CSE 331: Software Design & Implementation Spring 2026

Homework 1

Due: Wednesday, April 15th, 11:59pm

Task 1 – Every Log Has Its Day [6 pts]

We plan to provide the following method:

```
/** Calculates the integer, base-2 logarithm of n.
 * ...
 * @return the integer k such that 2^k <= n < 2^(k+1)
 */
public static int log2(int n);
```

When n is a power of 2, the integer k from the spec will satisfy $2^k = n$, i.e., $k = \log_2(n)$. When n is not a power of 2, $\log_2(n)$ is not an integer, and the method returns $\log_2(n)$ rounded down to the nearest integer (i.e., the *largest* integer k such that $2^k \leqslant n$).

a) This specification precisely defines the return value for positive `n`, but it does not make sense when `n` is not positive. Give **two** distinct ways of turning this into a specification that fully defines behavior for all integer inputs. Non-positive values should be treated as invalid inputs and your specifications should indicate that.

Your two specifications must still return the value described in the **original** specification when `n` is positive, but they should be **incomparable** with each other. Format your specifications using 331-style Javadoc tags. Your specifications must explicitly address what happens for invalid inputs using the appropriate Javadoc tags: do not try to work around the issue.

b) The original specification describes the return value as $\log_2(n)$ rounded *down*. What is another, equally reasonable but **incomparable**, way to describe the return value when n is not a power of 2?

c) Rewrite the `@return` section of the original specification to reflect your answer from part (b). Your new specification should use similar formulas to the original (i.e., inequalities involving powers of 2) and should be a complete drop-in replacement for the original `@return`.

Note: Your specification must fully define the return value for all positive `n`, including when n *is* a power of 2.

Task 2 – In Bloom [12 pts]

One of your TAs, Faraz, is trying to get to his Distributed Systems section in the Music Building. However, since the cherry blossom trees in the Quad have bloomed, the Quad is crowded with visitors to the UW campus and Faraz is unable to get there on time! Each visitor performs some activity, such as taking pictures, touring campus, or wrestling geese.

In order to help Faraz out, we plan to provide him with a method to count how many visitors there are at the Quad doing some specified activity at some specified time today. (Note: A `LocalTime` is a Java object representing a time without a date or time-zone):

```
/**
 * Returns the number of visitors at the Quad.
 * ...
 */
public static int countVisitors(Activity activity, LocalTime time);
```

The spec is too vague—is the number of visitors filtered by `activity` or `time`? Is it the total number of visitors all day? In order to provide the method, we must fill in the rest of the specification.

We are considering the following alternatives:

```
Spec A:
@throws NullPointerException if time is null or activity is null.
@return the total number of campus visitors performing activity
        within one hour of the given time.

Spec B:
@requires activity is not null
@return the number of campus visitors performing activity at the
        given time or the number of all campus visitors today if
        time is null.

Spec C:
@requires activity is not null and time is not null
@return the number of campus visitors performing activity at the given
        time today.
```

a) In the following questions, decide how the two specs compare, and explain your reasoning briefly in 1-2 sentences.

- **i.** Specs **A** and **B**.
- **ii.** Specs **A** and **C**.
- **iii.** Specs **B** and **C**.

For example, you may say "Spec X is stronger than Spec Y. Spec X has a stronger postcondition because it specifies that the output is a positive integer, while Spec Y says it's any integer. Both specifications have the same precondition."

b) We will now look at various implementations of `countVisitors`. First, note that activities and times are chunked together into a single class called `ActivityTime`, which represents a tuple of an activity and time. We then store every valid mapping between activity-time pairs and the number of visitors doing that activity at that time in a `Map<ActivityTime, Integer>` called `counts`. Our implementations of countVisitors are as follows:

i.

```
public static int countVisitors(Activity activity, LocalTime time) {
    if (activity == null) {
        throw new IllegalArgumentException();
    }
    if (time == null) {
        int total = 0;
        for (int numVisitors : counts.values()) {
            total += numVisitors;
        }
        return total;
    }
    ActivityTime at = new ActivityTime(activity, time);
    return counts.get(at);
}
```

ii.

```
public static int countVisitors(Activity activity, LocalTime time) {
    ActivityTime at = new ActivityTime(activity, time);
    return counts.get(at);
}
```

For each of the above implementations, describe which one of the aforementioned specifications it satisfies and which ones it does not, along with a brief explanation as to why it does so. Remember that when we compare the strengths of specifications, we know whether or not satisfying one spec implies satisfying another.

c) Bob has written two new Specs X and Y for `countVisitors` as well as implementations I_X (which implements Spec X) and I_Y (which implements Spec Y). However, we can't see the code because cosmic rays have bombarded Bob's laptop, corrupting his files. Despite this nefarious setback, Bob has graciously provided us with the following spec comparison table:

	A	**B**	**C**
X	—	W	W
Y	—	—	S

Based on the spec comparison table, which of A/B/C are I_X and I_Y guaranteed to satisfy?

Task 3 – Stop and Smell the Specifications [20 pts]

A florist is arranging bouquets for customers. The florist has an inventory of flower stems, and each stem has a price and a kind. Your job is to implement a method that chooses some stems from the inventory to form a bouquet whose total price is within a required range.

