8 Software testing

Objectives

The objective of this chapter is to introduce software testing and software testing processes. When you have read the chapter, you will:

- understand the stages of testing from testing, **during development to acceptance testing** by system customers;
- have been introduced to **techniques that help you choose test** cases that are geared to discovering program defects;
- understand **test-first development, where you design tests before writing code** and **run these tests automatically**;
- know the important differences between **component**, **system**, **and release testing** and be **aware of user testing processes and techniques**.

Content I.

8.1 Development testing
8.2 Test-driven development

Content II.

8.3 Release testing
8.4 User testing

The testing process **has two distinct goals**:

1. To demonstrate to the developer and the customer that the software meets its requirements. For **custom software**, this means that there should **be at least one test** for every requirement in the requirements document. **For generic software products**, it means that there s**hould be tests for all of the system features, plus combinations of these features,** that will be incorporated in the product release.
2. To discover situations in which the behavior of the software is incorrect, undesirable, or does not conform to its specification. These are a consequence of software defects.

Defect testing is concerned with rooting out undesirable system behavior such as system crashes, unwanted interactions with other systems, incorrect computations, and data corruption.

1. **The first goal leads to validation testing**, where you expect the system to perform correctly using a given set of test cases that reflect the system's expected use.
2. **The second goal leads to defect testing**, where the test cases are designed to expose defects. The test cases in defect testing can be deliberately obscure and need not reflect how the system is normally used.

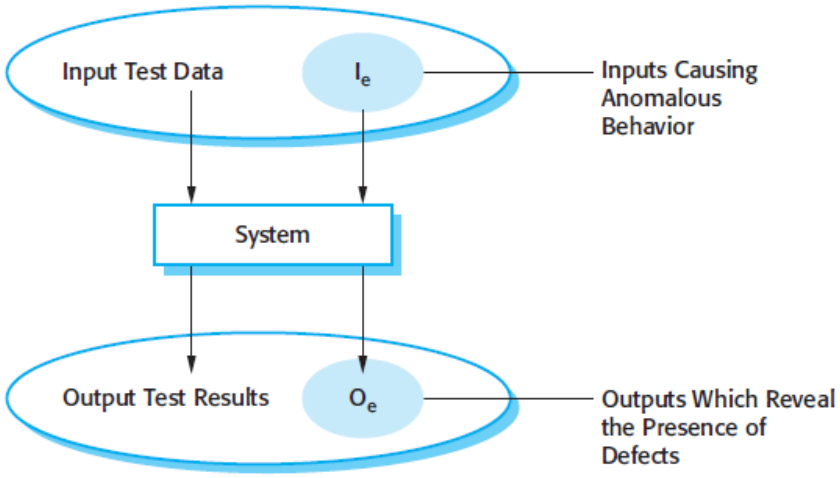


Figure 8.1 An input-output model of program testing

Figure 8.1 may help to explain the differences between **validation testing** and **defect testing**.

Think of the system being tested as a black box. The system accepts inputs from some input set I and generates outputs in an output set O. Some of the outputs will be erroneous.

These are the outputs in set O_e that are generated by the system in response to inputs in the set I_e.
The priority in defect testing is to find those inputs in the set I_e because these reveal **problems** with the system.

Validation testing involves testing with correct inputs that are outside I_e.
These stimulate the system to generate the expected correct outputs.

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

Testing is part of a broader process of software **verification and validation (V & V).**

Verification and validation are not the same thing, although they are often confused.

- **'Validation: Are we building the right product?'**
- **'Verification: Are we building the product right?'**

The aim of verification is to check that the software meets its stated functional and non-functional requirements.
Validation, however, is a more general process. The aim of validation is to ensure that the software meets the customer's expectations.

The ultimate goal of verification and validation processes **is to establish confidence** that the software system is '**fit for purpose'**.

This means that **the system must be good enough for its intended use**.

1. ***Software purpose*** **The more critical the software**, **the more important that it is reliable**. For example, the level of confidence required for software used to control a safety-critical system is much higher than that required for a prototype that has been developed to demonstrate new product ideas.

2. ***User expectations*** Because of their experiences with buggy, unreliable software,**many users *have low expectations* of software quality**. They **are not surprised when their software fails.** When a new system is installed, users may tolerate failures because the benefits of use outweigh the costs of failure recovery.**In these situations, you may not need to devote as much time to testing the software**.However, as software matures, users expect it to become more reliable so more thorough testing of later versions may be required.

3. ***Marketing environment*** When a system is marketed, the sellers of the system must take into account competing products, the price that customers are willing to pay for a system, and the required schedule for delivering that system. In a competitive environment, a software company may decide to release a **program before it has been fully tested and debugged because they want to be the first into the market**. If a software product **is very cheap**, users **may be willing to tolerate a lower level of reliability**.

