Hyperiondev

TASK

Software Testing

Visit our website

Introduction

Welcome to the Software Testing Task!

All software developers need to be able to ensure the quality of their code. Software testing is essential in this regard! This task, will explain the aims of software testing as well as some of the most important types of testing that should be done.

Get in touch

Connect for support

Remember that with our courses, you're not alone! You can contact your mentor to get support on any aspect of your course.

The best way to get help is to login to **www.hyperiondev.com/portal** to start a chat with your mentor. You can also schedule a call or get support via email.

Your mentor is happy to offer you support that is tailored to your individual career or education needs. Do not hesitate to ask a question or for additional support!

What is Testing?

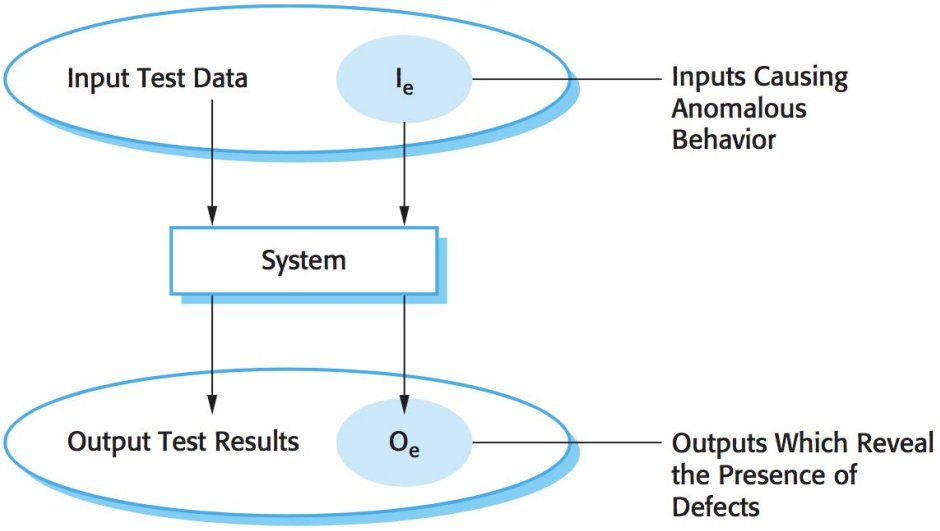
Testing aims to show that your program works as intended. It is also used to discover any defects in your program before it is put to use. When testing software, you execute your program using artificial data. You then check the results of the test run for errors or anomalies.

The testing process, in general, has two distinct goals:

1. To demonstrate that the software meets its requirements to both the developer and the customer.

2. To discover situations in which the behaviour of the software is incorrect, undesirable, or does not conform to its specification. These are normally a result of defects in the software.

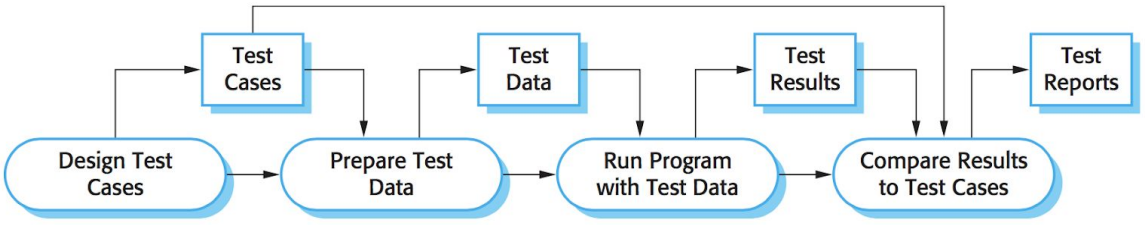
The first goal leads to **validation testing**. This is where you expect the system to perform correctly using a given set of test cases that reflect the system's expected use. The second goal leads to **defect testing**. This is where the test cases, which can be deliberately obscure and need not reflect how the system is normally used, are designed to expose defects. However, there is no distinct boundary between these two approaches to testing. During validation testing, you will probably find defects in the system. During defect testing, some of the tests will probably show that the program meets its requirements.



An input-output model of system testing (Sommerville 2010)

The diagram above helps explain the differences between validation and defect testing. You can think of the entire software system as being a black box. The system accepts inputs from some input set and generates outputs in an output set. Some of the outputs will be incorrect. With defect testing, the priority is to find the inputs within the set, since these reveal the problems with the system. Validation testing, on the other hand, involves testing with correct inputs that are outside the input set since these cause the system to generate the expected correct outputs.

