

Bubble shooter

Bavdaz, Luka
4228561

Clark, Liam
4303423

Gmelig Meyling, Jan-Willem
4305167

Hoek, Leon
4021606

Smulders, Sam
4225007

September 21, 2014

1 The Core

1.1 Responsibilities and collaborations of the main classes

The first important class is the GUI class, which is responsible for the user interface, as all buttons are contained there. Furthermore, it contains one JPanels containing the the game objects. It has a run method that calls the gameStep method of the cannon every frame.

The cannon class is responsible for converting mouse input to actions the cannon can take and managing all the bubbles that come in contact with the cannon untill they collide with the bubbleMesh. It is also responsible for drawing the cannon.

Another important class is the bubbleMesh class that is responsible for all the logic regarding the popping of bubbles, adding new rows of bubbles and checking for game ending conditions. It contains all the non moving bubbles.

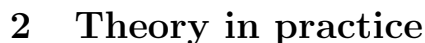
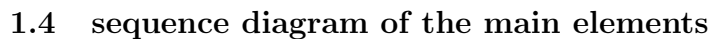
1.2 Less important classes

Other classes are less important because they contain mostly information about themselves instead of other objects. One missing responsibility is the lack of a class to manage all the game objects on the field. Also, the cannon class has multiple responsibilities: managing the cannon, handling user input and managing the bubble shot by the cannon. To repair this, a new class is introduced: The Room class.

Most central to the game logic is the Room class, which has the responsibility of containing all game objects of one player's side of the field. It is responsible for instantiating the game objects and calling their gameStep and render methods. Among other objects, it contains a cannon, the bubblequeue of bubbles that will be shot, the moving bubble that is shot and the bubbleMesh. The Room class takes the place of the cannon class as one of the main classes.

The cannon class is now only responsible for drawing the cannon and receives its input from a newly added cannon controller using an observer, which is also a main class. An additional benefit of this architecture is the ability to change how the cannon is controlled without modifying the cannon class.

1.3 class diagram of the main elements



2.1 Aggregation and composition

Our project uses composition mostly, since the member objects generally cannot exist without the containing class. The cannon and the bubblesmesh are always inside the same room; without the room these member objects cannot exist. Aggregation is used when the member object is a bubble. The bubbles

are created within the room, where they are placed in the queue. They do not stay there, since they can be shot and absorbed into the bubblemesh, thus being able to exist without either containing class.

