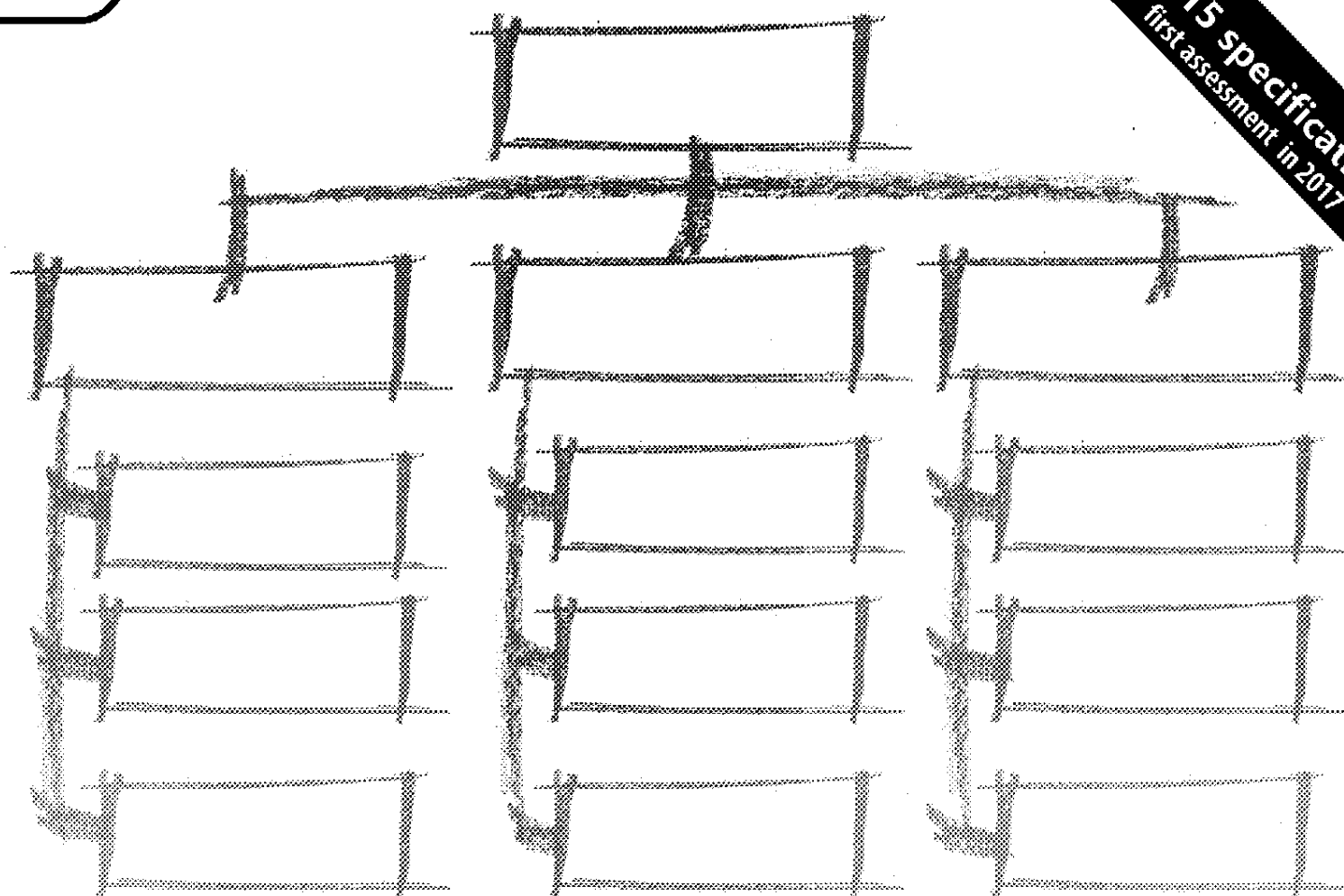




Computer Science

A Level | OCR | H446

2015 specification
first assessment in 2017



NEA Companion

for A Level OCR Computer Science

zigzageducation.co.uk

POD
10096

Publish your own work... Write to a brief...
Register at publishmenow.co.uk

Contents

Product Support from ZigZag Education.....	ii
Terms and Conditions of Use.....	iii
Teacher’s Introduction	1
Choosing a Project	2
Key considerations.....	2
Complexity.....	3
Project ideas	3
1: Analysis (10 marks).....	5
1.1: Computational methods.....	5
1.2: Stakeholders.....	6
1.3: Research of existing solutions	9
1.4: Essential features	10
1.5: Limitations	10
1.6: Hardware and software requirements	11
1.7: Success criteria	12
Analysis » Checklist.....	13
2: Design (15 marks)	14
2.1: Problem decomposition	14
2.2: Structure of the solution	15
2.3: Algorithm design	17
2.4: Usability features.....	21
2.5: Variables and validation	21
2.6: Iterative test data	23
2.7: Post-development test data	24
Design » Checklist.....	25
3: Iterative Development (15 marks).....	26
3.1: Iterative development stages.....	27
3.2: Modularity.....	29
3.3: Annotation.....	30
3.4: Naming conventions.....	31
3.5: Validation.....	32
3.6: Review	33
Iterative Development » Checklist	34
4: Iterative Testing (10 marks)	35
4.1: Testing	36
4.2: Remedial actions	36
Sample iteration	37
Iterative Testing » Checklist.....	40
5: Post-development Testing (5 marks).....	41
5.1: Testing for function	42
5.2: Testing for usability	42
Post-development Testing » Checklist	43
6: Evaluation (15 marks)	44
6.1: Examining success (or otherwise).....	44
6.2: Assessing usability	45
6.3: Maintenance and limitations.....	45
6.4: Quality of written communication	46
Evaluation » Checklist.....	47
Suggested Project Structure	48
Glossary	49

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.

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 things you need to bear in mind. Some of these are built into the qualification and the mark scheme and if you don't follow them you would cost you marks. Others are simply good advice. Make sure you can do all the things 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 solutions in the specified languages need to have a suitable GUI. If you have a text-based interface, it will not be awarded. For example, with a completely automated solution, it will not be awarded.
Are you going to be using one of the following languages? <ul style="list-style-type: none">• 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 can still do your project first with OCR. For the full list of languages, see appendix 5e of the OCR specification. If your solution is set to include one of the languages, each language should be either used or not used.
Will your solution have scope for validation?	The word 'validation' appears in the mark scheme, and the related word 'validation' is used. This means that a project that can be validated will lose marks in more than one place. Validation can exist in one of the following ways: <ul style="list-style-type: none">• Text boxes that incorporate validation of format, type or presence of data• A game board that blocks a rook diagonally in chess• A sensor or scanner that can detect a value outside an acceptable range
Do similar solutions exist?	There should be a computer-based solution in existence that performs a similar function. You need to fulfil exactly the same function, but for full marks in the project, 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 a solution that simply stores, manages or processes data. You've made an IT project. The project needs to be non-trivial. If your project could be done in Excel, without adding any code, it is not a computer science project.
Is it interesting to you?	You're going to spend dozens of hours working on a project that genuinely interests you. It's a struggle.


INSPECTION COPY

COPYRIGHT
PROTECTED



Complexity

When trying to settle on a project, both teachers and students struggle with the question of what is complex enough. It's not always a straightforward question to answer, because no part of the OCR A Level Computer Science mark scheme requires a complex project or removes them for a simple one. That's a point worth highlighting among us:



There are no 'complexity' marks on the OCR A Level Computer Science mark scheme. A project that is described fully by all top-band descriptors on the mark scheme can be considered complex enough.

Usually, a project that is too trivial will be unable to attain all of the marks simply because it doesn't meet a descriptor.

For example:

- In the design section, marks are awarded for defining 'in detail, the structure of the solution 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 algorithms that fully address each of those pieces; a trivial solution will either lack the complexity of a problem or fail to address it.
- Iterative testing requires the implementation of prototypes, which will not be possible if the solution is too trivial.
- Post-development testing requires candidates to address robustness, functionality and user requirements; 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 a computer science project instead of an IT project), it's probably complex enough. Always be guided by the mark scheme descriptor, and, if in doubt, a centre – but not a student – could consider it complex enough.

Project ideas

Ideas	Benefits	Challenges
Games	<ul style="list-style-type: none">• 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.	<ul style="list-style-type: none">• If you've never developed a game, you might find it complex.• Consider the extra time and effort required to learn new skills.• You should have a plan for validation if the project is too complex.
Desktop or web-based data-handling applications	<ul style="list-style-type: none">• 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.	<ul style="list-style-type: none">• It can be difficult to find a substantial solution that is not too trivial.

INSPECTION COPY

COPYRIGHT
PROTECTED





Ideas	Benefits	Challenges
Interactive learning resources	<ul style="list-style-type: none">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.	<ul style="list-style-type: none">To develop a good idea is likely to take time and effort.There is a risk of a simple idea that doesn't attract a large audience.If your idea is too simple, it amounts to a question of how to present it.
Mobile apps	<ul style="list-style-type: none">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.	<ul style="list-style-type: none">Not all projects will be successful, and your time and effort may be wasted.The same app as a service will still be the 'checklist' item, not the 'wow' factor.There is a lot of competition, and enough to make it difficult to stand out.
Control / monitoring systems	<ul style="list-style-type: none">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.	<ul style="list-style-type: none">The more complex the system, the more help from others will be needed.You still need to have a good understanding of the hardware and software involved.
Simulations	<ul style="list-style-type: none">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.	<ul style="list-style-type: none">A great deal of time and effort will be needed to create a realistic simulation.The complexity of the simulation may be too high for a single person to handle.The simulation may be too complex to be ready for use if time is short.

These are only broad categories; there are limitless potential computer systems. Consider your own interests, talk to friends and family, and have a look on the internet for other people's projects. Your work should be your own, but you are encouraged to build on what might be found.



1: Analysis (10 marks)

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

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
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.

You need to explain why a computational approach is suitable for the problem you are given. It is important to do this first (in fact, it's probably easier if you save it until the end of the analysis and 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 encountered elsewhere in computer science. The table below contains details of some of these.

Abstraction	Will your solution make a complex reality straightforward?
Decomposition	Does the problem (and prospective solution) lend itself to being broken down 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 graphics or tables?
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 how each approach is needed. You're not likely to need, for instance, visualisation techniques throughout, but perhaps the centrepiece of a dashboard form is a line graph.

e.g.	<i>The dashboard specified by the stakeholder needs to provide an interface that summarises the results of the analysis. The results should be at the centre of this part of the solution, with Internet plagiarism and peer-to-peer plagiarism highlighted in different colours [DESCRIBE]. There will also be, as stipulated by the stakeholder, a table showing the percentage of potentially plagiarised content across all work submissions. Using this approach will allow the user to make a judgement on whether to ignore or investigate the necessary information will be visible in a single place [JUSTIFY].</i>
------	--



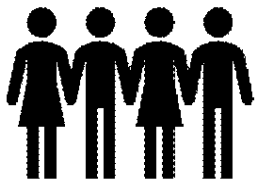
1.2: Stakeholders

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

MARK BAND 1	MARK BAND 2	MARK BAND 3
Identify the stakeholders as either individuals or groups, describe them and describe what they might want in your solution.	Mark band 1 plus: Describe <i>how</i> 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 <i>why</i> your solution will meet their needs.

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

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?

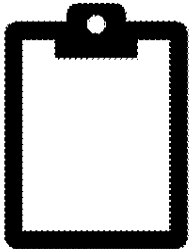



Ideally, stakeholders are real individuals of whom you can ask real questions, so you should talk to real individuals to whom you have some access.

No one will be conducting investigations as to whether your stakeholders spoke with them, but it's so much easier if you do. With a real stakeholder solution they want, make it, and then ask their opinion of it at the end. When you have to fabricate an interview, think long and hard about what features you have to fabricate 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 they need, as well as what they might want from a solution. There are several ways to this, and you can take more than one in combination. Other approaches may also exist, depending on the situation.

