Assignment 5
Temple Run

Group 5

TI2206 Software Engineering Methods

of the Computer Science curriculum

at the Delft University of Technology.

L. van Aanholt levi.vaan@gmail.com 4084101

M. Wolting matthijswolting@gmail.com 4356063

M.E. Kuipers marcelkuipers1996@gmail.com 4389042

N.E. Hullegien nilshullegien@gmail.com 4389069

H.M. Yeh H.M.Yeh@student.tudelft.nl 4386027

Exercise 1

Our requirements for this feature are:

- The obstacle we create must shoot a projectile.
- The projectile must travel towards the player at a speed so that the player is able to dodge it.
- If the player hits the projectile from the front, the player should die.
- If the player hits the obstacle that shoots the projectile from the front, the player should die.

To implement this feature, we created a class for the projectile, namely Bullet.java. This class creates the projectile that travels towards the player. This is done by using the LibGDX class Body, to which we applied a linear impluse (body.applyLinearImpulse()) with a negative speed. This way, the object moves indeed the direction we want it to move in.

Another class we created was the ShootingObstacle class. This class creates an object that moves with the game, so it looks like it is static. In the constructor of this class, we create a new Bullet object, which spawns at the same coordinates as the ShootingObstacle. After the game is started, the bullet starts moving away from this ShootingObstacle object, so that it looks like the object shot a bullet towards the player.

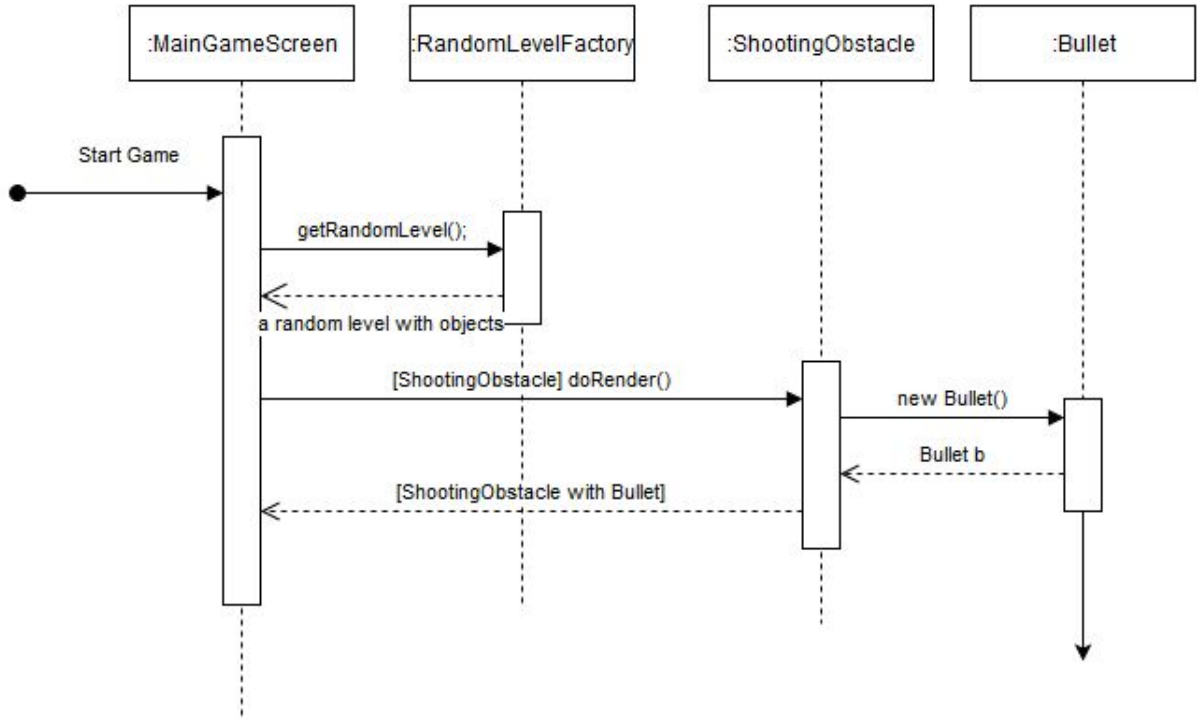
Next up, we created a BlockObstacle class. This class spawns a large cube in the game. It has the same functionality of the ShootingObstacle, the only difference is that it is larger.

