Bubble Beam - Assignment 5

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

October 26, 2014

1 20-Time, revolutions

1.1 Multiple game modes

In the previous sprint we introduced serveral game modes (from requirements `M-191` to `M-194`). A Game Mode basically defines what kind of bubbles a player can receive in his cannon. We have implemented several *Power-up Bubbles* (bubbles with a special effect). These Bubbles are constructed through a `BubbleFactory`, and the Game Mode was basically implemented by providing the `GameController` with another `BubbleFactory`.

Ofcourse it is a bit ambigious to let the `BubbleFactory` be the object that decides the `GameMode`. It also was a bit limited: we could provide new bubbles, but a more advanced game mode - Timed Game Mode (`M-193`) - actually failed because we had no possibility to hook on to the required methods - translating the bubbles - and events - time.

Speaking of events, over time, the game controller logic became a bit cluttered, after adding hooks and observers/listeners in various ways. Thus, in this sprint, we refactored the event handling system as a starting point for the more advanced game modes and multiplayer improvements.

1.1.1 Event handling

We already use event handling a lot: the `CannonController` triggers a `CannonShootEvent` (which is itself most likely triggered by an `MouseEvent`). The `GameController` listens for this `CannonShootEvent` and then starts doing its responsibility: allow the `MovingBubble` to move and check if it collides with other bubbles on its way, and if so, handle this collision in terms of snapping to the `BubbleMesh`, or popping with other bubbles.

All these actions are in fact events as well, and provide perfect hooks for game mode implementations and synchronization in the multiplayer. However, in the current version, this eventing system was just not complete enough to make this true. Luckily, the changes do not require a lot of new classes to be introduced, but rather requires to move around a few methods between classes and update their callers.

BubbleMesh

The `BubbleMesh` is a data structure for the `Bubbles`, and this structure needs to be maintained as bubbles gets snapped in to the mesh, popped, or inserted at the top. These events can be useful, as points need to be rewarded when bubbles pop. Also, when rows get inserted to the mesh, we want to send this to a potential multiplayer client.

`BubbleMesh`

Responsibilities	**Collaborations**
Datastructure containing the bubbles	`Bubble` objects
Logic to insert a new row of bubbles	`BubbleMeshListener`
Logic to snap a bubble into the mesh	
Logic to see if a snap caused any pops	
Notify `BubbleMeshListeners` of above events	

Figure 1: CRC-Card for the `BubbleMesh`

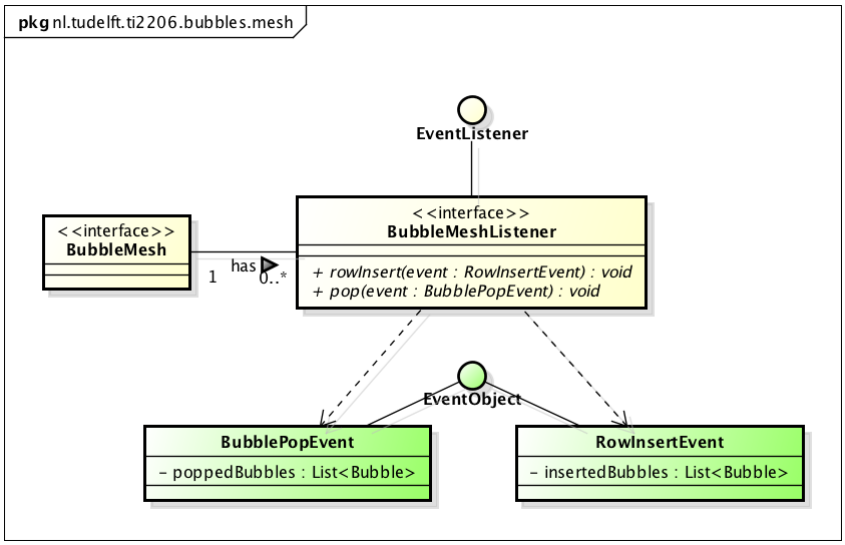

Figure 2: UML Diagram for the `BubbleMeshListener`

CannonController

The `CannonController` is responsible for the cannon specific logic. It triggers an event when the cannon shoots, which for example is necessary for the `GameController` to start translating the shot bubble.