	<p>Interviews are useful for gathering large amounts of data from stakeholders (including a single stakeholder). Questions can be asked, and questions can be asked. You might, for instance, ask what the current solution is, and be given a list of 20 items. In an interview, you can ask which are the most significant three, or ask for specifics about the current solution to work – it might differ from your own vision.</p> <p>A transcript (a record of everything that was said by both interviewers and the stakeholder) will need to be included for each interview conducted.</p>
--	---

	<p>Questionnaires lend themselves to collecting a small amount of information from a large number of stakeholders. If you wanted to create a college-wide survey, asking staff, you might want to collect information from a dozen or so. Conducting many interviews would be quite time-consuming.</p> <p>Long-answer questions tend not to lend themselves to questionnaires. They are more easily asked in an interview, and you're less likely to get a clear answer. Multiple-choice questions, ranking questions, and short answers are appropriate for questionnaires, and can make analysis straightforward and meaningful.</p>
	<p>Observation involves watching the stakeholder using the current system. This can give insights beyond what you might find in a questionnaire or interview. How long do they spend the most time? How long does it take to perform a task (or not do it)? Are there any features that they struggle to find?</p> <p>Since you're looking to create a solution that improves on what exists, you might simply spot a four-click activity that you could simplify.</p>

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


- Each question should produce an answer that genuinely helps in the development of a solution. Asking 'do you like the current system?' yields a 'yes' or 'no', neither of which could change one thing about the current system, what would it be?' gives you more information, even leaves the respondent the option of saying 'nothing'.
- Recognise that your technical knowledge might be greater than that of your stakeholders. 'Will this ever need to store fractions?' is less likely to be understood than 'will this ever need to store fractions of an integer?'
- Don't make it too easy on your stakeholders by always providing 'I don't know' as an option. Alternative approaches, such as placing features in order of importance, or using 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 and questionnaires are not enough. Describe how you plan to proceed with your analysis, how you're engaging with the stakeholders, and justify any choices you make. This could be done diagrammatically, perhaps by way of flow charts or data flow diagrams (DFDs).

Good and bad interview questions

Problematic	Better	
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 the person answering the question thinks the interface is bad. The better question allows for a range of responses.
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 assumes the interface might be bad and the performance might be abysmal. The better question allows for a range of responses and assumes that the respondent has used the system.
Do you always keep a printed copy?	How often do you keep a printed copy?	Yes/no questions don't allow for a range of responses and understanding. If the respondent says 'no', it's all but one of the things they don't use the system, the better question allows for a range of responses.




If you're conducting an interview, don't worry about going off-script. There's no harm in asking a question you weren't planning on asking. Often, an answer you receive might lead to a new question. The respondent might say 'I hate the menu structure', at which point you'd probably want to ask 'What do you like about the menu structure?'

COPYRIGHT
PROTECTED



Good and bad questionnaire questions

Problematic	Better	
How often do you use the system?	How often do you use the system? <input type="checkbox"/> More than once a day <input type="checkbox"/> 3–7 times a week <input type="checkbox"/> 1–2 times a week <input type="checkbox"/> Less than once a week	Pre-opt The mo the qui you
When did you last update the software? <input type="checkbox"/> Today <input type="checkbox"/> This week <input type="checkbox"/> Last week <input type="checkbox"/> Last month	When did you last update the software? <input type="checkbox"/> Within the last 24 hours <input type="checkbox"/> Between 1 and 7 days ago <input type="checkbox"/> Between 8 and 14 days ago <input type="checkbox"/> More than 2 weeks ago	The in 'To the and opt last
How should the data be stored? <input type="checkbox"/> Cloud <input type="checkbox"/> USB flash drive <input type="checkbox"/> Other	How should the data be stored? <input type="checkbox"/> Cloud <input type="checkbox"/> USB flash drive <input type="checkbox"/> Other (please state) _____	'Ot you You tid but any



You don't have to restrict yourself to lists of options, but you should keep written responses being no more than a few words. An exception to this is 'else...' open-ended question at the end, where people can write what they

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

e.g.	<p>I believe it would be best to incorporate access to the print function in three 'file' menu, on the toolbar and as a reaction to the CTRL+P key combination is that the questionnaires showed that people used all three of these options (20% shortcut key) [EXPLAIN]. While I could have incorporated only a toolbar currently – this would make the solution less intuitive for half of the prospect</p>
------	--

INSPECTION COPY

COPYRIGHT
PROTECTED





1.3: Research of existing solutions

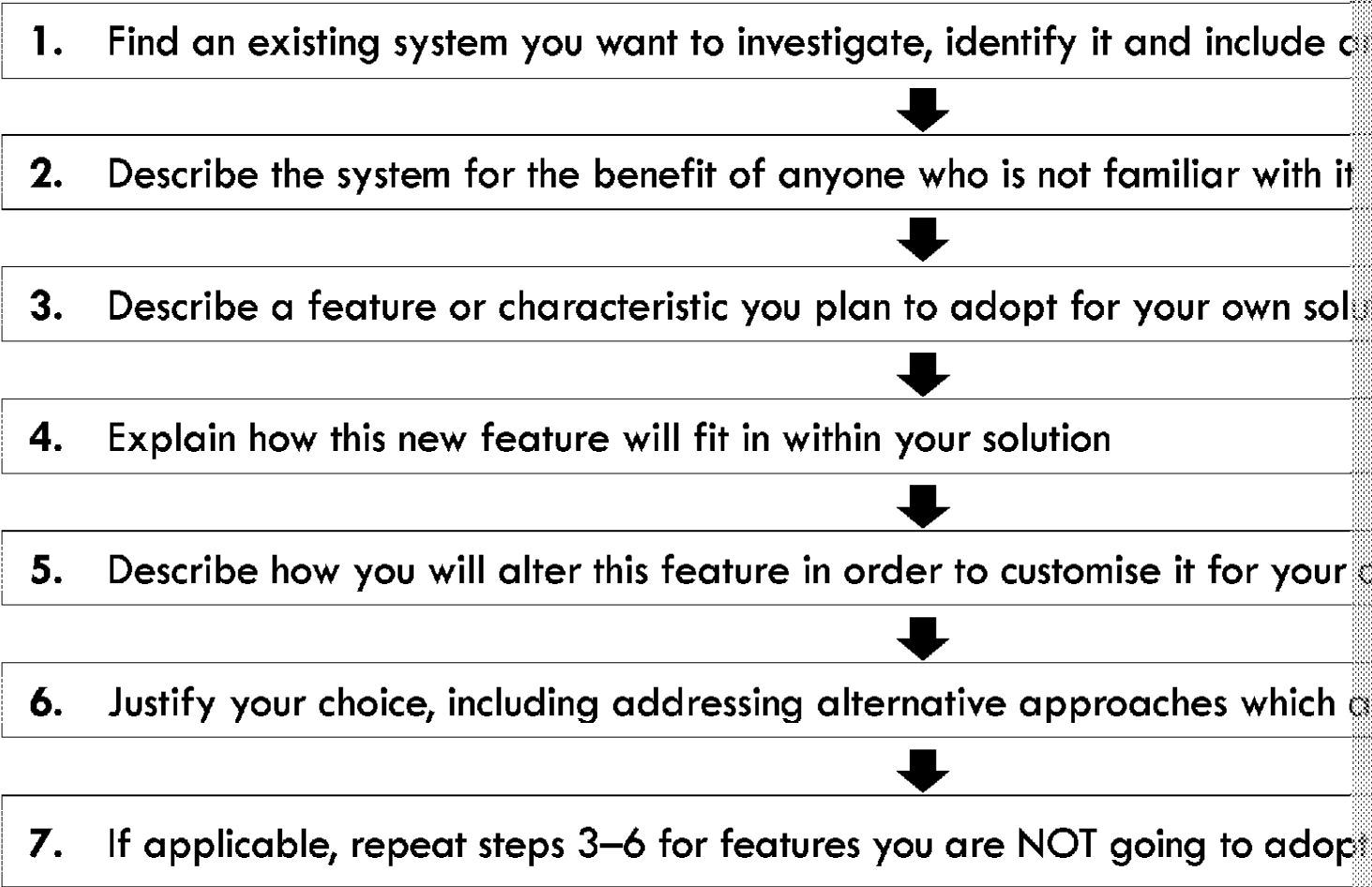
MARK BAND 1	MARK BAND 2	MARK BAND 3
List some features that you plan to include in your solution.	Mark band 1 plus: Conduct research of similar systems and provide evidence of that research. Clearly show how the features you plan to adopt have arisen from the research.	Mark band 2 plus: Describe (not list) the findings of your research and describe how you might approach your solution in light of this research.

Your aim is to create a solution to an existing problem, but you probably won't be the first to solve this problem. In this section, you'll have a look at previous approaches to the problem you're addressing. 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 different)

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



e.g.	<i>The system currently used by most kiln operators uploads temperature readings at regular intervals [DESCRIBE]. I will need to incorporate this into my own system, but not one near the kiln when it is in operation, and intervals need to be set as frequently as possible [JUSTIFY]. Currently, when the user wants to see the temperature history, they have to log in to the web interface [IDENTIFY]. I will develop my own system differently, so that users can see an up-to-date history line graph, without human involvement [DESCRIBE]. Although this may use more bandwidth, it should not present a problem, as it will be accessed over a wireless network [JUSTIFY].</i>
------	---



1.4: Essential features

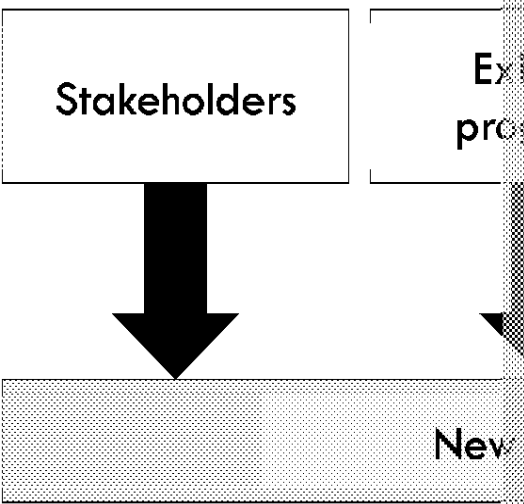
MARK BAND 1	MARK BAND 2	MARK BAND 3
Provide a list of some of the things your solution <i>must</i> do in order for it to address your chosen problem.	Mark band 1 plus: Ensure you have identified <i>all</i> essential features, rather than some.	Mark band 2 plus: Describe each feature treating each listed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

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.	<i>My system will require a 'create new account' feature. [STATE] This should be a 'new account' button and entering a username and a password. The system will not allow a user to proceed if their username is already chosen, otherwise it will let them proceed. [DESCRIBE] 100 new students will join the department each year, and each new student will need a unique username. Letting them choose their own username causes less work for staff [EXPLAIN]</i>
------	--

1.5: Limitations

MARK BAND 1	MARK BAND 2	MARK BAND 3
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.

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 should? For example, a game that does not save player progress? Is it a revision tool that does not allow for multiple users?
2. What constraints will prevent your program from being perfect? Think about limitations in your skill set, unavailability of key stakeholders, lack of network access, etc.

e.g.	<i>My system will not be able to work with real-time exchange rates [IDENTIFY]. During the transaction being processed, the user would be able to switch between currencies [DESCRIBE]. This will not be an option, as the most straightforward way to get real-time data is to pay for the data from a provider such as XE, and this is not a commercial proposition. I could spend time writing script that would extract the data from a relevant website [EXPLAIN], but this is not the solution, as my stakeholder has indicated that this would not be an essential feature.</i>
------	--



1.6: Hardware and software requirements

MARK BAND 1	MARK BAND 2	MARK BAND 3
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 just listed), and you should consider specifics, such as versions of software and screen resolutions.

In order to get as far as mark band 2, you simply need a comprehensive list. Let's say you're developing a data processing application, developed using Visual Basic, which interfaces with a database. The following would be enough:

Hardware: <ul style="list-style-type: none">• 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	Software: <ul style="list-style-type: none">• Visual Studio 2010• Microsoft Access 2010• Windows 7 (or later)• .NET framework 4.0
--	---

That's it for mark band 2. If there were omissions from this list, that would nudge you towards 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, disk space, etc. should come from the most demanding software you're going to run (Access 2017 in this case). The display should come from the on-screen size of the application you're planning to run. If you're unsure about any of these, you can always come back and fill them in once you've started.

For mark bands 3 and 4, you need to describe, explain and justify each item on the list above, so that means 10 small paragraphs, each following a similar format to the two-button mouse:

e.g.	<i>A two-button USB mouse will be needed [DESCRIBE]. USB ports will be available on the laptop which is a standard-issue college laptop [EXPLAIN]. Two buttons are important for the event of on-screen objects, while the right button accesses a help feature [JUSTIFY]. While the touch pad on a laptop will be adequate for this role, people prefer mice because they are more quickly when using them [JUSTIFY].</i>
------	--

The first point is 'describe' rather than 'identify', because it's a 'two-button 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
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 <i>how</i> each item will be measured.	Mark band 2 plus: Your checklist should cover all aspects of the proposed solution.

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

Probably the best way to understand a good success criterion is to have a look at

Attempt	
I should have a user-friendly interface	Admirable how to me
I should have a user-friendly interface, which will be measured by giving stakeholders a questionnaire after they have used it	Better, but more abo
There will be a question 'rank the usability of this application on a scale of 1 to 10'	Here we h how this v need a pa
In order for the solution to be a success in terms of usability, the score should be either 9 or 10	This is no needed ne
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. T justified. means the later on, v or not you

Most other success criteria will be easier to define, as there will be less of a 'hum You might aim to have a feature for a new user to create a new account. This co testing that the 'new account' part of the solution works.



