Testing Overview and Black-Box Testing Techniques

Software testing is an important technique for assessing the quality of a software product. In this chapter, we will explain the following:

- the basics of software testing, a verification and validation practice, throughout the entire software development lifecycle
- the two basic techniques of software testing, black-box testing and white-box testing
- six types of testing that involve both black- and white-box techniques.
- strategies for writing fewer test cases and still finding as many faults as possible
- using a template for writing repeatable, defined test cases

1 Introduction to Testing

Software testing is the *process of analyzing a software item to detect the differences between existing and required conditions (that is, bugs) and to evaluate the features of the software item* [9, 12]. Software testing is an activity that should be done throughout the whole development process [3].

Software testing is one of the "verification and validation," or V&V, software practices. Some other V&V practices, such as inspections and pair programming, will be discussed throughout this book. V*erification* (the first V) is *the process of evaluating a system or component to determine whether the products of a given development phase satisfy the conditions imposed at the start of that phase* [11]. Verification activities include testing and reviews. For example, in the software for the Monopoly game, we can verify that two players cannot own the same house. *Validation* is the *process of evaluating a system or component during or at the end of the development process to determine whether it satisfies specified requirements* [11]. At the end of development validation (the second V) activities are used to evaluate whether the features that have been built into the software satisfy the customer requirements and are traceable to customer requirements. For example, we validate that when a player lands on "Free Parking," they get all the money that was collected. Boehm [4] has informally defined verification and validation as follows:

Verification: Are we building the product <u>right</u>?
Through verification, we make sure the product behaves the way we want it to. For example, on the left in Figure 1, there was a problem because the specification said that players should collect $200 if they land on <u>or</u> pass Go. Apparently a programmer implemented this requirement as if the player had to <u>pass</u> Go to collect. A test case in which the player landed on Go revealed this error.

Validation: Are we building the <u>right</u> product?
Through validation, we check to make sure that somewhere in the process a mistake hasn't been made such that the product build is not what the customer asked for; validation always involves comparison against <u>requirements</u>. For example, on the right in Figure 1, the customer specified requirements for the Monopoly game – but the programmer delivered the game of Life. Maybe the programmer thought he or she

"knew better" than the customer that the game of Life was more fun than Monopoly and wanted to "delight" the customer with something more fun than the specifications stated. This example may seem exaggerated – but as programmers we can miss the mark by that much if we don't listen well enough or don't pay attention to details – or if we second guess what the customer says and think we know better how to solve the customer's problems.

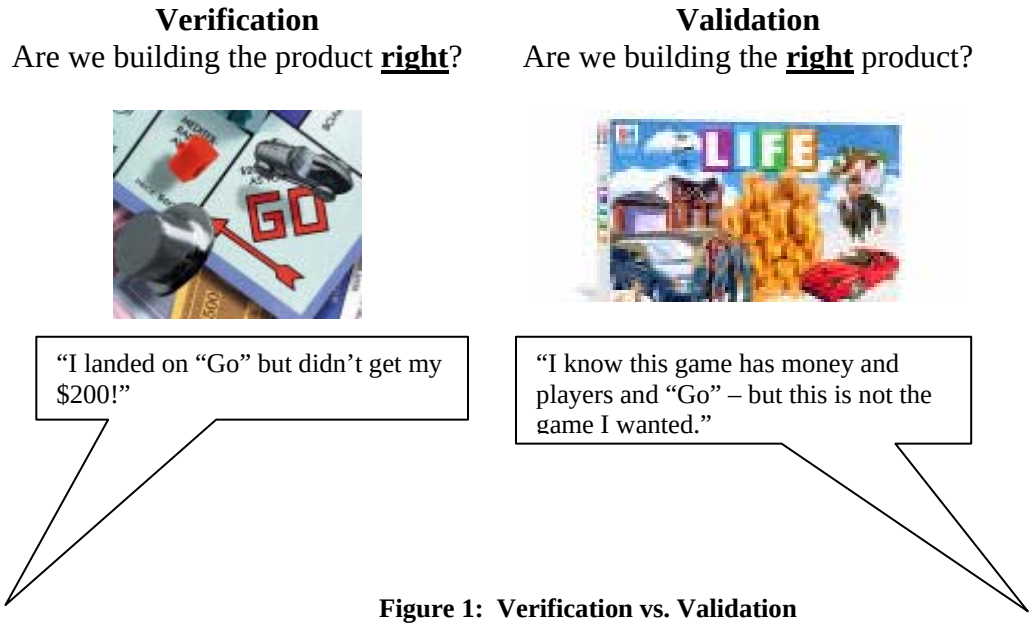


Figure 1: Verification vs. Validation

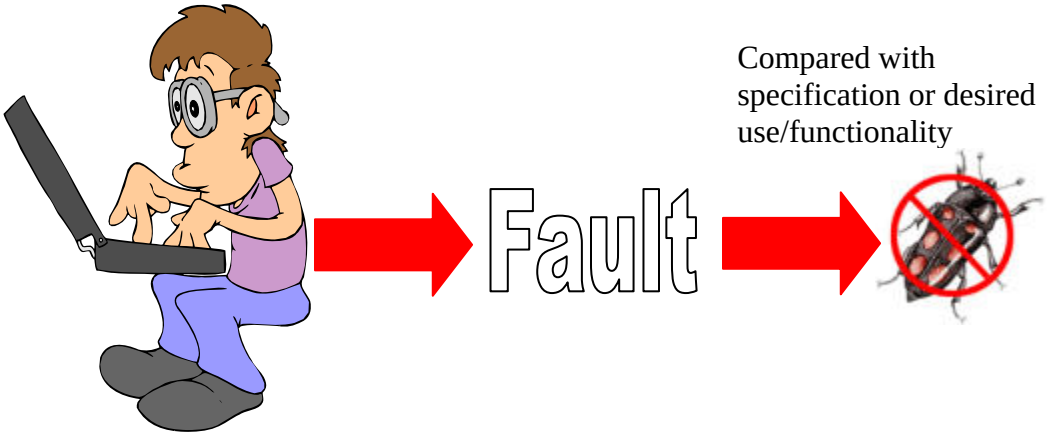
Both of Boehm's informal definitions use the term "right." But what is "right"? In software we need to have some kind of standard or specification to measure against so that we can identify correct results from incorrect results. Let's think about how the incorrect results might originate. The following terms with their associated definitions [11] are helpful for understanding these concepts:

- **Mistake** – a human action that produces an incorrect result.
- **Fault [or Defect]** – an incorrect step, process, or data definition in a program.
- **Failure** – the inability of a system or component to perform its required function within the specified performance requirement.
- **Error** – the difference between a computed, observed, or measured value or condition and the true, specified, or theoretically correct value or condition.
- **Specification** – a document that specifies in a complete, precise, verifiable manner, the requirements, design, behavior, or other characteristic of a system or component, and often the procedures for determining whether these provisions have been satisfied.