2.2 Hierarchies

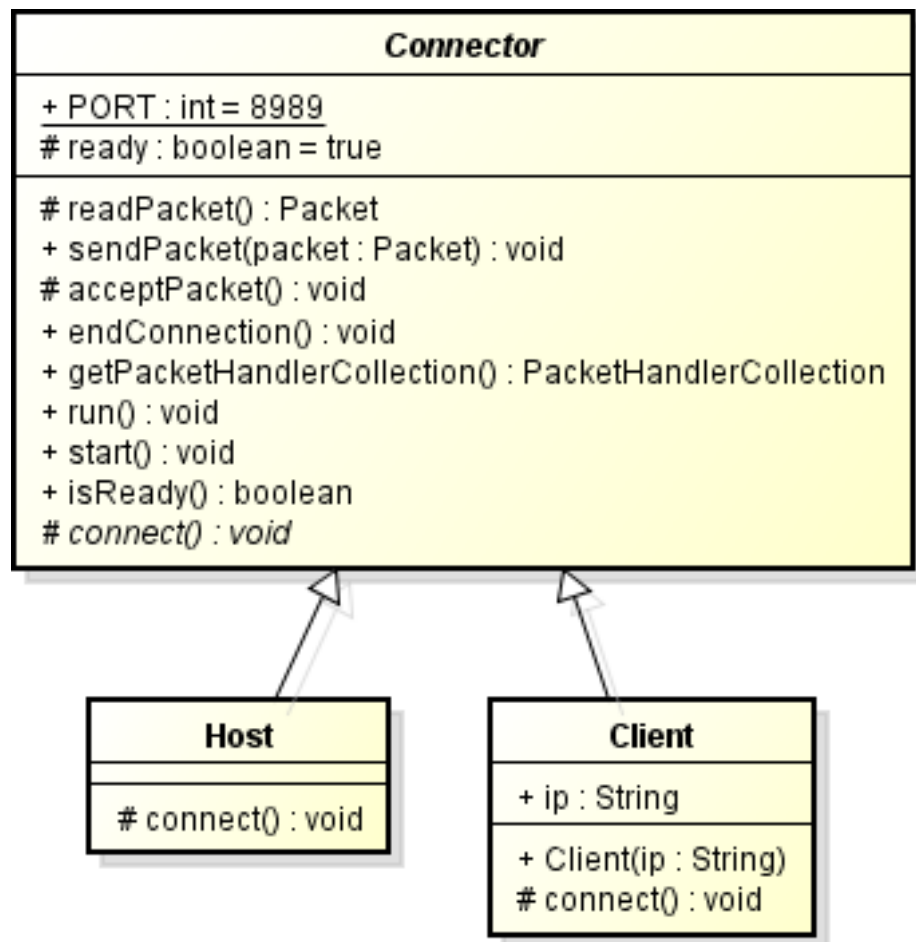


Figure 1

In figure 1, the Host and Client, share most of their responsibilities. So this is a case of the Host and Client is_a Connector. Also the classes who use the Host and/or Client, treat the classes in the same way. Both classes are used for sending and accepting packets in the same way. So there is polymorphism involved.

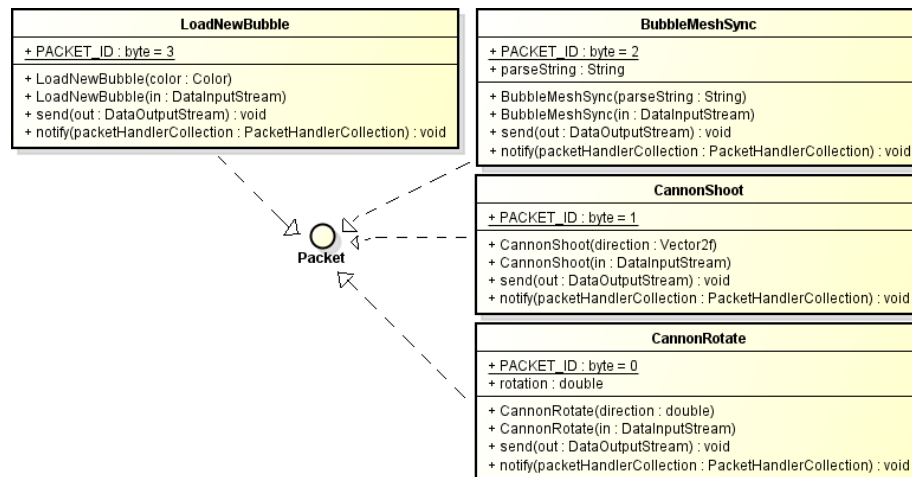


Figure2

In figure 2, we use polymorphism, because all packets share in how they are treated when they are being send.

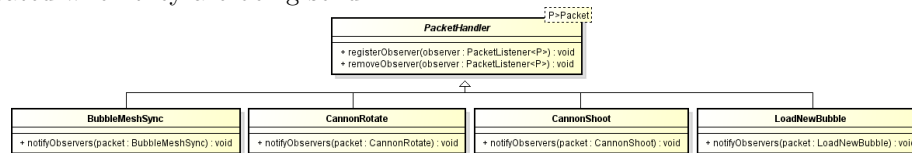


Figure3

The hierarchy of figure 3 is a case of is_a. All classes are PacketHandlers, they are all responsible for sending Packets to their observers.

