TI2206 Software Engineering
Bubble Shooter report: Assignment 3
EEMCS/EWI

Gerlof Fokkema 4257286
Owen Huang 4317459
Adam Iqbal 4293568
Nando Kartoredjo 4271378
Skip Lentz 4334051

October 12, 2014

Exercise 1: 20-Time

In this section improvements made to Bubbleshooter during the '20-Time' will be elaborated. Additionally an in depth description of what has been implemented and the resulting class diagram can be found in this section.

Responsibility Driven Design applied to the arcade-mode

It is usually good practice to think about something before actually doing it. This means that thinking about the various responsibilities of all the classes for implementing arcade mode is not something to be forgotten. *Arcade-mode* consist of multiple levels that must be played with 'one-life', once game-over the user should have to play all over again from the start. *Arcade-mode* also contains a timer in this case. When the timer hits zero, the user loses. The structure and the responsibilities of the losing conditions have not not changed that much (compared to the last sprint). This means that only a few new things should be introduced in order to make the *arcade-mode* work. Mainly there is a need for levels. Additionally it would be great if these levels can be easily added, read and stored somewhere. Using a simple parser for reading simple text files is a good solution. To distribute the various new responsibilities in a good manner, new design patterns could be used to achieve this. How this is exactly done can be read in the next paragraph.

Bubbleshooter arcade-mode using design patterns

In order to make the single-player mode more interesting the decision to create our own levels for our *BubbleShooter of Doom* has been made. This means that the levels are unique and not based on an existing Bubble Shooter game.

In order to achieve this, the following elements were needed:

1. A parser for parsing levels from a file.
2. A way to:
   a) Win a level.
   b) Keep your score.
   c) Start the next level.

To implement *arcade-mode*, the *Factory* design pattern was used. This design can be realized by making an abstract *BoardFactory* class to prepare a list of *Boards*.
Concrete childs of the *BoardFactory* can then prepare different lists of *Boards*, that will be used during the different game modes.
We have decided to create the following *BoardFactories*:

1. MultiPlayerBoardFactory *(MPBoardFactory)*.

2. ZenBoardFactory.

3. ArcadeBoardFactory.

To implement *arcade-mode*, as we chose *arcade-mode* would be, we needed to be able to progress to the next level. The Iterator pattern was used, so we could traverse to the next levels without specifying underlying code. The class and sequence diagram used to visually depict this design pattern can also be found in the next section in **Fig. 4** and **Fig. 5** respectively.

Responsibility Driven Design applied to the special bubbles
In the *Bubble Shooter of Doom* there is a distinction to be made between regular and special bubbles. These regular bubbles are classified in the *Bubble* class. To create special bubbles all we need to do is find a way to take the original regular bubble and apply some specialties to it. In order to do this, it is possible to make an extension of the original bubble and make some adjustments to it. This way of creating the special bubbles is favorable, because the responsibility of taking care of the special bubble is now in a seperate class. Furthermore the way the *Bubbles* behave should also be put in a different class, from now on *BubbleBehavior.* This is ideal because the responsibilities of having distinct qualities in either behaviour or physical appearance is now put in different classes. This way the responsibilities are nicely distributed over different kinds of classes. How this is exactly implemented and the reasoning behind choices being made can be read in the next paragraph. This implementation of the special (and regular) bubble also includes the use of a design pattern: the *Strategy* pattern.

Bubbleshooter special bubbles using a design pattern
A major feature added to the game is having special bubbles, as mentioned before. These bubbles behave in a different way compared to the regular bubbles. An example of one of the special bubbles is the *Stone* bubble. This is a bubble that can only be removed when the user manages to disconnect it from the ceiling/other bubbles.This means it will be a lot harder to remove such a bubble and potentially increases the difficulty of the level. There are many other special bubbles one may encounter during the levels! Currently all the special bubbles are:

1. Stone bubble (description above).

2. Bomb bubble (blasts away all adjacent bubbles).

3. Nuke/MichaelBay bubble (nukes whole level, causing instant win).

To implement these special bubbles, a *Strategy* design pattern was applied. This was especially useful because all bubbles have a certain behaviour and a different look (texture). A special bubble will have its own class and its own behaviour extending the *Bubble*. In its own class a different texture can be specified and a different color as well (by overriding the method responsible for the texture). Furthermore the regular bubble (which is not 'special') has a behaviour class (*BubbleBehavior*) which can be changed by creating additional behaviour classes overriding the methods responsible for its special behaviour. A visual depiction of the implementation of the special bubbles can be found **Fig. 1**.

