Space Invaders : Sprint 5 Assignment

Group 22
TI2206 Software Engineering Methods

Ege de Bruin
Bryan van Wijk
Dorian de Koning
Jochem Lugtenburg

October 22, 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.

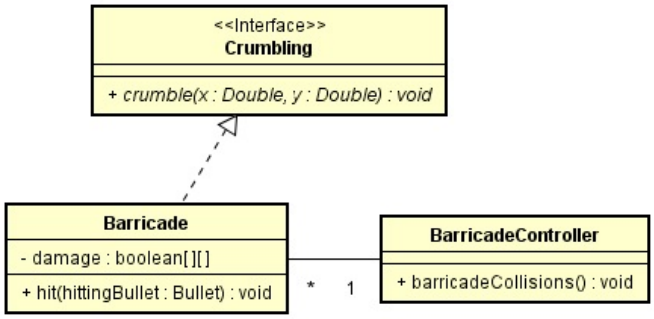

Figure 1: Barricade UML class Diagram

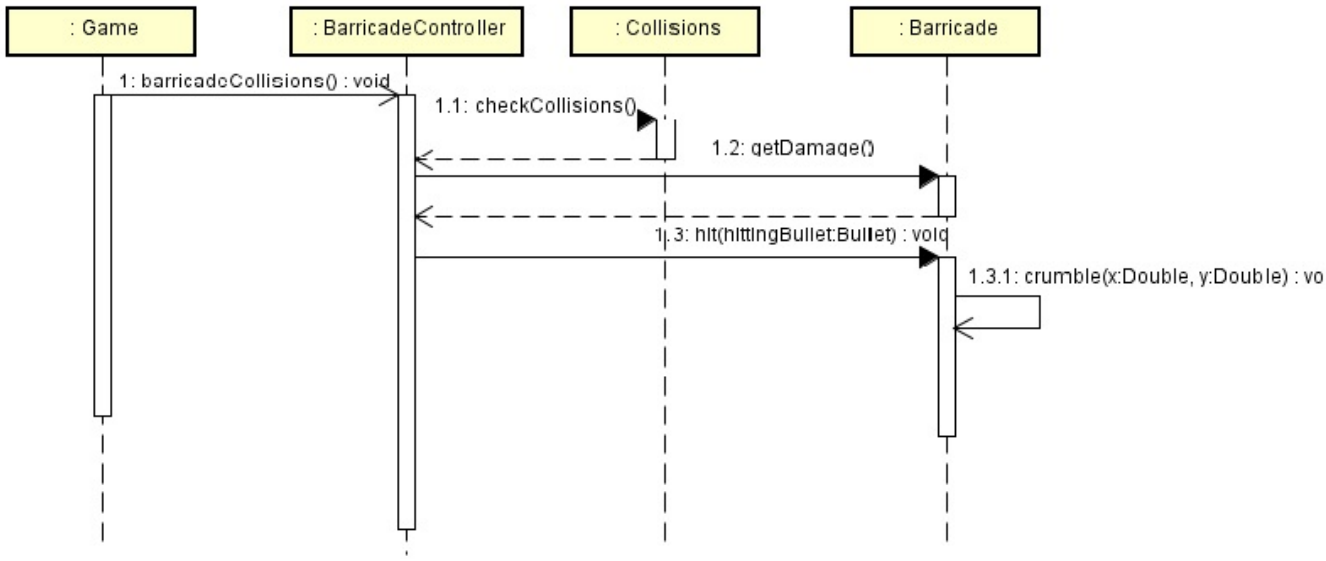

Figure 2: Barricade UML class Diagram

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.

1.3 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:

