H446 Decomposed Mark Scheme

Project Complexity

As a rule of thumb, projects should involve at least one of the following:

- Object oriented approach (most straightforward)
- Multi-layer approach (E.g. use of database, PHP, Javascript)
- Database-driven (but more than simple database access)
- A substantial programming element, e.g. use of 'A-Level' quality algorithms, e.g. A*
- E.g. scheduling, simulations, game vs. computer, multi-user, simulation.

Candidates must avoid taking on a project that is too simple or far too complicated given the timeframe (in which case candidates are often tempted to put too much time into programming and not enough into the writeup). However, an over-ambitious and incomplete project is far better than under-ambitious project as it can still get very high marks. LOOK AT OCR's EXEMPLAR 'Hand Gesture' project. You'll note that it uses objects, but the hand tracking code is very simple, if not copied from a TUTORIAL? The majority of code is trivial interface code. **To get marks for 'usability features'** candidates <u>MUST do a project that involves a significant user interface.</u>

Bad projects might feature:

- Static 'quizzes' / web sites with basic client-side password authentication.
- Basic text file storage
- Over reliance on frameworks, e.g. Unity's engine doing all the hard work with trivial coding on top.
- Simple record retrieval from databases

Possible languages/data structures: C/C#, Python, Java, Swift, JSON, NodeJS, Unreal/Unity, LUA, RobotX/Monkey X, Javascript WITH other layers/technology, e.g. Php/MySQL, ASP/MSSql.

Notes:

- Whilst a 'best-fit' approach should be taken, one should be careful to give added weight to the most substantial part of each section - namely in the design section, 'structure' and 'algorithms'. Moderators don't seem to take a best-fit approach, instead, look to give the lowest possible mark.
- Candidates should use **headings** to reflect each 'area' and **vocabulary** that reflects the mark scheme.
- Candidates would do well to spend **longer on their write up than they do the coding**. The coursework is testing candidates more in their ability to produce a write-up than it is to be able to produce clever code. Trust me.
- **Iterative development** means: Design each 'feature' by repeating development-testing until it is complete. Now add the feature to the main program. It should be largely possible to develop each feature in isolation from the main program.

Definition of command words:

- **Identify -** 'What' - state what it is from research
- **Describe** - 'How', make clear with descriptive language
- **Explain** - *How **and 'why'**, make your approach 'understandable' in the eyes of the reader*
- **Justify** - Consider different approaches, points of view etc before making a rational conclusion
- **Evaluate** - A verdict on the extent to which the project has been successful, based on evidence 'for' and 'against'. Conclude based on what you judge to be the most important factors and justify HOW you've come to that conclusion
- **Consider** - state what you think and observe - backed up by your own evidence and external evidence. Contrast with contrary views that relate to your original thinking and observations.

What is meant by an ‘iterative’ approach?

First things first, OCR definitely favours the write-up process over programming content. Candidates would do well to spend longer on their write-up than the actual programming. The adage ‘it’s not what you do but the way that you do it’ rings true.

The only part that students need to ‘iterate’ over is each ‘feature’ of the project that is being implemented. Or to put more simply, just show how features are programmed, debugged and fixed. Then repeat. There is no expectation or credit give (learned from bitter experience) for iterating over the complete analysis-design-development-testing-evaluation process. It is more like…. Analysis-design-development [iterating over each ‘feature’]->testing->evaluation.

Analysis

- **All projects should start with a paragraph to outline what the project is about, i.e. “This project is about developing a program to…”** ○ It is ‘ok’ to reinvent the wheel to some extent - candidates can challenge themselves to create their own take on an existing computer program.

