3S03 - Software Testing
Assignment 2

Mark Hutchison
hutchm6@mcmaster.ca

Jatin Chowdhary
chowdhaj@mcmaster.ca

April 13, 2022

Contents

Question 1 - Sprite Test Plan

Table 1: Question 1 Table

Objectives and Goals	The company in question has been tasked with the development of a First-Person Shooter (FPS) video game. Typically, first person shooters are programmed in object-oriented languages like C++ and C#. However, the `Sprite` test plan will be discussed as if the video game is programmed in Java. This is because we are more familiar with Java. Video games typically involve many elements being on screen at a time. Hence, it makes sense to create a uniform class system that can manage all of the on-screen sprites at a time.
Requirements for Review	The purpose of the `Sprite` class is to efficiently and effectively collect and maintain the information related to an in-game `Sprite` object. The class will be responsible for storing and maintaining the position of the `Sprite` instance, its location on a playing field, and recording/applying the bitmap of the `Sprite`. In addition to these values, we will need methods in the class responsible for managing movement of the `Sprite`'s position, field location, and collision detection.
Specification Testing	The specification should clearly describe what information is available and accessible within the `Sprite`. This includes the type of the information and a brief description - what does the data represent. Standard *getter* and *setter* tests should be developed according to the specifications to ensure data consistency. In addition, methods are also detailed in the specification; assuming it is well written. To that effect, if the *getter* and *setter* methods can be trusted, then they should be tested. Fortunately, these can be turned into unit tests, and automated to check against the descriptions in the specification.

Continued on next page

Table 1: Question 1 Table (Continued)

Implementation Testing	With respect to testing, having an actual implementation is very useful, because a QA team can manually test the implementation of the specification. Informally speaking, in the world of video games, if you want to break collision-detection or `Sprite` rendering, then you should hire a group of professional speed runners. Professional speed runners can break a video game - the implementation - in ways developers cannot fathom during development. The primary goal of implementation testing is to find areas in the code that operate as specified, but (potentially) in the wrong order or in an unforeseen manner. Furthermore, the specification - assuming it is well written - does not guarantee that the resulting implementation will be bug-free. For example, just because the specification describes `Sprite`'s in isolation, it does not mean that they will work as intended in a real system.
Interaction Testing	In order to hypothesize, or anticipate, what can break during interaction testing, it is imperative to think about what data is being updated (in the system). For instance, collision-detection is typically calculated in one instance at a constant rate. Consequently, players can take advantage of this by manipulating their character model at the right time, or in the right way, to break collision-detection. For example, the popular survival horror game "Five Nights at Freddy's" contains a game breaking bug. If you jump and move into the right wall beside the stairs of Gator Golf, you fall right through the wall and into the void below. Similarly, first-person shooters like *Call of Duty* are riddled with "elevator" bugs, where smashing buttons and moving the character in a specific way can trigger the player to fly straight up into the air. Based off these examples, it is very important that we ensure the collision-detection can handle all of the various forms of movement. Also, it is a good idea to check if movement can break the bitmap itself - as this is pretty common in amateur `Sprite` development, according to my Graphics professor.
Additional Information *(Something to think about)*	The requirements for this first-person shooter have been quite vague. This is an issue that needs to be addressed **before** any development begins. For instance, we need to be made aware of the target demographic. For example, if the target demographic is professional players, then mechanisms like collision-detection need to be thoroughly tested. In addition, these systems need to be extremely accurate. After all, the livelihood of professional video game players relies on the accuracy and speed of mechanisms like collision-detection. On the other hand, if the target market is casual players, then some error can be tolerated without compromising the final product and integrity of the game. Furthermore, the hardware upon which the game will run on needs to be brought in preliminary discussions, because it will help us (the developers) understand what can and cannot be done.

