ISYS3001
Managing Software Development

Topic 1

Testing

Contents

Testing

References

Ian Somerville 2016, Software Engineering, 10th edn, Pearson Education Limited, Edinburgh Gate. ISBN: 978-1-292-09613-1. Chapter 8.

Objectives

On completion of this topic you should be able to:

1. Describe common testing procedures: unit, component and system levels
2. Select test data
3. Develop structured test plans
4. Understand test-driven software development processes
5. Describe release testing: requirements-based, scenario and performance testing
6. Describe different approaches to user testing: alpha, beta and acceptance testing

Overview

The first important aspect of testing is to understand why we test software (and other technology). The textbook identifies two main reasons to test software:

1. To test software meets its stated requirements.
2. To find where software does not meet its stated requirements or fails in some way that is not mentioned in the requirements.

These two aims overlap somewhat but reflect two different approaches. The first approach is to show correctness where software is *validated* against some requirements specification. The second is to find problems or *defects* in the software, which may be outside the requirements specification.

To this point in your course you have probably only tested to the above point 1. You test your software as to whether it meets some assignment specification. Your markers however are trying to do the second type of testing, which is seeing how to break your software. In industry validation and defect testing are not only applied by a programmer to her or his code. Users and specialist testers can be involved in testing. Also, real software products are usually too large to be developed in-house so purchased or free components are incorporated to form the system. These components must also be tested for both validation and also defects, particularly when they interact with other products already in use.

Another important observation about testing is that from Dijkstra, et al. (1972):

> *Testing can only show the presence of errors, not their absence.*

In other words, for nearly all software it is impossible for you to prove the absence of bugs by testing. You can however find bugs that must be fixed. This means that there is human input into testing regimes to make them as good as required before accepting a software product as good enough.

It is also important to realise that testing is just one of the processes that contribute to making sure that the software is doing what the user expects it to do. What the user 'expects it to do' introduces two new terms which relate to this (Boehm, 1979):

1. "Validation: Are we building the right product?"
2. "Verification: Are we building the product right?"

Validation is about whether the product is the right one for the users. It is far more that testing or specifications, it also about finding what the user really requires. Verification is the easier process of making sure the product conforms to the given specifications. The gap between the specifications and what the user really requires is a problem that validation addresses.

A third important aspect of testing is that we do not actually need to have running software to do testing. Specifications may be tested by processes such as software inspection and reviews before the software is implemented. You may have seen these techniques in other units and we will briefly discuss these in other topics.

In the remainder of this topic we will look at three categories of testing:

1. Development testing – test done during software development
2. Release testing – an independent test of a complete systems before release
3. User testing – the end users test the system before formal acceptance or as a feedback mechanism.

Note that real systems development usually involves combinations of these types of testing. We can also automate these testing types to varying degrees as we will see below.

Activity 1-1: Testing Overview

Read the overview of testing in the textbook pp. 227-232 (including introduction to section 8.1).

1. Explain in your own words the difference between *validation testing* and *defect testing*.
2. Explain in your own words the difference between *validation* and *verification*.
3. Give examples where "fit for purpose" may require extreme testing or lesser testing. How do you decide how much testing is necessary?

Next we look at unit testing.

Unit Tests

A unit test refers to testing a suitable piece of code abstraction. In object-oriented languages it is frequently a *class* that is the subject of a unit test. However many people enrolled in this unit may not be familiar with classes so we will use a function call (method or procedure) as a unit for unit tests in this topic. Not all languages are object oriented so other types of 'unit' may also be used. Sometime it will be a collection of functions or perhaps a source code file with functions in it. In any case the unit test is the test of a re-usable code segment that is designed to be used in a larger system.

We can divide unit tests into three parts:

1. Initialise the test with data to be used and any other initialisation, e.g. create objects, open files, etc.
2. Invoke the code segment, e.g. method/function call
3. Check the results, this is known as assertion testing.

The setup part could be fairly elaborate if the code section requires a complicated state before it is invoked. For code that interfaces with complex systems, an environment may have to be simulated to do the unit test. This is not usually the case because simple variables and static versions of dynamic data (dummy data) can often be used in the testing process.

The 'assertion' is a simple Boolean check. If the assertion is true, then the test passes but if the assertion is false the test fails. If the assertion fails then there is a bug but the test does not give any more

information other than the data that caused the failure. The testing process has detected a bug but it does not attempt to fix it.

Activity 1-2: Unit tests

Read the introduction to Section 8.1 and Section 8.1.1 of the textbook pp. 232-234.

