent developer.

Annotation does not require its own section in your work, and you do not even need to signpost it. You will be assessed on the comments/annotations as they exist in your code.

The question most commonly asked is 'how much do I need to annotate?' This is a difficult question to answer, as much of it depends on how the person marking your work interprets the mark scheme. The phrasing *'code will be annotated to aid future maintenance of the system'* isn't conclusive, so asking the person who will mark your work would be a useful move.

There is no such thing as too much annotation, but you don't want to spend hours adding unnecessary comments that might not be worth any marks. The following should always be commented, assuming you're aiming for mark band 4:

- Every subroutine, typically with a comment on the line above the declaration, covering what the subroutine does, what other subroutines it calls, where it is called from itself and details of parameters and return values
- Every variable, with a comment either immediately above it or to the right of it, explaining its purpose
- Any sections of code that are key to the subroutine, or that are complex enough to require explanation.

For example:

```
1  'Main method (start of program), which calls 'QuickSortRecursive'
2  'No parameters
3  Sub Main()
4      'array to be sorted, including positives, negatives and duplicates
5      Dim data() As Integer = {-1, 25, 0, -58964, 8547, -119, 25, 0, 78596}
6      'Call to 'QuickSortRecursive', passing the array, with left and right pointers
7      QuickSortRecursive(data, 0, data.Length - 1)
8      'Display each value in the sorted array on a new line
9      For x = 0 To data.Length - 1
10         Console.WriteLine(data(x))
11     Next
12     'Pause execution before closing
13     Console.ReadLine()
14 End Sub
```

Note that not every single line is commented, even in this very short subroutine. A single comment is enough to cover the loop on lines 9–11, but the declaration has two lines of comment.

Your target audience for this activity is any subsequent programmer. They should be able to understand what each part of your code does, without reading any of the actual code.

Some languages offer facilities for marking multiple lines as comments at the same time. Where these features exist, such as in Java or C#, they should be used.

3.4: Naming conventions

MARK BAND 1	MARK BAND 2	MARK BAND 3	MARK BAND 4
Naming conventions is not required for mark band 1.	Some names for variables are appropriate in that they are self-documenting and follow some form of convention.	**Mark band 2 plus:** Extend the self-documenting naming and standard convention to most variables and data structures.	**Mark band 3 plus:** Ensure that everything that can be named is given a consistently styled self-documenting name.

As with annotation, credit for naming conventions will be given purely on the basis of your code. Anything for which you choose a name can influence how well you score in iterative development, including the following:

- Variables
- Classes
- Files
- Field names within files
- Forms
- Form controls such as buttons and text boxes

Good Names	Bad Names	Explanation
`levelInput`	`input`	Ignoring the fact that 'input' is a reserved word in some languages, this is not a meaningful enough name, as the user might input several things as the program runs.
`player1Score`	`p1s`	Nobody wants variable names to fill up the screen, but a longer-than-average variable name is less of a problem than a variable name that has a purpose you can't remember.
`btnOK`	`Button1`	Default names (such as 'Button1') are only appropriate by coincidence, such as if this button were a calculator button with a value of '1'. When it comes to form controls, employ a convention (such as prefixing all buttons with 'btn' and all text boxes with 'txt'), and use that convention consistently.

There's no reason to get anything other than full credit for naming conventions, as a bad name takes about as much time to type as a good name. Given the fact that, other than the code itself, you don't need to submit anything under 'naming conventions', this might be the easiest part of the project in which to pick up marks.

3.5: Validation

MARK BAND 1	MARK BAND 2	MARK BAND 3	MARK BAND 4
Some attempt at validation is required to reach the upper ranges of mark band 1.	**Mark band 1 plus:** Basic validation should be included and signposted.	**Mark band 2 plus:** The majority of data that can be subject to validation is validated, and validation routines are appropriate to the data.	**Mark band 3 plus:** Validation routines, appropriately chosen and implemented, should be applied throughout the solution, wherever applicable.

The word 'validation' is mentioned in the design section as well as the implementation section. In post-development testing, the word 'robustness' also involves validation. If you've chosen a project where validation cannot be implemented at all, you will lose marks.

It's possible to pick up full marks for validation simply by submitting your code. However, in order to make life a little easier for the person marking your work (and to reduce the likelihood of some of your hard work being overlooked), you are strongly recommended to signpost it. This would mean including a 'validation' subsection in each iterative cycle in which validation has been implemented.

Validation type	When to use it
Presence check	This ensures that the user enters some data, without any consideration of what that data is. A review might use this type of check, since it can be any length, and can include any type of character.
Range check	Useful with numbers or dates, where data values must be below/before a certain value, or greater than / after a certain value (or both). Useful to prevent wildly inappropriate quantities on an ordering system or to prevent insurance policies being purchased and then backdated.
Length check	Making sure a certain number of characters is entered. Like a range check, there can be an upper bound, a lower bound or both, as appropriate to your solution. Usernames and passwords are often subject to length checks.
Type check	Prevents numbers being entered in text-only fields and vice versa. Useful in forms in a wide range of situations.
Format check	Ensures the entry of the correct character types in the correct order. Email addresses, for example, must have an @ sign, but not at the start or end. There must be a full stop after the @ sign, but this can't be at the end, or immediately after the @ sign. Format checks are also useful for postcodes and National Insurance numbers, and they require somewhat more involved coding than most of the others.
Lookup check	This is applicable where every single valid entry that could be made by the user is on a list somewhere. This might be a dropdown list, from which a month is chosen, or it might be a text file containing every word in the English language. The latter would be useful in a lookup check applied to a word game. Presenting the user with a range of buttons (such as in choosing where to play in a game of chess) is also a type of lookup check.

When you're incorporating validation into your solution, you're making a decision, which means that you can pick up marks from part 3.1 if you justify that decision.

