Module 4: Testing Strategies and Techniques

Topics

Testing is one of the most important activities in the development life cycle of software. During testing, you can see if the software that you have been working on really does what you think it should do. Regardless of what theories you have about how the software should work, it is during the testing phase that you see what the software actually does. It is critical that you take advantage of this opportunity to make the testing as effective and efficient as possible. You want to learn as much as possible about the software during the test phase so that if it does not work as expected you can still change it before it is released and delivered to your customer.

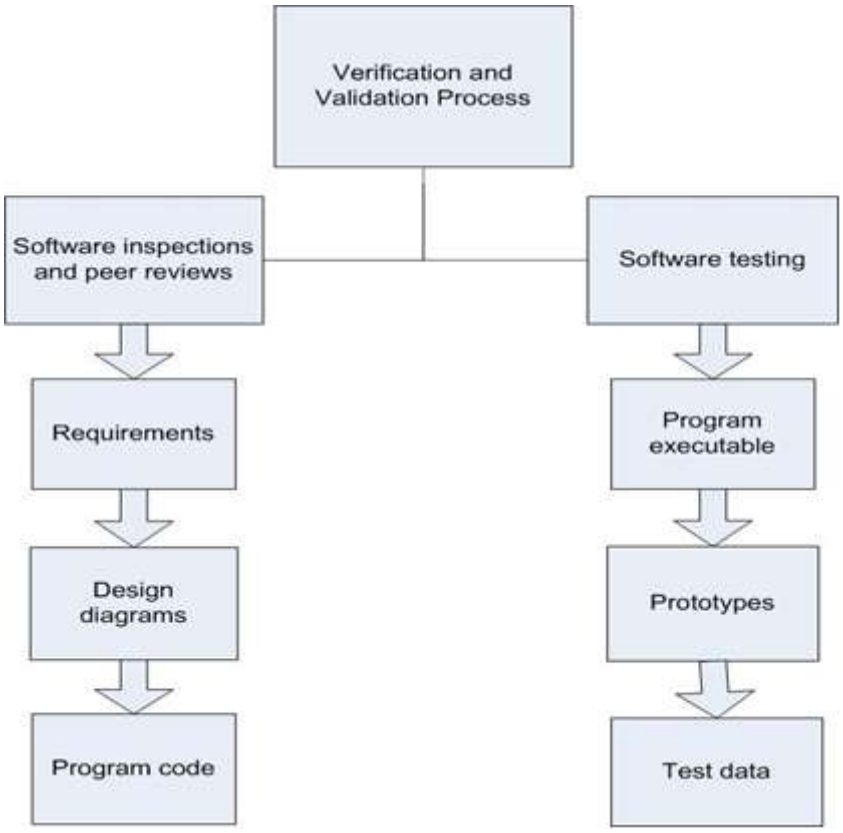
Testing is always necessary because no matter how careful you are, the software you develop will likely have defects that can cause the software to fail. One of the most effective ways to find and remove these defects is to run tests.

The problem is that testing can be very expensive, in terms of both labor and time. The tests have to be designed, developed, written, and applied to the software before you can determine how well the software works. In addition, it is almost impossible (except, perhaps, for the simplest of programs) to run all possible tests (exhaustive testing). No matter how many tests you are able to run, there will always be more for which you did not have time. Therefore, to be confident that your software will work as planned, you need to approach the testing very carefully, planning your tests for the maximum benefit.

1. Verification and Validation

Checking and analysis of the software is conducted by the software engineer and the test team at various stages of development, to ensure that it meets the functional requirements desired by the customer. The terms verification and validation are used to describe these checking and analysis processes. In figure 4.1 we demonstrate the two approaches used in the verification and validation process (Sommerville, 2004).

Figure 4.1
Verification and Validation Process



The first approach involves formal and informal reviews of the requirements and the design diagrams, and inspections of the code statements. In our discussion of requirements and design development, we have emphasized the importance of uncovering defects earlier rather than later in the development life cycle. The second approach, software testing, involves using the software executable and actual data inputs to validate the output results for correctness.

The ultimate goal of verification and validation of software components, before and after system integration, is to achieve a measure of confidence in the software's performance. Verification can be best described in the words of Barry Boehm as, "Are we building the product right?" (Boehm, 1981, p. 37). Validation can be best described in the words of Boehm as, "Are we building the *right product*?" (Boehm, 1981, p. 37).

