TI2206 Software Engineering
Bubble Shooter report: Assignment 2
EEMCS/EWI

Gerlof Fokkema 4257286
Owen Huang 4317459
Adam Iqbal 4293568
Nando Kartoredjo 4271378
Skip Lentz 4334051

September 27, 2014

Exercise 1: Simple Logging

In this section the process of how logging is implemented in the project is explained fully in depth.

Responbility Driven Design in logging

In order to include logging in the project, it is important to carefully think again about the different responsibilities every class carries. It is also important to know what to log and what not to. Some things are simply not necessary to log; knowing what buttons on the main menu were pressed for example is not something of upmost importance. Other things regarding the game are much more interesting to know or estabilishing a connection with another player in multi-player. Since most game logic is put together in one class called *BSMode*, having all the elements that we wish to observe one could apply the observer pattern to *BSDrawable*. This causes everything that is drawable, including most things that change to be observable. By creating a class *Loggable* which extends the *Java Observable* class, we can therefore let any class that needs to be Observable extend our *Loggable* class. The various observers are then of course added in the *BSMode* class, as earlier mentioned it contains all the crucial game elements (such as cannon and board). Now we introduce another class, called the *Logger* which will act as the Observer. This class will also implement the *Java Observer interface*. It will contain the *update* method, being called when the Observer is notified by its Observables. It is also possible to a step further, to include different kinds of logging: printing to console and to file. For this a new pattern may be introduced, a suitable pattern is for example the *Strategy* pattern. This takes care of being able to include independant kinds of logging. A visual depiction of the implementation of the logger can be found below in **Fig. 1**.

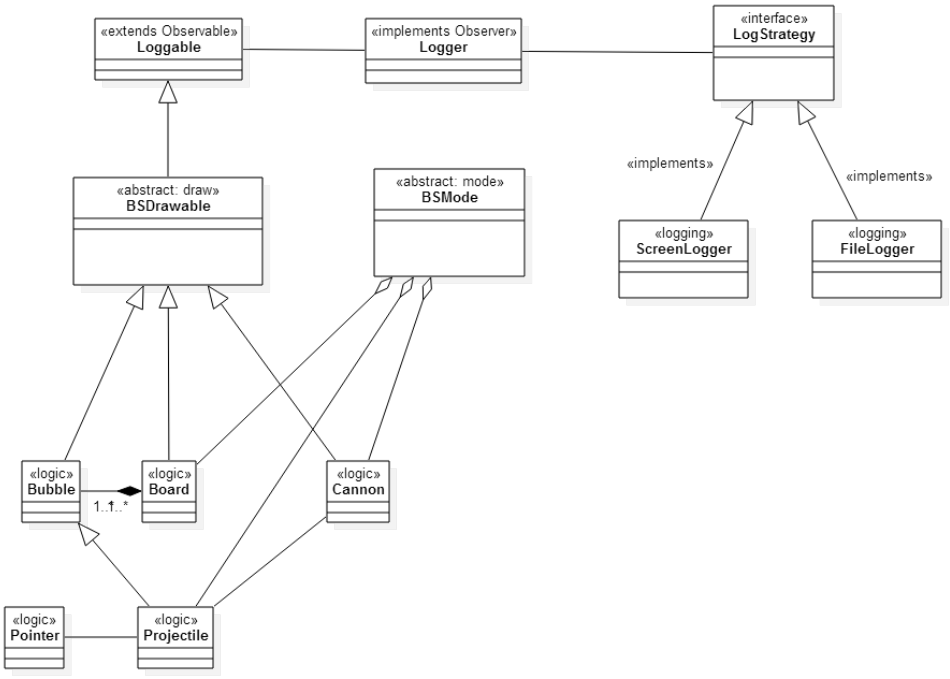

Fig. 1: Global layout of the implementation of the logger

Exercise 2: Design Patterns

In this section three design patterns are elaborated with fully in depth explanations. Furthermore these patterns come with a visual depiction of how they were implemented in the project.

2.1 | Pattern 1: Observer for Statistics

1. For the first design pattern, an *Observer* pattern was chosen for the statistics of the game. For example, a single-player game contains a *score* and a *timer*. These elements are constantly updated throughout the game, given that something has changed about them. A *timer* should count down in a continuous fashion, meanwhile the *score* should only be updated when the user has gained points. These are triggerable events. The timer is triggered by a change in time and the score by removing bubbles. So it makes sense to use something like an *Observer* pattern, since it will only notify the *Observer* when the *Observable* triggered something and notifies the *Observer* about it.