1. Consider the following Java function call:

```
// count the number of given chars in a string
public int countChar(String str, char c) …
```

   What setup, calls and assertions would be useful here (we will look specifically at how to select data in the next section)

2. What setup, calls and assertions would be useful for:

```
// square a floating point number
public double square(double n) …
```

3. What setup, calls and assertions would be useful for:

```
// search a file for a string and return occurances
public int searchText(File f, String pattn) …
```

Next we look more systematic selection of test data.

Choosing Unit Tests

Hopefully the example in the previous section made you think about which tests to do with your unit tests. Each of the examples had an extremely large number of possible input sequences that could be chosen for the test. So a strategy needs to be adopted to make the tests most useful.

The strategy in selecting tests should attempt two things:

1. Test all of the things the unit should do.
2. Apply tests that may result in bugs being revealed.

This is mainly an art resulting from theory and experience but there are a few ways to approach this. The textbook suggests two approaches:

1. Partition testing – here the tests are divided into groups of input which should be processed in similar ways.
2. Guideline based testing – these reflect the kinds of errors programmer usually make.

Partition testing is an attempt to pick out the different types of processing that may occur inside a unit. This is very reliant on the function of the unit. As an example, a numerical unit (e.g. the `Square()` function in the previous activity) may have different processing for positive and negative numbers which allows us to have two groups (partitions) of input data. Another form of partition is a set of values that should produce the same error result. Note that we will also see the word "partition" used later when describing how to split a large testing task into smaller more manageable tasks.

The guideline based testing takes into account the behaviour of programmers. For example, it is a good idea to test data on the boundary of a partition. For example, the number zero appears on the boundary of positive and negative numbers so the programmer may have made a mistake here. When a range of input values is allowable it is a good idea to test the boundary and we know programmers make mistakes like using '<=' operators instead of '<', or perhaps not quite getting the limit in a `for` statement correct.

Another consideration is whether you can inspect the code to see what to test. If you can inspect the code you can make decisions on partitions and partition boundaries by identifying loops, comparison expressions, if-then statements, etc. When the code is not available you may have to make decisions based on the documentation, available specification or simply on intuitions due to your own experience. There are two specific terms for these two testing regimes:

1. White-box testing – you inspect the actual code to see what to test. This can be very detailed, for example test every branch of code in the unit (called path testing).
2. Black-box testing – you inspect the code interface of specification to see what should work or should not work.

There are systematic ways to choose test data suggested in the textbook reading below. For example, you can chose to test all error messages, test all buffer overflow (text fields, input messages, etc.), use very large numbers where numeric input requested, etc. While some of these can be listed (and are) many ideas will come from experience.

Activity 1-3: Unit test data

Read the Section 8.1.2 of the textbook pp. 234-237.

1. Consider the following Java function call:

```
// count the number of a specific character
public int countChar(String str, char c) …
```

   What data should we call here (black box obviously).

2. What data should be use here?

```
// square a floating point number
public double square(double n) …
```

3. What data should we use here?

```
// search a file for a string and return occurances
public int searchText(File f, String pattn) …
```

Because of the nature of this unit, that many students have not done much programming, we will not be doing practical white-box testing. However, black box testing is an important activity that all IT students should be familiar.

Next we look some specific software for assisting with unit tests.

Unit test software

There are many software products that can be used to assist with unit tests. The usefulness of unit testing comes when they can be repeated automatically. There is a system known as xUnit that has spawned many specific systems, e.g. JUnit for Java systems and mUnit for visual studio based languages. This system uses the three unit test phases above to implement a test harness in the language of the unit. For example, two Java a unit test can be coded as follows:

```
@Test
public void testCountChar() {
    int expected = 3;
    int result = week1.Week1.countChar("this has three spaces", ' ');
    assertEquals(expected, result);
}

@Test
public void testCountCharNulls() {
    int expected = 0;
    String testString = null;
    int result = week1.Week1.countChar(testString, ' ');
    assertEquals(expected, result);
}
```

These are two unit tests in a NetBeans project called `Week1` which is applied to the following method declared in the `Week1` project:

```
//
//  countChar() - count the number of specified character in a string
//
public static int countChar(String str, char ch) {
    int counter = 0;
    if (str != null) {
        for (int i=0; i<str.length(); i++)
            if (ch == str.charAt(i))
                counter++;
        return counter;
    }
    return 0;
}
```

This is a simple method for counting the number of a specified character in a string. Notice how the setup phase in the unit tests gets the parameters of the unit organized. The function call to perform test phase calls the method using the package name and class name of the main program (`week1.`Week1). The result phase, where we decide if the text passed or fails, uses what's known as an *assertion* which returns true or false for the test status. We will work through a Netbeans unit test example in a laboratory session.