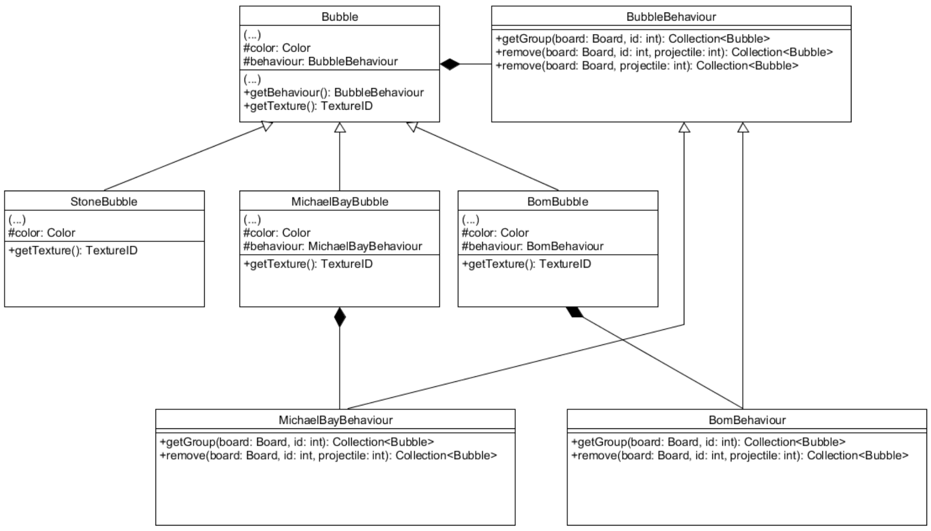
Bubble
(...)
#color: Color
#behaviour: BubbleBehaviour
(...)
+getBehaviour(): BubbleBehaviour
+getTexture(): TextureID
BubbleBehaviour
+getGroup(board: Board, id: int): Collection<Bubble>
+remove(board: Board, id: int, projectile: int): Collection<Bubble>
+remove(board: Board, projectile: int): Collection<Bubble>
StoneBubble
(...)
#color: Color
+getTexture(): TextureID
MichaelBayBubble
(...)
#color: Color
#behaviour: MichaelBayBehaviour
+getTexture(): TextureID
BomBubble
(...)
#color: Color
#behaviour: BomBehaviour
+getTexture(): TextureID
MichaelBayBehaviour
+getGroup(board: Board, id: int): Collection<Bubble>
+remove(board: Board, id: int, projectile: int): Collection<Bubble>
BomBehaviour
+getGroup(board: Board, id: int): Collection<Bubble>
+remove(board: Board, id: int, projectile: int): Collection<Bubble>

Fig. 1: Class diagram of the special bubbles.

Refactoring EndingConditions

Lastly, the code of the *EndingCondition* class was refactored. In the previous design, we used a ternary condition to indicate that the game was lost or won, or that it should continue. The condition was an `int`, where a negative value indicated that the game was lost, a value equal to zero indicated that the game should continue, and a value greater than zero indicated that the game was won. This wasn't really future-proof, in case we would want to notify other objects on the game's ending state. We therefore introduced an *EndingObserver* class, which *observes* the *EndingCondition*. You can see the class diagram for this in **Fig. 3**.

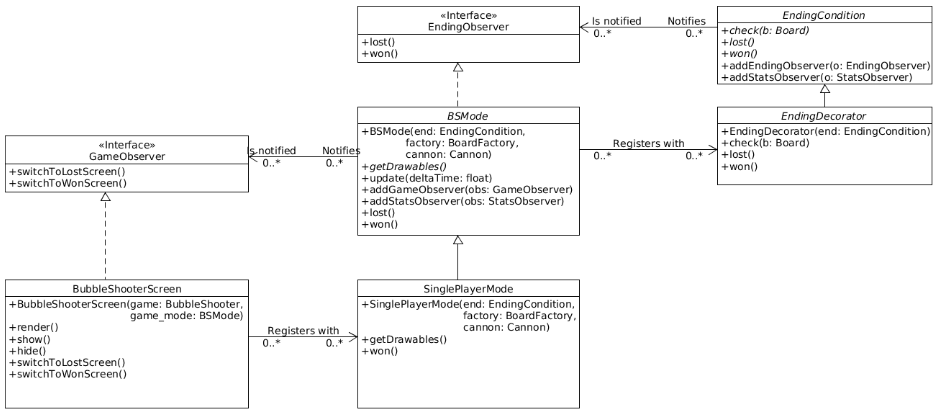
«Interface»
EndingObserver
+lost()
+won()
Is notified
0..*
Notifies
0..*
EndingCondition
+check(b: Board)
+lost()
+won()
+addEndingObserver(o: EndingObserver)
+addStatsObserver(o: StatsObserver)
BSMode
+BSMode(end: EndingCondition, factory: BoardFactory, cannon: Cannon)
+getDrawables()
+update(deltaTime: float)
+addGameObserver(obs: GameObserver)
+addStatsObserver(obs: StatsObserver)
+lost()
+won()
Registers with
EndingDecorator
+EndingDecorator(end: EndingCondition)
+check(b: Board)
+lost()
+won()
«Interface»
GameObserver
+switchToLostScreen()
+switchToWonScreen()
BubbleShooterScreen
+BubbleShooterScreen(game: BubbleShooter, game_mode: BSMode)
+render()
+show()
+hide()
+switchToLostScreen()
+switchToWonScreen()
SinglePlayerMode
+SinglePlayerMode(end: EndingCondition, factory: BoardFactory, cannon: Cannon)
+getDrawables()
+won()

Fig. 2: Class diagram of the ending- and game observer.

Another part that was refactored was how we could construct different kinds of *EndingConditions*. Before, we had to extend an abstract *Ending-Condition* class, and define a new *check* function for each kind of condition. We came up with a *TimerCondition* and a *BelowLineCondition*. But if we wanted to combine these two conditions, so that we had a *timer* and a *death-line*, we had to make a new class for this combination. This wasn't really flexible, so we decided to apply the *Decorator pattern*, with which we can easily combine different kinds of *EndingConditions*, to form a new *Ending-Condition* at runtime. The class diagram is shown in section 3.

Exercise 2: Design Patterns

In this section two design patterns are elaborated with fully in depth explanations. Furthermore these patterns come with a visual depiction of how they were implemented in the project.

2.1 | Pattern 1: Factory pattern for Boards

1. Factory gives us an interface for creating multiple (family) related objects. This without explicitly specifying their concrete classes. This design pattern falls under the *Creational design.*

   We had to choose a way to make our classes more encapsulated and relative easy to be extendable.

   We explicitly chose the factory patternthis gives us a way to have the freedom to make each different BoardFactory still as their 'own' concrete classes. Maybe an easier to understand definition is that Factory is a super-factory which creates other factories, like a factory for factories.

   With this pattern methods are easy to reuse as if we want to instantiate in any other or many places, the chance when missing something when making a new class is relatively small.

   It makes our code far more flexible for extension and testing, whenever we or someone else should decide there should be added another class, none of the source code should/has to be modified as we do not need to know anything about the implementation of the factory or any class that has used factory, we only need to implement/extend our factory.

   Whenever we want to test a specific class that has extended/implemented the factory class, we will write an own test for it. With this flexibility we can easily test every specific class and its required methods.

