Flashpoint Lite

Angel Dong, Sam Feig, Elly Landrum

Github Link: https://github.com/AngelD2000/OOAD/tree/master/Proj6RestartReal

1. Final State of System Statement

We've accomplished all of the functionality that we wanted to at the start of project 5. After project 7, we've created a functional (and fun) game to play. Some of the key features to accomplishing this goal are:

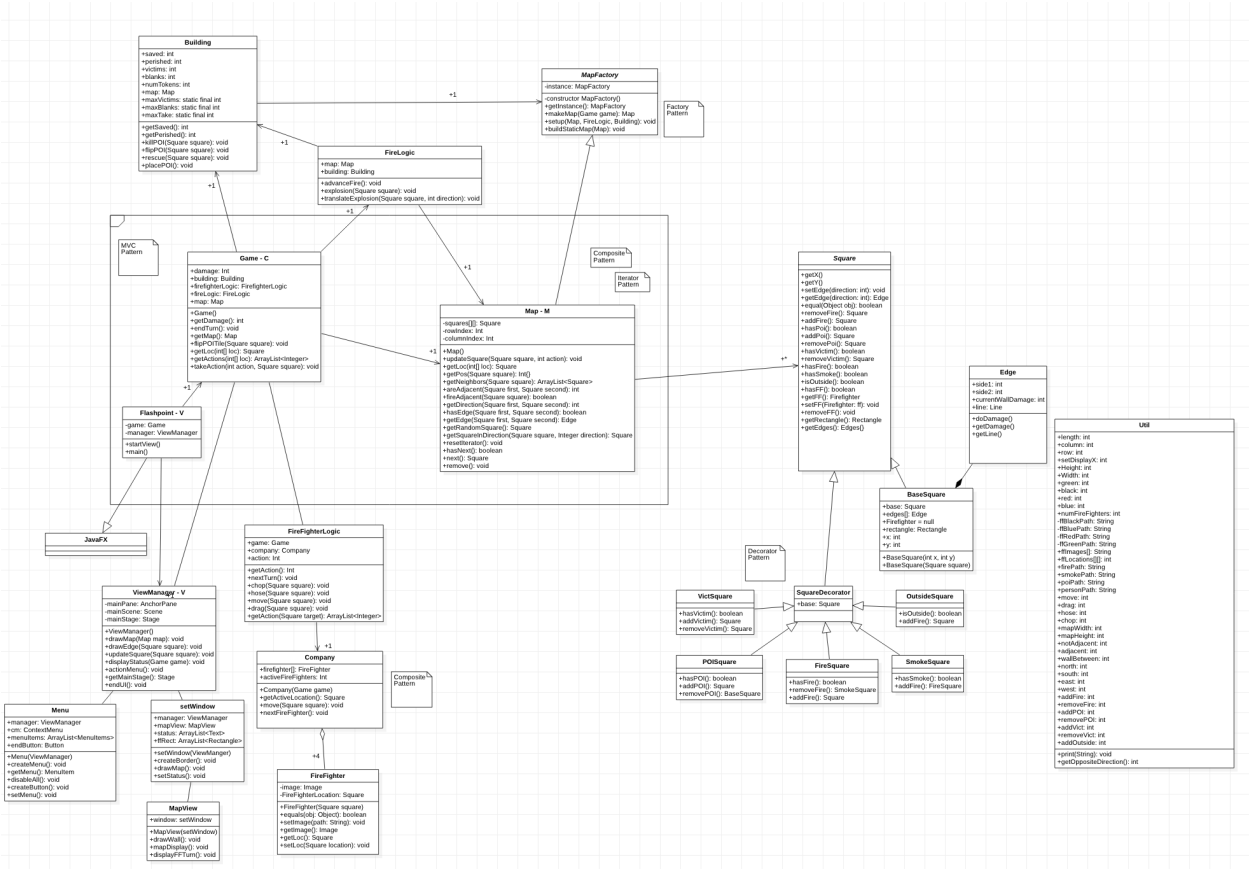
- Displaying a map and some key features through a GUI
- Setting up a map with 4 initial explosions
- Adding shared Edges between squares and handling damage to them
- Allowing user to command Firefighters to hose, move, drag, and chop
- Adding additional fire at the end of the turn
- Handling adding points of interest and revealing them to show victims and blanks
- Saving victims that are outside or killing victims in smoke or fire
- Tracking end of game conditions and displaying to the user when they won or lost
- Lots of bug fixes to logic and display code