Activity 1-4 – unit test example (Java)

Either wait for the online unit example or view it online now.

Next we look at the higher level testing at the component level.

Component tests

Components are a higher level of software abstraction than the 'unit' described in the previous section. A component is usually a collection of units or a collection of other components. They do however have a programming language like interface.

The textbook identifies four types of interfaces:

1. Parameter interfaces. These are the method/function calls you will be familiar with in your prerequisite programming.
2. Shared memory interfaces. These are mainly used by embedded hardware oriented systems. Here a section of memory is used to pass data between separate components.
3. Procedural interfaces. These are components where a set of procedures are used for the interface. For those who have studied object-oriented programming these correspond to a class.
4. Message based interface. These are components interfaces that rely on message passing. Client/server systems (see later topic) and some object-oriented systems have these interfaces. Note that messages can be both to or from a component.

Each of these interfaces are a point where we can do testing. The interfaces are a common source of errors in complex systems due to three main causes:

1. Interface misuse. This can occur when parameters of the wrong type, the wrong order or the wrong number of parameters are used. The components may not have the nice type checking that you are used to in your programming units, and some components may have no type checking at all.
2. Interface misunderstanding. Errors are made in understanding how to use an interface. This may be from documentation misunderstanding or just failing to read the documentation, if it exists.
3. Timing errors. In hardware oriented systems that use shared memory there is considerable effort in making sure data is read from shared memory correctly. One component must be sure another component is not half way through updating the data before it reads it. There are also timing problems with message based systems where the component and component user may operate at

different speeds so errors like a fast component responding before the user component is ready can occur.

Testing for these errors requires trying correct and incorrect values to the interface. However, many possible errors may not become apparent while testing. For example, timing errors may require a specific sequence of messages or some external factor that effects the running of the component to produce a problem. This may be very difficult to simulate under testing conditions. The textbook also notes that, because a component may be composed of other components, failure of an internal component may not be obvious because the interface may appear to return a sensible result.

The textbook suggests the following guidelines for testing components:

1. Examine the code to identify calls to external components and test the extreme limits of the parameters.
2. Where pointers are used as parameters always test the null pointer. Object oriented programmers will be familiar with this problem.
3. For procedural interfaces, deliberately cause the component to fail to check results (error messages or other status indicators). This is a common source of interface misunderstanding.
4. For message passing systems, use stress testing by generating more messages than will occur after deployment. This will reveal timing issues.
5. For shared memory interfaces, change the order that the components sharing the data are activated to reveal timing and ordering misunderstandings.

The textbook actually suggests code inspections and reviews are the best way to test interfaces. However for black-box testing, which this unit is mainly interested in due to prerequisite arrangements that will not be possible.

Activity 1-5: Component testing

Read the Section 8.1.3 of the textbook pp. 237-239.

1. In the textbook, a procedural interface is really a collection of parameter interfaces. What additional testing will be necessary for the procedural interface testing (in addition to parameter testing)?
2. A web browser interacts with a web server using a message based interface. Can you think of how the web server component could reveal timing issues?
3. JavaScript is a language where the type and number of parameters is not checked by the system. Why do you think JavaScript components are frequently misused by a programmer without the programmer realizing there is a problem?
4. Using the browser/web server example again, how do you think you can stress test the webserver component?

Next we look at system testing, which is a higher level abstraction.

System Testing

System testing is where all of a system's components are integrated so that we can test the whole system, detecting errors that result from component interactions. Components that form the system may be:

1. Purchase from other sources or public domain, and may not have source code implying black-box testing.
2. May be developed independently by different programmers or different groups in the organisation and these groups may have had limited communication during development.

Besides detecting errors due to unexpected component interactions, systems testing also reveals any misunderstanding between component developers.

Systems testing should concentrate on testing the interfaces between components. The component and unit testing discussed earlier can reveal problems at the lower level and should be done separately by the developers concerned. However, system testing brings together all the individual parts and will involve all developers in resolution of problems found by testing.

Normally, system testing is a use-case based testing strategy. By applying use-cases to the system, we can exercise all of the components in a way that the end system will be used and so reveal likely problems before they arise. However, you should keep in mind that, except for the most elementary systems, there are an almost infinite number of possible detailed use-cases so a testing strategy will not be able to test all possibilities.

Another issue with system testing is whether it should be automated or should real users test the system. Automated test is difficult to do with many systems but has been successfully deployed as we will see later in this unit. Using real users is an alternate strategy but the person cost is a factor in determining the extent of testing that can be used. There is also the option of automating some test and using real users together as available.

