## **Week 3 – Test Plan Assignment**

## Learning Objectives

* Use **Equivalence Class Partitioning** **(ECP)** to identify errors within equivalence classes.
* Use **Boundary Value Analysis (BVA)** to identify errors at the boundary of an equivalence class.
* Use **Decision Table Testing (DTT)** to identify errors when different combinations of input conditions lead to distinct program outcomes.

## Instructions:

It is not practical to test a program with every possible input value, so a selection of representative values must be chosen. This assignment introduces three widely used specification-based testing techniques aimed at minimizing the overall number of test cases to a manageable set, while still ensuring coverage of the functional requirements:

* Equivalence Class Partitioning (ECP)
* Boundary Value Analysis (BVA)
* Decision Table Testing (DTT)

Download and edit this document to fill in your test plans for the following tasks:

* Testing Task #1: ValidGPA.java
* Testing Task #2: LetterGrade.java
* Testing Task #3: IndoorComfort.java
* Testing Task #4: PizzaPrice.java
* Testing Task #5: Chapter 3 Programming Project 3.
* Testing Task #6: Chapter 3 Programming 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 3 programming projects.

*Note: This submission is only for the test plans. The Java code for Chapter 3 Programming Projects should be submitted in the Revel environment for grading.*

### Testing Task #1 – ValidGPA.java

**Equivalence Class Partitioning (ECP)** divides the input domain into equivalence classes (also called partitions) based on the similarity of input values. Each value in an equivalence class should display similar program output as all other values in that class.

* If one value in an equivalence class passes a test, all values in the class are expected to pass.
* If one value in an equivalence class fails a test, all values in the class should likewise fail.

Test cases are written to ensure each equivalence class is covered by at least one test.

Consider the following program requirements:

*Write a program that reads a number as a double and prints whether it is a valid GPA. A valid GPA falls within the inclusive range of values 0.0 and 4.0.*

The inclusive range 0.0 - 4.0, also denoted as [0.0,4.0], has a minimum value of 0.0 and a maximum of 4.0. The two boundaries result in three equivalence classes (dash indicates a range, not a minus sign):

|  |  |  |
| --- | --- | --- |
| **Invalid** | **Valid GPA** | **Invalid** |
| < 0.0 | 0.0 - 4.0 | > 4.0 |

Each equivalence class should be covered by at least one test case. A random value is selected from each equivalence class. For example:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Expected I/O | Actual I/O | Status (Pass/Fail) | Equivalence Class |
| 1 | GPA: **-1.0**  -1.0 is invalid |  |  | < 0.0 |
| 1 | GPA: **2.5**  2.5 is valid |  |  | 0.0 - 4.0 |
| 3 | GPA: **5.7**  5.7 is invalid |  |  | > 0.0 |

The **ValidGPA** class implements the requirements, but there is an error. You can run the program online at <https://onlinegdb.com/iuxE9oeQC>.

* Run the **ValidGPA** class for each test case, filling in the Actual I/O and Status columns.
* Confirm that test #1 fails, while tests #2 and #3 pass.

Recall that logical operators && and || can be used to check compound Boolean expressions:

|  |  |  |  |
| --- | --- | --- | --- |
| **a** | **b** | **a || b**  **(OR)** | **a && b**  **(AND)** |
| true | true | true | true |
| true | false | true | false |
| false | true | true | false |
| false | false | false | false |

The **ValidGPA** class compares the input value against the range maximum of 4.0, but it does not compare it against the range minimum of 0.0.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Expected I/O | Actual I/O | Status | Equivalence Class |
| 1 | GPA: **-1.0**  -1.0 is invalid |  |  | < 0.0 |
| 1 | GPA: **2.5**  2.5 is valid |  |  | 0.0 - 4.0 |
| 3 | GPA: **5.7**  5.7 is invalid |  |  | > 0.0 |

* Edit **ValidGPA** to ensure the input value falls within the range [0.0, 4.0]. HINT: Use an appropriate logical operator to compare the input value against both the range minimum and maximum.
* Rerun every test case and fill in the Actual I/O and Status columns. Confirm each test passes.
* Insert a screen print of your code solution that shows the console result from test #1.

