Assignment #1

Week 3

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 - The Core

1

For a complete overview of the hierarchy we created with CRC, see Appendix A.

Creating our CRC cards
We started out by creating a class for board and by subsequently adding relations between board, grid and unit. Grid was then further expanded into cells. For unit we added three subclasses: Player, Enemy and Bubble. There are also relations between Unit and Direction, and between Unit and Point. These relations are relevant for storing information about Units.
Then we started thinking about the ability to draw everything. We created an UI (subclass of JFrame) and we gave the UI the responsibility of displaying the Board. We then gave Board and all the related classes (described above) the ability to draw themselves.
We then wanted to be able to control the player, so we added a Keylistener class. Player and the UI use this class for movement and button presses respectively.
Finally we realized that we needed a class to handle collisions that might appear. The CollisionHandler uses the location information from unit and information about other units or the grid, regarding what type of collision is being checked.

Differences with the actual implementation.
The system we created using RDD is surprisingly similar to the one we implemented. There are however a few differences:

- In the implementation there is no Board class; this class was not needed because the Grid and Units are immediately loaded into the level.
- The level can draw the grid.
- Movements have been relocated to an entirely different class.
- The grid and the screen coordinates are different in the actual implementation. A class has been added to deal with this.
- Sprites are much more important in the actual implementation than can be derived from our CRC cards. On the cards we only mentioned saving sprites as a responsibility of Unit.
- The implemented UI is also quite a bit more extensive than we one created using the CRC cards.

2

We stated the classes we deem most important and their key responsibilities and/or collaborations below.

- Level
  - The level contains all Units and the entire grid.
- Game
  - Contains the level.
  - Loads new levels.
  - “Ticks”, a tick is a clock cycle of the game.
- UI
  - Displays all graphical elements of the game.
  - Listens to key presses.
- PlayingfieldPanel
  - Display the level.
- Unit
  - Unit stores the location and amount of lives of all Units.
  - Unit is the superclass of Player, Bubble and Enemy.
- Mover
  - Governs all movements of all units.
- Collision Handler
  - Interacts with all Units and the Grid to be able to detect collisions.

3

So one of the less important classes are for instance the subclasses of Unit (Player, Bubble & Enemy). All of the information about those is stored in the Unit class. These classes are still needed though because they have unique sprites and because collisions between the types have different outcomes. The player subclass also has different movement logic than the other unit types.

There is also a class gameConstants. This class is technically not required because all constants could be saved in the classes where they are needed. It is however very useful to have a single class where all constants are saved. This allows for easy manipulation of the game's parameters; thus the game can be run under different circumstances with relative ease.

A third example is the UI. We divided the UI into some smaller classes. Because the UI has a lot of different responsibilities, dividing it into multiple smaller classes improves the maintainability of the UI. Additionally this makes it very easy to move elements around in the UI.