Activity 1-6: System testing

Read the Section 8.1.4 of the textbook pp. 240-243.

1. When should system testing occur?
2. Why is "use-case based testing" a good idea for system testing?
3. Why is automated testing difficult for some systems?

Next we look at how testing can be used as a focus in the software development process.

Test-Driven Development

Test-driven development is a development process that implements a test before implementing the code. The idea is that the test forms a specification of what the code should achieve when it is implemented and it also can be used to prove the code works after implementation.

In the textbook reading the test-driven development process is one based on incremental development process. It has the following five steps:

1. Identify the increment in functionality that you wish to implement next.
2. Develop an automatic test for this increment. Note that no coding has been done.
3. Run the test with all the tests for previous increments. Note that the test will fail because there is still no code implemented.
4. Implement the incremental code and re-run the test. This process is repeated until all tests run successfully.
5. Add the test to all the previous tests and move onto the next increment.

The expected failure of the test at step three is a good idea because it proves the test is active in the test collection. The successful running of all previously developed tests ensures the new code increment does not break any of the previous code added during increments.

Usually the automatic tests are implemented using unit tests in the environments such as jUnit or mUnit we discussed earlier in this Topic.

The textbook identifies the following advantages of test-driven development over the traditional test-last strategy:

1. Code coverage. Because there is an automatic test for every code increment from the very beginning, you can be confident that all code is tested at each increment.
2. Regression testing. You run all the older tests which worked before a change is made, this finds exactly where a new bug has been introduced.

3. Simplified debugging. When the latest test fails it is easy to identify where you have changed code in the latest increment. Leaving testing until later means there is more code to examine.
4. System documentation. The tests are a form of documentation that describe the correct (or incorrect) behaviour of the code.

Naturally, test-driven development applies to systems where new code is being developed. Since most systems re-use other components the idea is not universally applicable. However, if some components are being developed to interface with re-usable components then the incremental testing process can be applied to those. The re-usable components will need their own testing strategies but, once developed as automated tests, those tests can be added to the incremental test database.

We will look at test-driven development processes in more detail when we discuss agile development techniques later in this unit. However, test-driven development is not restricted to agile development processes and is used on traditional plan-based processes as well.

Activity 1-7: Test driven development

Read the Section 8.2 of the textbook pp. 242-245.

1. Explain the logic of running a test before the code it is meant to test is implemented.
2. Explain *regression testing*.
3. Why does the practice of accumulating tests make it easier to debug the software?
4. What is the advantage of automated testing in test-driven development?

Next we look at release testing.

Release testing

Release testing is required before a system is handed over to people besides the developers. The other people could be end-users or customers of the developers. In large projects they may be other development groups who will use the component being released to them. You can consider release testing the final stage of developer testing before the product is considered good enough to hand over to others.

Release testing has two features that unit, component or system testing did not have:

1. The developers should not do release testing
2. The software is tested against user requirements

It follows from these features that release testing is usually a black-box procedure. The release test is concerned with practical aspects that ensure the software is ready for use by real users. This does not mean the process is under user control, the developers still should design the release testing to ensure they are happy the software is ready for release.

The textbook identifies three aspects to release testing requirement-based testing, scenario testing and performance testing that we will look at separately.

Requirements-based testing

Requirement gathering is the first stage in most development processes you have seen in your course. In your System Analysis unit, you have seen many processes for gathering requirements from users. The textbook uses a term which may be unfamiliar to you “Requirements Engineering”. This is just the computer science term for what you have studied previous. It is meant to instill an idea that requirements gathering and analysis should be taken seriously and systematically using well understood methods.

A good requirements specification will be easy to verify against the end software. Requirements-based testing is the task of taking the system requirements and testing that these are in fact achieved by the

system. This, like all testing, is not a straight forward process. Each requirement may require many tests to be created and used using a combination of creativity and experience for those involved.

To take a simple example, the following is a simple (and therefore unlikely) specification:

"The program will read two number from the console and print the product on the console"

To test this specification there are several things, and therefore several tests, that could be considered:

1. Is there a maximum number that can be entered for the first and second number (they may have different limits)?
2. Will two very large numbers enter correctly but produce an overflow in the printed product?
3. Will the system work with floating point number?
4. Are appropriate error messages printed if the user types a malformed number (or does program crash)?
5. Will negative numbers work?

These may not be the only possible vagaries in the specification that needs testing. Further, as we have mentioned previously, it is usually not possible to test all possible inputs so a strategy must be adopted by the testers to prove the system is good enough for use. There is a more complicated example in the reading below.

