

Quiz Section 2: Testing and Immutable Reasoning – Solutions

Task 1 – Bop it! Twist it! Pull it! Test it!

For each of the following, consider the method implementation and determine whether the given tests satisfy our testing heuristics:

```
a) /*
   * Returns the amount of money spent rounded up to the nearest 5.
   * @param amount The amount spent
   * @requires 0 <= amount <= 9
   * @return 0 if amount = 0, 5 if 0 < amount <= 5, and 10 otherwise.
   */
public static int amountSpent(int amount) {
    if (amount == 0) {
        return 0;
    } else if (amount <= 5) {
        return 5;
    } else {
        return 10;
    }
}

// Given tests:
@Test
void testAmountSpent() {
    assertEquals(0, amountSpent(0), "0 does not round");
    assertEquals(5, amountSpent(4), "4 should round up to 5");
    assertEquals(10, amountSpent(9), "9 should round up to 10");
}
```

No! While branch and statement coverage are achieved, since there are only 10 possible values, we have to test all 10!

```

b) /*
    * Attempts to purchase an item with price itemPrice, and returns
    * the remaining money. If the price is over 100 or the price is
    * higher than the available money, returns the available money.
    * Otherwise, returns the difference between the two.
    * @param availableMoney The amount of money available
    * @param itemPrice The price of the item
    * @requires availableMoney >= 0 && itemPrice > 0
    * @return The remaining money after the purchase, or
    *         all the available money if the purchase cannot be made.
    */
public static int buyItem(int availableMoney, int itemPrice) {
    if (itemPrice > 100 || availableMoney < itemPrice) {
        return availableMoney;
    } else {
        return availableMoney - itemPrice;
    }
}

// Given tests:
@Test
void testBuyItem() {
    assertEquals(102, buyItem(102, 101), "Should not buy items > 100 in cost.");
    assertEquals(0, buyItem(1, 1), "Should buy item and subtract cost, resulting in 0.");
    assertEquals(1, buyItem(2, 1), "Should buy item and subtract cost, resulting in 1.");
    assertEquals(1, buyItem(1, 2), "Should not buy item with cost > availableMoney.");
}

```

These tests are sufficient, since all statements and branches are covered.

Task 2 – East or Test, JUnit is the Best!

For each of the following, consider the following method specification and implementation. Then, write JUnit tests that fully cover our testing heuristics (the method scaffold is only given for part a):

```
a) /**
 * Compares two integers
 * @param a the first integer to compare
 * @param b the second integer to compare
 * @returns a negative value if a < b, 0 if a == b, and
 * a positive value if a > b
 */
public static int compare(int a, int b) {
    int comparison = 0;
    if (a > b) {
        comparison = 1;
    } else if (a < b) {
        comparison = -1;
    }
    return comparison;
}

@Test
public void testJoin() {
    assertTrue(compare(2, 1) > 0); // 1st branch coverage
    assertTrue(compare(1, 2) < 0); // 2nd branch coverage + rest of
                                    // statement coverage
    assertTrue(compare(1, 1) == 0); // 3rd branch coverage
}
```

```

b) /**
 * Returns the sum of the array
 * @param arr an array of integers
 * @requires arr is not null
 * @returns the sum of the elements in arr or 0 if
 * arr is empty
 */
public static int arraySum(int[] arr) {
    if (arr == null) {
        throw new IllegalArgumentException("arr cannot be null");
    }
    int sum = 0;
    for (int i = 0; i < arr.length; i++) {
        sum += arr[i];
    }
    return sum;
}

@Test
public void testArraySum() {
    int[] arr1 = new int[0];
    assertEquals(0, arraySum(arr1)); // 0 loop iterations

    int[] arr2 = {5};
    assertEquals(5, arraySum(arr2)); // 1 loop iteration

    int[] arr2 = {0, 3, 3, 1, 0};
    assertEquals(7, arraySum(arr3)); // 2+ loop iterations
    // Statement and Branch coverage are both covered by the tests above
    // We don't need to test the first if statement as it is a
    // disallowed input
}