A *mistake* committed by a person becomes a *fault* (or defect) in a software artifact, such as the specification, design, or code. This fault, unless caught, propagates as a defect in the executable code. When a defective piece of code is executed, the fault may become a visible anomaly (a variance from the specification or desired behavior) and a *failure* is

observed. Otherwise, the fault remains latent. Testing can reveal failures, but it is the faults that must be found and removed [3]; finding a fault (the cause of a failure) can be time consuming and unpredictable. *Error* is a measure of just how incorrect the results are.

The progression of a software failure is demonstrated in Figure 2. A purpose of testing is to cause failures in order to make faults visible [10] so that the faults can be fixed and not be delivered in the code that goes to customers. Another purpose of testing is to assess the overall quality level of the code. For example, a test team may determine a project with too many high-severity defects should be sent back to development for additional work to improve the quality before the testing effort should continue. Or, the management may have a policy that no product can ship if testing is continuing to reveal high-severity defects.



A programmer makes a **mistake**.

The mistake manifests itself as a **fault**[1] [or defect] in the program.

A **failure** is observed if the fault [or defect] is made visible. Other faults remain **latent** in the code until they are observed (if ever).

Figure 2: The progression of a software failure. A purpose of testing is to expose as many failures as possible before delivering the code to customers.

1.1 The Economics of Software Testing

In software development, there are costs associated with testing our programs. We need to write out test plan and our test cases, we need to set up the proper equipment, we need to systematically execute the test cases, we need to follow up on problems that are identified, and we need to remove most of the faults we find. Actually, sometimes we can find low-priority faults in our code and decide that it is too expensive to fix the fault

[1] The IEEE does not define *defect* however, the term defect is considered to be synonymous with fault.

because of the need to redesign, recode, or otherwise remove the fault. These faults can remain latent in the product through a follow-on release or perhaps forever.

For faults that are not discovered and removed before the software has been shipped, there are costs. Some of these costs are monetary, and some could be significant in less tangible ways. Customers can lose faith in our business and can get very angry. They can also lose a great deal of money if their system goes down because of our defects. (Think of the effect on a grocery store that can't check out the shoppers because of its "down" point-of-sale system.) And, software development organizations have to spend a great deal of money to obtain specific information about customer problems and to find and fix the cause of their failures. Sometimes, programmers have to travel to customer locations to work directly on the problem. These trips are costly to the development organization, and the customers might not be overly cheerful to work with when the programmer arrives. When we think about how expensive it is to test, we must also consider how expensive it is to not test – including these intangible costs as well as the more obvious direct costs.

We also need to consider the relative risk associated with a failure depending upon the type of project we work on. Quality is much more important for safety- or mission-critical software, like aviation software, than it is for video games. Therefore, when we balance the cost of testing versus the cost of software failures, we will test aviation software more than we will test video games. As a matter of fact, safety-critical software can spend as much as three to five times as much on testing as all other software engineering steps combined [17]!

To minimize the costs associated with testing and with software failures, a goal of testing must be to uncover as many defects as possible with as little testing as possible. In other words, we want to write test cases that have a high likelihood of uncovering the faults that are the most likely to be observed as a failure in normal use. It is simply impossible to test every possible input-output combination of the system; there are simply too many permutations and combinations. As testers, we need to consider the economics of testing and strive to write test cases that will uncover as many faults in as few test cases as possible. In this chapter, we provide you with disciplined strategies for creating efficient sets of test cases – those that will find more faults with less effort and time.

1.2 The Basics of Software Testing

There are two basic classes of software testing, black box testing and white box testing. For now, you just need to understand the very basic difference between the two classes, clarified by the definitions below [11]:

- *Black box testing* (also called functional testing) is *testing that ignores the internal mechanism of a system or component and focuses solely on the outputs generated in response to selected inputs and execution conditions.*
- *White box testing* (also called structural testing and glass box testing) is *testing that takes into account the internal mechanism of a system or component*.

The classes of testing are denoted by colors to depict the opacity of the testers of the code. With *black box testing*, the software tester does not (or should not) have access to the source code itself. The code is considered to be a "big black box" to the tester who can't see inside the box. The tester knows only that information can be input into to the black box, and the black box will send something back out. Based on the requirements knowledge, the tester knows what to expect the black box to send out and tests to make sure the black box sends out what it's supposed to send out. Alternatively, *white box testing* focuses on the internal structure of the software code. The white box tester (most often the developer of the code) knows what the code looks like and writes test cases by executing methods with certain parameters. In the language of V&V, black box testing is often used for validation (are we building the right software?) and white box testing is often used for verification (are we building the software right?). This chapter focuses on black box testing.

All software testing is done with executable code. To do so, it might be necessary to create scaffolding code. *Scaffolding* is defined *as computer programs and data files built to support software development and testing but not intended to be included in the final product* [11]. Scaffolding code is code that simulates the functions of components that don't exist yet and allow the program to execute [16]. Scaffolding code involves the creation of stubs and test drivers. *Stubs* are modules that simulate components that aren't written yet, formally defined as a *computer program statement substituting for the body of a software module that is or will be defined elsewhere* [11]. For example, you might write a skeleton of a method with just the method signature and a hard-coded but valid return value. *Test drivers* are defined as a *software module used to involve a module under test and often, provide test inputs, controls, and monitor execution and report test results* [11]. Test drivers simulate the calling components (e.g. hard-coded method calls) and perhaps the entire environment under which the component is to be tested [1]. Another concept is mock objects. *Mock objects* are temporary substitutes for domain code that emulates the real code. For example, if the program is to interface with a database, you might not want to wait for the database to be fully designed and created before you write and test a partial program. You can create a mock object of the database that the program can use temporarily. The interface of the mock object and the real object would be the same. The implementation of the object would mature from a dummy implementation to an actual database.

1.4 Six Types of Testing

There are several types of testing that should be done on a large software system. Each type of test has a "specification" that defines the correct behavior the test is examining so that incorrect behavior (an observed failure) can be identified. The six types and the origin of specification (what you look at to develop your tests) involved in the test type are now discussed. There are two issues to think about in these types of testing – one is the **opacity** of the tester's view of the code (is it white or black box testing). The other issue is **scale** (is the tester examining a small bit of code or the whole system and its environment).

1. Unit Testing
   Opacity: White box testing
   Specification: Low-level design and/or code structure

Unit testing is the *testing of individual hardware or software units or groups of related units* [11]. Using white box testing techniques, testers (usually the developers creating the code implementation) verify that the code does what it is intended to do at a very low structural level. For example, the tester will write some test code that will call a method with certain parameters and will ensure that the return value of this method is as expected. Looking at the code itself, the tester might notice that there is a branch (an `if-then`) and might write a second test case to go down the path not executed by the first test case. When available, the tester will examine the low-level design of the code; otherwise, the tester will examine the structure of the code by looking at the code itself. Unit testing is generally done within a class or a component.

