Space Invaders : Sprint 1 Assignment

Group 22
TI2206 Software Engineering Methods

Ege de Bruin
Bryan van Wijk
Dorian de Koning
Jochem Lugtenburg

September 18, 2015

Supervisor: Dr. A. Bacchelli
TA: Danny Plenge

Delft University of Technology
Faculty of EEMCS

1 Exercise 1 - The Core

Exercise 1.1

During this exercise, responsibility driven design was followed, starting with the requirements.
The first classes considered candidate classes are the following:

- Game
- Alien
- Aliengrid
- SpaceShip
- EnemyBullet
- Bullet
- MovementController
- PlayerBullet
- Barricades
- Powerup
- Soundcontroller
- Explosion
- EnemyController
- SpaceshipController
- Main
- Player
- UI
- UIController
- Bulletcontroller

For these candidate classes, CRC cards were made and distributed to the team.
By walking through scenarios unnecessary, missing classes and responsibilities were uncovered.
Finally the following list was created, listing classes with their responsibilities and collaborations:

Class	Responsibility	Collaborates with	Super	Sub
Game	All units in the game	all the controllers		
SpaceShipController	Activities of the spaceship	Game		
SpaceShip	Shooting a bullet	SpaceShipController	Unit	
ShipBullet		bulletController	Bullet	
Player	The score and lives of the player	Game		
AlienController	The movements of the aliens	Game		
Alien	Shooting a bullet	AlienController	Unit	
AlienBullet		Bullet	Bullet	
BulletController	Movements of the bullets	Game		
Bullet		BulletController	Unit	AlienBullet ShipBullet
ExplosionController	The development of the explosions	Game		
Explosion	Counter of the explosion	ExplosionController	Unit	
BarricadeController	The changes of the barricades	Game		
Barricade	Health of the barricade	BarricadeController	Unit	
Unit	Location, seize and speed of a unit			Alien, Explosion, Barricade, Bullet, SpaceShip
Collisions	The collisions between all types of units	Game		
Draw	The drawings of elements on the screen	SpaceInvadersUI		
Drawbarricade	The drawings of the barricades	Draw		
DrawBullet	The drawings of the bullets	draw		
DrawSpaceShip	The drawings of the spaceship	draw		
DrawAlien	The drawings of the aliens			
SpaceInvadersUI	The presentation of the game on the screen	GameUIController		
GameUIController	The keyboard inputs and refreshing the frames	SpaceInvadersUI		
Main	The launch of the game	Game, GameUIController		

The main difference with the actual implementation is that that Game and GameUIController have much more responsibilities.
It would have been better if these classes were split into smaller classes as in the responsibility driven design of classes as above. This will make small code changes easier, because

they only affect small classes, instead of causing a rewrite of a large class.
Classes that could be added to improve this are the Draw and control classes.

Exercise 1.2

The main classes implemented in the actual implementation are the Game and GameUIController. The game class has the responsibility to handle the movements of the spaceship and the bullets, the creation of barricades and to monitor the game status.
This class also handles the lists of all the units in the game. It collaborates with the AlienController, Collisions and Player class.
Another main class implemented in the actual implementation is the GameUIController which deals with the drawings of all the elements and animation, loading of the sprites, creation of a new game, and the key inputs. This class collaborates with the game and SpaceInvadersUI classes.

Exercise 1.3

Some of the classes are less important as the classes mentioned in previous exercise, because they have less responsibilities and could be merged more easy with another class.
They are also a lot smaller what makes it easier to merge them in another class. For example we have a ShipBullet and an AlienBullet class which are subclasses of the Bullet class.
The ShipBullet and AlienBullet class could have been merged together in the bullet class, The reason this has not been done is because the product is still under development.
It is now possible to use instances to check if a bullet is a player bullet or an alien bullet. This construction also makes it easier to add attributes or methods to the alien or ship bullet only.