Question 2 - Pre and Post Conditions for Linked Sets

1. Class Invariant:
   - Condition 1: If the linked set contains no items, it would contain a null "current" pointer, would return a count of 0 items, and wouldn't be able to call the first, has, at, item, or move item methods. Denoted in boolean form as
     $linked\_set = NULL \Rightarrow current = NULL \wedge count() = 0 \wedge is\_empty() \wedge \neg(first() \vee at(i) \vee has(v) \vee item() \vee move\_item(v))$.
   - Meanwhile, there are a certain amount of properties that exist for a filled linked set. Denoted in boolean form as
     $linked\_set \neq NULL \Rightarrow current \neq NULL \wedge count() \geq 1 \wedge \neg is\_empty()$.
     The behavior of individual methods is dependent on the current state of the linked set, meaning their individual post and pre conditions tell us more about their functionality and can't be universally assumed.
   - Condition 3: Ironically having no Discrete Math notation, we also have guarantees on Type Information, however these are covered in the default type system, as well as each condition breakdown below.
2. `linked_set()`
   - Preconditions:
     - There is no data to have a precondition on, thus no precondition.
   - Postconditions:
     - The postconditions of the constructor are simply validators on the initialization of the linked set. This would include:
       * The internal Linked List data structure is empty, or initialized to a null node.
       * The size of the linked set is 0 due to the fact that it is empty.
       * A null pointer would be established to represent our "current" node.
3. `G first()`
   - Preconditions:
     - The function assumes that there is an element at the front of the linked set to return the value of. Therefore, the Linked Set may not be empty. (i.e. a $count() > 0$ or $head \neq$ `NULL` condition)
   - Postconditions:
     - Due to the preconditions, we know the output set doesn't change, meaning that the set remains non-empty with the same *count*.
     - The output element must be a valid member of type G.
4. `G at(int i)`
   - Preconditions:
     - The value of i must be greater than, or equal to 0 (unless not 0 indexed, then 1).
     - The value of i must be less than the total amount of nodes in the Linked List (or equal to if not 0-indexed).
   - Postconditions:
     - Due to the preconditions, we know the output set doesn't change, meaning that the set remains non-empty with the same *count*.
     - The output element must be a valid member of type G.
5. `boolean has(G v)`
   - Preconditions:
     - The Linked List may be empty. It can never have an element of value v if it is empty. If this conditions fails though, the user may automatically return false instead of throwing an exception. Either way, it is a precondition that must be considered according to specifications.
   - Postconditions:

    - Due to the preconditions, we know the output set doesn't change, meaning that the set has the same *count*.
    - Simply return the result of $v \in$ `Linked_Set<G>` s. There really is no postconditions to this function.

6. `G item()`
    - Preconditions:
        - The "current" node pointer must be instantiated.
        - The set must not be empty.
    - Postconditions:
        - Due to the preconditions, we know the output set doesn't change, meaning that the set remains non-empty with the same *count*.
        - Since this returns an item of type G, it must be a valid member of the set. Apart from that, there really are no postconditions.

7. `int count()`
    - Preconditions:
        - Depending on the specifications, there may be no preconditions. But, if the specification states that a set must not be empty, then that is a precondition. However, majority of implementations do not enforce this precondition, so it is likely not a requirement.
    - Postconditions:
        - If emptiness is illegal, then a postcondition could include the set remaining non-empty. Regardless, the *count* of the set shouldn't change.
        - Returns an integer, or an error depending on earlier mentioned preconditions. There likely is no postconditions.

8. `boolean is_equal(Linked_Set<G> t)`
    - Preconditions:
        - Logically speaking, even an empty linked set could be equal to another empty linked set. So this wouldn't be an issue. However, depending again on specifications, a precondition may be required to enforce both are non-empty.
        - Also, since this is asking a question, it likely means you wouldn't require the two linked sets to contain the same count of elements. Therefore, this is not a requirement.
            - However, for coding purposes, if two linked sets don't have the same count of elements, then it is impossible for them to be equal.
    - Postconditions:
        - Returns a simple boolean, or potentially an error depending on earlier mentioned preconditions regarding emptiness.
        - If emptiness is illegal, then a postcondition could include the set remaining non-empty. Regardless, the *count* of the set shouldn't change.