`CannonController`

Responsibilities	**Collaborations**
Updating the `CannonModel` when the cannon rotates	`Cannon` instance
Preventing new shoot while shooting	`CannonModel`
Propagating `ShootEvent` to the `CannonListeners`	`CannonListener`

Figure 3: CRC-Card for the `CannonController`

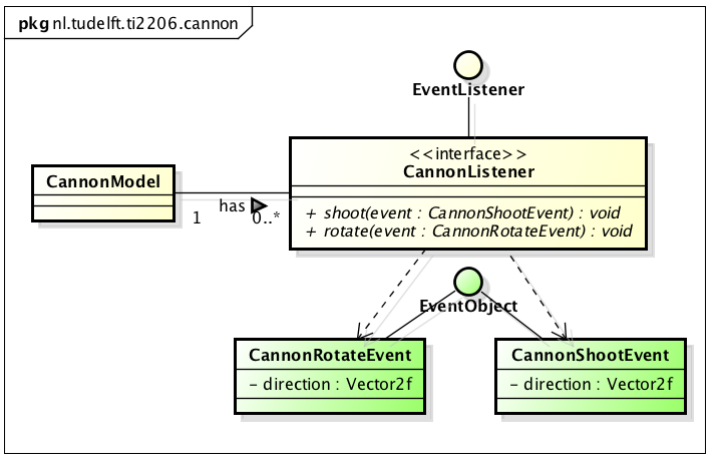

Figure 4: UML Class diagram for the `CannonListener`

GameController

The `GameController` is responsible for the generic game logic. See also the following CRC-card:

`GameController`

Responsibilities	**Collaborations**
Check collisions shot bubble	`BubbleMesh`
Update cannon ammunition	`CannonController`
Keep track of remaining colours	`GameListener`
Game Over handling	
Notify `GameListeners` of above events	
Propagate `ShootEvents` and `BubbleMeshEvents`	

Figure 5: CRC-Card for the `GameController`

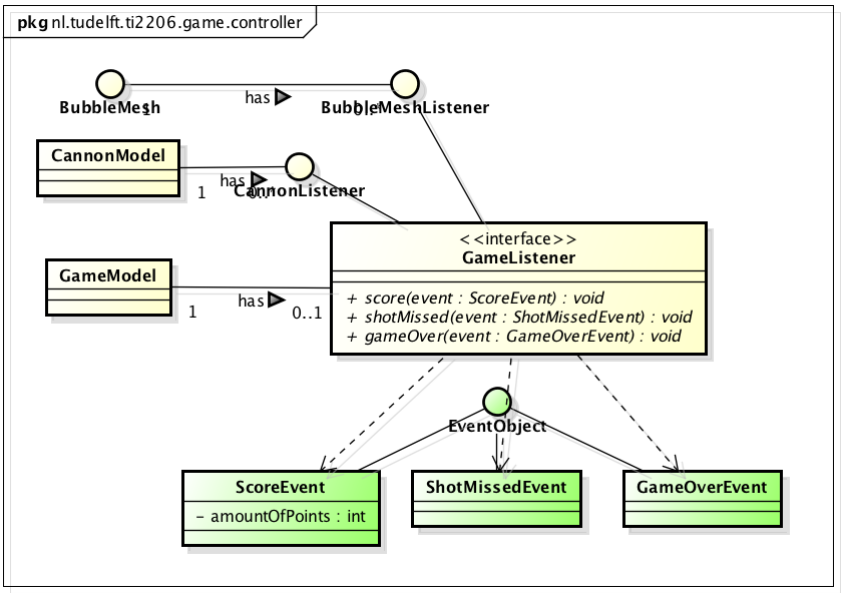

Figure 6: UML Diagram for the `GameListener`

1.1.2 Game Mode

From the requirements M-191 to M-194 we expect a Game Mode to have the following abilities: (1) it should be able to provide a certain BubbleFactory to the GameController, so that it can create the correct ammunation for the game mode; (2) it should be able to listen for GameEvents, for example to award points or insert new bubbles after a few misses; and (3) it should be able to listen on GameTicks to perform changes over time, such as pushing bubbles slowly to the bottom in the timed mode. Furthermore, we need to have access to the GameController to invoke these actions, and we also need to add some calls to the GameModel in the GameController.

