Group 7

Assignment 4

Authors:
Emilie de Bree - 4247558
Toine Hartman - 4305655
Jeffrey Helgers - 4318749
Jim Hommes - 4306090
Joost Pluim - 4162269
Matthijs Verzijl - 4282604

Supervisor:
Alberto Bacchelli

Teaching Assistant:
Aaron Ang

Delft University of Technology

October 16, 2015

Contents

1. Your wish is my command, Reloaded

1.1 Question 1

This question included adding a feature of choice by our TA. We together determined that this week we would add a multiplayer mode to the game. In Question 2 we will explain the RDD we applied and the UML of the improvements made to the game.

1.2 Question 2

1.2.1 Introduction

When realising a second player in the game, we came across multiple problems in our structure. The main problem being that it was not a pure Observer Design Pattern. That's where a huge refactoring came in: all elements of Model do not extend SpriteBase anymore but contain one. Every instance has a timer of their own and does not require the LevelController to call their update function anymore. (Which was our main problem in our design pattern).

Luckily the Player class and its Input class were already set up very well, so that physically adding the second player was easily done.

1.2.2 Multi-Player: Responsibility Driven Design

We will shortly describe the important classes of the Multi-player feature and how we used Responsibility Driven Design in determining the responsibilities and collaborations of the classes which needed to be changed (or refactored).

Observer

Responsibility:
The Observer class observes classes which extend the Observable-class. It gets notified by an Observable-object if the object changes and therefor knows when to update in this case the LevelController and/or the ScreenController.

Collaborations:
Both LevelController and ScreenController extend the Observer class and therefore these classes will collaborate. Furthermore the Observer class will also collaborate with the Observable class to be able to get notified when updates are available.

Observable

Responsibility:
The Observable class forms the basis for all classes in the game which easily change state and therefore have to notify for example the ScreenController and the LevelController to determine if for example a level is over. A class which extends the Observable class thus is able to send an update to an Observer class once data in it's class is changed.

Collaborations:
The Observable class forms the basis for all objects in the game and therefore collaborates with Player, Monster, Powerup and Bubble, so that they can send updates. Furthermore an Observable Class can have multiple Observers which it can update.

Player

Responsibility:
With a multiplayer mode it is of course necessary that there can be multiple Players which can all act autonomous. Therefore each player must have it's own Inputs.

Collaborations:
Player collaborates with Observable to be able to send updates to Observer classes. It collaborates with SpriteBase in order to be able to have x,y coordinate and detect collisions.

Monster

Responsibility:
With a multiplayer mode a monster has to be able to kill both players.

Collaborations:
Monster collaborates with Observable to be able to send updates to Observer classes. It collaborates with SpriteBase in order to be able to have x,y coordinate and detect collisions.

Powerup

Responsibility:
Every Powerup should be able to be picked up by both players in multiplayer mode. After being picked up, it should als give the desired Powerup to only the player who picked it up.

Collaborations:
Powerup collaborates with Observable to be able to send updates to Observer classes. It collaborates with SpriteBase in order to be able to have x,y coordinate and detect collisions.

Bubble

Responsibility:
It must be clear for every bubble who shot the bubble to keep track of who shot a bubble. Points are only given to the player who pops the bubble.

Collaborations:
Bubble collaborates with Observable to be able to send updates to Observer classes. It collaborates with SpriteBase in order to be able to have x,y coordinate and detect collisions.

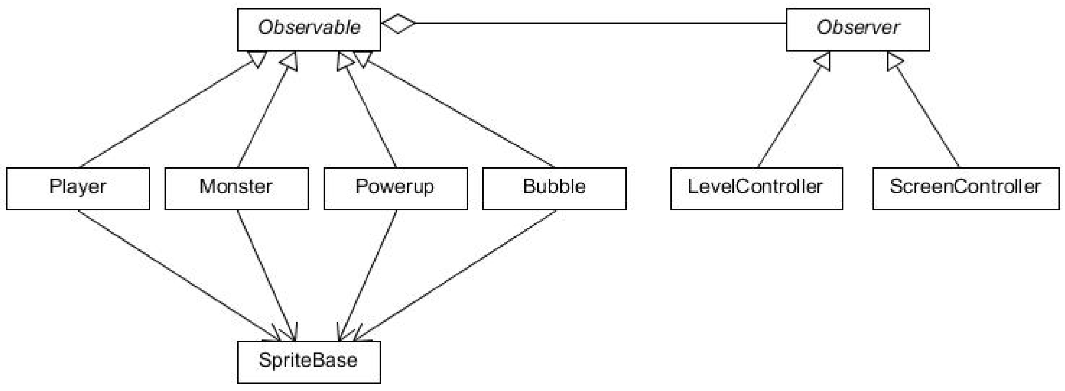
SpriteBase

Responsibility:
The Spritebase Class has to owned by all elements in the game after the refactoring. It will still contain all basic information for all elements in the game such as an x,y coordinate and an image.

Collaborations:
Since SpriteBase forms the basis for all elements in the game, it collaborates with Player, Monster, Powerup, Wall and Bubble.

