

James McCaffrey

SOFTWARE TESTING:

Fundamental Principles and Essential Knowledge

SOFTWARE TESTING

Fundamental Principles and Essential Knowledge

James McCaffrey

Copyright © 2009 James McCaffrey
All rights reserved.

ISBN: 1-4392-2907-4

EAN13: 9781439229071

Visit www.booksurge.com to order additional copies.

CONTENTS

INTRODUCTION	1
CHAPTER 1 - SOFTWARE TESTING AND TEST CASES	5
1.1 Test Cases	5
1.2 Equivalence Classes	7
1.3 Boundary Conditions	9
1.4 Test Suites	11
1.5 Code Coverage	12
1.6 Mutation Testing	14
1.7 Static Code Analysis	15
CHAPTER 2 - FUNDAMENTAL MATHEMATICAL TECHNIQUES	19
2.1 Measures Of Central Tendency	19
2.2 Measures Of Variance	21
2.3 Mathematical Permutations	23
2.4 Mathematical Combinations	25
2.5 Data Encryption	26
2.6 Crypto-Hashing	29
2.7 Pair-Wise Testing	30
CHAPTER 3 - PROJECT MANAGEMENT CONCEPTS	35
3.1 The Software Development Life Cycle	35
3.2 Project Management Constraints	37
3.3 Beta Distribution Estimation	39
3.4 Test Effort Documentation	40
3.5 Source And Version Control Tools	42
3.6 Bug Reports	44
3.7 Reproduction Steps	46
CHAPTER 4 - CORE PRINCIPLES	49
4.1 Black Box And White Box Testing	49

4.2	Manual And Automated Testing	51
4.3	Stress And Load Testing	52
4.4	Configuration Testing	54
4.5	Localization Testing	56
4.6	Record-Playback Testing	58
4.7	Random Input Testing	61
CHAPTER 5 - ESSENTIAL KNOWLEDGE		65
5.1	Character Encoding	65
5.2	Xml And Testing	66
5.3	Http Request-Response Testing	68
5.4	Http State Testing	70
5.5	Testing Database-Centric Applications	73
5.6	Copying Files	75
5.7	Network Troubleshooting	77
CHAPTER 6 - UNITS, MODULES, AND COMPONENTS		81
6.1	Dynamic Link Libraries	81
6.2	The Global Assembly Cache	82
6.3	Module Testing	84
6.4	Unit Testing	86
6.5	Kernel Debugging	88
6.6	The Windows System Registry	90
6.7	Stub Testing And Mock Testing	91
ANSWERS TO REVIEW QUESTIONS		95
INDEX		107

INTRODUCTION

What This Book Is About

There are several good books about software testing available. So why yet another? Over the course of several years, in my role as a Contributing Editor for Microsoft's MSDN Magazine I interviewed hundreds of software development managers and test managers. A nearly universal comment from these managers was something along the lines of, "When we hire new software test engineers they really have to learn on the job. We have books that describe software testing in a very general way, and we have books that provide detailed technical information about specific testing techniques, but what we don't have is a book that covers what we consider to be core software testing principles." I then started asking software testing managers what they thought these core software testing principles are. Not too surprisingly, there were many topics mentioned. However, certain testing topics seemed to appear over and over again in these conversations. The topics in this book are the topics that were mentioned most often by experienced software test engineers as being on their personal list of fundamental principles and essential knowledge for new testers.

In most computer science related books, the contents of the book are summarized in the introduction. I will forego that tradition and say instead that the best way to get a feel for what is covered in this book is to scan the table of contents; I know that's what I always do. That said however, let me mention a few topics that distinguish this book from other books on testing. Chapter 2, "Fundamental Mathematical Techniques", presents a concise summary of key mathematical skills needed in a software testing environment. In addition to basic combinatorics, including pair-wise testing, the chapter contains a quick overview of statistical and cryptographic techniques. Chapter 3, "Project Management Concepts", describes useful techniques for managing various aspects of a software testing effort, including time and cost estimation. Chapter 6, "Units, Modules, and Components", explains how to test the primary building blocks of modern software systems, including an explanation of the differences between unit testing and module testing.