2. Integration testing
   Opacity: Black- and white-box testing
   Specification: Low- and high-level design

Integration test is *testing in which software components, hardware components, or both are combined and tested to evaluate the interaction between them* [11]. Using both black and white box testing techniques, the tester (still usually the software developer) verifies that units work together when they are integrated into a larger code base. Just because the components work individually, that doesn't mean that they all work together when assembled or integrated. For example, data might get lost across an interface, messages might not get passed properly, or interfaces might not be implemented as specified. To plan these integration test cases, testers look at high- and low-level design documents.

3. Functional and system testing
   Opacity: Black-box testing
   Specification: high-level design, requirements specification

Using black box testing techniques, testers examine the high-level design and the customer requirements specification to plan the test cases to ensure the code does what it is intended to do. *Functional testing* involves ensuring that the functionality specified in the requirement specification works. System testing involves putting the new program in many different environments to ensure the program works in typical customer environments with various versions and types of operating systems and/or applications. *System testing* is *testing conducted on a complete, integrated system to evaluate the system compliance with its specified requirements* [11]. Because system test is done with a full system implementation and environment, several classes of testing can be done that can examine non-functional properties of the system. It is best when function and system testing is done by an unbiased, independent perspective (e.g. not the programmer) [3].

- *Stress testing – testing conducted to evaluate a system or component at or beyond the limits of its specification or requirement* [11]. For example, if the team is developing software to run cash registers, a non-functional requirement might state that the server can handle up to 30 cash registers looking up prices simultaneously. Stress testing might occur in a room of 30 actual cash registers running automated test transactions repeatedly for 12 hours. There also might be

a few more cash registers in the test lab to see if the system can exceed its stated requirements.

- *Performance testing – testing conducted to evaluate the compliance of a system or component with specified performance requirements* [11]. To continue the above example, a performance requirement might state that the price lookup must complete in less than 1 second. Performance testing evaluates whether the system can look up prices in less than 1 second (even if there are 30 cash registers running simultaneously).
- *Usability testing – testing conducted to evaluate the extent to which a user can learn to operate, prepare inputs for, and interpret outputs of a system or component*. While stress and usability testing can be and is often automated, usability testing is done by human-computer interaction specialists that observe humans interacting with the system.

4. Acceptance testing
   Opacity: Black-box testing
   Specification: requirements specification

After functional and system testing, the product is delivered to a customer and the customer runs black box acceptance tests based on their expectations of the functionality. A*cceptance testing* is *formal testing conducted to determine whether or not a system satisfies its acceptance criteria (the criteria the system must satisfy to be accepted by a customer) and to enable the customer to determine whether or not to accept the system* [11]. These tests are often pre-specified by the customer and given to the test team to run before attempting to deliver the product. The customer reserves the right to refuse delivery of the software if the acceptance test cases do not pass. However, customers are not trained software testers. Customers generally do not specify a "complete" set of acceptance test cases. Their test cases are no substitute for creating your own set of functional/system test cases. The customer is probably very good at specifying at most one good test case for each requirement. As you will learn below, many more tests are needed. Whenever possible, we should run customer acceptance test cases ourselves so that we can increase our confidence that they will work at the customer location.

5. Regression testing
   Opacity: Black- and white-box testing
   Specification: Any changed documentation, high-level design

Throughout all testing cycles, *regression* test cases are run. *Regression testing* is *selective retesting of a system or component to verify that modifications have not caused unintended effects and that the system or component still complies with its specified requirements* [11]. Regression tests are a subset of the original set of test cases. These test cases are re-run often, after any significant changes (bug fixes or enhancements) are made to the code. The purpose of running the regression test case is to make a "spot check" to examine whether the new code works properly and has not damaged any previously-working functionality by propagating unintended side effects. Most often, it is impractical to re-run all the test cases when changes are made. Since regression tests are run throughout the development cycle, there can be white box regression tests at the

unit and integration levels and black box tests at the integration, function, system, and acceptance test levels.

The following guidelines should be used when choosing a set of regression tests (also referred to as the regression test *suite*):

- Choose a representative sample of tests that exercise all the existing software functions;
- Choose tests that focus on the software components/functions that have been changed; and
- Choose additional test cases that focus on the software functions that are most likely to be affected by the change.

A subset of the regression test cases can be set aside as smoke tests. A *smoke test* is a *group of test cases that establish that the system is stable and all major functionality is present and works under "normal" conditions* [6]. Smoke tests are often automated, and the selection of the test cases are broad in scope. The smoke tests might be run before deciding to proceed with further testing (why dedicate resources to testing if the system is very unstable). The purpose of smoke tests is to demonstrate stability, not to find bugs with the system.

6. Beta testing
   Opacity: Black-box testing
   Specification: None.

When an advanced partial or full version of a software package is available, the development organization can offer it free to one or more (and sometimes thousands) potential users or *beta testers*. These users install the software and use it as they wish, with the understanding that they will report any errors revealed during usage back to the development organization. These users are usually chosen because they are experienced users of prior versions or competitive products. The advantages of running beta tests are as follows [8]:

- *Identification of unexpected errors* because the beta testers use the software in unexpected ways.
- *A wider population search for errors* in a variety of environments (different operating systems with a variety of service releases and with a multitude of other applications running).
- *Low costs* because the beta testers generally get free software but are not compensated.

The disadvantages of beta testing are as follows [8]:

- *Lack of systematic testing* because each user uses the product in any manner they choose.
- *Low quality error reports* because the users may not actually report errors or may report errors without enough detail.
- *Much effort is necessary to examine error reports* particularly when there are many beta testers.

Throughout all testing cycles, *regression* test cases are run. *Regression testing* is *selective retesting of a system or component to verify that modifications have not caused unintended effects and that the system or component still complies with its specified requirements.*

These six levels of testing are summarized in Table 1.

Testing Type	Specification	General Scope	Opacity	Who generally does it?
Unit	Low-Level Design Actual Code Structure	Small unit of code no larger than a class	White Box	Programmer who wrote code
Integration	Low-Level Design High-Level Design	Multiple classes	White Box Black Box	Programmers who wrote code
Functional	High Level Design	Whole product	Black Box	Independent tester
System	Requirements Analysis	Whole product in representative environments	Black Box	Independent tester
Acceptance	Requirements Analysis	Whole product in customer's environment	Black Box	Customer
Beta	Ad hoc	Whole product in customer's environment	Black box	Customer
Regression	Changed Documentation High-Level Design	Any of the above	Black Box White Box	Programmer(s) or independent testers

Table 1: Levels of Software Testing

It is best to find a fault as early in the development process as possible. When a test case fails, you have now seen a symptom of the failure [13] and still need to find the fault that caused the failure. The further you go into the development process the harder it is to track down the cause of the failure. If you unit test often, a new failure is likely to be in the code you just wrote/tested and should be reasonably easy to find. If you wait until system or acceptance testing, a failure could be anywhere in the system – you will have to be an astute detective to find the fault now.