9. `void move_item(G v)`
    - Preconditions:
        - Because this requires a cursor item to be present, the Linked Set must not be empty.
        - The cursor instance must exist.
        - The wording is slightly odd, but it sounds like v must be in the linked set in order to be moved to the left of the cursor. Therefore, it should pass the *has* function as a precondition.
    - Postconditions:
        - Again, due to the wording given, a post condition may include that the set may not grow in size. The *count* must remain consistent.
        - Because the count cannot change, the output set must not be empty either, and must include v due to preconditions not changing.

Question 3 - Fuel Tracker Testing Specifications

From a high level overview, the fuel tracker application needs to handle high or low volatility fuel. Based on this input, the application needs to determine the correct and appropriate settings for the truck of fuel. Somethings we can assume that are defined in the program are:

- The capacity of the tank (i.e. 1200) defined as a constant integer. We will call it `TOTAL_CAPACITY`.
- The expansion space of the tank (i.e. 800) defined as a constant integer. We will call it `SAFE_CAPACITY`.
- The cutoff capacity for high volatile fuels (i.e. 80%) defined as a constant integer and as a float. In other words, both 80 and 0.8 are defined as constant integer and float, respectively. We call these `CUTOFF_CAPACITY_PERCENT` and `CUTOFF_CAPACITY_DECIMAL`, respectively.
- The type of the fuel(s) defined as an *ENUM* in a separate class. For the purpose of this question, we will assume only 2 types of fuels are defined in the *ENUM* class, and they are `HIGH_VOLATILITY` and `LOW_VOLATILITY`.

Note: It is better to define the capacities as a constant float, rather than a constant integer. However, there is no strict ruleset, and for simplicity sake we will use integers.

The following test cases pertain to our fuel tracker application. I decided to use pseudocode rather than *jUnit* to write out each test case. This is because a mission critical application like fuel tracking should be written in a safe language. Hence, the test cases are written in pseudocode, allowing for maximum adaptability.

1. Test Case: *Return Total Capacity For Tank*
   - Name: testTotalCapacityForTank
   - Description: Checks if the application returns the correct total capacity of the tank. This capacity/number is used to determine how much `LOW_VOLATILITY` fuel can be loaded.
   - Input(s): fuelType = `LOW_VOLATILITY`
   - Pre-Condition: *N/A*
   - Test Condition: if (fuelType == `LOW_VOLATILITY`) then return `TOTAL_CAPACITY`.
   - Expected Output: `TOTAL_CAPACITY` is returned to the caller (*and maybe even printed to the screen*).
   - Post-Condition: *N/A*
2. Test Case: *Return Safe Capacity For Tank*
   - Name: testSafeCapacityForTank
   - Description: Checks if the application returns the correct total capacity of the tank. This capacity/number is used to determine how much `HIGH_VOLATILITY` fuel can be loaded.
   - Input(s): fuelType = `HIGH_VOLATILITY`
   - Pre-Condition: *N/A*
   - Test Condition: if (fuelType == `HIGH_VOLATILITY`) then return `SAFE_CAPACITY`.
   - Expected Output: `SAFE_CAPACITY` is returned to the caller (*and maybe even printed to the screen*).
   - Post-Condition: *N/A*
3. Test Case: *Load High Volatility Fuel*
   - Name: testLoadForHighVolatilityFuel
   - Description: Demonstrates that the application can correctly check if the input fuel is `HIGH_VOLATILITY`, and then load the fuel to the correct amount, `SAFE_CAPACITY`.
   - Input(s): fuelType = `HIGH_VOLATILITY`, fillAmount = `SAFE_CAPACITY`
   - Pre-Condition: currentTankLoad < `SAFE_CAPACITY` *(We prefer the tank to be empty to test this, but it is also plausible that some fuel already exists in the tank.)*
   - Test Condition: if (fuelType == `HIGH_VOLATILITY`) then fill tank with `HIGH_VOLATILITY` fuel upto `SAFE_CAPACITY`.
   - Expected Output: Tank is successfully filled to `SAFE_CAPACITY`

- Post-Condition: Tank contains a total of `SAFE_CAPACITY` litres of `HIGH_VOLATILITY` fuel.

