SWEN 647: Project One

Unit Test

V1.0

07/03/2017

Alexander R. Ramsey

**Table of Contents**

1. Unit Test Introduction 1

1.2.1 Test Case 1 5

1.2.2 Test Case 2 5

1.2.3 Test Case 3 5

1.2.4 Test Case 4 5

1.2.5 Test Case 5 5

1.2.6 Test Case 6 6

1.2.7 Test Case 7 6

1.2.8 Test Case 8 6

1.2.9 Test Case 9 6

1.2.10 Test Case 10 6

1.2.11 Test Case 11 7

1.2.12 Test Case 12 7

1.2.13 Test Case 13 7

1.3 Additional Test Cases for Data Coverage 8

1.3.2 Test Case 2 8

1.3.3 Test Case 3 8

1.3.4 Test Case 4 8

1.3.5 Test Case 5 9

1.3.6 Test Case 6 9

2. Defects 10

2.1 Defects 10

2.1.1 Defect 1 10

2.1.2 Defect 2 10

2.1.3 Defect 3 10

2.1.4 Defect 4 10

2.1.5 Defect 5 10

2.1.6 Defect 6 11

# 1. Unit Test Introduction

**Introduction to Testing**

Testing is a highly valuable and necessary part of the software development process. It seeks to validate and verify desired outcomes for the product. Testing should be completed early and often throughout the lifecycle.

The Systems development lifecycle identifies major phases of software development. The final of these phases is the testing and maintenance phases. According to Unit Testing, testing in software development should be a process which is initiated throughout the entire systems development model. It should occur during and after development has been completed rather than as an iterative step toward the end of the process, as some may misunderstand.

**White Box versus Black Box Testing**

Testing of completed units of a congruent application product is known as black box testing. This differs from white box testing, where disparate units are the focus of their own testing. White box testing is most relatable to unit testing because it aims to focus on specific chunks of the application, rather than taking a holistic view of functionality. White box testing is often also referred to as clear box testing. The reasoning behind white or clear in the naming convention describes the differential of perspective taken during the testing. Black box means that the application is not seen internally, it is tested based upon an external and qualitative perspective. Clear or white box testing takes on an internal focus of the application.

**Verification and Validation (V&V)**

Most testing occurs in the scope of verification testing. Verification testing involves checking that requirements to ensure their completion. Validation testing involves techniques such as usability testing to ensure the product meets expectations outside of the requirements specifications. Activities unrelated to Unit Testing that involve V&V include Planning, Requirements review, Acceptance Planning, Risk Assessment, and Defect Tracking & Analysis.

**Unit Testing Purpose**

The purpose of Unit Testing is to identify the working nature of specific units within a software program. A unit may be singular method, class constructor, or set of functionality that is pertinent to the working of the software. By breaking down tests into units, individual components of the application can be tested accordingly. Once individual Units are compiled, Integration testing can occur. This ensures that multiple Units work together in conjunction to form and execute complex groups of tasks. Unit Testing works to identify flaws in specific parts of code that may be overlooked or affected later during other methods of testing and V&V. In general, the software development team is responsible for generating Unit Tests.

**Goals and Objectives for Unit Testing**

Identify flaws in the software pertinent to White Box testing throughout all stages of development and hopefully long before shipment to production. For Unit Tests to be effective you must employ a Code Coverage and Data Coverage strategy. Code Coverage involves ensuring that Unit Tests are performed throughout each line of code. Data Coverage involves ensuring that the data which filters through the application is handled in a correct manner as to ensure proper representation. Achieving Unit Test completion is based upon achieving the set percentage of Code and Data Coverage. For the purposes of this project, we will seek to attain a 100% code and data coverage. To achieve this, each line of code including exception handling for methods must be defined as a unit test case. Data coverage is reached by testing parameter values through the setting of defined boundaries. These will seek out anomalous data issues like negative or zero numbers which may adversely affect the application.

Some criteria to keep in mind to attain 100% code coverage:

* Test every statement. Each method that is executed must be tested.
* Test every branch and clause. All conditional outcomes must have an ending test.
* Test every path. Every possible path of execution and data must be considered.

Like all forms of testing, Unit Testing being complete may never be an achievable and realistic goal. Resources devoted to the achievement of project results must be weighed to ensure a good majority of code coverage is agreeable among all involved parties. These code coverage criteria must be defined before unit testing is performed so that attainable goals are set relevant to the project’s completion.

**Advantages of Unit Testing**

* Problems are found early and often, before costly changes must be made.
* Encourages change and flexibility in code design.
* Integration of disparate units is cleaner and simplified.
* Test driven development can benefit code quality.