The success criteria section is important, as it forms the basis of the evaluation. You will need to provide an assessment, for each criterion, of whether you or meet it.

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

INSPECTION COPY

COPYRIGHT
PROTECTED



Analysis » Checklist

MARK BAND 4:
<ul style="list-style-type: none"><input type="checkbox"/> Features that make this problem appropriate to approach using computational methods (including abstraction and decomposition) are identified, described, explained and justified<input type="checkbox"/> Analysis of stakeholder requirements is presented, typically as a result of interviews and questionnaires, and the way in which a solution is about to be developed is described and explained<input type="checkbox"/> There is in-depth research covering multiple similar or related solutions<input type="checkbox"/> Research into similar or related solutions has provided insights on how to approach the problem, which are explained and justified<input type="checkbox"/> Essential features are identified and described, and there is clear explanation of why they are essential<input type="checkbox"/> All limitations on the solution are clearly described, explained and justified<input type="checkbox"/> Every piece of hardware and software required is described in full, and its role in the solution and its purposes of the solution is justified<input type="checkbox"/> Success criteria are measurable, with the means of measurement clear, and the success criteria cover the proposed solution in its entirety
MARK BAND 3:
<ul style="list-style-type: none"><input type="checkbox"/> Features that make this problem appropriate to approach using computational methods (including abstraction and decomposition) are identified, described and explained, but not justified<input type="checkbox"/> Analysis of stakeholder requirements is presented, typically as a result of interviews and questionnaires, and the way in which a solution is about to be developed is described and explained<input type="checkbox"/> There is in-depth research covering multiple similar or related solutions<input type="checkbox"/> Research informs descriptions of how (but not why) the problem will be approached<input type="checkbox"/> Essential features are identified and described, but not explained<input type="checkbox"/> All limitations on the solution are clearly described and explained<input type="checkbox"/> Hardware and software requirements are specified in full<input type="checkbox"/> Measurable success criteria are stated and cover the proposed solution in its entirety
MARK BAND 2:
<ul style="list-style-type: none"><input type="checkbox"/> Features that make the problem appropriate for computational methods are identified, but not explained<input type="checkbox"/> Description of how stakeholders will use the system is included, supported by evidence<input type="checkbox"/> Features from researched similar solutions that might transfer to your own solution are identified<input type="checkbox"/> Essential features, limitations and most hardware/software requirements are identified<input type="checkbox"/> Success criteria are identified, which must still be measurable
MARK BAND 1:
<ul style="list-style-type: none"><input type="checkbox"/> The computational methods section is characterised by identifying applications and describing them<input type="checkbox"/> Stakeholders, and some of their needs, are identified, but not necessarily described<input type="checkbox"/> Appropriate features are identified for incorporation into your solution, but not justified on any kind of research<input type="checkbox"/> Essential features, limitations and some hardware/software requirements are identified<input type="checkbox"/> Success criteria are identified, although they might not be measurable

INSPECTION COPY


COPYRIGHT
PROTECTED



2: Design (15 marks)

In the design phase, you plan the development of your solution. This covers data algorithms and planning out exactly how you will test your solution, both during If you have conducted your analysis properly, with meaningful examinations of ex with real stakeholders, this section should be quite straightforward. The aim is to 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



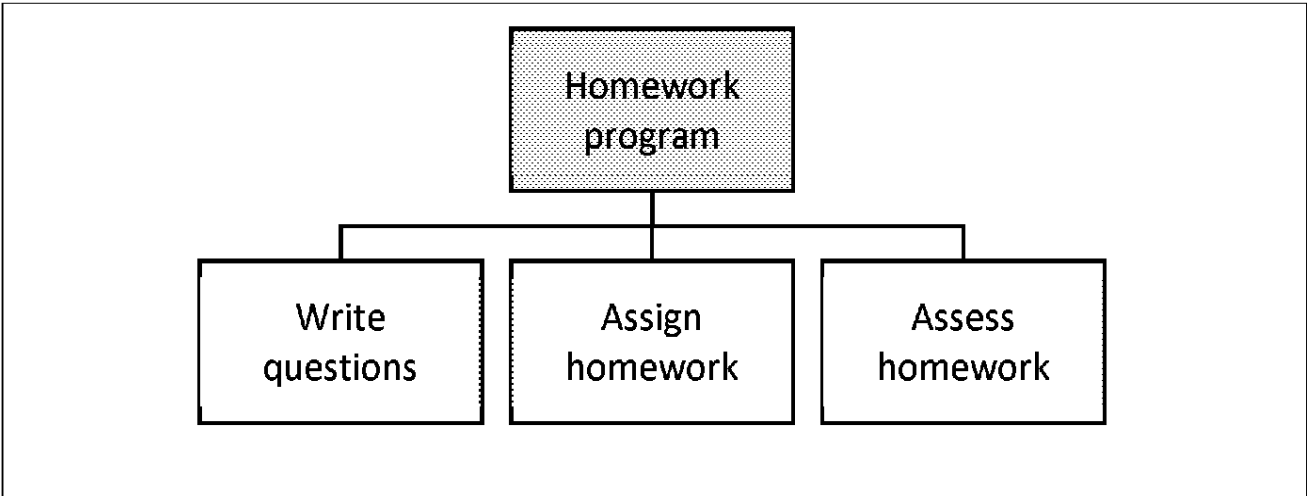
Obviously, your solution will not resemble your design precisely. You will best to proceed, and you're likely to add, remove and change quite a few th revisit your design to make retrospective changes when this happens, but changes that you make once you're developing. It can lead to a greatly en

2.1: Problem decomposition

Note than you are decomposing the **problem** here, and not the **solution**. This par analysis gives way to the design, so you're showing that you understand the curre

MARK BAND 1	MARK BAND 2	MARK BAND 3
<i>Problem decomposition is not required for mark band 1.</i>	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.

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



COPYRIGHT
PROTECTED



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.

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.

2.2: Structure of the solution

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

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

MARK BAND 1	MARK BAND 2	MARK BAND 3
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.

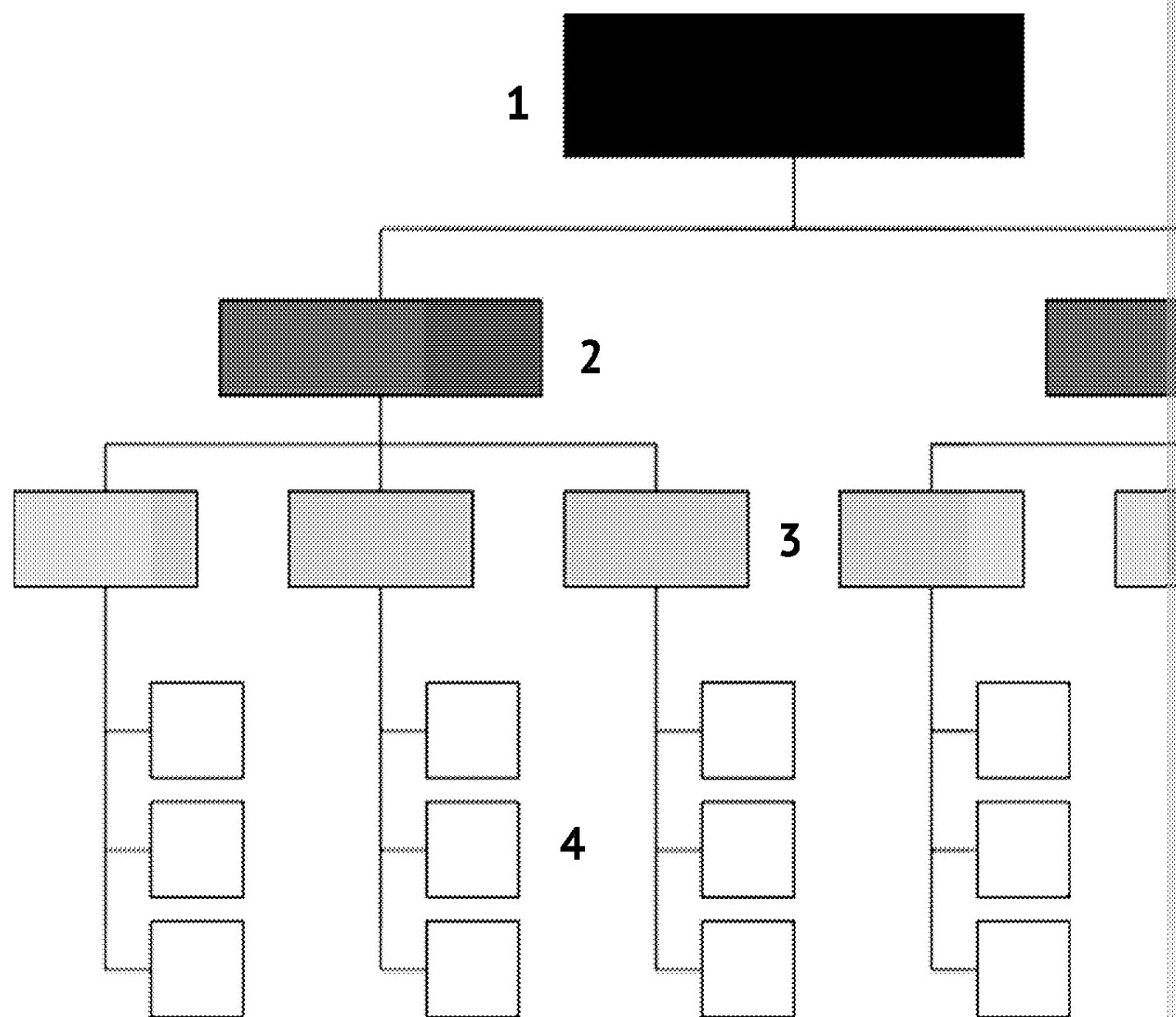


It might seem tempting to create identical hierarchy diagrams for the problem and the proposed solution. They will probably have many commonalities, but if they are exactly the same, the solution doesn't contribute anything new.

No justification is required by the mark scheme for this part; you simply need to show the structure of your proposed solution. Assuming you can fit the hierarchy diagram for your solution into the space like this (of course, each of your shapes will contain text, depending on what the problem is).

COPYRIGHT
PROTECTED

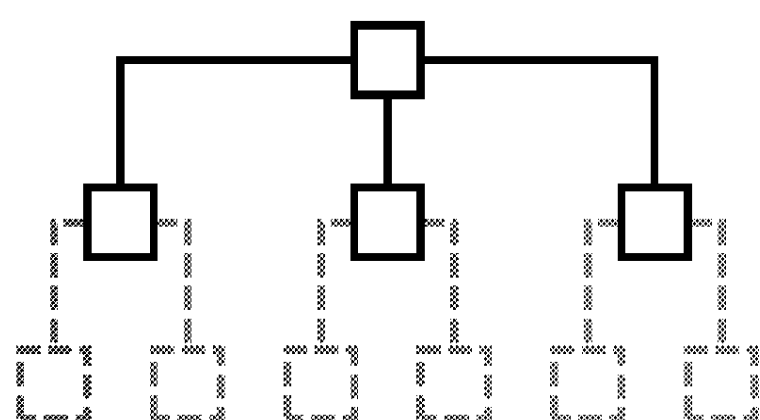




1. This is the name of your entire system. In this instance, it might be 'homework management system'.
2. At the first level of decomposition, the solution is divided up into major subsystems. For example, there might be a student subsystem (to complete homework) and a staff subsystem (to set homework).
3. This tier might represent forms or individual web pages (if applicable). For example, the student subsystem 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, the 'write questions' form might have a subroutine called 'save_mark' and another called 'download_questions'.



With no justification needed in this part of the write-up, top marks are given to students who define their solution in detail. This means breaking down each part into its constituent parts, as well as creating a design that covers everything mentioned in 'essential features'. If you need to, split your diagram across multiple pages rather than reducing the size of the boxes.



Decomposition has taken place, and the solution is detailed enough. Each terminal node (subroutine) should be straightforward to implement.

Either:

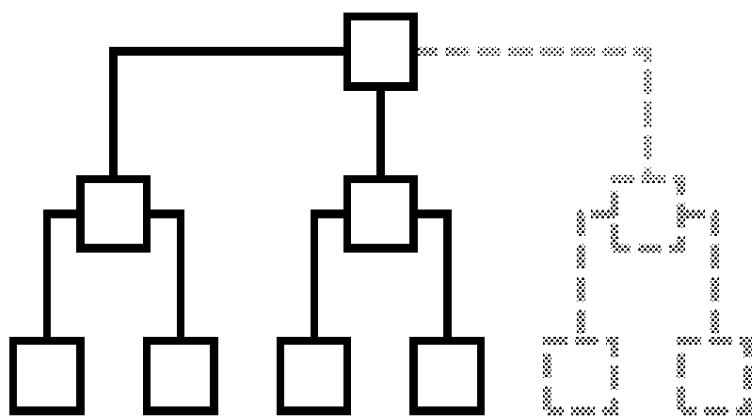
- given your structure diagram, you can write the code that would program that terminal node, and you can explain how you ultimately implement it.

or:

- any differences would be between the two, but both would be equally good with any other solution.