Activity 1-8: Requirements-based testing

Read the Section 8.3 and Section 8.3.1 of the textbook pp. 245-246.

1. Explain how release testing is under developer control but the developers should not do the testing?
2. Why is the term "good enough" used in relation to release testing?
3. The reading says requirements-based testing is "validation rather than defect testing". What should happen if a defect is found?

Next we look at scenario and performance testing during release testing.

Scenario and performance testing

A scenario is a typical sequence of user steps that a system user would do. A scenario is also called a user story, a sequence of steps that would be undertaken by a particular type of user. Scenarios are an important idea we will come back to when discussing modern development processes. They typically encompass more than one system specification idea, which reflects the modular design of system specifications. Different users will pick and choose different aspects of the system to achieve their aim.

Since a scenario is a typical user sequence, a collection of scenarios will exercise the system in a way that reflects the typical system use.

Another type of testing is performance testing. This involves putting the system under stress (stress testing) to make sure is satisfactorily handles the proposed load on the system. Most system specifications will include estimates of non-functional attributes such as performance, reliability, throughput, etc. Typically, to test these at release testing time the system is put to stress at a level that exceeds the stated requirements to check for problems. As we have mentioned before when discussing timing defects, a stress test will also uncover many other bugs that only occur when the system is under load. It is also useful to see if the system behaves as per specification (or reasonably if unspecified) when it is pushed to the point that it fails.

With stress testing is difficult to reproduce exactly the load that would be imposed after it is released to users. This is because it is difficult to exactly reproduce the environment that the user will use the live system. A typical user will use more than one software system so there will be interactions not reproduced during release testing.

Activity 1-9: Scenario and performance testing

Read the Section 8.3.2 and Section 8.3.3 of the textbook pp. 246-248.

1. Why do scenarios make good testing strategies? How many scenarios are ideal?
2. Will scenario based testing test the same requirement more than once?
3. Why would you want to stress the system until it fails?
4. What sort of systems does the reading say are particularly important to stress test? Why?

Next we look at user testing.

User testing

User testing is the testing carried out by the customer, the user that the system was intended to serve. This is different to the release testing discussed in the previous section because it is the user that is in control of the testing. User testing usually follows the release testing by the system developers, because the system is usually not made available to users until the developers are happy the release testing is complete.

User testing usually occurs before the customer pays for or accepts the product. This makes user testing an important part of any system development. The interesting part is when the user finds problems with the system and how these issues are resolved. Usually, it is not a case of "take it or leave it". Significant negotiation can occur in commercial products and in consumer products, e.g. games, the developers may have to invest significantly to resolve issues identified by the user testing.

The textbook identifies three types of user testing:

1. Alpha testing. Here a selected group of users are integrated with the release testing team, usually for early and possibly incomplete releases of the product. Typically, alpha testers identify where the requirements are deficient or have been interpreted incorrectly. As such they refine the requirements for the developers.
2. Beta testing. Here a system is made available to a large number of test users and the users are encouraged to experiment and give feedback on the problems they discover. Beta testing usually also involves monitoring of the system by the developers to detect issues or resolve bugs reported by the users.
3. Acceptance testing. This is where customers are involved in deciding whether the system is ready to be released for use.

Alpha testing is often used for stand-alone software products or mobile apps. Advanced users like to be involved so they can influence new features of the product and get advanced knowledge of the product before it is released. Alpha testing is important to agile development that we will discuss in a later topic.

Beta testing is important for software that is deployed in many different environments. Since a large group of users is involved, specific issues that cause problems in some environments can be identified early and addressed before the product is released. For example, a product that is deployed to a PC environment has to contend with many different variations in a PC build (graphics cards, Ethernet ports, CPU types, memory sizes, motherboard types, etc.), any of which may cause a problem with the software. Also for complex systems, many users will use the product in different ways so some will find issues that the developers may not have anticipated.

Acceptance testing is necessary for custom built software. It is the point where the customer decides whether the software is good enough to implement and any final payments should be made. There are six stages to acceptance testing:

1. Acceptance criteria must be defined. These are usually part of the contract for the system development. However, in practice requirements will change during system development as we discuss elsewhere in this unit. This usually means contracts will have provisions for change negotiation during the contract term.