**Disadvantages of Unit Testing**

* Does not encompass other areas of necessary testing. For example: performance.
* Significant increase in codebase size due to additional test cases.
* Difficulty in paralleling Unit Tests with realistic, needed outcomes.
* Unit tests that fail may be a false negative/false positive. This can steer the development team in the wrong direction or devote too much resource to the test development itself.

**Test Characteristics to Ensure Unit Test Effectiveness**

As outlined in our class notes, some good characteristics of Unit Tests according to Robert Martin’s book, *Clean Code* are:

* Fast. Easy to run in a finite and predictable period.
* Independent. Pertinent to the Unit at hand without dependence on another.
* Repeatable. Able to run in any order and at any time.
* Self-validating. Pass and Fail should be indicated via a Boolean expression as the final output of the test result.
* Timely. Tests are generated and completed during development process.

Additional good characteristics via our class notes according to Andrew Hunt and David Thomas’s book, *Pragmatic Unit Testing: In Java with Junit* are:

* Automatic. Must be written to accommodate and recognize differing states and outputs.
* Thorough. End to end coverage of all possible points of failure including error handling.
* Repeatable. Like above, able to run in any order and at any time.
* Independent. Like above, pertinent to the Unit at hand with dependence on another.
* Professional. Best practices for development are followed to ensure readability and reliability.

**Generating Tests with Junit**

Each class should be broken down into individual parts. Methods and statements for each class should be touched on to ensure complete code coverage. In addition, all paths for data to flow through should be noted. A brief description or comment can serve as a structure for the test. Tests for each method should be executed when the method is called throughout the application.

Example code documentation:

Class: Database Object

Method: createRecord(Data), readRecord(Data), updateRecord(Data)

Test Cases:

* Check creation of data record in database
* Check read of data record from database
* Check update of data record in database
* Check data handling of Data passed into method
* Check data handling of Data passed to database

**Handling Defects**

Unit Testing is the not the only source of discovery for defects. Others include syntax errors during compilation, run time errors, code review best practice issues, and functional and system testing throughout the V&V process. When a defect is found, it should be handled appropriately. Handling defects mainly involves communicating the reason for the defect so that an adjustment can be made. Unit Testing should feed results into a Defect Tracking System. These systems allow the entirety of the team to see and prioritize defects according to their severity or importance to the end client. A good defect report will outline have the following characteristics, according to Nick Jenkins in his article, *A Software Testing Primer*:

* Objective. Defect documentation does not criticize work, it outlines the current state and the desired state.
* Specific. One report should be logged per defect.
* Concise. Language should be minimal enough while explaining the cause of the defect.
* Reproducible. How to reproduce a defect in steps is an important part of garner mutual understanding of the defect.
* Explicit. Clearly state information about application functionality.
* Persuasive. Establish buy in for finding a fix with the development team.

Unit Testing is generally accomplished by a single developer or an agile team with pairs for code review. A test should be assigned a class name, method name, and should provide an assertion based on an inputted value. A defect is determined to have been found when the unit test outcome does not align with the expected result. Defects are typically handled by a team of developers or a single developer. When the defect is fixed, an additional round of unit testing should be performed to ensure that the original issue has been resolved.

**Unit Testing Automation**

Automation of Unit Testing can be employed by using a specialized IDE toolset or other tool to manage results and test occurrence. For the purposes of this paper, the Junit testing tool will be used with Eclipse IDE. Automation is helpful when the tests are developed alongside the functionality. However, it does not always serve to be as useful as manual testing and it does not mean that every aspect of testing should be automated.

1.2 100% Code Coverage Test Cases

### 1.2.1 Test Case 1

Test case name: testMyFaxDefaultConstructorStringType

Method being tested: MyFax()

Short Description: Assigns “None” String value to type.

Input Data to constructor and/or method you are testing: None

Expected Results: The type variable shall be assigned as a String with a value of “None”.

### 1.2.2 Test Case 2

Test case name: testMyFaxDefaultConstructorStringSerial

Method being tested: MyFax()

Short Description: Assigns “None” String value to serial variable.

Input Data to constructor and/or method you are testing: None

Expected Results: The serial variable shall be assigned as a String with a value of “None”.

### 1.2.3 Test Case 3

Test case name: testMyFaxDefaultConstructorStringNumber

Method being tested: MyFax()

Short Description: Assigns “None” String value to type and serial variable. Assigns 10 zeros to number variable.

Input Data to constructor and/or method you are testing: None

Expected Results: The number variable shall be assigned as a String with a value of “0000000000”.

### 1.2.4 Test Case 4

Test case name: testMyFaxConstructorStringTPositiveArg

Method being tested: MyFax(String t, String s)

Short Description: Checks value of the t String parameter to ensure not blank. If the values are correctly formatted then the t String will be assigned to the String variable type.

Input Data to constructor and/or method you are testing: t = 100, s = 50

Expected Results: The type variable shall be assigned a value of 100.

### 1.2.5 Test Case 5

Test case name: testMyFaxConstructorStringSPositiveArg

Method being tested: MyFax(String t, String s)

Short Description: Checks value of the s String parameter to ensure not null. If the values are correctly formatted then the s String parameter will be assigned to the String variable serial.

Input Data to constructor and/or method you are testing: t = 100, s = 50

Expected Results: The serial variable shall be assigned a value of 50.

### 1.2.6 Test Case 6

Test case name: testMyFaxConstructorNumberVarPositiveArg

Method being tested: MyFax(String t, String s)

Short Description: Checks value of the t String parameter and the s String parameter to ensure not blank and not null, respectively. If the values are correctly formatted then the number String variable will be assigned a value of “0000000000”.

Input Data to constructor and/or method you are testing: t = 100 and s = 50

Expected Results: The number variable shall be assigned as a String with a value of “0000000000”.

### 1.2.7 Test Case 7

Test case name: testMyFaxConstructorStringTNegativeArg

Method being tested: MyFax(String t, String s)

Short Description: Checks value of the t String parameter and the s String parameter to ensure not blank and not null, respectively. Tests that when they are an IllegalArgumentException is thrown.

Input Data to constructor and/or method you are testing: t = “” and s = 100

Expected Results: A IllegalArgumentException is thrown with the return message “Parameter cannot be null or empty”. Variable type = “None”.

### 1.2.8 Test Case 8

Test case name: testGetNumber

Method being tested: getNumber()

Short Description: Returns the value of the number attribute.

Input Data to constructor and/or method you are testing: None

Expected Results: Variable number = “000000000000”

### 1.2.9 Test Case 9

Test case name: testGetType

Method being tested: getType()

Short Description: Returns the value of the type attribute.

Input Data to constructor and/or method you are testing: None

Expected Results: Variable type = “None”

### 1.2.10 Test Case 10

Test case name: testGetSerial

Method being tested: getSerial()

Short Description: Returns the value of the serial attribute.

Input Data to constructor and/or method you are testing: None

Expected Results: Variable serial = “None”

### 1.2.11 Test Case 11

Test case name: testSetNumberPositiveArg

Method being tested: setNumber(String value)

Short Description: Checks value of the value String parameter to ensure it contains ten digits and no leading zeros. If they meet the criteria then the number String attribute is assigned the String parameter value.

Input Data to constructor and/or method you are testing: value = “1234567891”

Expected Results: The number String attribute should equal “1234567891”

### 1.2.12 Test Case 12

Test case name: testSetNumberNegativeArgNonDigit

Method being tested: setNumber(String value)

Short Description: Checks value of the value String parameter to ensure it contains ten digits and no leading zeros. This test checks that a non digit in the value parameter throws an IllegalArgumentException.

Input Data to constructor and/or method you are testing: value = “12345A7891”

Expected Results: A IllegalArgumentException is thrown with the message “Parameter cannot have non-digits”.

### 1.2.13 Test Case 13

Test case name: testSetNumberNegativeArgLeadingZero

Method being tested: setNumber(String value)

Short Description: Checks value of the value String parameter to ensure it contains ten digits and no leading zeros. This test checks that a leading zero in the value parameter throws an IllegalArgumentException.

Input Data to constructor and/or method you are testing: value = “0123456789”

Expected Results: A IllegalArugmentException is thrown with the message “Parameter must have 10 digits and cannot have leading 0s”.

## 1.3 Additional Test Cases for Data Coverage

**1.3.1 Test Case 1**

Test case name: testMyFaxConstructorDataInputStringT

Method being tested: MyFax(String t, String s)

Short Description: Checks value of the t String parameter and the s String parameter to ensure not blank and not null, respectively. Tests that when they are an IllegalArgumentException is thrown.

Input Data to constructor and/or method you are testing: t = null and s = 100

Expected Results: A IllegalArgumentException is thrown with the return message “Parameter cannot be null or empty”. Variable type = “None”.