4

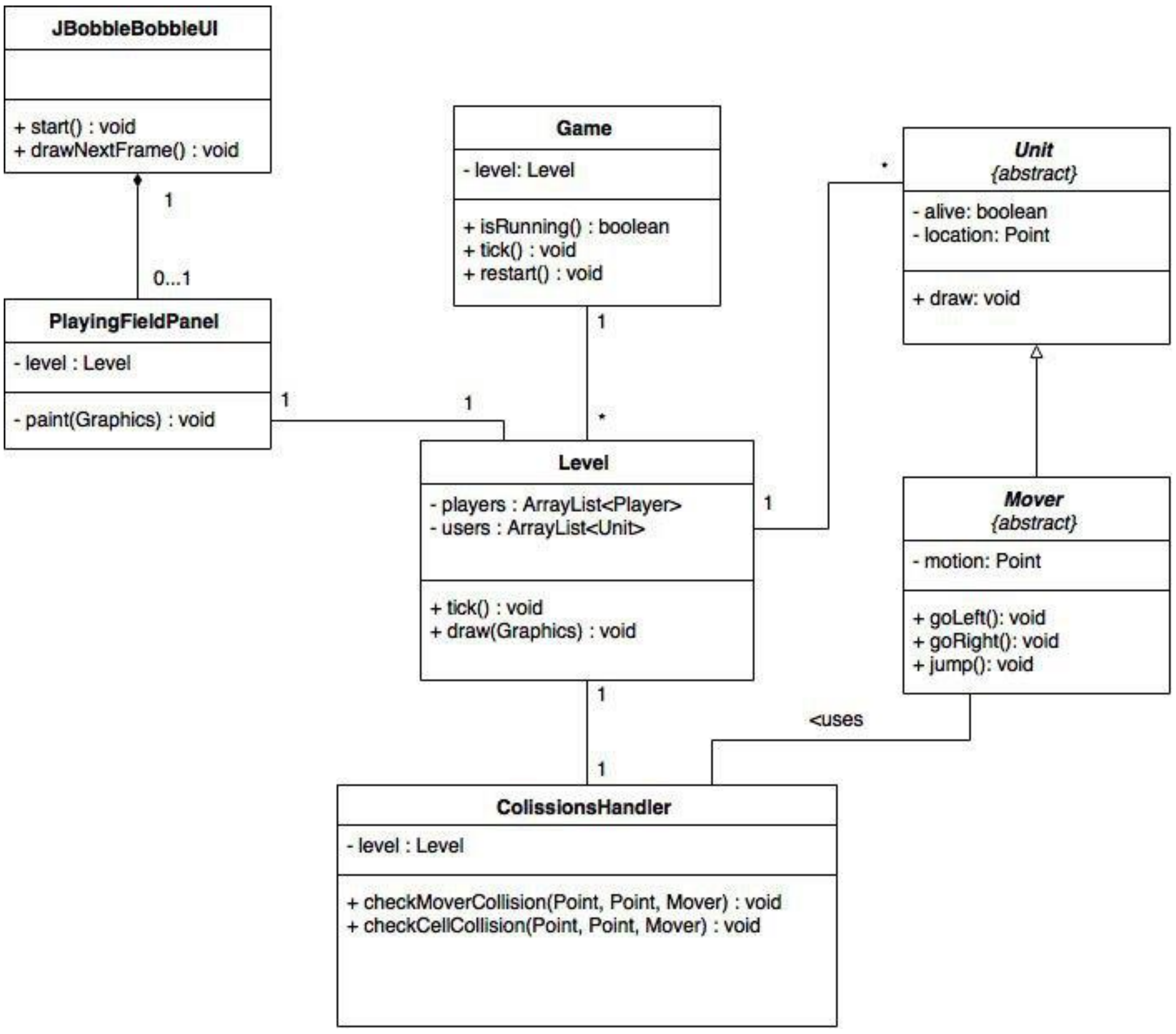


Figure 1: Class diagram of the most important classes.

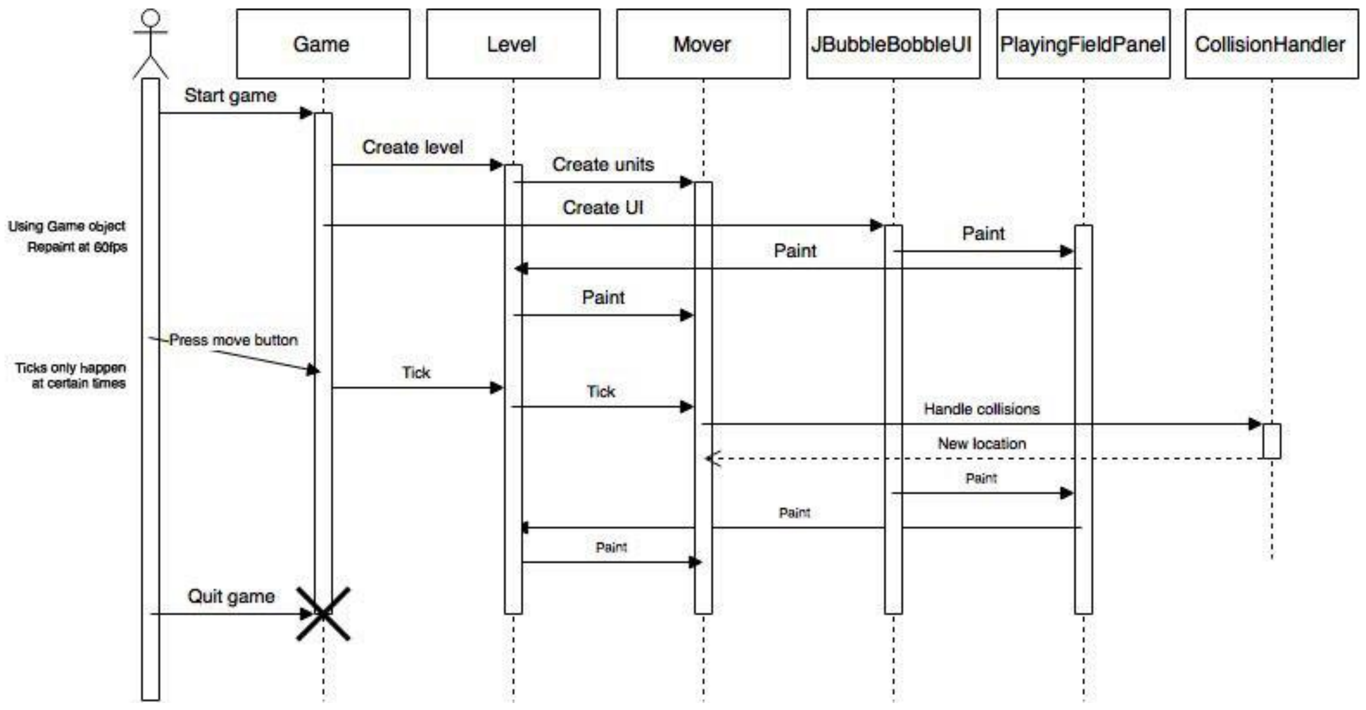


Figure 2: Sequence diagram of interaction between the main elements of the game

Exercise 2 – UML in practice

1

Aggregation vs. composition
Aggregation and composition both indicate parts of a larger 'whole'. The difference however is in the type of relation between parts. In the case of composition one part belongs to one whole; in the case of aggregation parts may be shared. In other words: in composition the part is 'owned', and is of no importance when taken out of the 'whole', in aggregation the part is 'used', and it can be used by other classes as well.