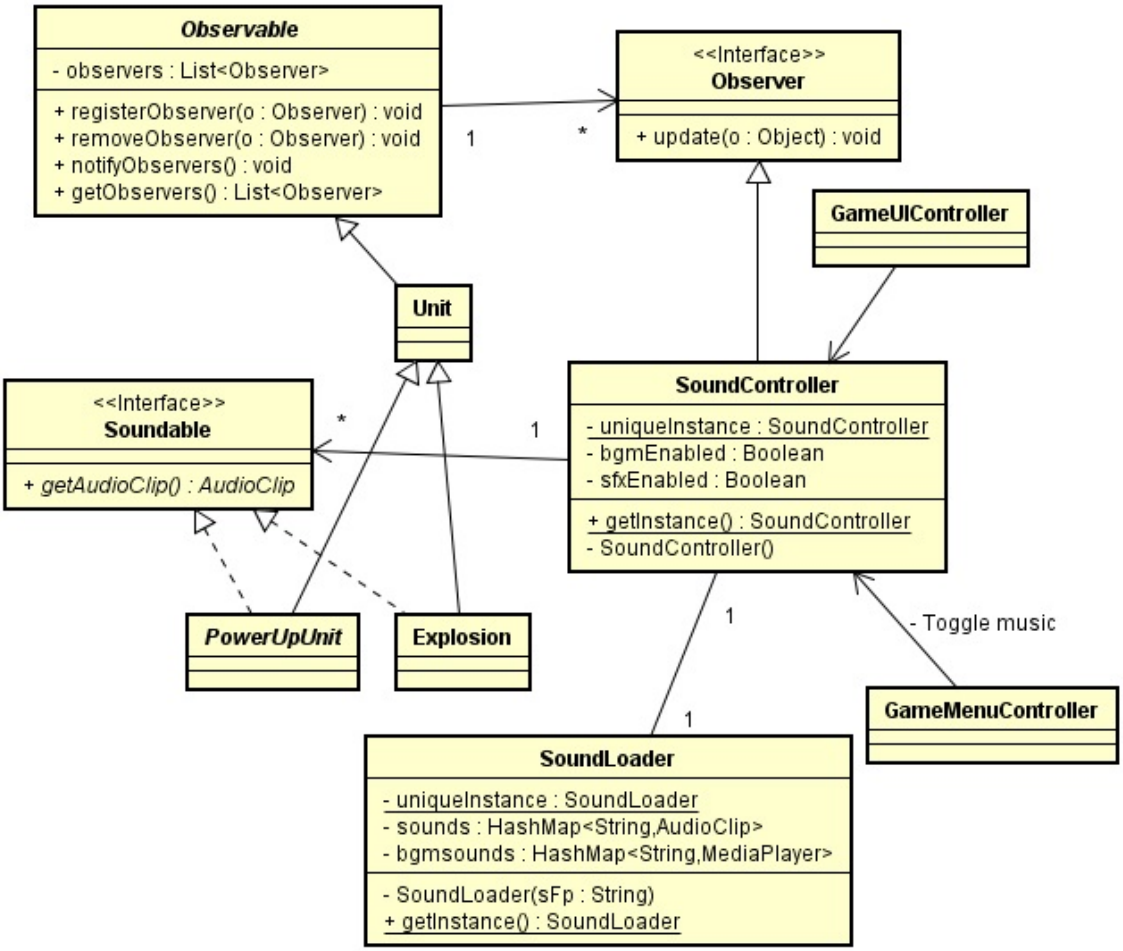

Figure 3: Sounds UML class Diagram

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

Figure 4: Singleton UML Class Diagram

2.3 Exercise 2.3 - Singleton

Figure 5 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.

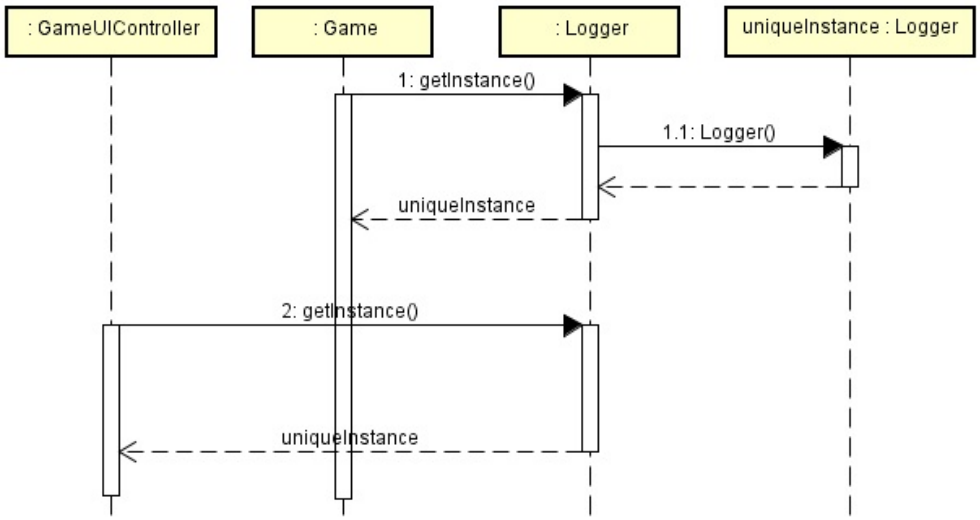

Figure 5: Singleton UML Sequence Diagram

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

2.6 Exercise 2.3 - Observer

Figure 6 contains the UML sequence diagram for the observer design pattern concerning the sound effects.

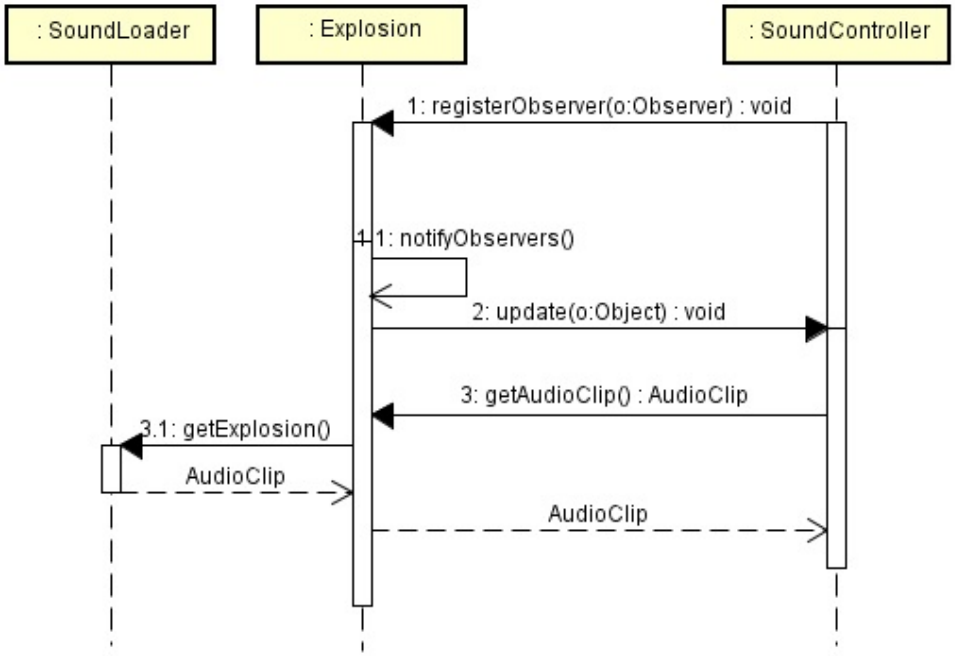

Figure 6: Sounds UML Sequence Diagram

3 Wrap up - reflection