Chapter 9. Software Testing

Table of Contents

Objectives 1
Introduction to software testing 1
The testers 2
  The developers 2
  An independent testing team 2
  The customer 2
Principles of software testing 3
  The completion of software testing 3
  Writing testable software 3
  Test cases and test case design 4
Testing strategies 4
  Unit testing 4
  Integration testing 5
  Validation testing 5
  System testing 6
Testing advice 6
  Flow graphs, cyclomatic complexity and white-box testing 6
  Black-box testing 10
  Object-oriented testing 11
Debugging 11
  Brute force debugging 11
  Backtracking 11
  Cause elimination 11
  Bisect 12
Review 12
  Questions 12
  Answers 13

Objectives

At the end of this chapter you have gained an understanding of:

- What software bugs are.
- Who tests the software.
- How to write testable software.
- Various testing strategies, including unit testing and regression testing.
- Debugging.

Introduction to software testing

Software never runs as we want it to: creating software is an attempt at formally specifying (using a programming language) a solution to a problem. Assuming that it implements the correct solution (as defined by the by the various requirements and design documentation), it very often implements this solution incorrectly. Even if it does implement the solution correctly, the solution may itself not be what the customer wants, or may have unforeseen consequences in the software's behaviour. In all of these situations the software may produce unexpected and unintended behaviour. The error which causes this unexpected and unintended behaviour is called a **software bug**.

Testing is the planned process of running software with the *intent* of discovering errors in the software. It is important to realise that testing is a purposeful process, and is not the accidental discovery of software bugs.

Discovering errors in the software implementation itself is an important aspect to software testing, but it is not the only aspect. Testing can also be used to discover how well the software conforms to the software requirements, including performance requirements, usability requirements, and so on. Understanding how to test software in a methodical manner is a fundamental skill required in engineering software of acceptable quality.

This chapter considers various aspects of software testing.

The testers

Many different people involved with the development of a particular piece of software can be involved in its testing, such as the *developers*, a team of *independent testers*, and even the *customers* themselves.

The developers

Testing will always begin with the developers. As they develop individual modules of the software they will need to determine that what they have developed conforms with the requirements, and that it functions as expected. Further, the developers will have to test that these modules still function without error when they have been integrated with each other.

However, the developers will often test the software far more gently than is needed for proper testing, since the developers may already have pre-conceived expectations concerning how the software they have written behaves. Even worse, the developers may even have a vested interest in showing that the work they have done performs correctly and meets the requirements; this works directly against the process of software testing.

An independent testing team

After the developer has produced working code, the code may be passed to an independent testing team. This is a group of developers who are *not* responsible for the original development of the software. This group's sole responsibility is to test the software that they have been given.

Independent testing teams are one solution to the problems that arise from having the original programmers also test all aspects of the software. Since the testing team did not develop the software, they will hopefully have less interest in reporting that the software functions better than it does. Also, teams dedicated to testing software can use more specialised testing frameworks and methodologies than the developers themselves. The original programmers may not be interested in testing the usability of the software, or may not have the resources to examine the performance of the software; the testing team, however, should have these resources.

The presence of testing teams does not mean that the original developers are not involved with the testing at all. The will still test their code as they develop it, and will be involved with the testing team as the team examines the software and reports on any errors in the software which they locate. The developer is usually the person responsible for correcting these errors.

The customer

During iterative and evolutionary development (and agile processes in general), software is frequently given to the customer for them to examine, report back on, and potentially use.

When the customer is closely involved with the software development, it is possible to have the customer perform limited testing of the software themselves. This is known as *beta testing*.

While the customer will not usually be able to test the software as thoroughly as the original software engineers, they will be able to examine how well the software meets their needs, whether the software requirements have to be changed, and whether there are any obvious bugs which the developers have missed.

Using beta testing is *not* a substitute for testing by the developers.

Principles of software testing

The completion of software testing

Software testing never completes. It is an ongoing process that begins at the project's inception and continues until the project is no longer supported. During the software's lifetime the burden of testing slowly shifts: from the developers during design and programming, to independent testing teams, and finally to the customers. Every action that the customer performs with the software can be considered a test of the software itself.

Unfortunately, time and money constraints often intervene: it may not be worth the developer's time or money to fix particular software errors. This trade-off between resources spent vs potential benefit can easily occur for small errors, and for errors which are not often encountered by the software's users.