COPYRIGHT
PROTECTED

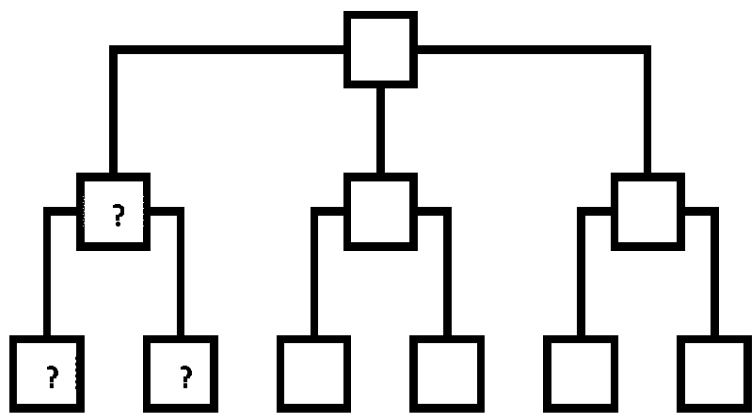




Some tasks are decomposed in a way that is not obvious. Check your structure diagram.

Does your structure diagram cover:

- Everything from your 'essential' tasks
- Everything from your 'success' tasks



You have included tasks that cover the whole of the analysis. There is no upper limit on the number of tasks you can develop, and it's not unusual to have tasks that span from design to development. It's all about the nature of the solution.

The nature of the solution should be reflected in the analysis and design. If it does, that project is a success. If it doesn't, the features section, or your success criteria, are incomplete.

2.3: Algorithm design

MARK BAND 1	MARK BAND 2	MARK BAND 3
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 <i>all</i> algorithms are designed.	Mark band 2 plus: Include commentary on each algorithm, explaining how it fits in with the overall solution.

There is no specification in the mark scheme as to the format in which algorithms are normally presented in the form of flow charts, pseudocode, actual program code. Of these options, the favoured two are pseudocode and flow charts. Structured programming does not follow a standard format, and program code, in this coursework, is assessed as well as it here as well.

COPYRIGHT
PROTECTED



Flow chart shape	Use
	Terminator Only two of these appear in each flow chart, one saying 'start' and one saying 'end'. If your algorithm has multiple paths, they can all come together at the 'end' terminator.
	Process Unless an action involves input, or output, it belongs in this shape. Process shapes can include calculations as well as declaring/initialising variables.
	Decision Used in place of an IF statement. The question should be a yes/no question. The flow continues from the 'yes' arrow, with the 'no' arrow leading to an alternative path.
	Input/output Items input from the user or output to the user using one of these shapes, which should be clearly labelled accordingly, as well as the name of the variable being used.
	Document This is most likely to be used if you are producing printed output, but it can also be used if a document is being scanned or translated into a different format in place of the input/output shape.
	Stored data This will represent files that are being used or created. They might be text files, binary files, or databases, but the solution will work with any type of data storage.
	Connector If you need to split a large flow chart into smaller sections, you'll use connectors. At the bottom of each section will end with a connector containing a letter. The next section would continue from whichever connector contains the same letter 'A' connector.
	Predefined process This is the shape you'll use for operations that are predefined by another flow chart. The name of the operation called should go into the shape, and the name of the corresponding title should be defined in the header of the predefined process flow chart.

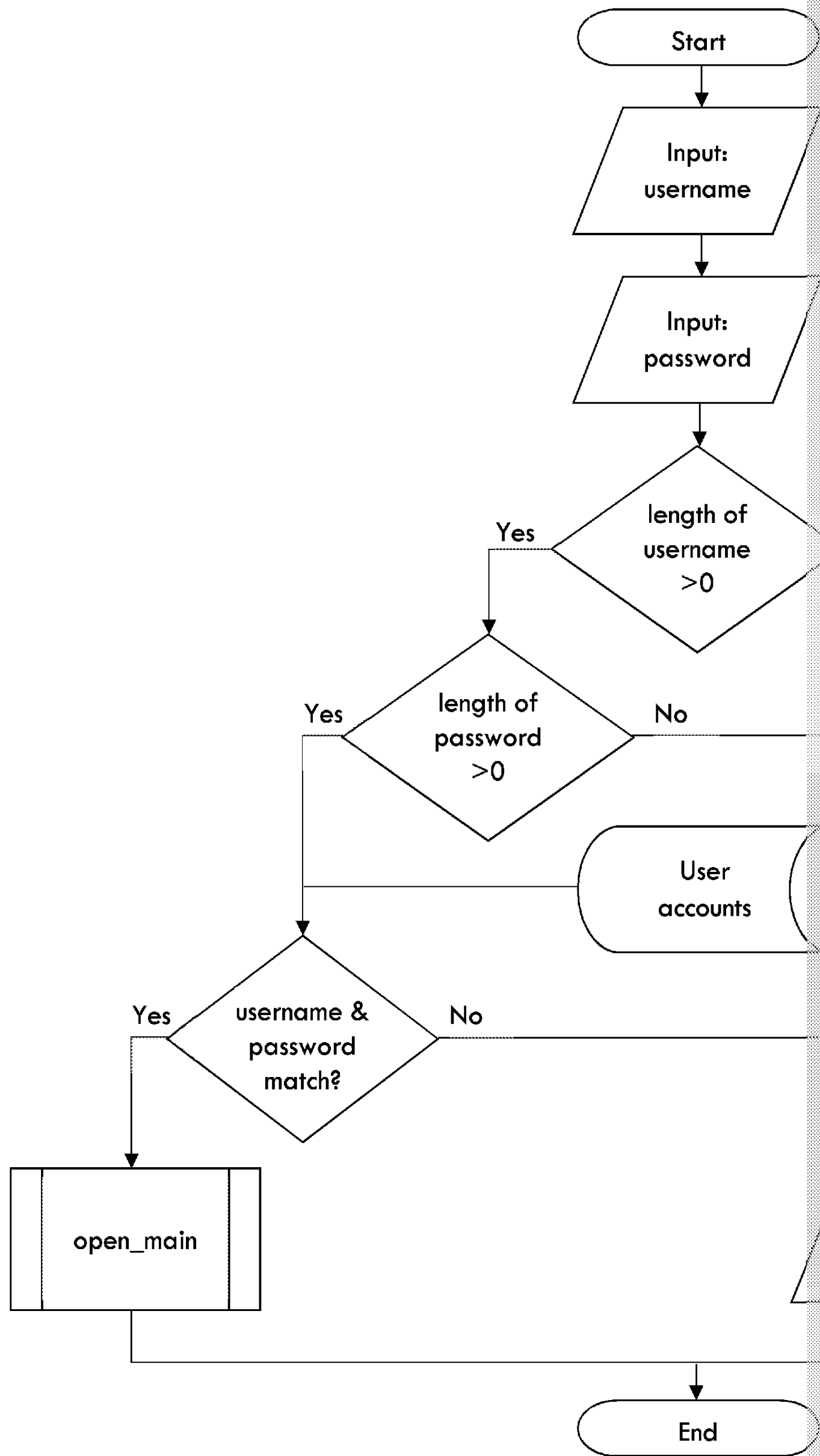


There are several tools that can help you to draw flow charts, but I would recommend a tool called 'draw.io'. Anything you create can be automatically saved in the cloud, making it quicker to work with than general purpose software such as PowerPoint.



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 to make 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 2.3
- Variables called 'username' and 'password' should also be designed in section 2.3

