1. Introduction

Writing software is a process where developers take a vision of what a program should be (the vision is usually provided by a customer – in your case the instructor), and then write code that implements the vision. However, the code that is written may not exactly match the vision of what the program should be, resulting in lost revenue for a company (or for this class, lost points on an assignment).

Software testing is a process for identifying software faults. A software fault, or logic error, is defined as “an incorrect step, process, or data definition in a program” [1]. By systematically testing software we can identify faults in our code and correct those faults as early as possible. The goal is to remove as many faults as possible before releasing the application to our customer. The International Organization for Standardization (ISO) defines software testing as “the dynamic verification of the behavior of a program on a finite set of test cases, suitably selected from the usually infinite execution domains, against the expected behavior” [2]. The keywords here are:

- Dynamic: meaning we execute the code under test;
- Suitably selected: meaning we systematically and deliberately choose test inputs; and
- Expected behavior: meaning we know what we want the program to do.

Software faults typically originate from a programmer mistake, which is “a human action that produces and incorrect result” [1]. When the mistake is written in the requirements, design, or code, the mistake becomes a fault. Faults remain latent in the software until running the software produces a failure. A software failure is “an event in which a system or system component does not perform a required function within specified limits”[1]. Failures are identified when a program’s behavior deviates from the expected program behavior. Testing is used to identify failures in a program’s behavior. Investigation of the failure (guided by the failing test case) will lead to one or more underlying faults in the software that must be fixed.

Testing consists of many executions of a program and each execution has a different input. A distinct execution is called a test case.

A test case contains the following information:

1. A unique identifier;
2. A set of program input(s);
3. The expected output(s) from executing the program input(s); and
4. The actual results of running the test case.

A unique identifier allows for a common name when discussing test cases and test case execution results. A test case should have a set of one or more program inputs. These inputs should be specific and repeatable. Both characteristics are required so that anyone can execute the test cases. Input into the test case must be specific so that the test cases can be repeated with reliable results. Test cases must be repeatable so that any failures can be observed by any interested party. If specific inputs are
not recorded, then you may spend more time trying to rediscover broken functionality, rather than spending time trying to fix functionality. The expected outputs from a test case must also be specific for clarity in determining if a test passes or fails. The actual results records what happens when the test case is run. Any difference in expected and actual results represents a failing test case.

Program requirements document what the customer wants a program to do and represent our expected results. If a test case fails, meaning the actual results of the program do NOT match the expected results, then we have not met our requirements. A test failure implies a debugging session to find the underlying fault. Figure 1 shows the flow for generating and running test cases on a program.

![Diagram](image)

**Figure 1:** Test generation and execution flow.

Developers start by writing their program and test cases. These tasks can be done simultaneously or asynchronously depending on personal preference; however, both the source code and the test cases are required before a test can be run on a program.

A test case is run by executing the program with the input(s) specified by a test case. If the actual output does not match the expected output, you’ve uncovered a fault. Now, you’ll need to study your source code and fix your mistake.

If you test cases pass and you still have more time before the deadline, you should continue to test your program. While software testing is useful for finding underlying faults, no amount of testing can unequivocally demonstrate the absence of faults in your code. Additional testing will provide some confidence that your program will perform as expected most of the time, at least for the cases you tested.

There are two basic testing techniques: black box testing and white box testing. The subsections about these basic testing techniques include strategies for maximizing the effectiveness of testing. For you,
the student, the cost of a failure (and the underlying fault) is a few points off of your programming assignment, but for large corporations, the cost of releasing a failure to users can be in the millions, if not billions, of dollars.

In the following sections, we will use the following example to illustrate black box and white box testing:

A client wants to collect information about the types of characters users use in a String input to a program. Specifically, the client wants to collect the number of times a digit between a minimum and maximum value, inclusive, is used in String input. The user specifies the string input, a minimum, and a maximum value. If the minimum and maximum values are not valid digits or if the maximum is less than the minimum, the error message, “Invalid bounds” is displayed to the user. Otherwise, the count of digits between the minimum and maximum values, inclusive, is displayed to the user in the following output: “The String contains X digits between minimum and maximum, inclusive.”