In the slides of lecture 4 (on BlackBoard), there is some further clarification on the matter:
When are the tires of a car an aggregation and when are they a composition?

- If we are running a service center, the tires are not interesting separately from the car, the tires are merely a part of the car's composition.
- If we are running a tire store, the tires are important independently of the car, therefore the relationship would be an aggregation.

Aggregation and composition in our own implementation
We use the class playingfieldPanel as a part the UI, and it is uninteresting to us separately, therefore we see the panel as a composition of the UI. Another example if composition in our project would be the relation between Grid and Cell; the cells are not of any importance separately from the grid.

An example of an aggregated class would be Level. The level is used by the Game class, but it is not without use apart from it: the collisionHandler also uses Level.

2

Parameterization in our project
There is one class in our project that uses parameterization: Grid. We use parameterization for the Grid class so that it can be re-used with different contents. Currently we are only using cells as content of a grid, but in the future we might want to store pickups as a grid with different contents. So the benefit of parameterization would be for instance the ability to reuse a structure with different contents.

3
Hierarchies in our project.

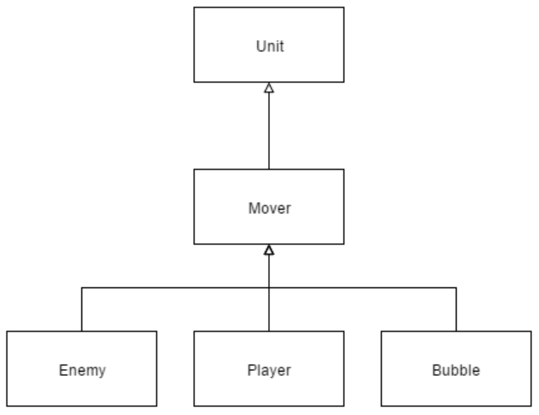


Figure 3: Hierarchy of 'Unit'

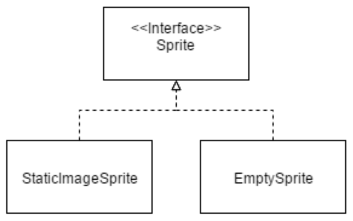


Figure 4: Hierarchy of 'Sprite'

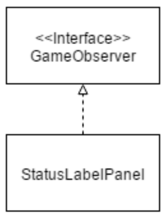


Figure 5: Hierarchy of 'GameObserver'

- 'Unit' *(figure 3)*
  - Type: Is-a.
  - We used this hierarchy because unit is a class that has some functionality, but can't provide all the necessary functionality by itself. The subclasses (Enemy, Player & Bubble) differ significantly from each other in some key aspects. Additionally we created the class Mover that contains all the movement logic. We did this to remove some functionality from unit, to better comply with the RDD ideology that a class should not have too many responsibilities.
- 'Sprite' *(figure 4)*
  - Type: Is-a.
  - We used this hierarchy because there are multiple kinds of sprites that have different functionality. For instance: some sprites are always the same, but others can change when the player moves in a different direction.
- 'GameObserver' *(figure 5)*
  - Type: Polymorphism.
  - We used this hierarchy because any kind of object has to be able to observe the game if it needs to without the game having to know what kind of object it is. Different kinds of objects that observe the game can for instance be: the score panel or a win/lose condition monitor.

Exercise 3 – Simple logging

1

Our requirements for the logger.

- The logger shall log the reading of a level map.
- The logger shall log the parsing of a level map.
- The logger shall log the creation of units.
- The logger shall log the movement of units.
- The logger shall log key presses.
- The logger shall log collisions.
- The logger shall be able to print the logs in the console.
- The logger shall be able to store the logs in a text file.
- The user shall be able create a log text file by pressing a button.

2

For an overview of the hierarchy we created with CRC, see Appendix B.

During the design of the logger we created three separate classes using CRC. First there is the Log class which creates a Map of all actions that are logged. Then there is the Logable class that which enforces that other classes have an addToLog(String action) function. Finally there is the DumpLog class that writes all the logs to a text file.

Below there is a class diagram to further clarify the structure of our logger. The interface Logable is being implemented by almost every other class in the system, to make sure that all important data is being logged. Please note that there are some additional get() and set() methods that have not been incorporated into the class diagram.

