2910210 Software engineering and development

Examiner's report: Zone A

General remarks

Introduction

The paper has been designed to examine the candidates' general software engineering knowledge. In particular, software development, software process techniques, software testing, state transition diagram, Control flow graph and interface issues.

Of course, the above topics were chosen by the Examiner to assess candidates' knowledge this year and will not necessarily be the same topics featuring in next year's exam. Also, even if the same topics may be featured, the questions could address different aspects or issues. So make sure you have good knowledge of the full spectrum of topics included in the curriculum for this module.

A good understanding of the basic concepts, ideas, methods and techniques is expected from you, and the proper use of terminology and notation is particularly important.

Some of the questions are straightforward bookwork type that can be answered basically by using material from the subject guide. However, you can also use material from other software engineering educational sources to enhance and supplement your answers. This can demonstrate greater knowledge and deeper understanding, and may lead to the award of higher marks.

The following are not 100% model answers; rather they are more of indicative statements showing how to answer the questions in a better way.

Comments on specific questions

Question 1

Section a

This question requires discussion based on your knowledge of the software development cycle especially the maintenance stage. A typical answer could be:

When a civil engineer builds a bridge, maintenance means making the bridge stay in place; anti-corrosion measures, oiling of moving parts, inspection and occasional replacement of components. Crucially, the environment in which the bridge has to operate remains largely unaltered and so the parameters which guided its construction remain the same throughout its lifetime.

One of the striking aspects of the operation of software is the way in which the environment changes. Often the environment changes as a result of

the installation of the software system itself. These changes are difficult to predict at the time the system is designed, and yet the system will have to cope with these changes in operating environment.

This is perhaps the most profound reason why software is so hard to maintain. Some of the other reasons (all of which are equally valid) are listed below:

Complexity: Software is complex; its specification domain and solution domain are complex and the process by which it is constructed is complex. Humans find complexity hard to manage. It seems likely therefore that more complex systems will require more maintenance than less complex ones and this might go some way towards explaining the large proportion of effort spent on software maintenance.

Flexibility: Software is flexible. Therefore customers and many developers expect software systems to be flexible in their environment; to change, adapt and evolve to meet new requirements. When systems fail to meet these demanding goals, the software developers become embroiled in a maintenance task to regain their 'credibility' and that of their system.

Interactivity: Software is interactive. Typically (even mechanical) interactive systems require more maintenance effort than their non-interactive counterparts. Interaction with the user changes the user's perception of the system and, often, of the context in which it operates.

This creates a feedback loop which requires further changes to the system. Also, if the system interacts with other systems, then as these systems themselves evolve and are replaced, so too the system which interacts with them will need to change and adapt.

Immaturity: Undoubtedly some of the blame for the large proportionate cost of maintenance rests with the software engineering community. Software engineering is a new and demanding engineering discipline. There remains much theory and practice to be established and many techniques remain undiscovered. There is still no lingua franca. There are few text books of universal acclaim, upon which an apprentice could base their study.

Section b

A good answer to this section would start with a typical definition of static backward slicing, and proceed to give an example that will help in explaining how it can help with program comprehension.

A (static, backward) program slice, s, of a program p, is constructed with respect to a slicing criterion (V, i), where V is a set of variable identifiers and i is a program point. Statements in p which **cannot affect** the values of V when the next statement to be executed is at point z", may be **removed** from p to form s.

1	read(n) ;	1	read(n) ;
2	s:=0;	2	s:=0;
3	p:=l;	3	
4	while n>0 do	4	while n>0 do
5	begin	5	begin
6	s:=s+n;	6	s:=s+n;
7	p:=p*n;	7	
8	n:=n-l	8	n:=n-l
9	end	9	end

A Static Backward Slice w.r.t. ({s},9)

Consider the example program above. The left hand section of the figure contains the original program, while the right hand section contains a slice constructed with respect to the slicing criterion ({s},9). The slice captures the computations which may affect the value of the variable s at line number 9. More importantly it removes statements which cannot affect the value of this variable at that point.

