Space Invaders : Sprint 5 Assignment

Group 22
TI2206 Software Engineering Methods

Ege de Bruin
Bryan van Wijk
Dorian de Koning
Jochem Lugtenburg

October 23, 2015

Supervisor: Dr. A. Bacchelli
TA: Danny Plenge

Delft University of Technology
Faculty of EEMCS

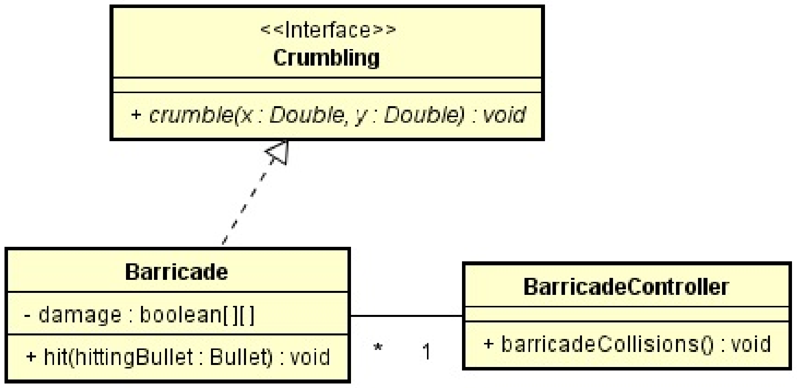
1 Exercise 1 - 20-times, Revolutions

1.1 Crumbling barricades

To implemented this the barricade class has to be changed. This class needs to extend the crumbling interface and implements it methods. The UIElementBarricade also needs to be changed to support drawing crumbled barricades. The third and last class that needs to be changed is the Collisions class so the data about the location where a barricade is hit can be used for crumbling the barricade. The collisons class also has to support collisions with partly crumbled barricades.

