TI2206 Software Engineering
Bubble Shooter report: Assignment 4
EEMCS/EWI

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

October 19, 2014

Exercise 1: 20-Time, Reloaded

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

Responsibility Driven Design applied to high scores

In (probably) almost any game, containing an arcade-mode, there should be high scores. It is always fun to be able to compete against yourself or other people! So this also meant that a choice to implement high scores for the Bubble Shooter of Doom was made. In order to come up with a good solution to implement this feature, thinking what the user would like to see is of the utmost importance. Questions that arose were for instance:

1. Are high scores displayed after a game ends?

2. Are they displayed in a seperate screen somewhere?

The decision to put the high scores in a seperate screen, accessible through the main menu and viewing the high scores after the game ends as well (the high score screen), was made through unanimous voting. This also meant that the use of the *AbstractScreen* could be easily used to create a new *HighScoreScreen*. Including an extra button to this *HighScoreScreen* in the *MainMenuScreen* was also not a big issue.
So after an arcade-game ends, a score should be recorded if it is a new high score. A quick and good solution was to simply save the score in a file (again, only if it is a new high score) So something in order to read and write to a file had to be introduced. The latter for viewing the high scores in the seperate screen dedicated to all the arcade-mode high scores.
To distribute the various new responsibilities in a good manner, design patterns could be used to achieve this. How this is exactly done can be read in the next paragraph.

Bubbleshooter high scores using design patterns

To give a player a reason to keep on playing, trying to best the previous score achieved, high scores are an easy incentive. This means that after an arcade-game ends a score is recorded into a file, if the new score is amongst the top 20 scores previously achieved. Additionally viewing back the old high scores is also something the player may want to do. Implementing this required a few new functionalities to be added:

1. Writing to a file.
2. Reading from a file.
3. A screen displaying all the high scores from the file.

To implement this, the *Iterator* pattern was chosen. This was particularly useful because it is only necessary to iterate through the various high scores without having to know the underlying implementation.

This Iterator was implemented through using a NavigableSet as a data structure. We chose this so that our data is already sorted, but by using a NavigableSet we have some extra flexibility compared with for example a SortedSet. A Navigable set will give you the ease to navigate through the highscores in ascending or descending order. An iterator is then used to iterate through the order.

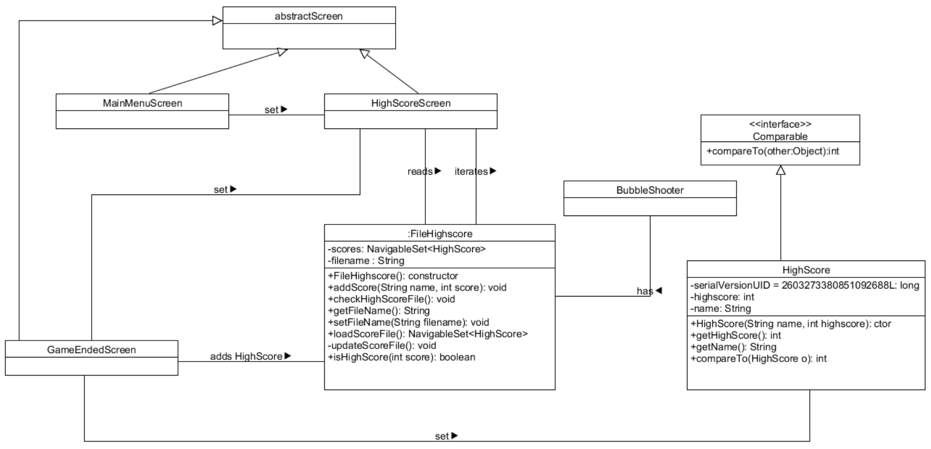

abstractScreen
MainMenuScreen
set ▶
HighScoreScreen
<<interface>>
Comparable
+compareTo(other:Object):int
reads ▶
iterates ▶
set ▶
BubbleShooter
:FileHighscore
-scores: NavigableSet<HighScore>
-filename : String
+FileHighscore(): constructor
+addScore(String name, int score): void
+checkHighScoreFile(): void
+getFileName(): String
+setFileName(String filename): void
+loadScoreFile(): NavigableSet<HighScore>
-updateScoreFile(): void
+isHighScore(int score): boolean
has ◀
HighScore
-serialVersionUID = 2603273380851092688L: long
-highscore: int
-name: String
+HighScore(String name, int highscore): ctor
+getHighScore(): int
+getName(): String
+compareTo(HighScore o): int
GameEndedScreen
adds HighScore ▶
set ▶

