**For the following function, model the input domain using both**

**the interface-based and the functionality-based approach.**

**/\*\***

**\* Given two maps with string keys and integer values,**

**return a new map containing only the keys that exist in both maps.**

**\* For each such key, the value should be the**

**absolute difference of the two values from the input maps.**

**\***

**\* Example:**

**\* map1 = {"A": 5, "B": 10, "C": 3}**

**\* map2 = {"B": 7, "C": 8, "D": 12}**

**\* Output: {"B": 3, "C": 5}**

**\***

**\* If there are no common keys, return an empty map.**

**\*/**

**public static Map<String, Integer> computeValueDifferences(Map<String, Integer> map1,**

**Map<String, Integer> map2);**

**Interface-Based Approach**

Two Parameters: map1 & map2

*Defining features and partitioning*

C1: **map1 is null {true, false}**

**C2: map1 is empty (map1.isEmpty()) {true, false}**

**C3: map2 is null {true, false}**

**C4: map2 is empty (map2.isEmpty()) {true, false}**

**Functionality-Basec Approach**

*Defining features and partitioning*

C5: containAtLeastOneCommonKey {true, false}

**BASE CHOICE COVERAGE (BCC)**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| C1:  map1 is null | C2:  map1 is empty | C3:  map2 is null | C4:  map2 is empty | C5: returns a common key | Expected |
| F | F | F | F | T | Feasible  (returns key) |
| T | F | F | F | T | infeasible  (X) |
| F | T | F | F | T | infeasible |
| F | F | T | F | T | infeasible  (X) |
| F | F | F | T | T | infeasible |
| F | F | F | F | F | feasible  ({}) |

Best Scenario: F F F F T

**Correctness Analysis**

(replacing infeasible cases with feasible cases)

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| C1:  map1 is null | C2:  map1 is empty | C3:  map2 is null | C4:  map2 is empty | C5: returns a common key | | Corrected Expectation | | --- |  |  | | --- | |  | |
| **T** | **F** | **F** | **F** | **T** | **Should throw an exception** |
| **F** | **T** | **F** | **F** | **F** | **Returns**  **empty map** |
| **F** | **F** | **T** | **F** | **T** | **Should throw an exception** |
| **F** | **F** | **F** | **T** | **F** | **Returns**  **empty map** |

**TEST CASES**

TEST CASE – 1: COMMON KEYS EXIST:

map1 = {"A": 2, "B": 3, "C": 5}

map2 = {"B": 6, "C": 2, "D": 8}

Output: {"B": 3, "C": 3}

TEST CASE – 2: NO COMMON KEYS:

map1 = {"X": 1, "Y": 2, "Z": 3}

map2 = {"A": 4, "B": 5, "C": 6}

Output: {}

TEST CASE – 3: ONE MAP IS EMPTY:

map1 = {}

map2 = {"A": 1, "B": 2, "C": 3}

Output: {}

TEST CASE – 4: BOTH MAPS ARE EMPTY:

map1 = {}

map2 = {}

Output: {}

TEST CASE – 5: **map1 is null** (should throw NullPointerException)

computeValueDifferences(null, {"A": 1, "B": 2})

TEST CASE – 6: **map1 is null** (should throw NullPointerException)

computeValueDifferences({"A": 3, "B": 4}, null)

**JUnit testing using AIGen**

package com.example.auditoryexercises.Example\_5.LaboratoryExercies01;  
  
import java.util.HashMap;  
import java.util.*Map*;  
  
public class ComputeValDiffs {  
 public static *Map*<String, Integer> computeValueDifferences(*Map*<String, Integer> map1, *Map*<String, Integer> map2) {  
 if (map1 == null || map2 == null) {  
 throw new NullPointerException("Input maps cannot be null");  
 }  
  