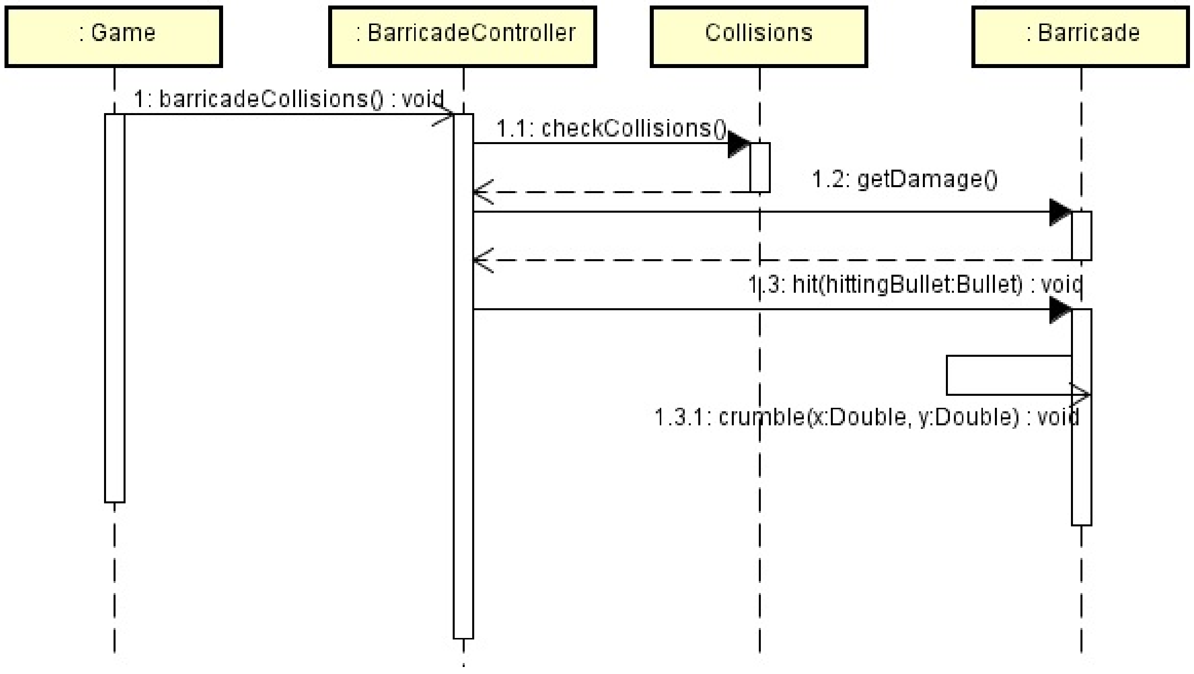
1.1.1 UML Crumbling barricades

Figure 1 contains the UML class diagram for the crumbling barricades. To make the working of the crumbling barricade more clear figure Figure 2 contains the sequence diagram concerning the crumbling barricades.



<<Interface>>
Crumbling
+ crumble(x : Double, y : Double) : void
Barricade
- damage : boolean[][]
+ hit(hittingBullet : Bullet) : void
*
1
BarricadeController
+ barricadeCollisions() : void

Figure 1: Barricade UML class Diagram



: Game
: BarricadeController
Collisions
: Barricade
1: barricadeCollisions() : void
1.1: checkCollisions()
1.2: getDamage()
1.3: hit(hittingBullet:Bullet) : void
1.3.1: crumble(x:Double, y:Double) : void

Figure 2: Barricade UML class Diagram

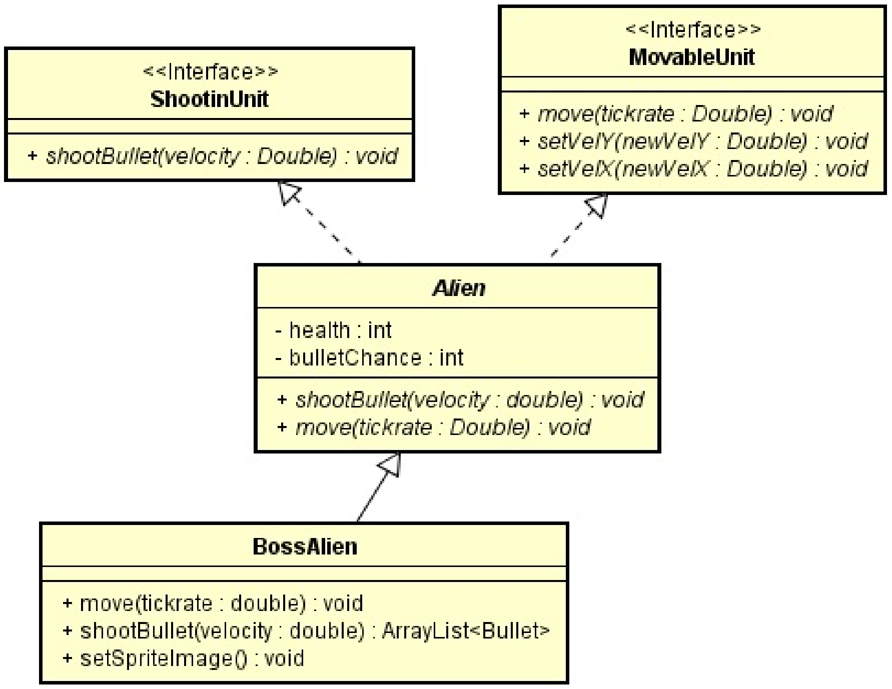
1.2 Boss Alien

Class	Responsibility	Collaborates with	Super	Sub
BossAlien	Shooting, movement		Alien	

The game has an integer which remembers in which wave we are so on the n-th wave the special bose wave can be loaded by the wavecontroller. A new wave with the bose alien has to be created. The wavepattern reader has to be changed to read a bossalien.

1.2.1 BossAlien UML

Figure 3 contains the UML class diagram for the boss aliens.



<<Interface>>
ShootinUnit
+ shootBullet(velocity : Double) : void
<<Interface>>
MovableUnit
+ move(tickrate : Double) : void
+ setVelY(newVelY : Double) : void
+ setVelX(newVelX : Double) : void
Alien
- health : int
- bulletChance : int
+ shootBullet(velocity : double) : void
+ move(tickrate : Double) : void
BossAlien
+ move(tickrate : double) : void
+ shootBullet(velocity : double) : ArrayList<Bullet>
+ setSpriteImage() : void

Figure 3: BossAlien UML class Diagram

1.3 Sound effects

Class	Responsibility	Collaborates with	Super	Sub
SoundController	Playing the sounds	observable classes that could play sounds		
SoundLoader	Load the sounds	SoundController		

The classes which perform actions that are assosiated with sounds has to implement the observable pattern. The SoundController will implement the observer pattern.

1.3.1 Sound effects UML

The UML concerning the sounde effects:

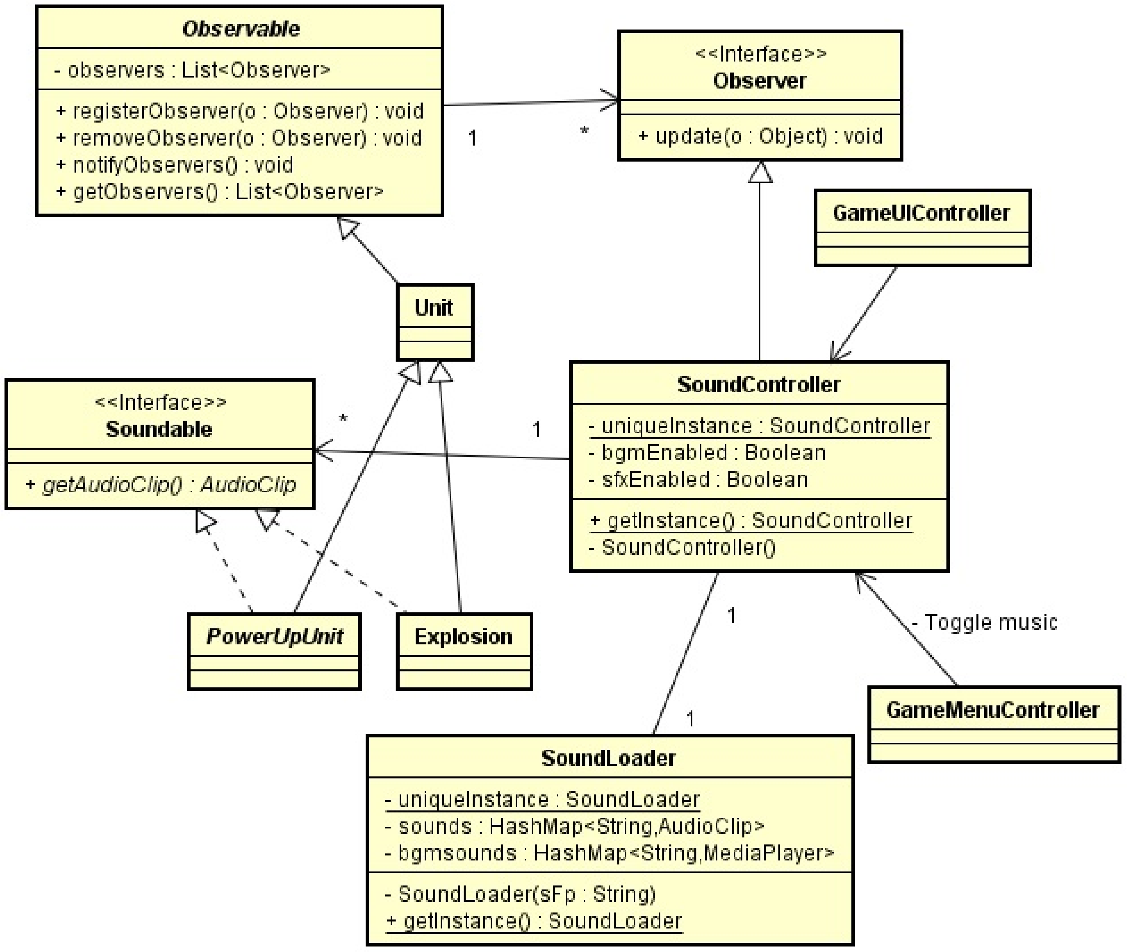

Observable
- observers : List<Observer>
+ registerObserver(o : Observer) : void
+ removeObserver(o : Observer) : void
+ notifyObservers() : void
+ getObservers() : List<Observer>
<<Interface>>
Observer
+ update(o : Object) : void
1
*
GameUIController
Unit
<<Interface>>
Soundable
+ getAudioClip() : AudioClip
*
1
SoundController
- uniqueInstance : SoundController
- bgmEnabled : Boolean
- sfxEnabled : Boolean
+ getInstance() : SoundController
- SoundController()
1
- Toggle music
PowerUpUnit
Explosion
1
GameMenuController
SoundLoader
- uniqueInstance : SoundLoader
- sounds : HashMap<String,AudioClip>
- bgmsounds : HashMap<String,MediaPlayer>
- SoundLoader(sFp : String)
+ getInstance() : SoundLoader

Figure 4: Sounds UML class Diagram

2 Exercise 2 - Design patterns

2.1 Exercise 2.1 - Singleton

