Assignment #4

Week 7

Group 2 - Bubble Bobble

Remi Flinterman
4362950, remiflint@live.nl

Arthur Guijt
4377338, a.guijt@student.tudelft.nl

Laurens Kroesen
4350286, l.kroesen@student.tudelft.nl

Thomas Overklift
4080890, t.a.r.overkliftvaupelklein@student.tudelft.nl

Lars Stegman
4365801, l.s.stegman@student.tudelft.nl

TI2206 Software Engineering Methods
of the Computer Science curriculum
at the Delft University of Technology.

Supervisor: Dr. A. Bacchelli
Teaching Assistant: Aaron Ang

Table of Contents

Exercise 1 – Your wish is my command, Reloaded

Part A – High score indicator in the UI

This week our TA decided that we need a high score to be displayed in the UI somewhere. There should preferably separate high scores for different players. The requirements for this feature are stated below.

1 – Requirements

1.1 Must Haves

- The UI shall have a field where a number representing the current score is displayed.
- The score shall start at 0 points for each player.
- The score shall increase by 50 point each time a player kills an enemy.

1.2 Should Haves

- -

1.3 Could Haves

- The UI shall have separate score indicators for different players.

1.4 Won't Haves

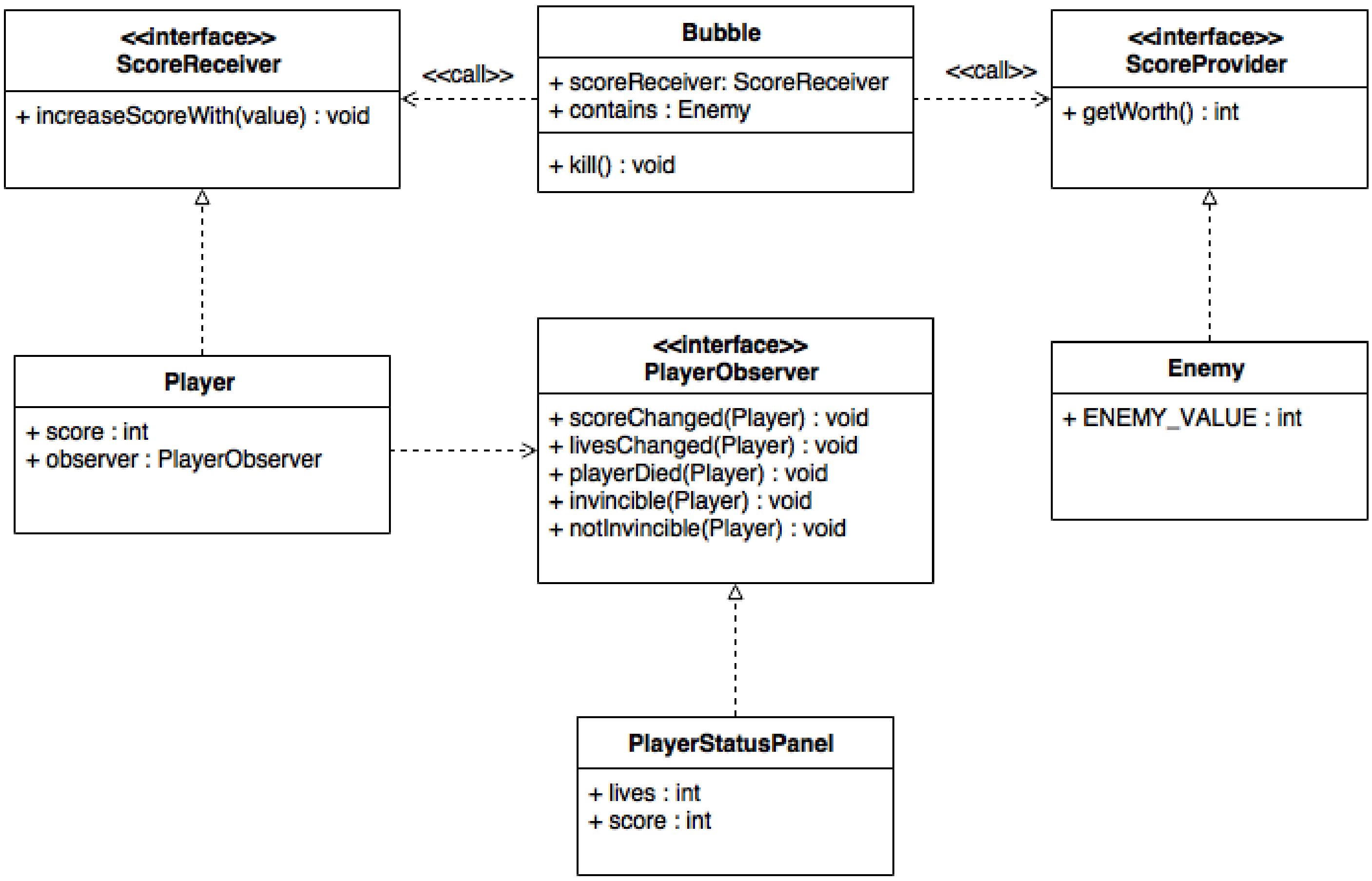
- The UI shall have an indicator of score in the playing field.