Fig. 1: Iterator class diagram for high scores

Responsibility Driven Design applied to chain behaviour with special bubbles
Before explaining in depth what new responsibilities arise and what other things should be taken into account when adding this feature the following was discussed:

'What is meant by chain behaviour?'

Previously special bubbles were introduced, and one in particular had a rather interesting behaviour: the *BomBubble*. This bubble removed all bubbles adjacent to the *BomBubble*. So, what should happen when a *BomBubble* is adjacent to another *BomBubble*?
This is what *chain behaviour* is all about: chaining certain behaviour triggered by behaviour of another *Bubble*. This feature is not only interesting for the current *BomBubble*, but may also be applied to future additions of special bubbles.
Since this chaining behaviour relates to the different kinds of *BubbleBehaviours*, this new feature does not come with new classes. This is because the structure of our *Bubble* and *BubbleBehaviour* classes did not really change, so adding this chaining behaviour can be easily done in the existing classes.
This also means that the previously applied design pattern still works: the *Strategy* pattern. How this feature is added can be read in the next paragraph.

Adding the ability to chain behaviour with special bubbles
Chaining behaviour is currently only applicable to the *BomBubble*. This means that a chain reaction can currently only be caused by a *BomBubble*. This means that other *BomBubbles* or *MichaelBayBubbles* may also be triggered when caught in the blast of a *BomBubble*.
While implementing the chaining of behaviour, we encountered a lot of difficulties. These difficulties will be described in the next part of this report. Because of how the *Board* and *Grid* classes were implemented, we could not easily chain behaviour without making radical changes.
Because implementing chaining without refactoring was virtually impossible, the class diagram for this chaining behaviour will be depicted later on in **Fig. 2**.

Refactoring Board and depending classes to facilitate chaining
Traversing the *Bubbles*s on the *Board* was relatively difficult: you had to pass around indices of the cells on the *Board*, and to traverse the *Bubbles*s outside of the *Board* class, you always needed a reference to the Board.
This was not really a problem, until we started to add *BubbleBehaviour* classes and chain behaviour. Therefore we decided to completely refactor the *Board* and *Grid* classes, and to partially refactor the *BubbleBehaviour* class (the overall structure remained the same).
We only wanted to use the cell indices inside of the *Grid* class, and nowhere else. We also wanted to be able to traverse and remove bubbles, without having to use the *Grid* class.
To achieve this, we decided to:

- Create a new class called *GridCell*, which will hold references to it's neighbors.
- Remove the *Board* class, and port methods from *Board* over to the *Grid* class.

The idea is as follows: A *GridCell* may or may not contain a *Bubble*. You can think of a *GridCell* as a wrapper around *Bubble*. All methods that are called, only have an effect if there's a *Bubble*. If there's no bubble (i.e. it's `null`), nothing happens. Because of this, we also only have to set the *Bubble* field to `null` to mark the *Bubble* as absent. The *GridCell* class also contains its neighboring *GridCell*s. This is what allows us to traverse the *Bubbles*s, without needing the *Grid* (or previously *Board*). We also don't need indices anymore to traverse the *Bubbles*s, because we already have the neighbors.
See **Fig. 2** for the class diagram after refactoring the *Board* class and implementing *chaining*t functionality.