Before this sprint iteration, the project already contained an implementation of the singleton design pattern. The implementation of the singleton pattern is in the logger class. The singleton pattern ensures a class has only one instance. The singleton pattern also gives a global point to access this instance. The advantage of using a singleton pattern, is that you ensure that not two of the same objects are created. This could be important for object that only could have one instance. This is for example as used in this project the logger class, but could also be another class from which only one instance should be created. The singleton pattern is implemented by making the constructor of this class private and creating a seperate getInstance mathod. This way you can ensure there is always one unique object of this class. The getInstance method and the logger instance variable are static, because they are used in a static way. Problems that could occur with this pattern is that you still can create two instances with singleton. This can happen when you use conccurent programming. By adding the volatile keyword it is ensured that threads handle the unique instance variable correctly when it is being initialized to the singleton instance. Another way to prevent this problem is by adding the synchronized keyword to the getInstance method. This forces every thread to wait its turn before it can enter the method.

2.2 Exercise 2.2 - Singleton

Figure 5 contains the UML Class diagram for the singleton design pattern concerning the logger.

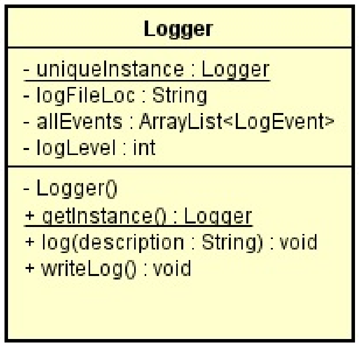


Figure 5: Singleton UML Class Diagram

2.3 Exercise 2.3 - Singleton

Figure 6 contains the UML Sequence diagram for the singleton design pattern in the logger. It contains two calls of the getInstance method from two example classes. In the first call the unique instance of the logger is created. In the second call the instance already exists so this is simple returned.

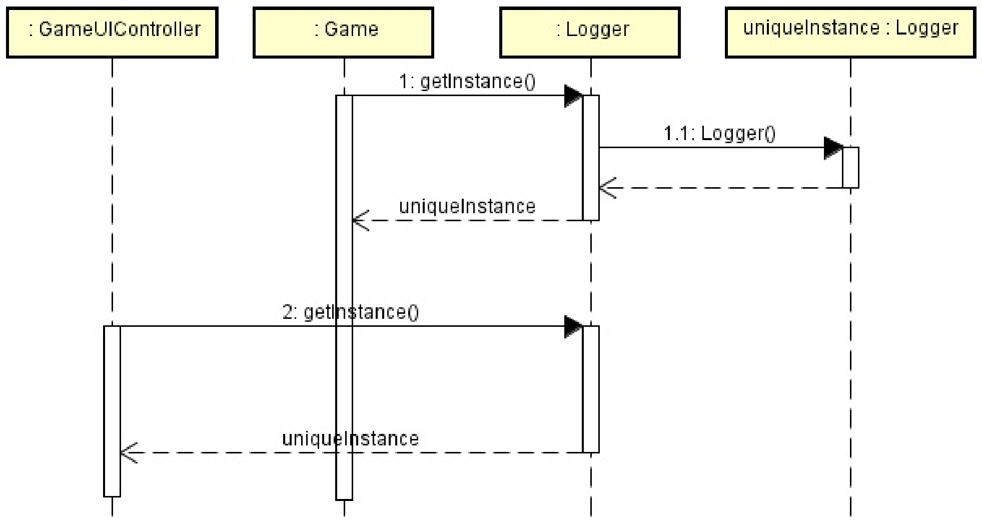


Figure 6: Singleton UML Sequence Diagram

2.4 Exercise 2.1 - Observer

In this sprint sound effects were created, for this new feature the observer pattern is used. The observer pattern ensures, when an observable object is changed the observer is notified. In the implementation for the sound effects is the soundcontroller the observer and the objects that could generate a sound by change the observables. This are for example the explosion objects which should generate an explosion sound. To implement this roles the observer interface and the Observable abstract method is used. An advantage of using the observer pattern instead of adding a method call to the soundcontroller in the the explosion class is that this way objects are loosely coupled. The objects can interact with each other but have very little knowledge of each other. It is also easy to add new observers withoud changing the observable class, so also other classes can be notified on a change.

2.5 Exercise 2.2 - Observer

Figure 4 contains the UML Class diagram for the observer design pattern concerning the sound effects.

2.6 Exercise 2.3 - Observer

Figure 7 contains the UML sequence diagram for the observer design pattern concerning the sound effects.

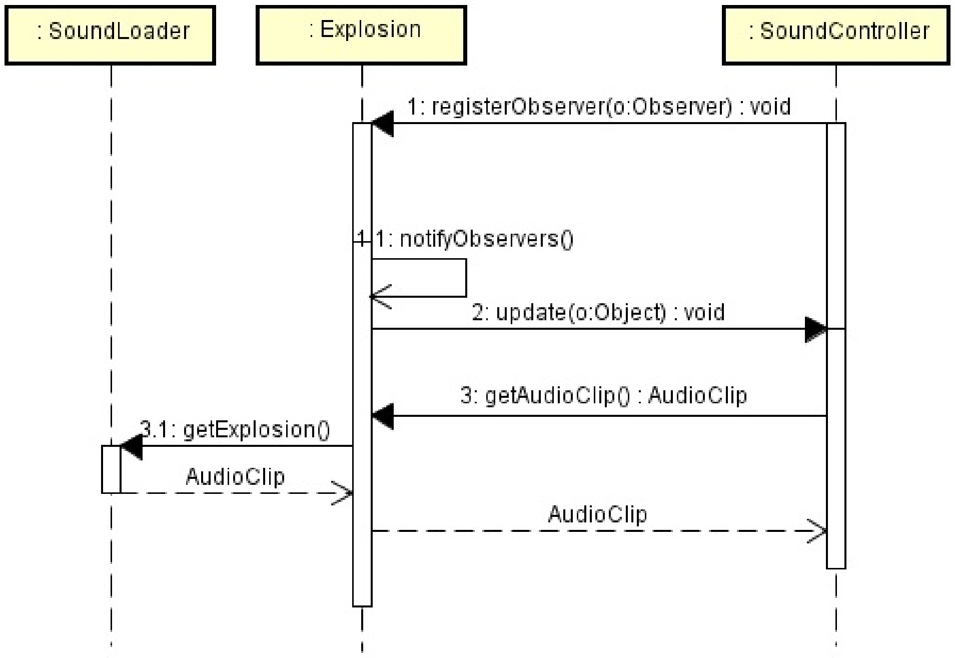


Figure 7: Sounds UML Sequence Diagram

3 Wrap up - reflection

Introduction

This document contains a reflection on the evolution of the source code and the route followed to create the final version. This essay covers how the team has worked together, and what has been learned. It will also reflect on what went well and what could be improved in a future project. To describe how the system has changed, the first version of the game is compared to the final product. Finally this document contains a conclusion with the knowledge gained from the course and this lab project.