1.5 Test Planning

Test planning should be done throughout the development cycle, especially early in the development cycle. A *test plan* is a *document describing the scope, approach, resources, and schedule of intended test activities. It identifies test items, the features to be tested,*

the testing tasks, who will do each task, and any risks requiring contingency plans [11]. An important component of the test plan is the individual test cases. A *test case* is a *set of test inputs, execution conditions, and expected results developed for a particular objective, such as to exercise a particular program path or to verify compliance with a specific requirement* [11].

Write the test plan early in the development cycle when things are generally still going pretty smoothly and calmly. This allows you to think through a thorough set of test cases. If you wait until the end of the cycle to write and execute test cases, you might be in a very chaotic, hurried time period. Often good test cases are not written in this hurried environment, and ad hoc testing takes place. With ad hoc testing, people just start trying anything they can think of without any rational roadmap through the customer requirements. The tests done in this manner are not repeatable.

1.6 Testing as Part of the Development Process

It is essential in testing to start planning as soon as the necessary artifact is available. For example, as soon as customer requirements analysis has completed, the test team should start writing black box test cases against that requirements document. By doing so this early, the testers might realize the requirements are not complete. The team may ask questions of the customer to clarify the requirements so a specific test case can be written. The answer to the question is helpful to the code developer as well. Additionally, the tester may request (of the programmer) that the code is designed and developed to allow some automated test execution to be done. To summarize, the earlier testing is planned at all levels, the better.

It is also very important to consider test planning and test execution as iterative processes. As soon as requirements documentation is available, it is best to begin to write functional and system test cases. When requirements change, revise the test cases. As soon as some code is available, execute test cases. When code changes, run the test cases again. By knowing how many and which test cases actually run you can accurately track the progress of the project. All in all, testing should be considered an iterative and essential part of the entire development process.

2 Performing Black Box Testing

Black box testing, also called *functional testing and behavioral testing*, focuses on determining whether or not a program does what it is supposed to do based on its functional requirements. Black box testing attempts to find errors in the external behavior of the code in the following categories [17]: (1) incorrect or missing functionality; (2) interface errors; (3) errors in data structures used by interfaces; (4) behavior or performance errors; and (5) initialization and termination errors. Through this testing, we can determine if the functions appear to work according to specifications. However, it is important to note that no amount of testing can unequivocally demonstrate the absence of errors and defects in your code.

It is best if the person who plans and executes black box tests is ***not*** the programmer of the code and does not know anything about the structure of the code. The programmers

of the code are innately biased and are likely to test that the program does *what they programmed it to do*. What are needed are tests to make sure that the program does *what the customer wants it to do*. As a result, most organizations have independent testing groups to perform black box testing. These testers are not the developers and are often referred to as *third-party* testers. Testers should just be able to understand and specify what the desired output should be for a given input into the program, as shown in Figure 3.

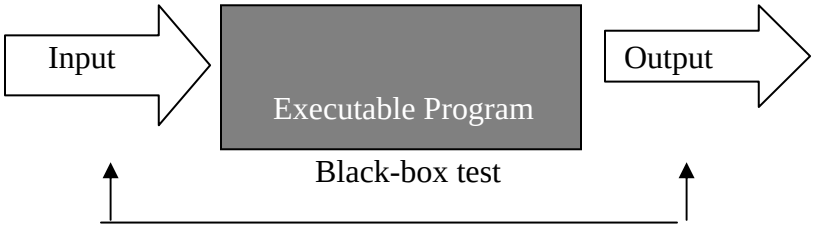


Figure 3: Black Box Testing. A black-box test takes into account only the input and output of the software without regard to the internal code of the program.

2.1 The Anatomy of a Test Case

The format of your test case design is very important. We will use a particular format for our test cases, as shown in Table 2. We recommend you use this template in your test planning.

Test ID	Description	Expected Results	Actual Results

Table 2: Test Case Planning Template

First, you give each test case a unique identifier. When you are tracking large projects, you might need to itemize those test cases that have not yet passed. This identifier is recorded in the first column. For example, you might need to say something like, "All my test cases are running except playerMovement1. I'm working on that one today." Next in the second column of the table, you specifically describe the set of steps and/or input for the particular condition you want to test (including what needs to be done to prepare for the test case to be run). The third column is the expected results for an input/output oracle – what is expected to come out of the "black box" based upon the input (as described in the "description"). An *oracle* is *any program, process, or body of data that specified the expected outcome of a set of tests as applied to a tested object* [1]; and *input/output oracle* is an *oracle that specifies the expected output for a specified input* [1]. In the last column, the actual results are recorded after the tests are run. If a test passes, the actual results will indicate "Pass." If a test fails, it is helpful to record "Fail" and a description of the failure ("what came out") in the actual results column.

2.2 Clear Descriptions

It is of prime importance that the test case description be very clear and specific so that the test case execution is repeatable. Even if you will always be the person executing the test cases, pretend you are passing the test planning document to someone else to perform the tests. You need your directions to clear enough for that other person to be able to follow the directions explicitly so that the exact same test is executed every time. For example, consider a basic test case to ensure that players can move on a Monopoly board. Example of poorly specified test case is shown in Table 3:

Test ID	Description	Expected Results	Actual Results
1	Player 1 rolls dice and moves.	Player 1 moves on board.	
2	Player 2 rolls dice and moves.	Player 2 moves on board.	

Table 3: Poor Specification of a Test Case

The problem is that the description does not give exact values of how many spaces the players moved. This is an overly simplistic problem – but maybe the program crashes for some reason when Player 1 and Player 2 land on the same spot. If you don't remember what was actually rolled (you let the rolls be determined randomly and don't record them), you might never be able to cause the problem to happen again because you don't remember the circumstances leading up to the problem. *Recreating the problem* is essentially important in testing so that problems that are identified can be repeated and corrected. Instead write specific descriptions, such as shown in Table 4.

Test ID	Description	Expected Results	Actual Results
3	Precondition: Game is in test mode, SimpleGameBoard is loaded, and game begins. Number of players: 2 Money for player 1: $1200 Money for player 2: $1200 Player 1 dice roll: 3	Player 1 is located at Blue 3.	
4	Precondition: Test case 3 has successfully completed Player 2 dice roll: 3	Player 1 is located on Blue 3. Player 2 is located on Blue 3.	