Area	1-2 marks	3-5 marks	6-8 marks	9-10 marks
Features that make it solvable by **computational methods**	Identified *some*	Described	Explained	Justified
Stakeholders and **how** they'll use the solution to meet their needs	Identified	Described (characteristics, lifestyle, role etc)	Explained why the solution will meet their needs	Justified why the solution will meet their needs
Solution features based on **research** of existing solutions	Identified *some*	Identified	-	-
Approach based on **research** of existing solutions	-	-	Described	Justified
Solution features, *in addition to researched features*	Identified	Identified what's ESSENTIAL (and what's not)	Described	Explained
Limitations	Identified	Described	Explained	Justified
Requirements - what needs to be in place/what needs to happen to be able to succeed?	Identified *some*	Identified *most*	...including hardware and software (IF APPLICABLE)	Justified
Measurable **success criteria** *in terms of 'functionality", 'usability' and 'robustness'*	Identified *some*	How they will be measured	Identified *most*	Justified

Area	Explanation and indicative content

Features that make it solvable by **computational methods**	It is unreasonable to expect candidates to link EVERY FEATURE of their intended solution to each method at this early stage. ***A <u>feature is something that the project could DO.</u>*** The following is NOT exhaustive, candidates must ONLY write about computational methods that are RELEVANT to their project. This section is ideally written as '***My proposed project is solvable using computation methods because it features…***" ● **Abstraction** ○ E.g. This project can be solved using abstraction because I will develop a game by using a framework (e.g. Unity) which hides the complexity of directly interfacing hardware… ○ E.g. This project is solvable with abstraction because the game will not need to be 100% realistic for it to be successful because... ● **Thinking ahead** ○ Inputs/outputs, e.g. This project can be solved because it has inputs and outputs including... ○ Preconditions ○ Caching ○ Reusable element e.g. This project is solvable using reusable elements because there are two players, each can share the same instance of a class... ● **Decomposition** ○ e.g. The project is solvable through developing components such as a player, head-up-display etc etc, each will be developed to keep the solution relatively organised and easy to understand ● **Procedures** ○ E.g. This project is solvable because it has key components that can be developed including… (could present a top-down diagram for example) ○ E.g. This project can be solved with a top-down and bottom approach because… ○ E.g. This project clearly has procedures including the need to update a score, reset a level etc... ● **Logic** ○ E.g. This project is solvable because it involves logic, for example when a player is attacked by a monster then, and only then… will happen. ● **Concurrency** (probably not applicable to most A-Level projects). Note that whilst many programs look like they are performing concurrent actions (e.g. a player moving across the screen collecting coins and the score going up), in reality, the code is not concurrent. Some students make the mistake that this happens at the same time.) ○ The project can be solved using concurrency because … and … could happen at the same time. NB for the highest marks, there needs to be (a) "**justification.** This involves consideration of a number of features and then concluding the best course of action. E.g. "I could make the game as realistic as possible and minimise abstraction or I could include the key elements. I will do the latter because…". (b) **why a computational approach is better than another approach** (efficiency, interactivity, convenience, accuracy, fewer people needed etc)