This problem has coding parts and written parts. Submit your code to the "HW1 Coding" assignment on Gradescope. Submit your written answers along with the rest of the problems to the "HW1 Written" assignment on Gradescope. AI use is not permitted for this problem.

a) Start by reading the specification for Stem and the static method arrangeBouquet below.

```
public class Florist {
    /**
     * An individual flower stem in the florist's inventory.
     *
     * @param kind The kind of flower (e.g. tulip, rose, daisy). Must not
     *             be null or empty.
     * @param price The price of the flower in cents. Must be non-negative.
     */
    public static record Stem(String kind, int price) { ... }

    /**
     * Arrange a bouquet of flowers using stems from the inventory.
     *
     * @param inventory The flower stems in the florist's inventory. Must
     *             not be null, and none of the stems it contains may be null
     * @param minimumPrice The minimum price of the bouquet in cents. Must
     *                  be non-negative.
     * @param maximumPrice The maximum price of the bouquet in cents. Must
     *                  be non-negative.
     * @requires
     *   - The minimum price is less than or equal to the maximum price.
     *   - The total price of the stems in the inventory is greater than
     *     or equal to the minimum price.
     *   - Every stem in the inventory has price less than or equal to
     *       maximumPrice - minimumPrice
     * @returns A non-null list of stems (bouquet) from the inventory with
     *          total price between minimumPrice and maximumPrice.
     *          A stem must not occur in the bouquet more times than it
     *          appears in the inventory.
     */
    public static List<Stem> arrangeBouquet(
        final Stem[] inventory,
        final int minimumPrice,
        final int maximumPrice
```

```
    ) { ... }
}
```

Notice that the specification places some requirements on the returned bouquet, but there can still be multiple allowed return values for a particular input. Write two implementations of arrangeBouquet that both satisfy the specification but are different from each other. In a comment within each implementation, briefly describe how that implementation chooses its bouquets.

b) Write unit tests for both of your implementations of arrangeBouquet. You should meet the test coverage standard from class and avoid testing on invalid inputs.

c) Give an example input where your two implementations return different outputs.

d) Consider your example input from part C. Suppose we want to write a unit test that calls arrangeBouquet on this input, and we want this unit test to pass on both of your implementations of arrangeBouquet. Explain why this is more challenging than writing a unit test for a single implementation.

e) How might we overcome the challenge from part D and write a unit test that will pass on any correct implementation of arrangeBouquet?

Task 4 – In the Bugout [9 pts]

It's baseball season! Benoit Le wants to know how many estimated innings his favorite pitcher, Le Benoit, will play in a season. He has written the following function to help him out.

```
/**
* Calculated the total estimated innings pitched for a starter in
* a 5-man rotation.
*
* @param startGame    the game number to begin the calculation (inclusive)
* @param endGame      the game number to end the calculation (inclusive)
* @param rotationPosition  the pitcher's spot in the rotation
*                          (0-based indexing e.g., 4 for the "fifth starter")
* @param avgInningsPerStart the average innings the pitcher
*                           lasts per game
* @requires 0 <= rotationPosition <= 4
            avgInningsPerStart >= 0
            startGame <= endGame
* @return the total estimated innings pitched in that range
*/
public static int totalInnings(int startGame, int endGame,
                           int rotationPosition, int avgInningsPerStart) {
    if (rotationPosition < 0 || rotationPosition > 4
        || avgInningsPerStart < 0 || startGame > endGame) {
        throw new IllegalArgumentException();
    }

    int totalInnings = 0;
    for (int game = startGame; game < endGame; game++) {
        if (game % 5 == rotationPosition) {
            totalInnings = totalInnings + avgInningsPerStart;
        }
    }

    return totalInnings;
}
```

a) Consider the following test and determine whether the given test satisfies 331 testing heuristics. Explain how each testing heuristic is either satisfied by a specific testcase or not.

```
@Test
void testTotalInnings() {
    // A
    assertEquals(6, totalInnings(4, 5, 4, 6));

    // B
```

```
        assertEquals(12, totalInnings(10, 25, 3, 4));

        // C
        assertEquals(0, totalInnings(9, 9, 3, 5));
    }
```

b) Senior developer Isayiah peeks over Benoit's shoulder, claiming that while the tests pass, the implementation fails to satisfy the specification under specific inputs. Identify and write out a test case that would reveal this and explain why the current tests failed to catch it.

c) Now imagine a more complex function with many more lines (maybe a function to calculate total Mariner wins this season). Weigh the pros and cons of using coverage-based testing as your only safety metric. What role does testing actually play if it cannot provide a proof of correctness?

Task 5 – Reasons Change and Our Love Went Cold [9 pts]

For each of the following methods, collect facts to show that the code is correct or incorrect according to the specification (i.e., assuming the precondition, the code fulfills the postcondition). In your answer, write out these collected facts and explain how the postcondition is satisfied/not satisfied in each case.

a)

```
/**
* Returns a negative integer
*
* @param x, y
* @requires x > 0
* @return a negative integer
*/
int returnNegative(int x, int y) {
    int a = 3;
    int b = -x;
    int c = 2 * a;
    return b * c * y;
}
```

b)

```
/**
* Calculates your discount in dollars, given a bill
*
* @param bill
* @requires bill >= 0
* @return bill/10 if 0 <= bill < 10, bill/5 otherwise
*/
int calculateDiscount(int bill) {
    if (bill < 0) {
        return bill / 20;
    }
    else if (bill < 10) {
        return bill / 10;
    }
    else {
        return bill / 5;
    }
}
```

c)

```
/**
* Calculates the number of digits a number has
*
* @param tens, ones
* @requires 0 <= tens < 10, 0 <= ones < 10
* @return 2 if tens >= 1. Otherwise,
          1 if ones >= 1 and 0 if ones = 0.
*/
```

```
int calculateDigits(int tens, int ones) {
    if (ones >= 1) {
        return 1;
    }
    else if (tens >= 1) {
        return 2;
    }
    else {
        return 0;
    }
}
```