This is useful because large programs often contain many intertwined computations and so the human may be swamped by the amount of information that has to be considered. Often, the human will only want to know about some aspect of the overall program. If this aspect can be denoted by a slicing criterion, then slicing will allow the human to avoid consideration of parts of the program which are irrelevant.

Section c

This section requires a direct answer, but make sure your example fits your explanation.

Slicing retains lines of the program which affect the slicing criterion. Therefore, if one line of the program causes an erroneous value to be computed, this line will be in the slice constructed with respect to a suitable criterion. However, if a line is omitted from the original program it will also be omitted from the slice, and so the slice will not contain the bug.

For example, suppose the program below for computing a summation, should have had an initialisation, **sum = 0**;. The program will produce the wrong value for sum, but the error would not be found in the slice on the final value of the variable sum.

```
i := 0;
while i <= 10 do
begin
read(a) ;
siim := sum + a;
i := i+1;
end
```

Question 2

All sections in this question are basically bookwork questions that require direct answers which include giving definitions and examples. Draw on knowledge from the subject guide and other resources.

Section a

This section requires an elaborate definition:

In mutation testing a program is 'corrupted' by changing a statement or a predicate in the program code. If the mutated program fails to produce the correct (i.e. specified) output for some test case then this test case is said to 'kill' the mutant.

If a test case kills a lot of mutants then it is believed to be a good test case because it is good at discovering faults. A set of mutants can therefore be used to assess how good a set of test data is. This concept is based upon the idea that the mutations we introduce reflect the kind of errors that programmers typically make and that therefore, the mutants we create represent realistic faulty programs.

100% Mutant Coverage definition: Given a set of mutants *m* and a set of test cases *t*, *t* achieves 100% mutation coverage with respect to *m* if all mutants in *m* are killed by at least one test case in *t*.

Section b

This section requires a definition with an example:

Definition of Equivalent Mutants: A mutant *m* of a program *p* is said to be equivalent to p if it produces the same output when supplied with the same input (for all possible inputs).

For example suppose the assignment statement

```
x:=x+x;
```

is replaced by the statement

```
x:=2*x;
```

in a mutation. Clearly this mutant behaves identically to the original program. If this statement in the original program is fault-free then the mutant will be fault free also (and therefore un-killable). The presence of such an un-killable mutant will preclude the possibility of achieving 100% coverage.

Section c

Small section so direct short answer with no elaboration:

A killing test case is any which satisfies $x < 0 \ \hat{} \ y = x$. A non-killing case is one which fails to satisfy $x < 0 \ \hat{} \ y = x$.)

Section d

This section requires a brief description of the problem followed by the example:

An infeasible path is one which is not executed by any possible input to the program. If all paths in the program to some statement are infeasible, then the statement is unreachable and any mutation of this statement will lead to the creation of an equivalent mutant.

For example consider the code fragment:

```
if (x>y) then
if (x<y) then y:=l;
```

The assignment to y in this example is unreachable, since the only path to it is not feasible. If the assignment is mutated then an equivalent mutant will be created.

Question 3

All sections in this question are bookwork type, so base your answers on material from your subject guide.

Section a

The answer to this section should be direct and to the point:

High coupling leads to errors because each module depends upon the others and so errors in one can lead to errors in the others to which the one is coupled. High coupling makes reuse difficult, because it makes it harder to isolate a single unit of computation. High coupling makes unit testing hard because it is difficult to be sure that the results of a computation depend solely upon a single module. Some level of coupling is required because each component of the system must pass some data to at least one other component in order for its result to be observed by the user. It must also receive data, from some other component in order for it to have a non-constant output.