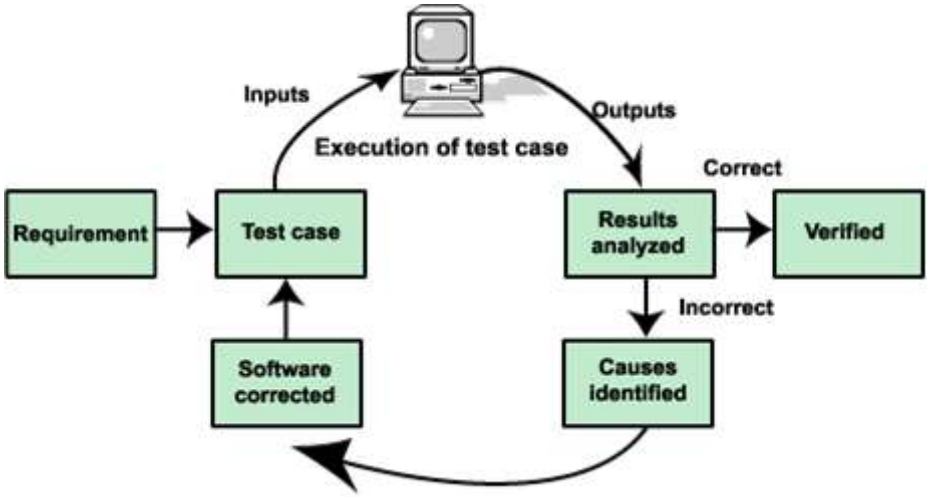
Verification is used throughout development to ensure that the product is being properly built and is coming together. In the context of the waterfall approach to software development, validation testing is used in the later phases of the development cycle to ensure that the final product behaves properly and meets all of its overall requirements.

Validation testing of software can be performed at different phases of development on executable code. It is common for validation testing to commence using prototypes in order to develop the initial test cases. Because software is part of a much larger computing system, testing is conducted before and after integration with the other components of the system. These two types of phases are referred to as *unit* and *integration* testing. Historically, integration testing was referred to as *incremental testing*, where the software components were integrated and tested in a stepwise manner until the entire system was assembled. For our discussion, we will refer to integration testing as the stage in which we integrate the end product of the software component with the hardware components to complete the system.

Please note that software testing is only one element of the overall verification and validation process, as we saw in figure 4.1. Testing ensures that the software will perform with a degree of confidence and the customer's expectations will be met in the delivered product. The customer has expectations as to how the software will perform with a level of confidence in the correctness of the software. It is the software engineer's job to deliver a product that meets these expectations. During the requirements analysis phase, the software engineer and the customer specify different types of requirements in the specifications. Specifically, performance and reliability requirements establish a goal for a level of

confidence that is to be achieved by the software and the overall system performance. The goals of testing are to find any errors and defects, and to prove that the level of confidence has been achieved by the software design. Realistically, it should be noted that comprehensive testing will uncover errors and defects, but may not achieve the goal of proving 100-percent correctness of the software. The common approach to software testing used in the industry is illustrated in figure 4.2 below.

Figure 4.2
Software Testing Process



The scope of validation testing for software and systems is broad, and this section is not meant to be comprehensive. We will concentrate on addressing the major issues involved in the validation testing phase of software development.

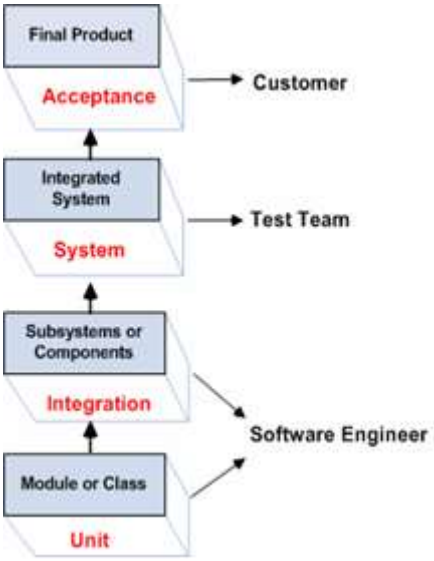
Software Quality Consideration

In module 2, we mentioned validating the requirements by providing a set of preliminary test cases before any code is written. In module 3 we mentioned that the software engineer's goal should be to design for quality by considering the attributes to satisfy the functionality, usability, reliability, performance, and supportability (FURPS) requirements. In table 3.2 we outlined a list of attributes that need to be considered during design to satisfy each of the FURPS requirements. The software engineer uses these design attributes and the preliminary set of test cases to develop a set of tests to measure the quality in the software.