2 – RDD, UML & explanation

The UI has the responsibility to display the information of the game, for instance the number of lives the player has left. The player on the other hand has the responsibility to notify the UI of any changes. Bubbles have the responsibility to get the worth of the enemy it contains when it dies and adding the points to the score of the player. To update the player's score when an enemy is killed, we created two interfaces, ScoreProvider and ScoreReceiver. When a player creates a bubble, it sets itself as the score receiver of that bubble so that, when the bubble is popped while it contains an enemy, the score can be added to the player.

When a bubble is popped it checks whether it contains a ScoreProvider, if it does it gets the score and then adds it to the score receiver, i.e., the player.
When the player receives more points it adds it to its score and notifies its observers of the change. It also notifies its observers when the remaining number of lives change. When the PlayerStatusPanel receives the notification it gets the changed value from the player and updates the display.

Below there is a UML diagram that displays the relation between captured enemies and the high score count.



<<interface>> ScoreReceiver
+ increaseScoreWith(value) : void
<<call>>
Bubble
+ scoreReceiver: ScoreReceiver
+ contains : Enemy
+ kill() : void
<<call>>
<<interface>> ScoreProvider
+ getWorth() : int
Player
+ score : int
+ observer : PlayerObserver
<<interface>> PlayerObserver
+ scoreChanged(Player) : void
+ livesChanged(Player) : void
+ playerDied(Player) : void
+ invincible(Player) : void
+ notInvincible(Player) : void
Enemy
+ ENEMY_VALUE : int
PlayerStatusPanel
+ lives : int
+ score : int

Part B – A life indicator for the player

This week our TA decided that we need a life indicator to be displayed in the UI somewhere. There should preferably separate indicators for different players. The requirements for this feature are stated below.

1 – Requirements

1.1 Must Haves

- The UI shall have a label that shows the current amount of lives.
- The current lives shall be represented by heart icons.

1.2 Should Haves

- The UI shall have separate Labels for different players showing their amount of lives.

1.3 Could Haves

- -

1.4 Won't Haves

- The UI shall have a blinking heart icon when the player gets hit.

2 – RDD, UML & explanation

The explanation at part A covers the 'player lives' mechanic as well: a player starts with 3 lives. When the player uses a life, the UI is notified, and the amount of remaining lives is adjusted accordingly. We decided to represent the amount of lives a player has by three heart icons: a red heart for a full life a gray heart for a depleted life.

An additional feature for the lives display that we still want to implement is the 'blinking' of a heart icon when a player loses a life. This might better convey that the player is invulnerable for a short time after it loses a life, a feature which is already in the game! Unfortunately we weren't able to implement this feature yet.