### Testing Task #2 – LetterGrade.java

Consider the following program requirements:

*Write a program that reads a numeric score between 0 and 100 and prints the corresponding letter grade based on the following mapping between letter grades and numeric ranges:* F = 0-59, D = 60-69, C = 70-79, B = 80-89, A = 90-100.

0 – 100 is specified as the valid range of scores, thus there are seven equivalence classes:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Invalid | F | D | C | B | A | Invalid |
| < 0 | 0 – 59 | 60 – 69 | 70 – 79 | 80 – 89 | 90 – 100 | > 100 |

The **LetterGrade** class implements the requirements, but there is an error. You can run the program online at <https://onlinegdb.com/_UNcmXkUA>.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Expected I/O | Actual I/O | Status | Equivalence Class |
| 1 | Score: **-10**  Invalid |  |  | < 0 |
| 2 | Score: **52**  F |  |  | 0 - 59 |
| 3 |  |  |  | 60 - 69 |
| 4 |  |  |  | 70 - 79 |
| 5 |  |  |  | 80 - 89 |
| 6 |  |  |  | 90 - 100 |
| 7 |  |  |  | > 100 |

* Fill in the **Expected I/O** for test cases 3-7 by picking a random value from the specified equivalence class.
* Run the program for each test case and fill in the columns for **Actual I/O** and **Status** (Pass/Fail).
* Confirm that test #7 fails, while all other tests pass.

The **LetterGrade** class checks for the invalid equivalence class of negative values, but it does not check for the other invalid equivalence class of values that exceed 100.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Expected I/O | Actual I/O | Status | Equivalence Class |
| 1 | Score: **-10**  Invalid |  |  | < 0 |
| 2 | Score: **52**  F |  |  | 0 - 59 |
| 3 |  |  |  | 60 - 69 |
| 4 |  |  |  | 70 - 79 |
| 5 |  |  |  | 80 - 89 |
| 6 |  |  |  | 90 - 100 |
| 7 |  |  |  | > 100 |

* Edit **LetterGrade** to ensure the code correctly handles the seven equivalence classes.
* Rerun every test case and fill in the Actual I/O and Status columns. Confirm each test passes.
* Insert a screen print of your code solution that shows the console result from test #7.

### Testing Task #3 – IndoorComfort.java

**Boundary Value Analysis (BVA)** is a type of equivalence class partitioning that focuses on testing the values on or near the boundaries between equivalence classes, as this is where logic errors often occur.

|  |  |
| --- | --- |
|  | * Test cases include the boundary value **b** and values just above and below **b**. * An offset of **1** is used if the boundary is an integer  and **0.1** if the boundary is a double. |

Why pick values at boundaries? Consider the following program requirements:

*Write a program that reads an age and prints whether it meets the minimum of 18 for legal adulthood.*

The minimum boundary value 18 results in two equivalence classes for minors and adults.  **BVA** picks ages 17, 18, and 19 for testing. The following table shows correct and incorrect expressions that might be used to test whether an age meets the minimum value of 18. The set {17, 18, 19} is effective in identifying the expressions that place one or more values into the wrong equivalence class.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | **Expression** | **17** | **18** | **19** | **Age Placed in Correct Equivalence Class** |
| age > 17 | false | true | true | Correct |
| age >= 18 | false | true | true | Correct |
| age >= 17 | true | true | true | Incorrect |
| age > 18 | false | false | true | Incorrect |
| age == 18 | false | true | false | Incorrect |
| age < 18 | true | false | false | Incorrect |

A range **[min, max]** represents two boundaries **min** and **max** that result in three equivalence classes of valid/invalid or in-range/out-of-range values.

|  |  |
| --- | --- |
|  | * Test cases include 6 or 7 values:   + (2) The range min & max   + (2) Just above the min & max   + (2) Just below the min & max   + (1) a nominal value (optional). * The nominal value is usually computed as (max + min) / 2. |

Consider the following program requirements:

*Research suggests that most people feel comfortable indoors when the humidity level is between 40 and 60. Write a program that reads in a humidity value and prints whether it is comfortable or not.*

