Space Invaders : Sprint 3 Assignment

Group 22
TI2206 Software Engineering Methods

Ege de Bruin
Bryan van Wijk
Dorian de Koning
Jochem Lugtenburg

October 9, 2015

Supervisor: Dr. A. Bacchelli
TA: Danny Plenge

Delft University of Technology
Faculty of EEMCS

1 Exercise 1 - 20-Time, Reloaded

The following new classes were created for the implementation of the new wave features. In the table the responsibilities and collaborations are presented for every class.

Class	Responsibility	Collaborates with	Super	Sub
AlienWaveFactory	Creates alienWave	alienWaveReader, alienWave		
AlienWaveReader	Creates alienWave patterns from file	AlienWavePattern		
AlienWavePattern	Remember pattern of alienWave			
AlienWave	Data about alienWave	Alien, AlienController		

1.1 Wave Functional Requirements

A list of functional requirements considered for the implementation of different waves using the MoSCoW method described in the previous section.

1.1.1 Must Haves

The Waves must meet the following requirements:

- A new wave shall have a different pattern of aliens.
- A wave pattern shall be stored in a file.
- Patterns shall be loaded upon starting the game.

1.1.2 Should Haves

The Waves should meet the following requirements:

- Aliens shall have different sizes.
- Aliens shall have different types.
- The game shall start with a standard pattern.
- The game shall load a random pattern upon loading a wave.
- Alien types shall have different colors.
- Alien types shall have different shooting speeds.

1.1.3 Could Haves

The Waves could meet the following requirements:

- Different alien types shall take more hits before they die.
- Rows of aliens shall move in independent directions of each other.

1.1.4 Would/Won't Haves

The Waves won't meet the following requirements:

- Aliens shall have different movement speeds.