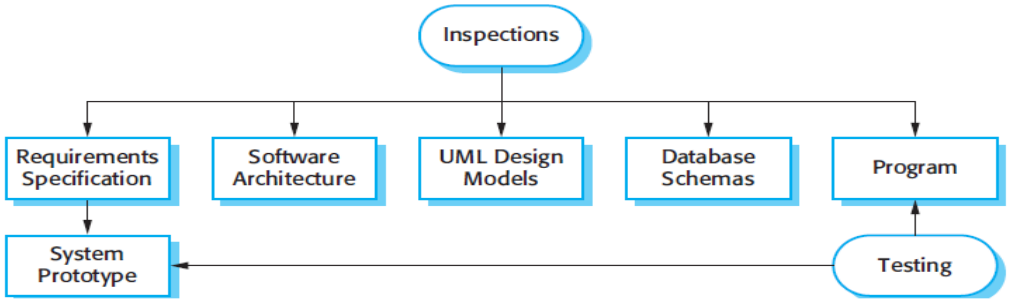


Figure 8.2 Inspections and testing

As well as software testing, the verification and validation process **may involve software inspections and reviews.** Inspections and reviews analyze and check the system requirements, design models, the program source code, and even proposed system tests. **These are so-called 'static' V & V techniques in which you don't need to execute the software to verify it**.

Inspections mostly focus on the source code of a system but any readable representation of the software, such **as its requirements or a design model, can be inspected**. When you inspect a system, you use knowledge of the system, its application domain, and the programming or modeling language to discover errors.

There are **three advantages of software inspection over testing**:

1. **During testing, errors can mask (hide) other errors**. When an error leads to unexpected outputs, you can never be sure if later output anomalies are due to a new error or are side effects of the original error. Because inspection is a static process, you don't have to be concerned with interactions between errors. **Consequently, a single inspection session can discover many errors in a system**.

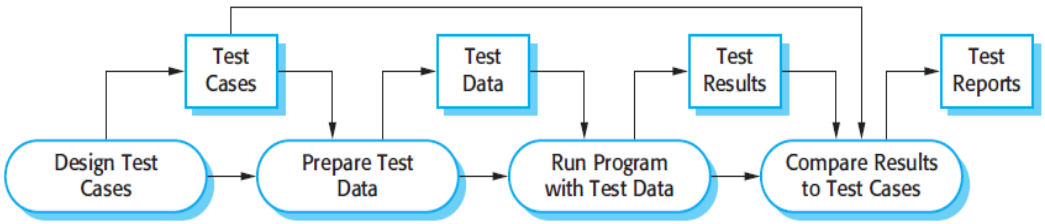
Test planning
Test planning is concerned with scheduling and resourcing all of the activities in the testing process. It involves defining the testing process, taking into account the people and the time available. Usually, a test plan will be created, which defines what is to be tested, the predicted testing schedule, and how tests will be recorded. For critical systems, the test plan may also include details of the tests to be run on the software.
http://www.SoftwareEngineering-9.com/Web/Testing/Planning.html

3. **Incomplete versions** of a system **can be inspected without additional costs**. If a program is incomplete, then you need to develop specialized test harnesses to test the parts that are available. This obviously adds to the system development costs.
4. As well as searching for program defects, **an inspection can also consider broader quality attributes of a program, such as compliance with standards, portability, and maintainability**. You can look for inefficiencies, inappropriate algorithms, and poor programming style that could make the system difficult to maintain and update.

Program inspections are an old idea and there have been several studies and experiments that have demonstrated that inspections **are more effective for defect discovery than program testing**. Fagan

However, inspections cannot replace software testing. Inspections are not good for discovering defects that arise because of unexpected interactions between different parts of a program, timing problems, or problems with system performance.

Furthermore, especially in small companies or development groups, it can be difficult and expensive to put together a separate inspection team as all potential members of the team may also be software developers.



.
Figure 8.3 A model of the software testing process

Typically, **a commercial software system has to go through three stages of testing**:

1. **Development testing,** where the system is tested during development to discover bugs and defects. System designers and **programmers** are likely to be involved in the testing process.

2. **Release testing**, where a separate testing team tests a complete version of the system before it is released to users. The aim of release testing is to check that the system meets the requirements of system stakeholders.

3. **User testing,** where users or potential users of a system test the system in their own environment. For software products, the 'user' may be an internal marketing group who decide if the software can be marketed, released, and sold.

Acceptance testing is one type of user testing where the customer formally tests a system to decide if it should be accepted from the system supplier or if further development is required.

In practice, the testing process **usually involves a mixture of manual and automated testing.**

In **manual testing**, a tester runs **the program with some test data and compares the results to their expectations**. They note and report discrepancies to the program developers. **In automated testing**, the **tests are encoded in a program that is run each time the system under development is to be tested**. This is usually **faster** than manual testing, especially **when it involves regression testing**—**re-running previous tests to check that changes to the program have not introduced new bugs**.