Descriptor	How to get it
Mark band 1: **Identify**	State the validation routine you have added; for example, 'length check applied to the creation of a new username'.
Mark band 2: **Describe**	Add some specifics. In the instance of a length check, there will be either an upper bound, a lower bound or both. For example, 'a length check has been applied to new usernames; any entry that is over 16 characters or under 8 characters will trigger an error message that tells users the minimum and maximum length of a new username'.
Mark band 3: **Explain**	Explain can mean 'why' or 'how'. In this instance it means both. In answer to 'why', you should explain your choice of validation routine, as well as you choice of an acceptable range being defined as 8 to 16 characters. In terms of 'how', this is when you would present your code and walk the reader through it.
Mark band 4: **Justify**	In this instance, a length check was chosen, although other options would have worked. A format check could have been used to ensure that the username consisted only of letters and numbers. A lookup check could have been used to check that the username didn't already exist. Why not one of these? Why not a combination of multiple validation checks?

3.6: Review

MARK BAND 1	MARK BAND 2	MARK BAND 3	MARK BAND 4
Formal review is not required for mark band 1.	Comment on the success or otherwise of the solution at some point during its development.	**Mark band 2 plus:** Extend the review to most key stages throughout the process.	**Mark band 3 plus:** Review throughout the iterative process.

A review is a mini evaluation at the end of each iterative cycle. To get into the top mark band, you need to have a review at the end of every individual iterative cycle, and enough iterative cycles to cover development of your whole solution.

To reach the top of that top mark band, the reviews need to be of a good quality, covering the following:

- A summary of what you've done in the outgoing iteration, ideally linked to your 2.1/2.2 breakdowns
- A comparison of what you accomplished with what you set out to do
- A description of what went well and what you found easy
- A description of what went poorly and what proved to be challenging
- Input from the stakeholder(s) as to their opinion of the solution so far
- Lessons that you have learned that might influence your approach to subsequent iterations
- An overall assessment of the success or otherwise of the iteration.

Iterative Development » Checklist

MARK BAND 4:	13–15 MARKS

- ☐ Collectively, the evidence for the iterative stages covers your entire solution (i.e. all of your code is documented in one or more of the iterative stages)
- ☐ Each iterative stage is clearly linked to the breakdown of the problem documented in section 2.1
- ☐ Each stage is backed up with detailed descriptions (what are you doing?), explanations (why are you doing that?) and justifications (why did you do A when you could have done B?)
- ☐ Each iterative stage constitutes a prototype; this does not mean you need to have a fully formed version of your solution at each stage – simply that you should have produced something self-contained, usable and testable
- ☐ Code demonstrates modularity and good structure at all stages
- ☐ Annotations cover all subroutines and all variables throughout your code
- ☐ Every single variable, along with anything else you have chosen a name for, has a self-documenting identifier, such that the purpose is clear just by reading the name
- ☐ Evidence of validation is provided at all appropriate stages
- ☐ All iterative stages are reviewed, with the review then feeding into the next iterative stage (the review for the final iterative stage covers what you might have done next, had the process continued)

MARK BAND 3:	9–12 MARKS

- ☐ Iterative stages are still required to cover your entire solution
- ☐ Each stage should still be linked to the breakdown of the problem documented in section 2.1
- ☐ Each stage should be described (what did you do here?) and explained (why did you do it?), but justification is not required in this mark band
- ☐ Prototypes (plural) are required, but not for every iterative stage
- ☐ The solution needs to be modular; classes as required, and subroutines called when necessary, with minimal duplicate code
- ☐ The majority of the code should still be commented, although parts that are trivial, of obvious functionality or a near-copy of already-commented code might be missing comments
- ☐ Most variables and data structures have sensible, self-documenting identifiers, although there are some exceptions to this
- ☐ Validation is implemented and documented more often than it is missing
- ☐ A minority of stages might be missing a review section

MARK BAND 2:	5–8 MARKS

- ☐ Some parts of your code, which may be evident in the full listing in your appendices, are not demonstrated or discussed in this section
- ☐ Some skill is demonstrated in making the solution modular and well structured, but duplicate code might exist, and the most efficient means of solving a problem might not always be apparent
- ☐ Some annotation, which is useful, rather than simply of a token presence, is included, but not throughout
- ☐ Some variables and data structures have self-documenting identifiers
- ☐ Some useful validation is included and documented
- ☐ Reviews exist for some iterative stages, and offer some genuine insight