It is also possible (although difficult) to be statistically rigorous when discussing software errors. For instance, it is possible to develop a statistical model of the number of expected software failures with respect to execution time. Error rates over a given period can then be specified given a particular probability. When that probability is low enough, testing could be considered “complete”.

Writing testable software

An important aspect of testing is to ensure that the written software is written in a way that it can easily be tested. As the software becomes more difficult to test, so the software will be tested less often.

There are a number of guidelines that software engineers can follow in order to write software that can be easily tested. The design principles mentioned in ???, are a good place to start. In addition to this, here are some further guidelines:

- **Operability**: This is partly a self-fulfilling quality: the fewer bugs that a software system has, the easier the software will be to test, since the testing will not progress erratically while time is taken to repair bugs. Clearly, the more care is taken during software development to produce bug-free code, the easier the testing that follows will be.

- **Observability**: Software tests examine the outputs produced by the software for particular inputs. This means that software is easier to test if the inputs produce distinct, predictable outputs. Further, it can be helpful to be able to examine the software's internal state, especially where there may be no predictable outputs, and especially when an error has been discovered.

- **Controllability**: As just mentioned, software testing involves examining outputs for given inputs. This means that the more easily we can provide inputs to the software, the more easily we can test the software. This implies that software is more testable when the tester has the ability to easily control the software in order to provide the test inputs. This controllability applies to the tests as well: tests should be easily specifiable, automated, and reproducible.

- **Decomposability**: When software can be decomposed into independent modules, these modules can be tested individually. When an error occurs in an individual module, the error is less likely to require changes to be made in other modules, or for the tester to even examine multiple modules.

- **Simplicity**: Clearly, the simpler the software, the fewer errors it will have, and the easier it will be to test. There are three forms of simplicity that software can demonstrate: **functional simplicity**, in which the software does no more than is needed of it; **structural simplicity**, in which the software is decomposed into small, simple units; and **code simplicity**, in which the coding standards used be the software team allows for the easy understanding of the code.

- **Stability**: If changes need to be made to the software, then testing becomes easier if these changes are always contained within independent modules (via, for instance, decomposability), meaning that the code that needs to be tested remains small.

- **Understandability**: Clearly, the more the testers understand the software, the easier it is to test. Much of this relates to good software design, but also to the engineering culture of the developers: communication between designers, developers and testers whenever changes occur in the software is important, as is the ability for the testers and developers to easily access good technical documentation related to the software (such as APIs for the libraries being used and the software itself).

Test cases and test case design

Test cases are controlled tests of a particular aspect of the software. The objective of a test case is to uncover a particular error. Testing software, then, is the development of a suite of such test cases, and then their application to the software.

Tests should be kept simple, testing for specific errors (or specific classes of errors) rather than testing whole branches of the software's functionality at a time. In this way, if a test fails the failure will point to a particular area of the software that is at fault.

Testing strategies

Just as it is important to develop software in a way that eases software testing, it is also important to both design tests well, and to have a good strategy as to how the software should be tested.

Testing is incremental: it begins by testing small, self-contained units and progresses to testing these units as they interact with each other. Ultimately, the software as a whole — with all units integrated — is tested.

Testing can be broken down into the following stages: **unit testing** (testing individual modules), **integration testing** (testing modules as they are brought together), **validation testing** (testing to see if the software meets its requirements), and **system testing** (testing to see how well the software integrates with the various systems and processes used by the customer).

Unit testing

Unit testing is concerned with testing the smallest modules of the software. For each module, the module's interface is examined to ensure that information properly flows to and from the module. The data structures which are internal to the module should be examined to ensure that they remain in a consistent state as the module is used. The *independent paths* (see the section called “ Flow graphs, cyclomatic complexity and white-box testing ” in this chapter) through the module should each be tested. Boundary conditions should also be closely examined to ensure that they are properly handled (such as, for example, not reading past the end of an array). Importantly, remember to test the error handling code and ensure that they handle and report errors correctly.

Unit tests can easily be incorporated into the development process itself: unit tests can be written while each module is written. Indeed, some software design methods (such as extreme programming, see ???) ask for unit tests to be developed before the software itself. Regression tests (a form of integration testing), too, can be incorporated into the development process — one way of doing so is to have them automatically run each night, after all code developed that day has been submitted to a central repository.