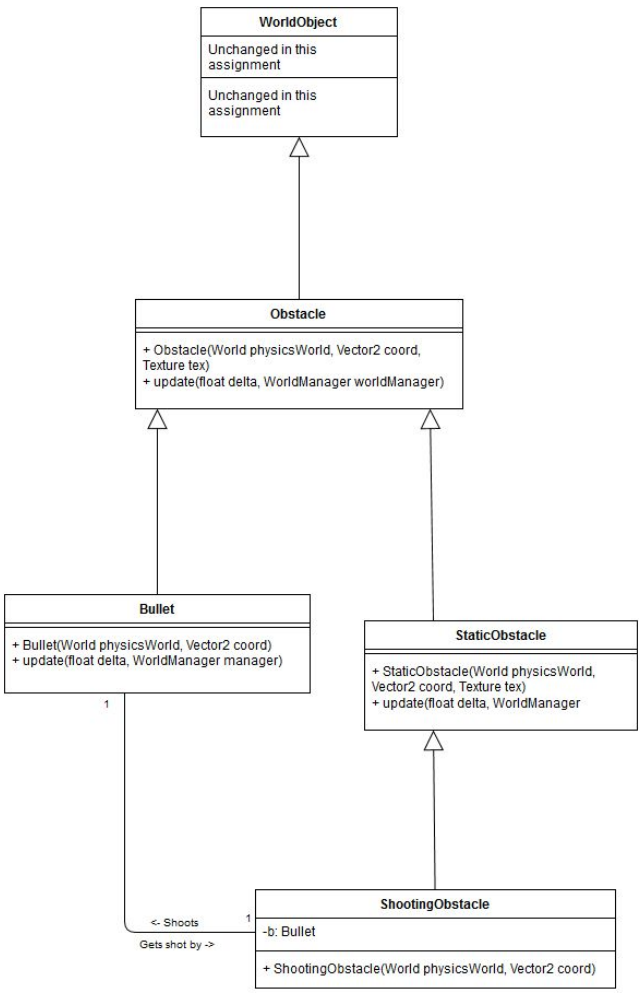
The BlockObstacle class is created from the earlier Obstacle.java class.

Due to this comparison, we found out that, instead of just keeping he original Obstacle.java class, we needed a class that could be a super class of Objects that do not move, like the BlockObstacle and the ShootingObstacle objects. Herefore, we created a class called StaticObstacle. This class is a subclass of Obstacle and is a super class of ShootingObstacle and BlockObstacle.

We decided not to create a super class for the Bullet class, because we this is the only moving obstacle in the game. If we ever implement more moving obstacles, we probably will create a super class for Bullet.

For now, the Bullet class is a sub class of the Obstacle class.

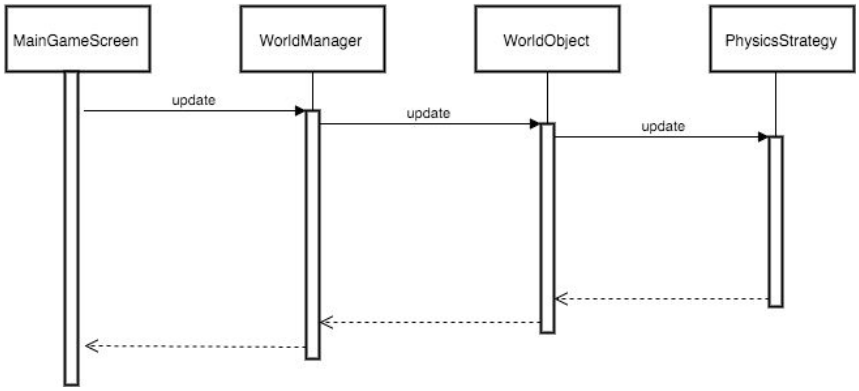


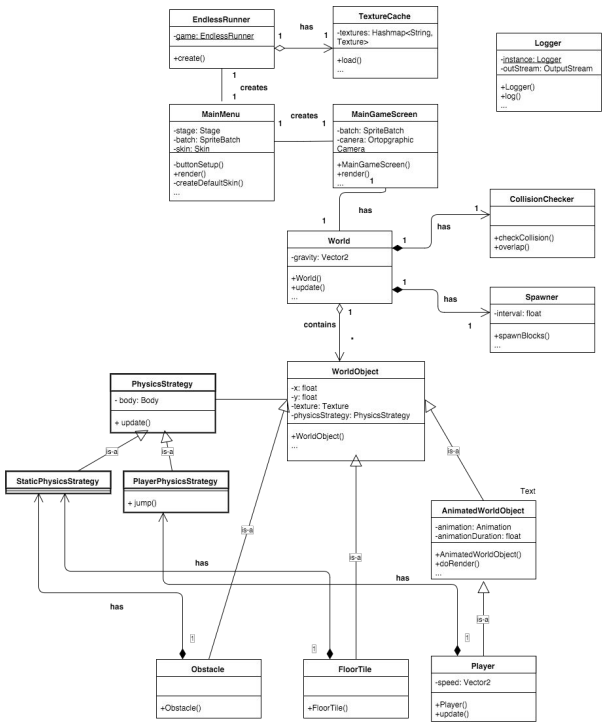


sequence and class uml Exercise 1.