MARK BAND 1:	1–4 MARKS

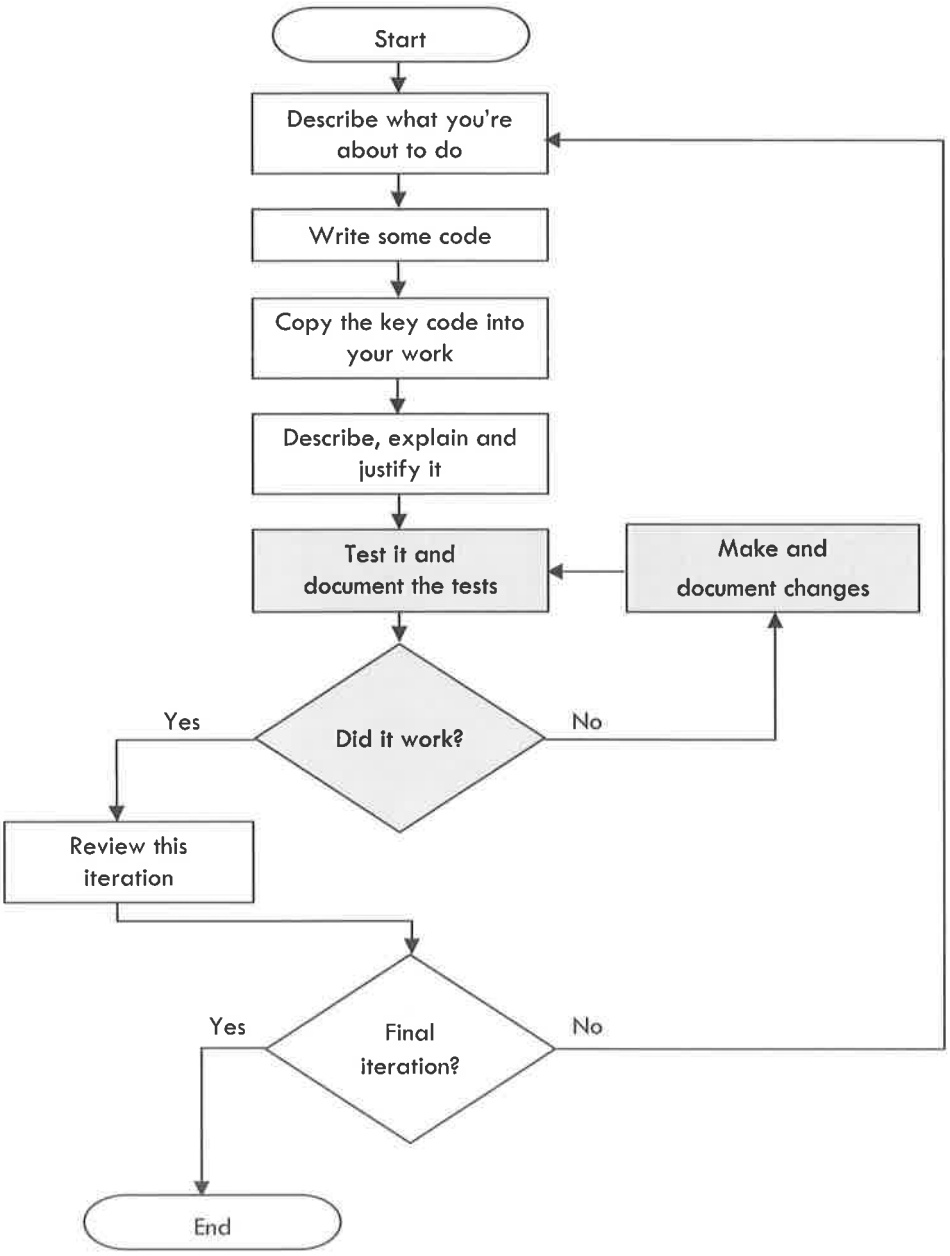
The mark scheme for mark band 1 is a list of aspects that are missing rather than aspects that are included. If the majority of modularity, annotation, self-documenting identifiers, validation and review are either missing, or only exist in some token way, you can expect a score from mark band 1. The more that's missing, the lower that score will be.

4: Iterative Testing (10 marks)

Iterative testing, unlike post-development testing, takes place at the same time as iterative development. Once you have produced some testable code, you should test it, document your tests, fix any problems that have arisen and go back to development. In order to pick up all of the marks in this section, it should be completed at the same time as section 3, iterative development.

Mark band 1	1–2 marks
Mark band 2	3–5 marks
Mark band 3	6–8 marks
Mark band 4	9–10 marks

Here, we focus on the shaded areas of the flow chart:



Every time you run your program, whether it's working or not, you're conducting a test. You're not expected to document every single one of these, as there could be thousands, but you are expected to show every part of the solution being tested, which will involve a large number of tests that are not successful.

4.1: Testing

MARK BAND 1	MARK BAND 2	MARK BAND 3	MARK BAND 4
Provide some evidence of testing throughout the iterative process, i.e. before the solution is complete.	*The wording in the mark scheme is identical for mark bands 1 and 2, so the emphasis is on the frequency and quality of testing.*	**Mark band 2 plus:** Extend testing to cover *most* of the iterative development stages.	**Mark band 3 plus:** Extend testing to cover *all* of the iterative development stages.

Testing at this stage should be informed by what you planned in stage 2.6 (iterative testing design). You only need to test any new code added since the last iteration. If you have deviated from the 2.6 plan, that is absolutely fine. There is no need to go back and edit section 2.6, but you should explain what changes you have made since the design phase, and why they were necessary ('there was insufficient time to complete this part' is, incidentally, a perfectly legitimate reason).

4.2: Remedial actions

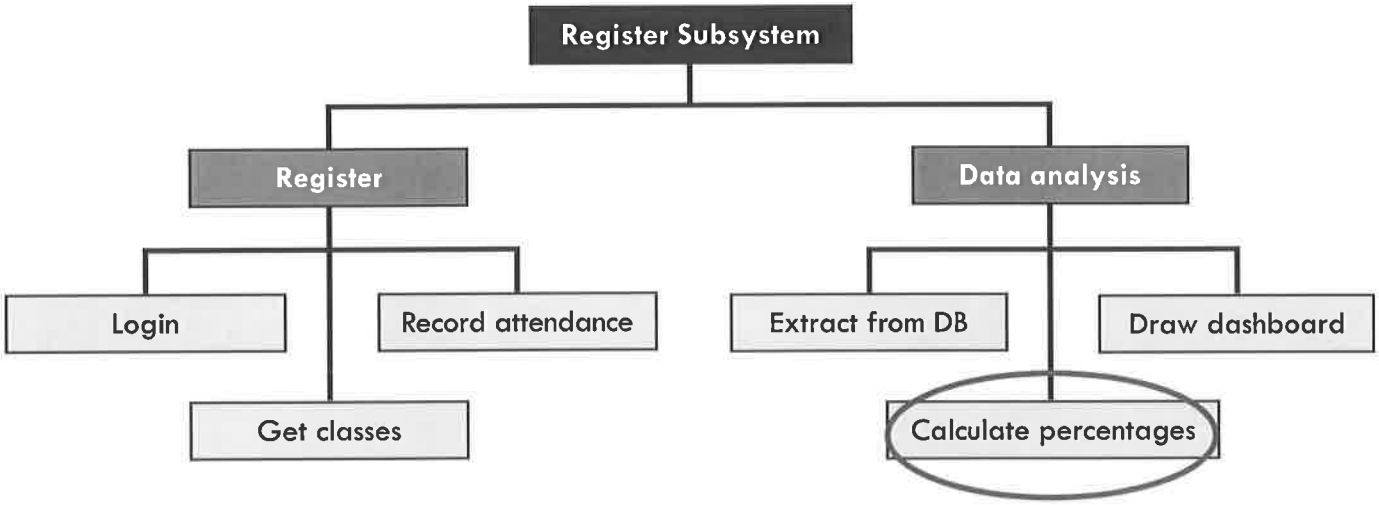
MARK BAND 1	MARK BAND 2	MARK BAND 3	MARK BAND 4
Remedial actions are not required for mark band 1.	Demonstrate, with evidence and commentary, how at least two tests failed to produce the expected outcome, and what changes were made as a result.	**Mark band 2 plus:** Include explanations of why those particular changes were made.	**Mark band 3 plus:** Ensure all failed tests are addressed, with the commentary on changes extended to cover alternative plausible approaches to fixing.