Part C – Local multiplayer

This week our TA decided that we need local multiplayer to be added to the game. There should be unique key bindings for both player characters. The requirements for this feature are stated below.

1 – Requirements

1.1 Must Haves

- The game shall have multiplayer support.
- Multiple players shall be controllable with different key presses.
- Players shall be controllable simultaneously.

1.2 Should Haves

- A player's death shall have no effect on other players.
- The game shall be lost when all players are out of lives.

1.3 Could Haves

- Each player shall have their own spawning point.
- Each player shall have it's own score.
- Each player shall have a unique sprite.

1.4 Won't Haves

- The game shall have online multiplayer.

2 – RDD, UML & explanation

Most of the code that we wrote before was written with extensibility in mind, this made adding multiplayer relatively easier than expected. However, there were issues related to the display of score and lives (the display panel got overwritten), but those problems have been resolved (see part A & B for some more about lives and scores).

There is one unresolved issue regarding the AI performance though. The AI has been implemented to chase the first player, so the second player doesn't get chased at all. In the future we want the enemies to pick a player to chase and then chase that player. This can be evenly divided or maybe even dynamically decided taking the positions of players and enemies into account. The problem turned out to be a bit tricky too fix just yet, so for now player 1 will get chased a lot!

Exercise 2 – Software Metrics

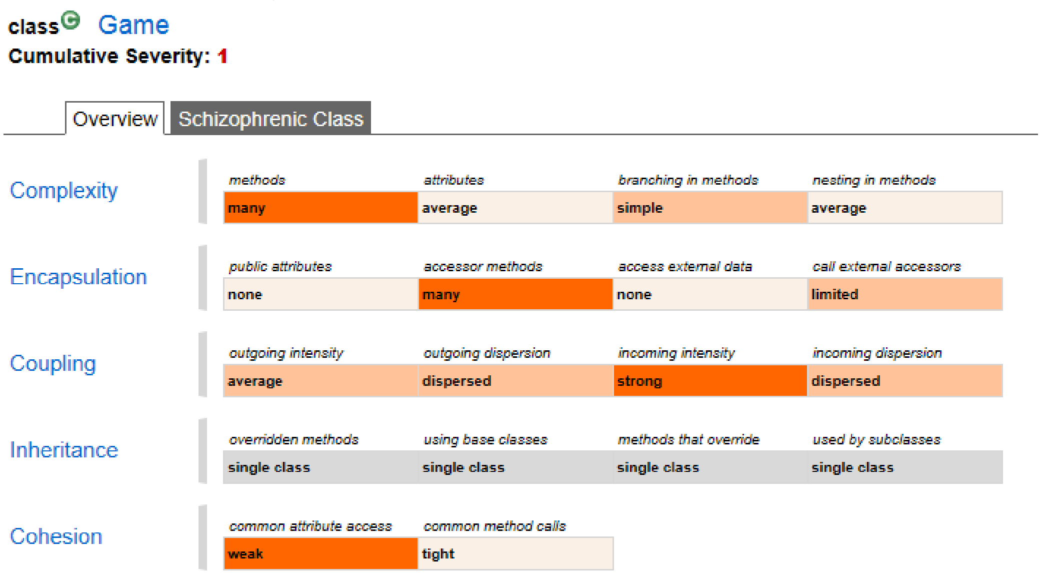
1 – Analysis location

We committed the analysis files we created to the documentation folder of our git repository. There's a subfolder InCode that contains all the result files.

2 – Examining the design flaws

The part below isn't clearly divided into three design flaws that have been fixed.
There are a few reasons for this, firstly: there were "seven" design flaws according to InCode, and we decided to address all of those one way or the other. Secondly, most of the design flaws were easily resolved, and we even decided to leave most flaws be; we of course motivated how and why below. Therefore we decided we might need to do a bit of extra work. Thus we also analyzed some other statistics InCode came up with, and some things we think the tool could've analyzed differently or better in some way.

