Traditional and Agile Software Engineering

Michele Marchesi, Giancarlo Succi, Laurie Williams

Chapter 8 – Black Box Testing

To be published by Addison-Wesley in 2003

Please do not quote this manuscript, as it is a very preliminary version, still subject to several modifications, corrections, etc

If you have any question or suggestion, please e-mail to: Michele@diee.unica.it, Giancarlo.Succi@unibz.it, Williams@csc.ncsu.edu

8. Black Box Testing

In this chapter, we will first describe the basics of software testing throughout the entire software development lifecycle. There are two basic kinds of software testing, black box testing and white box testing. This chapter on black box testing follows the chapters on requirements because black box test cases are based on requirements. Depending upon the type of black box testing, these tests could either be based on the "pure" requirements in the eyes of customer or a requirements analysis or requirements specification document. And, as we will explain, the act of writing test cases can drive the requirements gathering process to be more complete and specific.

8.1 Software Testing

Software testing is one of the "verification and validation," or V&V, software practices and is considered to be a software quality assurance practice. Some other V&V practices, such as unit testing, inspections, and pair programming, will be discussed throughout this book. Through *verification* (the first V), we ensure that software correctly implements a specific function. For example, by running specific tests we can verify that new trains that are added to our subway cannot have the same name as existing trains. Through *validation* (the second V), we ensure that the features of the software that has been built can be found in the customer requirements. For example, we validate that trains can be removed from the subway system. Boehm [3] has summarized the difference between verification and validation as follows:

Verification: Are we building the product right?
Validation: Are we building the right product?

8.1.1 The Economics of Software Testing

In software development, there are costs associated with testing our programs. We need to plan out our test cases, we need to write the 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 defects we find. (Actually, sometimes we can find low-priority defects in our code and decide that it is too expensive to fix the defect because of the need to redesign, recode, etc. These defects can remain in the product through a follow-on release or perhaps forever.) When we didn't discover and remove them beforehand, there are also costs associated with shipping software defects to our customers in our systems. When this happens, customers can lose faith in our business and can get very angry. Customers can also lose a great deal of money if their system goes down because of our defects (like a grocery store that can't check out the shoppers using 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 their bugs. Sometimes, programmers have to get on airplanes and fly to customer locations to work directly with them on their problems. These trips are costly to the development organization, and the customers are also not overly cheerful to work with when the programmer arrives.

We also need to consider the loss potential of each individual 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! [5]

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 defects. 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 defects 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 the maximum number of errors with the minimum amount of effort and time.

8.1.2 The Basics of Software Testing

There are two basic types of software testing, black box testing and white box testing. For now, you just need to understand the very basic difference between the two types of testing. With *black box testing*, the software tester does not (or should not) have access to the code itself. The code is a big black box to them (hence, the name), and they can't see inside the box. They just know that they can input stuff to the black box and the black box will send something back out. Based on their 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, etc. Back to 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. Chapter 15 focuses on white box testing.

8.1.3 Plan and Execute Tests Early and Often!

Sadly, software testing can sometimes be considered as an afterthought. This is very harmful to the quality assurance of a product and to the economics of the entire software development lifecycle. Instead, we encourage you to plan and execute test cases as early and as often as you can. It is most economical to remove bugs (in our code and in our documents) as quickly as possible after they are injected. By planning test cases as soon as you have the right information (we'll talk more about this) and by executing test cases as soon as you have running code, you can prevent defects from occurring, and you can remove them as quickly as possible.

There are several levels of testing that should be done on a large software system. We'll explain these, going from the lowest level of testing to the highest level of testing.

1. Unit Testing. 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. 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. 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 – data could get lost across an interface, messages may not get passed properly, interfaces weren't implemented as specified, etc. To plan these integration test cases, testers look at **high- and low-level** design documents.
3. Functional and System testing. Using **black box testing** techniques, testers examine the **high-level design** and the **customer requirements analysis** 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 requirements works. *System testing* involves putting the new program in many different environments to ensure the program works in typical customer environments -- various versions of operating systems, with other typical applications running, etc.
4. Acceptance testing. After functional and system testing, the product is delivered to a customer and the customer runs their own **black box** acceptance tests based on their own understanding of their **requirements**. These tests are often pre-specified by the customer. The customer reserves the right to refuse delivery of the software if the acceptance test cases do not pass; these are often run in the customer's own environment. A word of caution: customers are not trained software testers. Customers generally do not specify a 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. Also, we will discuss running the customer acceptance test cases ourselves so that we can increase our confidence that they will work at the customer location.
5. Regression testing. Throughout all testing cycles, *regression* test cases are run. 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. However, in the case of small to medium sized systems (up to about one hundred classes), always running all tests may be feasible. Since regression tests are run throughout the

development cycle, there can be **white box** regression tests at the unit and integration level 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.