1.2 Wave UML

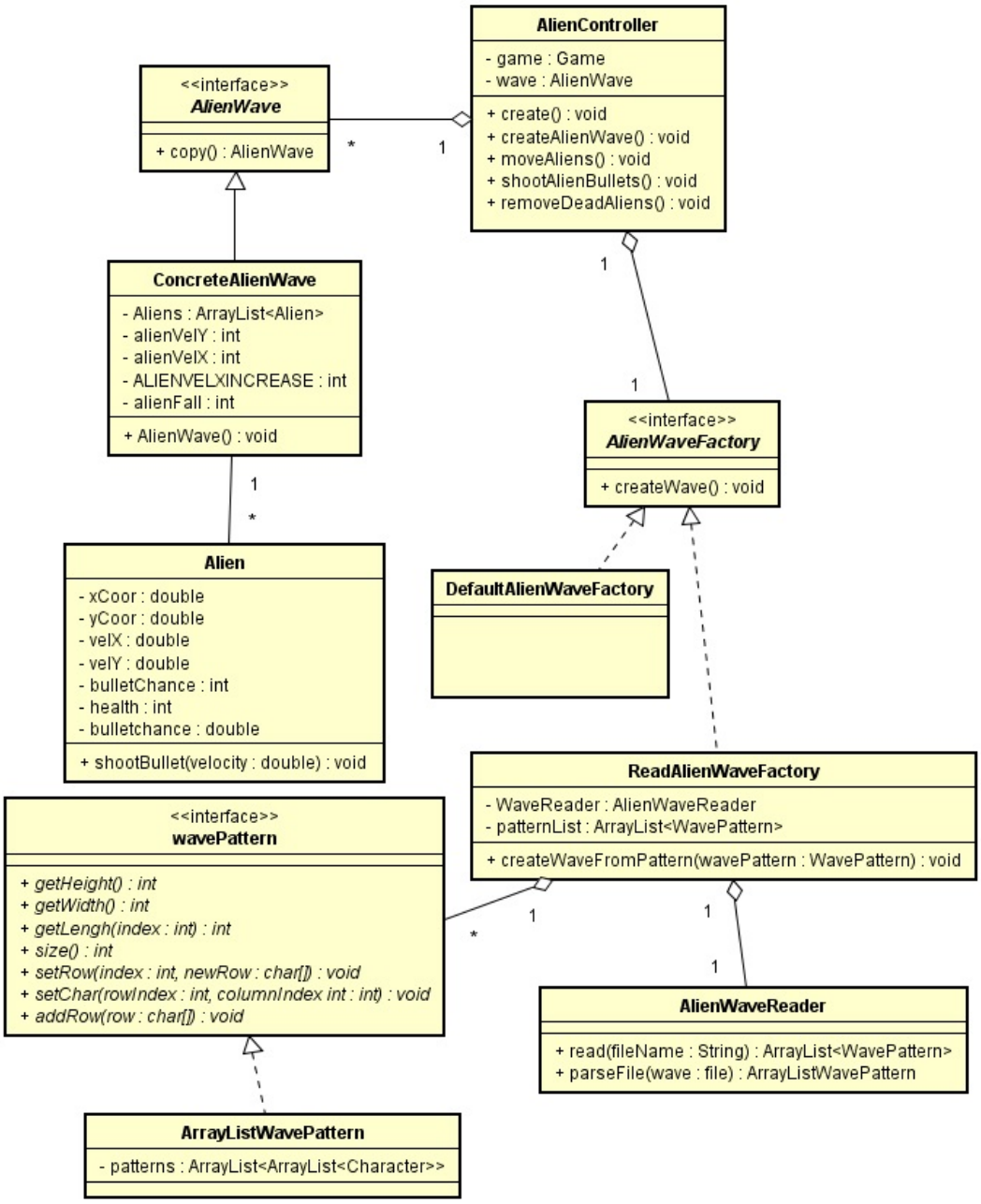

Figure 1: Wave UML Class Diagram

2 Exercise 2 - Design Patterns

2.1 Exercise 2.1 - Strategy Pattern

Before this sprint iteration, the project already contained a small implementation of the strategy pattern. The MovableUnit interface was already there to provide methods a Unit should implement if it can move on the screen.

Since not every Unit can move, it is more flexible to add an interface for this. The methods for moving are not needed in classes such as Barricade. Instead of adding these methods in the superclass, it is better to add them in an interface so only subclasses implementing the interface, implement the methods.

The same can be done for shooting, and activating a unit on the player (such as a powerup). The project was extended to include an interface for shooting, ShootingUnit and an interface for activating a unit on the player, ActivatableUnit.

2.2 Exercise 2.2 - Strategy Pattern

Figure 2 contains the UML Class diagram for the strategy pattern in Unit.

2.3 Exercise 2.3 - Strategy Pattern

Figure 3 contains the UML Sequence diagram for the strategy pattern in Unit. This diagram only contains the case concerning MovableUnit and ShootingUnit. It visualizes that all methods defined in the interfaces are implemented and used correctly.

2.4 Exercise 2.1 - Abstract Factory Pattern

A factory pattern can be useful for the creation of a PowerUpUnit. If a powerup is created in game, it does not need knowledge of the implementation. In the project, there are three 'flavors' of powerups, Speed, Life and Shoot. If a factory is used, the factory handles the creation of each powerup and the game can easily obtain it by calling a method on the corresponding factory.

In the project, an interface AbstractPowerupFactory was created. The interface defines a create method which should be implemented. Three factories: LifePowerupFactory, SpeedPowerupFactory and ShootPowerupFactory implement this interface and create the corresponding powerups. In the PowerUpController, these three factories are used for the creation of the required powerup, instead of calling the constructor.

2.5 Exercise 2.2 - Abstract Factory Pattern

Figure 4 contains the UML Class diagram for the abstract factory pattern concerning the powerups.

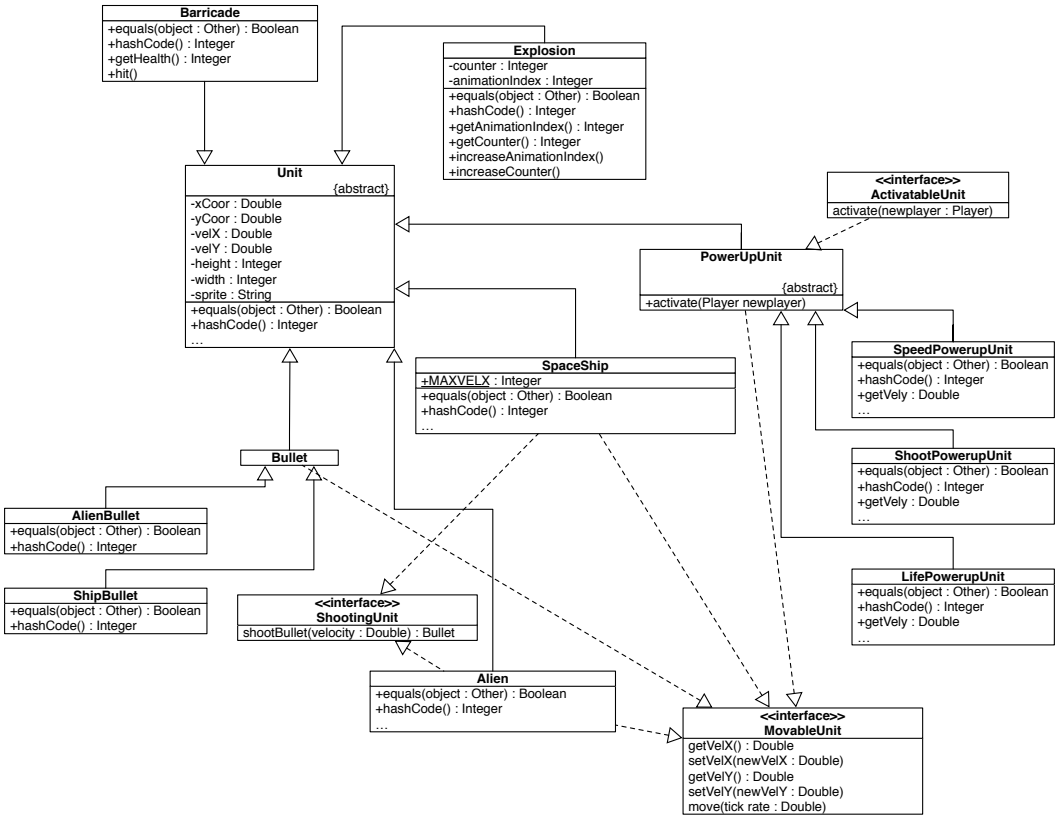

Figure 2: Strategy Pattern UML Class Diagram

2.6 Exercise 2.3 - Abstract Factory Pattern

Figure 5 contains the UML Sequence diagram for the abstract factory pattern concerning the powerups. It only contains the case of a Life powerup. Since the selection of these powerups is random, each time this sequence is visited, one of three powerup factories may appear.