Integration testing

Once unit testing is complete, the next important step is to combine the separate modules and ensure that they function correctly together. One may think that because each module functions correctly by itself that they will function correctly together. This is not always the case. For example, module *A* might have certain expectations about the behaviour of module *B* that are not being met, causing *A* to function incorrectly.

It may seem that integration testing can be carried out by combining all modules at once and testing how they function. Unfortunately, this “big bang” approach can make it difficult to track an error down to any one particular module. A better approach is to combine modules together incrementally, testing their behaviour at every step. Each increment brings us closer to having the complete software, but each increment remains constrained enough for us to properly test.

There are two general methods for performing module integration: the *top-down* and *bottom-up* approaches. **Top-down integration** testing begins by creating the overall software where much of its functionality is provided by empty stub modules. These modules perform no function other than to allow the software to compile, and to provide a way to exercise the modules which are being tested. The stubs are then replaced with the actual modules, one at a time, beginning with the modules involved with user-interaction and ending with the modules which perform the software's functionality. As each module passes its tests, a new module is integrated.

Clearly, top-down integration is difficult because of the need to create stub modules. The proper testing of some modules may rely upon a wide range of behaviour from the sub-modules, and so either the testing must be delayed until the stubs have been replaced, or the stubs must offer much functionality themselves (and may themselves be buggy).

The alternative to the top-down approach is **bottom-up integration** testing: here, modules which do not rely on other modules are combined together to create the software. Because we begin with modules that do not rely on other modules, no stub code is needed at all. At each step we test the combined software. When all tests have passed we add another module together. Ultimately, all the modules will be combined into the functioning software.

Whether a top-down approach, a bottom-up approach, or a mixture of both approaches, is used, whenever a new module is added to the software in order to be tested, the *whole* software has changed. Modules which rely on the new modules may behave differently, and so once again *all*the modules have to be tested. **Regression testing** is the testing of the previously tested modules when a new module is added. Regression testing should occur at every incremental step of the integration testing process.

Validation testing

Unit and integration testing asks the question, “Are we developing the software correctly?” Validation testing, on the other hand, asks, “Are we developing the correct software?” In other words, validation testing is concerned with whether the software meets its requirements.

Validation testing focuses on the user-visible portions of the software, such as the user-visible inputs and outputs, and the software's actions. The tests examine these user-visible portions to ensure that they meet the software requirements. While not part of the software itself, the documentation should also be examined to ensure that they meet any requirements concerning them.

We have previously mentioned beta testing as the process of the customers themselves testing the software. This can be a useful tool in the validation testing process, since the developers cannot foresee exactly how the customers may use the software.

System testing

Software is always employed within some larger context, such as all the systems and processes which a business customer may have in place. **System testing** is concerned with how the software behaves as it integrates into the broader system in which it will be used.

For example, when the software fails or suffers from an error it must not cause the whole system that is using it to fail. **Recovery testing** examines how the software behaves when it fails. **Security testing** examines how well the software protects sensitive information and functionality from unauthorised access. **Stress testing** examines how the software functions under an abnormal load. While software may perform well by itself, its behaviour can be quite different when its used in a larger setting; **performance testing** examines the software's performance within the context of the system as a whole.

Testing advice

While the previous sections have mostly given advice and guidelines on designing the overall testing strategy, in this section we discuss more concrete advice on creating individual tests, with a focus on testing for implementation bugs (i.e., unit and integration testing) rather than validation and system testing.

An initial distinction to be made when creating a test is the difference between *white-box* and *black-box* testing. **Black-box testing** treats the module as an object whose inner-workings are unknowable, except for how the module handles its inputs and outputs. Black-box testing does not examine the module's inner state, and assumes that if the module correctly handles its inputs and outputs, then it is error free. **White-box testing**, on the other hand, also examines the module's inner state in an attempt to ensure that its internal operations are correctly performed, no matter how the module handles its inputs and outputs.

White-box testing allows us to conceivably test every line of code in the module: since we are examining the software's internal state, we can determine where and how that state changes, and so construct tests to exercise not only every line of code, but every logical choice that can be made when executing the code. This process of testing all lines of code and all logical choices in a software module is called **exhaustive testing**, and it is extremely important to realise that, except in the most trivial of cases, exhaustive testing is impractical to perform. To see this, one need only consider that whenever the software contains, for example, **if** statements within **if** statements, or loops within loops, the number of logical paths through the software increases exponentially, and so does the time required to test each of these choices. Even software of only a few hundred lines of code can quickly require more time than is feasible to test every logical decision that could possibly be made by the software. How this exponential explosion of choice is to be handled is an important aspect of white-box testing.