Stakeholders and **how** they will use the solution	**Identified -** The stakeholders are… *Technically this should include the candidate who will be aiming for high marks!* This section might or might not include explicit or anecdotal evidence of discussions with stakeholders. Candidates don't need a 'real' 3rd party end user, but still need to consider 'stakeholders'. **Described** - (a) Characteristics as appropriate, e.g. age, attention span, what they like, existing knowledge/confidence with technology etc. (b) How they will use the program - i.e. the 'features' they will want to use. **Explained why the solution will meet their needs** - as a result of using the features, an explanation of what it will achieve for the stakeholder. **Justified why the solution will meet their needs -** balanced discussion of features they'd like and which will be *most* important to them. Or contrast a manual method with a computational method, explaining why the computational method is a good approach. **"***The stakeholder would like features x, y, z, and in the future may like to see a, b, c. Whilst all these are important, I will focus on developing x and y because…"* *"The stakeholder could use a manual system [explanation of this system]. A computational approach will be better because…"* Candidates do not need extensive primary research (interviews and transcripts/surveys etc) from stakeholders at all, but could include some key points from primary research if it helps to address the above.
Solution **features** based on **research** of existing solutions	**The existing systems must include COMPUTER PROGRAMS** rather than research into manual systems that the candidate wishes to automate. NB: Whilst there is no requirement for candidates to write about 'usability features' until the design section. HOWEVER, OCR expect candidates to consider user interface features in this section. **Interestingly, features do NOT have to be derived from stakeholders.** Candidates typically either screen grab existing programs or write about existing manual systems in place - for at least THREE existing solutions. Candidates must clearly derive which features they want to include in their own programs from this research. NB at this stage, features only need be IDENTIFIED (not discussed or justified). This may be presented as a bullet-point list.
Approach based on **research** of existing solutions	Moderators appear to look for '***other ways that the user's needs could be met***'. So if, for example a candidate is creating a game, what other types of game would also satisfy the user's need to be entertained? If creating a football league scoring system, what other approaches could be taken - for example using a spreadsheet instead. Whilst seemingly largely an irrelevant section, it gets candidates thinking about alternative approaches to satisfying a need. Candidates should then conclude reasons for their approach. Whilst the following doesn't seem to be seen as relevant to moderators, to cover all basis and get candidates thinking, they should cover 'approaches' in terms of: (a) Programming style. (b) Programming language. (c) Development model, i.e. Rapid application development, (d) Order in which a project could be designed and developed and *optionally* (e) User interface design.

	Described - “This program looks like it was developed using a OOP approach because….therefore I will also do this…” **Justified** - “The data for this project could be stored in a file or database. I think a simple file is the best option for me - although this could potentially be easily hacked by the user, I will use a file because…” When considering (d) Order of design and development - which elements will be built first and why? Will it be to design the structure first (e.g. an explanation of the key modules/components to be developed and in what order etc) - top-down, and then fill in the details by designing the algorithms for each, or to create individual modules first and then piece them together to form the final solution (bottom-up). Most candidates do a combination of the two. ...And for the general project approach… “Most modern games seem to be made using a rapid approach, and OCR demands that I take an ‘iterative’ approach. I could use a linear approach, but this doesn’t get any marks and can result in not meeting stakeholder objectives, I will therefore take a … approach to this project…”
Solution features, *in addition to researched features*	NB: Usability features can be covered in ‘success criteria’ - the last part of the analysis. This is the one of the MOST important part of the analysis as should later be referred to in the development section.

Source of feature	**Feature**	**Essential? y/n** *(or could be given a number to indicate importance)*	**Description**	**Explanation**
Stakeholder	High score table	no	A table listing high scores for the top 10 players	This gives players something to aim towards… etc I’ve decided that this feature is not essential because...
Research				
Own idea				

Features could be *listed in order of importance* to show what’s most and least essential.

Limitations	Limitations of the proposed solution: - How far will the project go towards an ‘ultimate’ solution. I.e. it is perfectly normal for a project to focus on some key functionality rather than implement a complete fully-featured solution. - Which features will be implemented, and which will be left out and why. I.e. considers which are the most important features and which can be left out, or developed at a later stage - How features will be implemented - for example it may be that a program will accept inputs via a text input box rather than using a mouse/touchscreen because it is quicker to complete and more likely to finish by the given deadline. “For this game, I will only implement the first level because…” “I will only go as far as developing the underlying logic because…” General limitations tend to be further expressed in terms of: 1) Time 2) Existing knowledge 3) Resources such as additional programmers, art developers etc 4) Money Discussion may consider making use of third-party resources Justification tends to be a balanced discussion about the feasibility of which features could/shouldn’t or can’t be limited. “I need to learn how to code using… I could do this by watching tutorial videos or reading a book, I will … because…” “Ideally the game would be web-based because it will be accessible to anyone using any web browser, but because I only know Python, I’ve decided to tackle this project using Pygame…” etc “This game would benefit from original sounds and music tracks to be engaging and commercially viable, but I won’t have time to do these myself or the funds available to commission the content, therefore for the purpose of this project I will…”
Requirements - what needs to be in place/what needs to happen to be able to succeed?	OCR’s definitions may vary, e.g. ’requirements’ in this section is taken to mean ‘what needs to be in place for this project to proceed…’ as opposed to ‘project requirements’. This can overlap with ‘limitations’. Candidates should reference hardware and software requirements. This could include version numbers. “For this project to be successful, I will need to…” ● Need access to computers that have...installed ● Be able to install a web server… ● Use xyz programming language ● Have access to a HTML5 compatible web browser ● Have access to a graphics hardware that supports OpenGL 4.4 Justification may involve discussion of different options before concluding which requirement is most feasible or suitable.