### 1.3.2 Test Case 2

Test case name: testMyFaxConstructorDataInputStringS

Method being tested: MyFax(String t, String s)

Short Description: Checks value of the t String parameter and the s String parameter to ensure not blank and not null, respectively. Tests that when they are an IllegalArgumentException is thrown.

Input Data to constructor and/or method you are testing: t = 50 and s = “”

Expected Results: A IllegalArgumentException is thrown with the return message “Parameter cannot be null or empty”. Variable serial = “None”.

### 1.3.3 Test Case 3

Test case name: testMyFaxConstructorStringSNegativeArg

Method being tested: MyFax(String t, String s)

Short Description: Checks value of the t String parameter and the s String parameter to ensure not blank and not null, respectively. Tests that when they are an IllegalArgumentException is thrown.

Input Data to constructor and/or method you are testing: t = 50 and s = null

Expected Results: A IllegalArgumentException is thrown with the return message “Parameter cannot be null or empty”. Variable serial = “None”.

### 1.3.4 Test Case 4

Test case name: testMyFaxConstructorNumberVarNegativeArg

Method being tested: MyFax(String t, String s)

Short Description: Checks value of the t String parameter and the s String parameter to ensure not blank and not null, respectively. Tests that when they are an IllegalArgumentException is thrown.

Input Data to constructor and/or method you are testing: t = “” and s = null

Expected Results: A IllegalArgumentException is thrown with the return message “Parameter cannot be null or empty”. Variable number = “0000000000”.

### 1.3.5 Test Case 5

Test case name: testSetNumberDataInputStringValueNonDigit

Method being tested: setNumber(String value)

Short Description: Checks value of the value String parameter to ensure it contains ten digits and no leading zeros. This test checks that a non digit in the value parameter throws an IllegalArgumentException.

Input Data to constructor and/or method you are testing: value = “12345B7891”

Expected Results: A IllegalArgumentException is thrown with the message “Parameter cannot have non-digits”.

### 1.3.6 Test Case 6

Test case name: testSetNumberDataInputStringValueDigitCount

Method being tested: setNumber(String value)

Short Description: Checks value of the value String parameter to ensure it contains ten digits and no leading zeros. This test checks that a non digit in the value parameter throws an IllegalArgumentException.

Input Data to constructor and/or method you are testing: value = “123456789012”

Expected Results: A IllegalArgumentException is thrown with the message "Parameter must have 10 digits and cannot have leading 0s”

# 2. Defects

This section lists all identified defects.

## 2.1 Defects

### 2.1.1 Defect 1

**Method/Constructor:** *MyFax(String t, String s)*

**Defect Description**: *Based upon the input values, an IllegalArgumentException should have been thrown. An IllegalArgumentException was not thrown and instead a NullPointerException was thrown.*

**How Found**: *testMyFaxConstructorStringTNegativeArg() identified the defect.*

### 2.1.2 Defect 2

**Method/Constructor:** *setNumber(String value)*

**Defect Description**: *Based upon the input values, an IllegalArgumentException should have been thrown. An IllegalArgumentException was not thrown and the leading zero was not detected in the input value as being a false value.*

**How Found**: *testSetNumberNegativeArgLeadingZero() identified the defect.*

### 2.1.3 Defect 3

**Method/Constructor:** *MyFax(String t, String s)*

**Defect Description**: *Based upon the input values, an IllegalArgumentException should have been thrown. An IllegalArgumentException was not thrown and the blank input value was not detected as a false value.*

**How Found**: *testMyFaxConstructorDataInputStringS() identified the defect.*

### 2.1.4 Defect 4

**Method/Constructor:** *MyFax(String t, String s)*

**Defect Description**: *Based upon the input values, an IllegalArgumentException should have been thrown. An IllegalArgumentException was not thrown and the null input value was not detected as a false value.*

**How Found**: *testMyFaxConstructorDataInputStringT() identified the defect.*

### 2.1.5 Defect 5

**Method/Constructor:** *MyFax(String t, String s)*

**Defect Description**: *Based upon the input values, an IllegalArgumentException should have been thrown. An IllegalArgumentException was not thrown and instead a NullPointerException was thrown.*

**How Found**: *testMyFaxConstructorStringSNegativeArg() identified the defect.*

### 2.1.6 Defect 6

**Method/Constructor:** *MyFax(String t, String s)*

**Defect Description**: *Based upon the input values, an IllegalArgumentException should have been thrown. An IllegalArgumentException was not thrown and instead a NullPointerException was thrown.*

**How Found**: *testMyFaxConstructorNumberVarNegativeArg() identified the defect.*