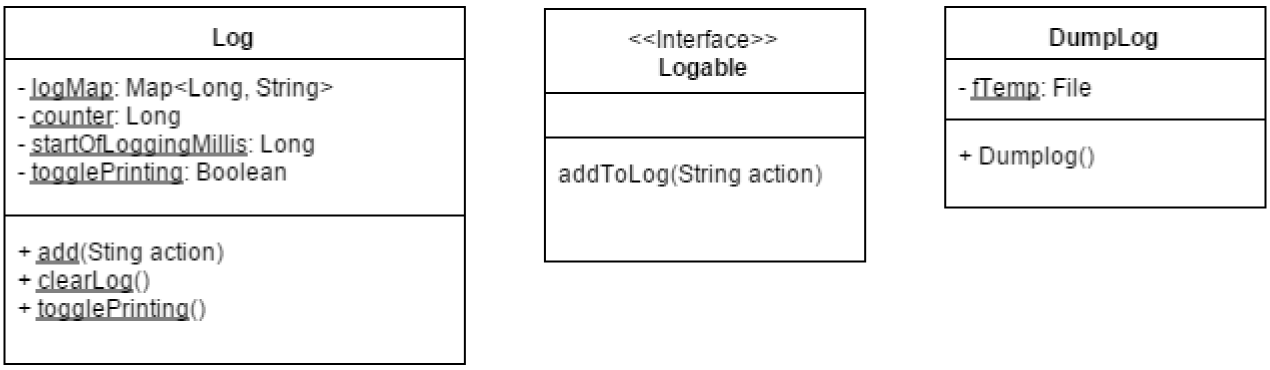
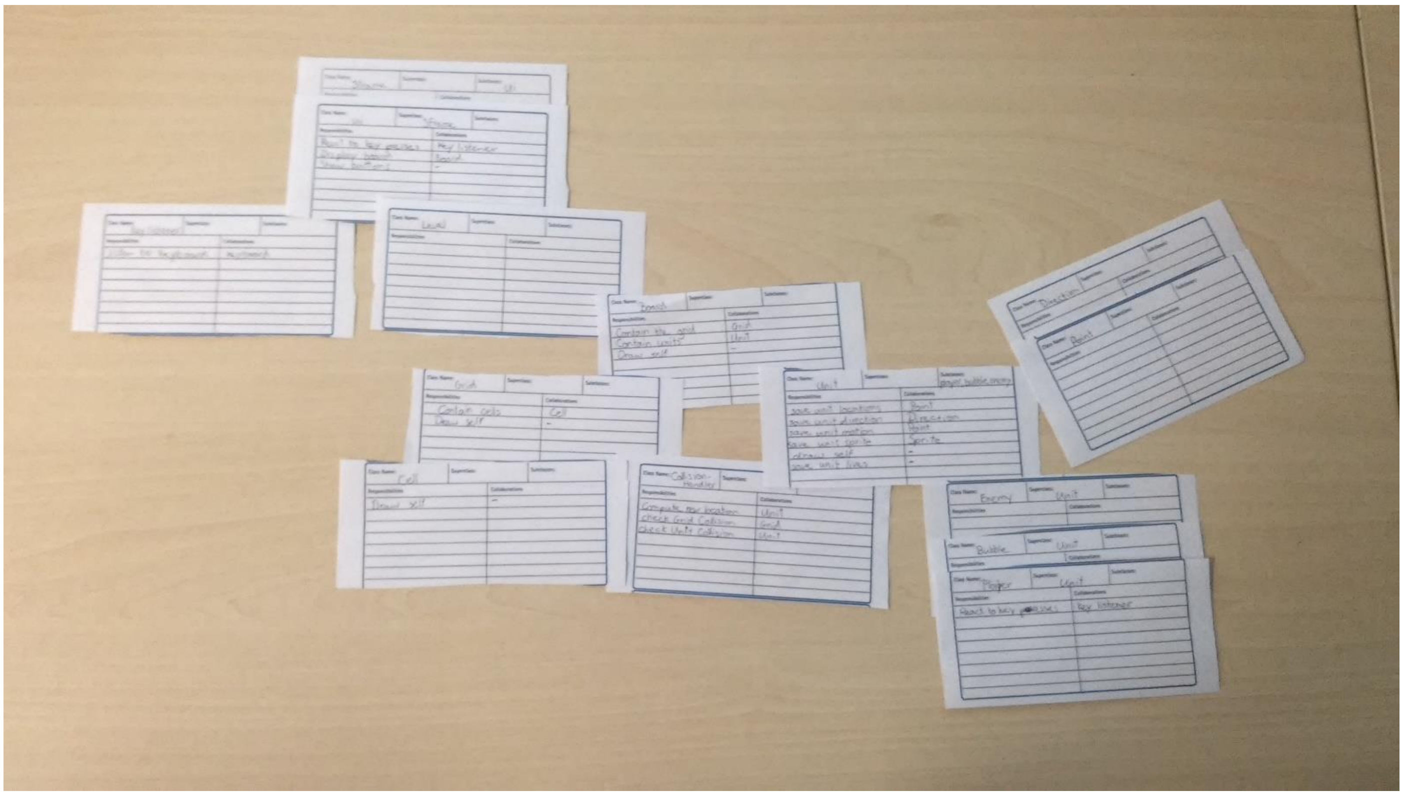


Figure 6: Hierarchy of 'Unit'

All data is added to the logger via the Logable interface. An example: when a player gains points, the method addScore(int points) is called. This method then calls the method addToLog(String action), which has been added there because of the Logable interface. The action is then subsequently added to the log and stored in the logMap. All actions are added to the map in chronological order; a timestamp of when the action took place is also added. The size of the map is being monitored by the counter variable. The entire log can be wiped by the method clearLog(). This method simply empties the map & resets the counter and the printer toggle to the default value. The dumpLog() class is there for saving the logs to a file if needed.