Section b

In answering this section, make sure you give the required examples with the definitions of the seven levels of cohesion as the Examiner is expecting to see that:

Coincidental: loose or nonexistent relationship, e.g.,

```
print statement; update address;
```

Logical: tasks belong to a common category, e.g.,

```
print statement; print branch-report;
```

Temporal: tasks performed within a span of time, e.g.,

```
delete account; archive-transaction-log;
```

Procedural: tasks on an externally defined sequence, e.g.,

```
open account; request-deposit;
```

Communicational: operate on common data structures, e.g.,

```
update name; print date-of-birth;
```

Sequential: tasks are necessarily sequenced, e.g.,

```
open-account; send-welcome-pack;
```

Functional: all tasks are subtasks of the same function, e.g.,

```
identify-negative-balance; charge-interest;
```

Section c

A brief and to the point answer to this section should suffice, and could be on the lines of:

In both levels of coupling, some data is passed from one model, *m* to another *m*'. The modules *m* and *m*' are said to be coupled. In stamp coupling, only a subpart or component of the data is passed, whereas in data coupling the entire data structure is passed.

An example will define a compound data structure (array or record) and will show one procedure or function passing the entire record to another (data coupling) and another function or procedure passing only one field or index of the record or array to the other.

Question 4

Section a

A work Breakdown Schedule which could have resulted in the construction of the PERT chart given in the question, is depicted below:

Task	Duration (in days)	Depends upon
T1	10	
T2	10	T1
T3	5	T2, T6
T4	30	T2, T1
T5	5	T3, T2
T6	5	T1
T7	5	T6
T8	10	T7
T9	5	T8
T10	10	T9, T5, T4
T11	10	T10

Section b

To answer this section draw on the knowledge from the subject guide regarding Gantt charts.

It is particularly important to pay attention to normal start point for a task, normal end point for a task, slippage end time for a task, milestones proper placing, labelling of x and y axes with durations or task names respectively.

Section c

A basic definition will suffice as an answer to this section:

The critical path is the (a) longest path through the graph that the PERT chart denotes. (Either 'the' or 'a' is acceptable, thought strictly speaking 'a' is better.)

Section d

This section only needs a short and simple answer:

The critical path is:

T_1 , T_2 , T_4 ,T_{10} , T_{11}.

Question 5

Section a

A good way to answer this section is to give a definition and list the main stages of the spiral and the waterfall models supported by diagrams of both models.

Your answer should include a brief description of the various stages of each model. The example answer below is only indicative (typical diagrams of both models can be found in the subject guide).

The waterfall model is based upon classical engineering principles. Each stage of the model normally leads to the next, but there is a possibility for feedback, as a result of errors occurring at each stage. Each error which occurs can take the development effort back to, either the previous stage, or to some arbitrary prior stage in the development process. This is because each error might highlight some fault that occurred at some earlier stage in the process. The further back one has to travel in correction of an error, the worse (i.e. the more expensive) the fault. Therefore, the goal of software development using the waterfall method, is to attempt to ensure that there is no requirement to travel along the 'back' edges of the graph. These are indicative points pertaining to the waterfall model only.

The waterfall life cycle model has the following stages:

Analysis: this stage involves development of the software system requirements.

Design: this stage includes development of the architectural layout of the system, preparing the formats of the main data structures selecting efficient algorithms for the basic units, deciding of the software system interface, designing a formal description of the whole system in terms of a pre-selected description language, and eventually building of a system prototype.

Coding: this stage supposes implementing of the software system using a contemporary programming language chosen with respect to the system requirements.

Testing: this stage ensures that all program statements have been verified internally, and all predefined requirements are externally provided to the user.

Project maintenance: this is more like a set of additional activities that serve to support the future use of the software after it has been delivered to the client.

The spiral model is an iterative software development method that makes a virtue out of the necessity to repeat stages in the software development life-cycle. For example, it is accepted that the initial requirement specification will not capture complete and accurate user requirements, and so there may be a need to revisit this stage of the process. Crudely speaking, the spiral model can be viewed as a cyclic waterfall mode. However, in the spiral model, there is also provision for a decision point at which progress is reviewed and risk analysis is employed to decide whether or not to proceed with the development. This point is known as the 'go/no-go' decision point. The spiral model also includes, explicitly, a customer evaluation stage into the development life cycle. It is often said that the 'engineering' phase of the spiral model could correspond, roughly, to a compressed version of the conventional waterfall model. These are only indicative points concerning the spiral model.