Exercise 1.4

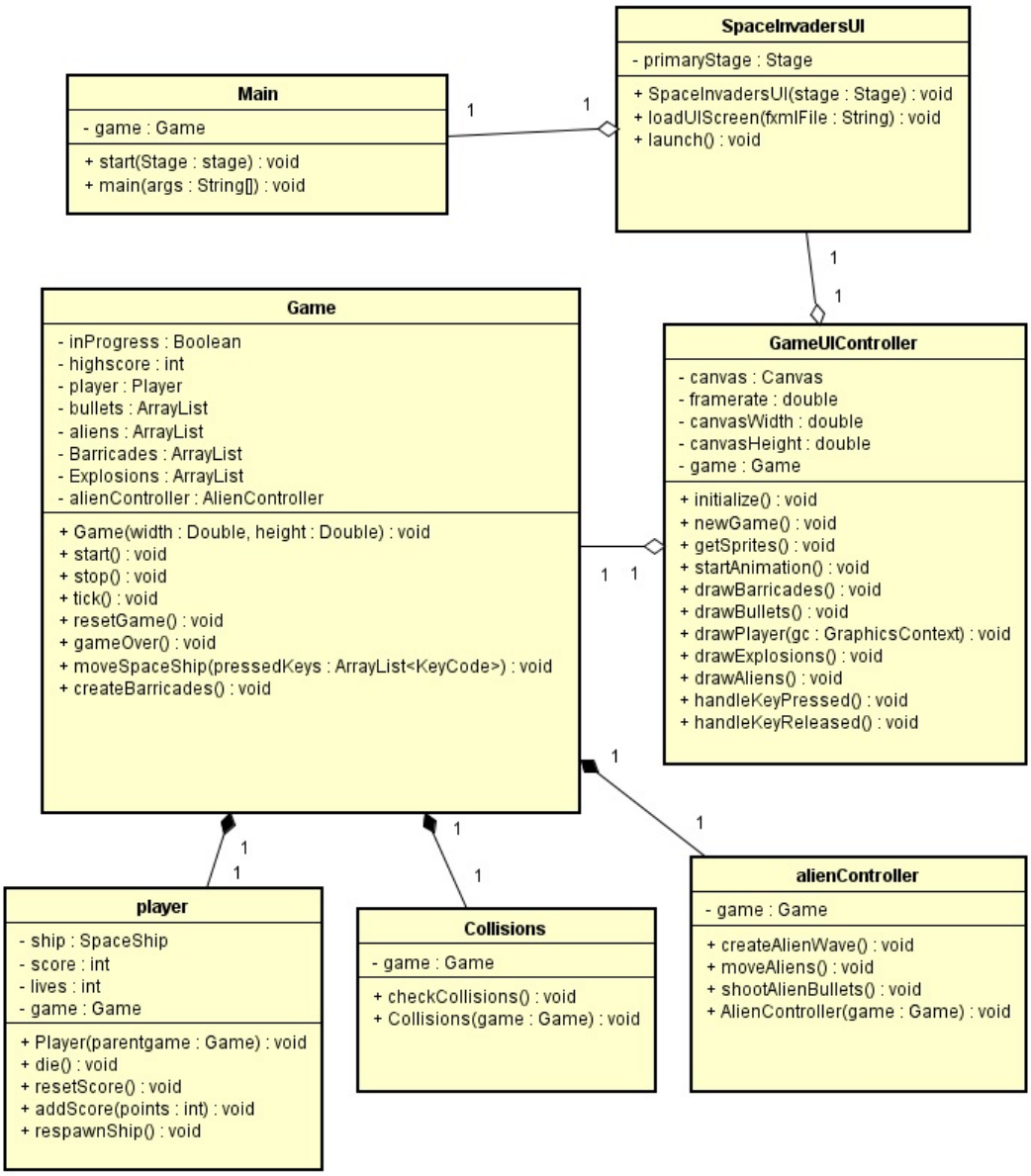

Figure 1: Main classes UML Class Diagram

Exercise 1.5

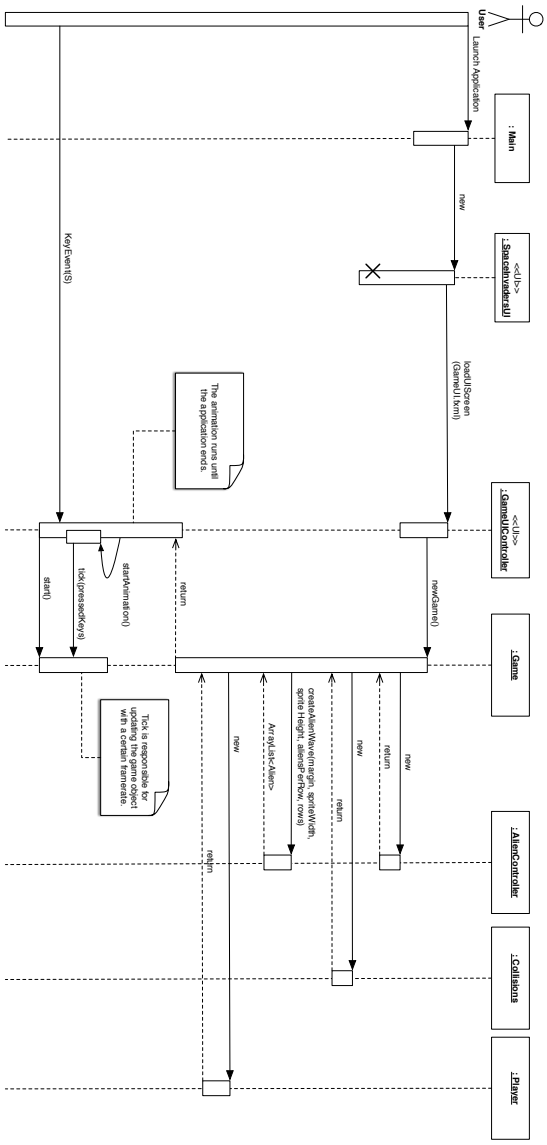

Figure 2: Main classes UML Sequence Diagram

This diagram contains the scenario of launching the game, until the player presses S to start the game.

2 Exercise 2 - UML in practice

Exercise 2.1

UML Aggregation applies to a "Consists of" relation. Parts of the whole may be shared, which means that child classes can exist independent of the parent class. If the parent is destroyed, the child can remain without the parent.

UML Composition applies to a "Contains" relation. A part belongs to a whole. If the parent class does not exist, the child cannot exist separate of the parent. If the parent is destroyed, the child will also be destroyed.

Aggregation is used in the relations between the Game class and classes which assist the Game class. A Game consists of multiple Units (players, aliens, barricades, explosions and bullets), Collisions and the alienController. These are a composition relation, because without these classes, it is not possible to play the game, thus making the Game class useless.

Exercise 2.2

Parameterized, generic classes use type parameters which enable the user to re-use code for a different object type.

An example of a parametrized class used in the project is the java.util.ArrayList class. It takes an input Type, and allows creation of a list, reusing the same code for every object type.

The project itself does not contain classes using Parameterized classes.

Exercise 2.3

The project contains one major hierarchy, treating the different game Units. Within the hierarchy there is a sub-hierarchy for bullets. There are two bullet types, one for aliens and one for the player. The subclasses directly linked to Unit are "Polymorphism" relations, since the behavior of each unit is different. The subclasses of Bullet are "Is-A" relations, since they have the same behavior as their superclass.