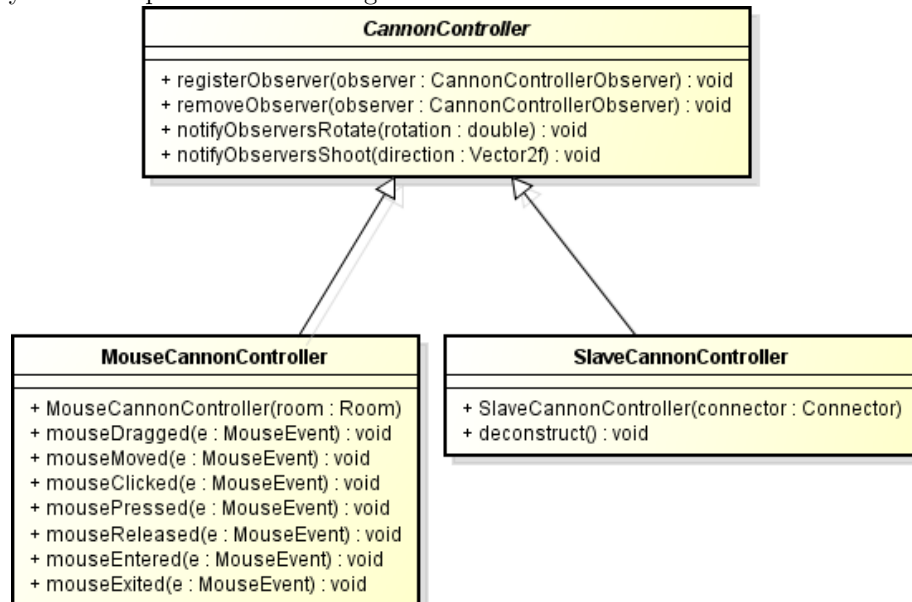


Figure4

The reason for inheritance for the CannonController in figure 4 is both, polymorphism and is_a. Both classes should determine the behaviour of the cannon and they should both be able to be observed in the same way, so they do share

responsibilities. So for both classes is a CannonController. Also polymorphism is involved, both classes are uniformly treated as in the CannonController is determined.

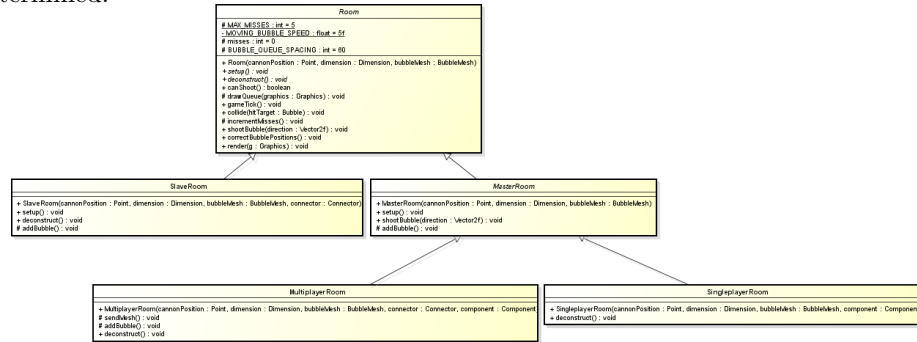


Figure 5

In the hierarchy in figure 5, there is a case of is_a, all classes share responsibilities. Also polymorphism is involved, all classes are uniformly treated as in the Room is determined. We choose to add an extra layer between the MultiplayerRoom and SingleplayerRoom between Room, because both have in common that they determine themselves what happens in the room, while the SlaveRoom asks the Connector, by receiving Packets, to determine what to do.