GameMode

Responsibilities	Collaborations
Provide a BubbleFactory	BubbleFactory
Listen for GameEvents	GameController
Interact with BubbleMesh	BubbleMesh
Interact with GameControler	GameTick
Ability to hook onto GameTicks	

Figure 7: CRC-Card for the GameMode

Since we want a GameMode to hook onto GameEvents, we decided it should be a GameListener. Because we also want to hook on GameTicks, we decided a GameMode should also be Tickable. For the BubbleFactory and name of the GameMode, we defined getters in the GameMode interface.

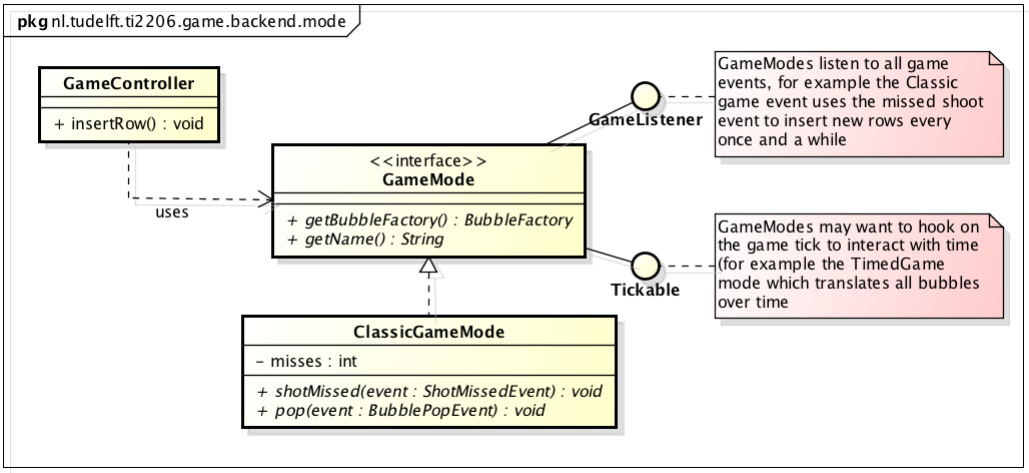

Figure 8: UML class diagram for the GameMode

Interactions

The ClassicGameMode provides bubbles through the DefaultBubbleFactory (which creates only ColouredBubbles and no Power-up bubbles), this is provided through the getBubbleFactory method. When bubbles pop, the player is awarded some points. This is achieved by overriding the pop method from the GameListener. When a shot bubble snaps into the BubbleMesh without popping, it is concidered a miss. After a few misses, a new row is inserted. The same as with the pop, this is done by implementing the shotMissed method. See also the sequence diagram 9 for these interactions between the GameMode and the GameController.