2. Acceptance testing must be planned. This is the usual project planning for a component of the project. The planning can occur in parallel to the development but must fit in with the delivery schedule and contract terms.
3. Derive acceptance tests. This is the usual test planning we have discussed previously. They should provide coverage of all of the system requirements and cover non-functional issues such as performance.
4. Run acceptance tests. This is the actual testing of the software after it is released for testing.
5. Negotiate test results. Most acceptance testing uncovers some problems. This stage is where negotiations occur about whether the system is good enough or what must be done to make the system acceptable.
6. Accept or reject. Either the system is accepted or if it is not good enough then the system is sent back to the developers to fix and the acceptance tests will need to be run again. In the worst case the system may be abandoned.

It is important to realise that negotiations are frequently necessary to resolve issues identified during acceptance testing. It may or may not be the fault of the developer that the system does not meet the users' expectations. Also, a system may not be rejected outright, some parts may be deployed to the users and other parts delayed until they are satisfactory for the customer.

We will see how agile development processes have emerged which greatly assist this developer versus customer relationship in a later topic.

Activity 1-10: User testing

Read the Section 8.3.2 and Section 8.3.3 of the textbook pp. 246-248.

1. What is the difference between alpha and beta testing?
2. Why would users want to be involved in alpha testing of a stock market trading app?
3. Why would beta testing be used by computer game developers?
4. Why will the acceptance criteria change during the system development?
5. Explain in your own words why negotiation is usually required after acceptance tests are completed.
6. Explain why outright rejection of a system due to a failed acceptance test usually not an option (at least in the first few acceptance tests).

That's enough for testing. There is a laboratory session associated with this topic where we will plan some testing for software products. Since will be using existing products, we will take the role of release testers or acceptance testers in developing test plans.

Summary

In this topic we first looked at unit tests, component tests and system tests which represent different levels of abstraction in a software system. We then looked a test-driven development processes where tests are developed before the software is implemented to serve as both documentation and specification. We then examined release testing where the developers test whether the system is ready for release to the users. With release testing, we saw how requirements-based, scenario and performance tests could be used for release testing. Finally, we looked at user testing, which include alpha, beta and acceptance testing.

Answers to Activities

Activity 1-1: Testing overview

1. *Validation testing* is testing against some agreed specification. To do this you can work from the specification to make sure the software does what is specified.

   Defect testing can take a more abstract approach because the aim is to expose bugs in the software. This can rely on the tester's knowledge outside of the specification. For example, knowledge of how programmers commonly make mistakes will suggest test data that may break the system. Another example is knowledge of the platform on which the software runs may suggest some sources of problems that can be exposed by defect testing. This specialist knowledge may not be considered in the specification.

2. Validation as for question 1 is checking the software does as the user requires. However, testing is just one of the activities in validation. This includes functional and non-functional requirements.

   Verification is a more technical process that checks the software meets the technical specification. Validation goes beyond the technical specification as many requirements may not be stated in the specification due to a variety of reasons. For example, a technical specification may not even be readable by a user.

3. Extreme testing is required when safety (e.g. aircraft control), secure (e.g. many financial systems), or any other system where the user would be less tolerant to bugs.

   On the other hand, lesser testing may be used to get a product to market early. Users of leading software can sometimes expect bugs to appear with new or unique software applications. However, when a product matures or there are high quality competitors in the market this does not apply. There is also the expectation of users to consider. When users are used to buggy software they may not be particularly resistant to new buggy software.

   The other way to look at this is to decide how much software testing is necessary. Where the user's expectations are high, more testing may be necessary. The inverse situation also applied, where expectation is low, less testing will be required. Market pressures also affect the time available for testing. In some cases, the market release schedule may dictate the amount of time available for testing.

Activity 1-2: Unit tests

1. For the function:

   ```
   public int countChar(String str, char c) …
   ```

   Setup would be to assign values to the two parameters.

   The call would be made using selected parameter values

   The assertion would check the returned `int` value and whether the function actually returned. (for those more knowledgeable in programming the assertion may also check for exceptions).

2. For the function:

   ```
   public double square(double n)
   ```

   Setup would assign values to the single parameter.

   The call would be made using selected parameter values

   The assertion would check the returned `double` value and whether the function actually returned. (for those more knowledgeable in programming the assertion may also check for exceptions).

3. For the function:

   ```
   public int searchText(File f, String pattn)
   ```

   Setup would be to assign values to the two parameters. In this case the, first parameter may use a File object that does not correspond to an actual file.

The call would be made using selected parameter values

The assertion would check the returned `int` value and whether the function actually returned. (for those more knowledgeable in programming the assertion may also check for exceptions, e.g. for when the `File` object did not correspond to an actual file).