Unless you have failed tests, together with responses to those failures in order to remedy them, your iterative testing section is not realistic. Essentially, you're making the claim that everything worked perfectly first time, and that not a single mistake was made.

This part of the project is best addressed through the use of annotated screenshots. Show the reader what happened and talk them through it:

1. Remind them of the test data, as spelled out in section 2.6
2. Show them the failed test via a screenshot and annotated code, and explain what's happened
3. Fix the code and show it to the reader, highlighting and explaining any changes
4. Retest (and repeat from step 2 if it still doesn't work)

Sample iteration



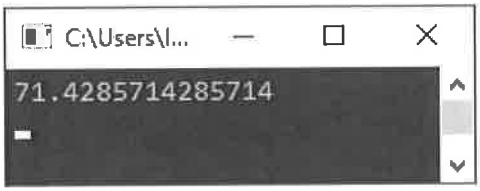
In this phase, I will create the code that will calculate the percentage attendance of a single individual based on number of classes attended and maximum number of classes. Ultimately, my system will create a dashboard comprising all attendees, sortable into either alphabetical order or order of attendance. Before that dashboard can be created, I am going to ensure that a single individual's data is correctly calculated. This way, if an error occurs, that error exists only once, rather than repeatedly.

I am creating the code independently of the interface for now, as the interface is part of the 'draw dashboard' subsystem. Accordingly, all data will be output to the console.

```
1   Sub Main()
2
3     'Array, which will normally be populated from DB
4     Dim attendanceArray() As Char = {"\", "\", "O", "A", "\", "\", "\", "O"}
5
6     Dim possible As Integer = 0 'number of possible attendandes
7     Dim actual As Integer = 0 'number of actual attendances
8
9     'loop through the array
10    For loopCount = 1 To attendanceArray.Length
11          'don't count A for authorised absences, otherwise increment possible
12          If Not attendanceArray(loopCount - 1) = "A" Then
13                possible += 1
14          End If
15          'for a present mark, increment actual
16          If attendanceArray(loopCount - 1) = "\" Then
17                actual += 1
18          End If
19    Next
20
21    'calculate attendance as a percentage and display
22    Console.WriteLine(actual / possible * 100)
23    Console.ReadLine()
24  End Sub
```


The attendance array would normally be populated with data extracted from the appropriate record of the database, but for now it is populated with the test data specified in my design. I chose this data as it includes all possible attendance marks – a slash for present, a capital O for absent and a capital A for an authorised absence. Authorised absences should be ignored by the system and should not affect the attendance figure.

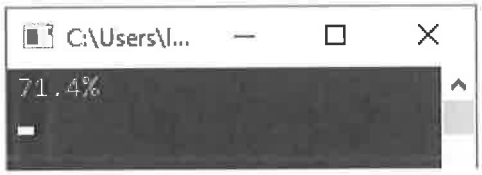
The attendance in this case should be 71.4%.



Having tested this subroutine, it is a success, but I note that far more decimal places are used than are necessary. It might also be useful to have the percentage sign displayed. The target is for the output to be '71.4%' for this data set, and line 22 has been modified as follows:

```
'calculate attendance as a percentage and display
Console.WriteLine(Math.Round(actual / possible * 100, 1) & "%")
```

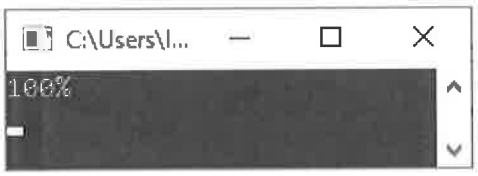
Here, I have used the `Math.Round` library function to display the answer to a single decimal place. I have then used the ampersand concatenation operator to append a percentage sign. This approach was chosen because the values of 'actual' and 'possible' will remain unchanged and can be used in subsequent calculations. This can now be retested:



This retest has demonstrated the desired result. In order to ensure that this part of the solution is working perfectly, I need to test it with a range of data sets:

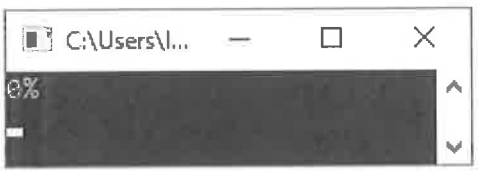
```
Dim attendanceArray() As Char = {"\", "\", "\", "\", "\", "\", "\", "\"}
```

To test that 100% attendance is correctly calculated:



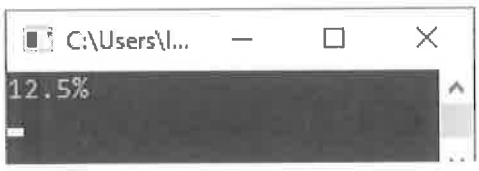

```
Dim attendanceArray() As Char = {"O", "O", "O", "O", "O", "O", "O", "O"}
```

To test that 0% attendance is correctly calculated:



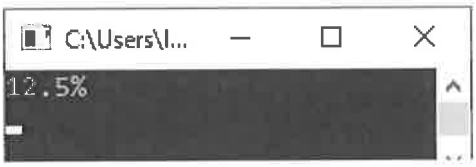

```
Dim attendanceArray() As Char = {"\", "O", "O", "O", "O", "O", "O", "O"}
```

To check that the loop picks up a single 'present' mark in the first element of the array (12.5%):




```
Dim attendanceArray() As Char = {"O", "O", "O", "O", "O", "O", "O", "\"}
```

To check that the loop picks up a single 'present' mark in the last element of the array (12.5%):



I am not testing for invalid data (i.e. a character other than \, A or O) as validation is handled elsewhere in the system – in the 'Record Attendance' subroutine. This is a fairly small part of the system, but as it is central to the whole solution, it was appropriate to have created it and tested it thoroughly before proceeding. I note that the single decimal place is not displayed when the percentage figure is a whole number, but I do not believe this is a significant enough anomaly to spend any additional time here – there is nothing ambiguous about '100%' or '0%'.