5. Black Box Testing

Black box testing, which is also called functional testing, ignores the internals of the program and instead focuses on the inputs to the program and the generated outputs. The program is treated as a “black box” by the tester; the process that generates the program output is unknown. The tester identifies the expected program output from the requirements and can compare the actual output with the expected output. Figure 2 presents a pictorial representation of black box test cases.

![Diagram of Black Box Testing](image)

**Figure 2:** Black box testing. The executable program, a black box, only considers the program’s input and output without regard to the internal workings (source code) of the program.

Black box testing can be used to find the following types of failures [3]:
- Incorrect or missing functions,
- Interface errors,
- Errors in data structures or external data base access,
- Behavior or performance errors, and
- Initialization and termination errors.

Typically, black box tests are run by an unbiased third-party; not the programmer who developed the code. However, in the absence of the third-party tester, a developer can test their own code by ignoring the internals. One good strategy for effective black box testing is to write the test cases before starting the code. Black box tests test what the customer wants the program to do, and are obtained from the program requirements.
Like all test cases, black box test cases, have an identifier, a set of specific and repeatable inputs, a set of expected outputs, and then the actual results from executing the test case. The test information can be organized into a Black Box Test Plan document using table in Table 1.

<table>
<thead>
<tr>
<th>Test ID</th>
<th>Description</th>
<th>Expected Results</th>
<th>Actual Results</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The test ID column of the test plan table contains a unique identifier for the test case. The unique identifier provides a common language for talking about test cases with developers and testers. The test ID could be a number, or a descriptive phrase about the test case. The test ID column can also contain additional information like the test author’s name, the requirement under test, the type of test, etc.

The description column contains the steps and inputs the tester enters into the program. Any information about how to prepare to start the test should also be included. The information may also include preconditions for running the test like another test that should pass before running the current test. The information that is recorded in the description should be specific and repeatable, which means that actual values are given. Instead of saying, “enter a dollar amount between 10 and 20” the description should say “enter 15.” For black box testing, the description provides all inputs a user must enter into a program.

The expected results column reports what the program should do. Like the description column, the expected results should be specific. For black box tests cases, the expected results report what the program presents to the user through the user interface.

The actual results column is filled in when the test is executed. Ideally, the actual results should match the expected results. When they do not match, the test fails, and work to debug the program commences. Failing and passing test cases can be rerun after debugging. Passing tests should be rerun to ensure that any changes to the program under test did not cause a new fault.

We are covering three black box testing strategies: test customer requirements, equivalence partitioning, and boundary value analysis. Some of the test cases generated by one strategy may also be generated by another strategy.

2.1 Test Customer Requirements
When testing a program, you should ensure that all of the customer requirements are tested at least once. Our string analyzer program has the requirement that users should be able to enter a string of text and two values specifying a minimum and maximum value. The count of the digits between the minimum and maximum values is returned. The following test case in Table 2 would test that requirement. The bold text in the description shows the values the tester would enter into the StringAnalyzer program at the given prompts.
### Table 2: Test customer requirements example

<table>
<thead>
<tr>
<th>Test ID</th>
<th>Description</th>
<th>Expected Results</th>
<th>Actual Results</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>validInput</code> (Heckman)</td>
<td>Preconditions: StringAnalyzer program started</td>
<td>The String contains 2 digits between 1 and 7, inclusive.</td>
<td></td>
</tr>
<tr>
<td>Test Type: Requirements, EC on bounds</td>
<td>String? <strong>A string with 3 numbers 4 9.</strong> Minimum? 1 Maximum? 7</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### 2.2 Equivalence Partitioning

Equivalence partitioning is a testing strategy where the input space is broken up into equivalence classes. There is only a limited amount of time to test and an infinite number of possible test cases. We want to focus on test cases that will uncover new errors, not test cases that are essentially equivalent to other test cases. We can divide the input space in to ranges or categories of data that we want to test from, and generate test cases by picking one representative value from each of the possible inputs ranges or categories. The program under test treats all possible inputs within an equivalence class the same; therefore, testing one representative value in each equivalence class covers the other possible inputs.