Flow graphs, cyclomatic complexity and white-box testing

It will be useful to introduce some simple graphical notation for representing the execution flow of a program. While testing can (and often is) discussed without mention of flow graphs, they do provide a graphical tool for better describing the various testing processes.

Figure 9.1, “Flow graph notation”, displays various examples from the notation. The nodes in the graph represent some unit of processing that must occur. All logical decisions which affect the flow of program execution are represented by the edges in a graph: when an **if** statement decides on which branch to take (its **then** or **else** branches), this is represented by multiple edges leading to separate nodes. Similarly, various loops, case statements, and so on, are represented in a similar way.

An **independent path** through a program is a path through a flow graph which covers at least one edge that no other path covers. Examine Figure 9.2, “An example flow-graph”. The paths 1,2,3,5,7,8 and 1,2,3,5,7,5,7,5,7,8 are not independent, because neither of them have an edge which the other does not. However, the paths 1,2,3,5,7,8 and 1,2,4,6,8 are independent.

The set of all independent paths through a flow graph make up the **basis set** for the flow graph. If our testing can execute every path in the basis set, then we know that we have executed every statement in the program at least once, and that we have tested every condition as well.

Note

While we may have tested every line of code, and every condition, we have still not tested every possible combination of logical choices that could be made by the conditional statements during software execution. In this way we limit the explosion of test cases the exhaustive testing would produce. However, this does mean that *some* bugs may still escape detection.

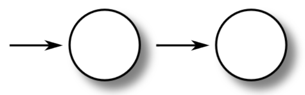
The **cyclomatic complexity** of a flow graph informs us how many paths there are in the graph's basis set (in other words, how many independent paths there are needing to be tested). There are three ways in which it can be computed:

- By counting the number of regions in the flow graph.
- If E is the number of edges in the flow graph, and N the number of nodes, then the cyclomatic complexity is: $E - N + 2$.
- The cyclomatic complexity is also $P + 1$, where P is the number of nodes from which two or more edges exit. These are the nodes at which logical decisions controlling program flow are made.

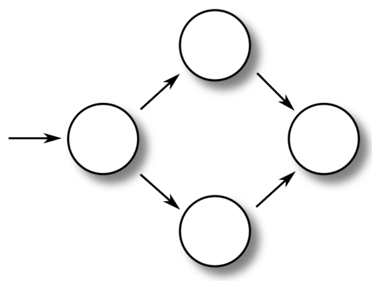
In Figure 9.2, “An example flow-graph”, we can calculate the complexity in all three ways. There are four regions in the graph (remember to count the space surrounding the graph, and not only the spaces inside the graph). There are ten edges and eight nodes. There are three nodes from which two or more edges leave. Using the three methods above, we get:

- Four regions give a cyclomatic complexity of 4.
- Ten edges and eight nodes give a cyclomatic complexity of $10 - 8 + 2 = 4$
- Three nodes with two or more exiting edges gives a cyclomatic complexity of $3 + 1 = 4$