These five levels of testing are summarized in Table 1.

Testing Level	Tests Based Upon	Kind of Testing
Unit	Low-Level Design Actual Code Structure	White Box
Integration	Low-Level Design High-Level Design	Black Box White Box
Functional and System	High Level Design Requirements Analysis	Black Box
Acceptance	Requirements	Black Box
Regression	Change Documentation High-Level Design	Black Box White Box

Table 1: Levels of Software Testing

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. This will be discussed more in Section 8.7. To summarize, the earlier testing is planned at all levels, the better. We will continue to emphasize this point.

It is also very important to consider test planning and test execution a very iterative process. As soon as requirements documentation is available, 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. Knowing how many and which test cases actually run is an excellent way to track the progress of the project. All in all, *testing should be considered an iterative and essential part of the entire development process*.

8.2 Introduction to 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: (1) incorrect or missing functions; (2) interface errors; (3) errors in data structures used by interfaces; (4) behavior or performance errors; and (5) initialization and termination errors. [5] Through this testing, we can determine if the functions appear to work according to specifications. However, it is important to note that no rational amount of testing can unequivocally demonstrate the absence of errors and defects in your code.

In order to perform black box testing, one does not need to know anything about the internal code structure of the software. Actually, it is best if the person who plans and executes black box test is NOT the programmer of the code and does not know anything about the structure of the code. Testers should just be able to intimately understand and specify what the desired output should be for a given input into the program, as shown in Figure 1. 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. These independent testers should not be biased by understanding the inner workings of the program and should just consider what input into the system and what expected output would come out of the system if it operated as desired.

Figure 1: Black Box Testing

8.3 Test Case PlanningDesign

Test planning should be done throughout the development cycle, *especially early in the development cycle*. Often the act of writing test cases serves to elicit questions which ultimately clarify customer requirements. (If we find we can't test a requirement, how can we know when we're done? We'd better go back to the customer and ask more questions.) Additionally early in the development cycle, things are generally still going pretty smoothly and calmly. This allows you to think through a thorough set of test cases make a plan full of nicely specified 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.

The format of your test case design is very important. In this chapter, we will use a particular format for our test cases, as shown in Table 2. We recommend do the same in your test planning.

Test ID	Description	Expected Results	Actual Results

Table 2: Test Case Planning Template

- First, you give each test case an identifier. When you are tracking large projects, you might need to give a status on which test cases (by identifier) have not yet passed. For example, you might say, "All my test cases are running except Station-1. I'm working on that one today."
- Next, you specify the description, the set of steps or input conditions (e.g. the value of the input parameters) for the test case to execute the particular condition you want to test.

  It is of prime importance that the test case description be very 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 be very clear for them. Why is this important? You need to make sure that you execute the SAME test every time. For example, consider a program that asks you to input a name and an age. The program is intended to only allow users that are age 14 and up.

  An example of a poorly specified test case is shown in Table 3:

Test ID	Description	Expected Results	Actual Results
1	Input valid name and age	"Hello _name_" echoed on the screen	

Table 3: Poor Test Case Design

The first time you run the test case, you input "Mickey Mouse, age 13." The test case fails (doesn't achieve the expected results). The program tells the user they are not old enough to use the program instead of greeting them with "Hello" and their name. As a tester, you go and tell the programmer that when you input a valid name and a valid age, and the program doesn't do what it is supposed to do. The programmer will want to re-run the test case so she can see how it fails. She will ask you what you input when it failed. Well, maybe you've been testing all

day and can't remember what you input. If you didn't document EXACTLY what you intended to do, you might forget. So, you (incorrectly) tell the programmer you put in "Minney Mouse, age 14." She runs the program with that input, and low and behold the program works just fine. You look foolish, and the defect (an off-by-one error) remains. This is a simplified example. The input to black box test cases are often much more complex with several input values and steps. *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
1	Input good name= "Mickey Mouse" and age=13.	"Hello Mickey Mouse" echoed on screen.	

Table 4: Preferred Test Case Design

- The expected results must also be very specific. 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. Think of every test case as a mathematical function, like **y = f(x)**. By definition, **f** is a mathematical function if when it is applied to argument/input **x**, the computation results in one and only one answer, **y**. For your program, **f(x)** is the input (x) and processing of the program and is specified in the description (second column). The expected results are **y**. You need to determine what your program is supposed to do ahead of time, so you'll know right away if your program responds properly or not.
- Lastly, you leave a place in our template to record the actual results -- what actually happens when the test case is run.

8.4 Black Box Testing by Example

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. We also 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.

In this section, we demonstrate strategies for writing a good set of black box test cases. Using these strategies, you could rank your set of test cases according to their cost and their probability of catching mistakes. We will use our subway train example from Chapter 4. The examples use XML. If you're not familiar with XML, take a few minutes and read the short primer on XML element syntax in the sidebar.

Sidebar: **A Short Introduction to XML Element Syntax**

XML (which stands for e**X**tensible **M**arkup **L**anguage) was developed by the W3C (World Wide Web Consortium[1], an organization in charge of the development and maintenance of most Web standards) to overcome the limitations of HTML. The building block of XML is the *element*. Each element has a tag and a content/value. The content of an element is delimited by special start and end tags. If you are familiar with HTML, the syntax for each element will look very similar to that of HML. For example, in HTML an element might look like:

<title> Black Box Testing </title>

The tag is **title** and its value is **Black Box Testing**. The element begins with a start tag **<title>** and ends with a end tag **</title>** where the **/** in the second tag denotes the closing of the element. In HTML, the tags (such as **title**) are predefined in the language. The tag **title** has special meaning in the HTML language; when a **title** tag is encountered, the language knows what to do with it. The value for title (**Black Box Testing)** is usually displayed as a text header by the browser in a special title or status window.

When you use XML, there are no predefined tags. You need to create the tags you need, and you need to define what should be done with the value in an element. You describe what should be done with your elements in another file called a Document Type Definition (or DTD). Aside from the need to define your own tags, the syntax of an XML element is similar to HTML. For example,

<length>150<length>

This element has a tag **length** and a value **150**. The length tag would need to be defined in the DTD.

8.4.1 Traceable to 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

[1] See http://www.w3.org

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) for the customer 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 have some kind of errors in them, errors that users can accidentally input. We must make sure that the program behaves predictably and gracefully in the face of these errors.

We'll start with one basic subway train 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: The user should be able to define the trains, specifying their identifier, length, acceleration, maximum speed, and start track in a text file, using a pre-specified XML format (see Figure 2). The words in the /* */ are comments to clarify valid input values..

Figure 2: Train Specification

```
<TRAIN>
        <id>            /* string, 3-15 characters */      </id>
        <length>        /* integer, value 100-300 */       </length>
        <acceleration> /* float, value 0.5 -1.0 */         </acceleration>
        <speed>         /* integer, value 60-90 */         </speed>
        <startStation>  /* string, 3-15 characters */      </startStation>
        <side>          /* L (for left) or R (for right) */ </side>
</TRAIN>
```

All data fields are mandatory. Train identifiers must be unique. The startStation is the identifier of the station where the train resides when the system is started. Side indicates whether the train is in the right track or the left track; this also determines the train direction. Two trains cannot be starting from the same station going in the same direction. Length values are in meters, speed values are km/hour, acceleration and deceleration values are in m/sec^2.

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

1. All data fields have valid input values;
2. Some data fields are input without values;
3. Non-unique train identifiers;
4. Non-existent station identifiers;
5. Trains starting on top of each other (starting on same side and same station);
6. Values out of the ranges specified above; and
7. Input not of the correct data type.

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 two obvious passing test cases and two obvious failing test cases, as shown in Table 5.