This part of the solution will now be tested alongside the 'Extract from DB' subroutine, to ensure that data extracted from the database has the same effect as data hard-coded into the array.

This is a single iterative cycle in a development phase that would contain many such cycles, given the small scale of what it does. Nevertheless, all of the pieces are there. There is a description of what is about to be coded, placed in the context of the whole system. Code (which is modular, annotated and self-documenting) is written, described and tested. The outcome suggests that a modification should be made, which is coded and retested before being reviewed. At all times, we are being told *why* an action is being taken. Why this test data? Why the console output? Why no validation? The last sentence then leads seamlessly into what the next iterative cycle would be.

Iterative Testing » Checklist

MARK BAND 4:	9–10 MARKS
☐ Each stage of the iterative development process involves testing; note that you are only eligible for this mark if your iterative development work contains enough stages (i.e. you are in mark band 3 for section 3) ☐ There is evidence, and more than simply token evidence, of tests failing ☐ Documentation has been included as to what was done to remedy all failed tests ☐ Remedial action (action to fix code after a failed test) is described (what did you do?), explained (why did you do it?) and justified (why did you do A when you could have done B?)	
MARK BAND 3:	**6–8 MARKS**
☐ Most iterative stages involve testing, but not necessarily all; you might have tested all of a partially documented solution, or part of a fully documented solution ☐ Some failed tests are included, though not necessarily in all iterative stages ☐ Documentation has been included as to what was done to remedy all apparent failed tests ☐ Remedial action is described (what did you do?) and explained (why did you do it?), but not justified	
MARK BAND 2:	**3–5 MARKS**
☐ Some of the iterative stages include testing **Note that the mark scheme descriptor '... provided some evidence of testing during the iterative development process' covers mark bands 1 and 2, so is anywhere between 1 and 5 marks. To score at the top of this band, you would expect around half of the iterative stages to include some documented testing. More than this would be '... most stages of the iterative development...', which would place it in mark band 3.** ☐ Some failed tests are included ☐ Remedial actions are described, with supporting evidence, but not explained	
MARK BAND 1:	**1–2 MARKS**
☐ Testing needs to have taken place during iterative development; if there is no testing until after development has concluded, no marks can be awarded ☐ No failed tests or remedial actions are expected in this mark band	

5: Post-development Testing (5 marks)

This is the final round of testing, after which no additional development takes place. If you use the outcome of any testing to improve your solution, you're still in section 4. This is the smallest section in terms of the marks allocated to it, so you should spend the least time here. Nevertheless, it's still important, and a good post-development testing section can make the difference between two grades in the coursework.

Mark band 1	1 mark
Mark band 2	2 marks
Mark band 3	3–4 marks
Mark band 4	5 marks

In the iterative development section, you were testing smaller pieces of the overall solution as you developed them. Here, you should aim to test whether those separately developed pieces work correctly together. At this point, a formal test table can be used, which should be informed by the test data you set out in section 2.7. The test table, presented in landscape, should contain the following headings:

1. A test number; you'll want to refer to individual tests in the evaluation, so this will be needed later
2. A description of the test, in sufficient detail that a competent person could read it and carry out the test exactly as you carried it out
3. Test data; for example, when you tested the login screen, exactly what username and password were entered?
4. Expected outcome; this is a description of what a successful test looks like, and should be specific enough that a competent person could judge a passed or failed test without any other input from you
5. The actual result, which will be either 'success' or a description of exactly what happened
6. A reference to any supporting evidence, such as 'see screenshots #7 and #8' (it's best to avoid placing the screenshots inside the table itself, as space is at a premium)
7. Commentary; a description of what that test shows us and why it was needed, ideally cross-referenced to the original user requirements

Test #	Description	Test Data	Expected Outcome	Actual Outcome	Evidence	Commentary

A test table is not essential; it is not mentioned in the mark scheme, and some of OCR's exemplar candidate work uses a more narrative structure. However, use of a table will immediately alert you to any parts that are missing, because the table will contain empty cells.

When providing evidence, you're quite likely to use screenshots of the program in operation, but you might also include screenshots of data stores, such as a database, before and after an operation intended to add, edit or delete data. You might include photographs of a screen if screenshots are unavailable, or photographs of an external output peripheral, depending on your solution. Each of these, placed after the testing table, should be uniquely numbered, with those numbers being entered into the 'evidence' column.

If your test evidence takes place on video rather than screenshot, the 'evidence' column should contain precise time indexes instead.

5.1: Testing for function

MARK BAND 1	MARK BAND 2	MARK BAND 3	MARK BAND 4
Include evidence to show testing of some aspect of your solution after development has concluded.	**Mark band 1 plus:** Ensure that some of the tests cover testing for function.	**Mark band 2 plus:** Provide a commentary in addition to evidence for the tests.	**Mark band 3 plus:** Include additional evidence and commentary to cover robustness in addition to function.

Testing for function answers the question 'does it work?' What you will actually test will depend on the nature of your solution, but you can expect to cover the likes of the following:

- Do calculations provide correct results?
- Are data retrieved from and inserted into data stores as expected?
- Do key functions such as logging in and out work correctly?
- Do validation routines work?