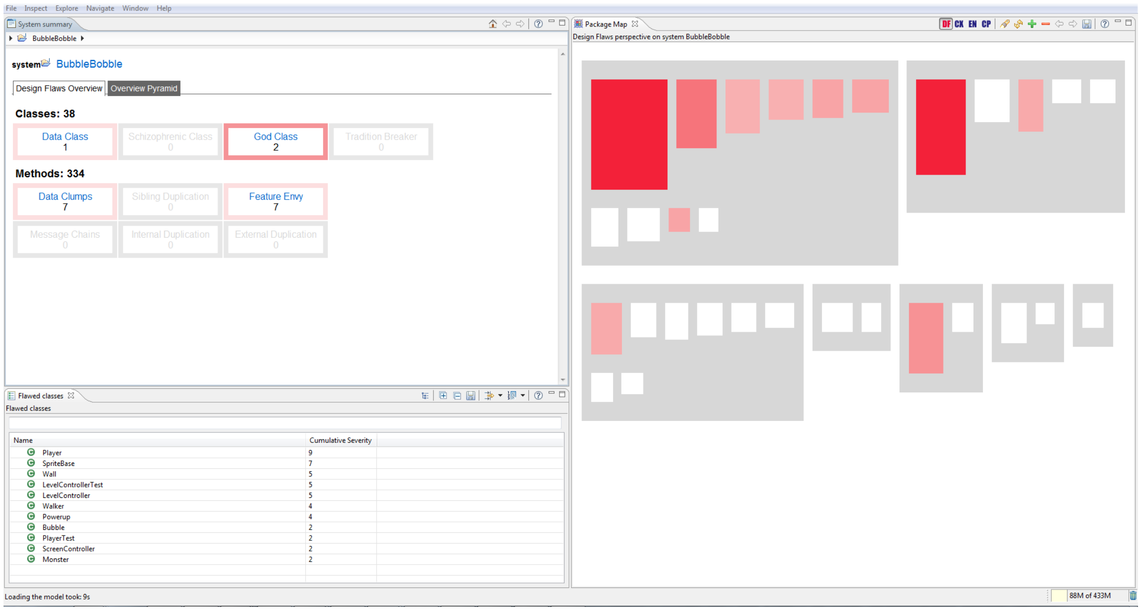
1.2.3 Multi-Player: UML

The UML of the important classes for the Multiplayer and therefore also the refactoring of the game, are displayed in the figure below.



2. Software Metrics

2.1 Question 1



File Inspect Explore Navigate Window Help
System summary
BubbleBobble
Package Map
Design Flaws perspective on system BubbleBobble

system BubbleBobble

Design Flaws Overview | Overview Pyramid

Classes: 38

Data Class 1	Schizophrenic Class 0	God Class 2	Tradition Breaker 0

Methods: 334

Data Clumps 7	Sibling Duplication 0	Feature Envy 7
Message Chains 0	Internal Duplication 0	External Duplication 0

Flawed classes

Name	Cumulative Severity
Player	9
SpriteBase	7
Wall	5
LevelControllerTest	5
LevelController	5
Walker	4
Powerup	4
Bubble	2
PlayerTest	2
ScreenController	2
Monster	2

Loading the model took: 9s

88M of 433M

This is a print screen of the inCode analysis. The actual inCode snapshot can be found in the HAND-IN folder.

2.2 Question 2

2.2.1 First Fatal Flaw: God Classes

Design choices

The first fatal flaw we decided to solve was the god class, in our code there were two god classes (LevelController and Player). A god class is a class that uses many attributes from other classes. The design choice that led to this flaw was different for both classes. In the LevelController class we handled everything that happened in our program. For the Player class this was different, this class had a lot of attributes from one specific class, SpriteBase. This happened because the SpriteBase held the location of the player, so every method that needed the location of the player, the SpriteBase was called to get the current location.

The fix or the explanation

The LevelController class was fixed by restructuring a few methods and replacing other methods to a new class. The other god class, Player, was solved with making to new methods, 'getLocation' and 'setLocation'. These two methods called to the SpriteBase for the location of the sprite, now everywhere in the Player class were the player needed its location, the SpriteBase class did not need to be called anymore.

2.2.2 Second Fatal Flaw: Data Clumps

Design choices

The second fatal flaw are Data Clumps. This means that there were too many large groups of parameters. SpriteBase, Wall, Walker, Player, Monster, Powerup and Bubble all had the x, y, r, dx, dy and dr coordinates in their constructors, which was unnecessary. The design choice that led to this was the idea that it was better for every class to have all the coordinate information, but taking a step back and looking at the project as a whole shows that this was not needed. A class that handled all these parameters in one go was a much better idea. Of course, it is never a good idea to have too many parameters, so that was an error that could have been avoided earlier on.

The fix

This was fixed by adding a Coordinates class which handles all the coordinates for each of the other model classes. This in turn means that each model class only needs to call up the Coordinates class, and then using getters and setters can change the coordinates that need to be changed.

2.2.3 Third Fatal Flaw: Feature Envies

Design choices

The third fatal flaw are Feature Envies. Feature Envy means that a method uses (almost) only data from another class. This does probably mean that the method should be moved to the other class.
This was seen in the mehtod `SpriteBase.move`, which only used data from the `Coordinates` class. This was a result of a refactoring process, in which the coördinates of the `SpriteBase` were replaced by a dedicated class, but the method was not looked at.

The fix

This was fixed by adding an `apply()` method to `Coordinates`, which did exactly the same as the `move()` method in `SpriteBase`: added dx to x, dy to y and dr to r. `SpriteBase.move` does only call `Coordinates.apply`.