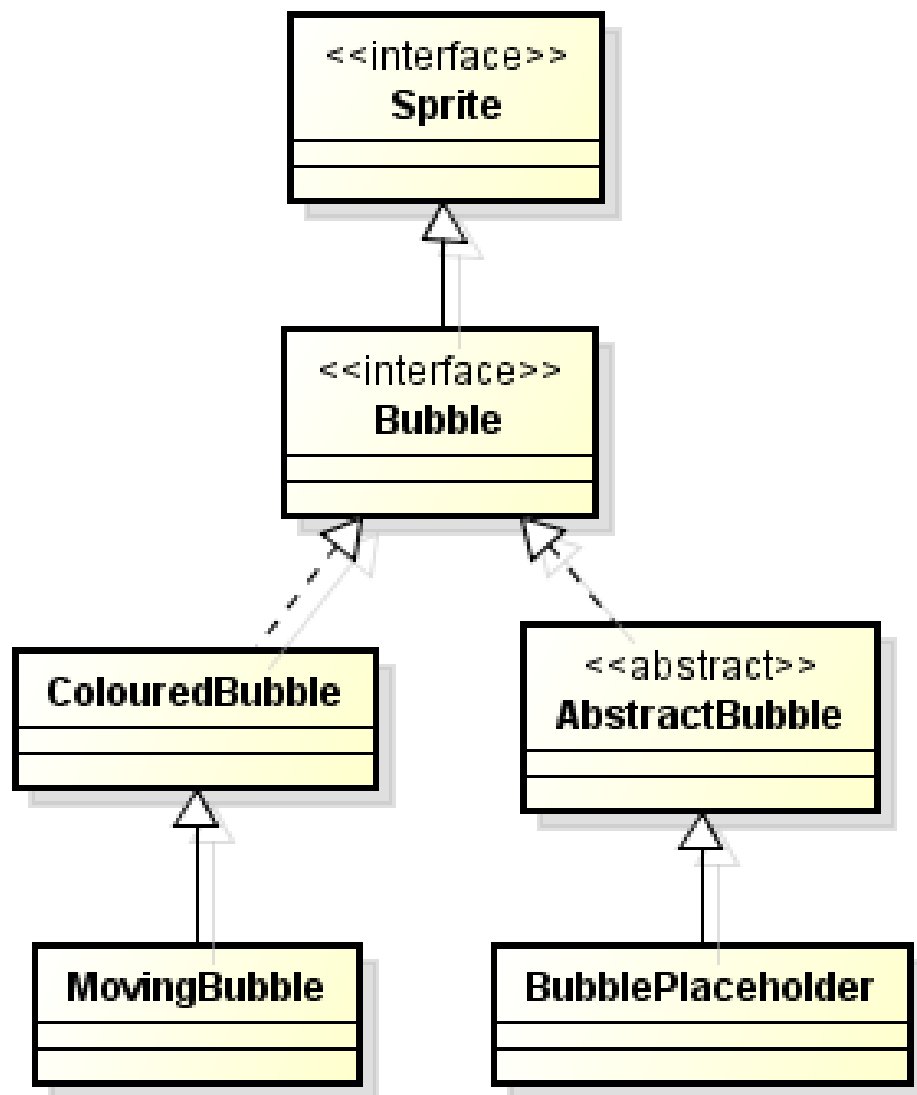


Figure 6

In figure 6 we use both generalisation and realisation. On the top of the hierarchy, we got the Bubble interface extending the Sprite interface. The Bubble should behave like a Sprite, since every Bubble should be able to be drawn. So here we use polymorphism. The ColouredBubble and AbstractBubble should both behave like a Bubble, so here we use polymorphism again. For the MovingBubble we use “is – a”, because the MovingBubble is a ColouredBubble with an extra behaviour. For the BubblePlaceholder we use “is – a” and polymorphism, we use polymorphism because we expect to add more AbstractBubble classes, and we want to treat them in the same way as the AbstractBubble. “is – a” because BubblePlaceholder shares responsibilities with AbstractBubble.

2.3 2.3 - If and switches

We placed a switch in the `nl.tudelft.ti2206.network.packets.PacketFactory.class`, the here switch is necessary to detect the difference between the raw data of packages before we can reconstruct the packages.

We left an if statement in `nl.tudelft.ti2206.cannon.MouseCannonController.class`, here we calculate the right angle for the cannon from raw data from the `MouseMovementListener`.