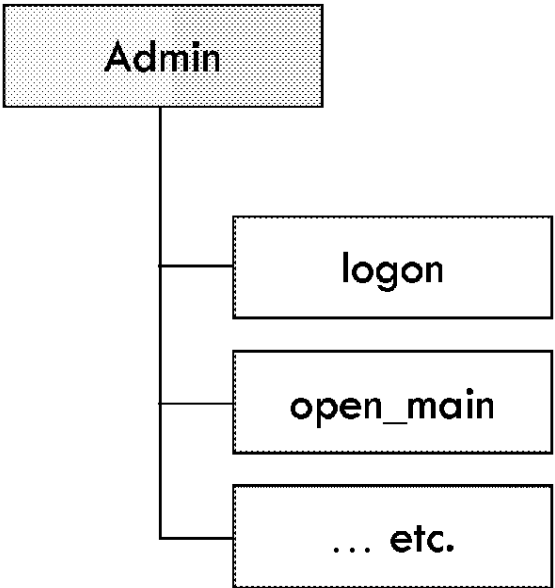


An alternative to the flow chart would be pseudocode (for OCR's pseudocode syntax see their website):

```
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  
        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 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 approach diagram) should have an algorithm to correspond to it. The diagram in this guide would be either 18 flow charts or 18 sets of pseudocode (or a combination of the algorithms should correspond with the names written or typed onto the terminal



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

e.g.	<i>I decided to create a separate, self-contained logon subroutine [DESCRIBE], to improve maintainability [EXPLAIN]. If a new logon system were to be developed, so the logon subroutine would need to be rewritten [DESCRIBE]. Had I incorporated this logic into the main program instead, subsequent maintenance would have been more difficult, as any code changes would first need to be located within a larger body of code [JUSTIFY].</i>
------	--

Providing a justification for each individual algorithm should see you safely into the next section.

2.4: Usability features

MARK BAND 1	MARK BAND 2	MARK BAND 3
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?

You are not required to design screen layouts, but you might find it a useful prelude to identify the usability features you intend to use. They can also reduce your word count 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. For example, a search button always goes in the bottom right, a help button is available on all screens.
- Use of icons
- Accessibility features, such as settings to change font size, colour combinations, etc.
- 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.

e.g.	<i>On my main menu bar, I intend to use tooltips on each of the icons [IDENTIFY]. I will use the words 'new', 'open', 'save', 'print', 'close' or 'help' icons, the word that denotes the action that will appear in a temporary box that disappears when the cursor moves on [DESCRIBE]. I will use a consistent layout so that users can be sure of the purpose of the button they're about to click [EXPLAIN]. I did consider using text buttons, but they take up more space than is available [JUSTIFY]. I am confident that users are familiar with the standard icons for each of these processes [JUSTIFY].</i>
------	---

2.5: Variables and validation

MARK BAND 1	MARK BAND 2	MARK BAND 3
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?'

It's not just variables that need to be designed in this section, but anything that stores data. This includes variables, data structures (such as arrays), classes and external files. In a formulaic way in which you will describe each variable in turn, a table is probably





Variables

Name	Data type	Scope	
password	String	Local to the 'logon' subroutine of the User class	N/A – variable
userID	String	Attribute of User class	Private, no
userCount	Integer	Attribute of the Log module	Public, s

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

Data structures

Name	Data type	Scope
arrScores	Integer array	An attribute of the 'game' class
dtbUsers	Access database	N/A: this is outside the bounds of the program

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

Class diagrams

Character
- energy: integer - name: String - isEnemy: Boolean
+ move(): void + guard(): void + getEnergy(): integer

Again, there is plenty here to justify. Why is 'energy' an integer? Why is it private? the 'move' method have a void data type? Why do we even need a 'character' class to be done?

Try not to repeat yourself too much when working on this section. If several ways, consider grouping them together alongside a single justification.

Data validation



It is critically important that your solution involves some form of data validation. Data validation should not be lost in each of the design, implementation and testing stages.

You're not expected to provide validation routines for every individual variable. Unnecessarily, you'd be quite likely to lose marks, as the top-band mark scheme says '...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 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 characters
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 data item

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

e.g.	<p>For the user's email address, I will apply a format check [IDENTIFY]. There is a <code>@</code> symbol within the entry, but this should not be at the start or the end. There should be a <code>.</code> symbol after the <code>@</code> symbol, which should also not be at the end [DESCRIBE]. This is a suitable format for email addresses follow this convention, with a domain following the <code>@</code> sign consisting of a <code>.</code> and a domain name [EXPLAIN]. I could have also implemented a length check, but the format check would catch addresses that are too short, and I am not aware of any upper limit to the length of an email address [JUSTIFY].</p>
------	--

2.6: Iterative test data

MARK BAND 1	MARK BAND 2	MARK BAND 3
Identify data that will later be used for testing the solution during development (i.e. not after development).	<p>Mark band 1 plus:</p> <p>Expand the range of test data so that the entire solution can be tested in a way that examines all interactivity, functionality and validation.</p>	<p>Mark band 2 plus:</p> <p>For each piece of test data, explain why it is needed; what exactly the purpose of each test is.</p>

You need to plan out your test data in advance, and there should be test data for each input. The test data should fit into categories of **typical**, **boundary** and **erroneous**. To illustrate each category, let's consider a system that allows estate agents to list houses for sale that have between 1

**COPYRIGHT
PROTECTED**



Typical	Boundary	
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 No ac Es Th be " En te 3. Wi ra sh

As you describe each piece of test data for each subroutine, you should explain it useful to organise your test data into a table (as below). This can help to ens


Subroutine	Field	Data	
sell_house	number_of_bedrooms	1	This
sell_house	number_of_bedrooms	11	Al out
sell_house	number_of_bedrooms	3.4	Whi the a d

2.7: Post-development test data

MARK BAND 1	MARK BAND 2	MARK BAND 3
<i>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 →</i>	Identify data to be used in testing after the solution has been completed.	Mark band 2 plus For each piece of data, explain why it is needed; what exactly the purpose of each

This is test data that will be applied after development has completed, unlike the test the solution as it is being developed. You can use exactly the same approach to test data.

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



Testing is not something that is considered only when coding is complete. You should conduct testing as you develop, conduct testing post-development and conduct testing. You are advised to look ahead to sections 3 to 6 in order to be

COPYRIGHT
PROTECTED



Design » Checklist

INSPECTION COPY

MARK BAND 4:
<div><input type="checkbox"/> The problem has been broken down in a systematic manner, such as using a clear explanation of why the problem was broken down in the way that it was</div> <div><input type="checkbox"/> The structure of the solution is defined down to the level of each algorithm and the algorithms relate to each other</div> <div><input type="checkbox"/> Every individual algorithm, as identified by the point above, is defined using a flow chart</div> <div><input type="checkbox"/> The presence of each algorithm is explained and justified, to make it clear why the solution in the way it has been incorporated</div> <div><input type="checkbox"/> All usability features are identified, described and explained, with choices justified</div> <div><input type="checkbox"/> All key variables and data structures are identified, described and explained</div> <div><input type="checkbox"/> All validation is identified, described and explained, with choices justified</div> <div><input type="checkbox"/> Test data that will be used during iterative development is outlined in detail</div> <div><input type="checkbox"/> Test data that will be used after iterative development, as a basis for evaluation, is identified and justified</div>
MARK BAND 3:
<div><input type="checkbox"/> The process of breaking down the problem is explained, and is still expected to have the justification</div> <div><input type="checkbox"/> The structure of the solution is specified down to the level of each algorithm</div> <div><input type="checkbox"/> All subroutines need to be defined using either pseudocode or a flow chart, and be explained, but there is not a justification as to how they fit in with the rest of the solution</div> <div><input type="checkbox"/> All usability features are described and explained, but not justified</div> <div><input type="checkbox"/> All key variables are described and explained, but not justified</div> <div><input type="checkbox"/> All validation is described and explained, but not justified</div> <div><input type="checkbox"/> Test data that will be used during iterative development is outlined in detail</div> <div><input type="checkbox"/> Test data that will be used after iterative development, as a basis for evaluation, is identified and justified</div>
MARK BAND 2:
<div><input type="checkbox"/> The problem is broken down into smaller problems, with commentary on the breakdown</div> <div><input type="checkbox"/> The structure of the solution is defined, although there may be gaps or errors down to algorithm level</div> <div><input type="checkbox"/> All algorithms are defined in pseudocode or flow chart form, but there is no explanation or describe these algorithms</div> <div><input type="checkbox"/> The variables are identified, as are necessary validation routines</div> <div><input type="checkbox"/> Usability features are described</div> <div><input type="checkbox"/> A range of iterative test data is defined</div> <div><input type="checkbox"/> A range of post-iterative test data is defined</div>
MARK BAND 1:
<div><input type="checkbox"/> There are some algorithms defined using at least an attempt at a standard notation</div> <div><input type="checkbox"/> Some usability features are described, but there could be gaps</div> <div><input type="checkbox"/> The variables are identified</div> <div><input type="checkbox"/> Some test data is presented for either the iterative phase or the post-iterative phase</div>

COPYRIGHT
PROTECTED



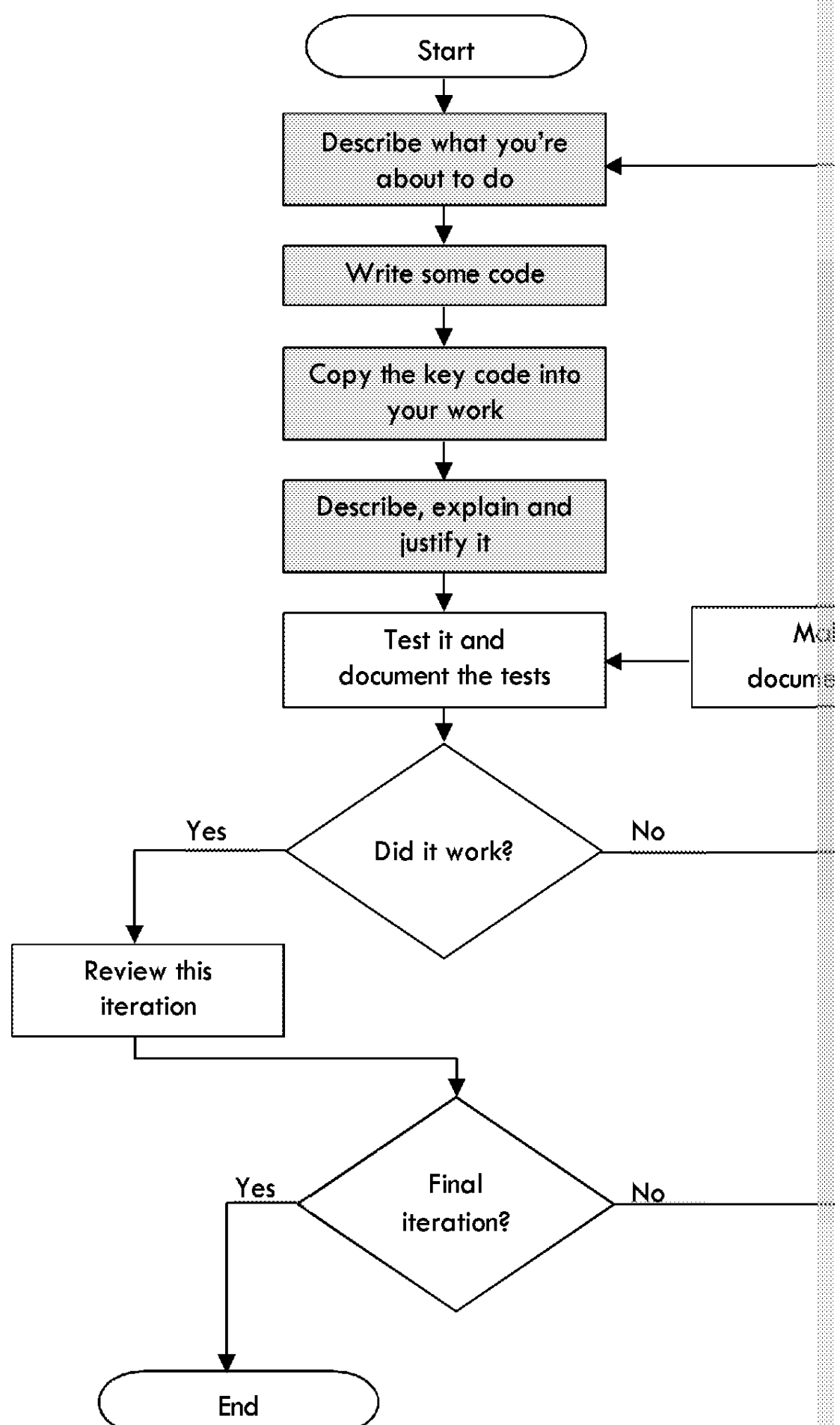
3: Iterative Development (15 marks)

One of the dangers, especially for talented programmers, is the temptation to keep working until the solution is complete. While that might result in a good-quality program, it could mean 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, then tests, reflecting on what you have done, then moving on to the next part or the next iteration. This needs to take place in tandem with section 4, iterative testing. For this reason, you should write section 3 and section 4 at the same time, rather than as separate sections within your program.

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:



INSPECTION COPY

COPYRIGHT
PROTECTED





3.1: Iterative development stages

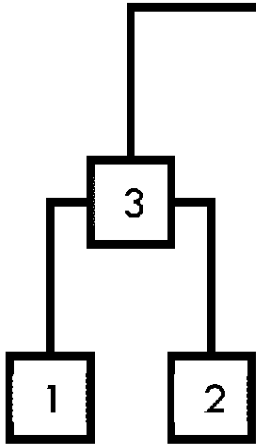
It cannot be overstated that you should not simply create a program, from start to finish. If you take a non-iterative approach, you would not even get into mark band 1. Instead, you develop your program piece by piece, documenting as you go.

MARK BAND 1	MARK BAND 2	MARK BAND 3
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 and intermediate stages.

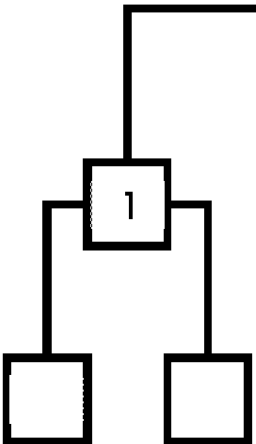
The flow chart on the previous page shows how each iterative development stage, in each design phase, you will have broken your solution (the problem too, but this is about the solution), into pieces, and each of those pieces is one iterative development stage.

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

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).

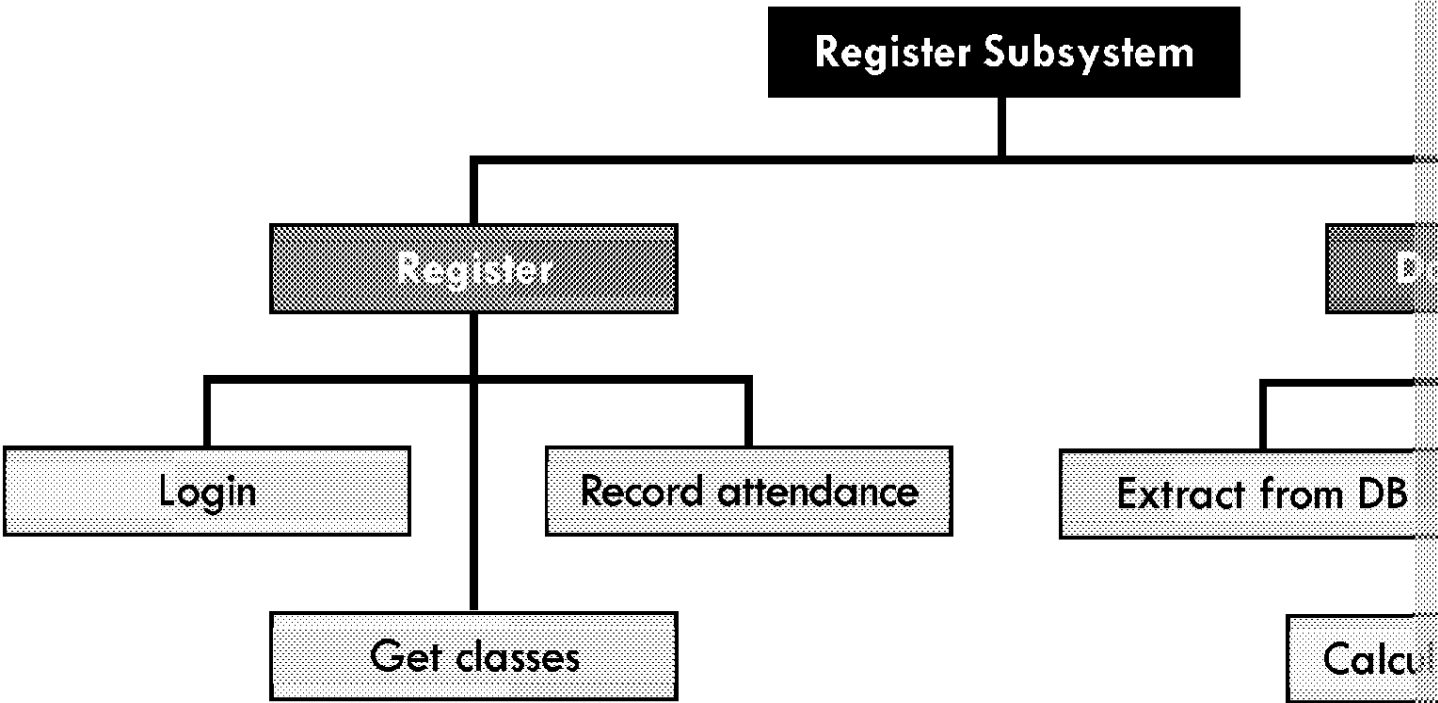


B. In this approach, cycle '1' covers that node as well as the associated child nodes.
This would result in four cycles (again, including a final one that brings the pieces together), but each cycle would include more commentary, more code and more testing.





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, before finally assembling the 'register' subsystem, which contains
- B. Alternatively, we could spend a longer time developing the 'register' subsystem first developing the subparts.

There is no single 'best' way to conduct or document your development, but there are some things to be observed:

- Describe everything you're doing and explain why you're doing it – there should be evidence of 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
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.

You are not required to have a separate 'modularity' section of your work; you are required 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 structural programming work (especially as you have a choice of languages), but here are some examples of good practice.

Modular and well structured	Not modular
Breaking your program into separate, self-contained pieces (called, generically, modules), which can be independently tested.	A single, typically quite large block of code that cannot be tested until the whole program is complete.
Code that is frequently needed is placed into subroutines and called whenever required.	Completely identical sections of code repeated 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 a large number of copy and paste operations.
Arrays, trees, other data structures and your own custom-written classes are used to store data.	Large numbers of variables, each with a unique name, which they each have a specific purpose. NB Variables are not 'the right tool for the job' - arrays, lists, etc. are. If you need a list of names, named, perform a very simple search. An array might have been better.
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 of the solution were to, for example, change a constant of 20% should be a constant of 20% at time of writing, 20% of the code would be countless instances of 20%.