2. The class diagram for the *BoardFactory* can be found in **Fig. 2**.

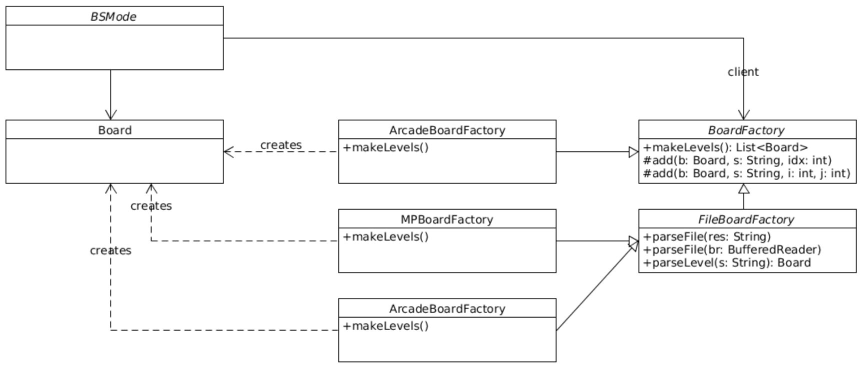

Fig. 3: BoardFactory class diagram

3. The sequence diagram for the *BoardFactory* can be found in **Fig. 4**.

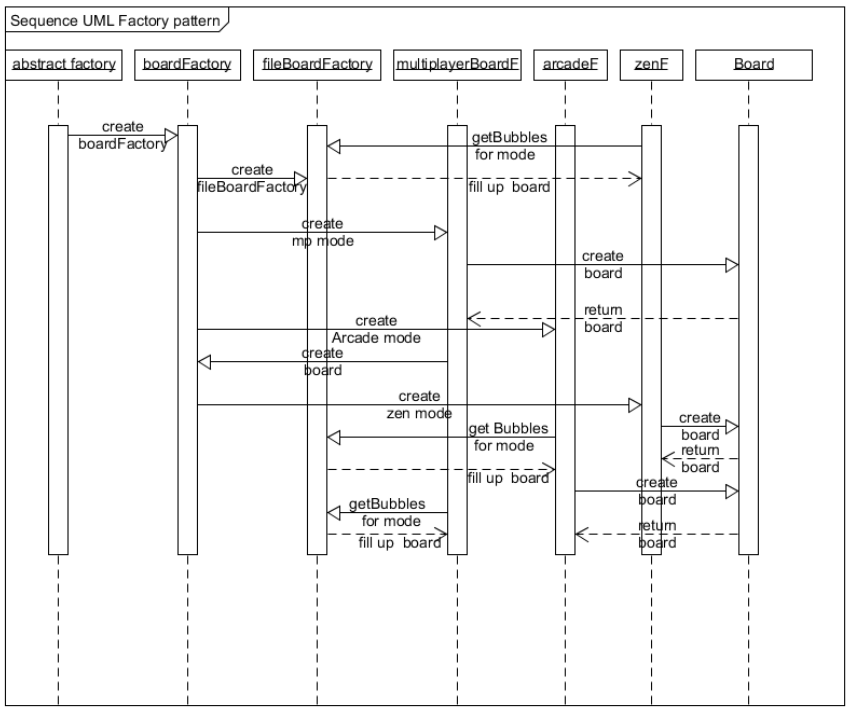

Fig. 4: BoardFactory sequence diagram

2.2 | Pattern 2: Iterator pattern for levels

1. We needed a way to traverse through our data without any need to know any underlying code/implementation.

   We chose for the iterator pattern, as this pattern is specifically designed for iterating/traversing through any collection of data. The iterator pattern is a behavioral pattern. It is for security reasons very important that though there is a way to access our elements/levels, they are not exposed. Their structure/code should remain blocked.

   This basically outlines the idea of the iterator pattern. The idea was that there was a pattern, which takes responsibility of how to access and pass through the collection, by traversing through the objects. Then the iterator gets the object and presents it while still knowing where it left, so it can go on to the next, or even previous. It keeps track of what elements are next to be iterated and which is the current one.

2. The class diagram for the *Iterator pattern* can be found in **Fig. 5**.

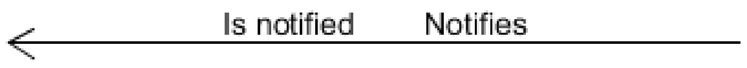

Fig. 5: Iterator class diagram

3. The sequence diagram for the *iterator* can be found in **Fig. 6**.

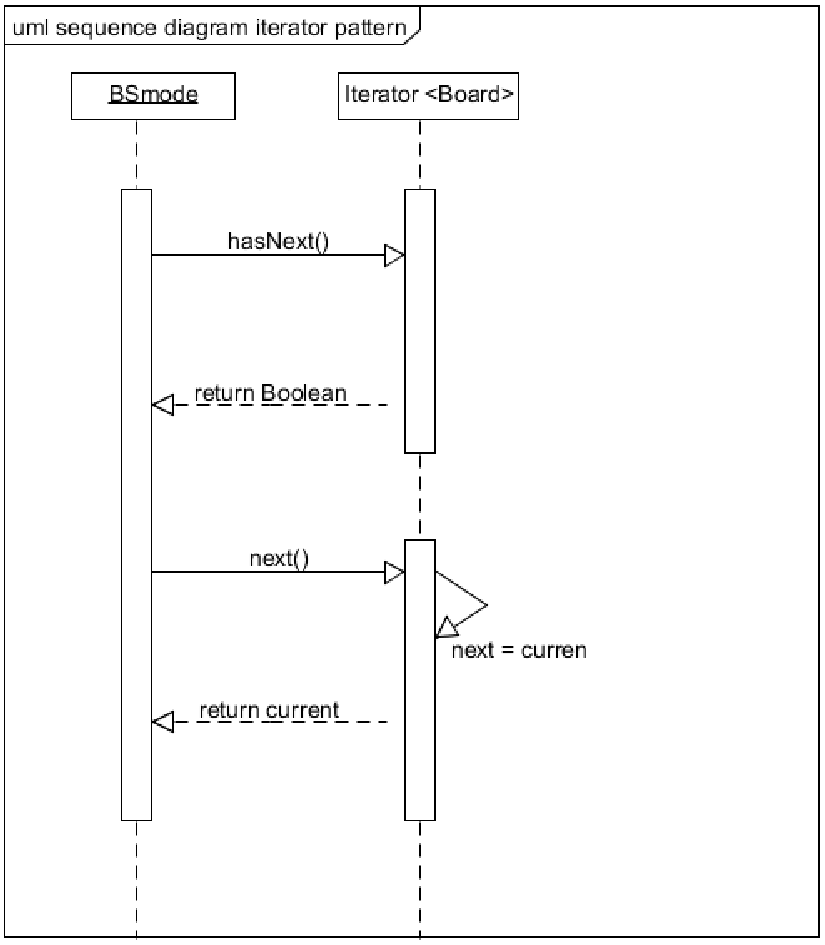

Fig. 6: Iterator sequence diagram

Exercise 3: Optional - One more design pattern

In this section an additional design pattern is implemented and elaborated.

