## **Week 2 – Test Plan Assignment**

## Learning Objective

* Use Specification-Based testing techniques to identify and fix errors in Java programs.

## Instructions:

A key goal of software testing is to detect and resolve errors in the code. While numerous software testing techniques exist, CMSC 115 introduces two popular approaches:

* Specification-Based Testing (opaque-box) assesses a program's functionality without examining its internal structure. Tests are created based on the program's requirements and the characteristics of the input domain.
* Structure-Based Testing (clear-box) analyzes a program's internal structures and logic to evaluate how thoroughly the tests exercise the program.

Specification-based testing techniques are used for the Week#2 programming projects.

**Download and edit this document to show the test results for the following tasks:**

* Testing Task #1: InchesToFeet.java.
* Testing Task #2: YearsToMinutes.java.
* Testing Task #3: Average.java.
* Testing Task #4: TransmissionRate.java.
* Testing Task #5: Chapter 2 Project 1.
* Testing Task #6: Chapter 2 Project 5.

When editing the test cases:

* Fill in the expected I/O (input/output) **prior** to running the program.   
  Fill in the actual I/O and test status **after** running the program.
* Display input values in bold. It’s ok to wrap a line of output across multiple lines if the table cell is not wide enough to display it on one line. Assume input is entered on the same line as the prompt, even if it is displayed on a separate line in the test case.
* Refer to the Revel sample runs for the exact I/O for the Chapter 2 programming projects.

*Note: This submission is only for the test plans. The Java code for Project 1 and 5 should be submitted in the Revel environment for grading.* *The Revel autograder expects a class named* ***Exercise****. If you use an IDE and create a class with a different name, remember to change the name to* ***Exercise*** *when submitting to the Revel autograder.*

### Testing Task #1 – InchesToFeet.java

Consider the following program requirements:

*Write a program that converts inches to feet. Read in an integer representing the number of inches, then calculate and display the equivalent number of feet. There are 12 inches in one foot.*

Each row in the table below contains a test case for a specific input value.

* The **Expected I/O** (Input/Output) column can be filled out based on the specified program requirements. Input values are displayed in bold.
* The **Actual I/O** and **Status** columns are filled out after writing and executing the program. Status will contain *Pass/Fail* based on whether the expected and actual I/O columns match.

|  |  |  |  |
| --- | --- | --- | --- |
| **Table 1: InchesToFeet Test Cases** | | | |
|  | Expected I/O | Actual I/O | Status |
| 1 | Enter inches: **3**  3 inches = 0.25 feet |  |  |
| 2 | Enter inches: **12**  12 inches = 1.0 feet |  |  |
| 3 | Enter inches: **18**  18 inches = 1.5 feet |  |  |
| 4 | Enter inches: **24**  24inches = 2.0 feet |  |  |

The **InchesToFeet** class implements the requirements. However, there is an error due to integer division. You can execute the program online at <https://onlinegdb.com/HzV46gD1A>. The **Run** button starts the program and prompts for the inches in the console.

* Run the program using the input specified in each test case in Table 1.
* Fill in the **Actual I/O** and **Status** (Pass/Fail) columns based on the program output.
* Confirm that tests #2 and #4 pass, while tests #1 and #3 both fail.

A failed test indicates an error exists in the code.

Recall how division works in Java. If both operands are integers, the result is an integer and any remainder is discarded. However, if either operand is a double, the result is a double.

|  |  |
| --- | --- |
| **Expression** | **Value** |
| 3/12 | 0 |
| 3/12.0 | 0.25 |
| 18/12 | 1 |
| 18/12.0 | 1.5 |

|  |  |  |  |
| --- | --- | --- | --- |
| **Table 2: InchesToFeet Test Cases** | | | |
|  | Expected I/O | Actual I/O | Status |
| 1 | Enter inches: **3**  3 inches = 0.25 feet |  |  |
| 2 | Enter inches: **12**  12 inches = 1.0 feet |  |  |
| 3 | Enter inches: **18**  18 inches = 1.5 feet |  |  |
| 4 | Enter inches: **24**  24inches = 2.0 feet |  |  |

* Edit **InchesToFeet** to fix the integer division error. The **Fork This** button creates an editable copy.
* Rerun the program for each test in Table 2 and record the results. Verify that all tests passed.
* Insert a screen print of your code solution that shows the console result from test #3 (18 inches).

### Testing Task #2 – YearsToMinutes.java

Consider the following program requirements:

*Write a program that reads in an integer representing the number of years and prints the equivalent number of minutes. Assume a year has 365 days, a day has 24 hours, and an hour has 60 minutes.*

The requirements do not specify a particular range of values for years, other than stating the input is an integer. We'll use the 1, 2, 1000, and 5000 as input values.

|  |  |  |  |
| --- | --- | --- | --- |
| **Table 3: YearsToMinutes Test Cases** | | | |
|  | Expected I/O | Actual I/O | Status |
| 1 | Enter years: **1**  525600 minutes |  |  |
| 2 | Enter years: **2**  1051200 minutes |  |  |
| 3 | Enter years: **1000**  525600000 minutes |  |  |
| 4 | Enter years: **5000**  2628000000 minutes |  |  |

The **YearsToMinutes** class implements the requirements. You can run the program online at <https://onlinegdb.com/MCCMmGTIZ>.