Types of Tests

Software is normally a part of a much larger system, so we will mention the various types of software and system-level tests. Software testing commences during the earlier phases of development and can be described as testing "in the small." If you recall our discussion in module 3 on the different methodologies followed during software analysis and design, the smallest unit, according to the functional approach, is the module or subprogram. Following the object-oriented approach, the smallest unit is considered to be at the class level. As we integrate all the smaller units together, we form a subsystem or component. Once all the components are built, we integrate them into the environment in which they will execute. Testing is conducted during all phases, from conception of code to integration into a working system that is ready for delivery to the customer. Figure 4.3 represents the types of testing at the different stages of software development and identifies the party who performs each test.

Figure 4.3
Different Types of Tests



Unit testing is performed by the original software engineers during the coding stage of the module or class. At the unit level it is necessary to develop a set of drivers or stubs in order to conduct testing. A small code segment or driver is developed to test the functionality of the module or to exercise the interface of the class. At times it may also be necessary to develop a placeholder for a subordinate module; a stub is used in this case. When you develop drivers and stubs, they should be kept as simple as possible because they are not released with the final product.

Integration testing brings all the code pieces together into a cohesive whole to form the software architecture. During the integration testing phase, errors and defects may be uncovered that are relative to the interfacing of the modules, classes, subsystems, and components. Once the software subsystems or components are integrated and tested to achieve a level of confidence in the correctness of the software, they are integrated with the hardware components to form the end-state system architecture. Integration testing at the software level is typically conducted by the original software engineers who were involved in the software design.

Ideally, **system testing** is performed by an independent test team on the cohesive whole, all the hardware and the software components working together. Several types of tests are performed during system testing, including:

- recovery
- security
- stress
- performance

Software test cases are designed to intentionally attempt to break the system to see if the software will detect and recover gracefully from faults. This type of testing is referred to as **recovery testing**.

Information is a valuable asset to companies and is manipulated and stored as part of many computer-based systems. We refer to information as sensitive when, if obtained and used by the wrong person, it can cause harm to one's identity, life, or property. This sensitive nature of information determines the amount of **security testing** that needs to be designed and conducted on software. Security tests are designed to intentionally attempt penetration into the system and detect and exploit vulnerabilities in the software. Once the system has been penetrated, the software code can be maliciously modified. Many companies will hire a third party to perform extensive security tests because the design of these tests requires extensive technical knowledge. However, the software engineer can design simple tests to exploit known vulnerabilities that exist in programming languages. For example, click here to read about the common buffer overflow vulnerability in the C and C++ programming language.

Acceptance testing is performed by the intended users of the software to determine if the product is performing at an acceptable level and meets the needs of the customer and the user. Acceptance testing is also referred to as *beta testing*, application and end-user testing. One form of acceptance testing is when a beta or trial version of the software is released and tested in the field, the intended environment in which the software will be used, before the software is formally delivered to the customer. The

experiences from using the beta version are gathered and evaluated by the software engineers and any necessary revisions are made before releasing the final version.

An acceptance test plan is developed between the software engineer and the customer during the requirements analysis phase. The acceptance plan is used to specify the acceptable criteria or test scenarios that need to be satisfied by the software. Acceptance tests are created from the use case scenarios that were developed during the software requirements analysis phase. The software engineer works with the customer to derive the appropriate tests to ensure that the end user receives the intended functionality from the software. Acceptance tests are a set of black-box test cases that validate the correct output is delivered by the software when it is given a specific input. We will discuss black-box testing more in section 2.

Overall Benefits

Many studies have been conducted by researchers and practitioners on the impact of the verification and validation process on the quality, reliability, and resources spent to develop software products. Christensen and Thayer (2001) have summarized how effective verification and validation practices can be in providing benefits to software projects in the following ways:

- provide a means to ensure that properties such as quality and reliability are considered in the planning stage and built into the final product
- expose development errors earlier in the life cycle
- supply important decision-making information to management
- reduce the total project costs
- secure the requirements

2. Approaches to Testing

Two approaches are used by software engineers to design software tests: structural and functional. Both testing approaches are used to develop a test plan and to design a comprehensive set of test cases. Structural testing is referred to as the *white-box* test approach; functional testing is commonly referred to as the *black-box* test approach. White-box testing is conducted during code development and black-box testing is ideally conducted by a test team when coding is complete. Both testing approaches are essential in validating the attained level of correctness of the software and compliance with requirements.

White-Box Testing

The white-box, or structured, testing approach is focused on the implementation details of the software's logic paths. Test cases are developed following certain paths through the logic that can detect faults in the hardware, the application code, the programming language, or the operating system. In white-box testing, you focus on a certain sequence of instructions. The sequence of instructions comprises the processes, control structures, and mathematical expressions that are implemented in the logic used in the code. The main idea is to test each logic branch, decision point path, and calculation. These tests are designed to follow the instruction execution logic from the entry point of the algorithm through the exit point of the algorithm (Beizer, 1984).

In module 3 we discussed two methodologies that can be followed for software analysis and design: structured and object-oriented. When we follow the structured methodology and we code the software, each call to a **subprogram**, which is also called a *subroutine* or a *function*, is a candidate for a test case. A subprogram can be one or more instructions in length. A structured software program typically consists of several subprograms.

In the object-oriented methodology to programming, we design each class to contain methods. In the software, each call to a class instance method is a good candidate for a test case. A method in a class contains one or more instructions of logic.

Decision-control structures are implemented in all programming languages as conditional branch instructions, and each path should be tested. The most common decision-control structures are:

- *if-then-else* constructs
- selection statements (commonly known as case in Java and C++)
- conditional loops

In white-box testing, we stress that every instruction and every decision direction must be executed at least once. The goal is to execute each logic path of the code to provide good coverage of the code statements in the software. To define independent execution paths, the software engineer needs to measure the logical complexity of the software. The logical complexity measure can then be used to define an independent set of execution paths (Pressman, 2001), which are called *basis paths*. To illustrate a white-box testing technique, we can examine a basis path test proposed by Tom McCabe that enables deriving a logical complexity measure for a structured design. The cyclomatic complexity is derived from mathematical graph theory as a measure of the complexity level of software modules (Craig and Jaskiel, 2002).

Example of Complexity Measure: Cyclomatic Complexity

Cyclomatic complexity (also known as McCabe complexity) equates the complexity of a component to the number of independent execution paths through that component.

Computing the cyclomatic complexity involves three steps:

- create a flow graph for the component
- count the number of edges (E) and the number of nodes (N) in the graph
- compute cyclomatic complexity according to V(G) = E - N + 2

Consider the following C program:

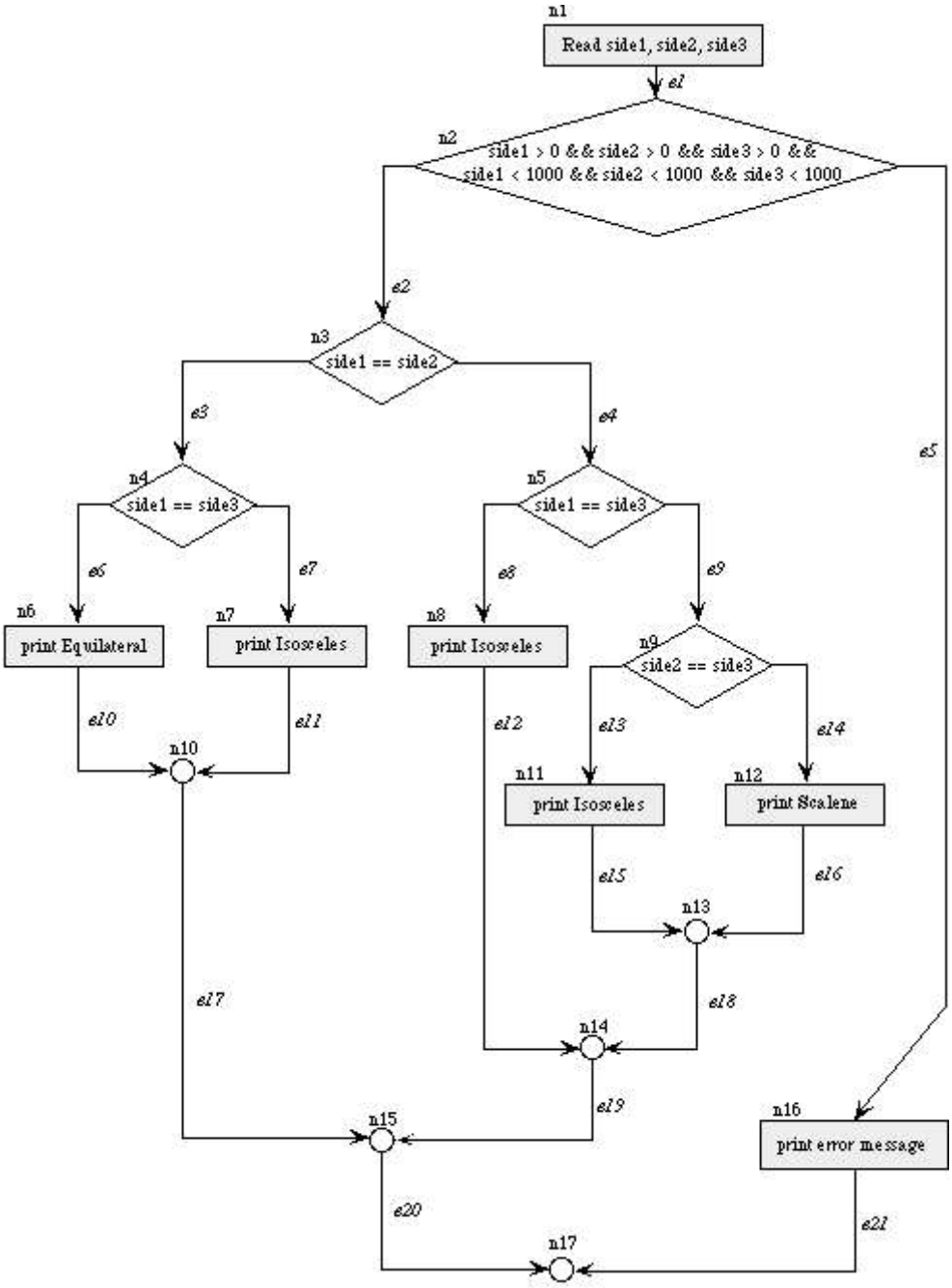
```
#include <stdio.h>
main()
{
int side1, side2, side3;
printf ( "This program will determine whether a triangle is\n");
printf ( "isosceles, equilateral, or scalene\n");

scanf( "%d %d %d", &side1, &side2, &side3 );

if (side1 > 0 &&
    side2 > 0 &&
    side3 > 0 &&
    side1 < 1000 &&
    side2 < 1000 &&
    side3 < 1000 )
    {if (side1 == side2)
        {if (side1 == side3)
            {printf ( "Equilateral\n");
            }
         else
            {printf ( "Isosceles\n");
            }
        }
     else
        {if (side1 == side3)
            {printf ( "Isosceles\n");
            }
         else
            {if (side2 == side3)
                {printf ( "Isosceles\n");
                }
             else
                {printf ( "Scalene\n");
                }
            }
         }
```

```
        }
    else
        {printf ("Illegal input value\n");
        }
}
```

Step 1: Draw the flow graph for the program.



Note that each node is labeled with ni and each edge is labeled with ei.

Step 2: Determine the number of nodes and the number of edges in the flow graph.

N = 17

E = 21

Step 3: Compute the cyclomatic complexity of the flow graph.

The cyclomatic complexity V(G) for this program is computed as follows:

V(G) = E - N + 2
= 21 - 17 + 2
= 6

The cyclomatic complexity of this component equals 6. V(G) represents the number of linearly independent execution paths through the software component's structure (Pressman, 2005). A higher cyclomatic complexity value will indicate the need for more white-box tests to be conducted on the software.

Black-Box Testing

With **black-box testing**, you develop your test cases based on the requirements of the component being tested. This way, you can make certain that all important requirements, and combinations of requirements, are validated to be working properly as specified in the test plan. Notice that this approach to testing is different from white-box testing. In fact, it is likely that if both techniques are applied, there will be many different test cases executed, which is a good thing, because it provides for more extensive testing of the software.

The black-box testing approach focuses on the *functional requirements of the software*, i.e., the FURPS requirements. This software testing approach attempts to find:

- incorrect or missing functions
- interface errors
- errors in data structure or external database access
- performance errors
- initialization and termination errors

The black-box test assumes that the tester has little or *no knowledge of the internals of the code*. The tests are designed based on the requirements' description of what the software is supposed to do. Each black-box test case focuses on software's input and the expected output. When the test is performed, if the expected output is correct, then the test is successful in validating the functional requirement of the software. Following an object-oriented approach, the *class* is a good unit for testing. We can write a test case to exercise each public method defined within the class scope.

There are several different types of black-box tests that should be conducted; we will discuss each type in the following sections. The types of black-box test cases are:

- equivalence partitioning
- boundary-value analysis
- data validation

Equivalence Partitioning

Equivalence partitioning tries to limit the test cases to a reasonable number. It is impossible to test every case because the combinations are too numerous, but a reasonable subset has a good chance of finding most errors.

Equivalence classes are determined based on the input. If the input is a range of values, one valid class (within the range) and two invalid classes (on either side of the range) are sufficient. If the input is a set of values, one valid class for each member of the set and one invalid class (for nonmembers) are sufficient. In the case of Boolean-type input, one valid class and one invalid class should suffice.

For example, if the valid input range for an application is 1 to 1,000, the following test cases can be used:

- 500 valid case
- −10 invalid case, below the range
- 1500 invalid case, above the range

Boundary-Value Analysis

Boundary-value analysis takes advantage of the fact that more errors occur on the boundaries of valid inputs than elsewhere. We derive the test cases from the edges of the partitions of the input value range.

Given a range of input values, we test the minimum and maximum values of the range, and just above and below the range. For example, if the valid range for the numeric inputs to the software is 1 to 1,000, we can derive the following test cases:

- 1 valid, minimum value
- 1000 valid, maximum value
- 0 invalid, below the minimum range
- 1001 invalid, above the maximum range

Test cases should also be designed to test the maximum boundary for program constructs—for example, the array construct. Design two tests, one to test the maximum allowable boundary and another to test above the boundary. If we use an array boundary of 100, the following two test cases should suffice:

- enter 100 input entries maximum allowed
- enter 100 + 1 additional entry invalid, maximum exceeded

If the input or output is an ordered set, concentrate on the first and last elements. For databases, develop test cases to access the databases on the maximum allowable entries.

Data Validation

Data-validation testing concentrates on protection of the input data values when entered into the software by non-technical users and intruders. The software engineer prepares test cases using any input that is considered unexpected or not valid. The following list provides the general guidelines that are used in setting up these test cases:

- If the software is expecting a string input, then test for an integer value.
- If the string input has a maximum length of 15 characters, then prepare a test case with 16 characters.
- Test for the possibilities of bad input data in all the data formats.

In black-box testing, test cases are developed to ensure that the software conforms to the requirements by producing the desired output when given a certain set of inputs. The software engineer writes the series of black-box test cases from the previously defined software requirements.

Black-Box Test Case Design

The key to planning effective and efficient testing is designing a good set of **test cases**. A test case is a specific set of inputs that will be read by your software to challenge its behavior. It is expected that, for each test case, your software will produce the appropriate answers as output. The selection of test cases should be as comprehensive and challenging as possible to the software so that your software will be exercised as much as possible. However, there exists the possibility that budget limits may restrict what you can test, so use good judgment in selecting and designing test cases.

Consider the following software function:

```
int hypo (int side1, int side2)
{
if (side1 > 0 && side2 > 0)
      {
      x = side1 * side1;
      y = side2 * side2;
      hypotenuse = sqrt(x+y);
      return hypotenuse;
      }
}
```

We can define several test cases as follows in table 4.1:

Table 4.1
Example of Black-Box Test Cases

Test Case			Expected result	Actual result
#	Side1	Side2		
1	2	4	4.47	
2	20	15	25	
3		5		

Note that each set of input values forms a single test case, with an expected result. When the tests are actually run, the results are recorded and compared with the expected results to see if the software satisfies its requirement and executes correctly.

Diagrams and Test Case Design

In module 3 we presented several types of diagrams that are created during the requirements analysis and design phases of the development cycle of software. Diagrams present a pictorial view of software behavior and logic. By tracing through the diagrams, we can design a complete set of white-box test cases. For example, the state transition diagram depicts the behavioral paths and the data flow diagram depicts the inputs and expected outputs for each software component. Test cases can be derived from the class diagram because it shows all the interface methods that can be invoked to manipulate the class data.

Software engineers use diagrams as a starting point to develop the white-box and black-box test cases. In the next section, we will discuss writing test cases to validate the user interface. In user interface testing, the use case diagram we created during the requirements analysis phase is used to create a set of black-box test cases to evaluate each scenario.

3. User Interface Testing

As we discussed in module 3, the design of a graphical user interface plays an important role in the usability of the software to produce correct results. When we test the user interface, the goal is to verify the correctness of the manipulative objects in getting the result we expect from our software. A set of user interface test cases can easily be derived from the scenarios that were developed during the requirements analysis phase. Interface testing based on previously defined user scenarios is commonly referred to as *scenario-based testing*.

Scenario-based Testing

The user scenarios outlined during the requirements elicitation phase are used to derive an initial set of black-box test cases in the requirements validation phase to validate the correctness of the user interface. The following steps (Leffingwell and Widrig, 2003) summarize writing a set of test cases for the user interface:

- select a use case scenario
- identify at least one test case for each scenario
- identify the necessary conditions for execution of the test case
- list the data values to be used as input
- list the expected result from a given set of data values

Example User Interface Test Case

In module 2 we presented a set of scenarios for the Self-Service Checkout System in table 2.3, the new purchase order use case. In table 4.2 we present the scenarios again so that we can demonstrate how to derive a set of black-box test cases.

Table 4.2
User Scenarios for the New Purchase Order Use Case

User scenario 1 – start the order

1. Customer selects either the English or Spanish language.

2. System displays the "Begin Order" option in English or the Spanish language.

User scenario 2 – set up the order

1. Customer selects the "Begin Order" option when ready to start the order.

2. System responds by displaying a message with instructions to scan frequent shopper discount card.

User scenario 3 – scan discount card

1. Customer places the discount card over the scanner.

2. System responds by reading the barcode imprinted on the card.

3. System displays welcome message and instructions to scan an item.

In table 4.3, we illustrate a set of black-box test cases to test the user interface using the use case scenarios for beginning a new purchase order for the Self-Service Checkout System.

Table 4.3
Definition of Test Cases for New Purchase Order Scenario

1.0 New Purchase Order Test Cases

1.1 Start the order

Precondition: System booted and in ready state.

1.1.1 Input: Touch the "English" button.

Output: The "Begin Order" button appears on the screen.

1.1.2 Input: Touch the "Espanol" button.

Output: The "Comenzar aqui" button appears on the screen.

1.2 Set up the order

1.2.1 Input: Touch the "Begin Order" button.

Output: The message "Place card over the window" and an` animated picture showing a person scanning a card appears in the center of the screen.

1.2.2 Input: Touch the "Comenzar aqui" button.

Output: The message "Pasar la tarjetz sobre la ventanz" and an animated picture showing a person scanning a card appears in the center of the screen.

1.3 Scan discount card

Precondition: The scanner is on and ready to read.

1.3.1 Input: Place a discount card for a valid customer, Christine Jones, with an active record in the database over the window.

Output: The message "Welcome Christine" or "Bienvenido Cristina" appears on the screen along with an animated cartoon illustrating a person placing a card over the scanner's window.

1.3.2 Input: Place a discount card for an invalid customer, David Johnson, with an inactive record in the database over the scanner's window.

Output: The message "Sorry no record" or "Lo sients no existe registro" is displayed along with an animated cartoon illustrating a person placing an item on the scanner's window.

Usability Studies

The usability of the software's interface is critical to the safe operation of the computer-based system that relies heavily on human-computer interaction. Testing efforts concerned with the usability of the interface design should be conducted with the intended user community. Sampling the intended user community will help to ensure the correct design and integrity of the interface.

Usability studies can be conducted either formally or informally. Informal testing is conducted by the original software engineers. Formal testing is conducted in existing laboratories by independent testers, and simulates the environment of the projected user community. Both types of usability testing require that the subjects be selected to represent a cross section of the user community. These subjects are then asked to use the interface under the watchful eye of the testers, and sometimes these sessions are even videotaped.

Testing is not cheap and the selection of informal or formal usability tests is dependent on the nature and criticality of the software and the budget allocation. It is recommended by software engineering practitioners that software interfaces used in mission- or safety-critical systems be tested to some degree of reliability both informally and formally.

The results of usability studies benefit the software engineers in identifying user problems when interfacing with the system. User problems can be caused by the layout or by misrepresentations of the objects on the interface. The ultimate goal of the study is to uncover any conditions causing interface error that may not have been detected during the black-box testing that was originally conducted on the graphical user-interface.

Usability testing is conducted at the end of the development cycle during acceptance testing when the user interface has been completed. The drawback to conducting usability testing at this later phase is that when inefficiencies are discovered they can be very expensive to fix. To address this problem, prototypes of graphical user interfaces (GUIs) can be created during the earlier phases of the development cycle. Developing GUI prototypes at earlier stages can be used in formal usability tests that are conducted in controlled laboratories to discover potential problems before the interface is actually built (Craig and Jaskiel, 2002).

4. Traceability and Documentation

Traceability of Test Cases

In Module 2, section 3-C, we discussed the cross-reference of the system and software requirements to the identified functionality of the software and hardware components of the system design. When we have identified and developed the test cases to validate each requirement, we can extend the traceability matrix to include the corresponding test case identifier, as shown in table 4.4:

Table 4.4
Traceability Matrix

Category	Description	System Req. #	Use Case #	Software Req. #	Test Case #	Pass/Fail
New Order	Language selection	1.0	1.0	1.1	1.1.1	

During testing, it is imperative that each requirement be verified and validated. The traceability matrix ensures a level of coverage of the required functionality has been successfully implemented in the software in all phases of the development cycle. The completed traceability matrix is delivered to the customer during acceptance testing.

Documentation of Test Cases

Test cases can be documented in a number of ways: a simple spreadsheet, a formal document, or automatically with a CASE testing tool. We will recommend and discuss the *IEEE Standard for Software Test Documentation*, *IEEE Std 829-1998*.

The *IEEE Std 829-1998* provides a template to fully describe each test case and can be used easily by independent testers. Each test case description can be summarized by providing the following information:

- test case identifier
- input specification
- output specification
- special environment conditions
- special procedural requirements
- execution procedure steps
- dependencies

For historical purposes, the results of tests must be documented and stored in a database. These records should be easily accessible by the maintainers of the software and made available to investigators as background information in case of an incident of software failure.

References

Beizer, Boris. *Software system testing and quality assurance*. New York: Van Nostrand Reinhold, 1984.

Boehm, Barry. *Software Engineering Economics*. Prentice-Hall, 1981.

Christensen, Mark J., and Richard H. Thayer. *The project manager's guide to software engineering's best practices*. Los Alamitos, CA: IEEE Computer Society Press, 2001.

Craig, Rick, and Stephan Jaskeil. *Systematic Software Testing*. Boston: Artech House Publishers, 2002.

Institute of Electrical and Electronics Engineers. IEEE Standard for Software Test Documentation, IEEE Std 829-1998. New York: IEEE, 1998.

Leffingwell, Dean, and Don Widrig. *Managing software requirements: A use case approach* (second edition). Boston: Addison-Wesley, 2003.

Pressman, Roger S. *Software engineering: A practitioner's approach* (fifth edition). New York: McGraw-Hill, 2001.

Pressman, Roger S. *Software engineering: A practitioner's approach* (sixth edition). New York: McGraw-Hill, 2005.

Sommerville, Ian. *Software engineering* (seventh edition). England: Pearson Education Addison Wesley, 2004.

Report broken links or any other problems on this page.