Mark band 4 adds a reference to 'robustness', involving the following questions in addition:

- Does validation prevent the program crashing (such as in the event of logging in with no user name)?
- Does the solution alert me to a missing database, rather than crashing or looping indefinitely looking for it?

When it comes to robustness, you're trying to break your program.

5.2: Testing for usability

MARK BAND 1	MARK BAND 2	MARK BAND 3	MARK BAND 4
Usability testing is not required for mark band 1.	*Usability testing is not required for mark band 2.*	Provide commentary and evidence to show testing for usability.	*The wording in the mark scheme is identical for mark bands 3 and 4, so the emphasis is on the number and quality of the tests as well as the insight offered by the commentary.*

Usability will also vary based on the nature of your solution, but the focus will be on the interaction between the user and the program:

- Do keyboard and mouse inputs produce the required responses?
- Do any accessibility features, such as changing font sizes and background colours, work correctly?
- Do any additional input/output devices, including speakers, function as expected?

In order to enter mark band 3, the commentary is essential. If there's anything about a test that isn't covered in any of the other columns, it should be placed into the 'commentary' column. This is most likely to be explanations of why a test was necessary, or how important a particular test was.

Post-development Testing » Checklist

MARK BAND 4:	5 MARKS
Testing, which includes evidence (such as with a screenshot) and commentary, covers the following: ☐ Function – do the key processes perform as expected? ☐ Robustness – does the solution function irrespective of invalid data input, missing files or anything that might make the solution crash or otherwise malfunction? ☐ Usability – do all aspects of the interface function as expected?	
MARK BAND 3:	**3–4 MARKS**
Testing, which includes evidence (such as with a screenshot) and commentary, covers the following: ☐ Function – do the key processes perform as expected? ☐ Usability – do all aspects of the interface function as expected?	
MARK BAND 2:	**2 MARKS**
☐ Evidence of final testing is included, which tests for function (checking that key processes perform as expected); commentary may be either weak or missing	
MARK BAND 1:	**1 MARK**
☐ Testing needs to have taken place during iterative development; if there is no testing until after development has concluded, no marks can be awarded ☐ No failed tests or remedial actions are expected in this mark band	

6: Evaluation (15 marks)

The evaluation is the point at which you look back at the solution and examine how well it addresses the problem. At this stage, you can actually gain marks for any shortcomings you made earlier, as long as you reflect well on them. The evaluation is informed by all previous sections, as you might wish you had done something differently in each of the analysis, design, development and testing stages. If that's the case, this is where you can talk about it.

Mark band 1	1–4 marks
Mark band 2	5–8 marks
Mark band 3	9–12 marks
Mark band 4	13–15 marks

6.1: Examining success (or otherwise)

MARK BAND 1	MARK BAND 2	MARK BAND 3	MARK BAND 4
Using the results of testing, talk about whether the solution is a success or a failure.	**Mark band 1 plus:** Include each of the success criteria (for part 1.7), commenting on the success or failure of each one.	**Mark band 2 plus:** Include *partial success* as an outcome in addition to 'success' or 'failure' for each success criterion. Describe how future development could address success criteria that have not been fully met.	**Mark band 3 plus:** Justify any choices you have made in ensuring that the solution will meet stakeholder needs.

A poorly written evaluation can be quite woolly, and can lack direction. The best way to begin an evaluation that's assessed by this particular mark scheme is with structure. Towards the end of your analysis, you specified a set of success criteria, which were to be the standards by which you planned to judge the success, or otherwise, of your solution.

Each of those criteria should now become a title to a paragraph, in which you address the following:

1. To what extent have you met that success criterion? You might not have attempted it as you ran out of time; you might have exceeded the standard you set for yourself; it might be somewhere in between. Don't shy away from success or failure in this part of your work, as you get credit for how well you recognise them. You should pay close attention to any success criteria which lie somewhere in the grey area between success and failure.
2. Explain **why** you have come to the conclusion you reached in step 1, and provide evidence to back up your claim. If it doesn't work, show the error message and explain what caused it. If you went further than expected, perhaps line up the design with reality and highlight the differences.
3. Describe and justify any improvements you would make to your solution in order to better address this success criterion in future. Remember, this is a hypothetical future, and you'll never have to actually make these changes, so don't be afraid to be ambitious. Say **what** changes you would make, describe **how** you would go about making them, and finally talk about **why** these changes would make your solution a better one.

If possible, try to get some stakeholder input into the evaluation. That would count as evidence, making it useful for step 2.

6.2: Assessing usability

MARK BAND 1	MARK BAND 2	MARK BAND 3	MARK BAND 4
Usability assessment is not required for mark band 1.	Provide evidence and commentary of usability features that have been set out in section 2.4.	*The wording in the mark scheme is identical for mark band 2 and mark band 3, so the emphasis is on the number and quality of usability features.*	**Mark band 3 plus:** Describe, with explanatory evidence, whether the use of usability features has been a success, partial success or failure.

Usability was initially addressed in section 2.4, where you spelled out which usability features you intended to deploy within your solution. These might have been particular layouts, specific colour combinations, accessibility features or a combination of all of these. Under the heading 'usability', examine how well you have done in this regard, in the same way that you addressed section 6.1:

1. Make a judgement as to the extent to which you have produced a usable, accessible solution
2. Explain, with supporting evidence, how you came to that conclusion
3. Suggest improvements to the usability of your solution, justifying as you go

6.3: Maintenance and limitations

MARK BAND 1	MARK BAND 2	MARK BAND 3	MARK BAND 4
Maintenance and limitations are not required for mark band 1.	Describe limitations of the solution in terms of features it does not include and factors that have prevented the inclusion of additional features.	**Mark band 2 plus:** Include issues of maintainability. To what extent have you produced a maintainable solution?	**Mark band 3 plus:** Describe how changes could be made in the future to reduce/mitigate limitations and to make the solution more maintainable.