Classes have private attributes and variables are local, passed as parameters whenever required.	Classes have public attributes and variables are used in place of parameters.
Classes demonstrate high cohesion – each class models the behaviour of a single real-world object, without including code that's not relevant to the class.	Classes may not be well defined. For example, a class might represent a room, rather than the room and the furniture belonging to separate classes.
Classes demonstrate loose coupling – data shared between classes is kept to the minimum of what is necessary.	Classes demonstrate tight coupling. Data is shared between classes and return values and public methods are used to pass data between classes.

Ultimately, modular programming works in your favour. Not only does it gain you time, you also 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 still demonstrate features that make your work modular. In the iterative development section, for each piece of produced code that shows characteristics of modularity, draw the reader's attention to the feature you have used, describe alternative approaches you might have taken and ask 'did you do it this way?'



3.3: Annotation

MARK BAND 1	MARK BAND 2	MARK BAND 3
<i>Annotation is not required for mark band 1.</i>	Code must include some comments, which will focus on the most important elements.	Mark band 2 plus: Comments must extend at least one per subroutine and one per variable.

Annotation does not require its own section in your work, and you do not even need to 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 as much of it depends on how the person marking your work interprets the mark. The phrase *be annotated to aid future maintenance of the system* isn't conclusive, so asking the person marking your work would be a useful move.

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


- Every subroutine, typically with a comment on the line above the declaration, what it does, what other subroutines it calls, where it is called from itself and details of its parameters.
- Every variable, with a comment either immediately above it or to the right of its declaration.
- Any sections of code that are key to the subroutine, or that are complex enough to warrant 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 0
5     Dim data() As Integer = {-1, 25, 0, -58964, 8547, -119, 256, 123456789}
6     'Call to 'QuickSortRecursive', passing the array, with left and right indices
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. The comments 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 see what each part of your code does, without reading any of the actual code.



Some languages offer facilities for marking multiple lines as comments and if such 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
<i>Naming conventions is not required for mark band 1.</i>	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 a standard convention to most variables and data structures.

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

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

Good Names	Bad Names	Explanation
<code>levelInput</code>	<code>input</code>	Ignoring the fact that 'input' is a keyword in many languages, this is not a meaningful name. You might input several things as time goes on.
<code>player1Score</code>	<code>p1s</code>	Nobody wants variable names that are longer-than-average variable names. A variable name that has a purpose is better.
<code>btnOK</code>	<code>Button1</code>	Default names (such as 'Button1') are not good. A coincidence, such as if this button has a value of '1'. When it comes to naming convention (such as prefixing text boxes with 'txt'), and use of underscores.



There's no reason to get anything other than full credit for naming conventions if you spend about as much time to type as a good name. Given the fact that, other than the need to submit anything under 'naming conventions', this might be the easiest way to pick up marks.


INSPECTION COPY

COPYRIGHT
PROTECTED



3.5: Validation

MARK BAND 1	MARK BAND 2	MARK BAND 3
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.



The word 'validation' is mentioned in the design section as well as the implementation and development testing, the word 'robustness' also involves validation. If you do not implement validation at all, you will lose marks.

It's possible to pick up full marks for validation simply by submitting your code. It's a little easier for the person marking your work (and to reduce the likelihood of something being overlooked), you are strongly recommended to signpost it. This would mean including a comment 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 a null value or no data is. A review might use this type of check, since it ensures that a user includes any type of character.
Range check	Useful with numbers or dates, where data values must be within a certain value, or greater than / after a certain value (or both) to prevent inappropriate quantities on an ordering system or to prevent items being purchased and then backdated.
Length check	Making sure a certain number of characters is entered. This can be an upper bound, a lower bound or both, as appropriate. Usernames and passwords are often subject to length checks.
Type check	Prevents numbers being entered in text-only fields and vice versa. Useful in a wide range of situations.
Format check	Ensures the entry of the correct character types in the field. For example, an email address must have an @ sign, but not at the start or end. It must stop after the @ sign, but this can't be at the end, or it must be followed by a dot. Format checks are also useful for postcodes and National Insurance numbers, though they require somewhat more involved coding than most other checks.
Lookup check	This is applicable where every single valid entry that can be entered is in a list somewhere. This might be a dropdown list, from which a user selects, or it might be a text file containing every word in the English language. It can be useful in a lookup check applied to a word game. For example, a range of buttons (such as in choosing where to play in a game) can be used for a range of lookup check.



When you're incorporating validation into your solution, you're making a decision. You can earn up to 4 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, 'creation of a new username'.
Mark band 2: Describe	Add some specifics. In the instance of a length check, there should be an upper bound, a lower bound or both. For example, 'a length check that ensures usernames are between 8 and 16 characters long'. A message that tells users the minimum and maximum length.
Mark band 3: Explain	Explain can mean 'why' or 'how'. In this instance it means you should explain your choice of validation routine, as well as how the range being defined as 8 to 16 characters. In terms of 'how', you should present your code and walk the reader through it.
Mark band 4: Justify	In this instance, a length check was chosen, although other checks could have been used. A format check could have been used to ensure that the user input was only letters and numbers. A lookup check could have been used to check if the username already existed. Why not one of these? Why not a combination of them?

3.6: Review

MARK BAND 1	MARK BAND 2	MARK BAND 3
<i>Formal review is not required for mark band 1.</i>	Comment on the success or otherwise of the solution at some point during its development.	Mark band 2 plus: Extend the review to more than one key stage throughout the 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 the whole solution.

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

- A summary of what you've done in the outgoing iteration, ideally linked to your previous iteration.
- 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.

INSPECTION COPY

COPYRIGHT
PROTECTED



Iterative Development » Checklist

INSPECTION COPY

MARK BAND 4:
<ul style="list-style-type: none"><input type="checkbox"/> Collectively, the evidence for the iterative stages covers your entire solution (documented in one or more of the iterative stages)<input type="checkbox"/> Each iterative stage is clearly linked to the breakdown of the problem document<input type="checkbox"/> Each stage is backed up with detailed descriptions (what are you doing?), explanations (what is that?) and justifications (why did you do A when you could have done B?)<input type="checkbox"/> Each iterative stage constitutes a prototype; this does not mean you need to produce a full version of your solution at each stage – simply that you should have produced something that is working and testable<input type="checkbox"/> Code demonstrates modularity and good structure at all stages<input type="checkbox"/> Annotations cover all subroutines and all variables throughout your code<input type="checkbox"/> Every single variable, along with anything else you have chosen a name for, has a meaningful identifier, such that the purpose is clear just by reading the name<input type="checkbox"/> Evidence of validation is provided at all appropriate stages<input type="checkbox"/> All iterative stages are reviewed, with the review then feeding into the next stage. The final iterative stage covers what you might have done next, had the project continued
MARK BAND 3:
<ul style="list-style-type: none"><input type="checkbox"/> Iterative stages are still required to cover your entire solution<input type="checkbox"/> Each stage should still be linked to the breakdown of the problem document<input type="checkbox"/> Each stage should be described (what did you do here?) and explained (why did you do that?); justification is not required in this mark band<input type="checkbox"/> Prototypes (plural) are required, but not for every iterative stage<input type="checkbox"/> The solution needs to be modular; classes as required, and subroutines called where appropriate to avoid minimal duplicate code<input type="checkbox"/> The majority of the code should still be commented, although parts that are clearly necessary for functionality or a near-copy of already-commented code might be missing<input type="checkbox"/> Most variables and data structures have sensible, self-documenting identifiers, with a few exceptions to this<input type="checkbox"/> Validation is implemented and documented more often than it is missing<input type="checkbox"/> A minority of stages might be missing a review section
MARK BAND 2:
<ul style="list-style-type: none"><input type="checkbox"/> Some parts of your code, which may be evident in the full listing in your appendix, are demonstrated or discussed in this section<input type="checkbox"/> Some skill is demonstrated in making the solution modular and well structured, and the most efficient means of solving a problem might not always be used<input type="checkbox"/> Some annotation, which is useful, rather than simply of a token presence, is included<input type="checkbox"/> Some variables and data structures have self-documenting identifiers<input type="checkbox"/> Some useful validation is included and documented<input type="checkbox"/> Reviews exist for some iterative stages, and offer some genuine insight
MARK BAND 1:
<p>The mark scheme for mark band 1 is a list of aspects that are missing rather than a list of aspects that are present. If the majority of modularity, annotation, self-documenting identifiers, validation, or reviews are missing, or only exist in some token way, you can expect a score from mark band 1. The lower that score will be.</p>

COPYRIGHT
PROTECTED

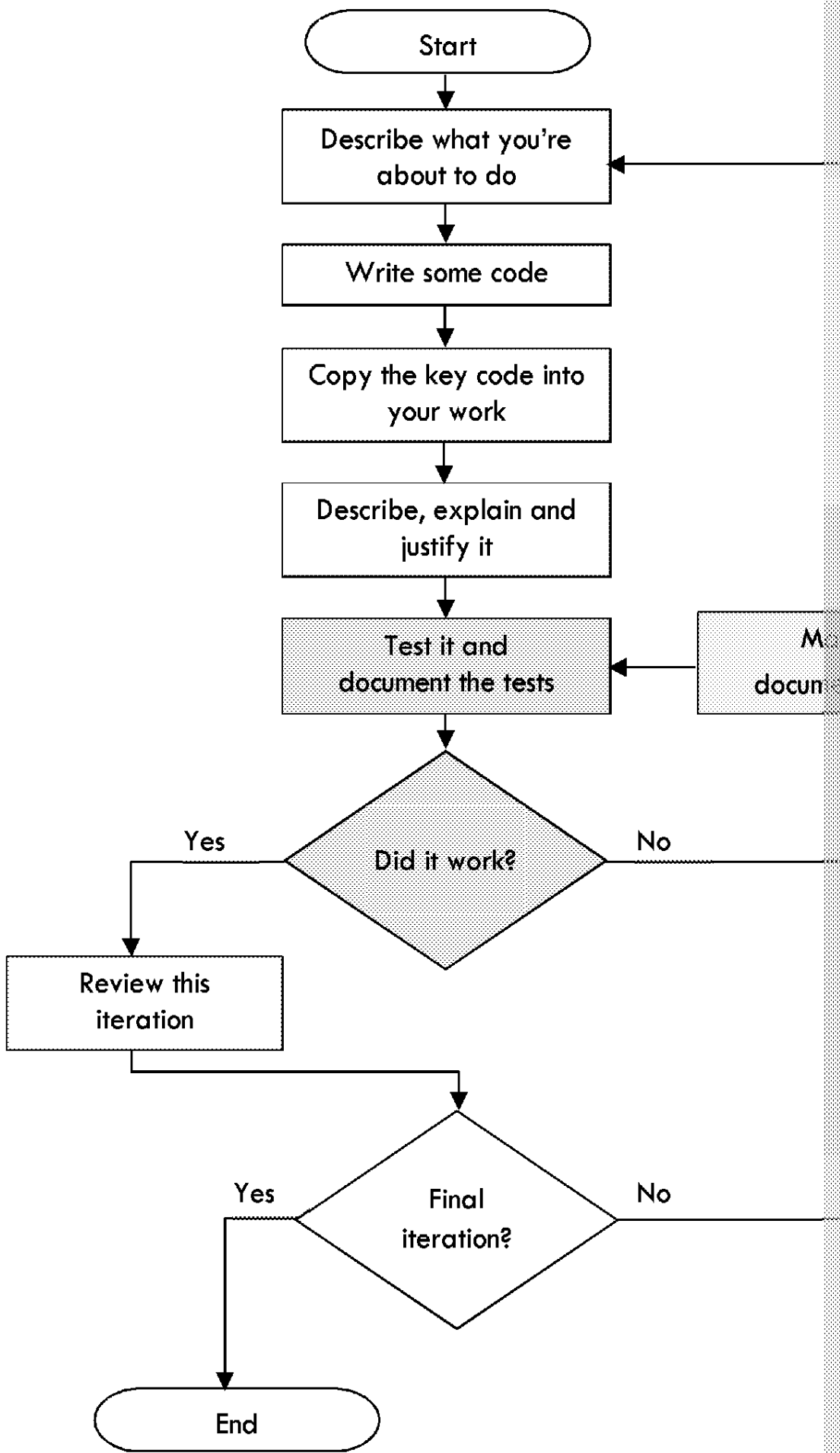


4: Iterative Testing (10 marks)

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



INSPECTION COPY

COPYRIGHT
PROTECTED



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



4.1: Testing

MARK BAND 1	MARK BAND 2	MARK BAND 3
Provide some evidence of testing throughout the iterative process, i.e. before the solution is complete.	<i>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.</i>	Mark band 2 plus: Extend testing to cover most of the iterative development stages.

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

4.2: Remedial actions

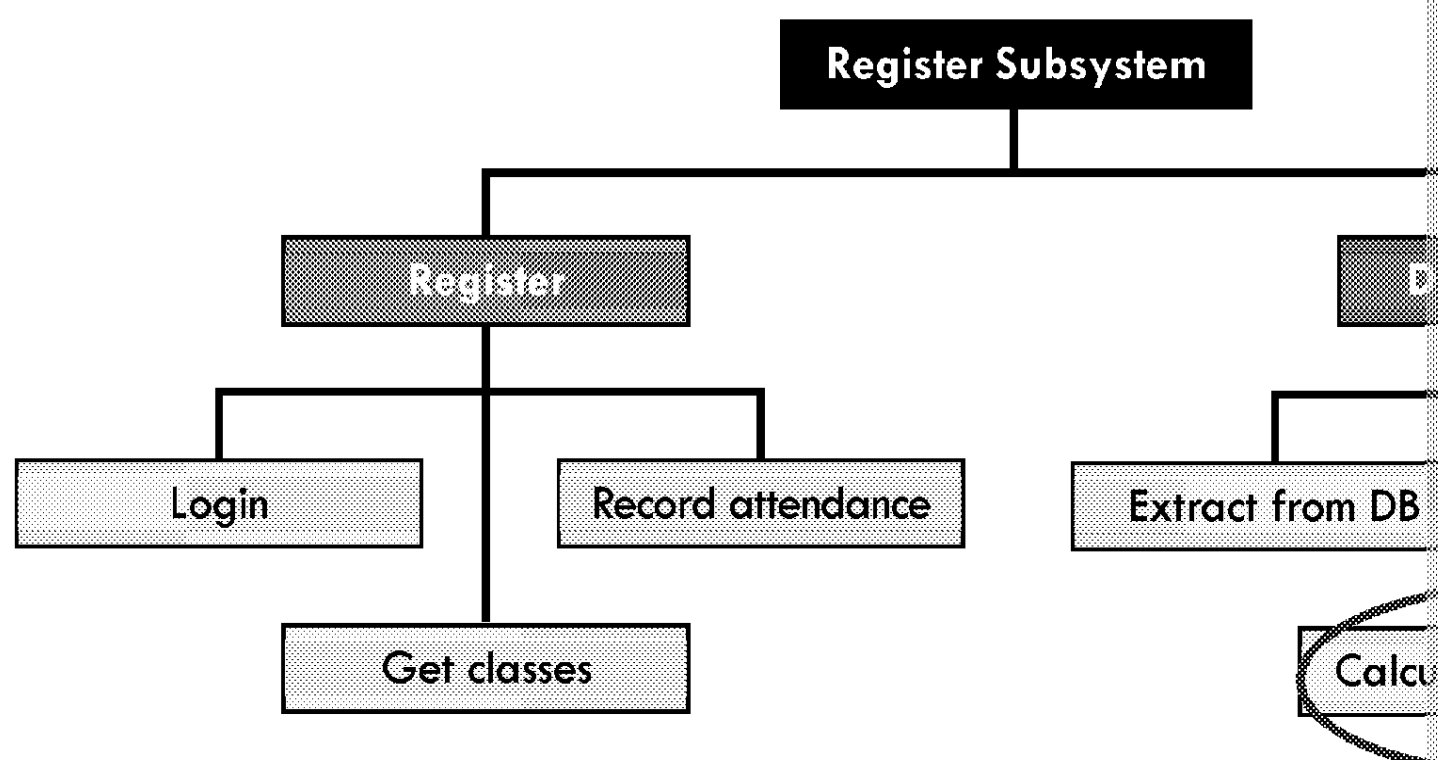
MARK BAND 1	MARK BAND 2	MARK BAND 3
<i>Remedial actions are not required for mark band 1.</i>	Demonstrate, with evidence and commentary, how at least two tests failed to produce the expected outcome, and what changes were made as a result.	Mark band 2 plus: Include explanations of why those particular changes were made.

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

This part of the project is best addressed through the use of annotated screenshots showing 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 went wrong
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 number of classes attended and maximum number of classes. Ultimately, my system comprising all attendees, sortable into either alphabetical order or order of attendance can be created, I am going to ensure that a single individual's data is correctly calculated occurs, that error exists only once, rather than repeatedly.

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

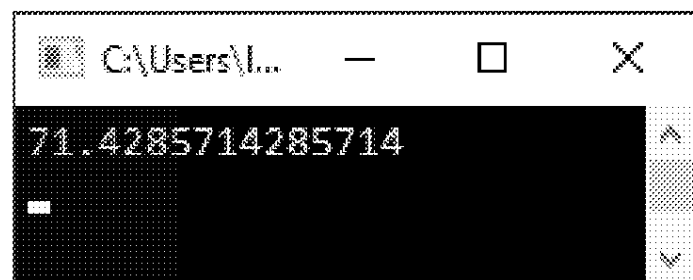
```
1 Sub Main()
2
3     'Array, which will normally be populated from DB
4     Dim attendanceArray() As Char = {"\", "\", "0", "A", "\"}
5
6     Dim possible As Integer = 0 'number of possible attendances
7     Dim actual As Integer = 0 'number of actual attendances
8
9     'loop through the array
10    For loopCount = 1 To attendanceArray.Length
11        'don't count A for authorised absences, otherwise
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
```

INSPECTION COPY

COPYRIGHT
PROTECTED



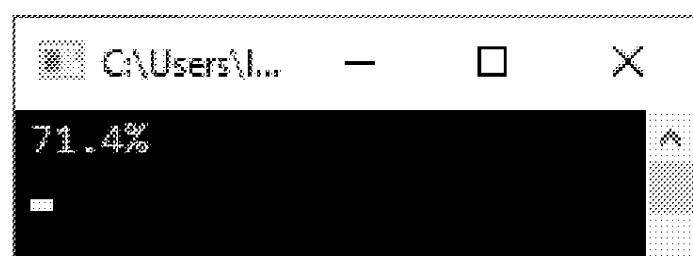
The attendance array would normally be populated with data extracted from the database, but for now it is populated with the test data specified in my design. I use the following possible attendance marks – a slash for present, a capital O for absent and a capital A for authorised absence. Authorised absences should be ignored by the system and should not affect the attendance calculation. 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 displayed than are needed. It might also be useful to have the percentage sign displayed. The target is for the 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) & "%")
```