* Run the program for each test case in Table 3 and record the results.
* Confirm tests 1-3 pass, while test #4 fails.

An **overflow** occurs when a calculation produces a result that is too large for the declared data type.

* An int can store whole numbers ranging from: -2,147,483,648 to 2,147,483,647.   
  Java provides constants **Integer.MIN\_VALUE** and **Integer.MAX\_VALUE** for these values.
* The last test case results in an overflow error because the variable **minutes** is declared as an int and can't store a value as large as 2,626,000,000.

A long can store a value as large as 9,223,372,036,854,775,807.

|  |  |  |  |
| --- | --- | --- | --- |
| **Table 4: YearsToMinutes Test Cases** | | | |
|  | Expected I/O | Actual I/O | Status |
| 1 | Enter years: **1**  525600 minutes |  |  |
| 2 | Enter years: **2**  1051200 minutes |  |  |
| 3 | Enter years: **1000**  525600000 minutes |  |  |
| 4 | Enter years: **5000**  2628000000 minutes |  |  |

* Fork the code to create an editable copy. Update the code to declare variable **minutes** as long.
* Rerun the program for each test case in Table 4 and record the results.

**long minutes = years \* 365 \* 24 \* 60;**

Unfortunately, test#4 still fails to produce the expected output.

Even though the variable **minutes** is declared as long, the expression **years \* 365 \* 24 \* 60** produces an int because the variable **years** and numeric literals are all ints. The solution is to either cast **years** as a long or to specify the first numeric literal value as a long by appending **L**.

|  |  |  |
| --- | --- | --- |
| **Expression** | **Type** | **Comment** |
| years \* 365 | int |  |
| (long) years \* 365 | long | Doesn’t modify the actual variable type. Creates a copy of the value stored in memory as a long. |
| years \* 365L | long |  |

|  |  |  |  |
| --- | --- | --- | --- |
| **Table 5: YearsToMinutes Test Cases** | | | |
|  | Expected I/O | Actual I/O | Status |
| 1 | Enter years: **1**  525600 minutes |  |  |
| 2 | Enter years: **2**  1051200 minutes |  |  |
| 3 | Enter years: **1000**  525600000 minutes |  |  |
| 4 | Enter years: **5000**  2628000000 minutes |  |  |

* Fix the overflow error by casting the variable **years** as long in the multiplication expression.
* Rerun the program for each test in Table 5 and record the results. Verify that all tests passed.
* Insert a screen print of your code solution that shows the console result of test #4 (5000 years).

NOTE: The data type **long** is a good choice when applications result in extremely large numbers. However, many programs in CMSC 115 involve much smaller numbers and thus **int** is adequate.

### Testing Task #3 – Average.java

Consider the following program requirements:

*Write a program that reads two integers and prints the average value as a double.*

The **Average** class implements the requirements. You can run the program online at <https://onlinegdb.com/62RWeV1M3>.

|  |  |  |  |
| --- | --- | --- | --- |
| **Table 6: Average Test Cases** | | | |
|  | Expected I/O | Actual I/O | Status |
| 1 | Enter 2 numbers: **10 24**  Average = 17.0 |  |  |
| 2 | Enter 2 numbers: **13 42**  Average = 27.5 |  |  |

* Run the program for each test case in Table 6 and record the results.
* Confirm that both tests failed.

The code contains a calculation error due to operator precedence. Java follows the standard mathematical order of operations (PEMDAS):

1. Parentheses: ()
2. Exponents: represented by the Math.pow() method
3. Multiplication, Modulo, Division: Multiplication (\*), Modulo (%) and Division (/) have the same precedence and are evaluated left to right.
4. Addition and Subtraction: Addition (+) and subtraction (-) have the same precedence and are evaluated from left to right.

|  |  |  |  |
| --- | --- | --- | --- |
| **Table 7: Average Test Cases** | | | |
|  | Expected I/O | Actual I/O | Status |
| 1 | Enter 2 numbers: **10 24**  Average = 17.0 |  |  |
| 2 | Enter 2 numbers: **13 42**  Average = 27.5 |  |  |

* Fix the error by using parentheses to override the default operator precedence.
* Rerun the program for each test in Table 7 and record the results. Verify that all tests passed.
* Insert a screen print of your code solution that includes the console result from test #2.

### Testing Task #4 – TransmissionRate.java

R0 (R-naught) is a term to describe the reproduction rate of infectious pathogens. A disease with R0 of 5 indicates a person will transmit to an average of 5 other people. Three iterations of transmission results in 53 = 125 infections.

