Software Engineering Methods TI2206

Assignment 5

Group 47

Course Manager: Alberto Bacchelli
TA: Aaron Ang

Lillian Lehmann 4009037
Karin van Garderen 4144384
Christian Maulany 4097238
Rogier Vrooman 4001257
Fieke Miedema
4141369
Bas Böck 4366174

Exercise 1 – 20-Time, Revolutions (40 pts)

Requirements

Must have:

- The waterfall shall appear as a floating bubble filled with water every 20 seconds starting from the top of the level.
- The waterfall shall begin streaming when the Player pops the bubble.
- The waterfall shall appear only in levels with a gap in the top and in the bottom of the level.
- The waterfall shall <move> from the top to the bottom of the level.
  - <move> when colliding with a wall it goes the other way until it finds a gap to move downwards.
- The waterfall shall <take> the Player with it upon collision with the Player.
  - <take> Transports it through the gap on the bottom, waterfall will disappear while Player will reappear through gap on top of level.
- The enemies in the game shall have a animated movement
- A score shall be visible when the player receives points.
- The shown score shall be removed from the screen after 2 seconds.
- The shown score shall be located just above the location where the player received points.

Should have:

-

Could have:

-

Won't have:

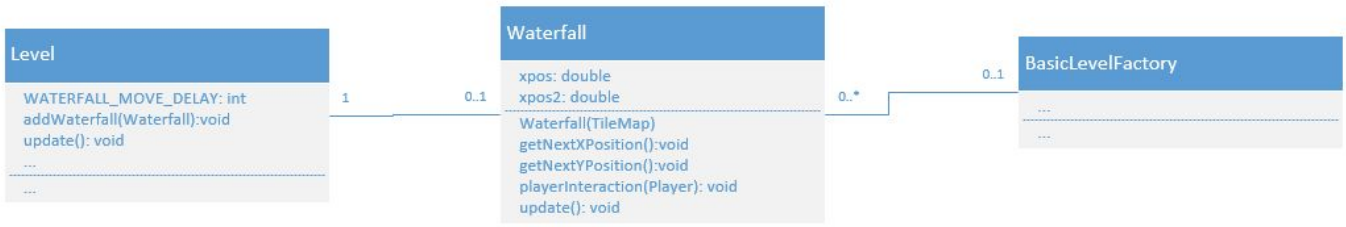
-

2. Responsibility Driven Design

Waterfall

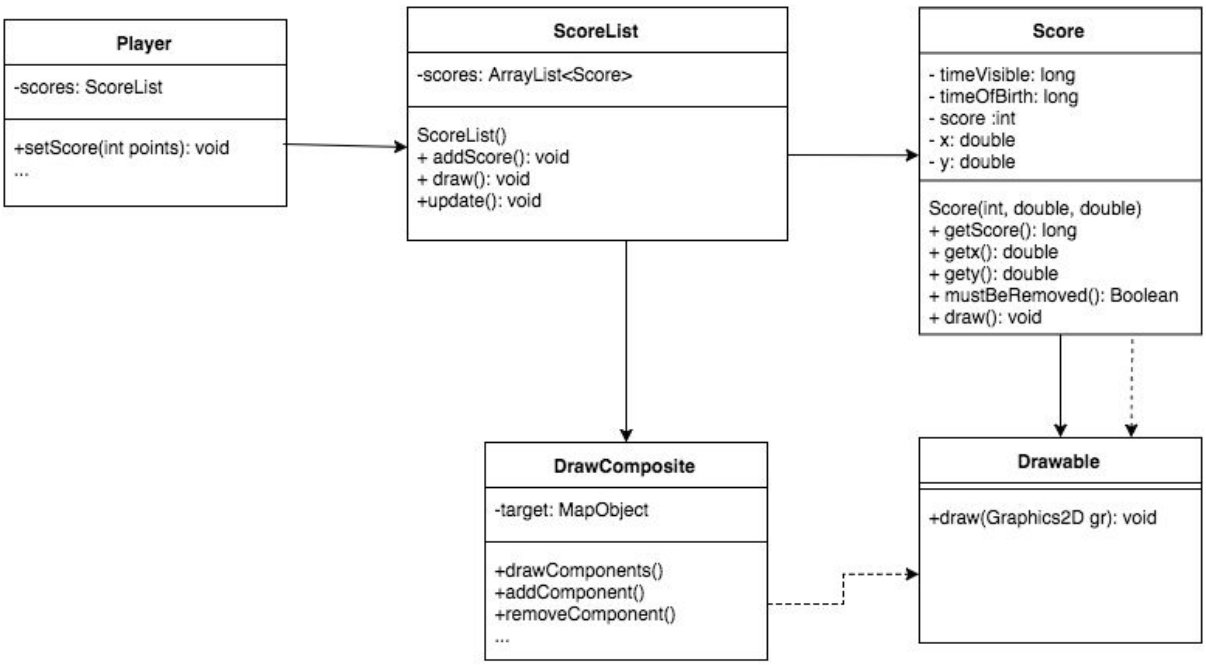
The waterfall is a MapObject that depicts a waterfall unit. It spawns after 5 seconds on top of the level and moves downwards from left to right, at the collision with a wall. When it collides with a Player, it takes the Player with it disabling the Player from any possible movement. The waterfall disappears when it drops through the floor of the level.

Waterfall	
Superclasses: none	
Subclasses: none	
Move through level	Player
Take Player with it on collision	



Score

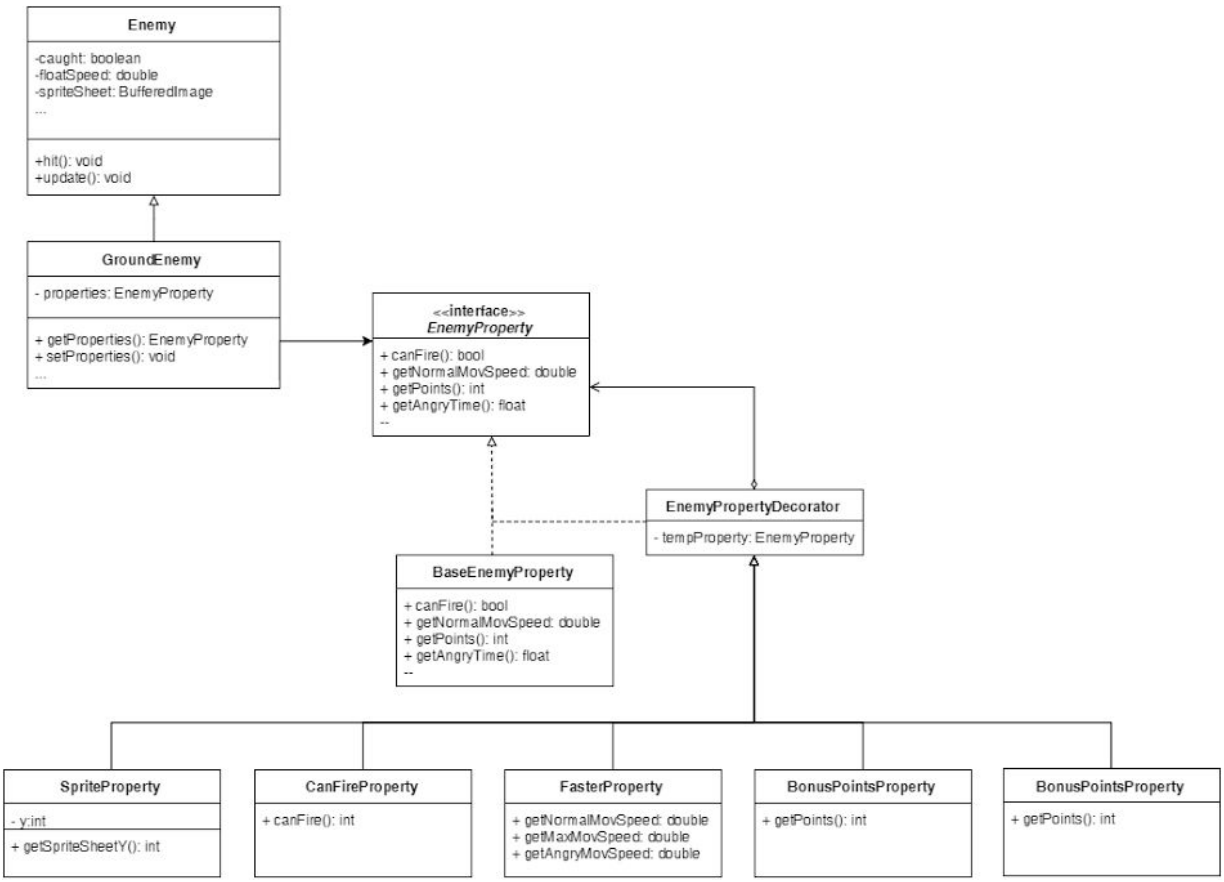
The drawable Score is now visible when a player receives points. For a moment of 2 seconds, the received points are shown above the player's location where the points are received. Like the projectiles, the player keeps a list of scores: the ScoreList, which interact with Drawable and DrawComposite. The scores are shown, until the function mustBeRemoved() returns a true value, which is checked for each Score in ScoreList.update().



Exercise 2 – Design patterns (30 pts)

Decorator pattern for enemies

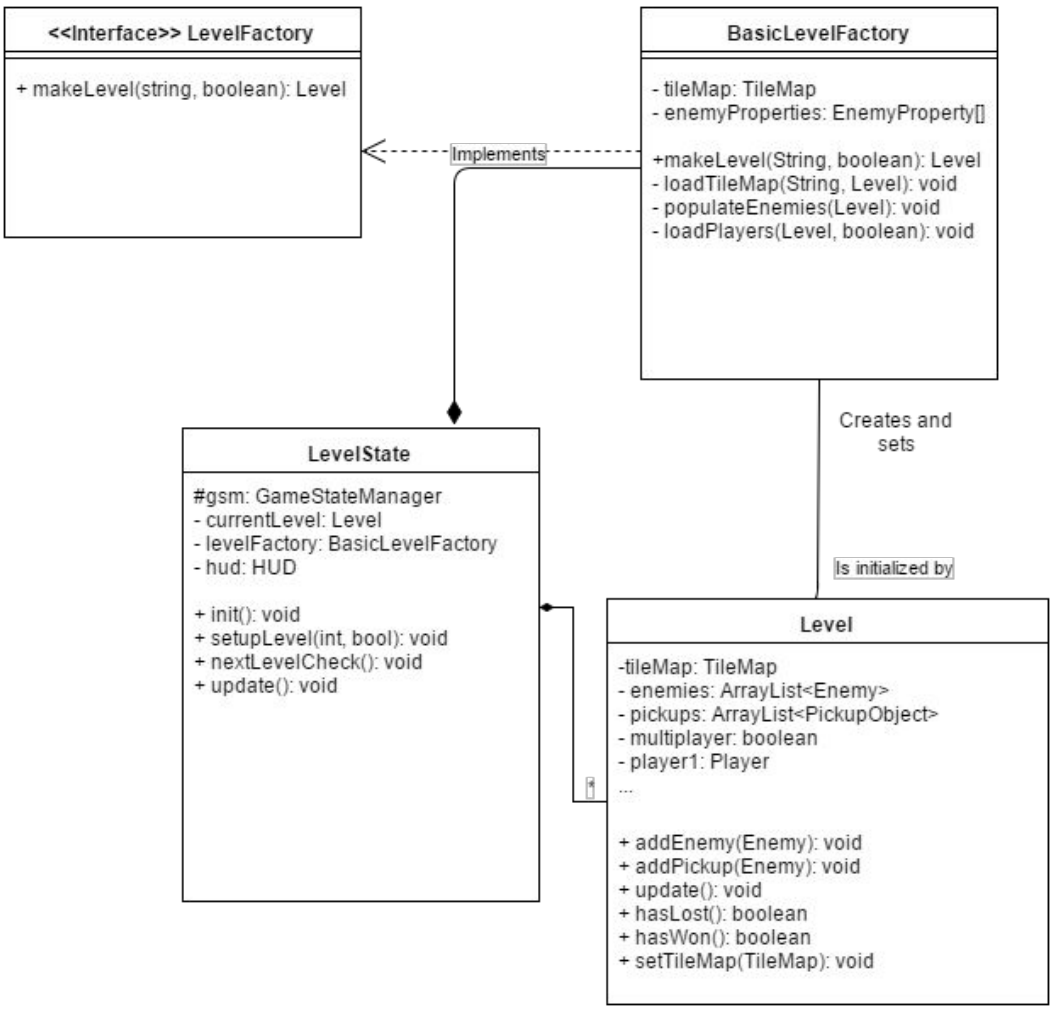
In the current build of the game there are two types of enemies(exclusing Magiron). One basic enemy which strolls around ands contains all the behavior for interacting with the enemy. The second type is similar, except that it has additional behavior for firing projectiles when the enemy is in front of them. In order to implement new types of enemies a new subclass has to be made and the new statistisch such as move speed and special behavior has to be hard coded. All this can be avoided using design patterns. This is where the decorator pattern comes in. The multiple enemies are now merged in a single class named GroundEnemy, and the properties for the enemy are added on top using decorators. For example you can take the basic enemy increase its speed using the FasterProperty, enable firing behavior using the CanFireProperty and even modify it's sprite using the SpriteProperty.



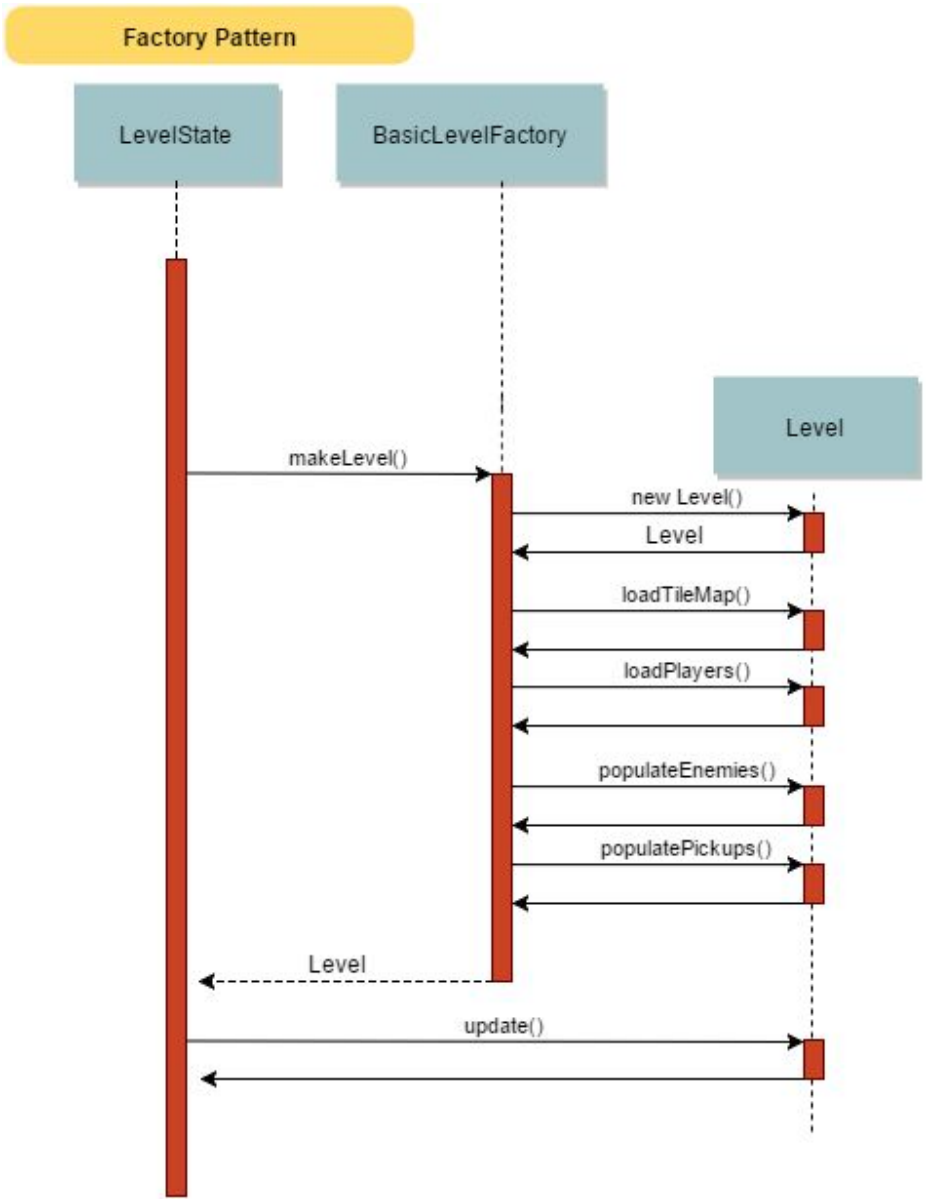
Factory Pattern for Level

The aim of the Factory pattern is to delegate creation of an object to a Factory class, instead of talking directly to the object constructor. This makes it possible to make diverse instantiations of the same (super)class. Different factory classes make different objects, but they all implement the same Factory interface, which simply demands that they are able to make this object with the same method call.

In our case, we chose to implement the Factory pattern for the creation of levels. Before implementing this, all Level functionality was embedded in the GameState LevelState and this had become quite a monster class. The first step was to create the class Level which has an instantation for each new level. Levels are now made through a LevelFactory, which in our case has only one implementation: BasicLevelFactory. For now, we have only one kind of level. However, we have been talking about the option to make bonuslevels, which would require making different kinds of levels.



The main result for now is that the level logic, which used to be in a single class, is divided evenly over three classes. LevelState still manages the score, HUD and input for the player. Level now contains all the logic for interaction and updating MapObjects in the game. BasicLevelFactory handles the creation of levels from files. This has improved the readability greatly. In the future, we might also want to use the advantage of the Factory pattern in creating various different instantiations of Level through different LevelFactory implementations.



Exercise 3 - Wrap up reflection (15 pts)

Looking back at the weeks of work that lay behind us, we think we can say that we are very content about the progression we have made with the game from the first version. There are a couple of things that we would like to explicitly reflect upon: us working as a group, the use of the Software Engineering Methods and finally of course a reflection on the game we have build.

Teamwork

Our team is, looking at all the other groups working, special in its composition. Six transition students: two from Industrial Design, two from Technology, Policy and Management, one from physics and one from the IT track on HBO-level. Our strong suits vary because of our different backgrounds, which made splitting tasks among team members easy. In our group there is a clear distinction visible between the people who are more comfortable with just writing the code, or people who rather work on for example the documentation or with the plug-ins. There is however also a downside to this: We almost never went 'out of our comfort zone' and tried something that we weren't familiar with yet. Everyone worked on an isolated field. This led to the fact that no-one masters all the skills that were taught this course.

We never had problems with people not finishing their deadline without extensive consultation with the group. Sometimes something happens beyond control of the team member and within time communication, we were always able to fix the game before the deadline. The sprints were a useful tool in keeping everyone focused on their tasks and taking responsibility for its execution.

We did not use roles like problem owner and sprint master during our sprints. Since the size of this project is quite small, it wasn't a problem we faced, but it might be a good idea for next project to try this. For instance handing in our assignment on Friday was a bit of a struggle every week, since no-one knew exactly if all the tasks were done and who was responsible for handing everything in. A specific distribution of roles could help to solve this problem.

The main lesson we have learned with regard to teamwork is that you should not underestimate the need for constant communication, especially when four people are working on new features in different branches. By working in the same room at the same time (sometimes in pairs, sometimes with everyone), we were able to minimize the amount of merge conflicts or failing builds.

Software Engineering Methods

Since almost every sprint consisted of implementing new features, each week started with deriving valid requirements. It took us a couple of weeks to get it right, but in the end the MoSCoW model proved to be a useful tool in formulating

requirements. The requirements also formed the base to our sprint tasks every week.

Furthermore the UML helped us in understanding the game in several ways. It helps the developer to think about how he/she should adjust or add code to achieve their goal. But it is also nice for the rest of the team, because a class or sequence diagram is much easier to read than code! Therefore it gives a quick, but complete overview of the expanded game.

Then, 'Tools are for fools!', or so we thought. After having to repair more than 800 bugs in sprint 4, we learned the hard way that the Maven reports actually provide useful warnings. Even if you decide to don't change anything, it makes you explain your code and design choices more explicitly. From that moment on, we have used more software metrics to detect anomalies in the structure of the code.

The design patterns were also a very useful way to structure our code in a thoughtful manner. Some patterns were more effective than others, but they all improved our code quality significantly. The main problem with the design patterns was that they were often quite drastic. This led to a lot of merge conflicts and errors in other branches. Also it took the rest of the team some time to get used to the new game structure, but here the UML came in handy!

There were a few things that we didn't pay much attention to at the beginning of the project, but became gradually more important, such as the extensibility and maintainability of the code. Those topics didn't get the attention they deserve, mainly because our code review was below par. After three weeks of insufficient code review, we assigned people to specifically review the code of other team members. This ensured that everyone was responsible for the code review of one other team member and reserved time to check that code more thoroughly than others.

Finally our tests were pretty bad, which led to fiddling build failures and errors. Also bug finding is a lot easier when you have high quality test cases. Another consequence was that we had to restructure and write all the tests in the last two weeks. Early testing is something we definitely must do in our next projects!

The game features

Last but not least our game features! We started out good and were able to implement a lot of features in our first version of the game. The player could walk left and right, jump, fire projectiles, kill enemies, earn points and lose lives. Enemies walked around randomly and all three map-objects stayed within the borders of the game. Pretty nice!

This jump start enabled us to extend the game in all sorts of ways. We added things like animation of the map-objects and background, sound effects, three

new kind of enemies, local multiplayer possibilities, power-ups, et cetera et cetera. We think our game looks a lot like the real bubble bobble and we are very proud of the result so far!