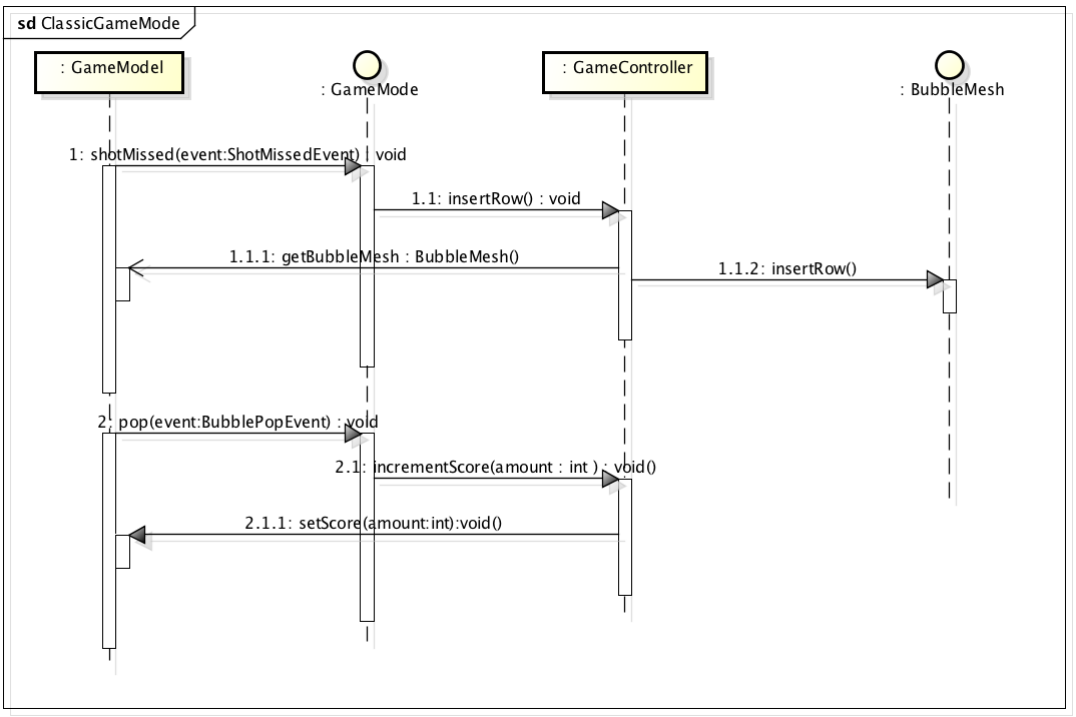

Figure 9: UML sequence diagram for the `GameMode`

1.1.3 BubbleMesh improvements

In the previous iteration a `Bubble` knew its position and was able to paint itself on a `Graphics` object. In the `paintComponent` function of the `GamePanel` we iterate over all bubbles in the `BubbleMesh`, and invoke the `render` method. For the `TimedGameMode`, this was not enough. In the `TimedGameMode` we want all bubbles to slowly fall down at a certain speed. When they reach the bottom, the game is over, or when the `BubbleMesh` is empty, you have defeated the game mode. We needed to be able to translate the entire `BubbleMesh`.

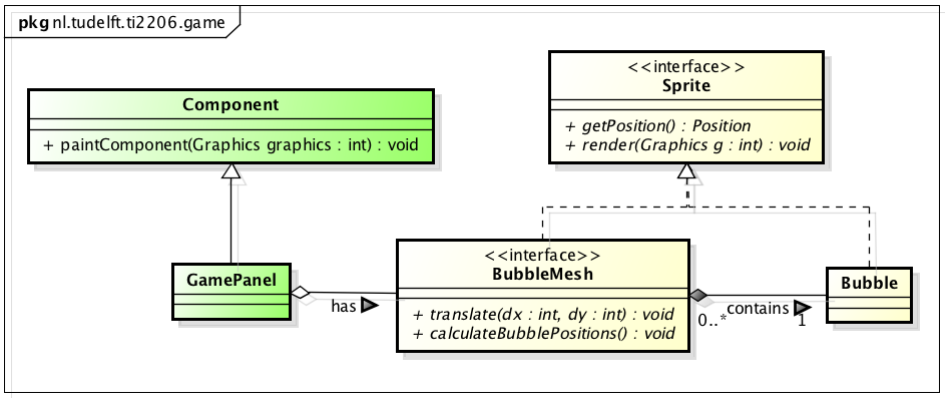

Figure 10: UML class diagram for the `BubbleMesh`

We figured out that it would be more clear to let the `BubbleMesh` be a `Sprite` as well, and give it the ability to draw itself and its bubbles. Then the `GamePanel` calls the render method of the `BubbleMesh` instead of the induvidual bubbles. Also we gave the `BubbleMesh` a position which can be translated - also moving the bubbles in the `BubbleMesh`.

With these adjustments to the `BubbleMesh` and the event listener changes described in section 1.1.1 and 1.1.2, we now have all the ingredients to make the `TimedGameMode` work: in the `GameMode` we can now hook onto the `GameTick` and then slightly translate the `BubbleMesh`.

1.1.4 Game modes for multiplayer

In the previous version we only transmitted the `CannonEvents` and some `BubbleMesh` syncs, and let the client then guess what other events might have been triggered. Also, we just hardcoded to always pick the `PowerUpBubbleFactory` (so what now would be the `PowerUpGameMode`). This did not give us the ability to play other `GameModes`, or use any of the Game Mode logic introduced in section 1.1.2.