Consider the following program requirements:

*Write a program that reads in a value for R0 and calculates the number of infections after 3 iterations, i.e. (R0)3.*

The **TransmissionRate** class implements the requirements. You can run the program online at <https://onlinegdb.com/0G8CvlcRL>.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Table 8: TransmissionRate Test Cases** | | | | |
|  | Expected I/O | Actual I/O | Status | Comment |
| 1 | Enter R0: **2**  3 iterations result in 8.0 infections |  |  | 23 = 8 |
| 2 | Enter R0: **3**  3 iterations result in 27.0 infections |  |  | 33 = 9 |
| 3 | Enter R0: **5**  3 iterations result in 125.0 infections |  |  | 53 = 125 |
| 4 | Enter R0: **1.5**  3 iterations result in 3.375 infections |  |  | 1.53 = 3.375 |

* Run the program for each test case in Table 8 and record the results.
* Confirm test#2 passes, while the other tests fail.

There is an error in the call to **Math.pow**. Read the documentation to understand the parameters and return value <https://docs.oracle.com/javase/8/docs/api/java/lang/Math.html#pow-double-double->

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Table 9: TransmissionRate Test Cases** | | | | |
|  | Expected I/O | Actual I/O | Status | Comment |
| 1 | Enter R0: **2**  3 iterations result in 8.0 infections |  |  | 23 = 8 |
| 2 | Enter R0: **3**  3 iterations result in 27.0 infections |  |  | 33 = 9 |
| 3 | Enter R0: **5**  3 iterations result in 125.0 infections |  |  | 53 = 125 |
| 4 | Enter R0: **1.5**  3 iterations result in 3.375 infections |  |  | 1.53 = 3.375 |

* Fix the error in the call to **Math.pow**.
* Run the program for each test case in Table 9 and record the results. Verify that all tests passed.
* Insert a screen print of your code solution that includes the console result from test #3.

### Testing Task #5: Chapter 2 Project 1 Test Plan

NOTE: Submit your Chapter 2 Programming Project 1 Java code to Revel for auto-grading.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Table 10: Project 1 Test Cases** | | | | |
|  | Expected I/O | Actual I/O | Status | Comment |
| 1 | Enter the subtotal and a gratuity rate: **10 15** The gratuity is $1.5 and total is $11.5 |  |  | subtotal $10 rate 15% |
| 2 | Enter the subtotal and a gratuity rate:  **12.8 10.5**  The gratuity is $1.344 and total is $14.144 |  |  | subtotal $12.8 rate 10.5% |
| 3 |  |  |  | subtotal $0  rate 20% |
| 4 |  |  |  | subtotal $27.50  rate 0% |
| 5 |  |  |  |  |
| 6 |  |  |  |  |

* Fill in the **Expected I/O** column for test cases 3 and 4 based on the input values specified in the **Comment** column. Display input values in **bold**. It’s ok to wrap a line of output across multiple lines if the table cell is not wide enough to display it on one line.
* Define two additional test cases 5 and 6. Describe your choice of input in the **Comment** column.
* Run your Chapter 2 Project 1 solution for each test case in Table 10 and record the results.
* Try to fix any errors identified by the tests. If you are unable to get a test case to pass, mention it in the lessons learned.
* Describe lessons learned while implementing Chapter 2 Project 1.

LESSONS LEARNED:

### Testing Task #6: Chapter 2 Project 5 Test Plan

NOTE: Submit your Chapter 2 Programming Project 5 Java code to Revel for auto-grading.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Table 11: Project 5 Test Cases** | | | | |
|  | Expected I/O | Actual I/O | Status | Comment |
| 1 | Enter investment amount: **1000.56** Enter annual interest rate in percentage: **4.25** Enter number of years: **1** Accumulated value is $1043.92 |  |  | Sample run.  $1000.56 investment, 4.25%, 1 year |
| 2 | Enter investment amount: **1043.92**  Enter annual interest rate in percentage: **4.25**  Enter number of years: **1**  Accumulated value is $1089.16 |  |  | $1043.92 investment, 4.25%, 1 year |
| 3 |  |  |  | $1000.56 investment, 4.25%, 2 years |
| 4 | Enter investment amount: **2000**  Enter annual interest rate in percentage: **5**  Enter number of years: **10**  Accumulated value is $3294.01 |  |  | $2000 investment, 5%, 10 years |
| 5 |  |  |  | 0 years |
| 6 |  |  |  | 0 rate |
| 7 |  |  |  | 0 investment |

* Update Table 11 to fill in the **Expected I/O** for test case #3 based on the values of tests #1 and #2.
* Fill in the **Expected I/O** for test cases 5 – 7, using a positive value for any variable not assigned to 0 in the **Comment** column.
* Run your Chapter 2 Project 5 solution for each test case in Table 11 and record the results.
* Try to fix errors identified by the tests. If you are unable to get a test case to pass, mention it in the lessons learned.
* Describe lessons learned while implementing Chapter 2 Project 5.

LESSONS LEARNED: