Assignment 3

Exercise 1: Natural language descriptions

Observer-pattern

Why an observer-pattern?
Our game currently has an update cycle that runs X amount of times per second in a different thread from the UI. In order for the UI to properly get information from the game without redundantly checking, we use an observer pattern. Not only the UI, but any future components (multiplayer, sound) would be able to hook into this system as well. This also means that if we decided to switch from JavaFX to something else, our game code would not require any changes.

How is the observer-pattern implemented?
After creating the game the gameController (UI) registers itself as an observer to the game. Whenever something important happens in the game, such as a fired ball moving or bubbles getting destroyed. The game will notify all observers with a specific update type, after which the observers can retrieve the relevant information from the game based on their specific needs.
Of course as per an observer pattern this means all the observers have an update method which is called by the game, and the game maintains a list of all observers, together with methods to add and remove to this list.
For our implementation we are specifically using the observer pattern to communicate to the "outside" such as the previously mentioned UI, sound and possibly multiplayer/internet connection. For internal communication between components we are not (yet) using the observer pattern.

Factory pattern

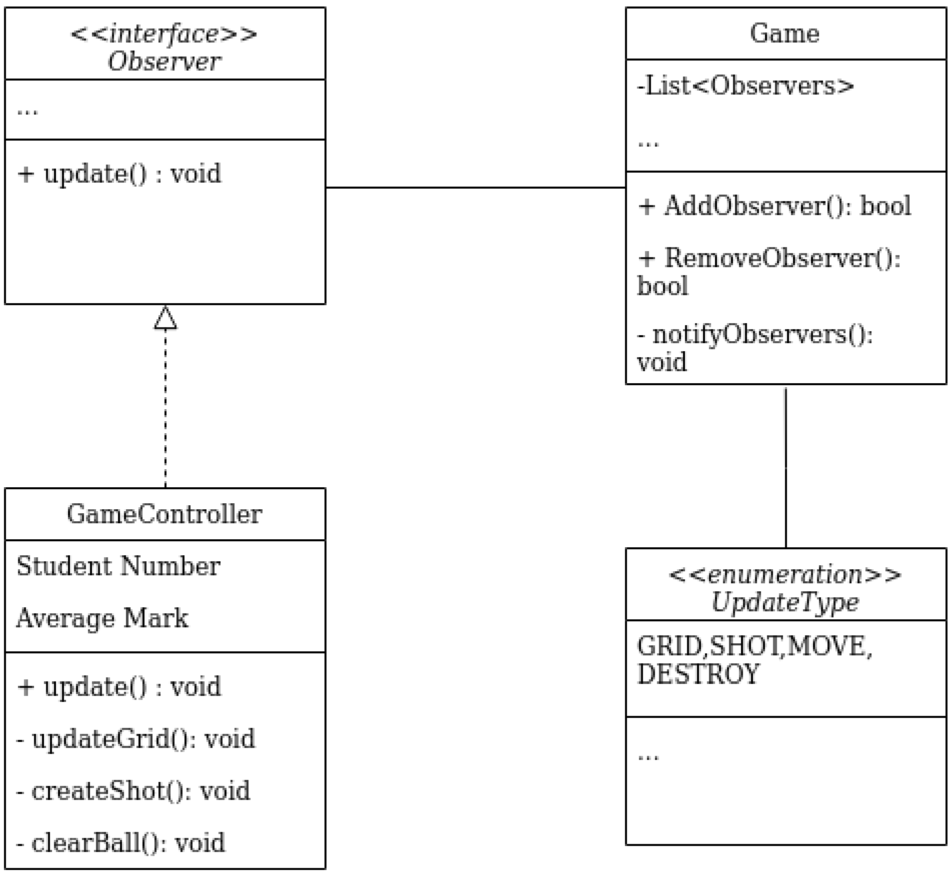
Why a factory pattern?
We chose a factory pattern to abstract the creation of bubbles away from the game and grid. At the moment we only have gridBubbles, shotBubbles and centerBubbles. Therefore the abstraction is not necessary yet. However, when we want to add more types of bubbles in the future, for example exploding bubbles, this will be easier when the creation of the bubbles is abstracted away from the game and grid. This is, because when we want to (possibly randomly) create bubbles at the start of the game, the factory will be able to take care of this instead of the game and the grid having to do everything.

How is the factory pattern implemented?
At this moment, we have two different factory patterns. One for the shotBubbles and one for the gridBubbles. In the future we want to make one abstract factory pattern out of this.