2. The class diagram for the *Stats Observer* can be found in **Fig. 2**.

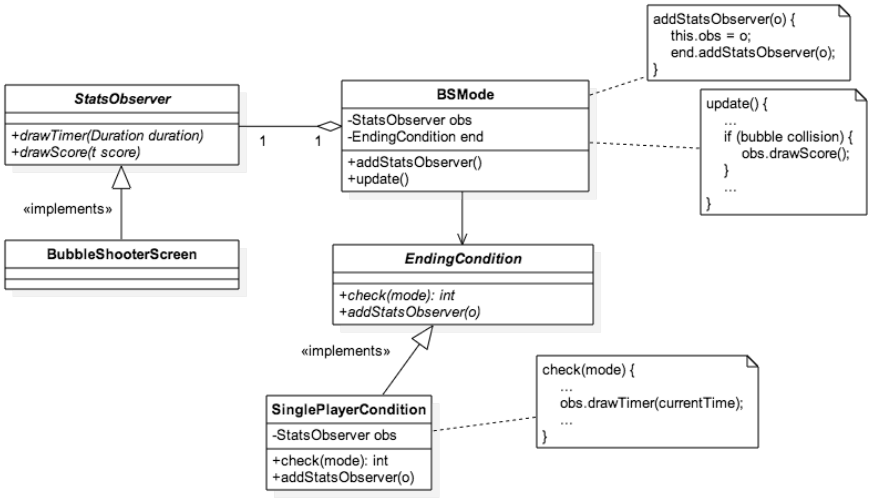

Fig. 2: Stats Observer class diagram

3. The sequence diagram for the *Stats Observer* can be found in **Fig. 3**.

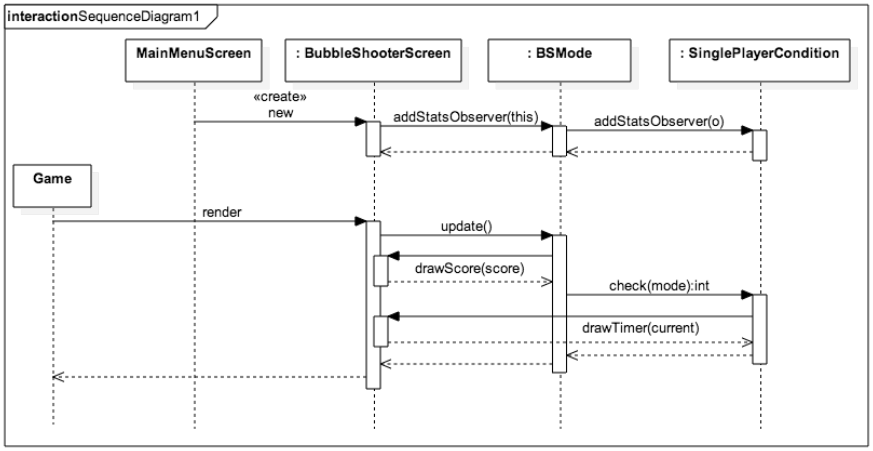

Fig. 3: Stats Observer sequence diagram

2.2 | Pattern 2: Strategy for Ending Condition

1. The second pattern we used is the *Strategy* pattern, which we applied on the *EndingCondition* interface, which is to be used by the *BSMode* class.

   The pattern is applied here because one should favor object composition over inheritance. We could for example have used inheritance, and make a subclass of *BSMode* for each game mode. However, a lot of the game logic is the same, and really the only difference is how the game should end.

   Using the *EndingCondition* class, we get a greater amount of flexibility: we pass the desired *EndingCondition* to the constructor of a *BSMode* class, and automatically the logic for the end of the game is done for us, by means of polymorphism.