Measurable **success criteria** *in terms of 'functionality", 'usability' and 'robustness'*	This is the other MOST important part of the analysis as should later be referred to in the development section. Candidates tend to struggle with this. The question is, "How will I know if I've been successful?". Top tip: to be able to complete the project evaluation, the success criteria MUST include reference to 'functionality", 'usability' and 'robustness'. Interestingly, "performance" is not mentioned. This is typically presented as a table: **Justification** could include discussion about why the criteria is important and/or suitability of how it will be measured. Candidates should be able to realise that some success criteria are only measurable by subjective means whereas others are definitely measurable on a met/not met basis.

Criteria	Category	Measured by (test method)	Justification
All essential features included	Functionality	Tick list	
	Usability	Development testing	
	Robustness		
Error-free	Robustness		
Secure	Robustness		
Reduce the time it takes to ...	Functionality		
Reduce costs for...	Functionality		
Provide entertainment	Usability & robustness		
User-friendly	Usability	- Use of standard GUI components - Interview	
A viable model for future development	Functionality		
Performance: do x in y seconds (or work on the given hardware)	Functionality		
Completed on time and on-budget	Functionality		
First release completed by Christmas	Functionality		

Design

Area	1-4 marks	5-8 marks	9-12 marks	13-15 marks

Program **structure**	Described	Decomposed (e.g. use of top-down design, modules, functions/procedures names)	Explained why they have been broken down the way they have	Justified
Algorithm design	Described, might be inaccurate (implied)	Appropriate and accurate	Explained	Justified
Usability features (consistency, ease-of-use, user interface design etc)	Described *some*	Described	Explained	Justified
Key **variables, data structures, classes** *as appropriate... from 'algorithm design'*	Identified		Justified	
Validation required	-	Identified	Explained	Justified
Test data that could be used DURING development	Identified some	Identified	Justified	
Post development data - AKA test data that can be used AFTER development	-	Identified	Justified	

Area	**Explanation and indicative content**

Program **structure**	This should start with an outline of the program - i.e. what are the main parts of the program? This could initially be presented as a top-down 'Jackson Structured Programming' diagram or similar, or more easily, a table. If using JSP, I'd recommend finding a program that lets you create these efficiently, or do them by hand and scan them in. This is much quicker than faffing around with drawing tools in a Word Processor! Diagrams could include top-down/JSP, OOP class diagrams, relational diagrams etc. **Program Part Description Explanation Justification** Menu Game High-score Settings
Algorithm design	Typically conveyed with pseudocode. Although flowcharts and data flow charts could be used, they will be time-consuming to make and may well not convey what needs to be achieved. There should be one algorithm for EACH element of functionality. A function should ideally have ONE purpose. For each algorithm… ● Start with a description of what it does "this algorithm is designed to…" ● Write the algorithm (pseudocode) ● Explain why you did it the way you have… ● ...incorporating ideas (ONLY IF RELEVANT) about other ways the same solution could be achieved. ● Explain how the algorithm fits in with the overall program structure/program design. *If a third-party developer could develop the project from the algorithms designed, then it is good enough.*
Usability features (consistency, ease-of-use)	It is also good practice to include accessibility considerations. Consider: Consistent vocabulary, accessible vocabulary, large buttons (if using touch-screen), consistent appearance (if window-based) This could include annotated drawings - including illustrated ideas interface/screen designs. Described - Description of usability features Explained - Explain the reason for chosen usability features, **in relation to project requirements**. Justified - Candidates should consider different possible usability features and conclude which one they will go with.

