Bubble Beam - Assignment 3
Group-12

Bavdaz, Luka 4228561
Clark, Liam 4303423
Gmelig Meyling, Jan-Willem 4305167
Hoek, Leon 4021606
Smulders, Sam 4225007

October 12, 2014

1 Game extension

The analysis and design phase document is located on our devhub repository on the master branch, under "report/Assignment3 - Analysis and design phase document.pdf".

2 Design patterns

2.1 Decorator

2.1.1 Natural language description

The decorator design pattern was used to implement the powerups of the bubbles that are shot. Using a decorator for this enables a bubble to have multiple powerups without having to add new classes. Furthermore, making changes to existing powerups will not force us to change code oustide of those specific powerup classes.

We added the DecoratedBubble interface which extends the Bubble interface, so every method is availeble for each concrete decorator class. The concrete decorator classes include all powerups, such as bomb bubbles and joker bubbles. Every concrete decorator passes the called method to its next wrapped object, so when a method is called, all the concrete decorators are called and finally the concrete component is called. Our concrete components are ColouredBubble and AbstractBubble.

2.1.2 Class diagram

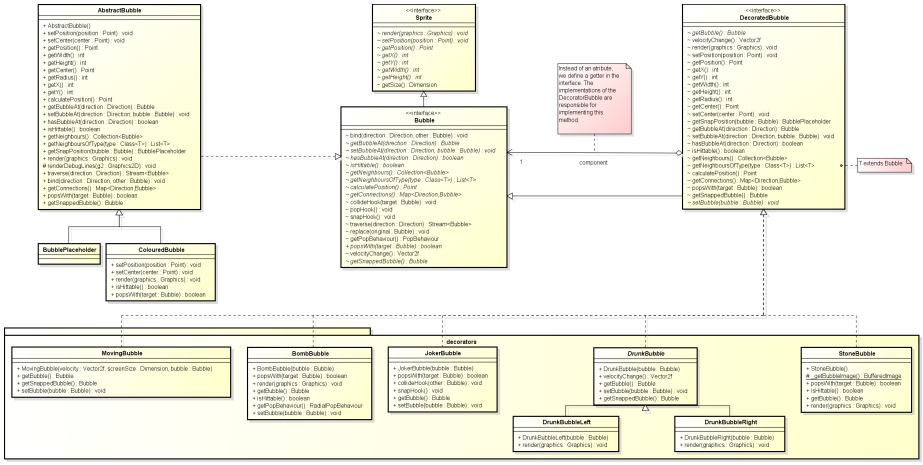

Figure 1: The class diagram for the implemented decorator pattern

2.1.3 Sequence diagram

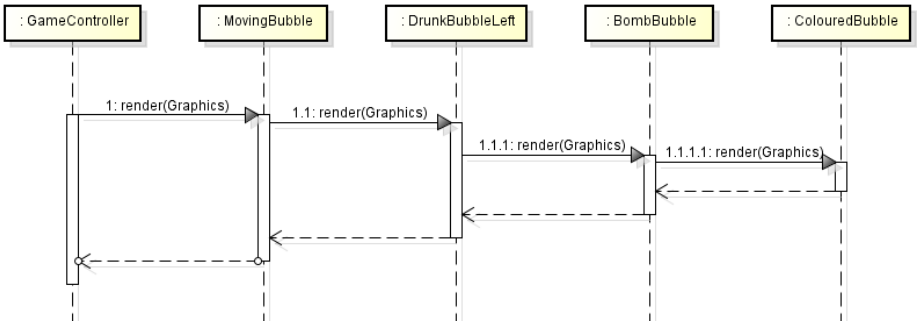

Figure 2: A sequence diagram for the implemented decorator pattern

All the other methods in the decorator behave in a similar way. Each method will also access the same method in the wrapped object. Some methods will only perform this method call (like MovingBubble and JokerBubble in figure 2), while others will also do more (for example, the render method in DrunkBubbleLeft will also render arrows). This works the same for all combinations of concrete decorators.

2.2 Strategy

2.2.1 Natural language description

We used the strategy pattern for the pop behaviours of bubbles. Using the Strategy patern makes it possible to add new behaviour without changing other classes. The reason for implementing this pattern was to make the program

more flexible, because the behaviours are loosly coupled. This means we can easily add new pop behaviours. This also enables us to give diffrent decorators diffrent pop behaviours.