Equivalence class partitioning requires at least one value from each equivalence class, but the selected value can be arbitrary. In contrast, BVA requires specific values from each equivalence class as shown in the following table.

|  |  |  |
| --- | --- | --- |
| **Uncomfortable**  **min - 1** | **Comfortable Humidity**  **min, min+1, nominal, max-1, max** | **Uncomfortable**  **max + 1** |
| 39 | 40, 41, 50, 59, 60 | 61 |

The **IndoorComfort** class implements the requirements, but there is an error. You can run the program online at <https://onlinegdb.com/wV6W1H-1Aw>.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Expected I/O | Actual I/O | Status | Humidity BVA |
| 1 | Humidity: **39**  Uncomfortable |  |  | min – 1 |
| 2 | Humidity: **40**  Comfortable |  |  | min |
| 3 |  |  |  | min + 1 |
| 4 |  |  |  | nominal |
| 5 |  |  |  | max – 1 |
| 6 |  |  |  | max |
| 7 |  |  |  | max + 1 |

* Fill in the **Expected I/O** for test cases 3 - 7 using the value specified in the Humidity BVA column.
* Run the program for each test case and fill in the columns for **Actual I/O** and **Status** (Pass/Fail).
* Confirm that tests #2 and #6 fail, while all other tests pass.

The **IndoorComfort** class contains boundary errors for the range [40, 60]. The range should be inclusive, i.e. the minimum 40 and maximum 60 should both be deemed comfortable.

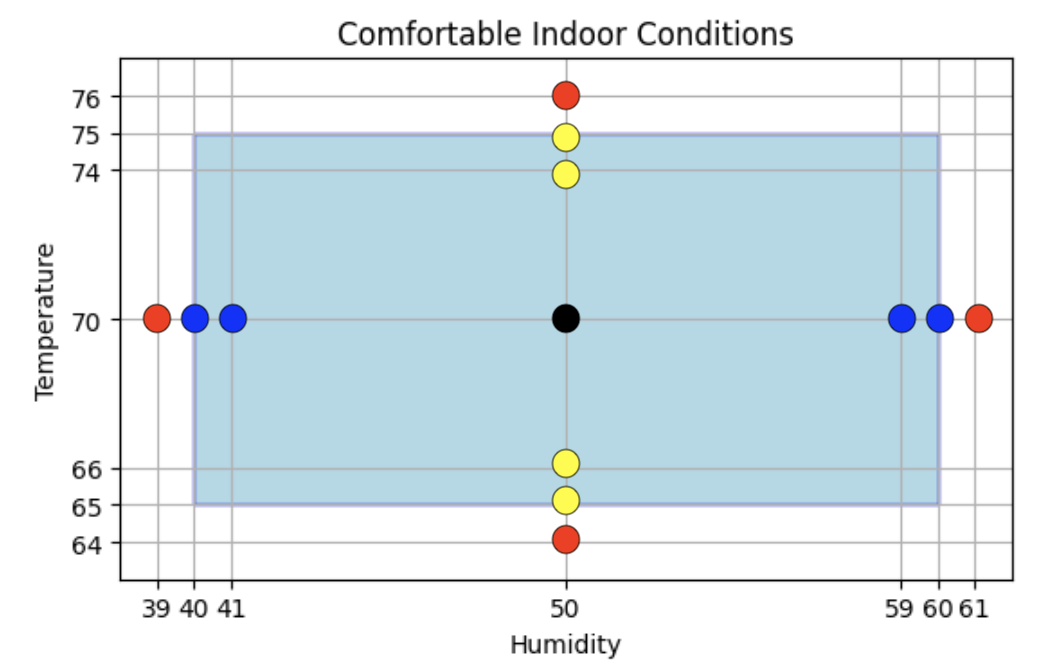
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Expected I/O | Actual I/O | Status | Humidity BVA |
| 1 | Humidity: **39**  Uncomfortable |  |  | min – 1 |
| 2 | Humidity: **40**  Comfortable |  |  | min |
| 3 |  |  |  | min + 1 |
| 4 |  |  |  | nominal |
| 5 |  |  |  | max – 1 |
| 6 |  |  |  | max |
| 7 |  |  |  | max + 1 |

* Edit **IndoorComfort** to fix the range boundary error.
* Rerun every test case and fill in the Actual I/O and Status columns. Confirm each test passes.

