Analysis and design phase document - Assignment 4.1

In this document we describe the analysis and design phase for exercise 1.

First we describe how we analysed and designed the pop animation, game modes and highscore. Then we will describe how we analysed and designed the competitive multiplayer. We do this separate, because the pop animation, game modes, and highscore where implemented by adding new parts to our project, while the competitive multiplayer required mostly changes in the existing classes.

Classes extracted from the requirements

For the pop animation of requirement C-193 we came up with a PopAnimation, which would be drawn in the existing GamePanel class. For each bubble popped, the GamePanel should add a PopAnimation to a list of PopAnimations.

Different sets of bubbles can be generated in the classic gamemode, the powerup gamemode and the drunk gamemode. Therefore we made a different bubble factory for each gamemode that implements the BubbleFactory interface. This resulted in the ClassicBubbleFactory, the PowerUpBubbleFactory and the DrunkBubbleFactory classes.

A Highscore class is needed to maintain the highscore for single player games. Each entry in Highscore contains a name and a score, which is contained in the ScoreItem class. The HighscorePopup class is used to display the highscores.

Responsibilities of classes

The PopAnimation is responsible for drawing the animation on its position on the window.

The GamePanel is responsible for drawing all components assigned to it, including all the PopAnimation objects.

The BubbleFactory classes mentioned previously (ClassicBubbleFactory, PowerUpBubbleFactory and DrunkBubbleFactory) have the responsibility to generate bubbles appropriate for the game mode.

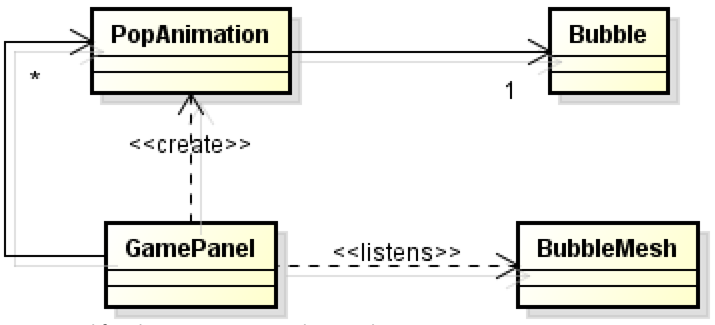
The Highscore class is responsible for maintaining and saving the ordered list of highscores. The responsibility of the ScoreItem class is to contain the variables of one highscore. The HighscorePopup is responsible for displaying the highscores.

Collaborations between classes

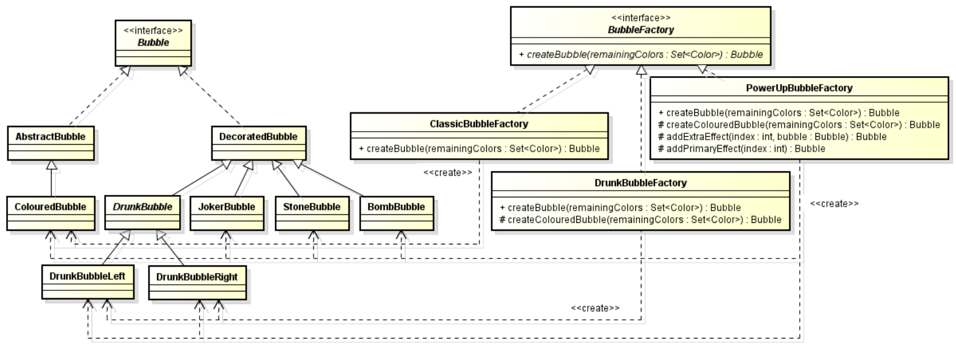
The GamePanel is listening to pop events of its BubbleMesh. When a Bubble pops, the GamePanel listening to the BubbleMesh adds an new PopAnimation to its list of PopAnimations. The PopAnimation is constructed with the popped Bubble as parameter. On every repaint of a GamePanel, the GamePanel should paint all PopAnimations.

The factory method is applied for generating the bubbles for different game modes. The BubbleFactory class functions as the creator, which is an interface that is implemented by all the concrete creators. The concrete creators are the bubble factories used in the different game modes: ClassicBubbleFactory, PowerUpBubbleFactory and DrunkBubbleFactory. Each concrete factory can create a different set of bubbles when the factory method, createBubble(), is called. The product of the factories is a class that implements the Bubble interface. The concrete products created by the factories are ColouredBubbles or subclasses of DecoratedBubble.