Under the heading of 'limitations', talk about what your solution doesn't do that you had planned for it to do. You can also include any shortcomings that became apparent as you developed. For example, perhaps you realised part way through development that a web interface did not display correctly on a mobile device. This might not have made it into your original success criteria, but it can still be discussed under 'limitations'. For each limitation:

1. What was supposed to happen?
2. What, if anything, did happen? Your limitation might come in the form of partial success.
3. Why did this take place? This might be a technical reason, or it might have something to do with an over-ambitious project or an underestimation of the complexity involved.
4. How would you approach problems like this differently in future in order to avoid similar limitations?

'Maintainability' is a measure of how readily another person could make changes to your solution if changes were required. The following are some characteristics of maintainable code:

- **Self-documenting identifiers** – everything you have named, you have named in such a way that its purpose is apparent from the name alone. This applies to variables, data structures, classes, forms, form controls, external files, fields within external files and anything else that you, the programmer, gave a name to.
- **Modularity** – each subroutine should be self-contained, and separately designed, produced and tested, with no reliance on global variables that could have been unexpectedly changed by other subroutines.
- **Appropriate use of variables and constants** – variables and constants should be used in all calculations and other processes, rather than literals. Additionally, these variables and constants should be easy to locate and change. If the VAT rate changes, for example, and your solution uses that rate in some way, there should be a single easy-to-locate change to make to your code to reflect the new rate.
- **Detailed annotation** – there should be comments throughout your code describing what each part of it does. There should never be a section of code without comments.
- **Appropriate version numbers** – given the sequence of prototypes that have been rolled out throughout the project, it's possible that someone might accidentally begin changing the wrong version of your solution. There should be measures in place to ensure that older versions survive, but the latest version is apparent.

Your section on maintainability should mirror previous sections within the evaluation:

1. To what extent is your solution maintainable (using the descriptions above)?
2. Why have you come to that conclusion (provide evidence)?
3. How could your solution be made more maintainable (including a justification of your answer)?

6.4: Quality of written communication

The final element of your work to be assessed is how well written your evaluation is. As you might expect, it's beyond the scope of this resource to teach you how to write well, but there is some advice which, if followed, can make the best use of your current abilities:

- Don't make unsubstantiated claims. If you say that something worked, or didn't work, or partially worked, provide evidence to back this up. Each claim in your evaluation should be supported by a screenshot or a reference to a test number.
- Incorporate structure into your work. This can be done using subtitles in much the same way that they are used here. Have a section entitled 'test results', where you talk about the test results. Then a section about 'usability', and so on. Avoid bouncing back and forth between different topics, as it can throw the reader and cost you marks.
- Include a table of contents to allow sections to be located more quickly (and present a structure?).
- Keep it relevant. You should write about success criteria, testing, usability, limitations, maintenance and potential improvements. Anything else is not creditworthy, and not worth your time.
- Use a spellchecker, read through your work and ask someone else to read through it as well.

For some students, the evaluation can be a little too open-ended. If you're struggling to get words on the page at the start of the evaluation, or a section of the evaluation, the following templates might be of use to you. They don't cover everything, and are not intended to – each evaluation should be unique – but they might help you to make a start.

Upbeat opening	One key strength of the solution that is apparent throughout the testing is _____.
	Although challenges were encountered, including _____ and _____, testing has demonstrated that a great deal of functionality has been provided by the solution.
	During the analysis stage, it was apparent that the stakeholders' highest priority was _____, and this has clearly been delivered, as seen in test _____.
Downbeat opening	While some success has been encountered during the development of this solution, it is important to note that key functionality, namely _____, has not been delivered.
	A shortcoming borne out by the testing, which cannot be ignored, is _____.
	The key success criterion in this project was _____, and tests _____ and _____ provide definitive evidence that this criterion has been met.
Mixed opening	In some respects, this solution is both a success and a failure.
	Not all success criteria have been fully met, but it is important to note that the solution does offer some key functionality, including _____.
	Admittedly, the solution does not provide 100% of its intended functionality – both _____ and _____ are incomplete – but some success has been encountered.
Providing evidence	Evidence for this claim can be found in tests _____ and _____, which clearly show the process of _____.
	Test _____ shows the state of the system before _____, and test _____ shows the state of the system afterwards.
	This can be seen to be the case throughout section _____ of the post-development testing, in particular during test _____ and on the associated screenshot.
Future improvements	Although this solution fully satisfies all success criteria, there is plenty of scope for future development. Firstly...
	Naturally, the first priority in improving the system would be satisfying the presently unmet success criteria, but beyond that...
	There is scope for improvement, with the most likely future focus being...

Evaluation » Checklist

MARK BAND 4:	13–15 MARKS
☐ Evaluation needs to compare all test evidence (iterative and post-iterative) with all success criteria (section 1.7) in order to assess the success of the solution ☐ Each success criterion, including usability and maintainability, should be assessed as a success, partial success or failure, with explanations and evidence to back up each assessment ☐ Limitations should be critically addressed – what are they, and how significant are they? ☐ Future improvements should be suggested for partial successes and failures in terms of success criteria, usability and maintainability	
MARK BAND 3:	**9–12 MARKS**
☐ Evaluation still needs to compare all test evidence with all success criteria (section 1.7) in order to assess the success of the solution ☐ Each success criterion should be assessed as a success, partial success or failure, with explanations and evidence to back up each assessment; usability and maintainability do not need to be included here ☐ Evaluation should still include usability features and maintenance issues, but these only need to be described rather than explained ☐ Limitations should be critically addressed – what are they, and how significant are they? ☐ Future improvements should be suggested, but only with regard to the success criteria	
MARK BAND 2:	**5–8 MARKS**
☐ Comparisons of success criteria and test evidence are still required, but some test evidence and/or some success criteria might not be included in the evaluation ☐ Evidence of usability features should be incorporated into the evaluation in some form ☐ Limitations still need to be addressed, but only in a descriptive way; their significance might not be mentioned	
MARK BAND 1:	**1–4 MARKS**
☐ Determine whether the solution is a success or a failure based on the test evidence; there may or may not be any reference to the original success criteria	

Suggested Project Structure

ANALYSIS

- Stakeholders
- Research of existing solutions
- Essential features
- Limitations
- Hardware and software requirements
- Success criteria
- Computational methods

DESIGN

- Problem decomposition
- Structure of the solution
- Algorithm design
  - Algorithm 1
  - Algorithm 2
  - Algorithm 3
  - etc.
- Usability features
- Variables and validation
- Iterative test data
- Post-development test data

ITERATIVE DEVELOPMENT AND TESTING

- Prototype 1:
  - Introduction and reference to problem decomposition
  - Prototype code
  - Description of code
  - Testing of code
  - Identification of errors
  - Retesting of code
  - Review
- Prototype 2 (as above)
- Prototype 3
- etc.

