Assignment 2 Bust-a-Move

By
Calvin Nhieu
Jason Xie
Winer Bao
Justin Segond
Maurice Willemsen

TI2206 Software Engineering Methods

Supervisor:
Dr. A. Bacchelli

Teaching Assistants:
F. C. A. Abcouwer
R. van Bekkum
M. M. Beller
T. Boumans
A. Ferouge
J.E.Giesenberg

Exercise 1

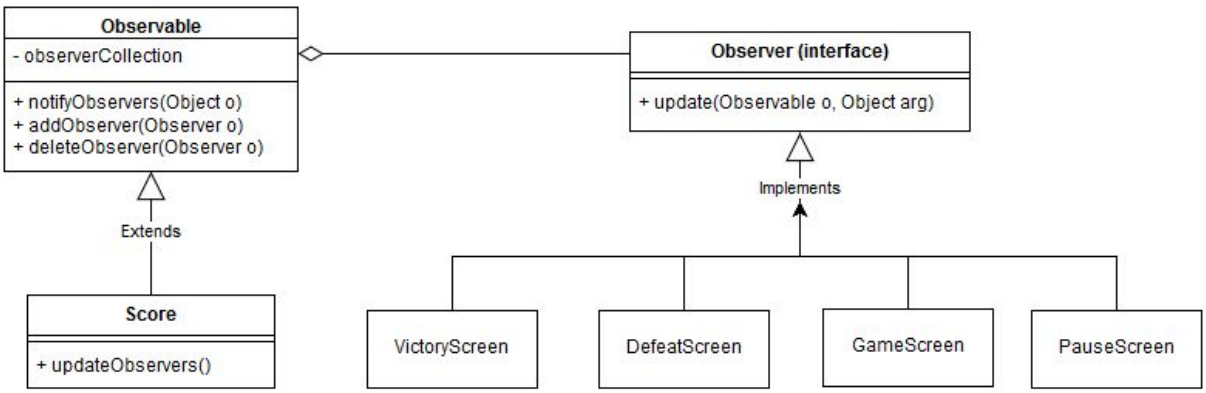
Observer pattern:

The first pattern that was implemented was the observer pattern.

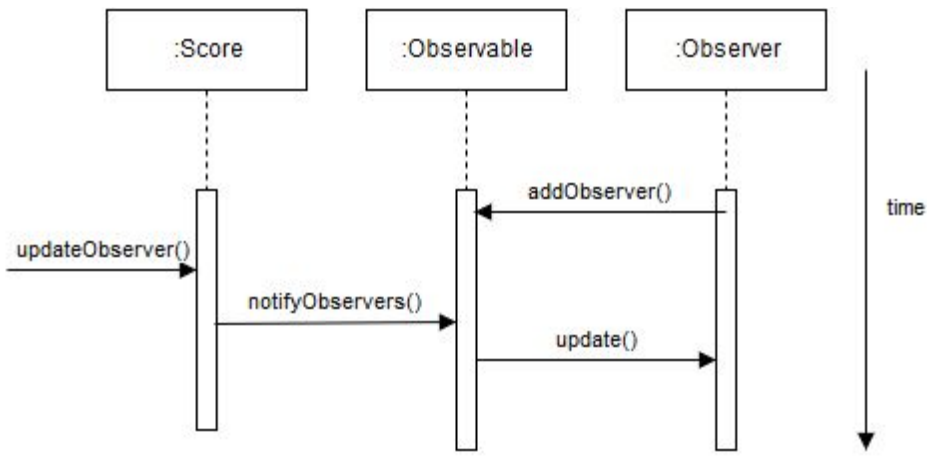
The observer pattern will be used to update a score on different screens. The screens needed a way to be kept up to date all time on the current score. This could be easily achieved by the observer pattern. As the observable object sends a message to all the observers about an update of the current object. And the screens then update the score they render.

The pattern is implemented as follows: The Score class is the observable object. And the different screens classes are the observers. The screens will first register to the score class. The score class then saves a collection of all its current screens that are registered. When the score gets updated in the score class, the score class sends a message to all the observers (screens) that the score has changed. The screens will update their score. The screens will then render the new score.

The pattern is further explained in the next class diagram.



And also in a sequence diagram.



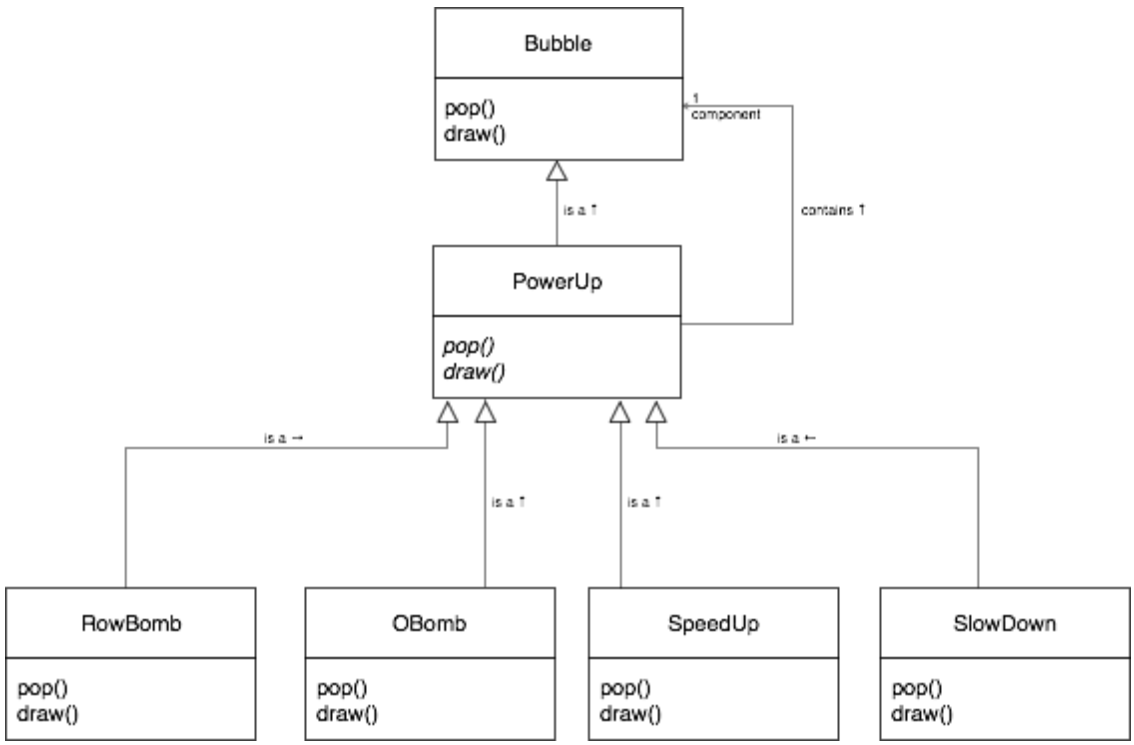
Exercise 1 pt2, Exercise 2

Decorator Pattern:

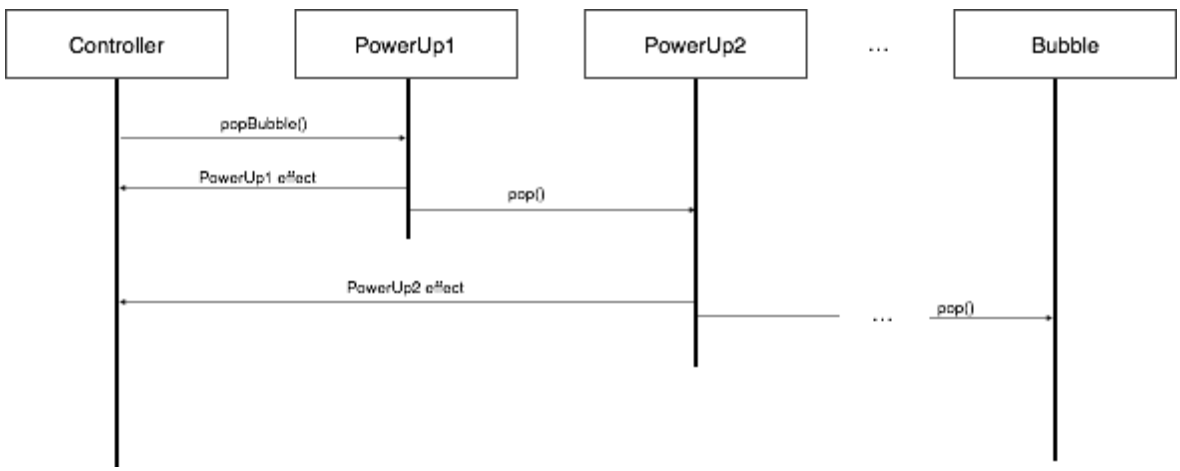
The decorator pattern will be leveraged to implement a power-up feature to bubbles in the Bust-A-Move game. A power-up is an item that is attached to a bubble, and grants the user with a special effect when popped. There may be many different types of power-ups and the flexibility to apply one or several power-ups to a single bubble is required. The decorator pattern is suitable for this implementation since it enables an object to dynamically take on new responsibilities. In this case, the pattern will be used to enhance the functionality of a bubble with the feature of a (or multiple) power-up(s).

In particular, power-ups provide additional functionality on pop of a bubble. On a high level, this will be implemented by wrapping vanilla bubbles inside of a power-up object. Further, power-up objects will be treated as bubbles themselves. This creates a recursive hierarchy that enables both power-ups and bubbles to be used synonymously as bubbles, but providing the option to enhance a bubble with a power-up. In technical terms, a power-up will be a subtype of a bubble and will be instantiated by first creating a vanilla bubble, and then initializing a power-up with that bubble. The difference is the power-up instance will override bubble's pop (and possibly draw) method to implement additional functionality on top of bubble's default implementation of pop. For example, a power-up, on pop, may increase the speed of bubbles shot for the next 5 shots. The result is a class hierarchy of bubbles with varying and flexible pop implementations.

Decorator pattern class diagram



Decorator pattern sequence diagram



Bust-A-Move PowerUp Requirements

Must haves:

• The game cannon and arena must generate with 3% chance a bubble with a bomb power-up that, when popped, will also pop all bubbles in its row.

• The game cannon and arena must generate with 3% chance a bubble with a bomb power-up that, when popped, will also pop all neighboring bubbles.

• The game cannon and arena must generate with 10% chance a bubble with a speed up power-up that will increase the speed of bubbles fired from the cannon by a factor of 1.5.

• The game cannon and arena must generate with 10% chance a bubble with a speed down power-up that will decrease the speed of bubbles fired from the cannon by a factor of 1.5.

Could haves:

• The game cannon should generate with 2% chance a rainbow bubble that is able to complete a connection of any color.

Exercise 3

3.1 Requirements for code restructuring/ quality improvement.

Must haves

- We need to put all bubble storage related functionality in it's separate class.
- We need to move the collision detection into its separate class as well/
- Use Point class instead of x and y variables
- Create a Screen interface.
- Proper javadocs.
- New and restructured code follows the coding rules of checkstyle, PMD and FindBugs.
- 75% test branch coverage in the Bubble, Cannon and BubbleStorage classes.
- The Score class must extend the Observable class.
- The VictoryScreen is an Observer of a Score object.
- The DefeatScreen is an Observer of a Score object.
- The PauseScreen is an Observer of a Score object.
- The Game is an Observer of a Score object.
- Rename packages to a shorter name by removing bust_a_move20162017.bust_a_move_framework prefix.
- Reduce the amount of packages by placing closely related classes together in a package.
  Arena class -> Game package
  Cannon class -> Game package
- Create class diagram of the restructured classes.

Should haves

- No checkstyle, PMD and FindBugs violations

Could haves

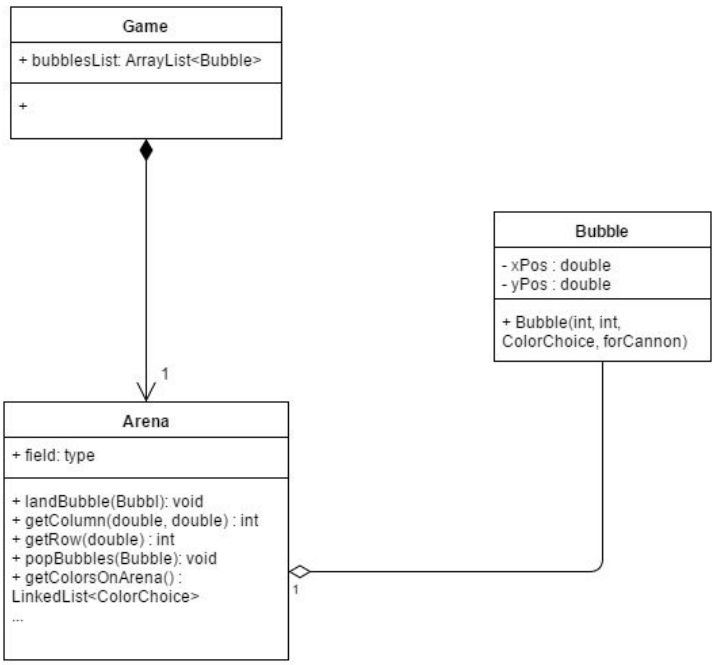
- Split the model from the screens/game states.

Would/won't haves

- ----------

3.2

Before refactoring



After refactoring