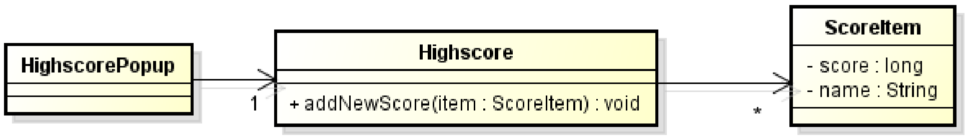
UML used for the design phase



UML used for the pop animation design phase


<<interface>>
Bubble
<<interface>>
BubbleFactory
+ createBubble(remainingColors : Set<Color>) : Bubble
PowerUpBubbleFactory
+ createBubble(remainingColors : Set<Color>) : Bubble
# createColouredBubble(remainingColors : Set<Color>) : Bubble
# addExtraEffect(index : int, bubble : Bubble) : Bubble
# addPrimaryEffect(index : int) : Bubble
AbstractBubble
DecoratedBubble
ClassicBubbleFactory
+ createBubble(remainingColors : Set<Color>) : Bubble
<<create>>
DrunkBubbleFactory
+ createBubble(remainingColors : Set<Color>) : Bubble
# createColouredBubble(remainingColors : Set<Color>) : Bubble
<<create>>
ColouredBubble
DrunkBubble
JokerBubble
StoneBubble
BombBubble
DrunkBubbleLeft
DrunkBubbleRight
<<create>>

UML used for the game modes design phase



UML used for the highscore design phase

Design choices during development

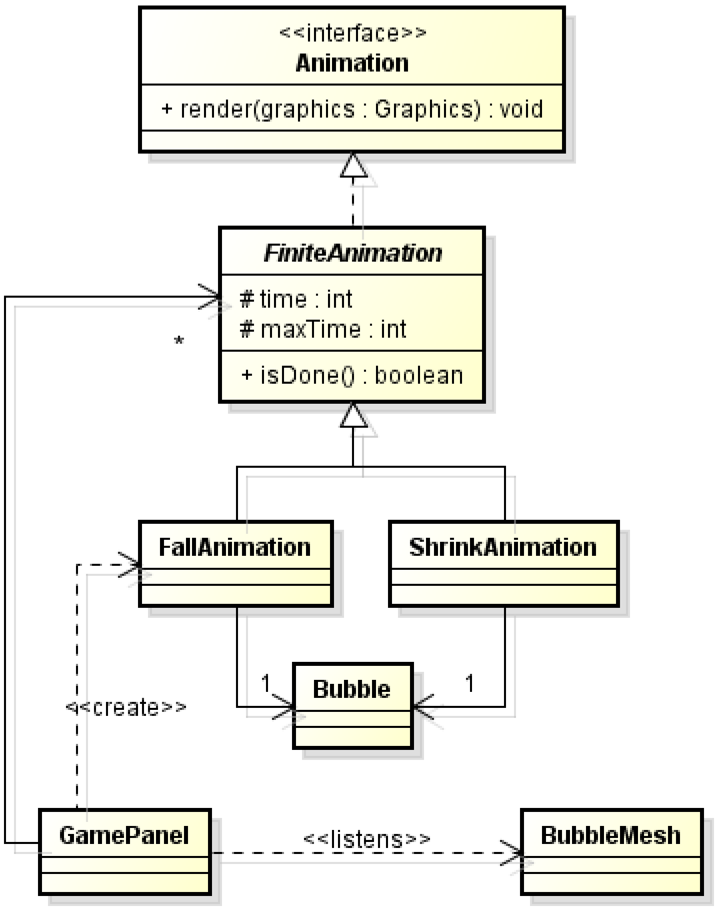
We chose to make multiple pop animations, called ShrinkAnimation and FallAnimation. We will chose one of them or we could add different animations to different bubbles later. To implement this, we required a common superclass / interface. Both pop animations are finite animations, which means, they stop at some point. So we added the abstract class FiniteAnimation, which is responsible for determining the stop condition. Because there might be more animations which have nothing to do with Bubbles, we moved the Bubble associations to the animations themselves.

We may add other non-FiniteAnimations later, so we added the Animation interface, which is implemented by the FiniteAnimation.