To make the game modes available for the multiplayer mode (requirement `M-200`), we decided to rework the multiplayer logic. First, when we create a room (player 1 clicks "start multiplayer"), we want to be able to select one of the `GameModes`. Then we want to create two `GameModels` with this `GameMode` and an initial `BubbleMesh` and ammunition `Bubbles`. When a client connects (player 2 clicks "find multiplayer"), we need to transmit and process this initial data.

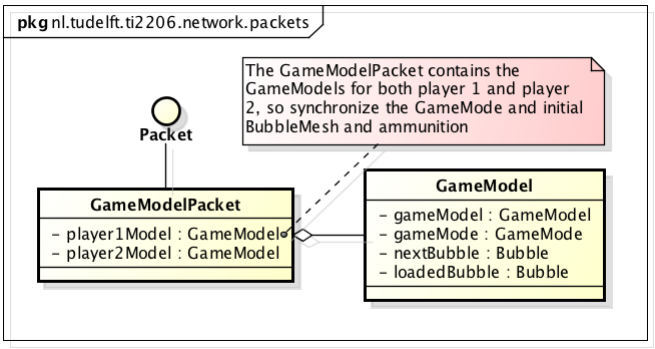

Figure 11: UML class diagram for the `GameModelPacket`

After transmitting the `GameModelPacket` both players can start playing. Now we need to transmit all actions between the two clients. In the previous version we used some dedicated packets for this, but the implementation was incomplete. Now we have an advanced event handling system (section 1.1.1), and all we have to do is listen for a `GameEvent` being triggered in the active game panels, wrap it in an `EventPacket`, transmit it to the client (figure 12).

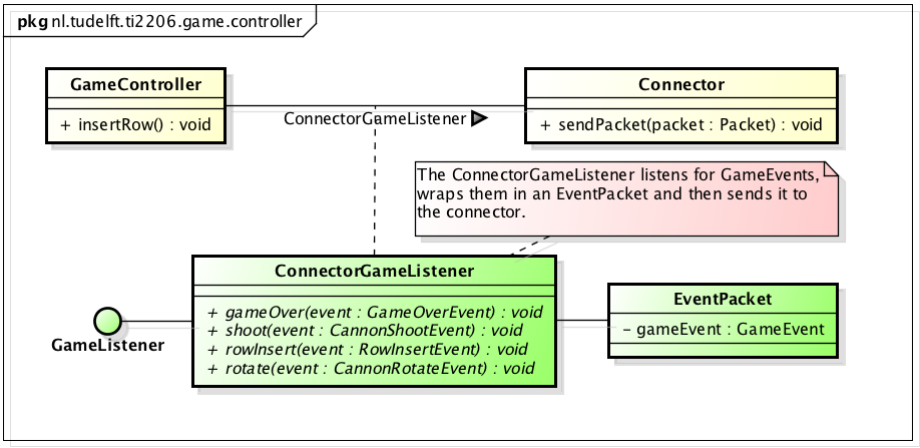

Figure 12: UML class diagram for the `ConnectorGameListener`

When the client receives an `EventPacket`, it needs to update the `GameController`. Therefore we have made a `PacketListener` that listens for `EventPackets`, and then invokes the event on the `GameController` (figure 13).