Now that you’ve got the program correctly identifying comfortable humidity values, update the program based on the following revised requirements:

*Research suggests that most people feel comfortable indoors when the* ***humidity level is between 40 and 60******and the temperature is between 65 and 75****. Write a program that reads in two values representing humidity and temperature and prints whether it is comfortable or not.*

Comfortable conditions depend on both humidity and temperature. We can plot this in 2D with humidity [40, 60] along the horizontal axis and temperature [65, 75] along the vertical axis. Any point that falls within the light blue rectangle represents a comfortable indoor condition.



We've seen how to use BVA for the range [40, 60] to select 7 **humidity** values:

|  |  |  |
| --- | --- | --- |
| **Uncomfortable Humidity**  **min - 1** | **Comfortable Humidity**  **min, min+1, nominal, max-1, max** | **Uncomfortable Humidity**  **max + 1** |
| 39 | 40, 41, 50, 59, 60 | 61 |

Now you will use BVA to select 7 **temperature** values.

|  |  |  |
| --- | --- | --- |
| **Uncomfortable Temperature**  **min - 1** | **Comfortable Temperature**  **min, min+1, nominal, max-1, max** | **Uncomfortable Temperature**  **max + 1** |
|  |  |  |

* Edit the table above to specify the appropriate temperature values for the range [65,75].

Is it necessary to test every combination of the 7 humidity and 7 temperature values, i.e. 49 test cases? Thankfully, the answer is no if we make a **single fault assumption**, which states that failures are rarely the result of the simultaneous occurrence of two (or more) faults. We can combine the 7 humidity levels with the nominal temperature, and the 7 temperatures with the nominal humidity. Two cases represent the same input (nominal humidity, nominal temperature) thus we need only 13 test cases.

* 9 test cases for points inside the rectangle (comfortable)
* 4 test cases for points outside the rectangle (uncomfortable)

The test cases represented by the 13 points are as shown.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Expected I/O | Actual I/O | Status | (humidity, temperature) |
| 1 | Humidity and Temperature: **50 70**  Comfortable |  |  | (nom, nom) |
| 2 | Humidity and Temperature: **40 70**  Comfortable |  |  | (min, nom) |
| 3 | Humidity and Temperature: **41 70**  Comfortable |  |  | (min+1, nom) |
| 4 | Humidity and Temperature: **59 70**  Comfortable |  |  | (max-1, nom) |
| 5 | Humidity and Temperature: **60 70**  Comfortable |  |  | (max, nom) |
| 6 | Humidity and Temperature: **50 65**  Comfortable |  |  | (nom, min) |
| 7 | Humidity and Temperature: **50 66**  Comfortable |  |  | (nom, min+1) |
| 8 | Humidity and Temperature: **50 74**  Comfortable |  |  | (nom, max-1) |
| 9 | Humidity and Temperature: **50 75**  Comfortable |  |  | (nom, max) |
| 10 | Humidity and Temperature: **39 70**  Uncomfortable |  |  | (min–1, nom) |
| 11 | Humidity and Temperature: **61 65**  Uncomfortable |  |  | (max+1, nom) |
| 12 | Humidity and Temperature: **50 64**  Uncomfortable |  |  | (nom, min-1) |
| 13 | Humidity and Temperature: **50 76**  Uncomfortable |  |  | (nom, max+1) |

* Edit **IndoorComfort** to read in both humidity and temperature. Revise the code to check both conditions in determining a comfortable environment.
* You don’t need to run all 13 tests. At a minimum, run tests #1, #5, #9, and #13.   
  Fill in the Actual I/O and Status columns. Confirm the four tests pass.
* Insert a screen print of your code solution that shows the console result from test #1.