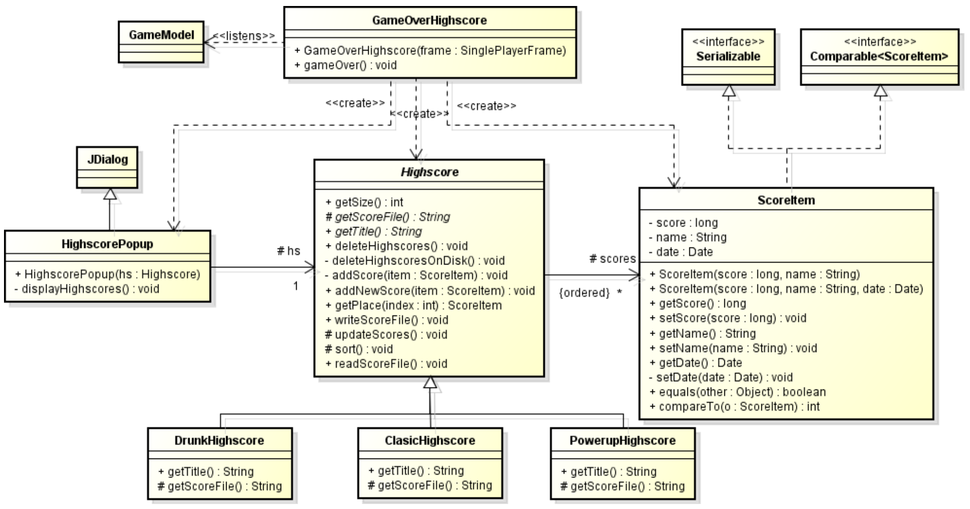
To allow for different lists of highscores depending on the game mode, subclasses of the Highscore class were introduced: ClassicHighscore, PowerUpHighscore and DrunkHighscore. To store the ScoreItem we made it implement Serializable, so we could store it as an Object in a file.

The class GameOverHighscores was created to listen to the GameModel for the occurrence of a GameOver. It implements the GameOverEventListener. The GameOverHighscores requests the users name with a JDialog and then creates a new ScoreItem. This ScoreItem is added to the Highscore class of the gamemode that is played at that moment. Then the highscores are shown in a HighscorePopup.