Exercise 2

Pattern 1 - Strategy





We implemented the Strategy pattern in our game as a way to give different types objects separate physics behavior within the game. For example, in our game the floor the player walks on is a static body, and no updates are needed during the game. On the other hand, the player is a dynamic body, a constant lateral force is exerted on it, and it has to jump, tumble, etc. By giving every WorldObject a PhysicsStrategy, and using that in the object's update method, a uniform interface is available for updating the game's physics bodies. By varying the PhysicsStrategy implementations that are passed into the object, their physical behavior can be altered without modifying the structure of the WorldObjects.

Exercise 2.1

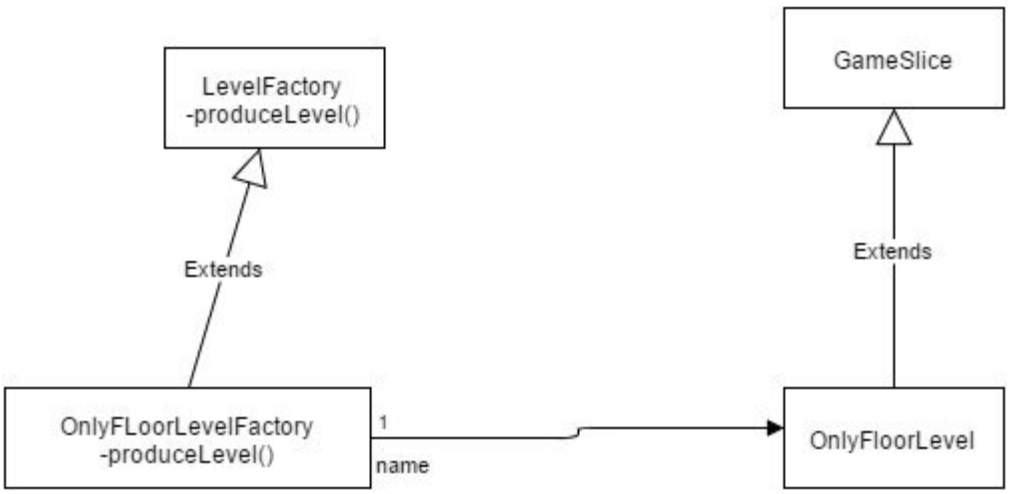
Factory method:

Our game stays in perpetual motion by constantly adding game piece (*GameSlice*) to a queue. The characteristics of these game pieces can be modified to contain different objects or have different parameters. Because we want these differences to be diverse we have chosen to declare them in different *Level* classes. Because different levels might not all have the same behaviour it is useful to use a factory method to produce a level.

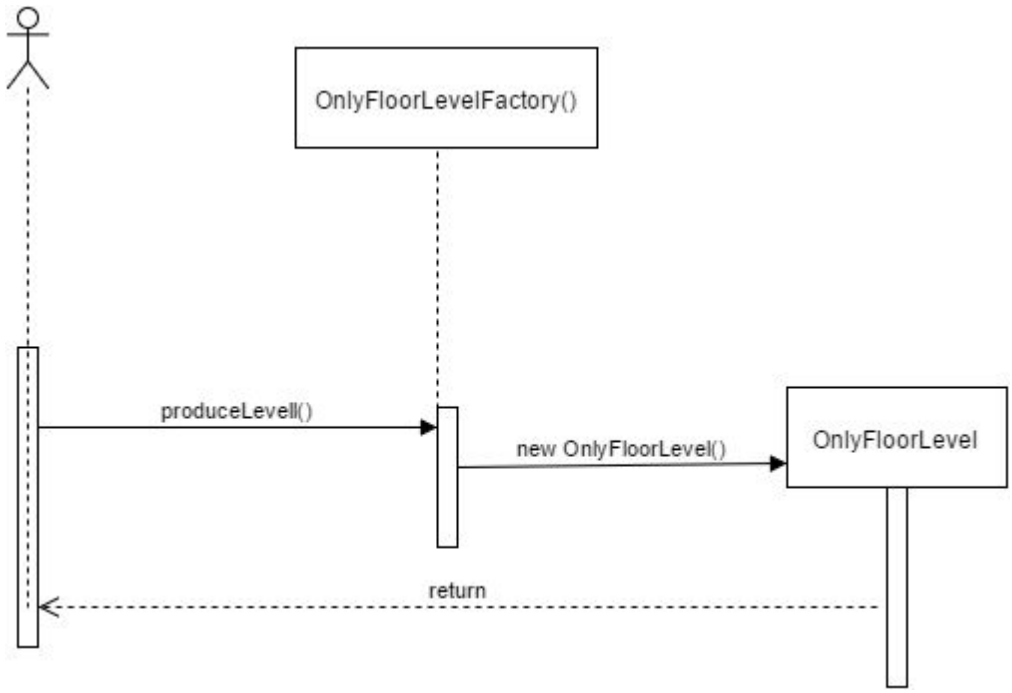
By declaring an abstract *LevelFactory* class we can allow multiple types of factories to be instanciated. By using concrete subclasses of this method we can created levels that can all be called with the same method, but might behave very differently.