POST-DEVELOPMENT TESTING

- Test table
- Test evidence

EVALUATION

- Comparing success criteria and test data
- Usability
- Maintenance and limitations
- Future improvements

Glossary

Throughout this resource, there are some important words that you will encounter quite a few times. It's important to understand what they mean, so if you read this and are still unclear, ask your peers, ask your teacher, check definitions online or in a textbook, or do whatever you can to ensure a good understanding.

Annotation	In terms of programming, 'annotation' or 'comments' make up the English language pieces of text that sit alongside program code. Compilers ignore annotation, since it is there for humans, rather than the computer. Good-quality program code is always annotated as an aid to maintainability; it helps the next person who deals with the code to understand it more quickly.
Data structure	Anything more complex than a variable that can store data is a data structure. This includes arrays, lists, graphs, trees, queues, stacks and dictionaries, as well as external files.
Describe	The most open-ended verb in any mark scheme, 'describe' can cover any of 'who', 'what', 'when' and 'where'. You might be describing a piece of existing software (what does it do?), a user interface (where is the 'OK' button?), a potential user of your system (who is the stakeholder?) or a process (when does the validation take place?). Often, it's a combination. If you're writing in general terms about something, you're probably describing it, and 'describe' can be found clustered around the bottom-to-mid mark bands.
Evaluate	The word 'evaluate' comes from the word 'value'. If you're evaluating your solution, you're assessing its overall worth, as well as the worth of any individual features of it. For example, if your interface looks good (positive), but doesn't work (negative), you would need to weigh these two factors against each other. Coming to the conclusion that the failure for it to work outweighs its appearance would be an evaluation.
Evidence	At some point, either in the iterative development phase or during the evaluation, you'll come to the conclusion that something either works, doesn't work, or is somewhere in between. As well as stating that, for instance, 'this causes the program to crash', you should provide evidence. This is usually in the form of a screenshot, but it might vary, depending on what you're trying to prove. Evidence might be in the form of a completed questionnaire if your claim is 'my stakeholder prefers the GUI interface'.
Explain	A common verb among the mid-mark ranges; if you're explaining something, you're usually answering the question 'why', sometimes 'how' and occasionally both. For example, if you're explaining your choice of programming language, why did you choose that language?
Identify	You can identify something in a sentence or less. If you were asked to identify the external peripherals required for your solution, 'mouse, keyboard, monitor' could be enough. Any more detail, such as the resolution of the monitor or the language layout of the keyboard, would be describing.
Iterative	This describes the process of making multiple passes through the same task. In a linear approach to software development, you would design, then produce, then review a solution. In an iterative approach, which is what this project requires, following the design, you produce part of the solution, then review it. After this, you might produce another part, have a second attempt at producing that same part, or even go back to the design and revisit that phase. In an iterative process, some stages take place repeatedly.

Justify	The word 'justify' appears in the mark scheme 18 times, typically in the higher mark bands. Simply put, it means giving an explanation for something you have done. Unfortunately, the word 'explain' also appears quite a lot, which doesn't help matters. To justify something, try this three-step approach: A. Describe what you did – whatever it is you're going to justify. B. Describe what you might have done instead, but didn't. You might have opted to store data in a text file, in which case, you didn't store it in a database. C. Explain why you opted for plan A instead of plan B.
Modularity	This is a measure of how well a problem or a solution is broken into pieces. In terms of software development, you should be aiming to make a modular solution. This might involve the creation of classes, and it will certainly involve representing each individual task with an individual subroutine. Through good use of variables, parameters and return values, modularity reduces the amount of code by minimising duplication of code. If you're copying and pasting large amounts of code, the chances are that you're not developing a modular solution.
Problem	In programming, this simply means the situation for which you are going to write a program. It doesn't necessarily mean there's something wrong, although usually, if you're trying to improve a situation, it's not perfect.
Prototype	A prototype is a version of an unfinished solution. It might be an attempt at the whole solution, or perhaps some part of it. It is typically imperfect, and the main purpose of a prototype is to learn something from it in order to make the next version better.
Solution	This covers a program, together with any associated hardware. It is what you are going to create in response to the problem you identify.
Stakeholder	A stakeholder is anyone with an interest in your solution. The most important stakeholder is the person who will use your program, but there may well be other stakeholders too.
Usability	A usability feature is some aspect of a program's user interface that maximises ease of use. A shortcut key and an icon with a tooltip are usability features. A facility that magnifies text is also a usability feature, because it makes a program easier to use for people who struggle with small text. The following list of usability features is by no means exhaustive: • Help features, whether that's a discrete 'help' section, or perhaps a characteristic whereby right-clicking on a button tells you what that button's purpose is • Error messages that offer guidance on how to interact differently with the system • Consistent positioning of controls, such as always placing the OK button in the bottom-right of a window • Ensuring resemblance to other software, maximising a user's ability to make immediate use of the software • Minimising the number of clicks needed to perform an action • Structuring screen layout so that controls are positioned in the order in which they are needed, i.e. left to right, top to bottom • Preventing user error by disabling error-inherent features, e.g. causing any non-numeric entry into a particular text box to be ignored • Speeding up data entry by including easily removable default values in text fields
Validation	This is a process that ensures that entered data is reasonable and sensible. Any invalid data is prevented from entering the system. This would include entering a date of birth as being some day in the future, or a 50-character postcode.