### Testing Task #4 – PizzaPrice.java

**Decision Table Testing** is a technique to evaluate program behavior based on various combinations of inputs and their corresponding actions. It is particularly useful for testing scenarios where multiple conditions affect the outcome. A **decision table**, also called a cause-effect table, specifies the actions to perform for a given set of input conditions. A decision table has the following structure:

* **Conditions**: The different inputs that influence the outcome. A condition can have two or more values (true/false, yes/no/maybe, etc.).
* **Actions**: The outputs the system should produce.
* **Rules**: A unique combination of condition values and the corresponding action.

Consider the following program requirements:

*The base cost of a pizza is determined by the size (small=8.25, medium=12.25, large=16.25). A medium or large pizza is discounted by $1.00 for customers having at least 100 reward points. Write a program that reads in the size and points as input and prints the price.*

There are two input conditions that determine the price:

* pizza size {1=small, 2=medium, 3=large}
* reward points >= 100 {true, false}

There are 6 combinations for the 2 conditions. The decision table contains one rule per combination.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Pizza Price Decision Table** | | | | | | |
|  | **Rule 1** | **Rule 2** | **Rule 3** | **Rule 4** | **Rule 5** | **Rule 6** |
| **CONDITIONS** |  | | | | | |
| size (1/2/3) | **1** | **1** | **2** | **2** | **3** | **3** |
| points >= 100 (T/F) | T | F | T | F | T | F |
| **ACTIONS** |  | | | | | |
| price | 8.25 | 8.25 | 11.25 | 12.25 | 15.25 | 16.25 |

Notice that the reward points have no effect on the price of a small pizza. We can merge rules 1&2 and use a dash **-** to denote "don't care" for the points condition. The reduced decision table has 5 rules:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Pizza Price Decision Table** | | | | | |
|  | **Rule 1**  **(old rule 1 & 2)** | **Rule 2**  **(old rule 3)** | **Rule 3**  **(old rule 4)** | **Rule 4**  **(old rule 5)** | **Rule 5**  **(old rule 6)** |
| **CONDITIONS** |  | | | | |
| size (1/2/3) | **1** | **2** | **2** | **3** | **3** |
| points >= 100 (T/F) | - | T | F | T | F |
| **ACTIONS** |  | | | | |
| price | 8.25 | 11.25 | 12.25 | 15.25 | 16.25 |

To summarize, the decision table rules can be reduced by merging columns:

* Combine columns where certain conditions do not affect the outcome.
* Use “-” to denote that a condition’s value does not affect the outcome.

The decision table rules are used to create test cases as shown:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Expected I/O | Actual I/O | Status | Comment |
| 1 | Pizza Size 1=small,2=medium,3=large: **1**  Reward Points: **200**  Price: $8.25 |  |  | Rule 1  (no discount) |
| 2 | Pizza Size 1=small,2=medium,3=large: **2**  Reward Points: **200**  Price: $11.25 |  |  | Rule 2  (discount) |
| 3 | Pizza Size 1=small,2=medium,3=large: **2**  Reward Points: **50**  Price: $12.25 |  |  | Rule 3  (no discount) |
| 4 | Pizza Size 1=small,2=medium,3=large: **3**  Reward Points: **130**  Price: $15.25 |  |  | Rule 4  (discount) |
| 5 | Pizza Size 1=small,2=medium,3=large: **3**  Reward Points: **20**  Price: $16.25 |  |  | Rule 5  (no discount) |

The **PizzaPrice** class implements the requirements, but there is an error involving logical operator precedence. You can run the program online at <https://onlinegdb.com/YfjeGYDbJg>.

* Run the program for each test case and fill in the columns for **Actual I/O** and **Status** (Pass/Fail).
* Confirm that tests 1-4 pass, while test #5 fails.

Recall logical AND **&&** has higher precedence than logical OR **||**. The expression **a && b || c** is equivalent to **(a && b) || c**, which makes the entire expression true whenever **c** is true.