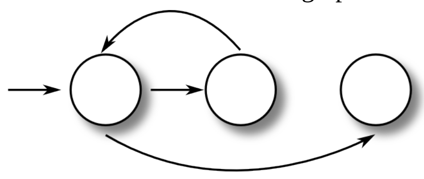
Figure 9.1. Flow graph notation

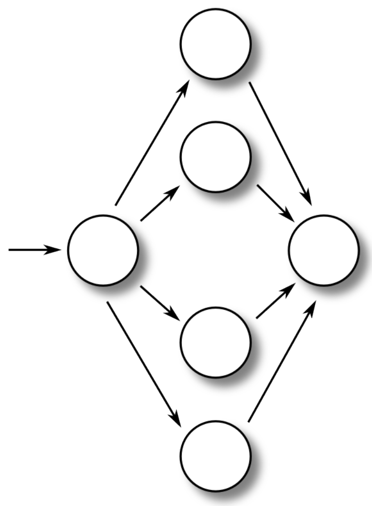


A sequence flow graph

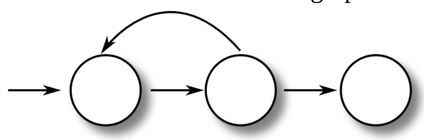


An **if** statement flow graph



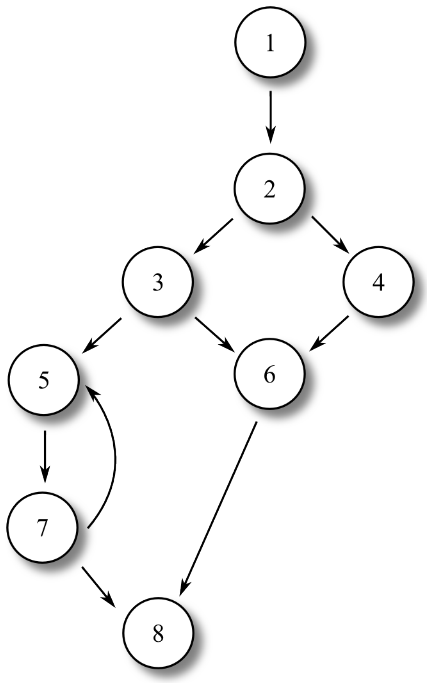


A **case** statement flow graph



An **until** statement flow graph

Figure 9.2. An example flow-graph



An example flow graph, with labeled nodes. Execution begins at node 1, and proceeds through an **if** statement, and possibly through a loop. Execution terminates at node 8.

As we can see, each method agrees with each other, and states that there are four independent paths through the program.

The third definition has broad applicability, since it provides a method for calculating complexity without using flow graphs: one can merely count the number of conditional statements in a program and add one to this number.

These concepts of independent paths, basis sets and cyclomatic complexity are important to testing, because they give us a concept of how well our tests may be exercising the code. Importantly, those portions of the code which are used least are the portions which are the least likely to be tested, and the most likely to retain their errors. Discovering the independent paths through a program, and testing them all, allows us to ensure that these errors do not go unchecked. Various studies have also shown that the higher the cyclomatic complexity of a given package, the more likely it is to have errors.

We want to again point out that testing all independent paths is not the same as exhaustive testing. Exhaustive testing wishes to test *all* possible paths through the flow graph, as determined by examining all possible combinations of logical choices that could occur at the conditions. In the particular example used here, the number of all paths through the program depends on the number of times that the loop needs to iterate. If the loop were to contain other loops, or **if** statements, this number of paths would increase dramatically. The use of independent paths keeps the number of tests to a reasonable level, while ensuring that all lines of code are tested.

The testing methodology which tests all independent paths through an application is called **basis path testing**, and is clearly a white-box testing method.

Basis path testing can be tedious to perform. It does provide suggestions for determining tests in general, however. When determining how to test code, always test the logical conditions (this is called **condition testing**). Also, focus on the conditions and validity of loop constructs (**loop testing**).

Black-box testing

While white-box testing is interested in *how* the module performs its function, black-box testing is interested only in *what* the module should be doing. Black-box testing tests the *requirements* of the software module, and not at all with how it manages to meet these requirements.

These requirements cover a wide range of areas to be tested, and includes:

- Input and output errors, which includes not only errors which may occur when making use of the software module, but also errors that may occur when the software module attempts to use other modules, such as a database.
- Incorrect and missing functions.
- Initialisation and termination errors.
- Behaviour errors.
- Performance errors.
- Reliability errors.

We will examine two methods of black-box testing: *equivalence partitioning* and *boundary value analysis*.

Equivalence partitioning

Equivalence partitioning divides a software module's input data into equivalence classes (note that these are not classes in the sense of object-oriented programming). The test cases are then designed so as to test each one of these input classes; a good test case will potentially discover errors in whole classes of input data.

Generally, for any input, we will have at least *two* classes. For example, if the input to the software is a Boolean, then this is clearly the case (in this case, each class has one value: one true, the other false). Similarly, if the input is a member of a set, then you will have multiple classes. For example, if we had a software module from a graphics package which, when given a rectangle, used the lengths of its sides to determine whether the rectangle was a square or not, then we would have two classes: one for the rectangles which are square, one for rectangles which are not.

The number of classes strongly depends on the type of the input data. If the input data requires a specific number, then there are *three* classes: one for that number, one for all numbers less than it, and one for all numbers greater than it. Similarly, if the input should be from a range of numbers, we again have three classes.

Testing each input class reveals whether the software module correctly handles the range of input that it could receive.