InCode: "Game is schizophrenic"



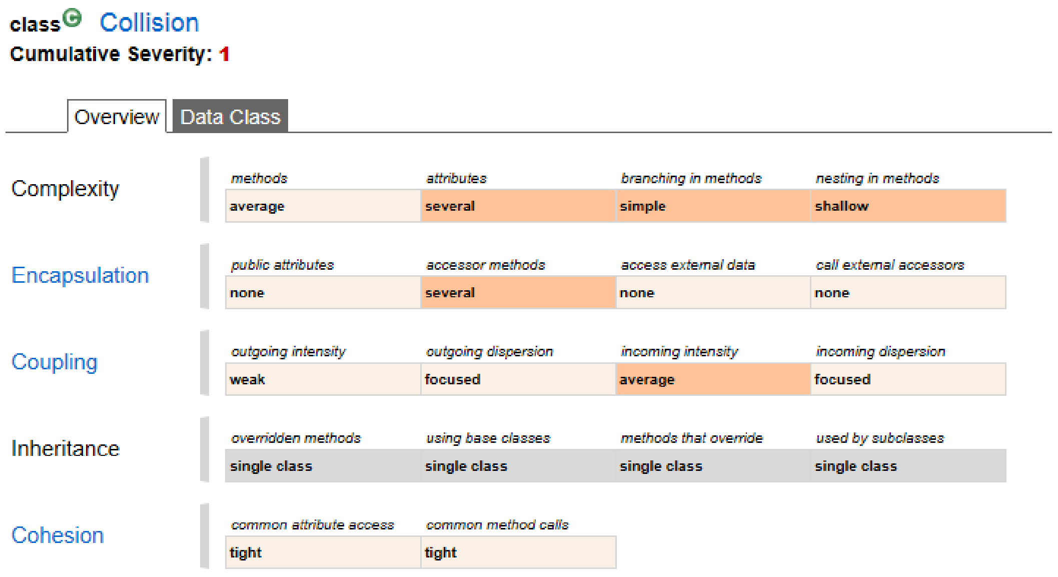
class Game
Cumulative Severity: 1

Overview | Schizophrenic Class

Complexity	*methods* many	*attributes* average	*branching in methods* simple	*nesting in methods* average
Encapsulation	*public attributes* none	*accessor methods* many	*access external data* none	*call external accessors* limited
Coupling	*outgoing intensity* average	*outgoing dispersion* dispersed	*incoming intensity* strong	*incoming dispersion* dispersed
Inheritance	*overridden methods* single class	*using base classes* single class	*methods that override* single class	*used by subclasses* single class
Cohesion	*common attribute access* weak	*common method calls* tight		

According to InCode Game is a schizophrenic class. We looked into this and it seems rather debatable: there is some functionality in the game class that switches levels or gives the "game over". These methods are indeed called from several different places, but it seems like overkill to split this functionality into different classes.
InCode also incorrectly claims that the method getPlayers is never used, but it is used in the Launcher. We're not sure why InCode thinks otherwise.
All things considered we decided not to change the Game class.

InCode: “collision is a data class”