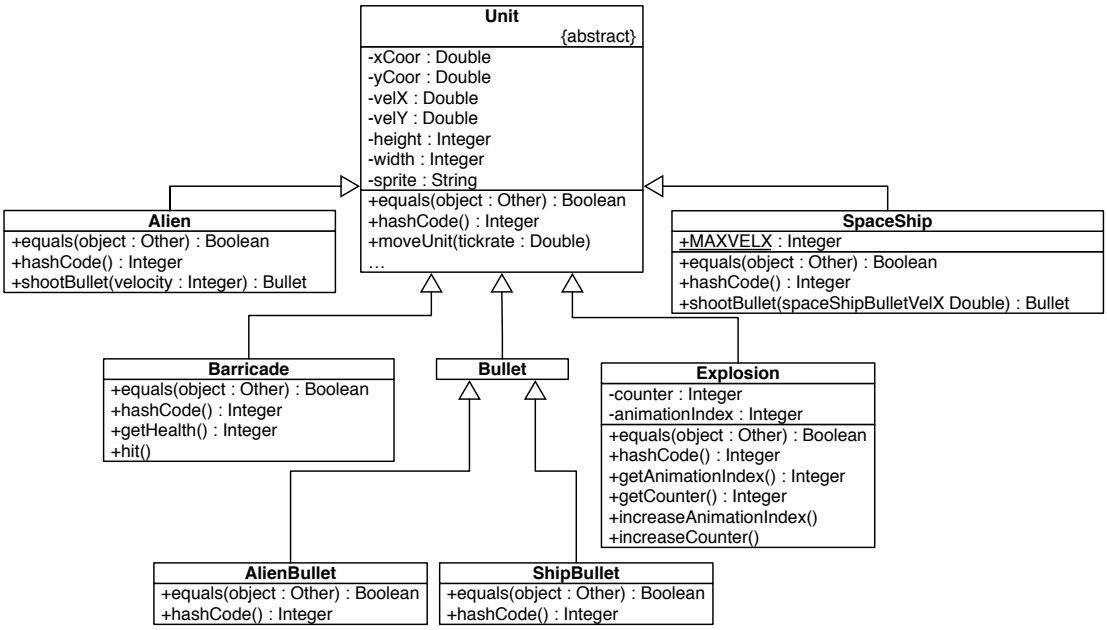

Figure 3: Unit hierarchy UML Class Diagram

Excercise 3 - Simple Logging

Exercise 3.1

Responsibility driven design was used as a tool to develop the logger. Starting with the requirements the following classes could be added to the product:

- Logger
- LogPlayer
- LogAliens
- LogBullets
- LogBarricades
- LogExplosions
- LogExceptions
- LogErrors
- WriteLog
- ReadLog

Next the classes to be used were selected from the list. The following classes will be used:

- Logger
- WriteLog

These classes were added to a table, listing their responsibilities:

Class	Responsibility	Collaborates with	Super	Sub
Logger	Log all the actions in the game	WriteLog		
WriteLog	Write the log to a file	Logger		

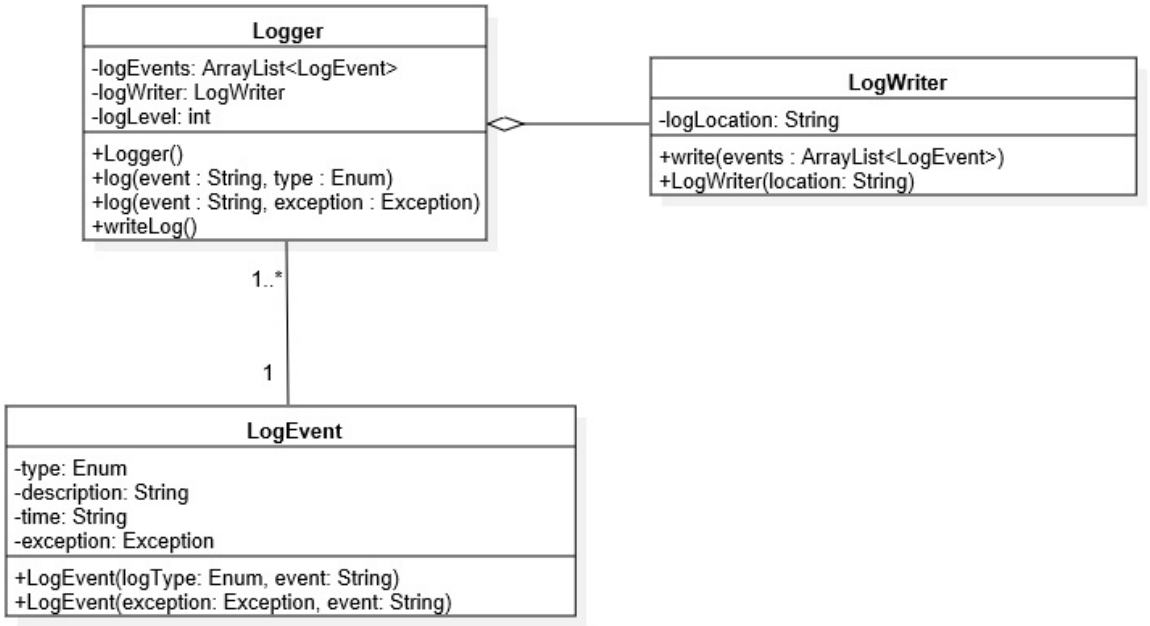

Figure 4: Logger UML Class Diagram

Logger Functional Requirements

A list of functional requirements considered for the extension of the game with a logger using the MoSCoW method described in the previous section.

Must Haves

The Logger of the game must meet the following requirements:

- The game shall be able to log all actions happening during execution the game.
- The logger shall be able to write the log to a file.

Should Haves

The Logger of the game should meet the following requirements:

- The logger shall keep track of the time each action is executed.
- The logger shall have different logging levels.
- The logger shall be able to log exceptions thrown.
- The logger shall be able to log errors.

Could Haves

The Logger of the game could meet the following requirements:

- ...

Would/Won't Haves

The Logger of the game won't meet the following requirements:

- The logger shall use text colors in the output log.