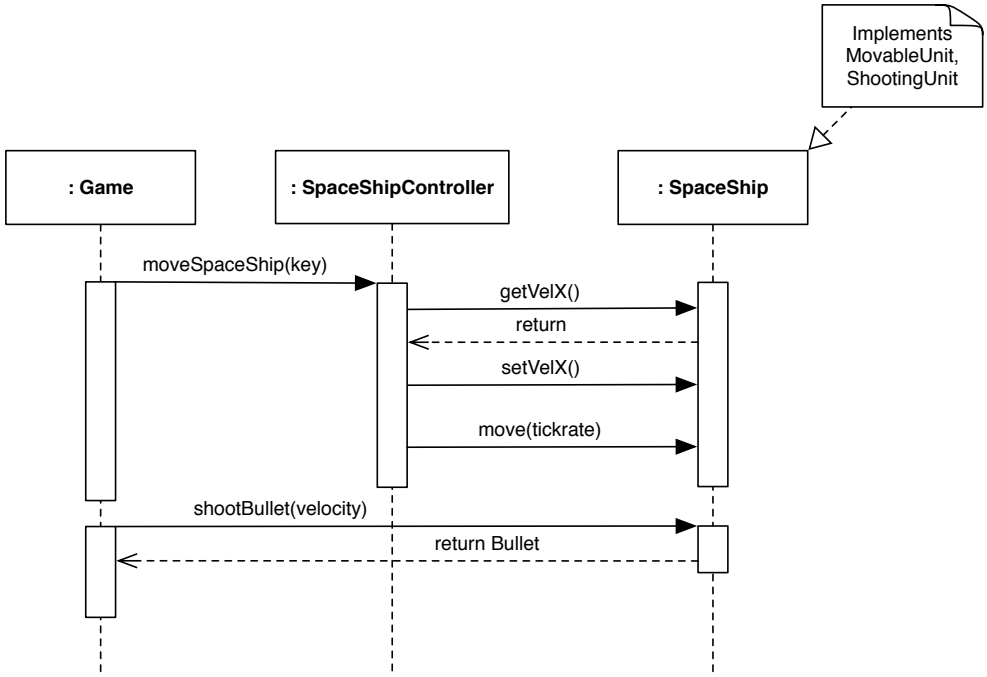

Figure 3: Strategy Pattern UML Sequence Diagram

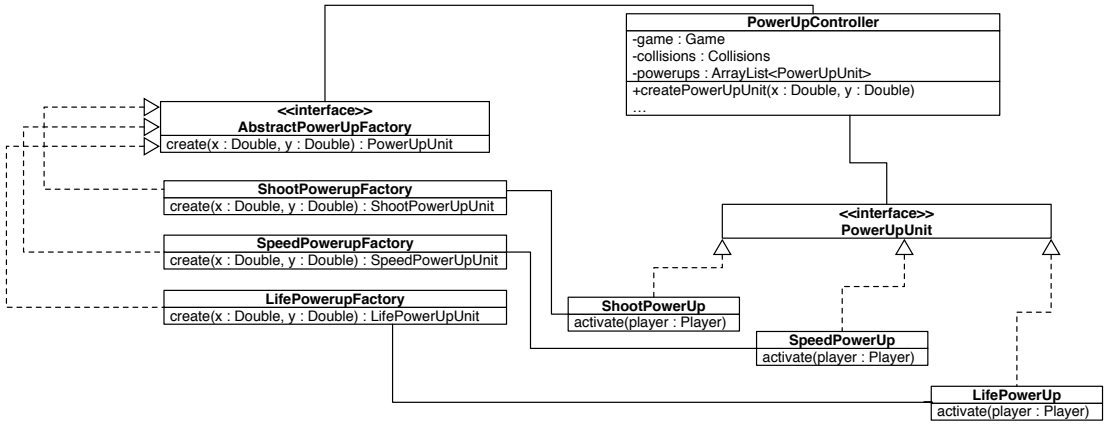

Figure 4: Abstract Factory Pattern UML Class Diagram

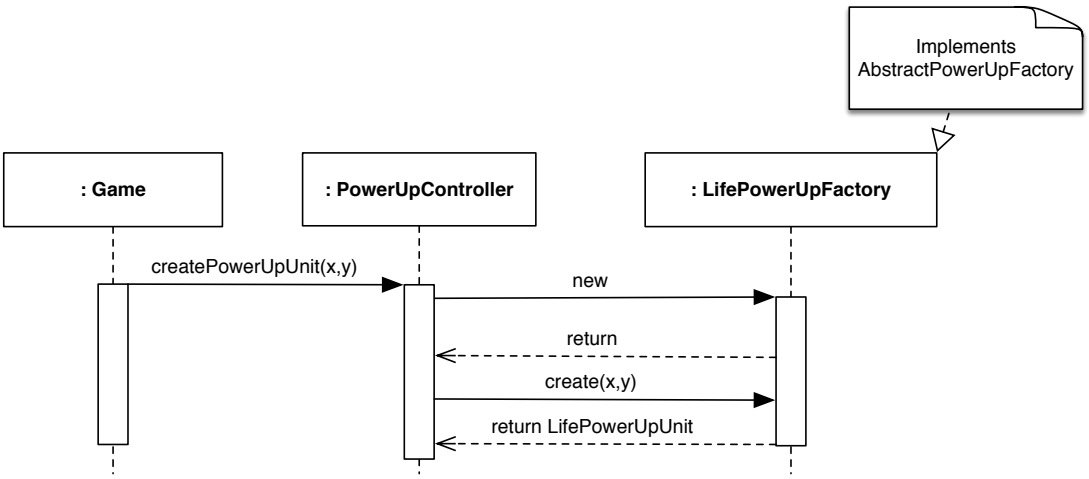

Figure 5: Abstract Factory Pattern UML Sequence Diagram

3 Exercise 3 - - Software Engineering Economics

3.1 Exercise 3.1 - Good and Bad practices

There are several factors which distinguish good practice software projects with bad practice software projects. The most important factors for good practices are that it is an fixed and experienced team, it is released based and the project has a steady heartbeat. The latter means that the project has a fixed length iteration as short as you can make it. All the three factors mentioned are related to an agile way of working. So for a good practice software project, it is very important to work in an agile manner.
The bad practice software projects are projects which are driven by rules and regulations, which have dependencies with other systems, which are technology driven and which are once-only projects. It is therefore very important for a project to have some freedom within the project, the team and the system, because then it is possible to avoid making the project a bad practice project.

3.2 Exercise 3.2 - Visual Basic

Of the Visual Basic projects, the most of those ended up in a good practice project. But it is not so interesting because there were only 6 Visual Basic projects, which is already a small amount of projects within the 352 projects analyzed in the paper. Furthermore, are on average less complex than other projects, therefore end up as a good practice project more often.

3.3 Exercise 3.3 - Good/Bad factors

1. Planned testing
   This belongs to a good practice. It is important to not test at the last moment. It needs planning to make sure that the testing is not only done when the schedule gets tight. It is also important for a project to plan test cases before the coding starts. This all makes sure that members in a project do not waste a lot of time in fixing many tests at the same time. Planning the tests will save a lot of time.

2. Code review
   This belongs to a good practice. A lot of problems are eliminated earlier when a project team regularly reviews each others code. It is important to review the code on bugs, design, architecture and everything to make the code better. This may be more effective than testing.