Activity 1-3: Unit test data

In answering each of these, you should keep in mind that the data shown is only a small number of the possible test data combination. Here we first trying normal data that we expect to work and then trying in an attempt to reveal bugs in the code.

1. For the function:

   ```
   public int countChar(String str, char c) …
   ```

   Normal data we would set up a string and char combinations to test zero, 1, 2, 3 and maybe more characters in the string.

   To find bugs we would setup the string to contain zero characters or a very long string. We would also try a null string (which is different to a zero length string).

   We should also try char values that are unusual character, e.g. the '\0' character and other unprintable characters.

2. For the function:

   ```
   public double square(double n)
   ```

   Normal values would be a few floating point numbers and check the response is correct, perhaps using a calculator. We should try positive, negative numbers and perhaps zero.

   To find bugs we should try very large floating point number as parameter. To be precise, we should look up the language specification to find the largest floating point number that can be passed as a parameter and check the returned value (which cannot be correct). We could also try a very large negative number.

3. For the function:

   ```
   public int searchText(File f, String pattn)
   ```

   Normal values would be strings from a given file that exists. Test strings that occur 0, 1 or a large number of times. The strings should be different lengths.

   To check for bugs we need to be a bit creative and test:

   - A file object that does correspond to a real file
   - A binary file
   - A file of zero length
   - A null string
   - A zero length string
   - A string that is longer than the file

Activity 1-4: Unit test examples

An online session has a recording on the MySCU site.

Activity 1-5: Component testing

1. The returned values from the procedures need to be checked. Those familiar with object oriented programming will also know that exceptions may also be thrown so these need to be checked as well. The most complex aspect is to test the effect of multiple procedure calls. For example, if one

procedure call updates some data values you will need to test that subsequent procedure calls return expected values, depending on the first call. This is very much dependent on what the component is designed to do. Non-functional aspects like performance and reliability may also need to be testing, depending on the form of the component.

2. It may be possible the webserver responds before the client can ask to receive the returned message. This may be difficult to observe on a normal desktop computer, but slower mobile devices may take longer to execute the require code, particularly when they are executing other code at the same time.

3. A JavaScript procedure will accept any type of parameter. From a programmer's point of view, the values returned by such a procedure may appear to be returning a sensible result. In such cases the programmer will not easily detect the error.

   It is also common in JavaScript libraries to overload procedures, as is done in object-oriented language. (this means the same procedure name is used but different code is executing depending on the parameters passes) In such cases the programmer may accidentally call the wrong code because the wrong parameter type is used.

4. First, a web browser can be simulated by writing a program to send messages to the webserver and receive the response. By placing such code in a loop, there does not have to be a human operator and the code will execute much faster than a web browser which also needs to display the result. If one program cannot produce adequate load for testing, the program can be run on multiple computers to increase the load on a single server.

Activity 1-6: System testing

1. Systems testing (obviously) cannot be attempted until an entire system is available. System testing can certainly be planned before this. If a use-case approach to system testing is implemented, then the use-cases can be found early at the requirements gathering phase of the development.

   It is important to recognize that system testing is not something that happens after the developers have finished their job. If automated system testing is available, then it is not uncommon for a daily system test system to be implemented to test the latest version of the system every day. In so doing bugs can be exposed as each daily increment of code is added to the system. We will see how Microsoft uses a daily build in a later topic. Unfortunately, system testing can require manual testing because of the user interfaces that many software products have implemented. In these cases the system testing may be restricted to each release.

2. Use-cases are scenarios that an actual user would use with the system. These are probably not how a programmer would test smaller code sections and they will usually test many components of the system.

3. Automated testing is difficult when there is a large amount of manual input using advanced input devices. For example, test a GUI based system that requires mouse movement is hard (but not impossible) to implement. Similarly touch devices such as mobile phones are difficult to automate testing. Even when they are automated, they tend to repeat exact mouse or touch movements instead of the fairly random movements of a human user which may expose some errors.

Activity 1-7: Test driven development

1. *Running* a test before the code is implemented is really testing the test. It should produce an error, which will verify that the test is working. In a less obvious fashion, writing the test also clarifies many of the requirements that the user may have articulated.

2. *Regression testing*, is testing the software still performs as it used to before the latest changes. This is necessary because the current changes may have broken the old functioning of the software. Remember, the programmer making the latest changes may be a different person to the one who wrote the previous code, and so may not remember or understand reasons why the code was written the way it was.

3. Accumulating tests ensures a product continues to perform as it did in older versions (see answer 2). Debugging is made easier because the code that has changed since the last increment will be small, therefor there is a small amount of code that could have introduced any bugs. It is important

to note, however, the bug may not be in the new code. The new code may have exposed bugs in older code or just incompatibilities in the old code that require the old code to be changed.