4. Test Case: *Load Low Volatility Fuel*
   - Name: testLoadForLowVolatilityFuel
   - Description: Demonstrates that the application can correctly check if the input fuel is `LOW_VOLATILITY`, and then load the fuel to the correct amount, `TOTAL_CAPACITY`.
   - Input(s): fuelType = `LOW_VOLATILITY`, fillAmount = `TOTAL_CAPACITY`
   - Pre-Condition: currentTankLoad < `TOTAL_CAPACITY` *(We prefer the tank to be empty to test this, but it is also plausible that some fuel already exists in the tank.)*
   - Test Condition: if (fuelType == `LOW_VOLATILITY`) then fill tank with `LOW_VOLATILITY` fuel upto `TOTAL_CAPACITY`.
   - Expected Output: Tank is successfully filled to `TOTAL_CAPACITY`
   - Post-Condition: Tank contains a total of `TOTAL_CAPACITY` litres of `LOW_VOLATILITY` fuel.

5. Test Case: *Incorrect Fuel Type Argument*
   - Name: testIncorrectFuelType
   - Description: Tests the application's ability to handle exceptions pertaining to incorrect fuel type as input. *Note: We have no information about how this program is implemented. Perhaps the user interface requires the user to manually type the name of the fuel they wish to load. Alternatively, a dropdown menu can be presented to the user. In this case, the fuelType can be NULL, if no option is selected. Regardless, we need to ensure that our application can handle incorrect fuel type input(s) or argument(s).*
   - Input(s): (fuelType = OTHER) *Note: If the application requires the user to manually type the name of the fuel, then this test would look different. In this case, we are testing for a String mismatch.*
   - Pre-Condition: *N/A*
   - Test Condition: if ((fuelType != `LOW_VOLATILITY`) || (fuelType != `HIGH_VOLATILITY`)) then raise/throw Exception
   - Expected Output: Throw exception and print error (indicating that the fuel type is not valid) (*and possibly allow user to retry, depending on the circumstance*).
   - Post-Condition: *N/A*

6. Test Case:
   - Name: testIllegalLoadLimit
   - Description: Demonstrates that the application can detect if incorrect load limits are specified (by the user or programmer).
   - Input(s): `SAFE_CAPACITY` = 1000, `TOTAL_CAPACITY` = 1200
   - Pre-Condition: *N/A*
   - Test Condition: if ((`SAFE_CAPACITY` <= (`TOTAL_CAPACITY`) * (`CUTOFF_CAPACITY_DECIMAL`)) && (`SAFE_CAPACITY` > 0)) then *True*, otherwise raise/throw exception.
   - Expected Output: Throw exception and print error (indicating that the safe capacity is too large given the total capacity) (*and possibly allow user to retry, depending on the circumstance*).
   - Post-Condition: *N/A*

7. Test Case:
   - Name: testPrematureExit
   - Description: Demonstrates that the program cannot be terminated while the user is loading up the fuel tank. This prevents several issues like not allowing the user to fill a tank past the specified limit. In addition, preventing the program from terminating while a tank is being loaded prevents extra fuel from being discharged/wasted.
   - Input(s): fuelType = `HIGH_VOLATILITY`, fillAmount = `SAFE_CAPACITY`.
   - Pre-Condition: *N/A*

- Test Condition: While program is in use, it will not terminate. (*In other words, if the program is not done, it will not exit.*)
- Expected Output: Throw exception and print error (indicating that the program cannot be terminated while it is in use). (*However, in the case of an emergency, there should be an emergency shutoff switch. But this should be a hardware killswitch, rather than a software implementation*).
- Post-Condition: *N/A*