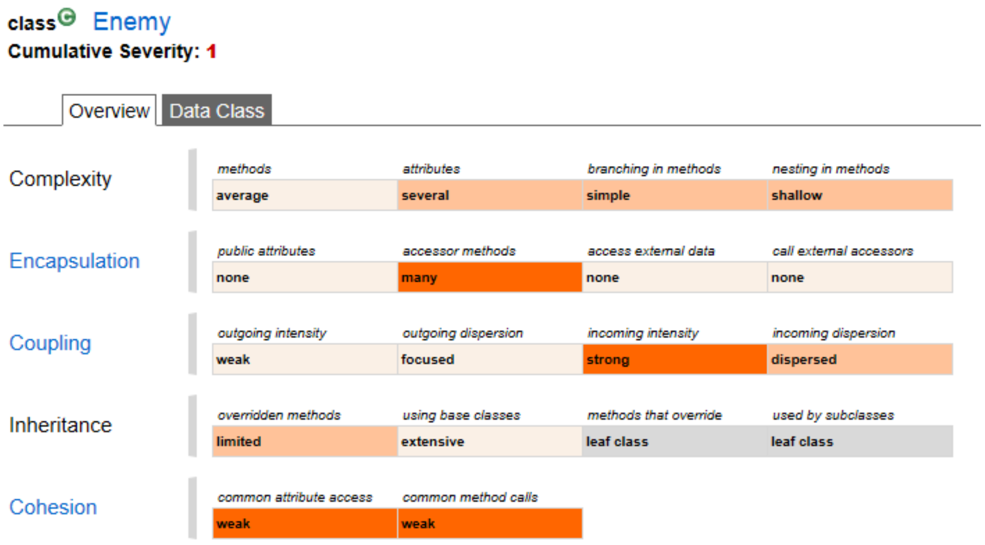
class Collision
Cumulative Severity: 1

Overview | Data Class

Complexity	*methods*	*attributes*	*branching in methods*	*nesting in methods*
	average	several	simple	shallow
Encapsulation	*public attributes*	*accessor methods*	*access external data*	*call external accessors*
	none	several	none	none
Coupling	*outgoing intensity*	*outgoing dispersion*	*incoming intensity*	*incoming dispersion*
	weak	focused	average	focused
Inheritance	*overridden methods*	*using base classes*	*methods that override*	*used by subclasses*
	single class	single class	single class	single class
Cohesion	*common attribute access*	*common method calls*		
	tight	tight		

InCode classifies the class Collision as a data class because it has more than a few accessor methods, and because it is being used more than a few times by another class. This is true because the CollisionComputer class relies on Collision a bit. We’ve introduced this relation because otherwise the class CollisionComputer would become too complicated. It already has a few extensive methods. The Collision type object are used simplify the data the methods in the CollisionComputer have to exchange.
As can be seen in the figure above, all of the concerns raised by InCode are of the least severity, and because we believe this construction improves (at the very least) the maintainability of the collisionComputer, we decided not to change the collision class.

InCode: “enemy is a data class”



class Enemy
Cumulative Severity: 1

Overview | Data Class

Complexity	*methods*	*attributes*	*branching in methods*	*nesting in methods*
	average	several	simple	shallow
Encapsulation	*public attributes*	*accessor methods*	*access external data*	*call external accessors*
	none	many	none	none
Coupling	*outgoing intensity*	*outgoing dispersion*	*incoming intensity*	*incoming dispersion*
	weak	focused	strong	dispersed
Inheritance	*overridden methods*	*using base classes*	*methods that override*	*used by subclasses*
	limited	extensive	leaf class	leaf class
Cohesion	*common attribute access*	*common method calls*		
	weak	weak		

The class Enemy has the same problems as the class Collision class above. This instance of the ‘data class’ flaw is even slightly more severe, as there are more accessor methods (according to InCode at least). When we start looking into the problem in detail we can see that the problem is (mostly) caused by the getters and setters the AI package uses.

We need those methods because we deem is necessary to store some information about enemies for longer than one tick, and therefore we can’t do that in one of the AI classes. These values improve AI behavior, because this way and enemy knows which side of the map it’s been last and it won’t keep running into a wall or something. Additionally we need to remember a random seed because otherwise it would be reseeded every tick. That would be fun, but the AI would just appear to be laggy and random movements would cause stutter. Refactoring this somehow would be quite difficult, because all the values are tied to specific enemies and the AI doesn't have any long term memory of it's own at this point. Any attempt to move this data from the Enemy class into another class would simply result in that class becoming a data class at this point.