3.1 | Pattern 3: Decorator pattern for EndingCondition

1. Using the *Decorator pattern*, we can easily combine different *EndingConditions*. If we want to have a new condition, we just extend from *EndingDecorator* and specify the condition, with a call to the wrapped *EndingCondition*.
   When check is called on the *EndingDecorator*, it does what it should do and then calls its wrapped object, by means of polymorphism. This happens until the base case is reached, namely the *BasicCondition*. This is the case for all methods of *EndingDecorator*(*lost, won* and *check*).

2. The class diagram for how we applied the *Decorator pattern* can be found in **Fig. 7**.

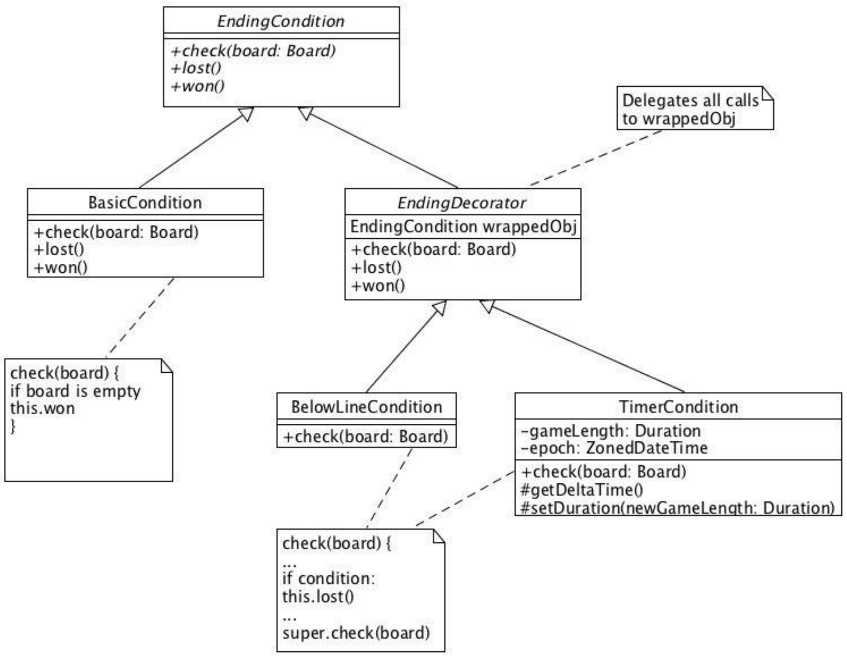

Fig. 7: *EndingCondition* and *EndingDecorator* class diagram

3. The sequence diagram for the *Decorator pattern applied to the EndingDecorator* can be found in **Fig. 8**.

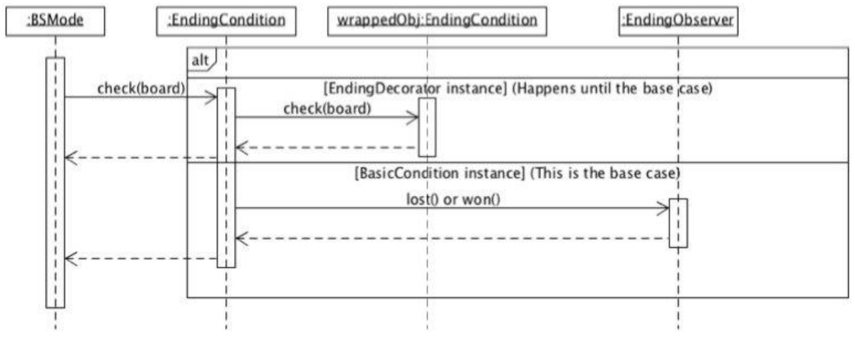
:BSMode
:EndingCondition
wrappedObj:EndingCondition
:EndingObserver
alt
check(board)
[EndingDecorator instance] (Happens until the base case)
check(board)
[BasicCondition instance] (This is the base case)
lost() or won()

Fig. 8: Decorator sequence diagram