Key **variables, data structures, classes** *as appropriate...* *from ‘algorithm design’*	Best presented as a ‘data dictionary’, e.g: Variables and data structures are best presented as a table. NB It only need show KEY variables etc, and not every variable, for example that used as an index in a loop. **Module/Routine Identifier Data type/structure Description Justification** Classes are best shown by class diagrams - unlike the ‘program structure’ section, they should include a list of attributes and methods. Database designs are best shown by a entity relationship model diagram.
Validation required	This could be presented as a table and incorporate suitable test data. **Module/ routine** **Input/File input** **Validation rule(s) used** **Explanation of validation rule(s)** **Justification of validation rule** **Test data** **Justification of chosen test data** User details email format Checks to see if it includes @ symbol and at least one . This can reduce incidents of typing in fred@blogs.co m This includes both a valid

	incorrect data format and sdgsgsdg invalid format Validation rules include: Closed input, range check, presence check, length check, format check etc. Remember that test data, where applicable, should consider both valid and invalid inputs.
Test data that could be used DURING development	*See above*

Post development data - AKA test data that can be used AFTER development	Some programs may be tested during development with limited data, but then after production, be tested with considerably **more test data** - for example testing a program with 5,000 user records instead of the one that is used for development. It may be that after the program is developed, **test data is generated by test users**, for example the number of users who are able to successfully do x, y, and z using the software. Where a project has limited inputs (such as a game involving arrow keys), test data could comprise of **spamming inputs** (i.e. repeatedly pressing keys, and multiple keys at the same time in a short space of time. Candidates may consider **questions they will ask their stakeholders** after the project is fully completed. E.g. “is it easy to navigate?”, “to what extent is it easy to navigate?”, “to what extent does this program….?” It is useful for candidates to recognise that some testing will be functional (i.e. test tables), and other testing will be informal (e.g. feedback from test users). Both are required to be able to develop from one prototype to the next.

Development

- Candidates may use code from a third party so long as it is ACKNOWLEDGED! However, if the majority of code was NOT developed by the candidate, then the project should be marked accordingly (but judging by some projects I’ve seen, many a student has got away with it…)
- Candidates MUST make lots of screenshots as they go along, of both **CODE** and **USER INTERFACE**.

Area	1-4 marks	5-8 marks	9-12 marks	13-15 marks
Iterative development - what happened at EACH *stage*	Linear (1-2 marks) Some (3-4 marks)	Mostly	Explained what was done CLEARLY related to **solution features** from analysis	Justification of what you did
Prototyping	-	-	Some	A prototype for every stage (or each 'solution feature')
Structure	-	Some	Modular (or use of clearly defined subroutines or similar)	Modular and "Well structured"
Coding	Inefficient	Efficient	Efficient	Efficient
Code Annotation	Inappropriate	Brief	All key components	Aids maintenance
Variable/structure appropriate **naming**	Inappropriate	Largely appropriate ≈ 60-70%	Most ≈ 70-80%	All
Validation	None Little	Some	Most	All key elements
Reviews	Little	Some	Most	After all 'key' stages