If we had more time, we might implement these bonus features:

- Choose initial firefighter locations and turn order
- Add special firefighter classes
- Make board randomly generated
- Movement with arrow keys
- Numbering squares
- Undo button with memento pattern

2. Final Class Diagram and Comparison Statement

Final Class Diagram:



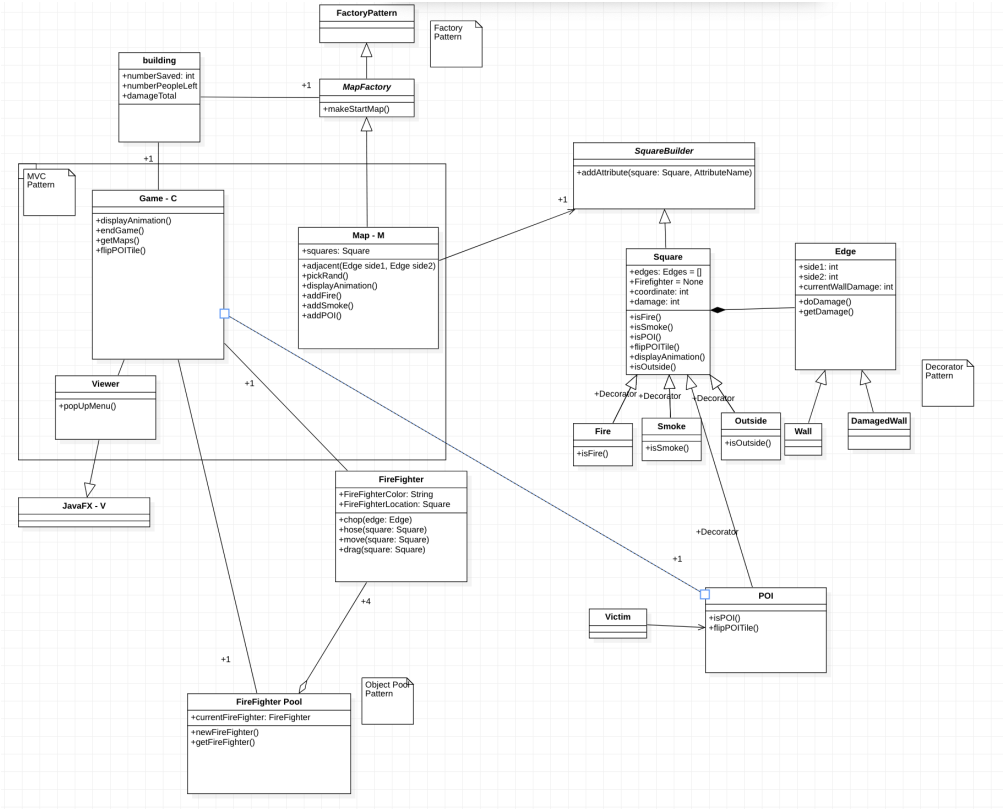
Patterns in Use:

- MVC: MVC describes the general interaction of our classes, and also how we divided the work. It was helpful for our organization to be able to assign Sam to the model (MapFactory, Map, Edge, and all the Squares), Angel to the view (MapView, Menu, setWindow, and ViewManager) and Elly to the controller (Game, Building, FireLogic, FirefighterLogic, Company, and Firefighters).
- Composite: Our design uses 2 different composites: The Map to store all the squares in the game, and the company to organize and interact with the firefighters. This Company was very helpful in the controller, because it allowed many of the actions taken in the game to just be passed to the company which can then route to the active Firefighter.
- Iterator: The Map composite is also an iterator pattern. It was key for both the UI and firelogic to be able to iterate over every square and display, analyze, or change it. By having Map implement Java's Iterator interface, Map can support functions such as hasNext() and next() to iterate over all the squares. The other functions of the Iterator interface (remove() and forEachRemaining()) were left unimplemented throwing exceptions as they are unneeded for the use case. Map

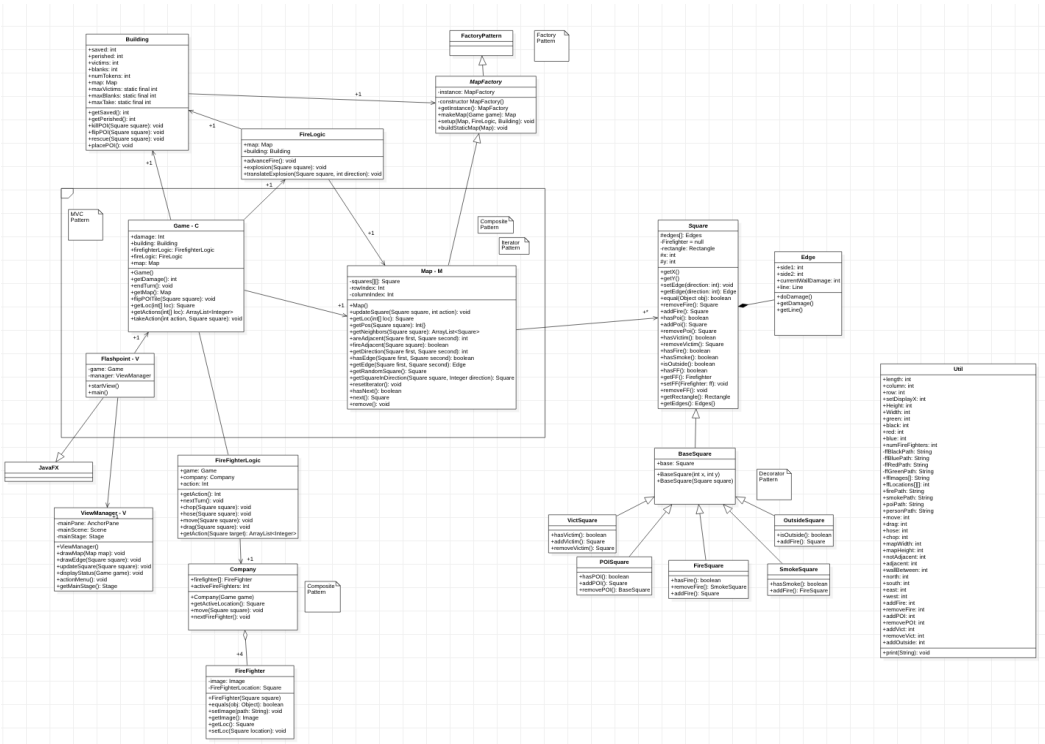
also has a resetIterator() function that resets the iterator when you are done using it.

- Factory: We created a map factory to handle the setup of the game board. Map factory creates the 8x6 map of squares adding outside squares and edges in the specified locations. It calls fire logic to place 4 explosions, calls building to place 3 POI, and then places the firefighters in their designated position.
- Singleton: The map factory also uses the eager singleton pattern as there is no need for more than one factory when creating the static map.
- Decorator: The squares on the board implement a decorator pattern to enable their different behaviors. The Square interface defines all the functions and behaviors a square can have. The BaseSqaure class is the representation of an empty square with nothing on it and implements all the Square functions. SquareDecorator is the decorator that acts as a passthrough, so all the functions of the square interface are passed through to the BaseSqaure that was used to create the SquareDecorator. From there, all the specialized squares (Fire, Smoke, Outside, POI, Victim) extend the SquareDecorator and override only the methods they need to change the behavior of the square letting all other implementations pass through to the BaseSquare's versions. For instance, in a FireSquare, hasFire() will return true and removeFire() will create a new Smoke decorated square when the default behavior in BaseSquare is to return false and return a null square.