3. Requirements check
   This belongs to a good practice. When a company wants a new program, it is very important as a team to understand exactly what the company wants. When you start the project without checking that it is understood what the company wants, there is a high risk that you don't do it right, which leads to unnecessay work. This takes a lot of time which can be avoided by checking the requirements.

3.4 Exercise 3.4 - Bad practice factors

The 3 bad practice factors which will be discussed are: Many team changes, dependencies with other systems and technology driven.
Many team changes is a factor where, as the name says, some of the team members leave within the project and/or there come new team members within the project. The latter is the biggest problem in a software project, because the member first needs to understand everything of the project. The member needs to, for example, read the code to get an understanding of it. This takes a lot of time to let the new member be a good part of the team. When a person leaves the project, it also takes time to take over the work of that person and to get an understanding what the member was working on. So when a team change occures, it takes time for one or more members to understand a new part.
When a project has dependencies with other systems, it is depended on the working of those systems, without the possibility to modify the systems. Therefore, when a system fails to work properly, the project members are not able to work on the parts of the project which are depended on that system. And because the members are not able to alter the system, they don't have control over the failing of the system. This will cost a lot of time, because when such problems occure, the members are not able to work on that part of the project.
Technology driven projects are projects where the project is driven by the newest technology to use, almost forgetting that the demand for that technology may not be in existence. This will cause that the members of the project will put effort and time in a part of the project, which is not demanded. Furthermore, because the technology may be that new that there is no possibility to know all capabilities of that technology, there can be no sustainable continuation plan.