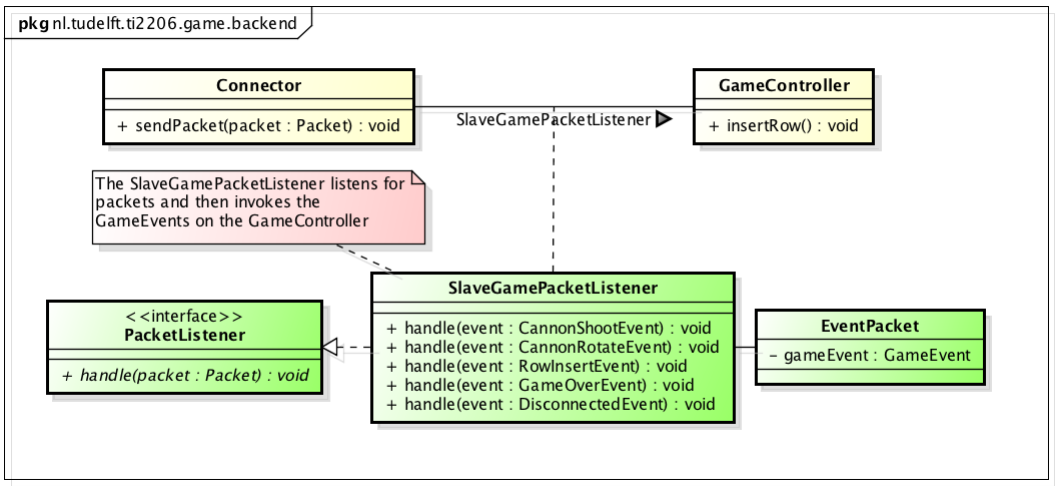

Figure 13: UML class diagram for the `SlaveGamePacketListener`

Now we have access to all required events in the multiplayer, which also allows the `GameModes` to work completely in multiplayer.

1.2 Pop animations

In the prvious sprint we already implemented two pop animmations for our requirement `M-123`. But we could only play a the same pop animation for all bubbles. In this sprint we want to add multiple pop animations and give different bubbles different animations (requirement `M-166`).

We already developed a `FallDownAnimation` and a `ShrinkAnimation`. We decided to give all normal bubbles the `ShrinkAnimation` and the `StoneBubble` the `FallDownAnimation`. For our `JokerBubble` we added the `ConfettiAnimation` class and for the `BombBubble` the `ExplosionAnimation` class.

To use different bubble animations for different bubbles we had to add a way to get the animation from the bubble. Therefore we added a method in the `Bubble` interface: `getAnimation() : FiniteAnimation`. In this method a `Bubble` returns the corresponding animation for when the (decorated) bubble pops. We also changed the way the animations are added to the game panel, we nog use the getAnimation method to get the annimation to be added. See Figure 16.

`Bubble`

New Responsibilities	**New Collaborations**
Logic to create a new `FiniteAnimation` on request	`FiniteAnimation`

Figure 14: CRC-card for the `Bubble`

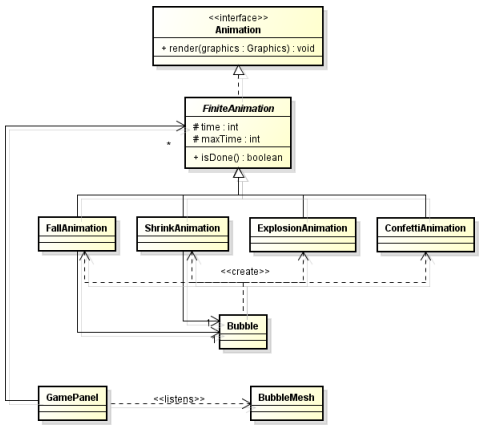

Figure 15: UML class diagram for the annimations

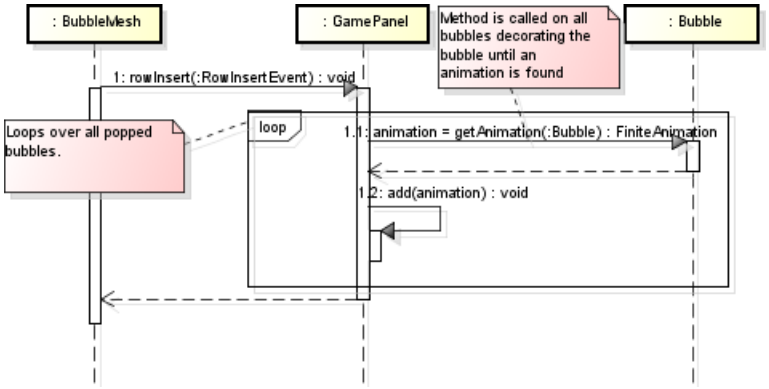

Figure 16: UML class diagram for the annimations

1.3 Multiple levels in game modes

According to requirement `M-201` we want new levels (new maps of bubbles) to appear when a player finishes in the singleplayer mode (requirement `M-129`). In the previous version we could already load a specific `BubbleMesh` from a text file, but we only used this for the default map. In this version, we updated the game to support a variety maps.

First, we enrichted the `GameMode` with two new methods: `hasNextMap` and `getNextMap`. This basically makes the `GameMode` an `Iterator`. Each `GameMode` should at least provide one map through these methods, but has the ability to provide more maps.

Then we needed to call these methods to actually use the maps. This was a bit more difficult: in order to create a game, we first create a `GameModel` with a `BubbleMesh`. Therefore it already needs to have the `BubbleMesh` while we yet do not have access to the `GameController` nor the `GameMode` instance - which lives in the `GameController`.

Therefore we changed the design a bit. The `GameModel` contains the `BubbleMesh`, but not necessarily has to be instantiated with it. It is then provided through the `GameController`, which also has the ability to reload it when the game is won. See also the sequence diagram in figure 17.

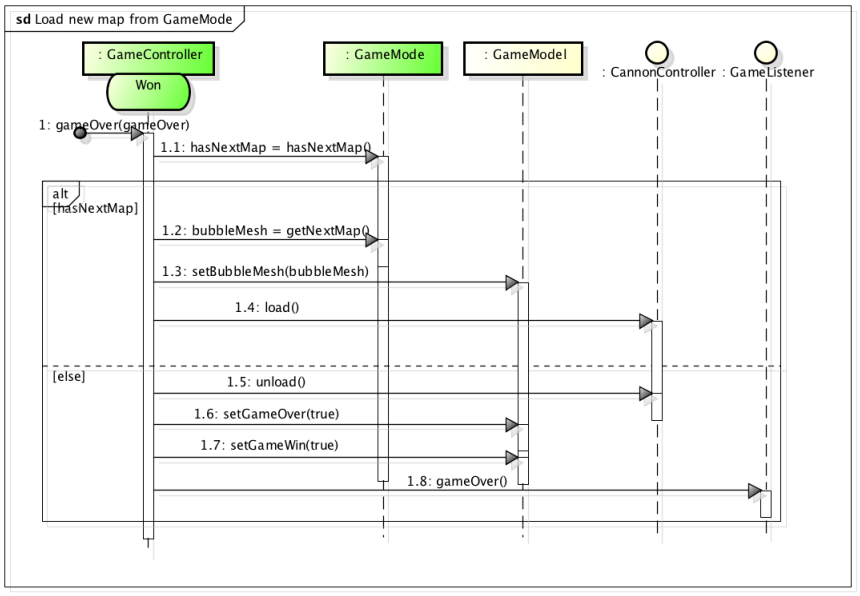

Figure 17: Sequence diagram for retrieving the next map for the game

2 Wrap up - Reflection

In the next paragraphs we will take a look on our progress and what we have learned during this project and course. We will discuss how using Scrum helped us to manage this project, what we have learned about ourselves as a team, how the use of design patterns influenced the project and helped us developing and finally we will look back at the beginning of the project.

2.1 Design Patterns

Design patterns have been really valuable to our project. Some complex problems could easily be solved by implementing a design pattern. These design patterns improved our overal code quality, gave our project some structure and also made it easier to extend our game.

Another big advantage was that the code was easier to understand for other project members because of the predefined way the design patterns should work. In combination with the UML and Javadoc it was easier to understand the work of others, without needing to ask them for their explanation.

The code quality was for example improved by implementing the state pattern for the cannon. This design pattern prevented many if-statements and resulted in better organized state handling.

The power-ups, for example, would have been implemented with one class for each combination of power-ups, which are resposible for all the behaviour. With the decorator pattern we where able to split the responsibilities and combine multiple power-ups, like the `SoundBubble` and `DrunkBubble`. This made it easier to add new power-ups and prevented a complicated inherentance or copying a lot of code.

By applying the factory method for generating bubbles, multiple game modes could be easily added to extend the game without needing a large generation method with many if-statements.

We will definitely use design patterns in our future projects. During this project the design patterns proved themselves most useful. Furthermore, in future projects design patterns can be used from the beginning to provide an effective structure for the rest of the code.

2.2 What we have learned about ourselves as a team

We have learned the importance of UML and documentation in general. It is very useful and necessary for communicating with team members about changes team members have made to the code. We learned that every team member has different strenghts, which could all be applied to different parts of the project.

We have also learned that it is difficult to work on an expanding project without getting overwhelmed by its growing complexity. Even though the design patterns brought some much needed clarity in our code we realized that we were spending a significant amount of time just reading the code to keep our understanding of the project up to date. That is a big difference compared to the smaller projects we did up until now. Now that we have completed this project we can be confident in ourselves that we have the tools and knowledge to tackle even bigger projects.

2.3 Practical Scrum

We highly valued the use of Scrum in our project. The methodology enforces a very task-based approach, which in our project lead to a very clear distribution of tasks. This is a good thing in any project because it gets rid of the "I don't know what I should do"-overhead. We also experienced Practical Scrum as a way to prevent a loss of focus inside our team when team members had different visions about what the next step in our project should be. We were forced to decide as one unit what the next course of action should be.

The weekly reflection on the previous sprint was an important tool in creating the next sprint. They were a great reference for how much we could do in a week's time. They also showed what areas still needed attention. Considering Scrum has become the industry-standard over the past few years, learning how to do it properly is a great step forward for all of us. We will use this methodology a lot in the coming years and are confident in our ability to use its strenghts to the fullest.

2.4 Evolution of the code

The code for BubbleBeam has come a long way since the first version. An enormous amount of features has been added, yet not in a way that endangered code quality or performance. During the development stages we have seen our simplistic game transform into a content-rich, strongly themed and noteworthy game. To be proud of the final product is a pleasant feeling for any software developer and we would like to share some of our story.

2.4.1 The intial sprint

The first part of development is always challenging. We had to decide what to build. We had tons of ideas, yet they had to be organized neatly in our requirements document. Many discussions were held whether something should be a must have or a should have, how to correctly phrase a requirement so that it is not too ambiguous and not in conflict with other requirements to ensure both developers and stakeholders understand and can agree on them. This was a new concept to us so we took our time for this and made sure the requirements were solid before we started programming.

One of the cornerstones classes of the project was made during this phase, namely the `BubbleMesh`. The `BubbleMesh` takes care of storing the bubbles and executing the logic of popping bubbles. At this stage this class was not perfect yet but it would soon become a solid foundation for the rest of the project.

All basic functionality was added during this phase but alot of classes had multiple responsibilities. For instance, the class `Cannon` had to do gamelogic, graphics and user input.

2.4.2 Assignment 1-2 - Refactor

We had to look back at our initial product and spot its flaws. We found classes with multiple responsibilities, bugs and missing requirements. We modelled the weak and cluttered parts of our code. Having a new arsenal of freshly learned design patterns we decided to tackle these problems. We chose to fix the issues with fitting design patterns, consisting of but not limited to: Observer, MVC and Strategy.

During this sprint and the next a lot of bug-hunting was done and many bugs were found. We made a logger to provide us with the vital information needed about the program, to find and solve more and more bugs.

We ran into major trouble with the multiplayer mode. There was not enough time to do it the way we wanted to do it. So we chose to make a more basic play-together mode first. Then in next few sprints we refactored the multiplayer mode the way it was defined in the requirements document.

2.4.3 Assignment 3 - Test it

With a solid foundation in place, the feature creep could start. We gave the game a solid theme and used more patterns to create more features while trying to maintain a clear control flow of the code. good sprint management was essantial here. We were lacking behind in the testing department so time was allocated in the plan to make up for this. we quickly achieved around 60% code coverage. and planned in more time in further sprint to up this to.

2.4.4 Assigment 4 - Incode the oracle hath speaketh

It was time to put our code quality to the test. Incode had to be run and with pleasing results. Incode discovered only two violations. The design patterns had kept coupling of classes and multiple responsibilities to a minimum. Besides fixing the violations we also upped the test coverage to around 85% for non-gui classes. But mostly this phase was about perfecting and reapplying what we learned earlier, to perfect the art.

2.4.5 Assignment 5 - The final stretch

The final phase of development were upon us, here we mostly polished the product we strenghted the theme with music, more animations and an overal better user interface. We also implemented the predesigned levels. We ensured it is a game we are proud of and that it represents what we have learned. We hope that everyone who plays this game will derive as much joy from it as we did!