```

```

c) /**
 * Returns the nth fibonacci number
 * @param n the index of the desired fibonacci number
 * @requires n >= 0
 * @returns the nth fibonacci number
 */
public static int fibonacci(int n) {
    if (n < 0) {
        throw new IllegalArgumentException("n must be non-negative");
    } else if (n == 0) {
        return n;
    } else if (n == 1) {
        return n;
    }
    return fibonacci(n - 1) + fibonacci(n - 2);
}

@Test
public void testFibonacci() {
    assertEquals(0, fibonacci(0)); // 0 recursive calls +
    assertEquals(1, fibonacci(2)); // 1 recursive call +
        // full branch and statement coverage
    assertEquals(5, fibonacci(5)); // 2+ recursive calls
    // The tests above give full statement coverage. (We don't need branch or
    // statement coverage for the first if statment/exception because it is
    // a disallowed input)
}

```

Task 3 – Just Give Me A(mmutable) Reason(ing)

For each of the following methods, collect facts to show that the code is correct according to the specification (i.e., assuming the precondition, the code fulfills the postcondition).

a) Consider the following Java method.

```
/*
 * Performs a magic math trick on a starting number.
 * @param x The starting number
 * @requires x > 0
 * @return x
 */
public static int magicNumber(int x) {
    int a = x * 2;
    int b = a + 4;
    int c = b / 2;
    int d = c - 2;
    return d;
}
```

Collect all facts from the method. Do they prove that the method is correct?

$d = c - 2$ and $c = b / 2$ and $b = a + 4$ and $a = x * 2$ and $x > 0$.

Yes! To prove the method is correct, we can use these facts to find the true value of d :

$$\begin{aligned}d &= c - 2 \\ &= (b / 2) - 2 \\ &= ((a + 4) / 2) - 2 \\ &= (((x * 2) + 4) / 2) - 2 \\ &= (x + 2) - 2 \\ &= x\end{aligned}$$

Since $d = x$ follows from the facts in P_1 , the method returning d satisfies the `@return` specification.

b) Consider the following Java method:

```
/*
 * Returns the amount of money spent rounded up to the nearest 5
 * @param amount The amount spent
 * @requires 0 <= amount <= 9
 * @return 0 if amount = 0, 5 if 0 < amount <= 5, and 10 otherwise.
 */
public static int amountSpent(int amount) {
    if (amount == 0) {
        return 0; // Return #1
    } else if (amount <= 5) {
        return 5; // Return #2
    } else {
        return 10; // Return #3
    }
}
```

Collect facts at each return. Do the three return statements satisfy the specification?

Return #1: amount = 0.

Return #2: amount != 0 and amount <= 5 and 0 <= amount <= 9.

Return #3: amount != 0 and 5 < amount and 0 <= amount <= 9.

Yes! In the first branch, 0 rounds to 0. In the second branch, amount must be between 1 and 5, and correctly returns 5. In the third branch, amount must be between 6 and 9, and correctly returns 10.

c) Consider the following buggy Java method:

```
/*
 * Returns the shipping fee based on the total cost of the order.
 * @param cost The total cost of the order
 * @requires 0 <= cost <= 100
 * @return 0 if cost >= 50, 10 if 0 < cost < 50, and 0 if cost == 0.
 */
public static int shippingFee(int cost) {
    if ((cost + 49) / 50 == 1) {
        return 10; // Return #1
    } else {
        return 0; // Return #2
    }
}
```

Collect facts at each return. Then, explain why the method fails to satisfy its specification.

Return #1: $(\text{cost} + 49) / 50 == 1$ and $0 \leq \text{cost} \leq 100$.

Return #2: $(\text{cost} + 49) / 50 \neq 1$ and $0 \leq \text{cost} \leq 100$.

No! The first branch incorrectly executes when the cost is exactly 50 (since $99 / 50 = 1$), returning 10. The specification states that if $\text{cost} \geq 50$, the method must return 0.