Area	Explanation and indicative content

Iterative development - what happened at EACH *stage*	The best students tend to: 1) Start with a top-down approach, e.g. define classes, data structures, variable names etc *as appropriate* 2) Continue with a bottom-up approach where code is added, data structures as populated etc. Each stage will be accompanied by a thorough 'thought-process' discussion. 1-2 = 'Here is the code that…', 'I then write the code to do…' 3-4 = 'Here is the code that…', 'I made it better by…', 'I then write the code to do…' (inconsistently) 5 - 8 = 'Here is the code that…', 'I made it better by…', 'I then write the code to do…' (consistently) 9 - 10 = Explanation = 'I did it like this because…' 11 - 12 = Linked to analysis '...this was to develop the solution feature x as identified in my analysis' 13 - 15 = Justification = 'I could do it like this, or like that, but decided to do it like this because…' Each iteration needs to be 'signed off' by a stakeholder. **If candidates simply present their solution, it implies that either they did not take an iterative approach, or <u>they got the code from somewhere else</u>.** **If explanations of the code lack understanding in A<u>NY PART</u> of the code, unless they have CLEARLY acknowledged where the code came from, this st<u>rongly suggests malpractice</u>.**
Prototyping	Remember that a prototype is a test version that might lack functionality, its purpose being to elicit feedback. Therefore, it makes sense for candidates to present a prototype at 'logical' points in the project - whether it be a single prototype or several.
Structure	OCR seem to have loose definitions for what is meant by 'Modular' and 'structured' code. Good practice may be identified by: - Clearly named and used classes, names comply to a naming convention - Subroutines with a single purpose - Subroutines that are reusable - Subroutines that are relatively short - Subroutines flow in a logical order - Globals defined in a single place at the start of the code Poor practice may be identified by: - Very few subroutines - Overly long subroutines - Similar, repeating code patterns - Variables and routines seemingly defined in ad hoc order

Coding	The focus here is on the quality of code itself, not the naming of variables. Good practice: ● Developed with reusability and maximum flexibility in mind ● Algorithms generally work in an efficient way (this can be difficult to determine at a glance and is possibly the trickiest part of marking a project). Poor practice: ● Repeating code - code that looks and works in a similar way, that could be better put into a subroutine. However - note that in some instances, it may be quicker from a development point of view to replicate code. Provided a candidate has mentioned this (in code annotation) then this should not be taken for poor practice.
Code Annotation	Annocation can be taken to mean ‘code comments’ or ‘code commentary’. It is good practice for candidates to put this in the code itself. Avoid ‘death by screenshot’. Line-by-line annotation is NOT required. Students MUST show blocks of code and accompanying commentary - either within the code itself, or next to screen grabs of the code blocks of code. Inappropriate = annotation doesn’t describe what’s going on Brief = sporadic comments All key components - i.e. annocation to outline the purpose of each subroutine, but code within a subroutine is barely commented Aids maintenance - i.e. sufficient commentary throughout. NOT EVERY LINE MUST BE COMMENTED! Good commentary tends to include REASONS for why code was written the way it was as opposed to what it does. In fact, if comments are basic, i.e. “This assigns the value 3 to x” without explanation, this suggests the code was not entirely developed by the candidate.
Variable/structure appropriate **naming**	Naming should consistently follow: A ***naming convention***, e.g. camelCasing, underscore_method. Constants should stand out from variables. Subroutine and class naming should also follow a consistent naming convention. Every identifier should have a **sensible name** that is **descriptive** but **not ridiculously verbose**.
Validation	Some projects may involve very little validation, for example games where a player is controlled by a few keys. This makes it seem unfair that projects involving a large number of inputs are subject to greater scrutiny. Every INPUT into a project, whether it be a manual input or input via a file. Arguably, checking files for data integrity can involve considerable additional coding, and may be left out, but it would be wise to reflect upon this, as well as security considerations in the evaluation. Test data could include reference to testing with (a) valid, (b) invalid, (c) borderline data where appropriate given the nature of the project. Interestingly, the mark scheme does not require justification. This is best presented as a table:

	Area/module of program Input/File Validation required Game loop Arrow keys Only accept arrow keys Add person form DOB Range between … and ... ~ Email address It is also worthwhile for candidates to clearly highly any use of validation in their code.
Reviews	After every stage, a simple 'www' (what went well) and 'ebi' (even better if) is sufficient. The review needs to be APPLICABLE to what the candidate has done, and AVOID comprising of 'generic' statements. "I've now created all of the outline classes. This went well because it now gives me a solid structure for developing the remainder of my program…" "This took me longer than anticipated, so I will need to…" "If I was to do this again, I would… because…"

Testing to inform development - <u>in-development testing</u>

- Candidates would do well to present this as part of the development section.

Area	**1-2 marks**	**3-5 marks**	**6-8 marks**	**9-10 marks**
Testing	At some stages		At most stages	At every stage
Remedial action of failed tests	-	*Identified failures*	Explained	Justified

Area	**Explanation and indicative content**

Testing

Candidates will not do any testing until it is possible! (Stating the obvious, I know). Keep this in mind.

Students may present a 'test table', but this must be backed up with screenshots showing both **CODE** and **USER INTERFACE**. If at all possible produce **<u>VIDEO EVIDENCE</u>**. The latter is going to be the ONLY way to convince moderators that your code works, otherwise they may well claim that despite the screenshots, there is 'insufficient evidence'. Whilst candidates could produce extensive ***trace-tables***, this is likely to be time-consuming and is best avoided.

Testing should cover all possible inputs to the system and expected outputs, preferably based on the test data identified in the design section. It is a good idea to ensure all testing covers all validation rules used. Evidently, if candidates are producing a game with limited inputs, the testing will be limited, but should not be penalised for this.

<u>NB There is NO explicit requirement to:</u>

- ***<u>Explain or justify CHOICE of tests</u>***
- ***<u>Be explicit about the input data</u>***
- ***<u>Do any specific type of test e.g. alpha, beta etc.</u>***

*feedback from a OCR moderator suggests candidates MUST include 'before' and 'after' screengrabs of code changes, and keep a daily log of changes, and how in particular the changes inform the development of each iteration.

<u>E.g.</u>

Test Number Module/Subroutine What to input Expected output Pass/Fail? 1 Menu Test menu options Pass 2

Game/Movement Valid input keys Player moves Passed 2nd time ~ Invalid input keys Nothing

It is likely that there will also be feedback in response from prototypes which is likely to elicit change requests..

Prototype number Stakeholder response

1 Stakeholder: Want to be able to allow club members to attend more than one session per week.

Remedial Action	For failed tests… NB Candidates MUST NOT deliberately introduce errors into their programs. If everything works, then everything works. However, obtaining REAL stakeholder feedback is likely to elicit demands to change functionality, stakeholders are generally good at doing this. **Test number Explanation of remedial action Justification of remedial action taken** 2 Up and down arrow keys were inverted, updated This was the only way of solving the problem! constant values used for keys **Prototype** **Explanation of remedial action Justification of remedial action taken** **number** 1 I said this is not possible. This requires a change to I could make the necessary changes but I need to balance the time the database and significant additional coding. requirements with completing the essential features. For this reason, this feature will not be included until a subsequent cycle.. 2 Candidates should avoid adding trivial detail such as “stakeholder didn’t like the colours” etc.. unless it really is a significant issue.

Testing to inform evaluation - post-development testing

- NB: There is no mention about relating this to stakeholders, or success criteria identified in the analysis section. However, ‘success criteria’ must be considered to be able to complete the evaluation!

Area	1 mark	2 marks	3-4 marks	5 marks
Functionality testing	Some of any type of testing	Written, clear functional tests	Annotated	
Usability testing		Some of any type of testing	Annotated	

Robustness testing			-	Annotated

Testing MUST be in relation to success criteria from the initial requirements section!!

Candidates MUST mention and use 'beta testers', making sure they include "actual quoted text" of what users said. I.e. do this: John said "I like the interface because…". Don't do: John said that he liked the interface...