However, testing cannot guarantee that the software is free of defects or that it will behave as specified in every circumstance. It is always possible that a test that you have overlooked could discover further problems with the system. As Edsger Dijkstra states (Dijkstra et al., 1972): *Testing can only show the presence of errors, not their absence.*



A model of the software testing process (Sommerville 2010)

The diagram above shows an abstract model of the 'traditional' testing process, as used in plan-driven development. Test cases are "specifications of the inputs to the test and the expected output from the system (the test results), plus a statement of what is being tested" (Shams, 2014) . Test data are the inputs that are being used to test a system. Test data can sometimes be generated automatically. However, automatic test case generation is impossible, as people who understand what the system is supposed to do must be involved to specify the expected test results. Nevertheless, test execution can be automated. The expected results are automatically compared with the predicted results.

Commercial software typically needs to go through three stages of testing:

1. **Development Testing:** Rather than testing once the program is developed, this stage tests the software during the development stage. The aim is to discover bugs and defects. System designers and programmers are typically

involved in the testing process.

2. **Release Testing:** Once a complete version of the software is ready, before it is released to users it is tested by an independent testing team to check that the system meets the requirements of system stakeholders.

3. **User Testing:** This is where users or potential users of a system test the system in their own environment. A customer can formally test the software to see if it is ready to be accepted from the system supplier or if more development is needed. This is known as Acceptance testing.

The testing process usually involves a mixture of manual and automated testing in practice. In manual testing, a tester runs the program with some test data and compares the results to their expectations. They then note and report any discrepancies to the program developers. In automated testing, tests are encoded in a program that is run each time the system is tested. This is usually faster than manual testing.

Development Testing

Development testing includes all testing activities that are carried out by the team developing the system. The person that tests the software is usually the programmer who developed that software, although this is not always the case. Some development processes use pairs of programmers and testers where each programmer has an associated tester who develops tests and assists with the testing process. A more formal process may be used for critical systems. This involves an independent testing group within the team that is responsible for developing tests and maintaining detailed records of test results.

Testing may be carried out at three levels of granularity during development:

1. **Unit testing**: This is where individual program units or object classes are tested. Unit testing focuses on testing the functionality of objects or methods.

2. **Component testing**: This is where several individual units are integrated to create composite components. Component testing focuses on testing component interfaces.

3. **System testing**: This is where some or all of the components in a system are integrated and the system is tested as a whole. System testing focuses on

testing component interactions.

Development testing is primarily a defect testing process, where testing aims 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.

Unit Testing

Unit testing is the process of testing program components, such as methods or object classes. The simplest type of components are individual functions or methods. Your tests should be calls to these functions or methods with different input parameters. You should design your tests to provide coverage of all of the features of the object when you are testing classes. This means that you should:

- test all operations associated with the object
- set and check the value of attributes associated with the object
- put the object into all possible states, i.e. simulating all events that might cause a state change.

Whenever you can, you should try to automate unit testing. In automated unit testing, you use 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 on the success or failure of the tests, often through some GUI. An entire test suite can often be run in a few seconds. Thus, it is possible to execute all the tests every time you make a change to the program.

There are three parts to an automated test:

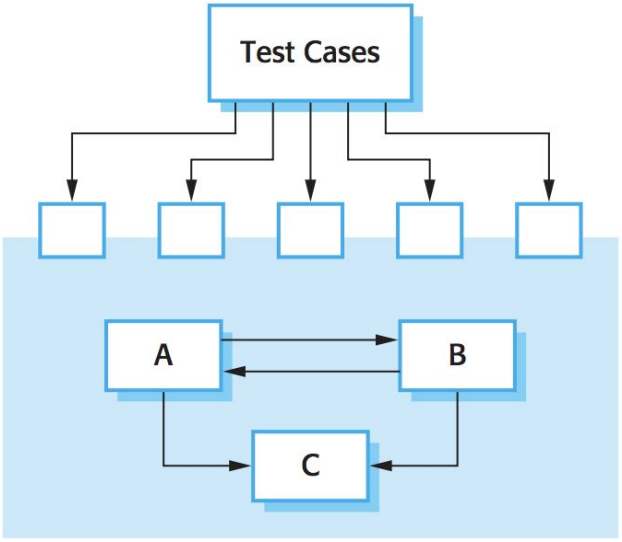
1. A setup part, where you initialise the system with the test case, namely the inputs and expected outputs.
2. A call part, where you call the object or method that you want to test.
3. An assertion part, where you compare the result of the call with the expected result. If the assertion evaluates to true, the test has been

successful. Otherwise, it has failed.

Sometimes the object that you are testing has dependencies on other objects. These objects may not have been written yet or slow down the testing process if they are used. For example, you could have a long setup process if your object calls a database before it can be used. For these cases, you could use mock objects. These are objects with the same interface as the external objects and they simulate their functionality. A mock object simulating a database may have only a few data items that are organised in an array. Therefore, they can be accessed quickly, without the overheads of calling a database and accessing disks. Mock objects can similarly be used to simulate abnormal operation or rare events. For example, if your system is intended to take action at certain times of day, your mock object can return those times, irrespective of the actual clock time.