The use of automated testing has increased considerably over the past few years.
However, testing ***can never be completely automated*** as automated tests **can only check that a program does what it is supposed to do**. **It is practically impossible to use automated testing to test systems that depend on how things look (e.g., a graphical user interface), or to test that a program does not have unwanted side effects**.

8.1 Development testing

Development testing includes all testing activities that are carried out by the team developing the system. **The tester of the software is usually the programmer who developed that software, although this is not always the case.**

Some development processes **use programmer/tester pairs** where each programmer has an associated tester who develops tests and assists with the testing process.

For critical systems, **a more formal process may be used**, with a separate testing group within the development team. They are responsible for developing tests and maintaining detailed records of test results

Debugging
Debugging is the process of fixing errors and problems that have been discovered by testing. Using information from the program tests, debuggers use their knowledge of the programming language and the intended outcome of the test to locate and repair the program error. This process is often supported by interactive debugging tools that provide extra information about program execution.
http://www.SoftwareEngineering-9.com/Web/Testing/Debugging.html

.

During development, testing **may be carried out at three levels** of granularity:

1. **Unit testing**, where individual program units or object classes are tested. Unit testing should focus on testing the functionality of objects or methods.
2. **Component testing,** where several individual units are integrated to create composite components. Component testing should focus on testing component interfaces.
3. **System testing,** where some or all of the components in a system are integrated and the system is tested as a whole. System testing should focus on testing component interactions.

Development testing is primarily a defect testing process, where **the aim of testing is to discover bugs** in the software. It is therefore usually interleaved with debugging—
the process of locating problems with the code and changing the program to fix these problems.

8.1.1 Unit testing

Unit testing is the process of testing program components, such as methods or object classes.

Individual functions or methods are the simplest type of component.
Your tests should be calls to these routines with different input parameters. You can use the approaches to test case design discussed in Section 8.1.2, to design the function or method tests.
When you are testing object classes, you should design your tests to provide coverage of all of the features of the object.
This means that you should:
• test all operations associated with the object;
• set and check the value of all attributes associated with the object;
• put the object into all possible states.
This means that you should simulate all events that cause a state change.

Whenever possible, **you should automate unit testing**. In automated unit testing, you make use of a test automation framework (such as JUnit) to write and run your program tests. Unit testing frameworks provide generic test classes that you extend to create specific test cases. They can then run all of the tests that you have implemented and report, often through some GUI, on the success or failure of the tests.
An entire test suite can often be run in a few seconds so it is possible to execute all the tests every time you make a change to the program.

An automated test has three parts:
1. A setup part, where you initialize the system with the test case, namely the inputs and expected outputs.

2. A call part, where you call the object or method to be tested.
3. An assertion part where you compare the result of the call with the expected result. If the assertion evaluates to true, the test has been successful; if false, then it has failed.

Sometimes the object that you are testing has dependencies on other objects that may not have been written or which slow down the testing process if they are used.

For example, if your object calls a database, this may involve a slow setup process before it can be used. In these cases, you may decide to **use mock objects**.
Mock objects are objects with the same interface as the external objects being used that simulate its functionality.
Therefore, a mock object simulating a database may have only a few data items that are organized in an array. They can therefore be accessed quickly, without the overheads of calling a database and accessing disks.
Similarly, mock objects can be used to simulate abnormal operation or rare events.

8.1.2 Choosing unit test cases

Testing is expensive and time consuming, so it is important that you choose effective unit test cases. Effectiveness, in this case, means two things:

1. The test cases should show that, when used as expected, the component that you are testing does what it is supposed to do.
2. If there are defects in the component, these should be revealed by test cases.

You should therefore **write two kinds of test case**.

1. **The first of these should reflect normal operation** of a program and should show that the component works. For example, if you are testing a component that creates and initializes a new patient record, then your test case should show that the record exists in a database and that its fields have been set as specified.

2. **The other kind of test case should be based on testing experience** of where common problems arise. It should **use abnormal** inputs to check that these are properly processed and do not crash the component.

two possible strategies here that can be effective in helping you choose test cases. These are:

1. ***Partition testing***, where you identify groups of inputs that have common characteristics and should be processed in the same way. You should choose tests from within each of these groups.
2. ***Guideline-based testing,*** where you use testing guidelines to choose test cases. These guidelines reflect previous experience of the kinds of errors that programmersoften make when developing components.

