Exercise 1 - Your wish is my command, Reloaded

Explanation of what we implemented:

Our task was to refactor all classes in the Screen package. This package contains all the screens of our game.
The longest and most messy class was the GameScreen class. We divided the GameScreen class into smaller methods and even made a new Painter and DrawPlayer class to divide it to smaller methods. This design is more responsibility driven because each class now have only 1 repsonsibility.
GameScreen uses Painter to paint all the elements on the screen. The Painter class paints all moving elements on the screen. Because of the more complex movement of the player we created an extra class named DrawPlayer. DrawPlayer only focusses on drawing the player in every state. DrawPlayer is divided into methods, such as playerFlyingLeft or playerFlyingRight etc for better readability and maintainability. We also made use of a ScreenBuilder class. Most screens had the same variables that had to be set on making the screen. They have the same size, are not resizable and had to be set to visible. So we made a class apart to control that for almost every screen.
For the WinningScreen and LosingScreen we added some hierarchy by making those screens childs of an EndGameScreen. Because Losing and WinningScreen share the same underlying structure. Both screens must have the same button (return to main menu). And both need to tell the player what happend, so whether he/she won or lost the game. Again did we do this to improve the code quality.
And we also divided all long methods (30 lines or more) into smaller methods (30 lines or less). So all classes are much more clear now and better maintainable.

Below you can find the CRC Cards and the UML Diagrams of the new structure.

ScreenBuilder	
Superclasses:	
Subclasses:	
Will setup the basics of every screen.	- GameScreen - EndScreen - LeaderBoardScreen - LevelOverviewScreen - LogScreen - PauseScreen - StartScreen

GameScreen	
Superclasses: JPanel	
Subclasses:	
Will display everything in a game on screen, using the painter.	- SurvivalDriver - NormalDriver - Painter - ScreenBuilder

Painter	
Superclasses:	
Subclasses:	
Will display everything in a game on screen. Player, bubbles, walls, rope, powerups etc.	- SurvivalDriver - NormalDriver - DrawPlayer

DrawPlayer	
Superclasses:	
Subclasses:	
Will display the player in a game.	- Painter

EndScreen	
Superclasses: JFrame	
Subclasses: WinningScreen, LosingScreen	
Defines what a screen at the end of a game must have.	- WinningScreen - LosingScreen - ScreenBuilder

WinningScreen	
Superclasses: EndScreen	
Subclasses:	
Creates a screen telling the player that he won the game.	- EndScreen - NormalDriver

LosingScreen	
Superclasses: EndScreen	
Subclasses:	
Creates a screen showing the player his score and telling that he lost the game.	- EndScreen - NormalDriver - SurvivalDriver

LeaderBoardScreen	
Superclasses: JFrame	
Subclasses:	
Creates a screen where all previous scores are shown.	- SurvivalDriver - Leaderboard - ScreenBuilder

LevelOverviewScreen	
Superclasses: JFrame	
Subclasses:	
Shows a grid with all levels that the player already finished so he can start from there.	- StartScreen - NormalDriver - ScreenBuilder

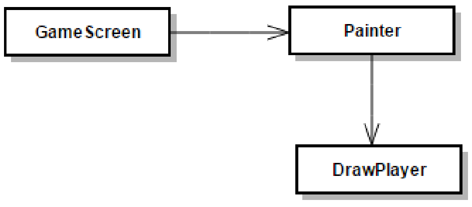
LogScreen	
Superclasses: JFrame	
Subclasses:	
Shows the logger with everything that is being logged.	- Logger - LogObject - LogFilters

	- ScreenBuilder

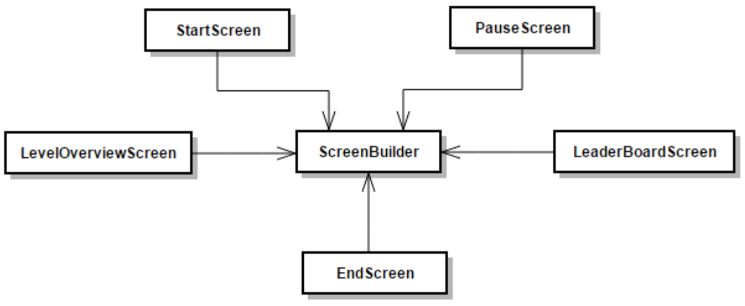
PauseScreen	
Superclasses: JFrame	
Subclasses:	
Pauses the game.	- Game - ScreenBuilder

StartScreen	
Superclasses: JFrame	
Subclasses:	
Screen that is first called. You can select a game type from here.	- NormalDriver - SurvivalDriver - LevelOverviewScreen - ScreenBuilder