We can divide StringAnalyzer input into the equivalence classes shown in Figure 3. There are two types of String input: input with digits and input with no digits. The invalid String input equivalence class is generated when we consider invalid inputs. We end up with three equivalence classes and we can select a representative value from each equivalence class to create three test cases that will test general String input into the StringAnalyzer program. The requirements test shown in Table 2 also tests the valid “String input with digits” equivalence class. Since we already test that equivalence class we can focus on writing additional tests for the other equivalence classes. To help minimize the amount of testing, one test can be used to test requirements and equivalence classes. Table 3 presents an example test case for no input.

<table>
<thead>
<tr>
<th>No Input</th>
<th>String input with digits</th>
<th>String input with no digits</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Figure 3:</strong> Equivalence classes for String input</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### Table 3: Test equivalence class example

<table>
<thead>
<tr>
<th>Test ID</th>
<th>Description</th>
<th>Expected Results</th>
<th>Actual Results</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>noInput</code> (Heckman)</td>
<td>Preconditions: StringAnalyzer program started</td>
<td>The String contains 0 digits between 1 and 7, inclusive.</td>
<td></td>
</tr>
<tr>
<td>Test Type: EC on No Input</td>
<td>String? Minimum? 1 Maximum? 7</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
The following guidelines are helpful for defining equivalence classes [3]:

- If the input condition is a range, you’ll create one valid and two invalid (above and below the valid range) equivalence classes.
- If input conditions are specific values, then the values are each a valid (for that value) equivalence class and all other values are in an invalid equivalence class.
- If input conditions are members of a set, then the members of the set are in the valid equivalence class and all other possibilities are in the invalid equivalence class.
- If an input condition is a Boolean, there is one valid and one invalid equivalence class.

Representative values for an equivalence class are the specific valid/invalid values or a middle value in a range. If valid or invalid input is a range of numbers, choose a value that is in the middle of the range, rather than a value at range boundary. The boundary values are considered next.

The above example identified three equivalence classes for the input space associated with the String the user enters. Other equivalence classes should also be considered. The input space can also be divided into the following sets of equivalence classes:

- single digit and non-single digit input for the minimum bound;
- single digit and non-single digit input for the maximum bound; and
- equivalence classes where min <= max or max < min.

2.3 Boundary Value Analysis

Programmers tend to make mistakes at the boundaries of equivalence classes. Think of the mistakes you may have made when specifying the start and end points of a for loop. Boundary value analysis guides the creation of test cases at the boundaries or edges of a problem. When testing a boundary, you want to test the boundary itself. You also want to test the values immediately on either side of the boundary.

For the StringAnalyzer program we only want to accept a single digit for the minimum bound. Therefore, our valid input is values between 0 and 9, inclusive. Therefore, we have two boundaries: 1) at 0 and 2) at 9. We also want to test to either side of the boundary. Figure 4 presents the equivalence classes for the possible numeric input, and identifies the six boundary values to test. Table 4 presents a test case at the 0 boundary.

<table>
<thead>
<tr>
<th>Test ID</th>
<th>Description</th>
<th>Expected Results</th>
<th>Actual Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>zeroMinBound</td>
<td>Preconditions: StringAnalyzer program started</td>
<td>The String contains 3 digits between 0 and 7, inclusive.</td>
<td></td>
</tr>
<tr>
<td>(Heckman)</td>
<td>String? String 345 String.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Test Type:</td>
<td>Minimum? 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BVA</td>
<td>Maximum? 7</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 4: Boundary values for minimum bound input

Table 4: Test equivalence class example
3. White Box Testing

With white-box testing, the code under test is known, and we can use what we know about the code and the paths through the code to guide our tests. Additionally, our test cases and knowledge about the code can help guide us to the code locations likely to contain errors. White box testing techniques allow the tester to [3]: 1) exercise independent paths within the source code; 2) exercise logical decisions as both true and false; and 3) exercise loops at their boundaries. Later classes in the CSC curriculum will provide instruction on using white-box testing to test internal data structures.