Test ID	Description	Expected Results	Actual Results
1	<TRAIN> <id>Kimberly</id> <length>105</length> <acceleration>0.7</acceleration> <speed>70</speed> <startStation>HollySprings</startStation> <side>L</side> </TRAIN>	Train added.	
2	<TRAIN> <id>Christopher</id> <length>110</length> <acceleration>0.6</acceleration> <speed>75</speed> <startStation>HollySprings</startStation> <side>R</side> </TRAIN>	Train added.	
3	<TRAIN> <id>Christopher</id> <length>115</length> <acceleration>0.6<acceleration> <speed>75</speed> <startStation>Cary</startStation> <side>R</side> </TRAIN>	Fail. Train id not unique.	
4	<TRAIN> <id>Brian</id> <length>115</length> <acceleration>0.7</acceleration> <speed>75</speed> <startStation>HollySprings</startStation> <side>R</side> </TRAIN>	Fail. Track already used.	

Table 5: Test Plan #1

In these tests, each train entered is recorded in the system, and is present when subsequent trains are entered. In this way, it is possible to test duplication errors. In other cases, the system state is reset before every test, to ensure that previous errors do not influence the automatic execution of the whole test suite.

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.

8.4.2 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. "An ideal test case single-handedly uncovers a class of errors (e.g. incorrect processing of all character data) that might otherwise require many cases to be executed before the general error is observed." [5] 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 program should behave similarly. Test cases should be designed so the inputs lie within these equivalence classes. [2] For example, the length of the train can be between 100-300 meters. Therefore, the three equivalence classes can be partitioned, as shown in Figure 3.

Figure 3: Equivalence Classes for Train Length

Less than 100	Between 100 and 300	More than 300

Once you have identified these partitions, you choose test cases from each partition. To start, choose a typical value, somewhere in the middle of each of these three ranges. See Table 6 for test cases written to test the equivalent classes of the train length.

Test ID	Description	Expected Results	Actual Results
1	<TRAIN> <id>Danny</id> <length>95</length> <acceleration>0.7</acceleration> <speed>70</speed> <startStation>Cary</startStation> <side>L</side> </TRAIN>	Fail. Train too short.	
2	<TRAIN> <id>Rita</id> <length>200</length> <acceleration>0.7</acceleration> <speed>75</speed> <startStation>Raleigh</startStation>	Train added.	

Test ID	Description	Expected Results	Actual Results
	<side>R</side> </TRAIN>		
3	<TRAIN> <id>Jerry</id> <length>305<length> <acceleration>0.7</acceleration> <speed>75</speed> <startStation>Raleigh<startStation> <side>L</side> </TRAIN>	Fail. Train too long.	

Table 6: Test Plan #2

For each equivalent class, the test cases can be defined using the following guidelines [5]:

1. If input conditions specify a *range of values*, create one valid and two invalid equivalence classes. In the above example, this is (1) less than 100/invalid; (2) 100-300/valid; (3) more than 300/invalid.
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.

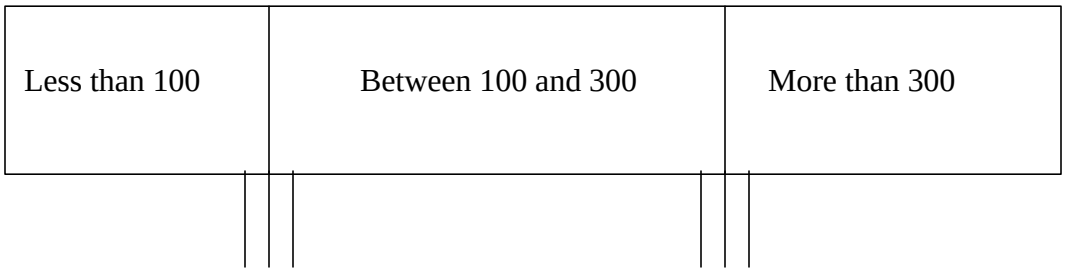
Equivalence class partitioning is just the start though. An important partner to this partitioning is boundary value analysis.

8.4.3 Boundary Value Analysis

"Bugs lurk in corners and congregate at boundaries." Boris Beizer [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 create test cases at the "edge" of the equivalence classes. In our above example, the boundaries of the classes are at 100 and 300, as shown in Figure 4. We should create test cases for the train length being 99, 100, 101, 299, 300, and 301. These test cases will help to find common off-by-one errors.

Figure 4: Boundary Value Analysis



When creating BVA test cases, consider the following [5]:

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, only one train is allowed to start in each direction on each station. 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).

8.4.4 Failure ("Dirty") Test Cases

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. 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?

8.5 Integrating Code and Testing Incrementally

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 feasible 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 can only be accomplished by integrating code and testing early and often.

As was said, often in industry programmers could keep their code secluded on their own machine for months at a time before letting the rest of the team see the code in the code base. Instead, we recommend that each developer integrates their code into the code base about once per day. Then, integration problems can be localized to the work that was done during that particular day. Localizing the code that contains a new defect is very important for efficient identification and removal of defects.

When running this kind of iterative integration testing and functional testing, it's best to start small. Hopefully, each developer is running some test cases on their own computer while he or she is developing the code. The developer shouldn't be integrating code into the code base unless his or her own test cases are working. Start by re-running these test cases, which should be tests that are isolated to the module that was just integrated – they should work when the code is placed in the code base. If these test cases don't work, then the cause of the defects should be found and fixed and possibly the new code should be backed out of the code base. If these test cases do work, more complex test cases should be run which stress more complex interactions between modules. The aim is to get to the point where you can incrementally run functional test cases that would let the team know that whole customer requirements are have been implemented successfully.

8.6 Acceptance Testing

> *Acceptance test cases are the ultimate documentation of a feature. Once the customer has written the acceptance test cases, which verify [at least preliminarily] that a feature is correct, the programmers can read those acceptance test cases to truly understand the feature.*
> *– Robert Martin [4]*

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 their own (software and hardware) environment. Sometime the customer shares the acceptance test cases with the team, which gives them a shared specific goal; sometimes 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 won'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.

In XP, customers explicitly work with the developers to specify the exact test cases they will run as acceptance test cases. The documentation of a requirement in XP is most certainly the union of the user story and the acceptance test cases that correspond to that user story. When these test cases are automated, they can then become compileable and executable documentation.

8.7 Black Box Test Case Automation

> *Acceptance test cases [can] serve as compileable and executable documentation of the features of the system. – Robert Martin [4]*

By their nature, black box test cases are designed and run by people who have no visibility to the inner workings of the code. Ultimately, system and acceptance cases are intended to be run through the product user interface (UI) so show that the whole product really works. Between not having knowledge of the inner workings of the software and needing to run through the UI, test automation can be difficult. However, the more automated testing can be, the easier it is to run the test case 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." [4]

If your role on the team is 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).

For example, consider that ultimately adding trains to the subway system is done through a UI. As a developer, it would be in your best interest to write some code to read train input from a file so that you can easily test inputting trains on a wide array of inputs. These test cases can be run many, many times almost effortlessly since you have automated them. This will enable the tests to automate their testing and will help you prepare for when the test group and the customer then starts to type in train input through the UI. For example, write some simple scripting code that will allow the testers to create files of train input, such as:

```
Danny
95
0.7
70
Cary
```

L

They can create these kinds of files and run the scripts without any knowledge of your code implementation. By using these scripts yourself, you will easily know if you broke "train input" when you added more functionality to the code. These automated test cases can get very extensive to truly simulate the kinds of things customers can do. For example, you can automate the creation of several trains, have them enter and exit stations, etc. If the developer automates the execution of the logic of these acceptance test cases to the extent possible, the developer can feel more confident that the code will behave properly when they run these test cases through the UI.

These automated test cases can also become compileable and executable documentation of what the system should do. When the code is enhanced or maintained in the future, these test cases should continue to be run so that the team can easily see if new code broke existing functionality. In this way, these test cases become a valuable asset of the software system.

8.8 In Summary

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 compileable and executable documentation of the system.

References:

[1] Beizer, B., *Software Testing Techniques*. Boston: International Thompson Computer Press, 1990.
[2] Beizer, B., *Black Box Testing*. New York: John Wiley & Sons, Inc., 1995.
[3] Boehm, B. W., *Software Engineering Economics*. Englewood Cliffs, NJ: Prentice-Hall, Inc., 1981.
[4] Martin, R. C., *Agile Software Development: Principles, Patterns, and Practices*. Upper Saddle River: Prentice Hall, 2003.
[5] Pressman, R., *Software Engineering: A Practitioner's Approach*. Boston: McGraw Hill, 2001.