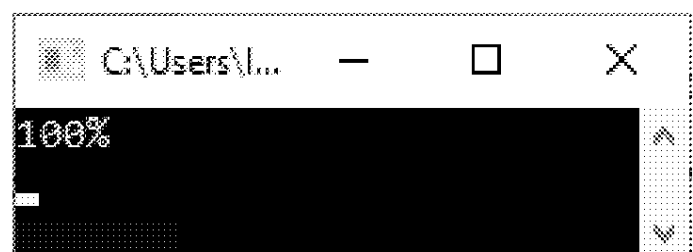
Here, I have used the `Math.Round` library function to display the answer to a single decimal place. I also used the ampersand concatenation operator to append a percentage sign. This approach will work as long as the values of 'actual' and 'possible' will remain unchanged and can be used in subsequent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



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

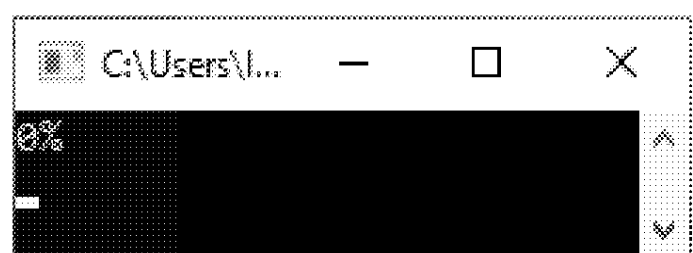
```
Dim attendanceArray() As Char = {"", "", "", "", "", ""}
```

To test that 100% attendance is correctly calculated:



```
Dim attendanceArray() As Char = {"0", "0", "0", "0", "0", "0",
```

To test that 0% attendance is correctly calculated:



```
Dim attendanceArray() As Char = {"\n", "0", "0", "0", "0", "0",
```

To check that the loop picks up a single 'present' mark in the first element of the



Dim attendanceArray() As Char = {"0", "0", "0", "0", "0", "0",

To check that the loop picks up a single 'present' mark in the last element of the



I am not testing for invalid data (i.e. a character other than \, A or O) as validation system – in the 'Record Attendance' subroutine. This is a fairly small part of the whole solution, it was appropriate to have created it and tested it thoroughly before. A single decimal place is not displayed when the percentage figure is a whole number. A significant enough anomaly to spend any additional time here – there is nothing. This part of the solution will now be tested alongside the 'Extract from DB' subroutine. 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 a small scale of what it does. Nevertheless, all of the pieces are there. They are about to be coded, placed in the context of the whole system. Code (which is self-documenting) is written, described and tested. The outcome suggests a change is made, which is coded and retested before being reviewed. At all times, we are being taken. Why this test data? Why the console output? Why no validation? This leads seamlessly into what the next iterative cycle would be.

INSPECTION COPY

COPYRIGHT
PROTECTED



Iterative Testing » Checklist

MARK BAND 4:
<div><div><input type="checkbox"/></div><div>Each stage of the iterative development process involves testing; note that mark if your iterative development work contains enough stages (i.e. you are not just testing the final product)</div></div> <div><div><input type="checkbox"/></div><div>There is evidence, and more than simply token evidence, of tests failing</div></div> <div><div><input type="checkbox"/></div><div>Documentation has been included as to what was done to remedy all failed tests</div></div> <div><div><input type="checkbox"/></div><div>Remedial action (action to fix code after a failed test) is described (what did you do it?) and justified (why did you do A when you could have done B?)</div></div>
MARK BAND 3:
<div><div><input type="checkbox"/></div><div>Most iterative stages involve testing, but not necessarily all; you might have a partially documented solution, or part of a fully documented solution</div></div> <div><div><input type="checkbox"/></div><div>Some failed tests are included, though not necessarily in all iterative stages</div></div> <div><div><input type="checkbox"/></div><div>Documentation has been included as to what was done to remedy all apparent failures</div></div> <div><div><input type="checkbox"/></div><div>Remedial action is described (what did you do?) and explained (why did you do it?)</div></div>
MARK BAND 2:
<div><div><input type="checkbox"/></div><div>Some of the iterative stages include testing Note that the mark scheme descriptor '... provided some evidence of testing during development process' covers mark bands 1 and 2, so is anywhere between the bottom of this band, you would expect around half of the iterative stages to include testing. More than this would be '... most stages of the iterative development process include testing' which is mark band 3.</div></div> <div><div><input type="checkbox"/></div><div>Some failed tests are included</div></div> <div><div><input type="checkbox"/></div><div>Remedial actions are described, with supporting evidence, but not explained</div></div>
MARK BAND 1:
<div><div><input type="checkbox"/></div><div>Testing needs to have taken place during iterative development; if there is no evidence of testing during development has concluded, no marks can be awarded</div></div> <div><div><input type="checkbox"/></div><div>No failed tests or remedial actions are expected in this mark band</div></div>

INSPECTION COPY

COPYRIGHT
PROTECTED



5: Post-development Testing (5 marks)


This is the final round of testing, after which no additional development takes place. If you have completed any testing to improve your solution, you're still in section 4. This is the smallest section, so the least time should be allocated to it, so you should spend the least time here. Nevertheless, it's still important. A good test development testing section can make the difference between two grades in the final mark.