Reflection

Requirements

In the early stages of the project, selection of the game to implement took place during the first lab hours. After selecting Space Invaders as the game to implement, the team started defining requirements using the MoSCoW method. During the project these requirements were extended, as new features were added to the game. Defining requirements in a clear and ambiguous way helped to distinguish the various tasks to implement, without requiring long discussions about how to implement a certain feature. During the following sprints, the requirements were defined after the creation of the weekly sprint plan.

Team

As a team, the project was started with five persons. Losing one person during the first iteration, the team had to carry out the work of one extra missing person. Because of this, the team quickly realized it had to maintain a clear organization. Making sure the organization of the project was defined using clear documents and following SCRUM, features could be implemented relatively fast with minimal overhead. There were also no problems with tasks that had not been done, or were incomplete so the team did not have to do work of a previous sprint in the following sprint. This resulted in the fact that the project constantly remained on schedule.

Sprint Iterations

The sprints were carried out using the SCRUM methodology where each iteration consisted of a meeting on Tuesday and a daily status report using the Whatsapp messaging service. Each Tuesday meeting started with an evaluation session of the previous sprint, resulting in a sprint reflection document. During this evaluation the team focused on improving the weekly iterations. Among the most important things the team improved were time management, making it more easy to approximate the required time for a task. Also the addition of smaller features to the game became an improvement to the process. The team discovered that implementing small features increased the quality of the product. Large features or major code refactors, such as a refactor of sprite loading, increased the amount of bugs on the final day of each sprint, causing risks to handing in a stable deliverable. During the final sprint, Code Review was also improved. Since a huge amount of Checkstyle errors slipped into the stable branch, the team decided not to merge pull requests containing Checkstyle errors.

After evaluation the new sprint was defined in a Sprint Plan which clearly states the tasks required to solve the weekly assignments. As more iterations followed, the Sprint Plan was improved with prioritization and a more clear definition of the responsibility per task. After defining the sprint plan the team began working on individual tasks together during the remaining lab hours. Each team member kept in touch using Whatsapp on a daily basis, posting the performed tasks, todo tasks and problems that occurred.

Code Review

Each feature addition or refactor was merged into a develop branch using pull requests. In these pull requests at least two team members had to approve the code change. The code changes were reviewed on implementation, CheckStyle and understanding of the code.

Code Review helped the team understand each others code, solved a small amount of bugs but most of all increased readability and clearness of the code. Before handing in the final version for a sprint, the develop branch was merged in to the master branch. The team decided to work with a separate develop branch, to make sure the master branch always contains a well-tested, stable version of the game.

Implementation

During the implementation of the product the team focused on maintainability and easy extension of the game. This payed off in later iterations, because it became very easy to add new features to the game, such as the Boss level using Waves. Design patterns such as the strategy pattern made it easier to achieve our goal of easy extension of the game. Comparing the first version to the current master branch, it becomes clear that a lot of features have been added.

Comparing version v1.0 of the game, it becomes clear this version had a good performance, and high test coverage. The current master branch also performs well, and has high test coverage, but the system has grown in size. The system is still extendable and maintainable in an easy way. Running inCode on the first version of the game, detects the Game class as a God class. In the current master branch this has been solved by sharing responsibilities over different classes.

Sharing responsibilities makes it more easy to extend and maintain the game in the way it is intended. This proves that the use of tools such as inCode help detect these problems in an early stage. While the God class design flaw was resolved in another iteration, this design flaw could have been detected before the first iteration using a code analysis tool, increasing code quality. The first version of the game also didn't make use of design patterns, which could have helped to make the code more extendable, which was solved later by for example implementing the Shootable and Movable interfaces.

Conclusion

During the Software Engineering Methods lab project, the team learned how to develop a Game by using standards and techniques taught during the lectures. It helped greatly in understanding SCRUM, design patterns, the use of Git(Hub) and the overall organization of a project. Knowledge was gained about methods which help to improve iterations, which can be used to make future projects more easy to manage. It was a very useful lab project, because the team learned how to use the theory covered by the lectures in a real project.