# CS225L Lab 6: Testing

# Learning Outcomes

* Why we need to test our software
* How to select test cases based on the requirements
* What is black box test and white box test

# Pre-lab: Intro to Testing

This lab will introduce software testing and the importance of testing throughout development.

**Overview**

The objective of software testing is executing the program being tested in such a way that any errors that may exist in the program will be revealed. Software testing is an important phase of software development that is often overlooked or minimized when a student is first learning computer programming. A great amount of effort is invested in determining algorithms, devising program logic structures and writing the source code in an unfamiliar language.

Once all that is accomplished, it is sometimes easy to delude oneself into the belief that the program works correctly once it compiles without errors and execution causes no run-time exceptions. Unfortunately, that is far from reality.

The algorithms and logic that a programmer designs into a program may in fact not be correct. A minor error in coding the program can cause erroneous results. It is an inescapable fact that software development is not complete without proper testing to validate that the application performs according to its specifications.

How software testing is performed is critical to the successful verification that a program is functioning as intended. A number of techniques have been developed for software testing. Fully adequate testing of a rather complex software system would probably involve the use of several of these techniques.

In addition, a number of software and hardware tools exist which can assist with the testing of large systems. In this lab, you will explore one of the most basic of software testing techniques, known as black box testing.

**Black Box Testing**

Black box testing is focused on the required functionality of a program. The actual internal construction and operation of the program is not given any consideration. It should be possible to trace all the required functions of the program back to the original requirements or problem statement that served as the foundation for the development of the program.

Another way to view this is to focus on the required inputs to the program and the corresponding required output. These inputs and outputs dene the interface between the program and the environment outside the program.

The relationship between the inputs and outputs dene the functionality. This means that all that is needed for a software engineer to design a black box test is a complete definition of the interface and functions.

The need to test a program's interface and functions, without knowledge of its internal workings, give rise to the term “black box” for this type of testing. The person designing and executing the test cannot “see” into the program (and doesn't want to), so it may as well be a black box in which the internals remain a mystery. This is contrasted with “white box” testing in which a detailed knowledge of how a program is written is necessary to design and perform tests.

One of the advantages of black box testing is that it is perfectly feasible for the testing to be performed by someone who did not develop the program. In fact, this situation is usually desirable so that the tests are performed without any biased views or misunderstandings that the program's developer may have. Persons who test their own programs have to exert special effort to approach the tests as an honest effort to “break” their program, which may feel somewhat like damaging your own creation.

**Program Interface & Functionality**

The program interface discussed above consists of:

* The number of required inputs.
* The data types of those inputs.
* The number of the outputs.
* The data types of those outputs.

The functionality of a program consists of complete definition of how any valid set of inputs is translated to a set of outputs. The most difficult item to deal with is the functionality of the program. It is absolutely essential that the person designing the tests obtain a full understanding of the desired functions of the program by carefully reviewing the program specifications. This can be more difficult for the person testing his or her own programs because testing might proceed with the same misunderstandings that may be used to develop the program.

Keep in mind that the inputs and outputs of a program can include values passed through parameters, data placed into files, interaction with the operating system and interaction with other programs.

**Test Cases**

It is usually necessary to develop several test cases or sets of inputs that are designed to fully test a program. The design of test cases for black box testing is vital to the overall success of the testing efforts. A number of considerations need to be made when creating test cases. We will discuss two of these considerations; equivalence partitioning and boundary value analysis.

Equivalence partitioning divides all possible combinations of input data into categories of test cases that are designed to uncover classes of errors, such as incorrect processing of upper case characters. Some guidelines for test cases include:

* If the input has a valid range, test one valid input and two invalid input (above and below the valid range).
* If input has a specified characteristic, test input with that characteristic and two others that are outside the characteristic on both sides. For example, if input must have 5 characters, you might test 5 characters, 3 characters and 8 characters.
* If input may or may not be present, test each situation.
* If input must be a member of a set of values, test one value in the set and one value outside the set.

Boundary value analysis addresses a phenomenon observed with software errors, which is that a higher proportion of errors occur with values at the boundaries of valid input values than with values well within these boundaries. This is somewhat intuitive, since many times additional logic must be included in a program to deal with boundary conditions. Guidelines for testing at boundaries include:

* If input has a valid range, test values at the boundary and also immediately adjacent to the boundary, both inside and outside (valid and invalid).
* If input determines a number of values or repetitions, test cases should test the minimum and maximum allowed values.
* The above two guidelines also apply to output that must observe boundaries. Test cases should be designed to place the output at boundaries and both above and below those boundaries.

**An Example**

A subprogram is required to accept one input value, perform a simple calculation and provide the resulting answer. The interface has been specified as follows:

FUNCTION Process\_Input (X : Integer ) RETURNS Integer;

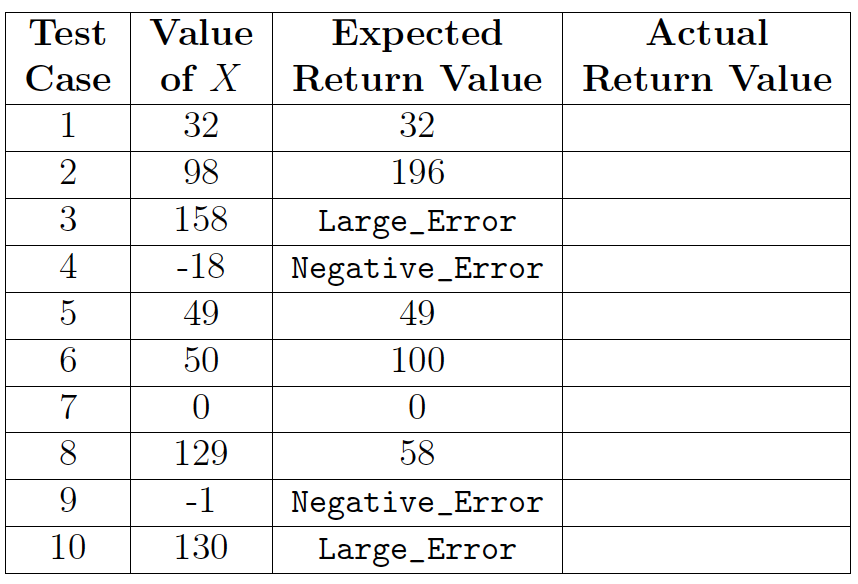
The functionality has been specified as follows, where is the return value:

* for
* for
* Large\_Error exception occurs if
* Negative\_Error exception occurs if

To exhaustively test this function would required in excess of 130 test cases. However, adequate testing can be accomplished with much less effort by defining fewer test cases based on the discussion above, as follows:

* One valid value of in the range
* One valid value of in the range
* One invalid value of above the boundary at 129
* One invalid value of below the boundary at 0
* Two valid values of at the boundary dened at ( and )
* Two valid values at the extreme boundaries of ( and )
* Two invalid values at the extreme boundaries of ( and )

The ten test cases should be listed in a table (shown below) showing the exact value of each input, the expected value of each output, and the actual value of each output which is recorded during the test. It would be best to develop the test cases before the function was actually developed, especially if the same person who programmed the function was also performing the testing.



**Lab Activities**

In this lab, you will test a piece of software that has the following requirements:

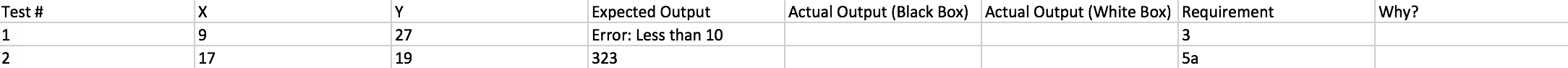
1. The program shall accept no less than two integer inputs (henceforth and , respectively) via the command line.
2. If the sum of and is not 36, an appropriate error message shall be displayed.
3. If either or is less than 10, an appropriate error message shall be displayed.
4. If and are the same value, an appropriate error message shall be displayed.
5. If there are no errors with and , the program shall display an output (henceforth ) that is determined as follows:
   1. for
   2. for

**Task 1: Test Cases**

1. Design a minimum of ten test cases to test this software. **(5 Points)**

2. Explain why you chose those values for your test cases. **(5 Points)**

You must provide all of your answers in a single .csv (comma separated values) file, following a template posted on Canvas. Remember that Excel (and its open-source counterparts) can edit these. Below are two examples that you may use to get started. Note that the Actual Outputs (both black box and white box) columns are blank; we will fill them in during the testing.



**Task 2: Black Box Testing**

During this Lab Eclipse WILL NOT be used. You will be executing from the command prompt instead. We will first test the program Test.class using black box testing methods.

1. Download the attached Test.class file and save it somewhere memorable.
2. Open Command Prompt and navigate to the directory where you stored the file. For help with this step, refer to Lab 1 or ask your lab instructor.
3. Run the program Test with command line arguments representing and to verify that the program executes. (The appropriate command would be java Test 12 24).
4. Now, run the program Test using your test cases. Record the actual black box testing results of each test.
5. Compare your actual results with your expected results. Based on your results, are there any requirements that failed to be met?
6. Please stop at this point in the lab and show a lab instructor the results of your testing. **(5 Points)**

**Task 2: White Box Testing**

The developer of the code has now asked you to fix the defects so that the program meets all of the requirements. Unfortunately, the developer has not released the source code. To that end, you had a friend decompile the provided .class to convert it from Java bytecode back into this .java source code file.

*Note: The decompiled code contains no comments. When Java programs are compiled, the comments are ignored because they are simply for human readability and serve no purpose to the JVM.*

1. Download the attached Test.java source code and save it somewhere memorable.
2. Open the source code using a plain text editor, such as Notepad, Notepad++ (Win), TextEdit (Mac), Atom, Sublime Text (any), etc.

*Note that the school computers use Code Blocks to open .java files by default you probably don't want to use that. The school computers do have Notepad++ installed, which is easily used if you right-click on the file.*

1. Find the defects in the source code and fix them.
2. Save the code, then recompile it on the command line using javac.

*On a school machine, you may have to set PATH again for this to work. Go back to Lab 1 on Canvas and copy and paste the command there.*

1. To contrast with black box testing (where the source code is hidden from the tester), we will now perform white box testing (where the source code is visible to the tester). Re-test the program using your test cases from earlier to make sure that your fixes worked. We are also re-doing these tests to ensure that no new errors have been introduced. The type of testing is known as regression testing. Record your test results in the Actual Outputs (White Box) column.
2. Please stop at this point and show a lab instructor your modified code and updated test results. What defects did you fix? Did your fixing result in any new errors? **(5 Points)**

**Post Lab Deliverables**

* Your modified Test.java file **(5 Points)**
* Your CSV file containing your 10 test cases, including actual outputs from both black box and white box testing AND answers to the following questions: **(5 Points)**

1. What is black box testing?
2. What is white box testing?
3. Why are boundary values particularly important to consider when developing test cases?
4. Why is it important to know which requirement is being tested by a particular test case?
5. Provide feedback:
   1. Which lab section did you attend? (Section # or day of week)
   2. What did you like about the lab? What did you dislike?
   3. What would you change about this lab to improve it?

To be clear, you should submit exactly two files on Canvas: The .java file with the corrected code, and a .csv file containing all of your tests, test results, and answers to only the questions above.

Total Points from Lab 6: **30 Points**