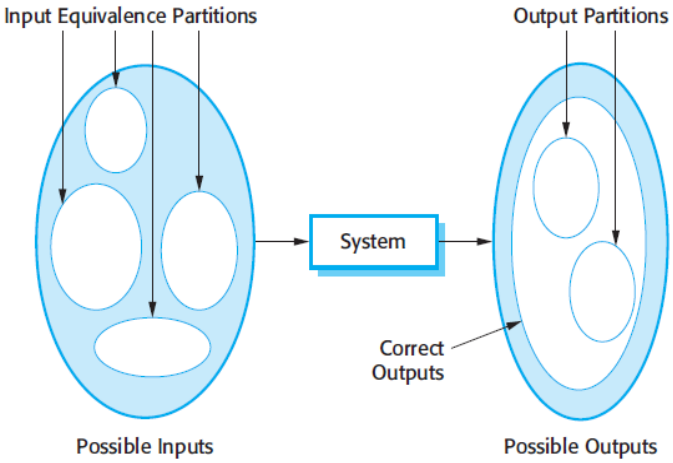


Figure 8.5 Equivalence partitioning

For example, when you are testing programs with sequences, arrays, or lists, guidelines that could help reveal defects include:

1. **Test software with sequences that have only a single value.** Programmers naturally

think of sequences as made up of several values and sometimes they embed this assumption in their programs. Consequently, if presented with a single value sequence, a program may not work properly.

2. **Use different sequences of different sizes in different tests**. This decreases the chances that a program with defects will accidentally produce a correct output because of some accidental characteristics of the input.
3. **Derive tests so that the first, middle, and last elements of the sequence are accessed**. This approach is reveals problems at partition boundaries.

Whittaker's book (2002) includes **many examples of guidelines that can be used in test case design**. Some of the most general guidelines that he suggests are:

- Choose inputs that force the system to generate all error messages;
- Design inputs that cause input buffers to overflow;
- Repeat the same input or series of inputs numerous times;
- Force invalid outputs to be generated;
- Force computation results to be too large or too small.

As you gain experience with testing, you can develop your own guidelines about how to choose effective test cases. I give more examples of testing guidelines in the next section of this chapter.

8.1.3 Component testing

Software components are often composite components that are made up of several interacting objects. For example, in the weather station system, the reconfiguration component includes objects that deal with each aspect of the reconfiguration. You access the functionality of these objects through the defined component interface. Testing composite components should therefore focus on showing that the component interface behaves according to its specification. **You can assume that unit tests on the individual objects within the component have been completed**.

Figure 8.7 illustrates the idea of component interface testing. Assume that components A, B, and C have been integrated to create a larger component or subsystem. The test cases are not applied to the individual components but rather to the interface of the composite component created by combining these components. **Interface errors in the composite component may not be detectable by testing the individual objects because these errors result from interactions between the objects in the component**.

There are different types of interface between program components and, consequently, different types of interface error that can occur:

1. ***Parameter interfaces*** These are interfaces in which data or sometimes function references are passed from one component to another. Methods in an object have a parameter interface.

2. ***Shared memory interfaces*** These are interfaces in which a block of memory is shared between components. Data is placed in the memory by one subsystem

and retrieved from there by other sub-systems. This type of interface is often used in embedded systems, where sensors create data that is retrieved and processed by other system components.
3. ***Procedural interfaces*** These are interfaces in which one component encapsulates a set of procedures that can be called by other components. Objects and reusable components have this form of interface.
4. ***Message passing interfaces*** These are interfaces in which one component requests a service from another component by passing a message to it. A return message includes the results of executing the service. Some object-oriented systems have this form of interface, as do client–server systems.

Some general guidelines for interface testing are:
1. Examine the code to be tested and explicitly list each call to an external component. Design a set of tests in which the values of the parameters to the external components are at the extreme ends of their ranges. These extreme values are most likely to reveal interface inconsistencies.
2. Where pointers are passed across an interface, always test the interface with null pointer parameters.
3. Where a component is called through a procedural interface, design tests that deliberately cause the component to fail. Differing failure assumptions are one of the most common specification misunderstandings.
4. Use stress testing in message passing systems. This means that you should design tests that generate many more messages than are likely to occur in practice. This is an effective way of revealing timing problems.
5. Where several components interact through shared memory, design tests that vary the order in which these components are activated. These tests may reveal implicit assumptions made by the programmer about the order in which the shared data is produced and consumed.

Inspections and reviews **can sometimes be more cost effective** than testing for discovering interface errors. Inspections can concentrate on component interfaces and questions about the assumed interface behavior asked during the inspection process. A strongly typed language such as Java allows many interface errors to be trapped by the compiler. Static analyzers (see Chapter 15) can detect a wide range of interface errors.

Incremental integration and testing
System testing involves integrating different components then testing the integrated system that you have created.
You should always use an incremental approach to integration and testing (i.e., you should integrate a component, test the system, integrate another component, test again, and so on). This means that if problems occur, it is probably due to interactions with the most recently integrated component.
Incremental integration and testing is fundamental to agile methods such as XP, where regression tests (see Section 8.2) **are run every time a new increment is integrated**.
http://www.SoftwareEngineering-9.com/Web/Testing/Integration.html

8.1.4 System testing

System testing during development involves integrating components to create a version of the system and then testing the integrated system. **System testing checks that components are compatible, interact correctly and transfer the right data at the right time across their interfaces.** It obviously overlaps with component testing but there are two important differences:

1. **During system testing, reusable components that have been separately developed**

and off-the-shelf systems may be integrated with newly developed components. The complete system is then tested.