Fix **PizzaPrice** to ensure the price is discounted only for the correct conditions.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Expected I/O | Actual I/O | Status | Comment |
| 1 | Pizza Size 1=small,2=medium,3=large: **1**  Reward Points: **200**  Price: $8.25 |  |  | Rule 1  (no discount) |
| 2 | Pizza Size 1=small,2=medium,3=large: **2**  Reward Points: **200**  Price: $11.25 |  |  | Rule 2  (discount) |
| 3 | Pizza Size 1=small,2=medium,3=large: **2**  Reward Points: **50**  Price: $12.25 |  |  | Rule 3  (no discount) |
| 4 | Pizza Size 1=small,2=medium,3=large: **3**  Reward Points: **130**  Price: $15.25 |  |  | Rule 4  (discount) |
| 5 | Pizza Size 1=small,2=medium,3=large: **3**  Reward Points: **20**  Price: $16.25 |  |  | Rule 5  (no discount) |

* Edit **PizzaPrice** to fix the operator precedence error.
* Rerun every test and confirm all pass.
* Insert a screen print of your code solution that shows the console result from test #5.

### Testing Task #5 – Chapter 3 Project 3

NOTE: Submit your Chapter 3 Project 3 Java code to Revel for auto-grading.

You will apply **equivalence class partitioning** and **decision table testing** for Chapter 3 Project 3. The textbook instructions do not describe what to do if the user enters an invalid month such as 0 or 13. The Revel/CodeGrade tests also don't check for that error. You can assume the user enters a valid month between 1 and 12 when you implement project 3, i.e. don’t bother checking for an invalid month.

|  |  |  |  |
| --- | --- | --- | --- |
| **Year Equivalence Classes** | | | |
|  | **Leap Year** | **Common year** | **Common Year** |
| Divisible by 4 and  not divisible by 100 | Divisible by 400 | Not divisible by 4 | Divisible by 100 and  not divisible by 400 |
| 1.  2. | 1.  2. | 1.  2. | 1.  2. |

* Complete the table above by providing 2 sample years for each of the 4 equivalence classes.   
  A year that does not qualify as a leap year is known as a common year.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Days in a Month Decision Table** | | | | | | | | | | | | | |
|  | **Rule 1** | **Rule 2** | **Rule 3** | **Rule 4** | **Rule 5** | **Rule 6** | **Rule 7** | **Rule 8** | **Rule 9** | **Rule 10** | **Rule 11** | **Rule 12** | **Rule 13** |
| **CONDITIONS** |  | | | | | | | | | | | | |
| Month (1-12) | **1** | **2** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** | **11** | **12** |
| Leap Year (T/F) | - | T | F | - | - | - | - | - | - | - | - | - | - |
| **ACTION** |  | | | | | | | | | | | | |
| **Days** | 31 |  |  |  |  |  |  |  |  |  |  |  |  |

* Two conditions determine the #days: (1) the month and (2) leap vs common year. Fill in the action in the decision table above based on the conditions. Recall dash **-** represents “don’t care”.

A subset of rules from the decision table will be used to define test cases.

