# CS225L Lab 2: Testing

# Learning Outcomes

At the end of this lab the student will be able to …

* Explain why we need to test our software
* Select test cases based on the requirements
* Explain the difference between black box testing and white box testing

# Pre-lab: Intro to Testing

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

**Overview**

The objective of software testing to execute 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. Introductory software courses tend to focus on 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 that the program works correctly once it compiles without errors and execution causes no run-time exceptions under normal conditions. Unfortunately, that is far from reality.

The algorithms and logic that a programmer designs into a program may in fact not be correct. Even with correct logic, 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. This lab introduces the student to black box testing, where the person doing the testing does not have access to the source code, and white box testing, where the person doing the testing does have access to the source code.

**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.

Software functionality is defined in terms of the relationship between inputs and outputs. 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**

A test case is a specific input paired with an expected response from the software. It is necessary to develop several test cases or sets of inputs in order to fully test a program. Each test case must contain a specific, concrete value for input, and a specific, concrete value for the expected program response. For example “Input: a five digit, positive integer” is not a test case because it does not list a specific integer value. A valid test case would be “Input: 25675.”

Students new to programming tend to think of testing as something done by a human providing input to the program. This is how you will conduct the tests in this lab. However, testing tend to be repetitive and tedious. Throughout the course you will be encouraged to write methods that perform tests and observe (or print out) the program’s response automatically.

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: Equivalence partitioning divides all possible combinations of input data into categories of test cases where one test case serves as a representative for all inputs in the partition. Since all inputs in the category are expected to result in the same program behavior, only one specific input needs to be tested. Each partition is designed to uncover a single class of error, 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: Boundary value analysis addresses a phenomenon observed with software errors, which is that a higher proportion of errors occur at the boundaries of valid input values than with values within these boundaries. 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;

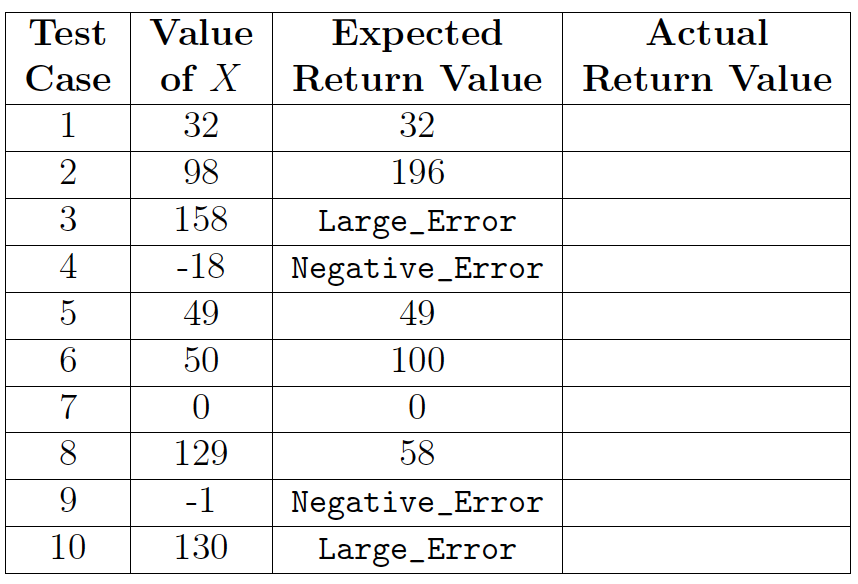
The functionality has been specified as follows, where is the return value:

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

To exhaustively test this function would require 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.



**In-Lab Activities**

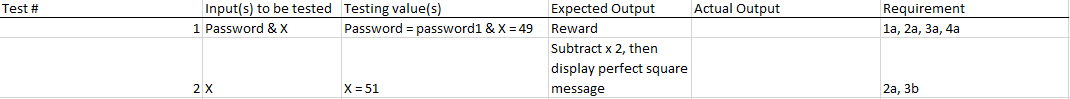
In this lab, you will be writing a series of test cases based on given requirements, then testing a pre-existing black box program to see if it meets the requirements. You will then write your own code for a set of requirements, and test it to make sure it meets the requirements. Once you are finished, make sure to show your test cases and code to an instructor. The requirements for the first 2 sections are below:

1. The user will be entering a password to unlock the entire functionality of the program. The correct password is “password1” (don’t judge)
   1. If the user enters the correct password, an appropriate message shall be displayed.
   2. If the user enters just “password”, a message that tells them it is too obvious of a choice, and incorrect, shall be displayed.
   3. If the user enters any other password, a message telling them it is incorrect shall be displayed.
2. The user will then be asked to enter a whole number (refer to as X for simplicity), without knowing what it is for.
   1. The number shall be within the range 3 < X ≤ 100.
   2. If X is not in the appropriate range, an appropriate message shall be displayed. Use separate messages for too large or too small.
3. After the user enters a number, and it is within the valid range, the program will check to see if it is a perfect square.
   1. If X is a perfect square, the program shall display a message containing the square root of X.
   2. If X is not a perfect square, the program shall subtract 1 from X and try again. It will display a message each time it has to subtract 1.
   3. The program shall keep track of how many times it had to subtract 1 from the user’s number.
4. Finally, the user will get a reward if they satisfied the program’s needs.
   1. If the user got both the password correct and took 3 or fewer ‘subtractions’ to get a perfect square, the code shall display a reward (this can be some ascii art, a motivational speech, whatever you feel is appropriate.)
   2. If the user took more than 3 ‘subtractions’ to get a perfect square, a message shall be displayed telling them.
   3. If the user entered the wrong password, there shall be a message telling them.

**Task 1: Test Cases**

1. Design a minimum of ten test cases to test this software (there are more than 10). Remember that a test case must have specific, concrete values for input and output.

2. Explain why you chose those values for your test cases.

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 Output column is blank; we will fill them in during the testing.

**Task 2: Black Box Testing**

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

1. Download the attached loops.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 (The appropriate command would be ‘java loops’).
4. Now, run the program Test using your test cases. Record the actual black box testing results of each test.

Compare your actual results with your expected results. Based on your results, are there any requirements that failed to be met?

**Task 3: White Box Testing**

For the white box section of the lab, you will be writing and testing code that determines which of two rectangles has greater area. . You can use Eclipse for this part of the lab if you wish.

The code will request the user to input four numbers. For purpose of this discussion, we will refer to these numbers as X1, Y1, X2, and Y2 (Note: These are not good variable names, and you are encouraged to create better ones in the code).

The area of Rectangle1, A1 = X1 \* Y1, and the area of Rectangle2, A2 = X2 \* Y2.

The program requirements are as follows:

1. The program shall accept four numbers through console input.
2. If any of the numbers are less than or equal to 0, the code shall immediately terminate,
3. If any of the numbers are less than or equal to 0, the program shall tell the user that their numbers must be non-zero, positive numbers.
4. The program shall compute A1.
5. The program shall compute A2.
6. The program shall display the areas A1 and A2.
7. If the areas of the rectangles are not equal, the program shall display which rectangle had the greater area.
8. If the areas of the rectangles are equal, the program shall indicate that the rectangles are of equal area.

Once you have finished writing your code, write 5 test cases, just like for the black box section, and use your code to test them. Fix any errors you find.