3.1 Control Flow Diagrams

Typically, when performing white box testing, you want to focus on testing individual methods. This type of testing is called unit testing. Unit testing means that we are testing a specific unit of code, in our case, individual methods. The logic of the StringAnalyzer program iterates over a String and counts the number of digits that are between a specified minimum and maximum values, inclusive. The method, called countDigits should accept three parameters: 1) the String input, 2) the minimum bound, and 3) the maximum bound. The method will return the count of digits between the minimum and maximum bounds, inclusive. The code for countDigits is listed in Figure 5.
/**
 * Analyzes String input from a user.
 * @author Dr. Sarah Heckman (sarah_heckman@ncsu.edu)
 */
public class StringAnalyzer {

/**
 * Counts the number of digits in the String between
 * the specified min and max values, inclusive. If the
 * min and/or max value is outside the range 0-9, or if
 * the min is greater than the max, -1 is returned.
 * @param line the user’s input
 * @param min the min bound
 * @param max the max bound
 * @return the count of digits between min and max,
 *         inclusive
 */
public static int countDigits(String line, char min,
char max) {
    int digitCount = 0;
    if (checkBounds(min, max)) {
        for (int i = 0; i < line.length(); i++) {
            char c = line.charAt(i);
            if (c >= min && c <= max) {
                digitCount++;
            }
        }
    } else {
        digitCount--;
    }
    return digitCount;
}

Figure 5: Implementation of the countDigits method in the
StringAnalyzer program.

When performing white-box testing we are interested in the control flow of the program. The control
flow is a graph that contains decision points and executed statements. The control flow for the
countDigits method is shown in Figure 6.
Each decision (conditional test in an if statement) in the method is shown as a diamond. The statements are in rectangles.

There are standard templates for each of the control structures that make decisions in our code. These templates are provided in Figure 7a-g.
If the if statement contains compound conditional tests (i.e. the conditional tests are separated by && or ||) then each conditional test within the compound statement is shown as a separate diamond. Figure 8a-b shows standard templates for conditional statements with compound predicates. Figure 8a shows two predicates that are anded together. Both statements have to true for the statement on the right to execute. If either predicate is false, then the inner portion of the conditional test will never execute. Figure 8b shows two predicates that are ored together. Either statement can be true for the body of the conditional (represented by the lower statement) to execute. If both statements are false, then the body of the conditional test never executes.
A measure called cyclomatic complexity, provides a guide for the number of possible paths through the code. When creating white box tests, we want to create a test case for each possible path through the code. There are several calculations for cyclomatic complexity, but the easiest is to add 1 to the number of decision nodes (diamonds) in the control flow graph. Therefore, the cyclomatic complexity of the control flow graph for `countDigits` is 4 diamonds + 1 = 5, implying there are 5 potential paths through the method and 5 potential tests in the minimum test set. Cyclomatic complexity provides us with the lower bound for white box testing. That is the minimum set of test cases that we should write. Additional test cases can be written using equivalence class partitioning and boundary value analysis like we did for black box testing, above.

For the control flow graph in Figure 6, the paths are as follows:

- 1 – 2 – 8 – 9
- 1 – 2 – 3 – 9
- 1 – 2 – 3 – 4 – 5 – 3 – 9
- 1 – 2 – 3 – 4 – 5 – 6 – 3 – 9
- 1 – 2 – 3 – 4 – 5 – 6 – 7 – 3 – 9

Now, that we have the paths, we can create input values that will test each of the paths. The section on writing white box test cases, below, will outline the process that we will use to programmatically write white box test cases.

Because our program contains a loop, we could create many more paths that would execute the loop more than once, leading to a potentially infinite number of test cases. There is typically not enough time to run all possible test cases, so only focus on the paths through the code where a loop is run once through its body.