Appendix A: Responsibility Driven Design: CRC (Project)



Class Name:	Superclass:	Subclasses:
JFrame		Ui

Class Name:	Superclass:	Subclasses:
Ui	JFrame	

Responsibilities	Collaborations
React to key presses	Key listener
Display board	Board
Show buttons	-

Class Name:	Superclass:	Subclasses:
Key listener		

Responsibilities	Collaborations
listen to keyboard	keyboard

Class Name:	Superclass:	Subclasses:
Level		

Class Name:	Superclass:	Subclasses:
Board		

Responsibilities	Collaborations
Contain the grid	Grid
Contain units	Unit
Draw self	-

Class Name:	Superclass:	Subclasses:
Grid		

Responsibilities	Collaborations
Contain cells	Cell
Draw self	-

Class Name:	Superclass:	Subclasses:
Cell		

Responsibilities	Collaborations
Draw self	-

Class Name:	Superclass:	Subclasses:
Unit		player bubble enemy

Responsibilities	Collaborations
save unit locations	Point
save unit direction	Direction
save unit motion	Point
save unit sprite	Sprite
draw self	-
save unit lives	-

Class Name:	Superclass:	Subclasses:
Collision-Handler		

Responsibilities	Collaborations
Compute new location	Unit
check Grid Collision	Grid
check Unit Collision	Unit

Class Name:	Superclass:	Subclasses:
Direction		

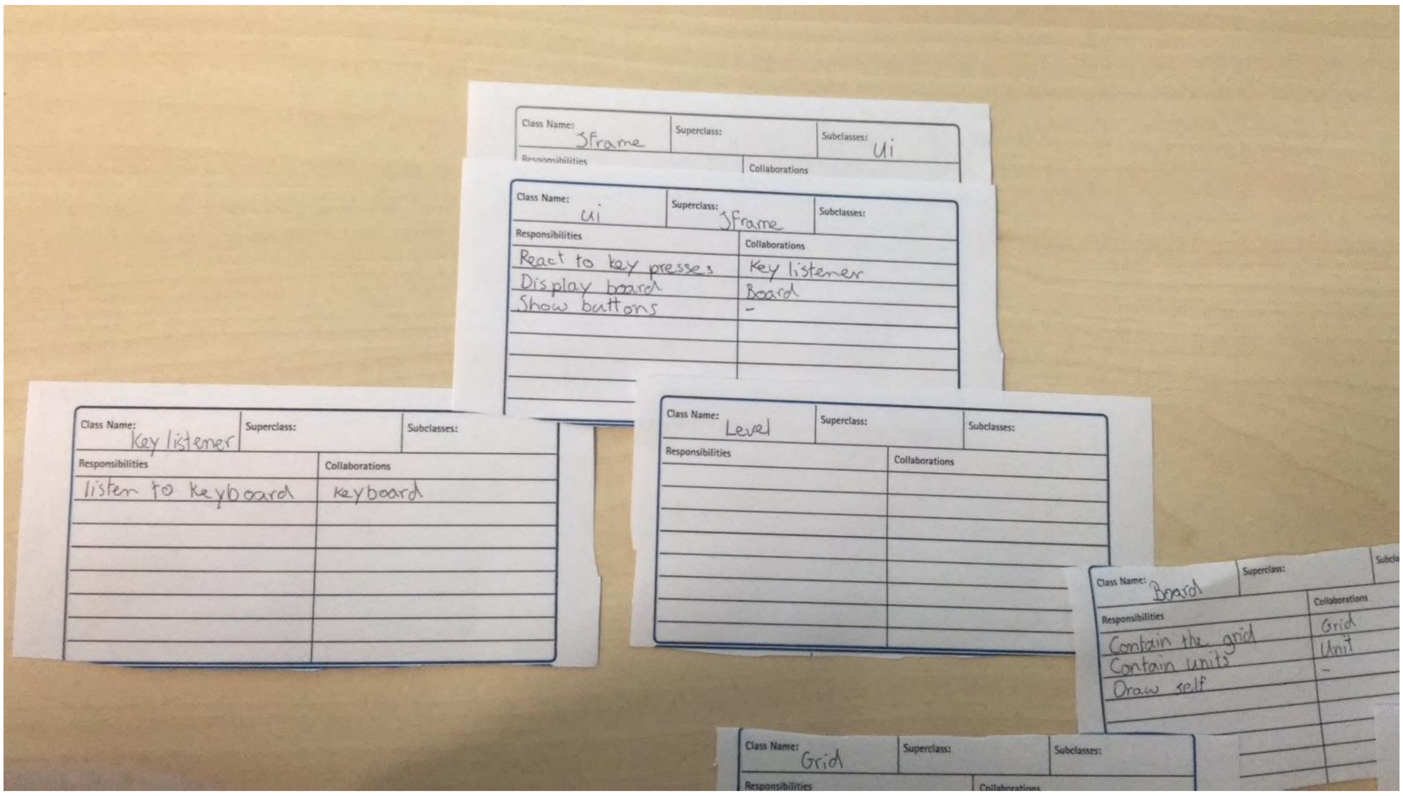
Class Name:	Superclass:	Subclasses:
Point		

Class Name:	Superclass:	Subclasses:
Enemy	Unit	

Class Name:	Superclass:	Subclasses:
Bubble	Unit	

Class Name:	Superclass:	Subclasses:
Player	Unit	

Responsibilities	Collaborations
React to key presses	Key listener



Class Name:	Superclass:	Subclasses:
JFrame		Ui

Class Name:	Superclass:	Subclasses:
Ui	JFrame	

Responsibilities	Collaborations
React to key presses	Key listener
Display board	Board
Show buttons	-

Class Name:	Superclass:	Subclasses:
Key listener		

Responsibilities	Collaborations
listen to keyboard	keyboard

Class Name:	Superclass:	Subclasses:
Level		

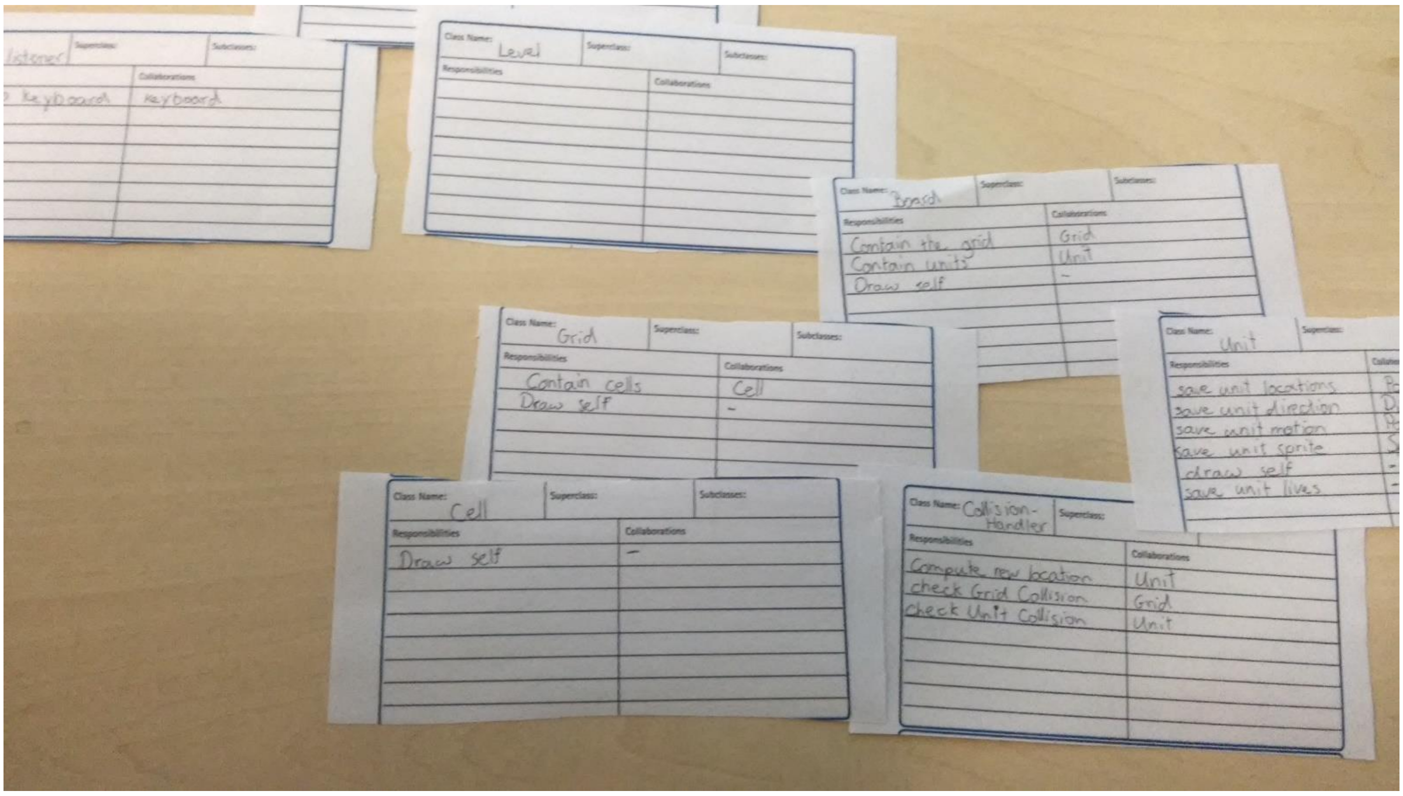
Responsibilities	Collaborations

Class Name:	Superclass:	Subcla
Board		

Responsibilities	Collaborations
Contain the grid	Grid
Contain units	Unit
Draw self	-

Class Name:	Superclass:	Subclasses:
Grid		

Responsibilities	Collaborations



Class Name: listener	Superclass:	Subclasses:
	Collaborations	
keyboard	keyboard	

Class Name: Level	Superclass:	Subclasses:
Responsibilities	Collaborations	

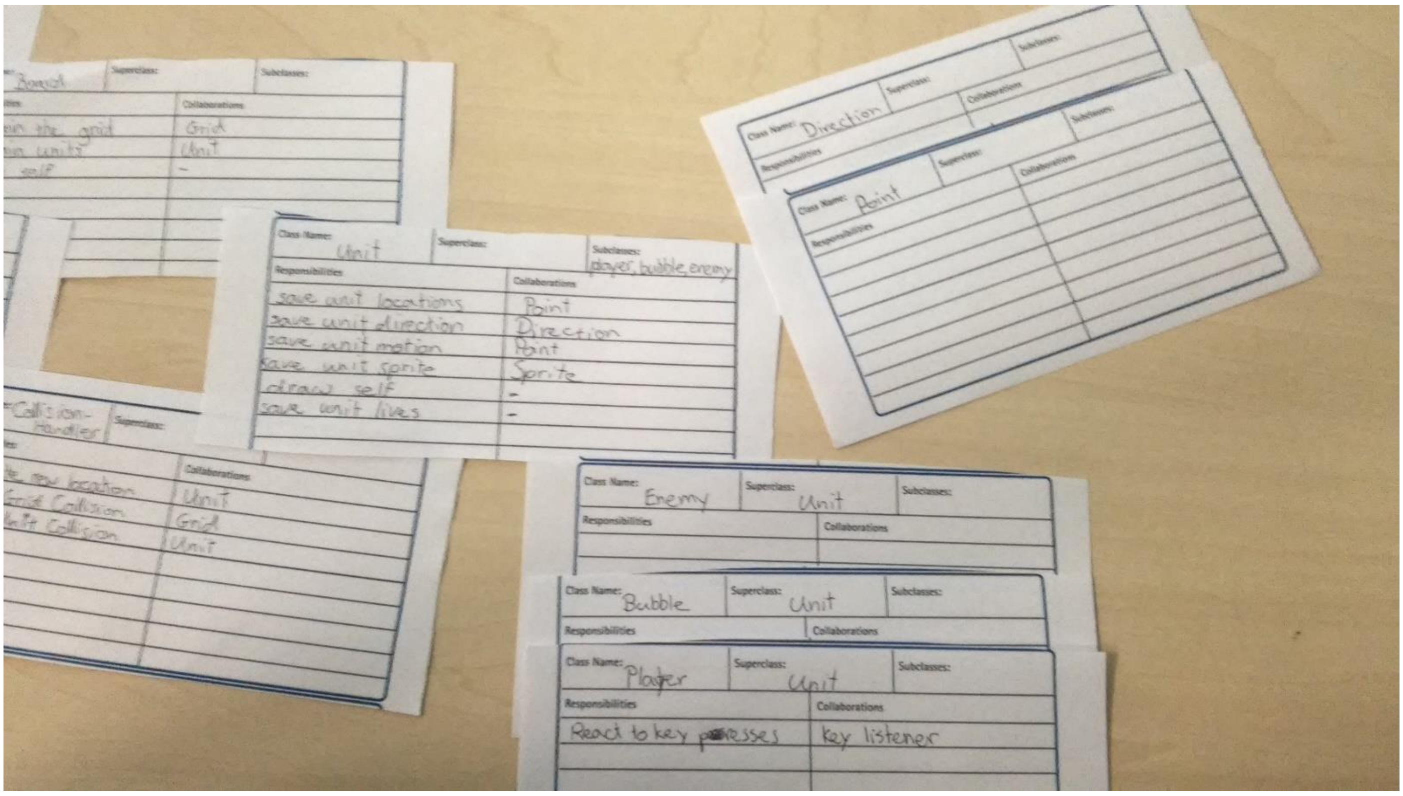
Class Name: Board	Superclass:	Subclasses:
Responsibilities	Collaborations	
Contain the grid	Grid	
Contain units	Unit	
Draw self	-	

Class Name: Grid	Superclass:	Subclasses:
Responsibilities	Collaborations	
Contain cells	Cell	
Draw self	-	

Class Name: Unit	Superclass:
Responsibilities	
save unit locations	
save unit direction	
save unit motion	
save unit sprite	
draw self	
save unit lives	

Class Name: Cell	Superclass:	Subclasses:
Responsibilities	Collaborations	
Draw self	-	

Class Name: Collision-Handler	Superclass:	
Responsibilities	Collaborations	
Compute new location	Unit	
check Grid Collision	Grid	
check Unit Collision	Unit	



Class Name: Unit	Superclass:	Subclasses: player, bubble, enemy
Responsibilities	Collaborations	
save unit locations	Point	
save unit direction	Direction	
save unit motion	Point	
save unit sprite	Sprite	
draw self	-	
save unit lives	-	