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. Here, you should aim to test whether those separately developed pieces work together. At this point, a formal test table can be used, which should be informed by the test data. 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 the test number should be unique
- 2. A description of the test, in sufficient detail that a competent person could repeat the test exactly as you carried it out
- 3. Test data; for example, when you tested the login screen, exactly what username and password you used
- 4. Expected outcome; this is a description of what a successful test looks like, so that a competent person could judge a passed or failed test without any other information
- 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' (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 to meet the original user requirements

Test #	Description	Test Data	Expected Outcome	Actual Outcome



A test table is not essential; it is not mentioned in the mark scheme, and some candidate work uses a more narrative structure. However, use of a table will be rewarded for 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 use. You should include screenshots of data stores, such as a database, before and after an operation to add or delete data. You might include photographs of a screen if screenshots are unavailable. Each of these, placed after the test table, should be uniquely numbered, with those numbers being entered into the 'evidence' column of the test table. If your test evidence takes place on video rather than screenshot, the 'evidence' column should contain time indexes instead.

INSPECTION COPY

COPYRIGHT
PROTECTED





5.1: Testing for function

MARK BAND 1	MARK BAND 2	MARK BAND 3
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.

Testing for function answers the question 'does it work?' What you will actually test 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

- Does validation prevent the program crashing (such as in the event of logging in)
- Does the solution alert me to a missing database, rather than crashing or looping


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
<i>Usability testing is not required for mark band 1.</i>	<i>Usability testing is not required for mark band 2.</i>	Provide commentary and evidence to show testing for usability.

Usability will also vary based on the nature of your solution, but the focus will be on 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
- 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 covered in any of the other columns, it should be placed into the 'commentary' column. It is likely to be explanations of why a test was necessary, or how important a test was.

Post-development Testing » Checklist

MARK BAND 4:

- Testing, which includes evidence (such as with a screenshot) and commentary, c
- ☐ Function – do the key processes perform as expected?
 - ☐ Robustness – does the solution function irrespective of invalid data input. i might make the solution crash or otherwise malfunction?
 - ☐ Usability – do all aspects of the interface function as expected?

MARK BAND 3:

- Testing, which includes evidence (such as with a screenshot) and commentary, c
- ☐ Function – do the key processes perform as expected?
 - ☐ Usability – do all aspects of the interface function as expected?

MARK BAND 2:

- ☐ Evidence of final testing is included, which tests for function (checking that expected); commentary may be either weak or missing

MARK BAND 1:

- ☐ Testing needs to have taken place during iterative development; if there is development has concluded, no marks can be awarded
- ☐ No failed tests or remedial actions are expected in this mark band

INSPECTION COPY

COPYRIGHT
PROTECTED



6: Evaluation (15 marks)

The evaluation is the point at which you look back at the solution and examine the problem. At this stage, you can actually gain marks for any shortcomings you may have well on them. The evaluation is informed by all previous sections, as you might have done differently in each of the analysis, design, development and testing stages. If that's the case, 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
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 <i>partial success</i> 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.

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

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

1. To what extent have you met that success criterion? You might not have attained it; you might have exceeded the standard you set for yourself; it might be a partial success. Don't shy away from success or failure in this part of your work, as you get credit for both. You should pay close attention to any success criteria which lie somewhere 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 it does work, 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 meet that criterion in future. Remember, this is a hypothetical future, and you'll never have to implement these changes, so don't be afraid to be ambitious. Say **what** changes you would make, how you would make them, and finally talk about **why** these changes would make you successful.



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

COPYRIGHT
PROTECTED





6.2: Assessing usability

MARK BAND 1	MARK BAND 2	MARK BAND 3
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.

Usability was initially addressed in section 2.4, where you spelled out which usability features you deployed within your solution. These might have been particular layouts, specific code snippets or features or a combination of all of these. Under the heading 'usability', examine your solution in 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
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?

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

1. What was supposed to happen?
2. What, if anything, did happen? Your limitation might come in the form of performance issues, a crash, or a feature that doesn't work as intended.
3. Why did this take place? This might be a technical reason, or it might have been a limitation of an ambitious project or an underestimation of the complexity involved.
4. How would you approach problems like this differently in future in order to avoid this limitation?

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

- **Self-documenting identifiers** – everything you have named, you have named it so that its purpose is apparent from the name alone. This applies to variables, data structures, class names, function names, external files, fields within external files and anything else that you, the programmer, create.
- **Modularity** – each subroutine should be self-contained, and separately designed, so that it has no reliance on global variables that could have been unexpectedly changed.
- **Appropriate use of variables and constants** – variables and constants should be used to store data for other processes, rather than literals. Additionally, these variables and constants should be easy to find and change. If the VAT rate changes, for example, and your solution uses the VAT rate, there should be a single easy-to-locate change to make to your code to reflect the change.
- **Detailed annotation** – there should be comments throughout your code describing what the code does. There should never be a section of code without comments.
- **Appropriate version numbers** – given the sequence of prototypes that have been produced for a project, it's possible that someone might accidentally begin changing the wrong version of the code. There should be measures in place to ensure that older versions survive, but they should not be used for development.



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)?

6.4: Quality of written communication

The final element of your work to be assessed is how well written your evaluation is. This is beyond the scope of this resource to teach you how to write well, but there is some advice to make the best use of your current abilities:

- Don't make unsubstantiated claims. If you say that something worked, or didn't, provide evidence to back this up. Each claim in your evaluation should be supported by a reference to a test number.
- Incorporate structure into your work. This can be done using subtitles in much the same way as used here. Have a section entitled 'test results', where you talk about the test results, 'usability', and so on. Avoid bouncing back and forth between different topics, as this will cost you marks.
- Include a table of contents to allow sections to be located more quickly (and more accurately).
- Keep it relevant. You should write about success criteria, testing, usability, 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 it.

For some students, the evaluation can be a little too open-ended. If you're struggling to start at the start of the evaluation, or a section of the evaluation, the following templates might help. They don't cover everything, and are not intended to – each evaluation should be your own. Use them to make a start.

Upbeat opening	One key strength of the solution that is apparent throughout the development is _____. Although challenges were encountered, including _____, and _____, the team has demonstrated that a great deal of functionality has been provided. During the analysis stage, it was apparent that the stakeholders' requirements were clear, and this has clearly been delivered, as seen in test _____.
Downbeat opening	While some success has been encountered during the development, 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 _____ 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 some key functionality, including _____, has been delivered. Admittedly, the solution does not provide 100% of its intended functionality, and _____ are incomplete – but some success has been encountered.
Providing evidence	Evidence for this claim can be found in tests _____ and _____, and the process of _____. Test _____ shows the state of the system before _____, and the state of the system afterwards. This can be seen to be the case throughout section _____ of the evaluation, in particular during test _____ and on the associated screenshots.
Future improvements	Although this solution fully satisfies all success criteria, there is potential for further development. Firstly... Naturally, the first priority in improving the system would be satisfying the remaining success criteria, but beyond that... There is scope for improvement, with the most likely future focus being _____.

Evaluation » Checklist

MARK BAND 4:
<ul style="list-style-type: none"><input type="checkbox"/> Evaluation needs to compare all test evidence (iterative and post-iterative) (section 1.7) in order to assess the success of the solution<input type="checkbox"/> Each success criterion, including usability and maintainability, should be assessed as a success or failure, with explanations and evidence to back up each assessment<input type="checkbox"/> Limitations should be critically addressed – what are they, and how significant are they?<input type="checkbox"/> Future improvements should be suggested for partial successes and failures, including usability and maintainability
MARK BAND 3:
<ul style="list-style-type: none"><input type="checkbox"/> Evaluation still needs to compare all test evidence with all success criteria in order to assess the success of the solution<input type="checkbox"/> Each success criterion should be assessed as a success, partial success or failure, with evidence to back up each assessment; usability and maintainability do not need to be explained<input type="checkbox"/> Evaluation should still include usability features and maintenance issues, but they should be described rather than explained<input type="checkbox"/> Limitations should be critically addressed – what are they, and how significant are they?<input type="checkbox"/> Future improvements should be suggested, but only with regard to the success criteria
MARK BAND 2:
<ul style="list-style-type: none"><input type="checkbox"/> Comparisons of success criteria and test evidence are still required, but some success criteria might not be included in the evaluation<input type="checkbox"/> Evidence of usability features should be incorporated into the evaluation in a descriptive way<input type="checkbox"/> Limitations still need to be addressed, but only in a descriptive way; their significance should not be mentioned
MARK BAND 1:
<ul style="list-style-type: none"><input type="checkbox"/> Determine whether the solution is a success or a failure based on the test evidence. There should be no reference to the original success criteria

INSPECTION COPY

COPYRIGHT
PROTECTED



Suggested Project Structure

INSPECTION COPY

COPYRIGHT
PROTECTED



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. It's important to understand what they mean, so if you read this and are still unclear, check definitions online or in a textbook, or do whatever you can to ensure you understand.

Annotation	In terms of programming, 'annotation' or 'comments' make use of text that sit alongside program code. Compilers ignore annotations, but humans do not. Annotations are written for humans, rather than the computer. Good-quality program code with annotations can greatly aid 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. Examples include arrays, lists, graphs, trees, queues, stacks and dictionaries, as well as more complex structures like linked lists and hash tables.
Describe	The most open-ended verb in any mark scheme, 'describe' can be used to describe 'when' and 'where'. You might be describing a piece of existing code (when does the validation take place?), a user interface (where is the 'OK' button?), a potential user (what is the user's role as a stakeholder?) or a process (when does the validation take place?). If you're writing in general terms about something, you're probably using 'describe'. 'Describe' can be found clustered around the bottom-to-mid-range of the mark scheme.
Evaluate	The word 'evaluate' comes from the word 'value'. If you're evaluating something, you're assessing its overall worth, as well as the worth of any individual components. For example, if your interface looks good (positive), but does not work (negative), you need to weigh these two factors against each other. Coming to a 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 once the final solution is reached, you come to the conclusion that something either works, doesn't work, or is a mix of the two. 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 or a log file, depending on what you're trying to prove. Evidence might be in the form of a completed questionnaire: '80% of users prefer the GUI interface'.
Explain	A common verb among the mid-mark ranges; if you're explaining something, you're answering the question 'why', sometimes 'how' and occasionally 'what'. For example, explaining your choice of programming language, why did you choose Python?
Identify	You can identify something in a sentence or less. If you were asked to list the peripherals required for your solution, 'mouse, keyboard, monitor' would be identifying. Any more detail, such as the resolution of the monitor or the type of keyboard, would be describing.
Iterative	This describes the process of making multiple passes through a task. In an iterative approach to software development, you would design, then develop, then test a solution. In an iterative approach, which is what this project is about, you produce part of the solution, then review it. After this, you might have a second attempt at producing that same part, or even a third. In an iterative process, some stages take place more than once.

INSPECTION COPY

COPYRIGHT
PROTECTED





Justify	<p>The word 'justify' appears in the mark scheme 18 times, to justify an answer. Simply put, it means giving an explanation for something. Unfortunately, the word 'explain' also appears quite a lot.</p> <p>To justify something, try this three-step approach:</p> <ul style="list-style-type: none">A. Describe what you did – whatever it is you're going to do.B. Describe what you might have done instead, but didn't. For example, you didn't 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	<p>This is a measure of how well a problem or a solution is broken down into smaller parts. In software development, you should be aiming to make a modular solution. This involves the creation of classes, and it will certainly involve breaking down a task with an individual subroutine. Through good use of variables and function values, modularity reduces the amount of code by minimising repetition. Copying and pasting large amounts of code, the chances are that you will create a modular solution.</p>
Problem	<p>In programming, this simply means the situation for which a program is written. It doesn't necessarily mean there's something wrong with the current situation. Trying to improve a situation, it's not perfect.</p>
Prototype	<p>A prototype is a version of an unfinished solution. It might be a complete solution, or perhaps some part of it. It is typically imperfect. The purpose of a prototype is to learn something from it in order to make the final solution better.</p>
Solution	<p>This covers a program, together with any associated hardware and documentation. It is created in response to the problem you identify.</p>
Stakeholder	<p>A stakeholder is anyone with an interest in your solution. The user is the person who will use your program, but there may well be others who have a say in what the program does.</p>
Usability	<p>A usability feature is some aspect of a program's user interface that makes it easier to use. A shortcut key and an icon with a tooltip are usability features. Plain, legible text is also a usability feature, because it makes a program easier to use than struggling with small text.</p> <p>The following list of usability features is by no means exhaustive:</p> <ul style="list-style-type: none">• Help features, whether that's a discrete 'help' section or context-sensitive help, whereby right-clicking on a button tells you what the button does.• Error messages that offer guidance on how to interact with the program.• Consistent positioning of controls, such as always placing the 'OK' button at the bottom-right of a window.• Ensuring resemblance to other software, maximising the user's familiarity and immediate use of the software.• Minimising the number of clicks needed to perform a task.• Structuring screen layout so that controls are positioned in a logical order, needed, i.e. left to right, top to bottom.• Preventing user error by disabling error-inherent features, such as preventing numeric entry into a particular text box to be ignored.• Speeding up data entry by including easily removable characters.
Validation	<p>This is a process that ensures that entered data is reasonable. It is a check to prevent data from entering the system. This would include checking for a date being some day in the future, or a 50-character postcode.</p>