Component Testing

Software components are usually composite components that comprise of various interacting objects. The functionality of these objects is accessed through the defined component interface. Therefore, testing composite components should 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.



Interface testing (Sommerville 2010)

The diagram above illustrates the idea of component interface testing. Assume

that components A, B, and C have been integrated to create a larger component. The test cases are not applied to the components themselves, but to the interface created by the combination of them. This type of testing reveals errors that result from interactions between the objects in the component.

There are different types of interfaces between program components. Therefore, different types of interface errors that can occur:

- **Parameter interfaces**: These are where data is passed from one component to another. Methods in an object have a parameter interface.

- **Shared memory interfaces**: These are where a block of memory is shared between components. Data is put in memory by one subsystem and then retrieved by others. It is often used in embedded systems, where sensors create data that is retrieved and processed by other system components.

- **Procedural interfaces**: These are where one component encapsulates a set of procedures that can be called by other components. Objects and reusable components have this type of interface.

- **Message passing interfaces**: These are where 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.

In complex systems, interface errors are one of the most common forms of errors. These errors fall into three classes:

1. **Interface misuse**: A calling component another component and makes an error in the use of its interface.

2. **Interface misunderstanding**: A calling component misunderstands the specification of the interface of the called component and makes assumptions about its behavior and so it does not behave as predicted. This causes further unexpected behavior in the calling component.

3. **Timing errors**: These occur in real-time systems that use shared memory or a message-passing interface. The producer and the consumer of data may operate at different speeds, and so the consumer could access out-of-date information if the producer of the information has not updated the shared interface information.

Some interface faults may only manifest themselves under unusual conditions, so testing for interface defects may be difficult. Another problem may arise because of interactions between faults in different modules or objects. Faults in one object could only be detected when some other object behaves in an unpredicted way. Some general guidelines for interface testing are:

- Examine the code to be tested and list each call to an external component. Design a set of tests where values passed to the external components are at the extreme ends of their ranges. These extreme values will usually show interface inconsistencies.

- Test the interface with null pointer parameters if pointers are passed across an interface.

- Where a component is called through a procedural interface, design tests that deliberately cause the component to fail.

- Use stress testing in message passing systems. I.e., design tests that create far more messages than you are likely to get in practice to reveal timing problems.

- Where several components interact through shared memory, design tests that vary the order in which these components are activated to attempt to show implicit assumptions made by the programmer about the order in which the shared data is created and used.

System Testing

In system tasting, all components are integrated and tested as one system. It checks that components are compatible and transfer the correct data when needed. There is some overlap with component testing. However, during system testing, reusable components, in-built systems and newly developed components can all be combined, even if they have been developed by different people. .

When you integrate components to create a system, some elements of system functionality only become apparent when you put the components together. Sometimes this emergent behavior is planned, but this behaviour might also be unplanned and unwanted. You have to develop tests that check that the system is only fulfilling its intended purpose. Therefore, system tests should focus on testing the interactions between the components and objects that make up a system. It is also a good idea to test reusable components to check that they work as expected

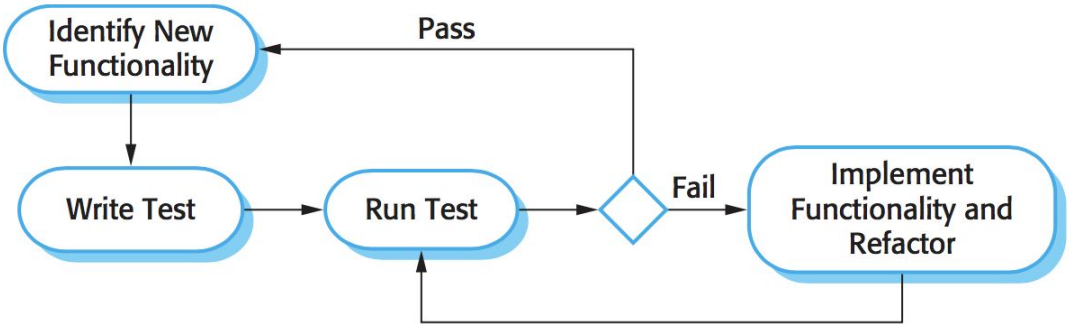
when used with new components. This interaction testing should discover those component bugs that are only revealed when other components in the system use a component.

It is difficult to know how much system testing is essential and when you should stop testing for most systems. Therefore, testing must be based on a subset of possible test cases. Software companies should ideally have policies for choosing this subset. Otherwise, they may be based on the experience of system usage and focus on testing the features of the operational system.

One particular form of system testing is automated system testing. This can often be tricky because it is based on predictions on what the output of a program is going to be. These predictions are built into a program of their own and the output thereof is then compared with the original program's output. As you can imagine, this could lead to complications if you have huge quantities of data, or data you cannot predict.