Project 5 Class Diagram:



Project 6 Class Diagram:



Key Changes since Project 5 and Project 6:

- Changes since 5
    - Additional functions added to at least every class to handle some of the specific functionality needed and missed in initial design.
    - Functionality that was originally just placed in the Game object has been moved between FireLogic and FirefighterLogic for better cohesion/one class = one responsibility.
    - Firefighter Pool renamed to Company, and is instead a composite of firefighters.
    - Made firefighter a type of decorator on the square, so firefighters, fire, POI, any object that is on a square will be a decorator for the frontend to replace the background of the square.
    - Both Square and Edge need a Javafx object to display. Rectangle and Line class are the two I was using.
- Changes since project 6
    - Added Menu class for creating objects that interact with users. This implemented the menu for each square. We also added the “End Turn” button to allow the player to save actions
    - Added setWindow class to set the initial window, status, and firefighter turns before the game is played
    - Added MapView class to draw the initial map given by Map class, namely the squares and edges, locations of firefighters, location of fire, smoke and POIs.
    - Displaying a different screen at the end of the game. This screen is also different for if it was a loss due to victims dieing, building collapse, or victory.
    - Square Decorators and logic was fleshed out and base squares were added

3. Third-Party code vs. original code Statement

On the whole, the only third party code we used was the JavaFX to create the GUI for the game. This was very helpful in allowing us to create a UI. It would have taken us much longer to make ourselves. In partnership with JavaFX usage, we also had to use a few tutorials to understand how to use the library. This included youtube tutorials (such as https://youtu.be/DkIuA5ZEZ_U) on how JavaFX is set up and used. It also shows how the UI code looked for that particular game created using JavaFX. After looking through that tutorial, I then created a simple version of JavaFX and tested how that worked before creating the actual UI of our program. This tutorial code was not included in our final project, but it was very helpful in building understanding.

4. Statement on the OOAD process for your overall Semester Project

One obvious part of our design process was the UML and designing that we did in project 5. We found that actually implementing the code was mostly straightforward once we mostly had a design in place. It was easy to see what functions and abilities each of the classes would need. It was especially helpful in the context of group work. As a group, we didn't have any code or logic duplicated between our parts, because we each had a high level idea of what all the classes were doing. The model shouldn't have logic about spreading fire, and the game shouldn't worry about the logic of adding fire to an outside square. It made it much easier to work as a group to create finished code.

The biggest issue we faced in our OO design was trying to avoid creating a god class. As you can see from our diagram in project 5, it was implied that the Game class handled a lot of different logic for the game. Furthermore, it also knew about a lot of the other classes in the game. When we sat down to implement this class, we realized that it would work a lot better as 3 different classes. Even with this broken out, we were still careful to try to limit how much the Game class knew about all the other classes.

The last issue we faced is once we integrated our parts together a lot of errors. Some of these errors popped up because we didn't do enough incremental tests throughout the creation of the program. Others occurred because of issues we faced working with Maven. Maven made our lives easier by integrating JavaFX and ensuring we were all working on the same versions of the libraries; however, it also meant that we were using the same versions of the libraries across disparate operating systems. When we pushed to git, it pushed the specific library files Maven downloaded for our OS and so caused errors that took a while to diagnose. Eventually we learned that you need to clean and then rebuild the Maven project after pulling from git to ensure it gets the proper versions of the libraries and builds the classes specifically for the system we are each on. Knowing this, along with a bit better testing throughout the project would have helped avoid the amount of errors and bugs we had to fix while combining our parts together towards the end of the project.