2. **Components developed by different team members or groups may be integrated at this stage**. System testing is a collective rather than an individual process. In some companies, system testing may involve a separate testing team with no involvement from designers and programmers.

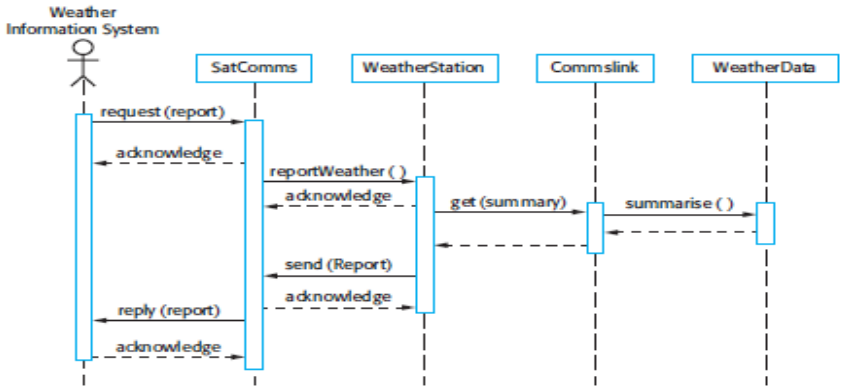


Figure 8.8 Collect weather data sequence chart

Figure 8.8 is an example of a sequence model, shown as a UML sequence diagram. This diagram shows the sequence of interactions that take place when an external system requests the summarized data from the weather station. You read sequence diagrams from top to bottom:

1. The **SatComms** object receives a request from the weather information system to collect a weather report from a weather station. It acknowledges receipt of this request. The stick arrowhead on the sent message indicates that the external system does not wait for a reply but can carry on with other processing.
2. **SatComms** sends a message to **WeatherStation**, via a satellite link, to create a summary of the collected weather data. Again, the stick arrowhead indicates that SatComms does not suspend itself waiting for a reply.
3. **WeatherStation** sends a message to a **Commslink** object to summarize the weather data. In this case, the squared-off style of arrowhead indicates that the instance of the WeatherStation object class waits for a reply.
4. **Commslink** calls the summarize method in the object **WeatherData** and waits for a reply.
5. The weather data summary is computed and returned to **WeatherStation** via the **Commslink** object.
6. **WeatherStation** then calls the **SatComms** object to transmit the summarized data to the weather information system, through the satellite communications system.

Therefore, issuing a request for a report will result in the execution of the following thread of methods:

SatComms:request _ WeatherStation:reportWeather _ Commslink:Get(summary)_ **WeatherData:summarize**

The sequence diagram helps you design the specific test cases that you need as it shows what inputs are required and what outputs are created:
1. An input of a request for a report should have an associated acknowledgment. A report should ultimately be returned from the request. During testing, you should create summarized data that can be used to check that the report is correctly organized.
2. An input request for a report to WeatherStation results in a summarized report being generated. You can test this in isolation by creating raw data corresponding to the summary that you have prepared for the test of SatComms and checking that the WeatherStation object correctly produces this summary. This raw data is also used to test the WeatherData object.

8.2 Test-driven development

Test-driven development (TDD) is an approach to program development in which you interleave testing and code development (Beck, 2002; Jeffries and Melnik, 2007).

Essentially, you develop the code incrementally, along with a test for that increment.

1. **You don't move on to the next increment** until the code that you have developed passes its test.
2. Test-driven development was introduced as part of agile methods such as Extreme Programming. However, it can also be used in plan-driven development processes.

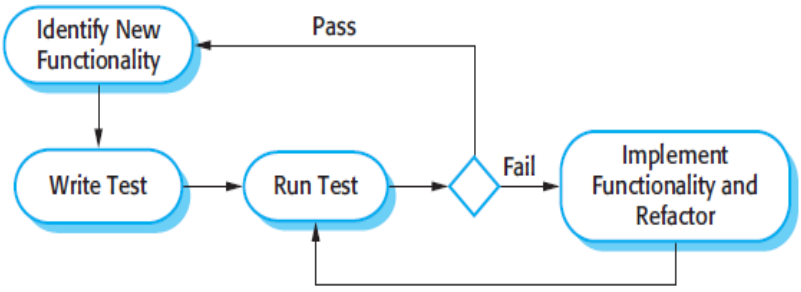


Figure 8.9 Test-driven development

The fundamental TDD process is shown in Figure 8.9. The steps in the process

are as follows:

1. **You start by identifying the increment of functionality that is required**. This should normally be small and implementable in a few lines of code.

2. **You write a test for this functionality and implement this as an automated test.** This means that the test can be executed and will report whether or not it has passed or failed.