Test-Driven Development

Test-driven development is an approach to program development in which you develop code incrementally, along with a test for that increment. You interleave coding and testing. You don't move on to the next increment until the code that you have developed passes its test. This approach to development was introduced as part of agile methods such as Extreme Programming.



The test-driven development process (Sommerville 2010)

The diagram above shows the test-driven development process. The steps in the process are as follows:

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

2. 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. 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. Implement the functionality and rerun 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, move on to implementing the next chunk of functionality.

An automated testing environment, such as the JUnit environment that supports Java program testing, is essential for test-driven development. You have to be able to run every test each time that you add functionality or refactor the program as the code is developed in very small increments. Therefore, the tests are embedded in a separate program that runs the tests and invokes the system that is being tested. It is possible to run hundreds of separate tests in a few seconds using this approach.

There are several benefits to test-driven development:

1. **Better problem understanding**: Test-driven development helps programmers clarify their ideas of what a code segment is supposed to do. To write a test, you need to understand what is intended, as this understanding makes it easier to write the required code.

2. **Code coverage**: 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 been executed. Code is tested as it is written. Thus, defects are discovered early in the development process.

3. **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.

4. **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. The users of test-driven development suggest that it is hardly ever necessary to use an automated debugger in test-driven development.

5. **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 cost 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.

However, automated testing, which is fundamental to test-first development, dramatically reduces the costs of regression testing. Existing tests may be re-run quickly and cheaply. After making a change to a system in test-first development, all existing tests must run successfully before any further functionality is added. As a programmer, you can be confident that the new functionality that you have added has not caused or revealed problems with existing code.

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.

During the development process, there are two important distinctions between release testing and system testing:

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 goal of the release testing process is to convince the supplier of the system that it is good enough for use. If so, it can be released as a product or delivered to the customer. Release testing, therefore, has to show that the system delivers its specified functionality, performance, and dependability, and that it does not fail during normal use. It should take into account all of the system requirements, not just the requirements of the end-users of the system.

Release testing is usually a black-box testing process where tests are derived from the system specification. The system is treated as a black box whose behavior can only be determined by studying its inputs and the related outputs.

Requirements-Based Testing

Requirements should be testable. In other words, 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. Requirement-based testing is an approach to release testing where you consider each requirement and derive a set of tests for it. You are trying to demonstrate that the system has properly implemented its requirements. You normally have to write several tests to ensure that you have sufficiently covered the requirement. You should also maintain traceability records of your requirements-based testing, which link the tests to the specific requirements that are being tested.

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. When you use a scenario-based approach, you are normally testing several requirements within the same scenario. Therefore, as well as checking individual requirements, you are also checking that combinations of requirements do not cause problems.

Performance Testing

Once a system has been completely integrated, it is possible to test for 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.

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 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.

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'.

Stress 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, system failure must 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.

User Testing

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

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. This means that the users can identify problems and issues that are not readily apparent to the development testing team.

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. It takes place after release testing. Acceptance implies that payment should be made for the system.

Compulsory Task

Follow these steps:

- Create a new text file called testing.txt inside this folder.
- Inside testing.txt, answer the following questions:
    - Why is it is not necessary for a program to be completely free of defects before it is delivered to its customers? Explain your answer.
    - Why can testing only detect the presence of errors, not their absence?
    - Some people argue that developers should not be involved in testing their own code and instead all testing should be the responsibility of a separate team. What are the pros and cons of testing being carried out by the developers themselves?
    - What is regression testing? How does the use of automated tests and a testing framework such as JUnit simplify regression testing?
    - 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 to system testing.

Optional Task

Follow these steps:

- Create a new program called myUnitTest.java inside this folder.

- Inside muUnitTest.java:
    - Create a method that accepts the year that the user was born as a parameter and calculates the user's age. Assume that the age is calculated using the formula age = currentYear - yearBorn. The method should either return an integer value that indicates how old the user is or an appropriate error message if the user has entered an invalid year of birth.

    - Read the appropriate documentation for creating a test using JUnit:
        - If you're using Eclipse: https://www.eclipse.org/eclipse/news/4.7.1a/#junit-5-support
        - If you're using IntelliJ IDEA: https://blog.jetbrains.com/idea/2016/08/using-junit-5-in-intellij-idea/
        - If you're using another development environment: https://junit.org/junit5/docs/current/user-guide/#running-tests-ide

    - See how to create a test using JUnit here: https://junit.org/junit5/docs/current/user-guide/#writing-tests

    - Write a test for the method that you have created.

Rate us

Share your thoughts

Hyperion strives to provide internationally-excellent course content that helps you achieve your learning outcomes.

Think the content of this task, or this course as a whole, can be improved or think we've done a good job?

Click here to share your thoughts anonymously.