Class diagram

..



Sequence diagram


OnlyFloorLevelFactory()
produceLevell()
new OnlyFloorLevel()
OnlyFloorLevel
return

Exercise 3: Reflection

This reflection will focus on the things we've learned in the SEM TU Delft course and how we will benefit from this knowledge in the future. Also, we'll talk about the software we've implemented for the course, and how we've implemented the things we have learned.

We reflect on the software development of the game we had to implement. When we look back at the repository from the initial version and the current repository we see a lot of differences. Most obviously, our project has grown. By now, we have implemented all Must Have and Should Have requirements that we defined at the beginning of the project. Also, in practice the game is more satisfying to play then at the start. The most important improvements have been made in the physics and the variety of levels. The game routine has improved and the gameplay feels more rewarding. Significant changes were also made to refactor the game, which really was necessary since the quick deadline of the first version forced us into some deliberate technical debt. The addition of design patterns has also made the code better organized definitely more modifiable.

While the current game itself is very simple, it is more important that we correctly utilized the material handed to us during the lectures and applied it to the project. In that regard, our main goal of gaining knowledge from the course has been achieved. Each group member contributed equally, tasks were distributed properly and punctual misdemeanour was punished justly. The fact that our group consists of people are who responsible and responsive to communication allowed us to work consistently and without any big problems. We would say the project was an overall success.

During the project there were multiple things we have learned. As a group, we have learned the importance of good planning, which was necessary due to the tight deadlines we have had to work with. Additionally, we have come to the realization that Scrum is the best method to reach these deadlines (even though the project still obligated us to follow it strictly). In that sense, it is arguable how useful Scrum has been.

We need to talk about Scrum. Learning Scrum for any one of us will surely be useful when we work in a company where Scrum is enforced. However, in terms of efficiency of the methodology we unanimously agree that Scrum is not the best way to go. We think that scrum elements like mandatory “scrum meetings” and the “scrum sprint run” are no substitutes for an adequate development team that enjoys working together. All the scrum elements might help inadequate teams with communication problems, but scrum also takes time, in which code could have been written. We agree more with the methods explained during the guest lecture of Erik Meijer, the hacker way. It came as a relief to us that we finally heard someone say something we were secretly, or not so secretly, all thinking .That we should just write code, and have code be the center of communication. Unfortunately we did not have that much of an opportunity to this insights in this project, as the lab work forced us to follow Scrum (quite strictly). Whether or not our knowledge of Scrum will come in useful will depend on the type of work we get into and on how the field of software engineering will develop over the following years. Either way, it is at least good that we have gotten both sides of the story, since now we can talk more informedly and nuancedly about software engineering methodologies.

On the subject of actually engineering the project, we have learned from the lab work. Using Responsibility-Driven Design we were able to oversee how different parts of our system were related to each other and how this could be improved. Most importantly, it made it very clear what function each part has. Design patterns were also useful since they provide clear and reusable solutions to problems you are guaranteed to run into when programming. Software metrics were helpful as well, because they actually point out parts of code that do work, but could have been designed in a better way. This has given us insights on how to solve or prevent these types of design flaws.

Also what we will likely be using the most in the future are design patterns. It is a good thing that these patterns were really focused on during the lectures. There is no question that these patterns will prevent us from getting stuck on trivial problems and will improve our code quality.

Another way our code quality will improve is by keeping software metrics in mind. Software metrics is about finding design flaws that come creeping into the code. This is code with a smell, without software metrics it is hard to put your finger on. Having

a way to define these flaws, which do not cause bugs or unexpected behaviour but worsen the code, is very valuable. By knowing about them our design will improve, increasing readability and making a system easier to manage and scale.

Something else that will likely improve our design is Responsibility-Driven Design. By defining for each class their collaborators and responsibilities, we can create a system in which each class communicates and interacts sensibly.

Putting our design on paper was also useful. UML provides us with a way to give a good overview of a system, while other types of diagrams, such as use case diagrams and sequence diagrams, allowed us define a specific functionality or purpose in a comprehensible manner.

All in all, we as a group are fairly satisfied with the things we have learned from the lectures and the lab work. Being provided with different approaches to problems, we have not only learned how all the techniques work, but we have also gotten insights into when (and when not) to apply them. These insights will probably eventually give us the knowledge that will most improve our capabilities as software engineers.