Boundary value analysis

Software bugs tend to occur more frequently at their “boundary values”, which are those values around which a conditional changes the value it evaluates to. For instance, boundary values are those values for which an **if** statement will change between choosing to execute its **then** or **else** portions, or where a loop decides whether to iterate or not.

This increase in errors can occur for simple reasons, such as using a greater-than comparison instead of a greater-than-or-equal-to comparison. When looping, common boundary mistakes include iterating one time too many, or one time too few.

Because of the increased frequency with which errors occur around boundary values, it is important to design test cases that properly exercise the boundaries of each conditional statement. These boundaries will occur between the various input classes in the equivalence partitioning method, and so boundary value analysis is well suited to being combined with that method.

Object-oriented testing

Testing methodologies can be modified slightly when the software is developed in an object-oriented manner.

The basic “unit” for unit testing becomes the class as a whole. This does have the consequence, however, that the various methods cannot be tested in isolation, but must be tested together. At the very least the class's constructor and destructor will always be tested with any given method.

Similarly, when performing integration testing, the class becomes the basic module which makes up the software. **Use-based** is a bottom-up integration method constructing the software from those classes which use no other, then integrating these with the classes which use them in turn, and so on. Classes can also be integrated by following program execution, integrating those classes that are required in order to respond to particular input (**thread-based** testing), or, similarly, to integrate those classes which provide some specific functionality, independent of the input to the software (**cluster** testing).

In general, you should not only test base-classes, but *all* derived classes as well. Inheritance is a special case of module integration, and should be treated as such.

Debugging

Once a test case has been executed and a bug located, debugging begins. **Debugging** is the process of locating the cause of a software error and correcting it. Unfortunately, this is not necessarily an easy process. Software engineers are only ever presented with the software's *behaviour*, and they do not directly see the error's *cause*. Debugging, then, relies on the software engineer to determine from the software's incorrect behaviour the causes of this behaviour.

Debugging begins with the software failing a test case. The software engineer debugging the software will then hypothesise a possible cause and implement the needed changes to correct this in the software. The failed test is then rerun. Further test cases may also be written to help narrow down the actual cause. This all occurs iteratively, with each step hopefully providing more information to the developer as to the root cause of the error.

A number of debugging tactics have been proposed. They can be used alone, although they become far more effective when used in combination with each other.

Brute force debugging

This is conceptually the simplest of the methods, and often the least successful. This involves the developer manually searching through stack-traces, memory-dumps, log files, and so on, for traces of the error. Extra output statements, in addition to break points, are often added to the code in order to examine what the software is doing at every step.

Backtracking

This method has the developer begin with the code that immediately produces the observable error. The developer than backtracks through the execution path, looking for the cause. Unfortunately, the number of execution paths which lead to any given point in the software can become quite large the further the cause of the bug is from where the error occurs, and so this method can become impractical.

Cause elimination

In this method, the developer develops hypotheses as to why the bug has occurred. The code can either be directly examined for the bug, or data to test the hypothesis can be constructed. This method can often result in the shortest debug times, although it does rely on the developers understanding the software well.

Bisect

Bisect is a useful method for locating bugs which are new to the software. Previous versions of the software are examined until a version which does not have the error is located. The difference between that version's source code and the next is then examined to find the bug.

Review

Questions

Review Question 1

Complete:

Testing is the [________] process of running software in with the intent of [______________] in the software. It is important to realise that testing is a [__________] process, and is not the accidental discovery of software bugs.

A discussion of this question can be found at the end of this chapter.

Review Question 2

Who are the different parties involved in software testing, and how does the testing shift from one party to another?

A discussion of this question can be found at the end of this chapter.

Review Question 3

What guidelines would you give for developing software that is easily testable?

A discussion of this question can be found at the end of this chapter.

Review Question 4

Complete:

Test cases are controlled tests of [__________________] of the software. The objective of a test case is [_____________]. Testing software, then, is the development of [___________________], and then their application to the software.

A discussion of this question can be found at the end of this chapter.

Review Question 5

What are the different stages of software testing?

A discussion of this question can be found at the end of this chapter.

Review Question 6

What is the difference between white-box and black-box testing?

A discussion of this question can be found at the end of this chapter.

Review Question 7

Why is it not practically possible to test every logical path through a piece of software? What alternatives are there?

A discussion of this question can be found at the end of this chapter.