3. **You then run the test, along with all other tests that have been implemented**. Initially, you have not implemented the functionality so the new test will fail. This is deliberate as it shows that the test adds something to the test set.

4. **You then implement the functionality and re-run the test**. This may involve refactoring existing code to improve it and add new code to what's already there.

5. **Once all tests run successfully, you move on to implementing the next chunk of functionality.**

To write a test, you need to understand what is intended, as this understanding makes it easier to write the required code.
Of course, if you have incomplete knowledge or understanding, then test-driven development won't help.

If you don't know enough to write the tests, you won't develop the required code. For example, if your computation involves division, you should check that you are not dividing the numbers by zero. If you forget to write a test for this, then the code to check will never be included in the program.

As well **as better problem understanding, other benefits** of test-driven development are:

1. ***Code coverage*** In principle, every code segment that you write should have at least one associated test. Therefore, you can be confident that all of the code in the system has actually been executed. Code is tested as it is written so defects are discovered early in the development process.
2. ***Regression testing*** A test suite is developed incrementally as a program is developed. You can always run regression tests to check that changes to the program have not introduced new bugs.
3. ***Simplified debugging*** When a test fails, it should be obvious where the problem lies. The newly written code needs to be checked and modified. You do not need to use debugging tools to locate the problem. Reports of the use of test-driven development suggest that it is hardly ever necessary to use an automated debugger in test-driven development (Martin, 2007).
4. ***System documentation*** The tests themselves act as a form of documentation that describe what the code should be doing. Reading the tests can make it easier to understand the code.

One of the **most important benefits of test-driven development is that it reduces the costs of regression testing.**

- Regression testing involves running test sets that have successfully executed after changes have been made to a system.

- The regression test checks that these changes have not introduced new bugs into the system and that the new code interacts as expected with the existing code.
- Regression testing is very expensive and often impractical when a system is manually tested, as the costs in time and effort are very high. In such situations, you have to try and choose the most
- relevant tests to re-run and it is easy to miss important tests.

Test-driven development has proved to be a successful approach for small and medium-sized projects. Generally, programmers who have adopted this approach are happy with it and find it a more productive way to develop software (Jeffries and Melnik, 2007). In some trials, it has been shown to lead to improved code quality; in others, the results have been inconclusive. **However**, there is no evidence that TDD leads to poorer quality code.

8.3 Release testing

Release testing is the process of testing a particular release of a system that is intended for use outside of the development team. **Normally, the system release is for customers and users.** In a complex project, however, the release could be for other teams that are developing related systems. **For** software products, the release could be for product management who then prepare it for sale.

There are **two important distinctions between release testing and system testing** during the development process:
1**. A separate team that has not been involved** in the system development **should be responsible for release testing**.
2. **System testing by the development team should focus** on discovering bugs in the system (defect testing). The objective of release testing **is to check that the system meets its requirements** and is good enough for external use (validation testing).

The primary goal of the release testing process is to convince the supplier of the system that it is good enough for use.

8.3.1 Requirements-based testing

A general principle of good requirements engineering practice **is that requirements should be testable;** that is, the requirement should be written so that a test can be designed for that requirement. A tester can then check that the requirement has been satisfied. Requirements-based testing, therefore, is a systematic approach to test case design where you consider each requirement and derive a set of tests for it. Requirements-based testing is validation rather than defect testing—you are trying to demonstrate that the system has properly implemented its requirements.

For example, consider related requirements for the MHC-PMS (introduced in Chapter 1), which are concerned with checking for drug allergies:
If a patient is known to be allergic to any particular medication, then prescription of that medication shall result in a warning message being issued to the

system user.
If a prescriber chooses to ignore an allergy warning, they shall provide a reason why this has been ignored.

8.3.2 Scenario testing

Scenario testing is an approach to release testing where you devise typical scenarios of use and use these to develop test cases for the system. A scenario is a story that describes one way in which the system might be used. Scenarios should be realistic and real system users should be able to relate to them. If you have used scenarios as part of the requirements engineering process (described in Chapter 4), then you may be able to reuse these as testing scenarios.

8.3.3 Performance testing

Once a system has been completely integrated, it is possible to test for emergent properties, such as performance and reliability. Performance tests have to be designed to ensure that the system can process its intended load. This usually involves running a series of tests where you increase the load until the system performance becomes unacceptable.

As with other types of testing, performance testing is concerned both with demonstrating that the system meets its requirements and discovering problems and defects in the system. To test whether performance requirements are being achieved, you may have to construct an operational profile. An operational profile (see Chapter 15) is a set of tests that reflect the actual mix of work that will be handled by the system. Therefore, if 90% of the transactions in a system are of type A; 5% of type B; and the remainder of types C, D, and E, then you have to design the operational profile so that the vast majority of tests are of type A. Otherwise, you will not get an accurate test of the operational performance of the system.

