SWE2410-111 – Lab 3: Tourist Observer – Jawadul Chowdhury – 9/23/2024

Objectives

State your objectives for the lab here: what is the lab supposed to do in your own words. It should be a short paragraph.

The main purpose of this lab is to implement the observer pattern within the JavaFX application. This means identifying the subject class, concrete subject class, observer class and concrete observer class. The other purpose of the lab is to implement the Art Challenge, where when a user drags a person towards the location of the museum, the artwork is displayed on the JavaFX application.

Requirements

Capture the detailed requirements here. Look at the grading rubrics, the lab assignment instructions to make sure you capture what you must do. Also, put in any requirements that you have added.

The detailed requirement for the lab is listed as follows:

- Ensure that when the person is near the museum, the picture as well as text for the art piece is displayed within the application

Requirements for Implementing the Observer Pattern:

- Ensure that there is a subject class, concrete subject class, observer class, and concrete observer class implemented. Not all the classes are required to be implemented
- Create an interface observer class for the art challenge, where it will have the update() method, and a concrete observer class, which will implement the interface and ensure that when the person is near the museum, the artwork and text is played
- Create a subject interface class, where the observers can be registered, removed and notified. This interface should be implemented by the Museum class, which will act as the subject class and be implementing those methods.
- Ensure that the controller, which is the CityMap class, has no business logic at all, and this mean not implementing any code regarding the challenge.

General Requirements for the Lab:

- Sketch a minimal solution diagram which illustrates the Observer pattern. Ensure to make minimal edits to the existing classes and add responsibilities to new classes.
- How the observer pattern is implemented matters. Identify where the events are generated in the code and making those parts of the code concrete subjects.
- Follow the following constraints, where the challengers are observers, and create a subject as a separate class. The subject and observer can’t be the same class, and ensure the design uses reasonable events.

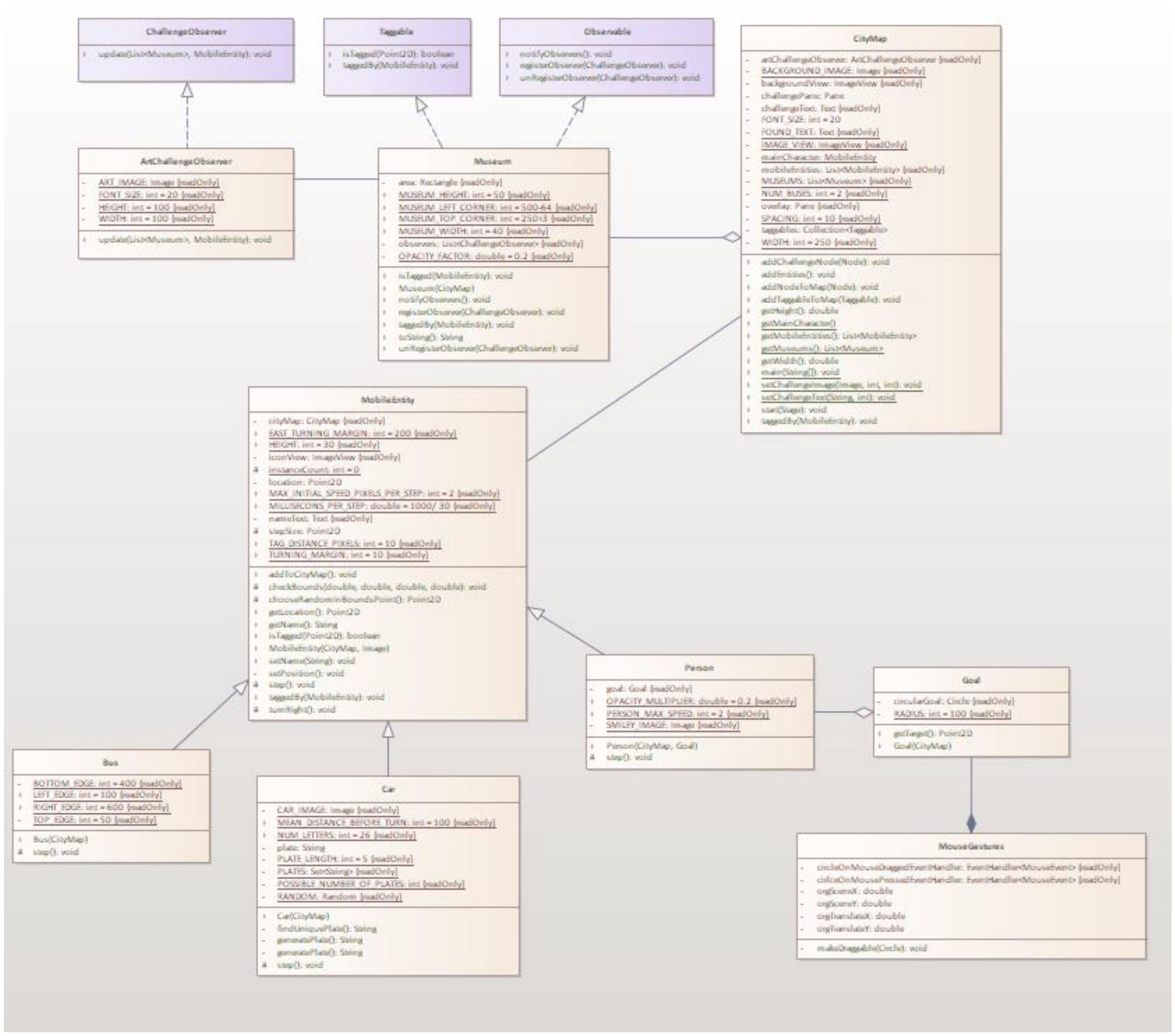
Additional Requirements for the Lab:

- Do not use Java.Util.Observable or Java.Util.Observer. Use your own names for the observer classes. Do not modify existing classes other than to introduce code tying to new feature patterns.
- New GUI code shouldn't be added to the CityMap and add only one line of code to the addEntities() method within the CityMap class.
- Ensure that CityMap.DEBUG_LEVEL is 0 so that any debug code is disabled.

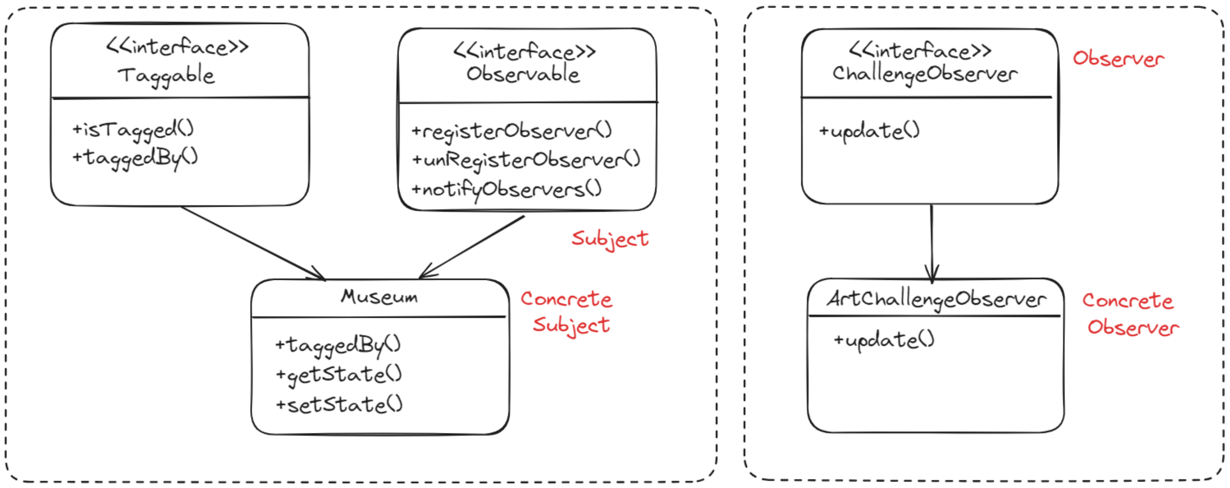
Design

Include appropriate design elements, such as UML diagrams which form the mental model of how it is intended to achieve the objectives. Sequence diagrams are also accepted.

This is a UML diagram of the how the observer pattern was implemented:



This is sketch of the Observer Pattern, and how it is going to be implemented:



Test Run & Results

What tests did you run to support the requirements of the lab? What was your first pass rate. Demonstrate the outcomes of the tests by providing appropriate screenshots and a short explanation of what the test proves.

With regards to the test runs, the testing strategy had changed different throughout development of the Observer pattern. These are the following tests that were performed:

- **Using the City.DEBUG_LEVEL:** the code containing this was set to 0, which allowed for debug messages to be printed out as the images were moving across the application. An experiment print statement was set up to check whether a statement was printed out when the person and the museum was in contact. This was done within the Museum class.

- **Using print statements:** Next, the interface subject class was set up, which contained methods such as registering, unregistering, and notifying the observer. These methods were implemented by the Museum class, which served as the concrete observer class. Print statements were placed.

    - A print statement was placed within the notifyObserver() method to ensure a the method was being called. This method called the update() method.

    - A print statement was placed within the update() method of ArtChallengeObserver, which served a concrete observer and contained the business logic for what the application would display when the person and museum came into contact.

- **The taggedBy() Methods:** to get a better understanding of the difference between taggedBy() method of the CityMap class and taggedBy() method of the Museum class, print statements were placed in both methods. It was these print statements that helped understand where to call the update() method for the artChallengeObserver.

Discussion

Conclusions, including what you learned from the tests that you ran.

Understanding the observer pattern and how to implement it was certainly a challenge. What made it easier to implement was the debug statements, as well as approaching the problem first by simply ensuring that when the person crosses the museum, the artwork and text is displayed.

Once that was figured out, the steps for figuring out the observer pattern was the next challenge, as well removing as business logic from the CityMap class, which served a controller class. The challenge was to identify what the concrete subject class, subject class, observer class and concrete observer class was. Once these were figured out, the next challenge was to figure out where to method calls, and how it should work to ensure there is no coupling of code.

By running the tests, I obtained a better understanding in terms of:

- Ensuring the methods within the observer patterns were working, as well reducing the coupling of code. This was made easier using the print statements.

- How the code generally worked. The debug statements helped to clear things up, as well as understand why the same methods within the CityMap and Museum classes were different.