Pressman [3] provides the following guidance for testing a simple loop (i.e., no nesting), where the loop is expected to iterate $n$ times:

- Fail the conditional test for entering the loop, so that the loop never executes;
- Execute the body of the loop only once;
- Execute the body of the loop twice;
- Execute the body of the loop $m$ times, where $m < n$;
• Execute the body of the loop \( n - 1 \) times;
• Execute the body of the loop \( n \) times; and
• Execute the body of the loop \( n + 1 \) times.

A loops execution ranges from the lower boundary to \( n \). The first set of three test cases test the loop’s lower boundary value. The 4th test case is a representative equivalence class test of the loop’s input range. The last 3 test cases test loop’s upper boundary value. Some of the tests may lead to redundancies if the loop’s bounds are dependent on the input, so create as many distinct tests as possible, ensuring at a minimum coverage of all paths through the loop.

Analyzer

Nested loops introduce additional complexity when testing. Pressman [3] gives the following guidance for testing nested loops:
• Keeping all outer loops to minimal values that reduce the number of iterations, test the innermost loop using the techniques listed above.
• Move up the level of nested loops, and test the loop using the techniques listed above. The outer loops should kept to minimal iterations and any inner loops should be iterated a “typical” number of times.
• Continue moving up the level of nested loops until all loops are tested.

3.2 Writing White Box Test Cases
For white box test cases, we can automate our test cases by creating a test class. The test class is another program that needs to be in the same directory as the program under test. To show the link between the test class and the code under test, the test class should be named \(<ProgramUnderTest>Test.java\). Therefore, the test class for a program called StringAnalyzer would be StringAnalyzerTest.

As a test program, your test class will have a main method. Running the test class will also run the program under test because the test class will call the methods in the program under test. If our program under test contains static methods, we can call methods in the program under test similarly to how we call methods of the Math class: \(<ProgramUnderTest>.[methodName](<parameters>)\). If our program under test contains instance methods, we will need to create an instance of the class under test before calling the methods.

When creating a test program, the test cases can be broken out into methods. You can have one or more test method for each method under test. A good naming convention is to call your test method test<MethodName><DescriptionOfTest>. The test method name is your test case’s unique id.

Figure 9 contains a test case for path 1 – 2 – 8 – 9 through the countDigits method. White box test cases require additional code so that all the required elements of the test case are given. The required elements are: unique id, specific and repeatable inputs, specific and repeatable expected results, and actual results. The unique id can be given as the test method name. The inputs to the test case are the parameters to the method. The actual results are the return value from the method call. You will need to identify the expected results for a comparison.
/**
 * Program to test StringAnalyzer
 * @author Dr. Sarah Heckman
 */
public class StringAnalyzer {

/**
 * Checks path 1 - 2 - 8 - 9 through the countDigits
 * method.
 */
public static void testCountDigitsInvalidBounds() {
   System.out.println("testCountDigitsInvalidBounds()");
   int expectedCount = -1;
   int actualCount = StringAnalyzer.countDigits("456", 'a', '9');
   System.out.printf("Expected: %d   Actual: %d\n",
                     expectedCount, actualCount);
}

/**
 * Starts the test program
 * @param args an array of command line arguments
 */
public static void main(String [] args) {
   testCountDigitsInvalidBounds();
}
}

Figure 9: Example test code for countDigits in StringAnalyzer

4. Summary
Testing is an important part of the process of developing software, and helps increase our confidence that our software meets the requirements set out by the customer (or your instructor). Systematic application of the strategies presented in this document will help you create a suite of test cases that will efficiently test your code. Black box testing is used to test the program as a whole by specifying program inputs and checking the generated outputs with the expected values. White box testing uses the source code to guide the development of test cases. Writing test cases for each method is a specific type of white box testing called unit testing. White box test cases can be written in another Java program, which allows for the automation of your testing efforts. Overall, testing is a technique that identifies underlying program faults by exercising the program you’ve developed.

5. References