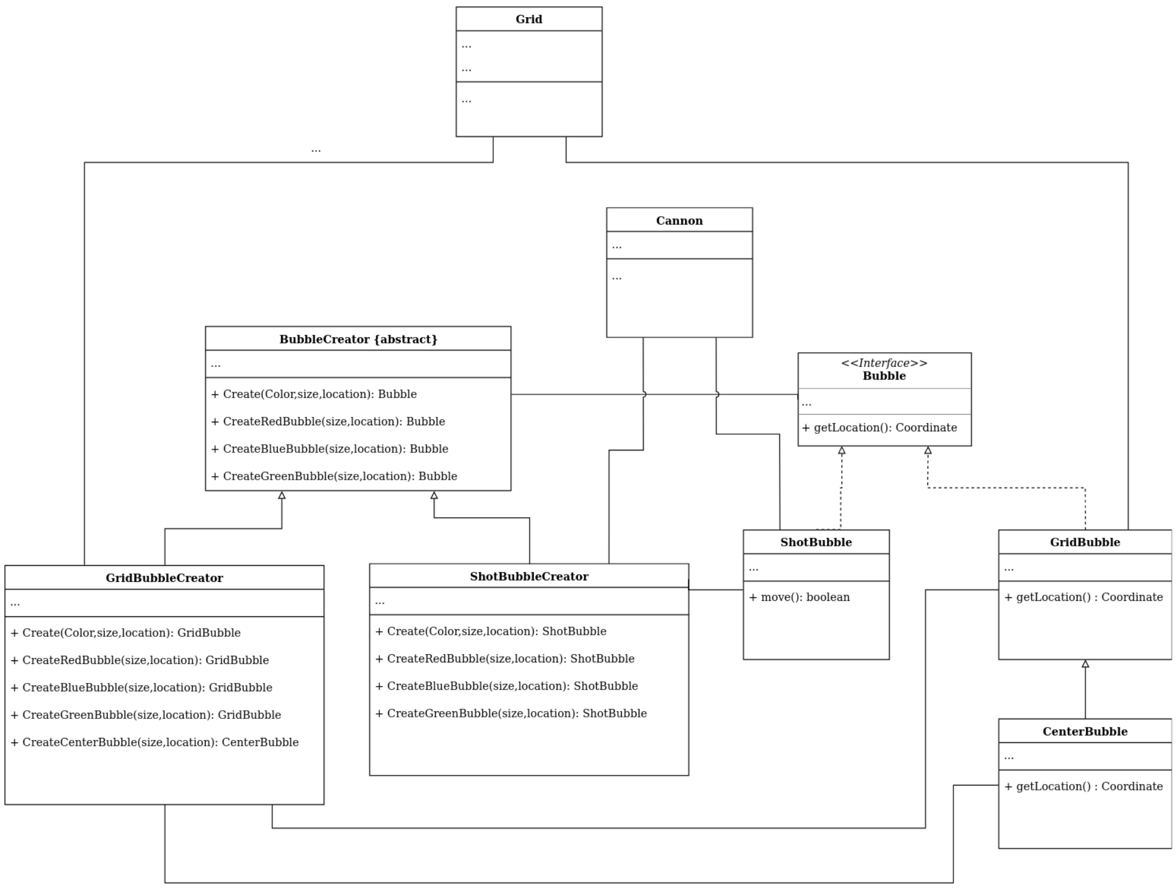
For now, the Grid points to the concrete GridBubbleCreator and the Cannon points to the concrete ShotBubbleCreator. The GridBubbleCreator creates gridBubbles. The shotBubbleCreator creates shotBubbles.
The centerBubble is an example of the before mentioned different types of bubbles. It inherits from the gridBubble, because it is a more strict type of gridBubble. It can only be in the center of the grid and the grid spins around it.
In the future, we want to change this, such that both the Grid and the Cannon point to the abstract BubbleCreator. From there the abstract BubbleCreator will do the everything.

2: Class Diagrams

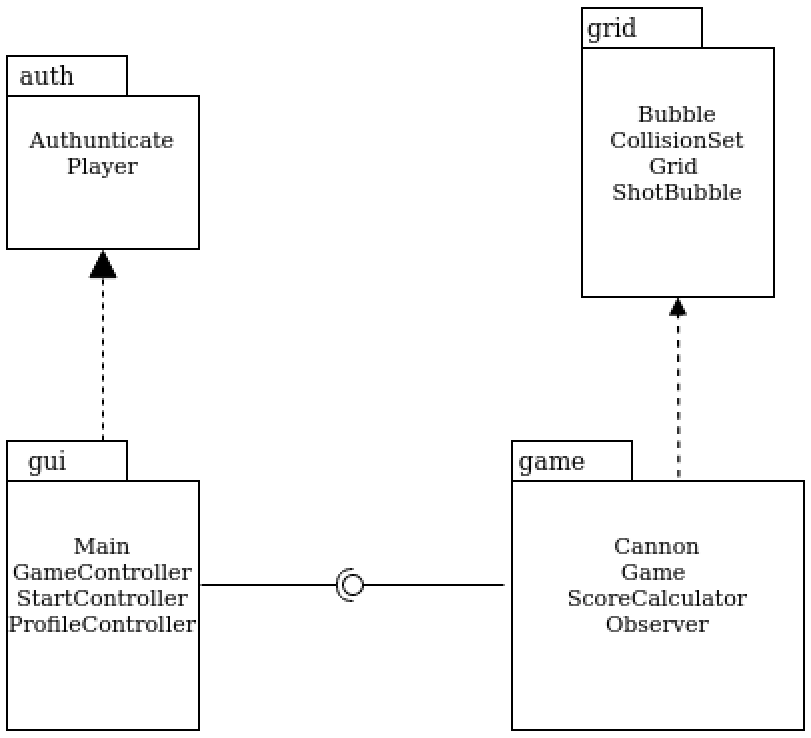
Observer pattern:



Factory Pattern:


Grid
...
...
...
...
Cannon
...
...
BubbleCreator {abstract}
...
+ Create(Color,size,location): Bubble
+ CreateRedBubble(size,location): Bubble
+ CreateBlueBubble(size,location): Bubble
+ CreateGreenBubble(size,location): Bubble
<<Interface>>
Bubble
...
+ getLocation(): Coordinate
GridBubbleCreator
...
+ Create(Color,size,location): GridBubble
+ CreateRedBubble(size,location): GridBubble
+ CreateBlueBubble(size,location): GridBubble
+ CreateGreenBubble(size,location): GridBubble
+ CreateCenterBubble(size,location): CenterBubble
ShotBubbleCreator
...
+ Create(Color,size,location): ShotBubble
+ CreateRedBubble(size,location): ShotBubble
+ CreateBlueBubble(size,location): ShotBubble
+ CreateGreenBubble(size,location): ShotBubble
ShotBubble
...
+ move(): boolean
GridBubble
...
+ getLocation() : Coordinate
CenterBubble
...
+ getLocation() : Coordinate

Exercise 2: Software Architecture



Game

The game is responsible for the main “loop”, each frame the game will “tell” the other components on the logic side what to do; It will rotate the grid, move the shot bubble, check for collisions and tell the UI to update. The game is also responsible for handling the “state” of the game, whether it is paused or playing, and what to communicate/do when the game is over.

Grid

The grid is responsible for keeping all of the bubbles “in check”. It handles collision checking between the shot bubble and all bubbles in the grid, calculates what the world position of bubbles in the grid should be, and handle adding and removing bubbles as required by the game logic, such as removing bubbles that are groups of three or more, or removing bubbles that have been disconnected from the center.
Simply the grid is the logical part of the game, where most of the logic has been implemented, which keeps the game controlled and check after every event.

Auth

The authorization is responsible for the connection to the database for authenticating user log in.
This component is also used to verify whether a valid login information has been provided, if so, the user will be granted access to the game. Otherwise the user must check his/er information and try again.

Gui

The graphical user interface (gui) is responsible for all of the interaction with the user. It takes care of taking in user input such as credentials, pause/play functionality and taking in mouse input, and displaying results, such as fired bubbles, destroyed bubbles, (un)successful user authentication back to the user.

General architecture

The most important part of the architecture is the disconnect between the UI and the game plus grid. This has been achieved with an observer pattern as discussed in exercise 1. The advantage of this is that the system is easily expanded on. If/When we choose to add audio, or even multiplayer/internet access to the game, all that is required is another observer that will listen to the game, and perform actions when told, without having to make any changes to the game. Because of the two way action between the UI and the game this part of the code is based on a model-view-controller, the game+grid is the model, the Views get updated as observers, and the UI is both viewer and controller.

Furthermore, because of the disconnect between the UI and the game, it also means that if for some reason we would choose to no longer use javaFX, we would also not require to make any changes to the game/logic part in order to use a different system to visualize the game.

Lastly there is a clear distinction between the game and the grid components, where the game is responsible for the “loop” of the game, as well as handling user input passed along by the UI, and making sure all the listeners are updated.
The job of the grid component is to apply core logic such as the collisions and updating of the grid and adding/removing bubbles.