4. Automatic testing reduces both time and the cost of human involvement in testing. If a test process takes a long time to implement there will be less testing applied. Similarly, the cost of human input into testing may make thorough testing less likely or less frequent.

Activity 1-8: Requirements based testing

1. Release testing is where the software developers are testing a system before it is released to the users. The software still has not been released so it is the last chance to make sure it meets the developer's understanding of the user's requirements. The idea of using independent (non-developer) testers is that it allows people who have not been involved in development to validate the specifications of the software without bias to the work they have already sunk into the project. If the tester is a coopted user, then the user's understanding of the specification could be better than the programmer's understanding of the written specification.
2. The first reason is one we have mentioned several times in the topic, that is it is impossible to test all combinations of inputs to a system due the number of possibilities. This means we must adopt a strategy based on experience and organisational standards. The "good enough" term is used when the developers believe, to the best of their knowledge, that the system is ready for release. As you have read, this may depend on the user expectations. A safety critical or highly secure system may require more thorough testing to be "good enough".
3. If a defect is found during validation of the specification, then obviously it must be fixed. However, depending on the severity of the problem, the development schedule, the difficulty of fixing the problem, and other reasons, the fix may be scheduled for another release.

Activity 1-9: Scenario and performance testing

1. Scenarios are meant to represent real user's intended use of the system. This means that they may be closer to the user's requirement than the elicited specifications. The ideal number scenarios is of course all possible scenarios. The ideal is not "ideal" for several reasons. First, a user who is well practiced in the domain of the system could easily forget some scenarios or assume the developers would understand that these are required. Secondly, once a system is implemented, it is not uncommon for new scenarios to be developed as users become used to the new system operation.
2. Yes. Scenarios could re-use system components in different ways. If the requirements is broken into component requirements, then those components can be included in many scenarios.
3. One of the reasons is to makes sure the system fails in a "nice" way. By "nice" we mean that the system does not lose or corrupt data as it fails. The system should also provide meaningful messages to users as it fails and, if possible, preserve partially completed work for the user.

   Another reason to stress to failure point is that it is common for any timing problems to become obvious when the system becomes stressed. These may be very rare under normal load and may not present in normal testing.
4. The reading suggests that stress test of distributed systems is important. We will discuss distributed system design in a later topic. Essentially these are systems that have multiple computers networked together. The reason why stress testing is important is that any of the components (computers or network links) may fail under load so tress testing will identify the point of failure and allow addition of any load management required, e.g. reject additional transaction.

Activity 1-10: User testing

1. Alpha testing is where users are involved in the development project at a early stage, well before release testing. With Beta testing the users are brought in close to the system release. Another way to look at the difference is that Alpha testers are involved in the later stages of system design, whereas Beta testers identify problems after the design is complete.

2. Alpha testers would want to be involved in a stock market trading app to see if it would give them an advantage over other market participants. In this case it may mean making more money.
3. Beta testers of computer games usually want to see the latest version of their favorite game before other people. In games where there is a player hierarchy, this may mean they get early plans and strategies in place for when the game is released more widely, hence enhancing their place in the user hierarchy.
4. The acceptance criteria will probably change during development for many reasons:
   - Requirements change due to environment changes (business environment, hardware/software environment, etc.)
   - Developer and users learn as the project progresses. Problems and advantages may emerge which change priorities and expected outcomes of the project.
   - User's priorities may change as they see developing versions of the project. Users may not understand the possibilities of the used technology until they see implemented components.
   - Not all projects run according to plan. Adjustments may need to be made to the expected outcomes due to less accurate predictions of timelines, budget, expertise availability, etc.
5. Negotiation is usually required because it is rare a system completely passes an acceptance test. If major changes are required, it may be economic to accept a lesser system so that efficiency advantages can be achieved. It is also common, that due to the evolving customer's environment, a system may meet the original requirements but not the ones at the time of acceptance. There may be negotiation in updating the system to the latest requirements.
6. At the end of a project there has been considerable investment already sunk in the project. From the customer's viewpoint involvement in specification and testing is not free, even if they do not have to pay the developer. There is also the time spent on the project. The prospect of abandoning the project and starting again will be unattractive when the customer has to continue to use the system that should have been replaced. It may be more expensive to continue an old system than to accept a modified, though imperfect, new system.

   From the developer's point of view, they will not be completely paid unless something is delivered. The developer will be open to negotiation to retrieve as much payment as possible. There is also reputation damage to consider with the developer.