A PopBehaviour interface was introduced, as displayed in 3, which is implemented in the subclasses, each respresenting a type of different behaviour. For example; the BombBubble pops all nearby bubbles, regardless of their colours. BombBubbles thus have the RadicalPopBehaviour subclass of PopBehaviour. While for example the normal bubbles should have different popping behaviour. Here we want to look at the colour of the adjacent bubbles, and we only want to pop them if they are equal in colour and in a group of 3 or more, for which the RecursivePopBehaviour subclass of PopBehaviour is used.

2.2.2 Class diagram

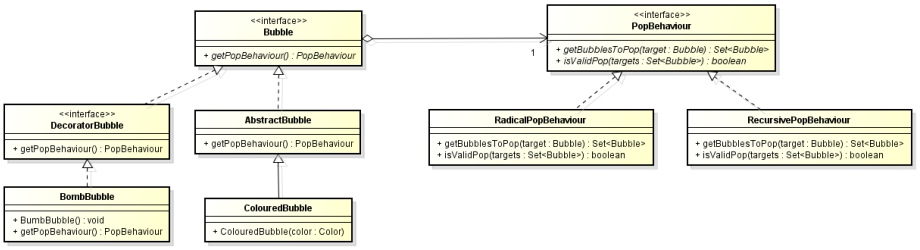

Figure 3: The class diagram for the implemented strategy pattern

2.2.3 Sequence diagram

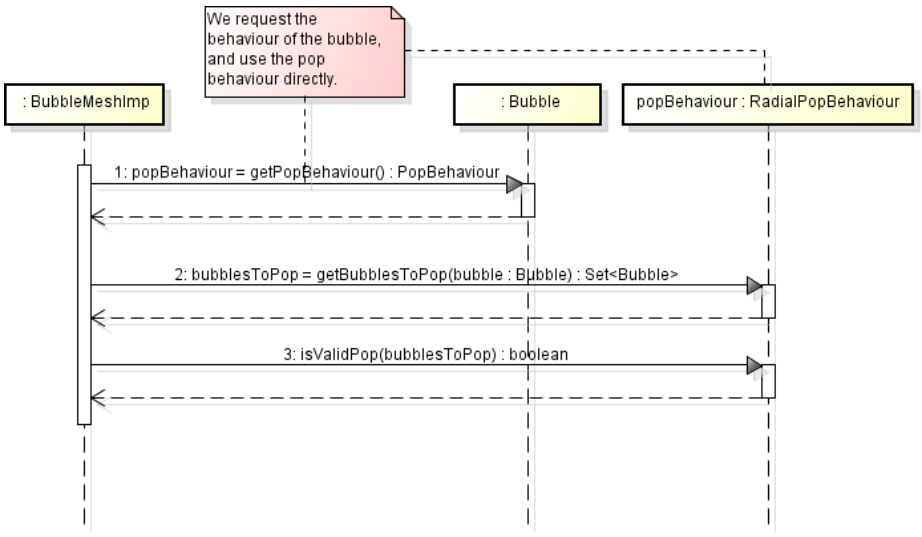

Figure 4: The sequence diagram of the RadialPopBehaviour

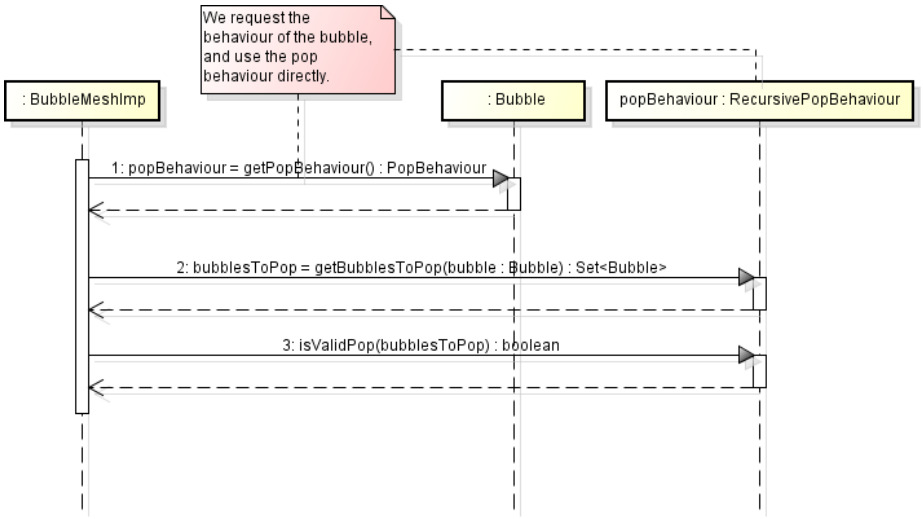

Figure 5: The sequence diagram of the RecursivePopBehaviour

For clarification, the return values are named so it is more clear when they are used in successive method calls.

3 One more design pattern

3.1 Model view controller

3.1.1 Natural language description

At the first stages of the development of our game back-end model logic and gui logic was heavily intertwined and dependent on each other. We set out to try and get better encapsulation for both through the use of the ModelViewController pattern. However in our implementation there are two MVC's working together we have a higher level one that controlls the Game. Which relies on another MVC that handles the Cannon. This does tighten the coulpling between the two but in the end they are still quite loosely copoupled. The view however doesn't update through observer but it simply requests the models state every cycle. The model however does have observer and it can notify the higherlevel game and changes for multiplayer over the socket.

In them implementation we quickly set up 2 of each models, controllers and views. The views come together in the gamePanel where they both draw in and the listeners are bound between the panel and them. (note: it doesn't appear in the uml because it doesn't affect the structure of the mvc). Because the interfaces needed to be used in two cases they are parameterized interfaces so they could be reused. Furthermore the view <C,M> interface binds it all together through the get model and getController methods (the relationships for the views are actually achieved through this). All in all this gives us a loosely coupled flexible design.

3.1.2 Class diagram

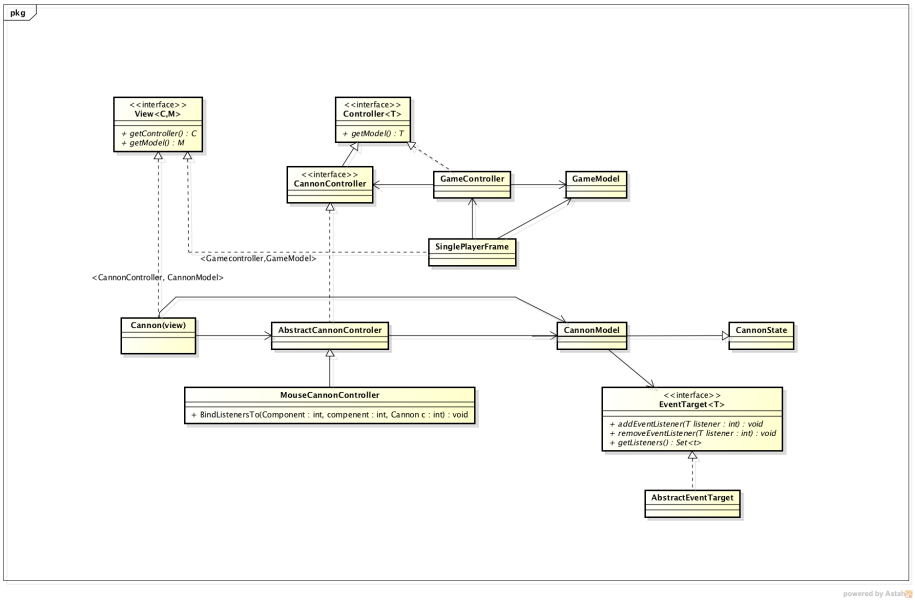

Figure 6: The MVC class diagram

3.1.3 Sequence diagram

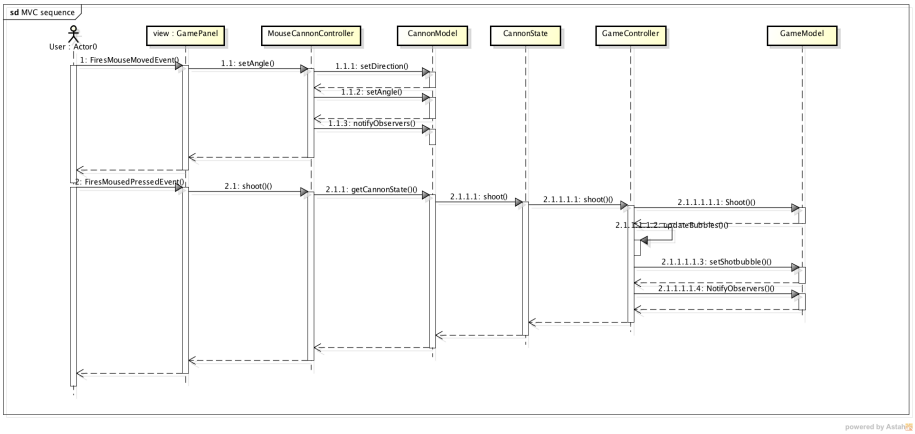

Figure 7: The MVC sequence diagram