Area	**Explanation and indicative content**
Functionality testing	The criteria should be copied and pasted from the relevant part in the analysis. **Test No. Criteria How it was measured Comment** 1 Handles 5,000 records Tested with 5,000 records Passed - the program ran smoothly. The 'comment' only need be that. No explanation or justification is necessary. This can wait until the evaluation. **Each test number should be accompanied by an annotated screengrab.** I.e. Test number x, annotated screen grab:
Usability testing	Same as above, but for usability. Note that usability testing is likely to be measured with use of feedback from users.
Robustness testing	Same as above, but for robustness. This should definitely incorporate validation rules used. Where possible, this could include tests including - Cross-browser, cross-platform - 'Spamming' inputs with data - Overloading with huge datasets

Evaluation

Area	**1-4 marks**	**5-8 marks**	**9-12 marks**	**13-15 marks**
Success, partial success or failure, in relation to test	Comments		Evaluated	Explained test results

results - from each success criteria			Comment on how partial/unmet criteria could be addressed	
Usability features	Evidenced			Justified whether fully/partially/not met
Limitations of the solution	-	Identified	Considered	
Limitations of potential improvements and how the program could deal with them.	-	-	-	Described
Maintenance	-	-	Considered	
General structure of response	Unstructured - no clear narrative	Limited - some sub-headings	Some structure Relevant content	Clear structure - relevant sub-headings
Use of evidence (test results) to back up comments	Unclear	Limited	Some	-
Content	-	Some relevance	Mostly relevant	Consistently relevant

- **“Considered”** = state what you think and observe - backed up by your own evidence and external evidence. Contrast with contrary views that relate to your original thinking and observations.

Area	**Explanation and indicative content**
Success, partial success or failure, in relation to test results - from each success criteria *identified in the analysis*.	Evaluated: A verdict on the extent to which the project has been successful, **based on evidence ‘for’ and ‘against’**. Conclude based on what you judge to be the most important factors and justify HOW you’ve come to that conclusion. This should relate DIRECTLY to the success criteria from the analysis section, and REFER TO (not copied and pasted) test results. Candidates should avoid trivial comments such as ‘stakeholder didn’t like the colours’ etc (unless it is indeed important!) - This is often an indication of weak thought, laziness and not giving the project proper care and attention. This MUST MUST MUST be backed up by test results.
Usability features	The usability features should link to those identified from the DESIGN section
Limitations of the solution	Describe limitations in relation to... Functionality, usability, robustness

Limitations of potential improvements and how the program could deal with them.	This is tricky. If you take OCR's description verbatim...you need to… 1) Identify a limitation 2) Suggest how additional coding could get around the limitation 3) Suggest the limitation that (2) could have, and additional code that could get around the code you've written to get around the problem. What OCR probably means is… 1) For each limitation identified 2) Describe how additional code could get around the limitation 3) Describe the EXTENT to which the limitation could be addressed without having to reprogram substantial parts of the program. If the project saves data, the most likely issue is backwards compatibility with previous version(s).
Maintenance	Comment on how maintainable the project is And to be on the safe side, comment on how easy it would be to maintain the program in terms of: Corrective maintenance - is the code modular and well-commented? Adaptive maintenance - again, is it well commented and easy to adapt? Could modern hardware/interface devices be used easily? Perfective maintenance - is there scope to make any of the code run more efficiently?

Final bits and pieces

● It can be written in first person
● Avoid sloppy language, e.g. 'due to the fact that' -> because
● Candidates should not be using vague vocabulary such as word 'things', 'stuff', 'certain features' etc. This demonstrates a lack of understanding or detail. ●
Be suspicious of incorrect technical vocabulary - this can suggest malpractice*.
● Be suspicious of areas of code that are incredibly efficient and other areas that are blatantly inefficient - this can suggest malpractice*, ●
ALL work should have been proofread, if not, it demonstrates a lack of understanding of the need for quality documentation.
● Every page must have a header and footer
  ○ Name + candidate number + centre number
  ○ Page number (a table of contents is less important)

*TIP: Interview candidates about their work to see how well they understand it. Follow JCQ guidelines.

Don't forget that this is for guidance only. You should refer to all relevant documentation.

Good luck :-)