Review Question 8

Debugging begins with the software [_________] a test case. The software engineer debugging the software will then [_____________] a possible cause and [___________________] in the software. The failed test is then rerun. This might not always provide an exact reason for the bug, and so further test cases may also be written to [_______________________]. This all occurs iteratively, with each step hopefully providing more information to the developer as to the root cause of the error.

A discussion of this question can be found at the end of this chapter.

Review Question 9

What are some common debugging techniques?

A discussion of this question can be found at the end of this chapter.

Answers

Discussion of Review Question 1

Testing is the **planned** process of running software in with the intent of **discovering errors** in the software. It is important to realise that testing is a **purposeful** process, and is not the accidental discovery of software bugs.

Discussion of Review Question 2

There are three parties involved in software testing:

1. The developers

2. Independent testers

3. The customers / users

Testing originally begins with the *developers*, who need to ensure that the software they have coded works as they intend. Testing is then taken over by *independent testers*, who are able to use more specialised testing frameworks than the developers themselves. They can also perform user-based testing, which most developers would not be able to do.

Finally, as the software is delivered to the customer and the software begins to be used, its users become the final group of testers.

Discussion of Review Question 3

- *Operability*. New code developed for relatively bug-free software will have fewer bugs than code developed for relatively buggy software.

- *Observability*. Software whose internal state can easily be examined, and that produces well defined outputs for particular inputs, is easier to test.

- *Controllability*. If the tester has the ability to easily control the software, testing becomes easier: controllability allows for the easier inputting of data and examining of output.

- *Decomposability*. Independent modules can more easily be tested, and changes made to a module are less likely to affect other modules.
- *Simplicity*. The simpler the software, the fewer errors it will have.
- *Stability*. Testing is easier if changes made to correct bugs are limited to independent modules.
- *Understandability*. The better the testers understand the software — through good software design, documentation, and so on — the better they can test the software.

Discussion of Review Question 4

Test cases are controlled tests of **particular aspects** of the software. The objective of a test case is **uncover a particular error**. Testing software, then, is the development of **a collection of test cases**, and then their application to the software.

Discussion of Review Question 5

- *Unit testing*, which tests the smallest modules of the software.
- *Integration testing*, which tests the software as the modules are brought together.
- *Validation testing*, which tests the software to ensure that it meets its requirements specification.
- *System testing*, which examines how well the software integrates with the various systems and processes used by the customer.

Discussion of Review Question 6

Black-box testing tests the module as an object whose inner-workings are unknowable, except for how it handles its inputs and outputs. Black-box testing therefor does not examine a module's inner state, only the interfaces to the module. It assumes that if it handles its inputs and outputs correctly, then the module itself behaves correctly.

White-box testing examines the software's internal state; it does not assume that because a module has handles its inputs and outputs correctly that the module has behaved correctly. For example, after correctly outputting some data, a module may leave its internal state with invalid values, and so any further action will fail.

Discussion of Review Question 7

The number of possible logical paths through a piece of software can grow exponentially as *if* statements and loops are added. This would make the amount of time required to test a large software package prohibitive.

An alternative testing strategy is to determine the software's cyclomatic complexity, which tells us how many independent paths there are through the software. Each of these independent paths can then be tested; this will provide good code coverage (it will execute every line of code) without exhaustively testing every logical path through the software, and so greatly reduces the amount of time needed to execute the software.

Discussion of Review Question 8

Debugging begins with the software **failing** a test case. The software engineer debugging the software will then **hypothesise** a possible cause and **implement the needed changes** in the software. The failed test is then rerun. This might not always provide an exact reason for the bug, and so further test cases may also be written to **narrow down the cause**. This all occurs iteratively, with each step hopefully providing more information to the developer as to the root cause of the error.

Discussion of Review Question 9

- *Brute force debugging*; the developer searches through stack-traces, memory-dumps, log files, and other data generated by the program to locate the error.
- *Backtracking*; the developer examines the code that immediately produces the observable bug, and then moves backwards through the execution path until the cause of the bug has been found.
- *Cause elimination*; the developer hypothesises reasons why the bug has occurred, and then either directly examines the code to see if these reasons exist, or produces further tests to narrow down the choice between various hypotheses.
- *Bisect*; the developer examines previous versions of the software until one without the bug is located. The difference between that version of the source code and the next (in which the bug does not exist) is where the bug will be located.