Class Name: Direction	Superclass:	Subclasses:
Responsibilities	Collaborations	

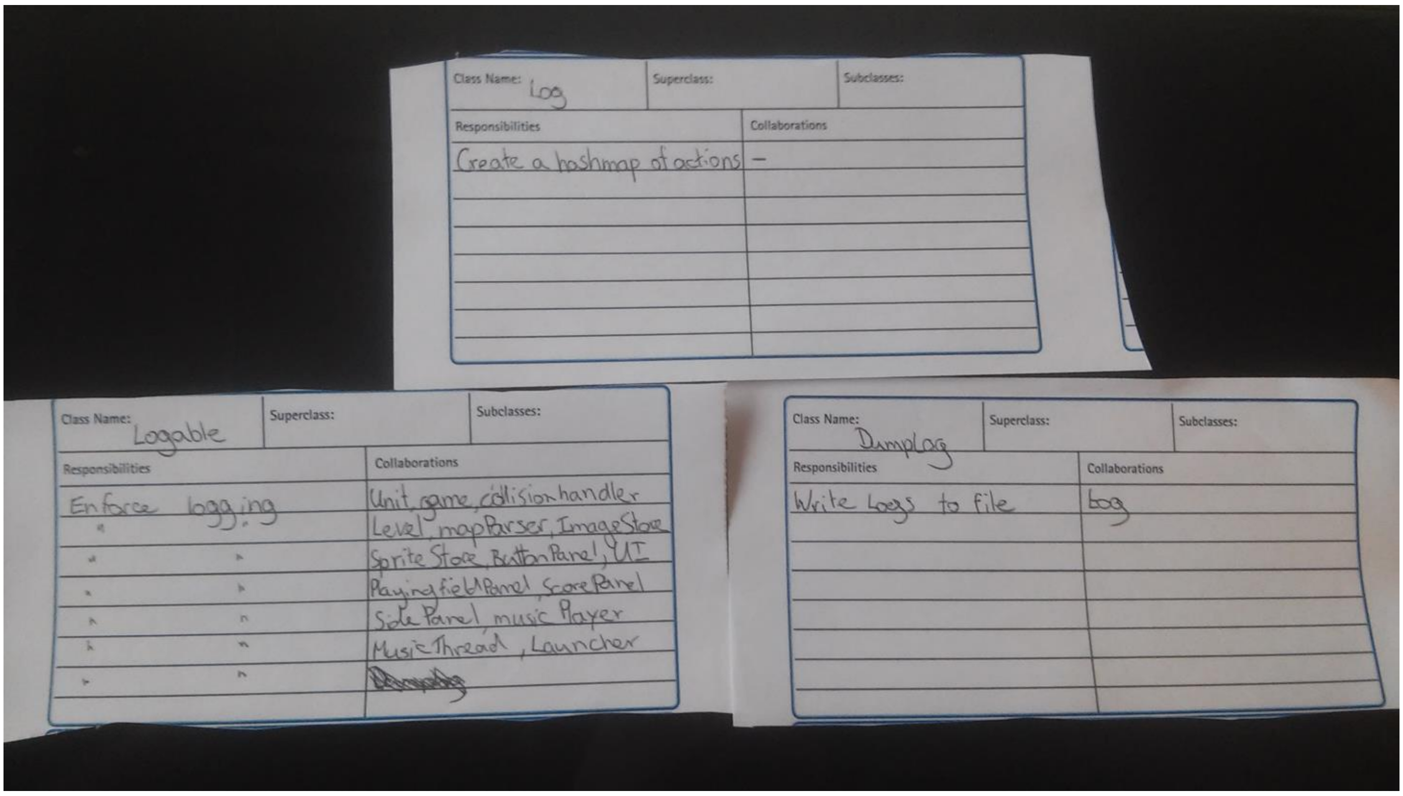
Class Name: Point	Superclass:	Subclasses:
Responsibilities	Collaborations	

Class Name: Enemy	Superclass: Unit	Subclasses:
Responsibilities	Collaborations	

Class Name: Bubble	Superclass: Unit	Subclasses:
Responsibilities	Collaborations	

Class Name: Player	Superclass: Unit	Subclasses:
Responsibilities	Collaborations	
React to key presses	key listener	

Appendix B: Responsibility Driven Design: CRC (Logger)



Class Name: Log	Superclass:	Subclasses:
Responsibilities	**Collaborations**	
Create a hashmap of actions	–	

Class Name: Logable	Superclass:	Subclasses:
Responsibilities	**Collaborations**	
Enforce logging	Unit, game, collision handler	
"	Level, mapParser, ImageStore	
" "	Sprite Store, ButtonPanel, UI	
" "	PlayingfieldPanel, ScorePanel	
" "	Side Panel, music Player	
" "	MusicThread, Launcher	
" "	[illegible]	

Class Name: Dumplog	Superclass:	Subclasses:
Responsibilities	**Collaborations**	
Write Logs to file	Log	