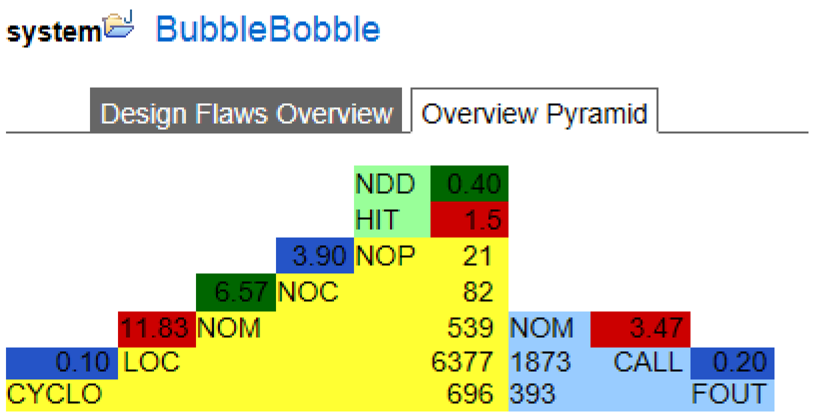
So because we think the data stored in the Enemy class is important and because we cannot change this easily (without impacting the AI behavior negatively) we decided to ignore this supposed design flaw and leave it be.

InCode: "The draw method is a data cluster"

According to the InCode tool four of the "draw" methods formed data clusters because too much information was being passed along. This might have been the case because we took a scalable UI into account. The draw function used quite a few values, apart from the x & y coordinate of the sprite. It is possible to change the width and height of the grid and the sprites would be scaled to compensate for that. We created a separate branch on GitHub that changed this, and that fixed the cluster according to InCode. We discussed this change in the group, but we decided against it, since we didn't want to give up extensibility of the code just to please the tool.

Analyzing the metrics

InCode found seven flaws in the system. Does that mean it was poorly designed? Probably not. The structure of the system has been pretty sound right from the start, and it has been improving over the last couple of weeks, ecspecially since the introduction of several design. On top of that, all of the design flaws had a cummulative severity of 1. That is the least severe kind of flaw.
Therefore it is safe to conclude that the flaws in the system have been minimal.



```
                     NDD  0.40
                     HIT  1.5
                3.90 NOP    21
           6.57 NOC         82
     11.83 NOM             539 NOM     3.47
0.10 LOC                  6377 1873    CALL 0.20
CYCLO                      696 393          FOUT
```

This all means that the system is in a pretty good shape right now. But how can we ensure that it stays that way in the future? Apart from sticking to good practices that have been taught, it might be wise to take a look at the so called "overview pyramid" of the system. In the overview pyramid all kinds of software metrics are gathered together and the relations between some of those metrics are displayed within the colored boxes. Green is below average, blue is average and red is above average. These averages have been taken from a large sample of Java systems that serve as a benchmark. When we look at our system we can see two things that jump out. The LOC per method (average is 10), and the external calls per method (coupling). Both of these metrics are above average in our system. This is not critical at this point but we must consider splitting methods and relocating data when new features are implemented in one of the affected classes.

Analyzing InCode

InCode is a pretty neat and easy to use tool that can give useful feedback in just a few seconds. But it can still be improved. We found it rather interesting that the tool had quite a few things to say about "data classes" and "data clusters", but hasn't mentioned the class GameConstants in the system summary. To us that class would seems like a huge data class that holds all kinds of public values that are used throughout the system.
Apart from that the tool is great for improving maintainability and reliability of a system, but it isn't really able to say anything about the performance of a system. InCode gains its information from metrics taken from the structure of a system, this makes it hard to say anything specific about the inner workings of a system. This is a feature that could be looked into in the future.

We made player schizophrenic!

When we reviewed or system with InCode once more at the end of the sprint we noticed we created an additional problem. Our changes made the player class into a schizophrenic one. While there are some things that could arguably be bad, fixing them at would require more work and analysis than we were capable of doing this sprint. Additionally when looking at the report InCode gave, it stated that the Player class has 7(!) unused services among which the tick method. Since tick is the heartbeat of the game this is absolute rubbish. At this point we are seriously questioning the reliability of InCode as a whole. It seems there may be more amiss with this tool than previously suspected.