 *Map*<String, Integer> result = new HashMap<>();  
 for (String key : map1.keySet()) {  
 if (map2.containsKey(key)) {  
 result.put(key, Math.*abs*(map1.get(key) - map2.get(key)));  
 }  
 }  
 return result;  
 }  
}

package com.example.auditoryexercises.Example\_5.LaboratoryExercies01;  
  
import org.junit.jupiter.api.Test;  
  
import java.util.\*;  
  
import static org.junit.jupiter.api.Assertions.\*;  
  
class ComputeValueDifferencesTest {  
  
 @Test  
 void testCommonKeysExist() {  
 *Map*<String, Integer> map1 = *Map*.*of*("A", 2, "B", 3, "C", 5);  
 *Map*<String, Integer> map2 = *Map*.*of*("B", 6, "C", 2, "D", 8);  
 *Map*<String, Integer> expected = *Map*.*of*("B", 3, "C", 3);  
  
 *assertEquals*(expected, ComputeValDiffs.*computeValueDifferences*(map1, map2));  
 }  
  
 @Test  
 void testNoCommonKeys() {  
 *Map*<String, Integer> map1 = *Map*.*of*("X", 1, "Y", 2, "Z", 3);  
 *Map*<String, Integer> map2 = *Map*.*of*("A", 4, "B", 5, "C", 6);  
  
 *assertTrue*(ComputeValDiffs.*computeValueDifferences*(map1, map2).isEmpty());  
 }  
  
 @Test  
 void testOneMapEmpty() {  
 *Map*<String, Integer> map1 = Collections.*emptyMap*();  
 *Map*<String, Integer> map2 = *Map*.*of*("A", 1, "B", 2, "C", 3);  
  
 *assertTrue*(ComputeValDiffs.*computeValueDifferences*(map1, map2).isEmpty());  
 }  
  
 @Test  
 void testBothMapsEmpty() {  
 *Map*<String, Integer> map1 = Collections.*emptyMap*();  
 *Map*<String, Integer> map2 = Collections.*emptyMap*();  
  
 *assertTrue*(ComputeValDiffs.*computeValueDifferences*(map1, map2).isEmpty());  
 }  
  
 @Test  
 void testMap1Null() {  
 *Map*<String, Integer> map2 = *Map*.*of*("A", 1, "B", 2);  
  
 *assertThrows*(NullPointerException.class, () -> ComputeValDiffs.*computeValueDifferences*(null, map2));  
 }  
  
 @Test  
 void testMap2Null() {  
 *Map*<String, Integer> map1 = *Map*.*of*("A", 3, "B", 4);  
  
 *assertThrows*(NullPointerException.class, () -> ComputeValDiffs.*computeValueDifferences*(map1, null));  
 }  
}

Once you have defined the characteristics,

you need to divide them into blocks, and

answer the following questions:

**A) Is the partitioning of the input parameters such that it ensures that the partitions are disjoint?**

**Why? If not, alter the partitioning to ensure this property is satisfied.**

Features C1, C2, C3, C4 are true/false type expressions.

Accordingly, the partitioning of all four features will be into 2 blocks – true T and false F.

With this division, we are assured that these features and their blocks satisfy the disjoint property,

since a map cannot be both null and not null at the same time.

Also, a map cannot be empty and contain elements at the same time.

Feature C5 also satisfies the disjointness property because we will either not have a common key or we will have at least one.

**B) Is the partitioning of the input parameters such that it ensures that the partitions cover the entire domain? Why? If not, alter the partitioning to ensure this property is satisfied.**

**The partitioning of the input parameters satisfies the completeness property. Each of the input states is covered.**

**C) Choose a base test and list all the necessary tests to satisfy the Base Choice Coverage (BCC) criterion. How many tests did you get?**

**map1 = {"A": 2, "B": 3, "C": 5}**

**map2 = {"B": 6, "C": 2, "D": 8}**

**Expected Output: {"B": 3, "C": 3}**

**The Base Test plus 5 variations give us 6 test cases in total.**

**D) Write JUnit tests using the BCC criteria for ISP coverage. Try to use AI tools to help you!**