Table 4: Preferred Specification of a Test Case

There a few things to notice about the test cases in Table 4. First, notice the Precondition in the Description field. The precondition defines what has to happen before the test case can run properly. There may be an order of execution [5] whereby a test case may depend upon another test case running successfully and leaving the system in a state such that the second test case can successfully be executed. For example, maybe one test case (call it Test 11) tests whether a new user can create an ID in a system. Another test case

(call it Test 22) may depend upon this new user logging in. Therefore Test 11 must run before Test 22 can run. Additionally, if Test 11 fails, than Test 22 cannot be run yet. Alternately, perhaps Test 11 passes but Test 22 fails. Later when the functionality is fixed, Test 11 must be re-run before the testers try to re-run Test 22. Or, maybe a database or the system needs to be re-initialized before a test case can run.

There's also something else important to notice in the Preconditions for test case 3 in Table 4. How can the test case ensure the player rolled a 3 when the value the dice rolls needs to be random in the real game? Sometimes we have to add a bit of extra functionality to put a program in "test mode" so we can run our test cases in a repeatable manner and so we can easily force a condition happen. For example, we may want to test what happens when a player lands on "Go" or on "Go to Jail" and want to force this situation to occur. The Monopoly programmers needed to create a test mode in which (1) the dice rolls could be input manually and (2) the amount of money each player starts with is input manually. It is also important to run some non-repeatable test cases in the regular game mode to test whether random dice input does not appear to change expected behavior.

The expected results must also be written in a very specific way, as in Table 4. You need to record what the output of the program should be, given a particular input/set of steps. Otherwise, how will you know if the answer is correct (every time you run it) if you don't know what the answer is supposed to be? Perhaps your program performs mathematical calculations. You need to take out your calculator, perform some calculations by hand, and put the answer in the expected result field. You need to pre-determine what your program is supposed to do ahead of time, so you'll know right away if your program responds properly or not.

3 Strategies for Black Box Testing

Ideally, we'd like to test every possible thing that can be done with our program. But, as we said, writing and executing test cases is expensive. We want to make sure that we definitely write test cases for the kinds of things that the customer will do most often or even fairly often. Our objective is to find as many defects as possible in as few test cases as possible. To accomplish this objective, we use some strategies that will be discussed in this subsection. We want to avoid writing redundant test cases that won't tell us anything new (because they have similar conditions to other test cases we already wrote). Each test case should probe a different mode of failure. We also want to design the simplest test cases that could possibly reveal this mode of failure – test cases themselves can be error-prone if we don't keep this in mind.

3.1 Tests of Customer Requirements

Black box test cases are based on customer requirements. We begin by looking at each customer requirement. To start, we want to make sure that every single customer requirement has been tested at least once. As a result, we can trace every requirement to

its test case(s) and every test case back to its stated customer requirement. The first test case we'd write for any given requirement is the most-used *success path* for that requirement. By success path, we mean that we want to execute *some desirable functionality (something the customer wants to work) without any error conditions*. We proceed by planning more success path test cases, based on other ways the customer wants to use the functionality and some test cases that execute *failure paths*. Intuitively, failure paths *intentionally have some kind of errors in them, such as errors that users can accidentally input*. We must make sure that the program behaves predictably and gracefully in the face of these errors. Finally, we should plan the execution of our tests out so that the most troublesome, risky requirements are tested first. This would allow more time for fixing problems before delivering the product to the customer. It would be devastating to find a critical flaw right before the product is due to be delivered.

We'll start with one basic requirement. We can write many test cases based on this one requirement, which follows below. As we've said before, it is impossible to test every single possible combination of input. We'll outline an incomplete sampling of test cases and reason about them in this section.

Requirement: When a user lands on the "Go to Jail" cell, the player goes directly to jail, does not pass go, does not collect $200. On the next turn, the player must pay $50 to get out of jail and does not roll the dice or advance. If the player does not have enough money, he or she is out of the game.

There are many things to test in this short requirement above, including:

1. Does the player get sent to jail after landing on "Go to Jail"?
2. Does the player receive $200 if "Go" is between the current space and jail?
3. Is $50 correctly decremented if the player has more than $50?
4. Is the player out of the game if he or she has less than $50?

At first it is good to start out by testing some input that you know should definitely pass or definitely fail. If these kinds of tests don't work properly, you know you should just quit testing and put the code back into development. We can start with a two obvious passing test case, as shown in Table 5.

Test ID	Description	Expected Results	Actual Results
5	Precondition: Game is in test mode. Number of players: 1 Money for player 1: $1200 Player 1 dice roll: 3 Player 1 clicks "End Turn" button.		
		Player 1 is sent to jail Only "Get Out of Jail" button is enabled for Player 1.	
	Player 1 clicks "Get Out of Jail" button.		
		Money for Player 1: $1150	
6	Precondition: Game is in test mode. Number of players: 2 Money for player 1: $1200 Money for player 2: $1200 Player 1 dice roll: 3 Player 1 clicks "End Turn" button.		
		Player 1 is sent to jail	
	Player 2 dice roll: 2 Player 2 clicks "End Turn" button.		
		Only "Get Out of Jail" button is enabled for Player 1.	
	Player 1 clicks "Get out of Jail" button.		
		Money for Player 1: $1150	

Table 5: Test Plan #1 for the Jail Requirement

You will also note that we should *test the simplest possible means to force the condition we are trying to achieve*. For example, in Test Case 5, we only have one player so we temporarily didn't have to spend our time with Player 2. We add Player 2 in Test Case 6 so we can observe that the loss of $50 and dice roll occurs on the next turn (after Player 2 goes). We could go on and test many more aspects of the above requirement. We will now discuss some strategies to consider in creating more test cases.

3.2 Equivalence Partitioning

To keep down our testing costs, we don't want to write several test cases that test the same aspect of our program. A good test case uncovers a different class of errors (e.g., incorrect processing of all character data) than has been uncovered by prior test cases. [17] Equivalence partitioning is a strategy that can be used to reduce the number of test cases that need to be developed. Equivalence partitioning divides the input domain of a

program into classes. For each of these equivalence classes, the set of data should be treated the same by the module under test and should produce the same answer. Test cases should be designed so the inputs lie within these equivalence classes. [2] For example, for tests of "Go to Jail" the most important thing is whether the player has enough money to pay the $50 fine. Therefore, the two equivalence classes can be partitioned, as shown in Figure 4.

Figure 4: Equivalence Classes for Player Money

Less than $50	$50 or more

Once you have identified these partitions, you choose test cases from each partition. To start, choose *a typical value somewhere in the middle of (or well into) each of these two ranges*. See Table 6 for test cases written to test the equivalent classes of money. However, you will note that Test Cases 6 (Player 1 has $1200) and 7 (Player 1 has $100) are both in the same equivalence class. Therefore, Test Case 7 is unlikely to discover any defect not found in Test Case 6.

Test ID	Description	Expected Results	Actual Results
7	Precondition: Game is in test mode. Number of players: 2 Money for player 1: $100 Money for player 2: $100 Player 1 dice roll: 3 Player 1 clicks "End Turn"		
		Player 1 is sent to jail	
	Player 2 dice roll: 2 Player 2 clicks "End Turn"		
		Only "Get Out of Jail" is enabled for Player 1.	
	Player 1 clicks "Get Out of Jail"		
		Money for Player 1: $50	
8	Precondition: Game is in test mode. Number of players: 2 Money for player 1: $25 Money for player 2: $25 Player 1 dice roll: 3 Player 1 clicks "End Turn"		
		Player 1 is sent to jail	
	Player 2 dice roll: 2 Player 2 clicks "End Turn"		
		Only "Get Out of Jail" is enabled for Player 1.	
	Player 1 clicks "Get out of Jail"		
		Player 1 is out of game	

Table 6: Test Plan #2 for the Jail Requirement

For each equivalent class, the test cases can be defined using the following guidelines [17]:

1. If input conditions specify a *range of values*, create one valid and one or two invalid equivalence classes. In the above example, this is (1) less than 50/invalid; (2) 50 or more/valid.
2. If input conditions require a certain value (for example R and L for the side in our train example), create an equivalence class of the valid values (R and L) and one of invalid values (all other letters other than R and L). In this case, you need to test all valid values individually and several invalid values.
3. If input conditions specify a member of a set, create one valid and one invalid equivalence class.

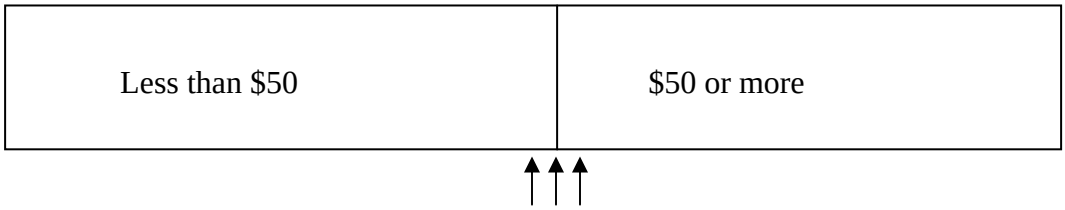
4. If an input condition is a Boolean, define one valid and one invalid class.

Equivalence class partitioning is just the start, though. An important partner to this partitioning is boundary value analysis.

3.3 Boundary Value Analysis

Boris Beizer, well-known author of testing book advises, "Bugs lurk in corners and congregate at boundaries." [1] Programmers often make mistakes on the boundaries of the equivalence classes/input domain. As a result, we need to focus testing at these boundaries. This type of testing is called Boundary Value Analysis (BVA) and guides you to create test cases at the "edge" of the equivalence classes. *Boundary value* is defined as a *data value that corresponds to a minimum or maximum input, internal, or output value specified for a system or component* [11]. In our above example, the boundary of the class is at 50, as shown in Figure 5. We should create test cases for the Player 1 having $49, $50, and $51. These test cases will help to find common off-by-one errors, caused by errors like using >= when you mean to use >.

Figure 5: Boundary Value Analysis. Test cases should be created for the boundaries (arrows) between equivalence classes.



When creating BVA test cases, consider the following [17]:

1. If input conditions have a range from **a** to **b** (such as a=100 to b=300), create test cases:
   - immediately below **a** (99)
   - at **a** (100)
   - immediately above **a** (101)
   - immediately below **b** (299)
   - at **b** (300)
   - immediately above **b** (301)
2. If input conditions specify a *number* of values that are allowed, test these limits. For example, input conditions specify that only one train is allowed to start in each direction on each station. In testing, try to add a second train to the same station/same direction. If (somehow) three trains could start on one station/direction, try to add two trains (pass), three trains (pass), and four trains (fail).

3.4 Decision Table Testing

Decision tables are used to record complex business rules that must be implemented in the program, and therefore tested. A sample decision table is found in Table 7. In the table, the conditions represent possible input conditions. The actions are the events that should trigger, depending upon the makeup of the input conditions. Each column in the table is a unique combination of input conditions (and is called a rule) that result in triggering the action(s) associated with the rule. Each rule (or column) should become a test case.

If a Player (A) lands on property owned by another player (B), A must pay rent to B. If A does not have enough money to pay B, A is out of the game.

Table 7: Decision table

	Rule 1	Rule 2	Rule 3
Conditions			
A lands on B's property	Yes	Yes	No
A has enough money to pay rent	Yes	No	--
Actions			
A stays in game	Yes	No	Yes

3.5 Failure ("Dirty") Test Cases

Donald Knuth is many times referred to as one of the fathers of computer science. He is also known as a stickler when it comes to bugs in his code (and in his books. He sends checks to readers who find errors in his books!). Anticipating the unexpected is one of his techniques. Think the way Knuth does when you write your test cases. Be mean and nasty!

My test programs are intended to break the system, to push it to its extreme limits, to pile complication on complication, in ways that the system programmer never consciously anticipated. To prepare such test data, I get into the meanest, nastiest frame of mind that I can manage, and I write the cruelest code I can think of; then I turn around and embed that in even nastier constructions that are almost obscene. [14]

Think diabolically! Think of every possible thing a user could possibly do with your system to demolish the software. You need to make sure your program is robust – in that it can properly respond in the face of erroneous user input. This type of testing is called *robustness testing*, whereby test cases are chosen outside the domain to test robustness to unexpected, erroneous input [3], and is included in *defensive testing* which includes *tests under both normal and abnormal conditions* [5]. Look at every input. Does the program respond "gracefully" to these error conditions?

1. Can any form of input to the program cause division by zero? Get creative!
2. What if the input type is wrong? (You're expecting an integer, they input a float. You're expecting a character, you get an integer.)
3. What if the customer takes an illogical path through your functionality?
4. What if mandatory fields are not entered?
5. What if the program is aborted abruptly or input or output devices are unplugged?

3.5 Test Early and Often

As was said in the beginning of the chapter, executing your test cases as soon as possible is an excellent way of getting concrete feedback about your program. In order to run test cases early, programmers need to integrate the pieces of their code into the code base often. Programmers could be tempted to work on their own computer until the finish implementing a "whole" requirement. In industry, this could quite feasibly mean they keep their code to themselves for several months. However, this is a dangerous practice – and can lead to what is known in industry as *integration hell*. Just because a component works on a programmer's own computer, this doesn't mean it will work when it is assembled with the code other programmers are working on. The earlier it is known that there are some interface problems or some data that's not getting passed properly the better. This knowledge can only be gained by integrating code and testing early and often. Then, integration problems can be more easily localized in the work that was just integrated. By localizing the code that contains a new defect, the programmer can efficiently identify and remove defects.

4 Acceptance Testing

Acceptance test cases are written by the customer. In custom software development, often contracts between the customer and the development organization state that the customer can refuse to take delivery of the product if their acceptance test cases do not run properly in the customer's own (software and hardware) environment. Sometime the customer shares the acceptance test cases with the team, which gives them a shared specific goal. Other times, the customer hides the acceptance test cases from the developers and runs them after receiving the code (in the same way as a teacher often doesn't tell the students the test cases they will run to grade their class projects). We believe it is much more productive for the customer and the development team to work openly and collaboratively on the creation of the acceptance test cases. Then, together the customer and the development team have a similar vision of what the software has to look like for the customer to be happy. In our experience, the collaborative acceptance test case creation serves as an excellent means of clarifying requirements by making requirements specified in a way that is quantifiable, measurable, and unambiguous long before testing commences. Likewise, they can together track the progress of system development as the team can tell the customer which acceptance test cases are passing.

5 Black Box Test Case Automation

By their nature, black box test cases are designed and run by people who do not see the inner workings of the code. Ultimately, system and acceptance cases are intended to be run through the product user interface (UI) to show that the whole product really works.

Test automation can be difficult because the developer has no knowledge of the inner workings of the software and because system and acceptance cases must be run through the UI. However, the more automated testing can be, the easier it is to run the test cases and to re-run them again and again. The simpler it is to run a suite of tests, the more often those tests will be run. The more the tests are run, the faster any deviation from those tests will be found. [15]

If your role on the team is as a software developer, it is always good to consider the types of black box test cases (functional, system, and acceptance) that will ultimately be run on your code and to automate test cases to test the logic (separate from the UI logic) behind these black box test cases. Automated test cases can be run often with minimal time investment once they are written. By automating the testing of the logic behind the black box test cases, (1) you are ensuring that the logic "behind the scenes" is working properly so that the inevitable black box test cases can run smoothly through the UI by the testers and the customers; and (2) you are more motivated to decouple program/business logic separate from the UI logic (which is always a good design technique).

When test cases are automated, they can then become compile-able and executable documentation.

6 Summary

Several practical tips for black box testing were presented throughout this chapter. The keys for successful black box testing are summarized in Table 8.

	You need to test for what the customer wants the program to do, not what the programmer programmed it to do. The programmer is biased (through no fault of her/her own) by knowing the intimate details of what the program does. Black box testing is best done by someone with a fresh, objective perspective of the customer requirements.
	Use the four-item test case template (ID, Description, Expected Results, Actual Results) when planning your test cases.
	In the test case, specify exactly what the tester has to do to create the desired input conditions and exactly how the program should respond (the output). Be explicit in this documentation so that multiple testers (other than yourself) would be able to run the exact same test case using the directions in the test case. These directions will be especially important if a failure need to be re-created for the programmer to a failure.
	Test early and often.
	Write the simplest test cases that could possibly reveal a mode of failure. (Test cases can also be error-prone.)
	Use *equivalence class partitioning* to manage the number of test cases run. Test cases in the same equivalence class will all reveal the same fault.
	Use *boundary value analysis* to find the very-common bugs that lurk in corners and congregate at boundaries.
	Use *decision tables* to record complex business rules that the system must implement and that must be tested.

	Run the equivalence class test cases first. If the program doesn't work for the simplest case (smack in the middle of an equivalence class), it probably won't work for the boundaries either. If you run a boundary test first, you'll probably go run the general case (equivalence class test) before investigating the problem. So, instead just run the simple case first.
	Avoid having test cases dependant upon each other (i.e. having preconditions of another test case passing). Consider that you have 17 test cases, each having a precondition of the prior test case passing – and you pass the first 16 test cases but fail the 17th test case. It take you some time (until the next day) to debug your program. Now, in order to re-run the 17th test case to see if it now passes, you have to re-run the 16 you know pass. This can be time consuming ☹
	Write each test case so that it can reveal one type of fault. Consider a test case that has three different forms of invalid input. If the test case fails, you might not know which of the three inputs make it the test case fail, and you will have to run different, smaller test cases to see which of the inputs caused problems.
	Think diabolically! What are the worst things someone could try to do to your program? Write test for these.
	Encourage a collaborative approach to acceptance testing with the customer.
	When black box test cases surface failures, they only reveal the symptoms of faults. You need to use your detective skills to find the fault in the code that caused the failure to occur.

Table 8: Key Ideas for Black Box Testing

Reminds Dijkstra, "Program testing can be used to show the *presence* of bugs, but never to show their absence!" [7] Mostly, testing can be used to check how well defect-prevention activities worked. As a beneficial side effect, testing can also be used to identify anomalies in code via dynamic execution of the code.

In this chapter, we learned that complete, exhaustive testing is impractical. However, there are good software engineering strategies, such as equivalence class partitioning and boundary value analysis, for writing test cases that will maximize your chance of uncovering as many defects as possible with a reasonable amount of testing. It is most prudent to plan your test cases as early in the development cycle as possible, as a beneficial extension of the requirements gathering process. Likewise, it is beneficial to integrate code as often as possible and to test the integrated code. In this manner, we can isolate defects in the new code – and find and fix them as efficiently as possible. Lastly, we learned the benefits of partnering with a customer to write the acceptance test cases and to automate the execution of these (and other test cases) to form compile-able and executable documentation of the system.

Glossary of Chapter Terms

Term	Definition	Source
Acceptance testing	formal testing conducted to determine whether or not a system satisfies its acceptance criteria (the criteria the system must satisfy to be accepted by a customer) and to enable the customer to determine whether or	[11]

	not to accept the system	
Black box testing (also called functional testing or behavioral testing)	testing that ignores the internal mechanism of a system or component and focuses solely on the outputs generated in response to selected inputs and execution conditions	[11]
Boundary value	data value that corresponds to a minimum or maximum input, internal, or output value specified for a system or component	[11]
Defect	See fault	
Defensive testing	Testing which includes tests under both normal and abnormal conditions	[5]
Error	the difference between a computed, observed, or measured value or condition and the true, specified, or theoretically correct value or condition	[11]
Failure	the inability of a system or component to perform its required function within the specified performance requirement	[11]
Failure path	a test case that intentionally forces an error condition to occur	
Fault	an incorrect step, process, or data definition in a program	[11]
Integration testing	testing in which software components, hardware components, or both are combined and tested to evaluate the interaction between them	[11]
Input/output oracle	an oracle that specifies the expected output for a specified input	[1]
Mistake	human action that produces an incorrect result	[11]
oracle	any program, process, or body of data that specified the expected outcome of a set of tests as applied to a tested object	[1]
Performance testing	testing conducted to evaluate the compliance of a system or component with specified performance requirements	[11]
Regression testing	selective retesting of a system or component to verify that modifications have not caused unintended effects and that the system or component still complies with its specified requirements	[11]
Robustness testing	Testing whereby test cases are chosen outside the domain to test robustness to unexpected, erroneous input	[3]
Scaffolding code	computer programs and data files built to support software development and testing but not intended to be included in the final product	[11]
Smoke tests	group of test cases that establish that the system is stable and all major functionality is present and works under “normal” conditions	[6]

Specification	a document that specifies in a complete, precise, verifiable manner, the requirements, design, behavior, or other characteristic of a system or component, and often the procedures for determining whether these provisions have been satisfied	[11]
Stress testing	testing conducted to evaluate a system or component at or beyond the limits of its specification or requirement	[11]
Stubs	computer program statement substituting for the body of a software module that is or will be defined elsewhere	[11]
Success path	a test case that execute some desirable functionality (something the customer wants to work) without any error conditions	
System testing	testing conducted on a complete, integrated system to evaluate the system compliance with its specified requirements	[11]
Test case	set of test inputs, execution conditions, and expected results developed for a particular objective, such as to exercise a particular program path or to verify compliance with a specific requirement	[11]
Test driver	software module used to involve a module under test and often, provide test inputs, controls, and monitor execution and report test results	[11]
Test plan	document describing the scope, approach, resources, and schedule of intended test activities. It identifies test items, the features to be tested, the testing tasks, who will do each task, and any risks requiring contingency plans	[11]
Unit testing	testing of individual hardware or software units or groups of related units	[11]
Usability testing	testing conducted to evaluate the extent to which a user can learn to operate, prepare inputs for, and interpret outputs of a system or component	[11]
Validation	the process of evaluating a system or component during or at the end of the development process to determine whether it satisfies specified requirements	[11]
Verification	the process of evaluating a system or component to determine whether the products of a given development phase satisfy the conditions imposed at the start of that phase	[11]
White box testing	testing that takes into account the internal mechanism of a system or component	[11]

References:

[1] B. Beizer, *Software Testing Techniques*. London: International Thompson Computer Press, 1990.
[2] B. Beizer, *Black Box Testing*. New York: John Wiley & Sons, Inc., 1995.
[3] A. Bertolino, "Chapter 5: Software Testing," in *IEEE SWEBOK Trial Version 1.00*, May 2001.
[4] B. W. Boehm, *Software Engineering Economics*. Englewood Cliffs, NJ: Prentice-Hall, Inc., 1981.
[5] L. Copeland, *A Practitioner's Guide to Software Test Design*. Boston: Artech House Publishers, 2004.
[6] R. D. Craig and S. P. Jaskiel, *Systematic Software Testing*. Norwood, MA: Artech House Publishers, 2002.
[7] E. W. Dijkstra, "Notes on Structured Programming," Technological University Eindhoven T.H. Report 70-WSK-03, Second edition, April 1970.
[8] D. Galin, *Software Quality Assurance*. Harlow, England: Pearson, Addison Wesley, 2004.
[9] IEEE, "ANSI/IEEE Standard 1008-1987, IEEE Standard for Software Unit Testing," no., 1986.
[10] IEEE, "ANSI/IEEE Standard 1008-1987, IEEE Standard for Software Unit Testing," no., 1987.
[11] IEEE, "IEEE Standard 610.12-1990, IEEE Standard Glossary of Software Engineering Terminology," 1990.
[12] IEEE, "IEEE Standards Collection: Glossary of Software Engineering Terminology," IEEE Standard 610.12-1990, 1990.
[13] C. Kaner, J. Bach, and B. Pettichord, *Lessons Learned in Software Testing*: John Wiley & Sons, 2002.
[14] D. E. Knuth, "The errors of TeX. Software--Practice and Experience," in *Literate Programming; CSLI Lecture Notes, no. 27*, vol. 19: CSLI, 1992, pp. 607--681.
[15] R. C. Martin, *Agile Software Development: Principles, Patterns, and Practices*. Upper Saddle River: Prentice Hall, 2003.
[16] G. J. Myers, *The Art of Software Testing*. New York: John Wiley, 979.
[17] R. Pressman, *Software Engineering: A Practitioner's Approach*. Boston: McGraw Hill, 2001.

Chapter Questions

1. What is the difference between black-box and white-box testing? During the software development, how can we derive black-box tests? How about white-box tests?
2. Dharma City is installing the AutoCop Traffic Law Enforcement System. AucoCop is a sensor-camera combo installed near a traffic light. When the sensor detects a speeding (faster than 40miles/hour) car passing by or a car running through the red light, AutoCop will activate the camera and take a picture of the plate. Use the equivalence partitioning and boundary value analysis methods to derive the test cases to test the camera activation logic.

3. From the perspective of automating software testing, what is the problem if the user interface and the business logic are heavily coupled?
4. Describe in your own words the difference between validation and verification.
5. In XP, the customer and developers work cooperatively to specify the acceptance tests. What are to pros and cons if the customer and developers work together on acceptance tests?
6. What's the advantage if acceptance tests can be automated?
7. Suppose you are writing a program that counts the number of alphanumeric characters in a string. May we apply equivalence partitioning for this program? What about boundary value analysis? Do we need more test cases to validate the program?
8. Suppose we are developing a program which decides, in a two-dimensional coordinate system, whether a point P falls in a circle C or on its edge. The program reads five real numbers. The first two numbers are the x- and y-coordinate of the center of C, the third number is the radius of C, and the fourth and fifth numbers represent the coordinate of P. Develop the test cases that you feel are adequate for this program.
9. Some organizations have independent testing groups. What tests are best designed by the testing group? What tests are best designed by the developers? And what tests are best designed by the customer? Justify your answer.
10. It is impractical to run all the test cases whenever changes are made. An organization may adopt a prioritization scheme for the test cases to choose appropriate test cases to run. Give three test case prioritization criteria.
11. You are testing an automatic auction system. Suppose there is an auction of which the bids can only be placed between 1/1/2004 and 1/7/2004. The starting bid price of this auction is $20.00, and a minimum increment of $5.00 is required for a successful bid. Using the equivalence partitioning and boundary value analysis methods to derive a set of test cases for the bid placement. Also give some "dirty" test cases.
12. Suppose you are writing a simple calculator program. This program can handle positive integer calculation, including addition, subtraction, multiplication, and division. The input is a string composed of digits (0, 1, 2, …, 9) and operators (+, -, *, /). No space is allowed. The input string can be at most 100 characters long, and each number can compose of at most 10 digits. Division of two integers produces one integer by truncation. If the answer contains more than 10 digits, this program simply outputs an overflow error message. Using the equivalence partitioning and boundary value analysis methods, derive a set of test cases for the program. Also give some dirty test cases.
13. Acceptance tests are specified by the customer with the help of developers. Usually the customer has better knowledge in their business than in programming. Therefore, it is next to impossible for the customer to write or understand the tests using the programming language. What do you think is a feasible form of acceptance tests? (Remember that we'd like the acceptance tests executable.)