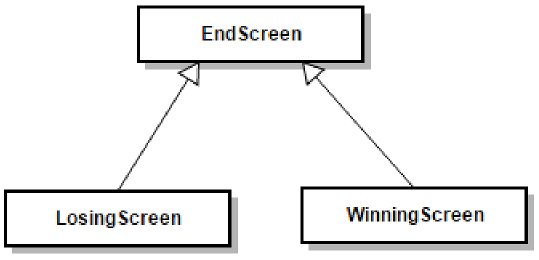
GameScreen UML Diagram



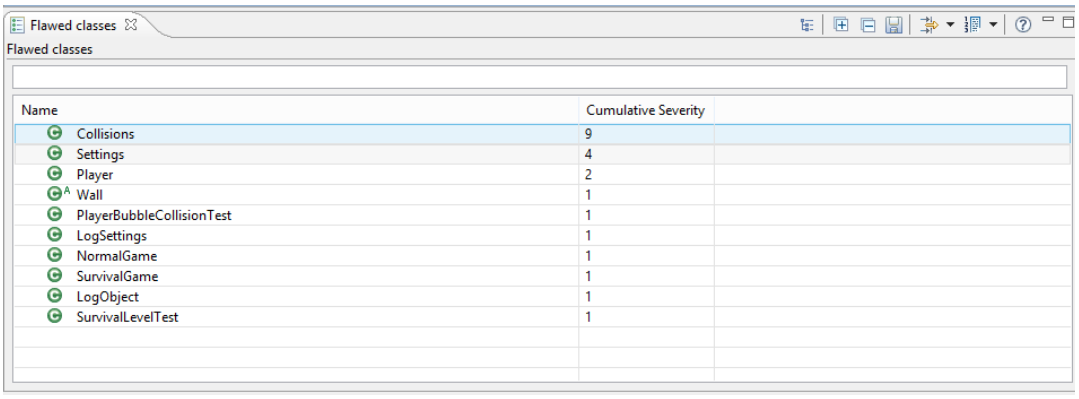
ScreenBuilder UML Diagram



EndScreen with inheritance UML Diagram



Exercise 2 - Software Metrics

1) The analysis file is located in the document folder: SEMgroup32/Documents/InFusion output.mse
2) 

Flawed classes

Name	Cumulative Severity
Collisions	9
Settings	4
Player	2
Wall	1
PlayerBubbleCollisionTest	1
LogSettings	1
NormalGame	1
SurvivalGame	1
LogObject	1
SurvivalLevelTest	1

Collisions

Cumulative Severity: 9
Problem
The Collisions class having two flaws. Those were Schizophrenic class and Feature envy. The feature envy had to do with specific methods within the Collisions class. The public interface of this class is large and is used in a non-cohesively way.

Design choices leading to this flaw
When we build this class we wanted it to do all the checks on collisions that might happen during runtime of the game. When doing that in a specific class it will cause many outgoing method calls and not many method calls to itself.

How to fix it
We split up the Collisions class into 5 different classes that each handle their own type of collision. The Collisions class got renamed to Collision and became abstract. This Collision class is now a blueprint for the 5 new implemented classes. Now the repsonsibilities are spread over the subclasses and the outgoing method calls and non-cohesiveness are toned down drastically. This also fixed the temporary attributes, just bringing that down to have game as a parameter. From that game all the things that are needed are retrieved. Also the notification of observers is done in a specific method within the subclasses. These changes reduced the amount of different methods drastically. And therefore fixed the design flaws of Collisions.

Controllers

Cumulative Severity: 5-7
Problem
The controllers are Tradition Breakers. Each of the classes implement too many methods that do nothing. Since their parent has a couple of abstract methods all of which are needed for some of the subclasses all of the subclasses had to implement those.

Design choices leading to this flaw
The implementation of the observer pattern on the collisions class. Since we have many different kinds of collisions between different objects we need many different update functions for the observers. This caused the amount of methods that we had. Many controllers of certain classes do nothing when certain things collide so their update functions stay empty.

How to fix it
Remove the ‘abstract’ from the methods declared in the abstract class observer. Then give all those methods an empty body. Next is removing all the redundant methods from the subclasses and only keeping the methods that are actually used. Make sure the actually used methods are still overriding the methods declared in their parents. This is still according to the observer pattern, since every observer and the actual observer class itself has all those update functions. Now that we have the empty methods removed the tradition breaker flaw is killed.

NormalGame and SurvivalGame

Cumulative Severity: 1
Problem
Both these game classes impelement empty methods from its parent (Game class). These methods case the tradition breaker flaw.

Design choises leading to this flaw
This flaw exists in our code since the beginning of the implementation of subclasses from Game. We did not think everything through when making abstract methods in the Game class. Not all of these abstract methodes are even used.

How to fix it
Remove the abstract method that is redundant in the Game class. For the other methods that are in the children but are empty use the same fix as Controllers. Just make the method with an empty implementation in the Game class then only override the method when you need it in the subclass. This fixed the tradition breaker flaw for those two classes too.

Settings

Cumulative Severity: 4
Problem
Settings is considered a Data class by inCode. This means that it is exposing a significant amount of data in its public interface. This is a problem because when the class becomes to big, it will contain a lot of unused data, making the class needlessly complex. Settings.java has many attributes, and all it’s methods are accessor methods.

Design choices leading to this flaw
The settings class was explicitly designed to be a data class. The whole purpose of the settings class is to make our life easier. It holds a lot of attributes that we change often for testing purposes, for example player movement speed and screenheight. Before the settings class existed these variables were scattered throughout all the code, and it became a pain to find these variables when you were trying to change them. With the settings class we would have all the variables in one place, with well defined names.

How to fix it
While we all agree we a settings class is very helpful, it turned out a bit to big. Currently the variables that are related are seperated by a whitespace. These related variables now have their own settings class.