UML after implementation



Class diagram of the animations after implementation

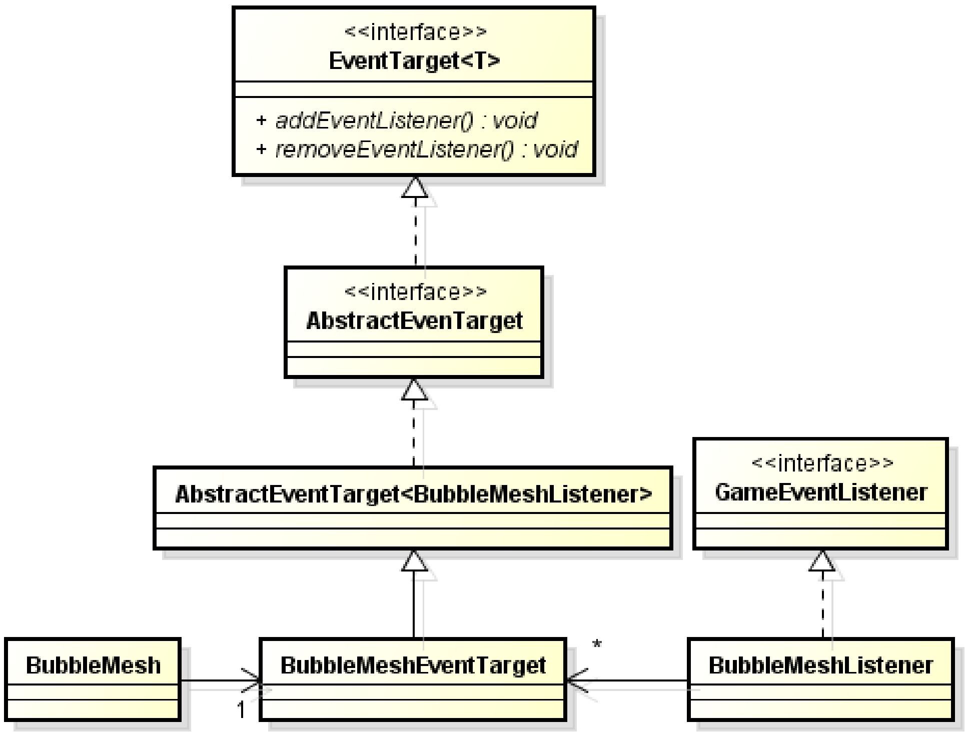

GameModel
<<listens>>
GameOverHighscore
+ GameOverHighscore(frame : SinglePlayerFrame)
+ gameOver() : void
<<create>>
<<create>>
<<create>>
<<interface>>
Serializable
<<interface>>
Comparable<ScoreItem>
JDialog
HighscorePopup
+ HighscorePopup(hs : Highscore)
- displayHighscores() : void
# hs
1
Highscore
+ getSize() : int
# getScoreFile() : String
+ getTitle() : String
+ deleteHighscores() : void
- deleteHighscoresOnDisk() : void
- addScore(item : ScoreItem) : void
+ addNewScore(item : ScoreItem) : void
+ getPlace(index : int) : ScoreItem
+ writeScoreFile() : void
# updateScores() : void
# sort() : void
+ readScoreFile() : void
# scores
{ordered} *
ScoreItem
- score : long
- name : String
- date : Date
+ ScoreItem(score : long, name : String)
+ ScoreItem(score : long, name : String, date : Date)
+ getScore() : long
+ setScore(score : long) : void
+ getName() : String
+ setName(name : String) : void
+ getDate() : Date
- setDate(date : Date) : void
+ equals(other : Object) : boolean
+ compareTo(o : ScoreItem) : int
DrunkHighscore
+ getTitle() : String
# getScoreFile() : String
ClasicHighscore
+ getTitle() : String
# getScoreFile() : String
PowerupHighscore
+ getTitle() : String
# getScoreFile() : String
Class diagram of the highscore after implementation

Competitive multiplayer

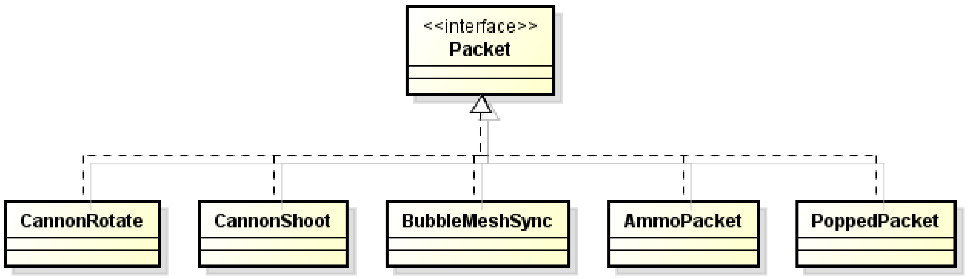
We realized we should be able to listen on a bubble-pop event and be able to transfer this information over the socket. We also wanted to centralize the way information was sent through the socket. At first, the bubblemesh was an event target triggering score events, so that score updates could be sent to the game controller. There was no event handling for inserting rows or popped bubbles, since all places were we needed access to these events, we're able to hook on to this directly.

For the multiplayer over the network however, this did not work. We needed a more advanced model for the event handling, at several levels: (1) the bubble mesh - which is responsible for inserting rows of bubbles and replacing bubbles once they have snapped or popped, (2) the cannon - which triggers cannon shoot events when a player shoots a bubble and (3) the game controller - which is responsible for triggering game over events.

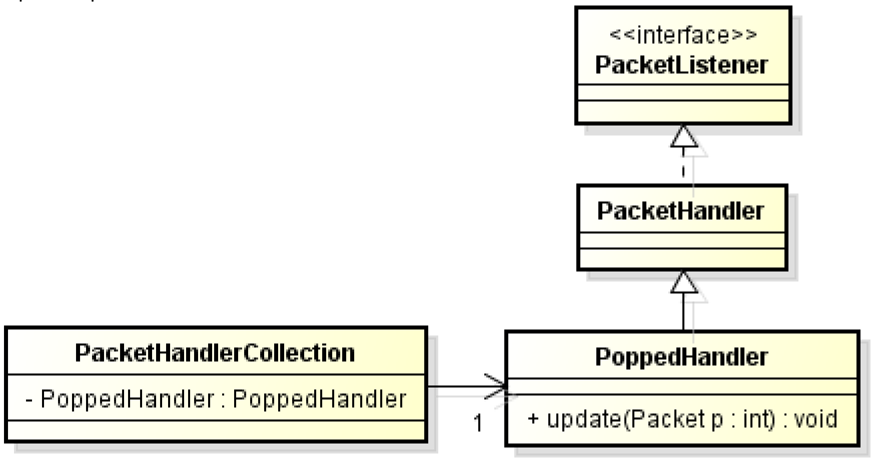
Besides that we needed extra Packets to send new information about these events over the socket. The score and cannon packets could be reused so we simply added a PopPacket that transfers the amount of bubbles popped from the pop events.



BubbleMesh event listener UML



Updated packets UML



Updated PacketListener UML