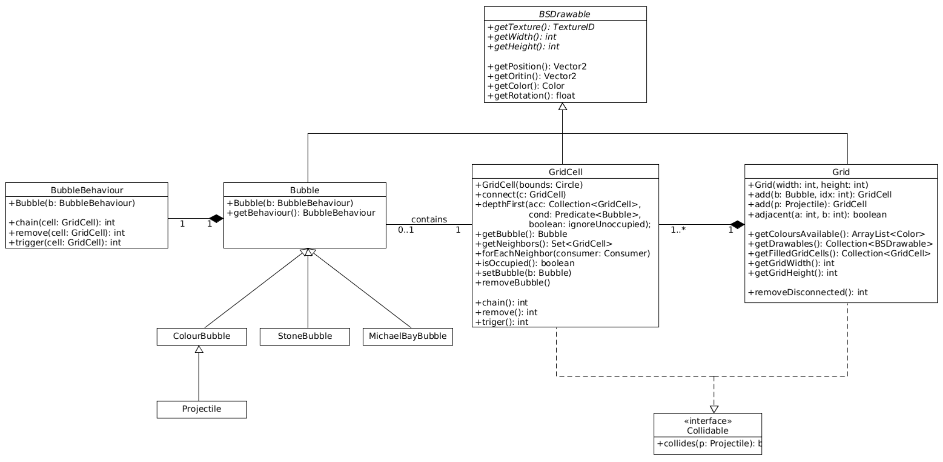
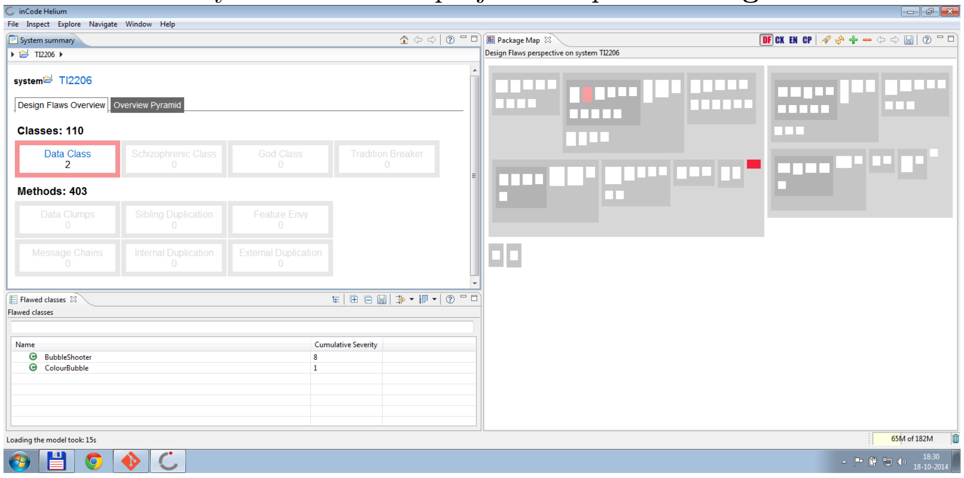

BSDrawable
+getTexture(): TextureID
+getWidth(): int
+getHeight(): int
+getPosition(): Vector2
+getOritin(): Vector2
+getColor(): Color
+getRotation(): float
BubbleBehaviour
+Bubble(b: BubbleBehaviour)
+chain(cell: GridCell): int
+remove(cell: GridCell): int
+trigger(cell: GridCell): int
1
1
Bubble
+Bubble(b: BubbleBehaviour)
+getBehaviour(): BubbleBehaviour
contains
0..1
1
GridCell
+GridCell(bounds: Circle)
+connect(c: GridCell)
+depthFirst(acc: Collection<GridCell>, cond: Predicate<Bubble>, boolean: ignoreUnoccupied);
+getBubble(): Bubble
+getNeighbors(): Set<GridCell>
+forEachNeighbor(consumer: Consumer)
+isOccupied(): boolean
+setBubble(b: Bubble)
+removeBubble()
+chain(): int
+remove(): int
+triger(): int
1..*
1
Grid
+Grid(width: int, height: int)
+add(b: Bubble, idx: int): GridCell
+add(p: Projectile): GridCell
+adjacent(a: int, b: int): boolean
+getColoursAvailable(): ArrayList<Color>
+getDrawables(): Collection<BSDrawable>
+getFilledGridCells(): Collection<GridCell>
+getGridWidth(): int
+getGridHeight(): int
+removeDisconnected(): int
ColourBubble
StoneBubble
MichaelBayBubble
Projectile
«interface»
Collidable
+collides(p: Projectile): b

Fig. 2: Board refactoring and chaining class diagram

Exercise 2: Software Metrics

In this section the inCode tool is used to assess the software metrics on the entire project. Additionally a fully in depth elaboration on what design flaws and why they were found is written in this section.

Using the inCode tool for evaluating software metrics:
The inCode analysis of the entire project is depicted in **Fig. 3**.



Classes: 110

Data Class 2	Schizophrenic Class 0	God Class 0	Tradition Breaker 0

Methods: 403

Data Clumps 0	Sibling Duplication 0	Feature Envy 0
Message Chains 0	Internal Duplication 0	External Duplication 0

Flawed classes

Name	Cumulative Severity
BubbleShooter	8
ColourBubble	1

Loading the model took: 15s

Fig. 3: Design flaws detected by inCode.

As it turns out, inCode detects two design flaws, both of which are a *Data Class* problem.

2.1 | Design flaw 1: *BubbleShooter* is a *Data Class*

1. Explain: design choices or errors leading to the detected design flaw.

   This detected design flaw is actually one we were already aware of. Since the beginning of the game, BubbleShooter has held a lot of common objects that are shared among the classes. This facilitated a single *SoundEngine* and a single *Assets* to be used for the whole game.

2. Fix the design flaw **or** explain why it's not a flaw; do not change it.

   Step 1: During a previous sprint the *Singleton Pattern* was already adopted for the *Assets* class. However, the unique *Assets* instance was still stored in *BubbleShooter* as a class member. All classes depending on *Assets* have been rewritten so that they now acquire the unique instance of *Assets* by themselves.

   Step 2: While refactoring the *Assets* class, we decided that the *SoundEngine* was also a good candidate for the *Singleton Pattern*. After implementing *SoundEngine* as a *Singleton* and rewriting all depending classes, the inCode software metrics evaluation was run again. It showed that *BubbleShooter* no longer had a *Data Class* design flaw.

2.2 | Design flaw 2: *ColourBubble* is a *Data Class*

1. Explain: design choices or errors leading to the detected design flaw.

   This design flaw was one we were not aware of. During this sprint the whole *Bubble* class hierarchy has been refactored to take the newly introduced *BubbleBehaviour* in to account. During the refactoring process, the enum *BubbleColours* was introduced. This enum was however defined inside the *ColourBubble* class.

2. Fix the design flaw **or** explain why it's not a flaw; do not change it.

   The design flaw has been fixed by moving *BubbleColours* to it's own file. Since this is a public enum, that has to be accessible to multiple classes, a separate file is indeed a better place for *BubbleColours*.

2.3 | Design flaw 3: *Bubble* might have been a *God Class*

1. Consider another design flaw that inCode could detect, and explain where it could have affected your system and how you managed to avoid it.

   One of the design flaws that inCode might have detected is a *God Class*, for example in *Bubble*. Basically this means that one class would have too many responsibilities as a result of 'doing' and 'knowing' too much. If all the 'special bubbles' and all the corresponding *Behaviours* were also put inside one class (for example: *Bubble*) it would mean that *Bubble* would become enormous. It would bear all the responsibilities of all the different kinds of special bubbles including the behaviour of these bubbles. A design flaw like this can be easily evaded, by breaking down a big problem down into smaller problems; smaller classes with relevant knowledge about only itself. This meant that seperate classes for *Behaviours* and *Type of Bubbles* were made. Additionally, the use of the *Strategy* pattern also helped preventing a *God Class* from happening.