Analysis and design phase document

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

Classes extracted from the requirements

For generating the power-up bubbles every once in a while, as explained in requirement M-160, a factory class should be used.

For the different power-ups we should use a different class for each type. Out of the requirements (M-160 to M-166) we came up with the classes BombBubble, JokerBubble, StoneBubble, DrunkBubble, SoundBubble, ReverseBombBubble.

Responsibilities of classes

The power-up factory is responsible for generating the power-up bubbles with meaningful combinations of concrete decorators.

Each bubble power-up class is responsible for implementing a specific power-up. The power-ups all extend the BubbleDecorator. The decorator bubbles are responsible for forwarding called methods to their wrapped bubble component.

Zo goed?

The BombBubble is responsible for the bomb behaviour. It should pop all bubbles in a certain radius once it snaps in the bubble grid. The pop behavior should pop all bubbles in the radius, regardless of the types of the bubbles within the radius.

The JokerBubble is responsible for the joker bubble behaviour, which means, it should act like any colour it touches, and pop the bubbles if the joker bubble connect more then a certain number of bubbles.

The StoneBubble is responsible for the stone bubble behaviour, which means, it shouldn't be popped by any other colour or stone bubble. Only a BombBubble, or popping the bubbles who are connecting the stone bubble to the top, should pop it.

The DrunkBubble is responsible for the drunk bubble behaviour, which means, it should make the bubble it is decorating move with a slight curve.

The SoundBubble is responsible for playing a sound when the bubble is popped.

The ReverseBombBubble is responsible for populating the grid within a certain radius once it snaps into the grid.

Collaborations between classes

The power-up factory creates the bubbles, including all the concrete decorators. The factory is used by the GameController, since it replaces the previous bubble generation code.

Since some power-ups can occur simultaneously in one bubble, a decorator would be a good solution. This means we will require an decorator class, which each concrete decorator will extend. As component for our decorator pattern we will use the Bubble interface, because power-ups are tied to each bubble. MovingBubble was also considered as the component, but it could not be used because the StoneBubble and the SoundBubble should exists even after snapping into the BubbleMesh. As concrete components we use the existing AbstractBubble class and all its subclasses.

All the power-up classes, mentioned earlier, BombBubble, JokerBubble, StoneBubble, DrunkBubble, SoundBubble, ReverseBombBubble, are our concrete decorators.

UML used for the design phase

Titel..

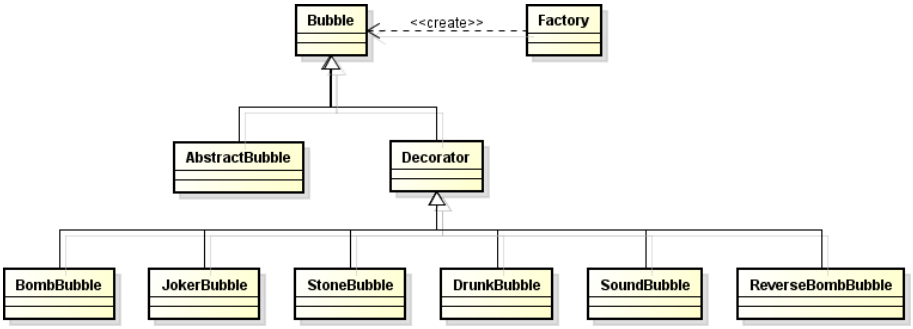

Figure 1: Class diagram used for the design phase

Design choices during development

Because the moving bubble also adds an effect to the bubble, like the other decorator bubbles, so we decided to make the moving bubble into a decorator bubble. This also simplifies the implementation of DrunkBubble.

During implementation we also decided to introduce the subclasses DrunkBubbleLeft and DrunkBubbleRight, to avoid if statements using advanced object oriented programming. Double dispatch is used to achieve this goal.

For our factory we also decieded to make it extend an abstract factory, so we could add other factorys later for implementation. This is useful for implementing multiple game modes where different power-ups are created with different probabilities.

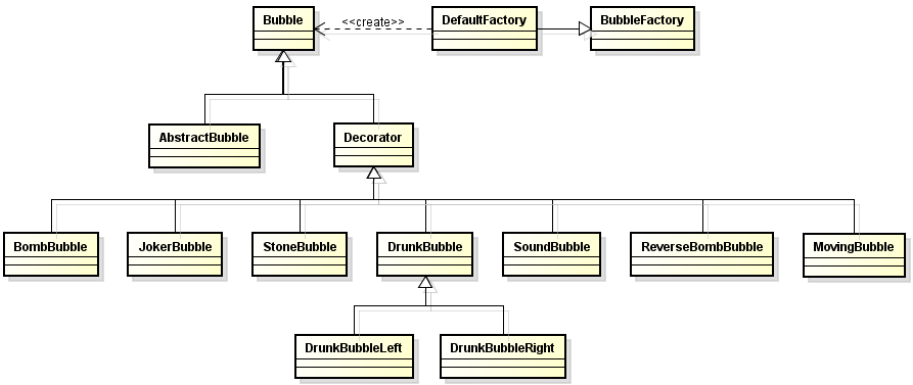

Figure 2: Class diagram after implementation