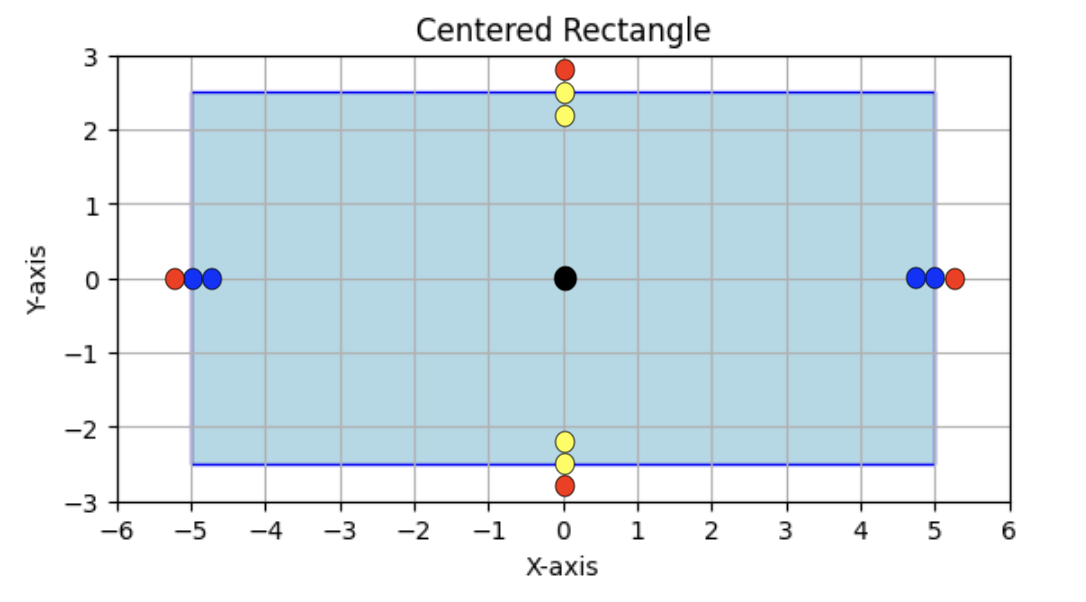
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Expected I/O | Actual I/O | Status | Comment |
| 1 | Enter a month in the year (e.g., 1 for Jan): **1**  Enter a year: **2000**  January 2000 has 31 days |  |  | Rule 1 |
| 2 |  |  |  | Rule 2 |
| 3 |  |  |  | Rule 3 |
| 4 |  |  |  | Rule 4 |
| 5 |  |  |  | Rule 5 |

* Define test cases for the **first 5 decision table rules**.Use a unique year from the various equivalence classes for each test case.
* Run your Project 3 solution for each test case and fill in the actual I/O and status based on the program execution.
* 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 programming project 3.

LESSONS LEARNED FOR PROJECT 3:

### Testing Task #6– Chapter 3 Project 5 Test

Programming project 5 reads in a point (x, y) and prints whether the point is within the rectangle having width 10 and height 5 and centered at (0,0).



|  |  |  |
| --- | --- | --- |
| **X Boundary Value Analysis (BVA)** | | |
| Outside range  min-0.1 | x in range [-5.0, 5.0]  min, min+0.1, nominal, max-0.1, max | Outside range  max + 0.1 |
|  |  |  |

* Complete the table above based on the range of x values [-5.0, 5.0]. Note the offset is 0.1 due to the boundary value being a double rather than an int.

|  |  |  |
| --- | --- | --- |
| **Y Boundary Value Analysis (BVA)** | | |
| Outside range  min-0.1 | y in range [-2.5, 2.5]  min, min+0.1, nominal, max-0.1, max | Outside range  max + 0.1 |
|  |  |  |

* Complete the table above based on the range of y values [-2.5, 2.5] and an offset of 0.1.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Expected I/O | Actual I/O | Status | (x, y) |
| 1 | Enter a point with two coordinates: **0 0**  Point (0.0, 0.0) is in the rectangle |  |  | (nom, nom) |
| 2 | Enter a point with two coordinates: **-5 0**  Point (-5.0, 0.0) is in the rectangle |  |  | (min, nom) |
| 3 |  |  |  | (min+0.1, nom) |
| 4 |  |  |  | (max-0.1, nom) |
| 5 |  |  |  | (max, nom) |
| 6 |  |  |  | (nom, min) |
| 7 |  |  |  | (nom, min+0.1) |
| 8 |  |  |  | (nom, max-0.1) |
| 9 |  |  |  | (nom, max) |
| 10 | Enter a point with two coordinates: **-5.1 0**  Point (-5.1, 0.0) is not in the rectangle |  |  | (min–0.1, nom) |
| 11 |  |  |  | (max+ 0.1, nom) |
| 12 |  |  |  | (nom, min-0.1) |
| 13 |  |  |  | (nom, max+0.1) |

* Define test cases 1-13 based on BVA values determined for the x and y variables. Test 1-9 are the various points inside the rectangle (black, blue, yellow), while tests 10-13 are the 4 red points outside the rectangle. Nominal is abbreviated “nom”.
* Run your Project 5 solution for each test case and fill in the actual I/O and status based on the program execution.
* 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 programming project 5.

LESSONS LEARNED FOR PROJECT 5: