Bubble Beam - Assignment 2

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

September 27, 2014

1 Simple Logging

1.1 Logger implementation

The logger is successfully implemented and committed.

1.2 Analysis and design phases

The documents used during the analysis and design phases of the logger can be found in the folder Assignment_2_logger_analysis_desing within the report folder.

2 Design patterns

2.1 Iterator

We used the iterator pattern to go over all the bubbles in a BubbleMesh. This is for example used to draw all the bubbles in a BubbleMesh.

We added methods for BubbleMesh to create an iterator for the BubbleMesh. These methods each create a new instances of the BubbleMeshIterator, which creates a list of all bubbles of a row in the BubbleMesh that require iterating. To iterate through the entire mesh, multiple iterators are used. Our BubbleMeshIterator has a method to remove the next Bubble on the list and return it and has a method to tell if the list is empty or is not.

2.1.1 Natural language description

2.1.2 Class diagram

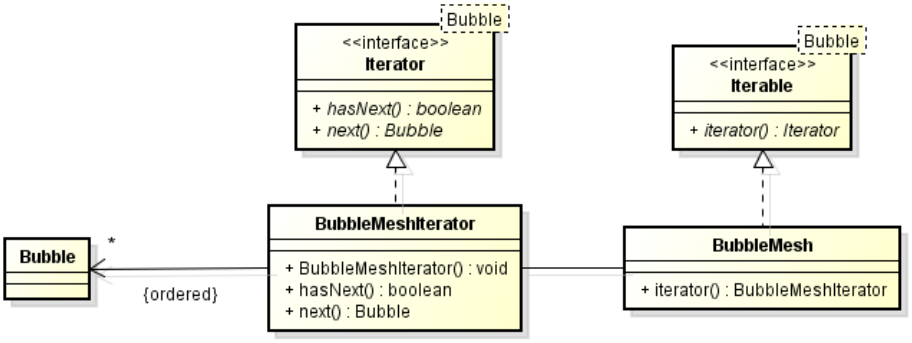

Figure 1: The class diagram for the implemented iterator pattern

2.1.3 Sequence diagram

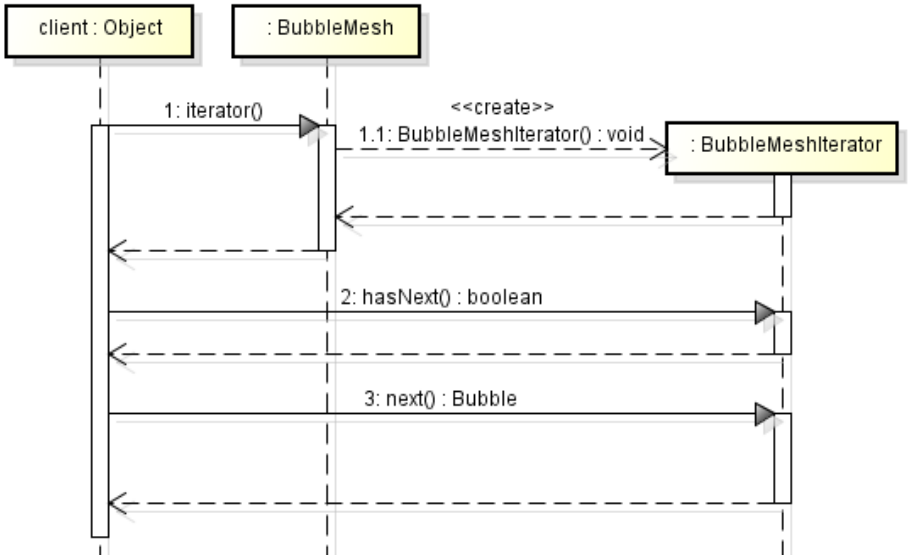

Figure 2: The sequence diagram for the implemented iterator pattern

2.2 Observer

2.2.1 Description of the Observer Pattern

We needed a nice way to encapsulate the GUI code and the logic behind our model and at the same time make the code more testable. We decided to make the Cannon class simply draw the cannon according to a model.

We decided on the workflow displayed in figure 4. The Cannon and the Controller are very loosely coupled through the model. The Cannon draw ac-

cording to the data from the model and doesn't need any logic for controlling it. This also makes testing the class easier since it now has just a single behaviour (drawing) instead of multiple behaviours of drawing and controlling.

We ran into trouble when the model needed to be able to shoot. This is an action and that can't be done through just observer you need an ActionListener for this. For that purpose in the AbstractCannonController listeners are bound to the model slightly tightening the coupling between them. This isn't perfect but it will work for now and it's still a great we eventually implemented it following the model in figure 3.

2.2.2 Class diagram

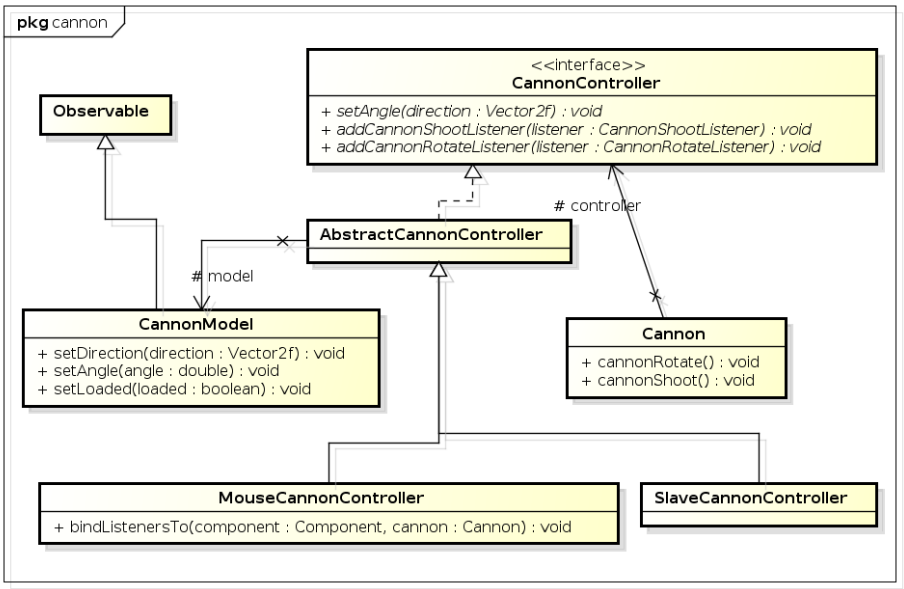

Figure 3: The class diagram for the implemented observer pattern

2.2.3 Sequence diagram

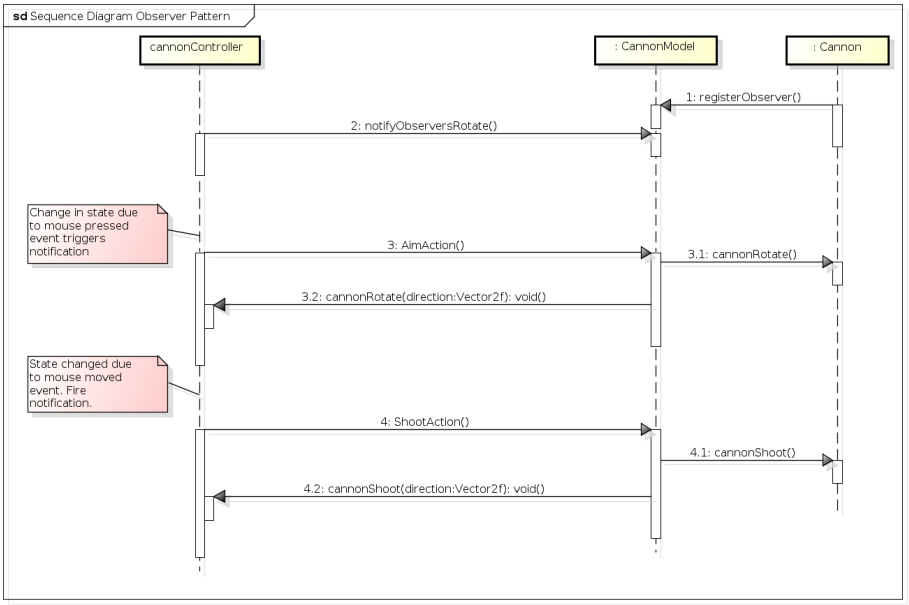

Figure 4: The sequence diagram for the implemented observer pattern

2.3 Strategy pattern

2.3.1 Natural language description

Using the Stratagy 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. For example, we could implement an teleportation bubble that teleports the first movingBubbles colliding with it, after which the behaviour changes. To achieve this, a new behaviour for teleporting has to be added.

A SnapBehaviour interface was introduced, as displayed in figure 5, which is implemented in the subclasses, each representing a type of different behaviour. In this case, the SnapToClosest algorithm returns the closest PlaceholderBubble using a search method, while the SnapToSelf returns itself. The AbstractBubble has an instance of the behaviour, which is set in the constructor depending on the subclass.

The SnapClosest algorithm is implemented by the ColouredBubble class, since it takes up a bubble spot where a new bubble cannot simply be placed without removing a bubble. The SnapToClosest algorithm will also be used by every new class at which a new bubble arrives but cannot stay. The SnapToSelf algorithm is used by the BubblePlaceholder class, because a newly arriving bubble can take its place.

2.3.2 Class diagram

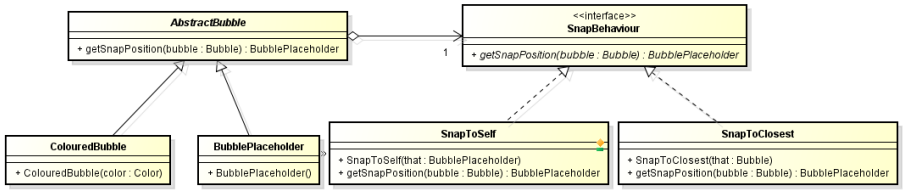

Figure 5: The class diagram for the implemented strategy pattern

2.3.3 Sequence diagrams

The sequence diagrams 6 and 7 show how the pattern works dynamically in our code.

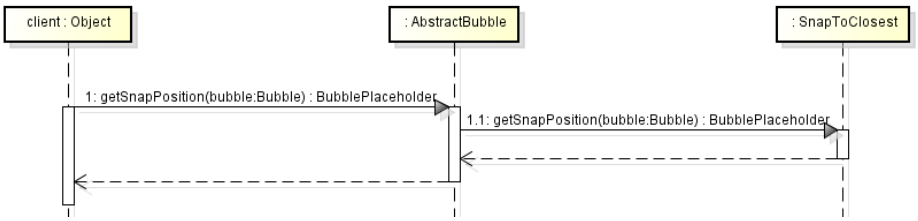

Figure 6: The sequence diagram for the implemented strategy pattern for the SnapToClosest behaviour

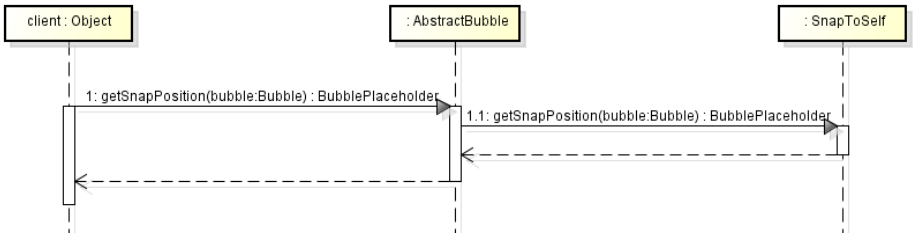

Figure 7: The sequence diagram for the implemented strategy pattern for the SnapToSelf behaviour

3 State pattern

3.1 Natural language description

The state design pattern was implemented to remove the conditional statements regarding the possibility of shooting and make the code more open to extension.

For every cannon state there is a seperate class that implements the Cannon-State interface. The cannon state of a cannon is tracked by a CannonController. This CannonController calls the shoot method, every time the CanonController

wants to shoot. The CannonState determines if shooting actually occurs. If. tI dthe cannoesn controller uses shoot on a CannonLoadedState, the CannonState notifys all the listeners of the CannonController. If the cannon is in the CannonShootState, the cannon can't shoot, and has to wait till the CannonState is set back to the CannonLoadedState. This is done by the GameController, once a moving bubble hits a bubble from the bubble mesh.

3.2 Class diagram

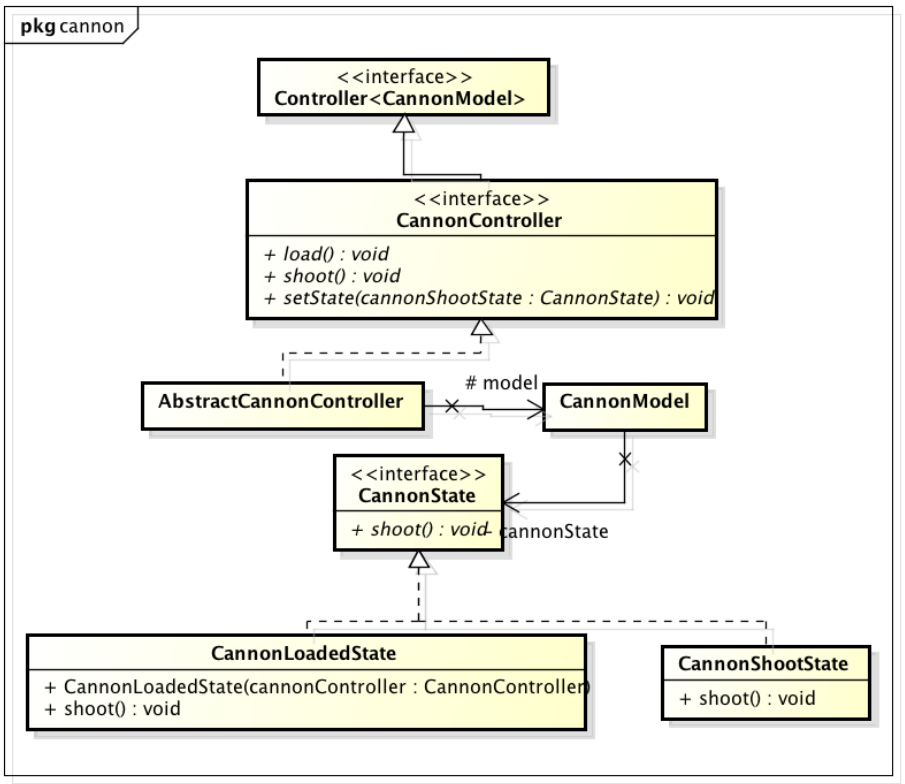

Figure 8: The class diagram for the implemented state pattern for the Cannonload behaviour

3.3 Sequence diagrams

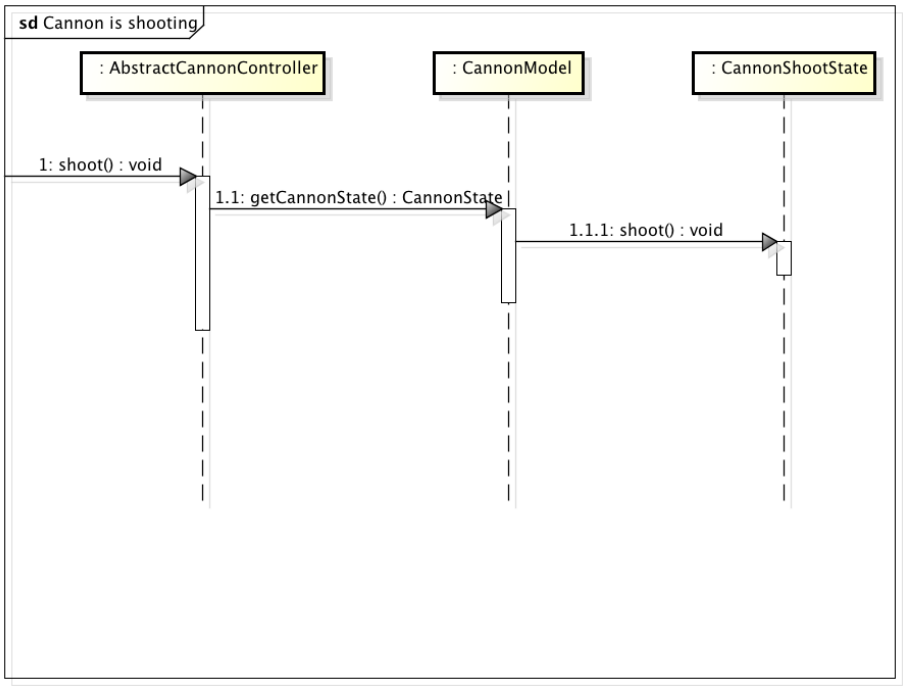

Figure 9: The sequence diagram for the implemented state pattern for the Cannonload behaviour

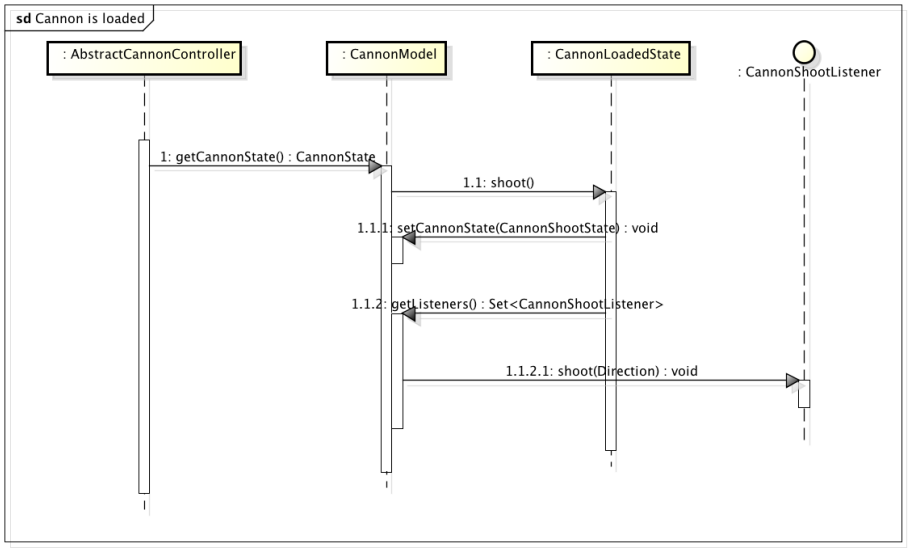

Figure 10: The class diagram for the implemented state pattern for the Cannonload behaviour