This approach, of course, is not necessarily the best approach for defect testing. Experience has shown that an effective way to discover defects is to design tests around the limits of the system. In performance testing, this means stressing the system by making demands that are outside the design limits of the software. This is known as ‘stress testing’. For example, say you are testing a transaction processing system that is designed to process up to 300 transactions per second. You start by testing this system with fewer than 300 transactions per second. You then gradually increase the load on the system beyond 300 transactions per second until it is well beyond the maximum design load of the system and the system fails. This type of testing has two functions:

1. It tests the failure behavior of the system. Circumstances may arise through an unexpected combination of events where the load placed on the system exceeds the maximum anticipated load. In these circumstances, it is important that system failure should not cause data corruption or unexpected loss of user services. Stress testing checks that overloading the system causes it to ‘fail-soft’ rather than collapse under its load.
2. It stresses the system and may cause defects to come to light that would not normally be discovered. Although it can be argued that these defects are unlikely to cause system failures in normal usage, there may be unusual combinations of normal circumstances that the stress testing replicates.

Stress testing is particularly relevant to distributed systems based on a network of processors. These systems often exhibit severe degradation when they are heavily loaded. The network becomes swamped with coordination data that the different

processes must exchange. The processes become slower and slower as they wait for the required data from other processes. Stress testing helps you discover when the degradation begins so that you can add checks to the system to reject transactions beyond this point.

8.4 User testing

User or customer testing is a stage in the testing process in which users or customers provide input and advice on system testing.

This may involve formally testing a system that has been commissioned from an external supplier, or could be an informal process where users experiment with a new software product to see if they like it and that it does what they need. User testing is essential, even when comprehensive system and release testing have been carried out. The reason for this is that influences from the user's working environment have a major effect on the reliability, performance, usability, and robustness of a system.

It is practically impossible for a system developer to replicate the system's working environment, as tests in the developer's environment are **inevitably artificial.**

For example, a system that is intended for use in a hospital is used in a clinical environment where other things are going on, such as patient emergencies, conversations with relatives, etc. These all affect the use of a system, but developers cannot include them in their testing environment.

In practice, there are **three different types of user testing**:

1. **Alpha testing,** where users of the software work with the development team to test the software at the developer's site.
2. **Beta testing,** where a release of the software is made available to users to allow them to experiment and to raise problems that they discover with the system developers.
3. **Acceptance testing,** where customers test a system to decide whether or not it is ready to be accepted from the system developers and deployed in the customer environment.

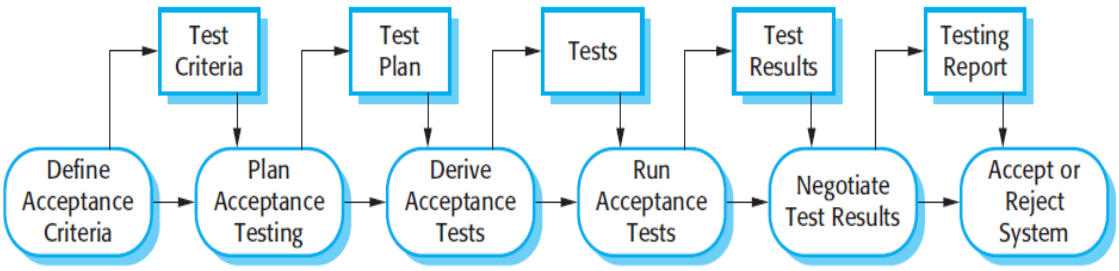


Figure 8.11 The acceptance testing process

There **are six stages in the acceptance testing process**, as shown in Figure 8.11. They are:

1. ***Define acceptance criteria*** This stage should, ideally, **take place early in the process before the contract for the system is signed.** The acceptance criteria should be part of the system contract and be agreed between the customer and the developer. In practice, however, it can be difficult to define criteria so early

in the process. Detailed requirements may not be available and there may be significant requirements change during the development process.

2. ***Plan acceptance testing*** **This involves deciding on the resources, time,** and **budget** for acceptance testing and **establishing** a **testing schedule**. The acceptance test plan should also discuss the **required coverage** of the **requirements** and the order in **which system features are tested**. It should define risks to the testing process, such as system crashes and inadequate performance, and discuss how these risks can be mitigated.

3. ***Derive acceptance tests*** Once acceptance criteria have been established, **tests have to be designed to check whether or not a system is acceptable**. Acceptance tests should **aim to test both the functional and non-functional characteristics (e.g., performance) of the system.** They should, ideally, provide complete coverage of the system requirements. In practice, it is difficult to establish completely objective acceptance criteria. There is often scope for argument about whether or not a test shows that a criterion has definitely been met.

4. ***Run acceptance tests*** The agreed acceptance tests are executed on the system. Ideally, this should take place in the actual environment where the system will be used, but this may be disruptive and impractical. Therefore, a user testing environment may have to be set up to run these tests. It is difficult to automate this process as part of the acceptance tests may involve testing the interactions between end-users and the system. Some training of end-users may be required.