Spiral model stages:

The planning stage: to define resources, time lines and other project related information.

The risk analysis stage: to assess both technical and management risks.

The engineering stage: to build one or more representations of the application.

The customer evaluation stage: to obtain customer feedback based on the evaluation of the software representation created during the engineering stage and implemented during the install stage.

Section b

A good answer to this section could be on the lines of:

The waterfall model is appropriate where the requirements are specified completely and precisely and where there can be no doubt about the customer's intention. The model also requires that it is possible to separate each phase of the software development process, so that there is little spillage of concern between each phase. The model assumes that the product being developed is not a novel product being developed for the first time. Rather, it is a variation upon a tried and tested theme, where the steps in the development process are well-understood and clearly delineated. (These points are not necessary, rather they are indicative.)

Section c

A good answer to this section could be on the lines of:

The spiral model is most ideally suited to a situation in which the requirements are not well understood. This might occur because the developers do not understand the user(s) or (more typically) because the user(s) are unsure about what, precisely, is required. The model also assumes that it is possible to opt out of the project, should the risk analysis indicate that the project has become unattractive, this assumes that it is possible to do so. For some projects, particularly those where

the developers are part of an in-house IT team, the option to opt out of the project is unavailable. The model assumes that the product will be developed through a series of revisions and that the product that the customer first sees will not be the final product. (These points are not necessary, rather they are indicative.)

Question 6

Section a

In answering this question, make sure to give no more than one example (each of an advantage and a disadvantage) after the brief description.

In formal proof the specification is formally defined and the system is proved to be correct with respect to the specification. The proof is typically constructed in terms of pre- and post-conditions. For each component, the proof demonstrates that the post-condition is satisfied if the pre-condition is satisfied.

Examples of acceptable answers for the disadvantage include the cost of employing and training staff in mathematical techniques, the difficulty of obtaining a completely formal specification, the difficulty in ensuring that the proof is valid and the difficulty in formally specifying non-functional requirements in a manner which admits formal proof. The principal advantage is that the proof, if correct, establishes the correctness of the system for all possible inputs, whereas the principal alternative (testing) can only prove the system's incorrectness (Dijkstra's observation on testing).

Section b

This section requires a definition plus a very brief description of its use:

The Pareto Principle dictates that 80% of the faults will result from 20% of the causes of faults. It is important because it allows effort in fault reduction to be targeted on those causes which are most likely to yield a benefit.

Section c

Simple definitions of both test types will suffice as an answer:

In black box testing, the test cases are constructed with respect to the specification and do not rely upon any knowledge of the structure of the code. In white box testing the test cases are constructed with knowledge of the code structure in an attempt to cover some syntactically defined aspect of the code, for example all statements or all branches.

Section d

Again, bookwork definitions from the subject guide will satisfy this section:

A fault is part of the software which, if executed, will cause the system to deviate from its specification. An error is the situation in which the observed behaviour of the system, for some input, is not what the user desired. A system may contain code which is incorrect (contains a fault), but which is never executed and therefore does not lead to an error. A system may conform to its specification for all inputs, yet, due to faults in the specification, it may exhibit behaviour which the user finds unacceptable.

Section e

A good answer to this question could be:

An argument which supports the conjecture argues that formal proof of all

aspects of a system is impossible and that a sufficiently large system, being a human artefact, can never be fully verified.

The argument which refutes the conjecture claims that a software system is a purely logical artefact, which can be specified mathematically and proved to be correct and. therefore, can be constructed with zero faults.

A perfect answer would demonstrate an appreciation of the pivotal issue of whether there is such an activity as 'software engineering' or whether engineering is an inappropriate paradigm in which to view software construction. This could be re-phrased as 'whether there is such an activity as 'formal mathematical program construction', or whether mathematics (and in particular, proof) is an inappropriate paradigm in which to view software construction.'