The topics in this book were written so that they are independent from each other for the most part so that you can read them in any order. Each topic has a quick self-assessment question that is intended to reveal if you grasped the key point of the topic. These reviews questions are very similar to the questions you might encounter on a software testing certificate exam.

Who This Book Is For

This book is intended for anyone who wishes to gain a quick, practical working knowledge of fundamental software testing principles, including software developers, project and program managers, students, support engineers, and of course software test engineers. Although the topics in this book will help you understand software testing, like many professions you can really only learn testing by actually testing software. The information in this book will provide you with a conceptual framework that will help you understand how different testing concepts and techniques are related, and enable you to learn software testing on the job more efficiently. This book is not intended to be a technical recipe-style collection of specific testing techniques, and this book is not an attempt to identify a definitive body of knowledge for software testing.

Based on my interviews and surveys, many experienced test engineers believe that great software testers have something usually called the testing mentality. There is no solid definition or description or unanimous agreement of exactly what this testing mentality characteristic is, but great software testers tend to approach testing differently from great developers. One of the traits commonly associated with a great software tester is the drive to break software. In other words, software test engineers enjoy trying to find errors in software and get satisfaction when they uncover a serious bug in a system. A closely related trait is the enjoyment of analysis. Great testers seem to take pleasure in examining exactly how a software system works. Another common perception is that great software testers have a creative aspect to their personalities which helps them exercise a system under test in interesting and unusual ways. And finally, most great software testers seem to have the propensity to be very detailed oriented and very methodical. A software testing

CHAPTER 1

SOFTWARE TESTING AND TEST CASES

Software testing is the process of investigating the quality of a software system. The most common activity in software testing is sending input to the system under test and then examining the result to determine a pass or fail result. The seven topics in the chapter deal with test cases, the single most important entity for all software testing activities.

1.1 Test Cases

A test case normally consists of at least the following items: a test case ID, a test case input, and a test case expected result or state. For example, suppose a component under test is simply a function `Sum(x, y)` which accepts two integer arguments, `x` and `y`, and returns their sum. A test case always produces a result of either pass or fail. In situations where input is sent to a software system and there is no expected result, and therefore does not produce a pass or fail result, the process is more accurately called test analysis or test exploration rather than a executing a test case. Data for a single test case for `Sum()` might look like:

```
ID: 001
input1: 3
input2: 4
expected: 7
```

If the test case is intended to be executed by test automation through the use of a test harness program, the test case might be part of a data file of test cases along the lines of:

```
001:3:4:7
002:2:2:4
003:0:6:6
etc.
```

If the test case is intended to be executed manually, the test case might be part of a document which also contains a list of step-by-step instructions for performing the test:

“Launch the current build of the system. Enter “3” into the text box labeled “input1”. Enter “4” . . . etc.

The sole purpose of a test case is to determine if there is an error of some sort in the system under test. In some academic circles the use of the term “bug” is frowned on, but it is commonly used in practice. Technically, terms such as anomaly, fault, error, failure, and defect are preferred by some academics but there is no consensus for the exact meanings of these terms. Therefore in practice the term bug has multiple meanings including an error in the source code of a system under development, and erroneous behavior of the SUT while executing.

Test case IDs are useful so that individual test cases can be combined into different collections called test suites. All test cases can be maintained in one central location and various test suites only need a list of the IDs of those test cases which make up each suite. Test case IDs are also useful to provide a reference to bugs which were found when the test case executed. For example, a bug tracking database can have a field in a bug report which holds the ID of the test case which revealed the bug.

Software utility programs called test case managers can be used to store and sometimes execute individual test cases. Test case managers can range from a simple spreadsheet file to sophisticated systems which not only store test case data, but also execute the test cases, store test case results, and maintain bug reports and other data.

In order for a test case to have an expected result or state, the system under test must start from some known state. For example, if you are testing the result of withdrawing money from a bank account, the expected result depends on the starting state of the account. Therefore, test case data must either assume some initial system state, or contain data to explicitly set the SUT to some state. Initialization data can range from simple, such as a single numeric value for a bank account module initial account balance, to complex, such as a large set of data for use in placing a SQL database into a known state. Similarly, determining the expected result or state of a SUT can range from simple, such as verifying a single numeric return value from a function, to complex, such as verifying the state of a database after some stored procedure has been executed.

The term test scenario is closely related to a test case and there is no clear distinction between the two. In general, a test case exercises the system under test to produce a single result or new state. A test scenario generally refers to situations where the SUT is manipulated through several state changes. For example, a test case might refer to a situation where a Web application is sent a single HTTP request, and the response stream is examined for an expected string of some sort. A test scenario might refer to a situation where a Web application is sent a series of inputs to generate a series of Web pages, and the final page is examined for an expected value. However, the distinction between test case and test scenario is arbitrary.

Review Question

Suppose you are testing a `PinballMachine` class which models a pinball machine and which is implemented using a language which supports object oriented programming (such as C#, C++, or Java). The `PinballMachine` class has a constructor which accepts a value representing an initial score, as well as `Increase()` and `Decrease()` methods which add points to and subtract points from a score member field of a `PinballMachine` object respectively. Which of the following sets of information best constitutes a reasonable test case?

- a. { test case ID, constructor argument, one argument for the `Decrease()` method, one argument for the `Increase()` method }
- b. { test case ID, constructor argument, two arguments for two calls to the `Decrease()` method, actual result value }
- c. { test case ID, constructor argument, three arguments for three calls to the `Decrease()` method, expected result value }
- d. { test case ID, constructor argument, comment }

1.2 Equivalence Classes

Equivalence class testing is a technique intended to reduce the number of test cases for a system under test without significantly reducing the effectiveness of the test effort. An equivalence class is a subset, or partition, of input values that are assumed to have roughly the same behavior in the sense that every value will either be handled

1.4 Test Suites

A test suite is a collection of test cases. Different test suites are typically used for different purposes. Build Verification Tests (BVTs) are a suite of tests run just after a new build of the system under development has been created. The purpose of a BVT test run is to determine whether or not the new build of the system meets a set of minimal requirements so that software test engineers can effectively test the build and so that development engineers can effectively continue to add new features and fix existing bugs. BVTs often contain one or more test cases which verify that the system under test can be successfully installed — if the system under development cannot be installed then it is unlikely that any effective testing or development can be performed. BVTs often contain a set of basic functionality tests for the same reason. Test cases in a BVT suite are generally automated tests so that the BVT suite as a whole can be automated as part of the build process. If a new build of the system under development passes all tests in the BVT suite, then team members are notified that the build is released to test. If the new build fails one or more tests in the BVT suite, then team members typically receive a message that the build has failed the BVT.

Developer Regression Tests (DRTs) are a suite of tests run against new code before the new code is checked into a source code control repository. DRTs are sometimes performed by individual developers after they write new code, or sometimes performed as part of an automated source code check-in system. The larger and more complex the system under development is, the more likely it is that DRTs will be used as part of the development process.

A full test pass suite generally consists of every current test case. Full test pass suites are run before any milestone type of release of the system under development, such as a beta release, an agile iteration release, or a major milestone (MM) release. A full test pass is also run before the final release of a product. Final releases are sometimes called RTM (release to manufacturing in the case where the system is transferred to physical media such as a DVD disk), RTW (release to Web in the case where the system is made available as a download via the Internet), or RTO (release to operations in the case where the system is a Web application and is turned over to a group in charge of hosting the application on Web servers).

CHAPTER 3

PROJECT MANAGEMENT CONCEPTS

The more complex a software testing effort is, the more likely it is that standard project management techniques will be useful to help the effort succeed. The seven topics in this chapter describe various project management principles and approaches which are useful in a software testing environment.

3.1 The Software Development Life Cycle

The software development lifecycle (SDLC) is a general term that refers to the overall process of creating a software system. There are several common categories of SDLC methodologies and many, many variations on each category. From a software testing perspective it is useful to categorize the type of SDLC methodology being used in a particular project, because different testing strategies and techniques are generally better suited for different types of methodologies.

The waterfall model is one of the oldest SDLC methodologies. The development process is organized into five distinct phases: requirements, design, implementation/coding, verification/testing, and release/maintenance. In the requirements phase, a highly detailed description of the system's goals is specified. The design phase results in a set of documents which describe the architectural details of the system. In the implementation phase, the system is actually coded by developers. The system is thoroughly tested in the verification phase. In the maintenance phase, the system under test is deployed and integrated with existing systems if necessary. In the waterfall model, each of the five phases is performed sequentially. The waterfall model tends to be well suited for software systems where requirements will not change during the development process, such as medical systems and weapons systems. In general, testing activities in a project which uses the waterfall methodology are highly structured, emphasizing the use of formal specification documents, detailed reporting, and traditional project management principles.

The spiral model is similar to the waterfall model in that there are a set of five general phases — requirements, design,

implementation/coding, verification/testing, and release/maintenance — however each set of these five phases is repeated several times. At the end of each cycle, a preliminary version of the system has been produced. Then the overall system requirements and design are revised to take into account experience gained. The final version of the system is typically released after three to five cycles through the requirements-through-release process. Each cycle of five phases takes roughly three to six months in many development scenarios. Software testing activities on a spiral model project are generally somewhat less formal than those on a waterfall model.

A relatively new variation of SDLC model called agile development became popular starting in the mid 2000s. Agile development can be thought of as an accelerated form of the spiral model. Again, the requirements, design, implementation/coding, verification/testing, release/maintenance activities are performed iteratively; however each iteration typically takes only two to four weeks. As with the waterfall and spiral models, there are countless variations on the agile theme. Some of the characteristics often associated with agile development are minimal reliance on formal written documentation, early and extensive customer feedback, test driven development (performing coding and testing simultaneously), and pair programming (developers always working in pairs). Scrum is a term which describes an approach to managing projects being developed using agile methodologies.

Two closely related forms of SDLC methodology called the Personal Software Process (PSP) and the Team Software Process (TSP) were developed in the mid 1990s. PSP/TSP is closely associated with the Capability Maturity Model (CMM) which is a highly structured set of guidelines for measuring the progress of the business processes associated with a software development project. CMM is in turn closely associated with the Software Engineering Institute (SEI), a hybrid federal-private organization which attempts to define software development best practices. PSP/TSP is an extremely structured approach characterized by large numbers of project managers, extensive use of form templates, and frequent review meetings.

Many product groups at Microsoft use a SDLC methodology which is sometimes called the synchronize and stabilize model, or simply

the Microsoft model. A primary characteristic of the Microsoft model is the production of a daily build of the system under development. Like the spiral model, the requirements through release set of activities are performed repeatedly. Phases of development have names such as minor milestone (mm), major milestone (MM), Alpha release, Beta release, and community technical preview (CTP). The final release activity is often called RTM (release to manufacturing), RTW (release to Web), or RTO (release to operations). The Microsoft SDLC model often uses dedicated software test engineers who work throughout the development process. There is often extensive use of both manual testing, generally performed by Software Test Engineers (STEs), and custom software test automation, generally written by Software Development/Design Engineers in Test (SDETs).

Review Question

Which of the following software development methodologies is least likely to be characterized by a test team which uses formal documentation and traditional project management techniques?

- a. A variation of the waterfall software development methodology paradigm.
- b. A variation of the spiral software development methodology paradigm.
- c. A variation of the agile software development methodology paradigm.
- d. A variation of the synchronize and stabilize software development methodology paradigm.

3.2 Project Management Constraints

In standard project management terminology, a project is an undertaking with defined start and finish dates that create a unique product or service. Many software development efforts as a whole, as well as software test efforts, fall into this definition. Arguably the most fundamental concept in project management is a principle known by various terms including the project management triple constraints, and the project management triangle. The three com-

tory contains a Key named NoSearchBox which has a default value of type REG_DWORD set to 0. By changing that value to 1, the search box will not be displayed. Registry values can also be manipulated using text-based .reg data files in conjunction with the regedit.exe tool.

The system registry data is stored in multiple locations on a test host machine. The exact physical locations of data stores which make up the system registry vary depending upon which version of Windows is running on the test host machine. Most variants of the Unix operating system do not have an equivalent to the Windows registry application program global data store. In general, Unix variants store application data as ordinary text files in the etc/ directory and its subdirectories, and store user data in the /usr directory or the user's ~ home directory. On Windows systems, the use of such text files is a possible alternative to the system registry, and these files usually have an .ini (initialization) or .config (configuration) file extension.

Review Question

Suppose you are testing some software application on a Windows system. The application behaves differently depending upon the identity of the current logged on user. You suspect the problem may be in the system registry. Which of the following system registry values is most likely the cause of the behavior?

- a. Hive=HKEY_LOCAL_MACHINE, Key=FullFunction, Value=REG_DWORD 0.
- b. Hive=HKEY_LOCAL_MACHINE, Key=FullFunction, Value=RegistryTypeString 0.
- c. Hive=HKEY_CURRENT_USER, Key=FullFunction, Value=REG_DWORD 0.
- d. Hive=HKEY_CURRENT_USER, Key=FullFunction, Value=RegistryTypeString 0.

6.7 Stub Testing and Mock Testing

Stub testing is the process of creating a small, relatively simple placeholder method with very little if any functionality which can

be called by the system under test. The term mock testing generally refers to a similar process where the placeholder is an OOP object and is created in part with the use of a code library. The concepts are best explained by example. Consider a system which makes airline flight reservations. A code snippet of a system which is designed using an OOP paradigm might look something like this:

```
Flight flt = new Flight("603");
string seat = "16e";
Reservation res = new Reservation();
res.Make(flt, seat);
// code to verify state
```

The first statement creates a `Flight` object, and the third and fourth statements make a seat reservation. Notice that in order to make a reservation, a `Flight` object must already exist. Now suppose you wish to test the method which makes a reservation but the `Flight` class has not yet been implemented. One solution is to create simple stub code along the lines of:

```
public class Flight {
    private string flightNumber;
    public Flight(string flightNumber) {
        this.flightNumber = "999";
    }
}
```

Here, the stub constructor merely assigns a dummy value to the flight number. Now a minimal `Flight` class exists and software test engineers can call and exercise the `Reservation.Make()` method to at least a minimal extent until the actual `Flight` class is implemented. In some testing scenarios stub code is created by testers with development skills, and in other scenarios testers make a request to the development team to create stub code for the purposes of testing. In general, the main problem with stub code is that because stub code has minimal logic and functionality, testing the methods in the SUT which rely on the stub code is often quite rudimentary. Of course stub code with complex functionality can be written but the level of development effort required to do so can be close to the level of effort necessary to implement the actual class being stubbed.

Mock object libraries are frameworks which help with the creation of stub code with some basic simulated functionality. Mock libraries exist for most common programming languages which support OOP design. Examples include NMock for the C# language and JMock for the Java language. Most mock libraries require that the system under development be designed using an interface-implementation paradigm. This means that each class is separated into an interface part which lists the methods in the class, and an implementation part which supplies the actual logic code. For example, testing the flight reservation system described above, where the Flight class has not yet been implemented might resemble something like:

```
IMock mockFlight = new
    DynamicMock(typeof(IFlight));
mockFlight.Expects( . . . );
string seat = "16e";
Reservation res = new Reservation();
res.makeFlight(mockFlight, seat);
mockFlight.Verify();
```

Notice that the creation of a mock object uses an interface. To summarize, in situations where classes in the SUT have not yet been implemented, testing with mock objects allows developers or test engineers to simplify the creation of placeholder code with a moderate amount of logic so that the system under development can be tested until the missing classes are completed.

Review Question

Which of the following statements is most often true about stub testing and mock testing?

- In general, stub methods are more complex than mock objects. Stub methods are typically created with the help of a framework library, and mock objects are typically coded from scratch.
- In general, stub methods are less complex than mock objects. Mock objects are typically created with the help of a framework library, and stub methods are typically coded from scratch.