5. ***Negotiate test results*** It is very unlikely that all of the defined acceptance tests will pass and that there will be no problems with the system. If this is the case, then acceptance testing is complete and the system can be handed over. More commonly, some problems will be discovered. In such cases, the developer and the customer have to negotiate to decide if the system is good enough to be put into use. They must also agree on the developer's response to identified problems.

6. ***Reject/accept system*** This stage involves a meeting between the developers and the customer to decide on whether or not the system should be accepted. If the system is not good enough for use, then further development is required to fix the identified problems. Once complete, the acceptance testing phase is repeated.

KEY POINTS

- Testing can only show the presence of errors in a program. It cannot demonstrate that there are no remaining faults.
- Development testing is the responsibility of the software development team. A separate team should be responsible for testing a system before it is released to customers. In the user testing process, customers or system users provide test data and check that tests are successful.
- Development testing includes unit testing, in which you test individual objects and methods; component testing, in which you test related groups of objects; and system testing, in which you test partial or complete systems.
- When testing software, you should try to 'break' the software by using experience and guidelines to choose types of test cases that have been effective in discovering defects in other systems.
- Wherever possible, you should write automated tests. The tests are embedded in a program that can be run every time a change is made to a system.
- Test-first development is an approach to development where tests are written before the code to be tested. Small code changes are made and the code is refactored until all tests execute successfully.
- Scenario testing is useful because it replicates the practical use of the system. It involves inventing a typical usage scenario and using this to derive test cases.
- Acceptance testing is a user testing process where the aim is to decide if the software is good enough to be deployed and used in its operational environment.

E XERCISES

8.1. Explain why it is not necessary for a program to be completely free of defects before it is delivered to its customers.
8.2. Explain why testing can only detect the presence of errors, not their absence.
8.3. Some people argue that developers should not be involved in testing their own code but that all testing should be the responsibility of a separate team. Give arguments for and against testing by the developers themselves.
8.4. You have been asked to test a method called 'catWhiteSpace' in a 'Paragraph' object that, within the paragraph, replaces sequences of blank characters with a single blank character. Identify testing partitions for this example and derive a set of tests for the 'catWhiteSpace' method.
8.5. What is regression testing? Explain how the use of automated tests and a testing framework such as JUnit simplifies regression testing.
8.6. The MHC-PMS is constructed by adapting an off-the-shelf information system. What do you think are the differences between testing such a system and testing software that is developed using an object-oriented language such as Java?
8.7. Write a scenario that could be used to help design tests for the wilderness weather station system.
8.8. What do you understand by the term 'stress testing'? Suggest how you might stress test the MHC-PMS.
8.9. What are the benefits of involving users in release testing at an early stage in the testing process? Are there disadvantages in user involvement?
8.10. A common approach to system testing is to test the system until the testing budget is exhausted and then deliver the system to customers. Discuss the ethics of this approach for systems that are delivered to external customers.

EXERCIȚII

8.1. Explicați de ce nu este necesar ca un program să fie complet lipsit de defecte înainte ca acesta să fie livrate clienților săi.
8.2. Explicați de ce testarea poate detecta doar prezența erorilor, nu absența acestora.
8.3. Unii oameni susțin că dezvoltatorii nu ar trebui să fie implicați în testarea propriului cod, cică toate testele ar trebui să fie responsabilitatea unei echipe separate. Dă argumente pro și contra testarea chiar de către dezvoltatori.
8.4. Vi s-a cerut să testați o metodă numită „catWhiteSpace" într-un obiect „Paragraf" care, în cadrul paragrafului, înlocuiește secvențele de caractere goale cu un singur caracter gol. Identificați partițiile de testare pentru acest exemplu și obțineți un set de teste pentru „catWhiteSpace"metodă.
8.5. Ce este testarea de regresie? Explicați modul în care utilizarea testelor automate și a unui cadru de testare precum JUnit simplifică testarea de regresie.
8.6. MHC-PMS este construit prin adaptarea unui sistem de informații la distanță. Tu ce faci cred că sunt diferențele dintre testarea unui astfel de sistem și testarea software-ului care este dezvoltat folosind un limbaj orientat obiect cum ar fi Java?
8.7. Scrieți un scenariu care ar putea fi folosit pentru a ajuta la proiectarea testelor pentru stația meteo în sălbăticie sistem.
8.8. Ce înțelegeți prin termenul „testarea stresului"? Sugerează modul în care ai putea testa testul de stress MHC-PMS.
8.9. Care sunt avantajele implicării utilizatorilor în testarea lansării într-un stadiu incipient al unui proces de testare? Există dezavantaje în implicarea utilizatorilor?
8.10. O abordare comună a testării sistemului este de a testa sistemul până când bugetul de testare este epuizat și apoi livrați sistemul clienților. Discutați despre etica acestei abordări pentru sistemele care sunt livrate clienților externi.