2. The class diagram for the *Ending Condition Strategy* can be found in **Fig. 4**.

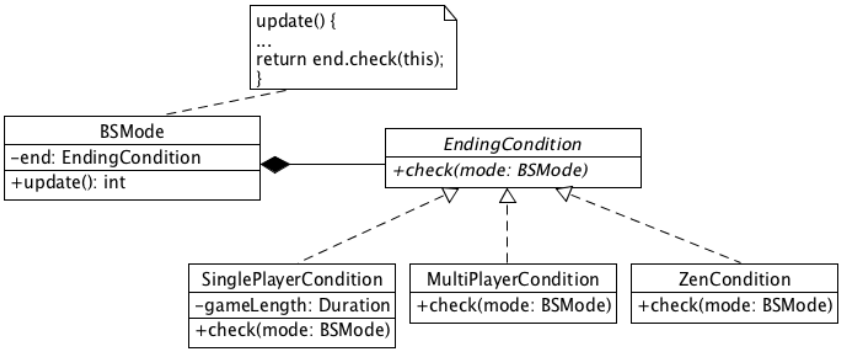

Fig. 4: Ending Condition Strategy class diagram

3.

2.3 | Pattern 3: Singleton for Assets

1. The Singleton pattern makes sure that there is only one instance of *Assets* and it can be accessed globally. This is useful because we only need one instance of *Assets* and we only need to load the assets once with the same instance. The loading happens when the application is initiated, in the very beginning before the user even sees anything. So, by applying the singleton pattern, it provides easy access for the class responsible for the initial launching of the application.

2. The class diagram for the *Assets Singleton* can be found in **Fig. 6**.

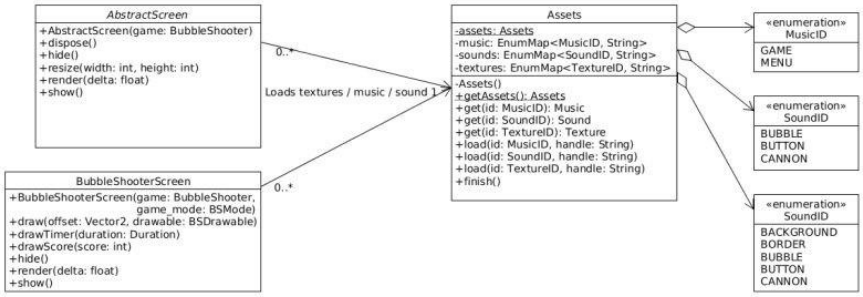

Fig. 5: Assets Singleton class diagram

3. The sequence diagram for the *Assets Singleton* can be found in **Fig. 7**.

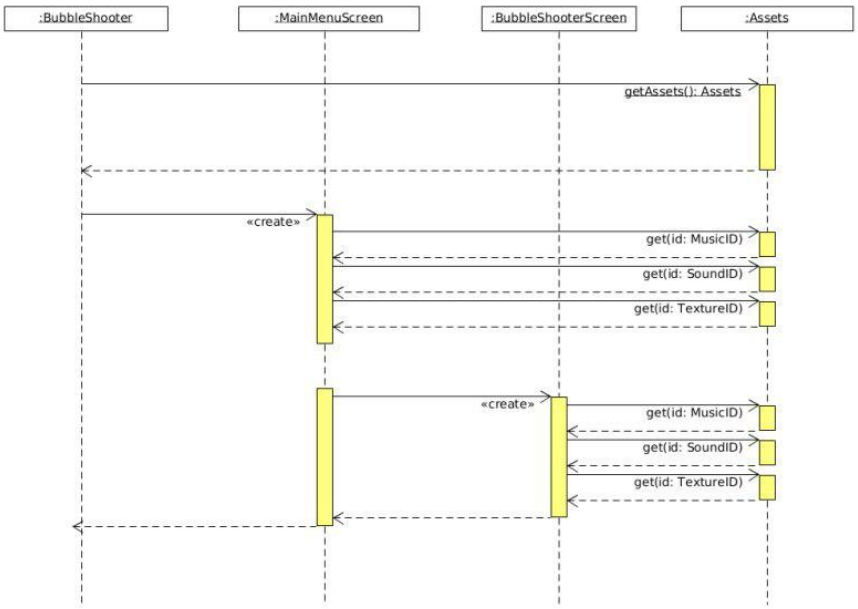

Fig. 6: Assets Singleton class diagram

Exercise 3: Optional - One more design pattern

In this section an additional design pattern is implemented and elaborated.

3.1 | Pattern 3: Observer for Networking

1. For the third pattern, we chose to use the *Observer* pattern again to improve our networking code.
   When playing against another player, the remote client only has to know about changes in the current state of the host. Therefore it makes sense to use the *Observer* pattern for sending our state to the remote client.
   We were able to improve networking performance by changing all *BSDrawable* objects into *Observable* objects and by letting the *MultiPlayerMode* class listen to changes made to them. Now the *MultiPlayerMode* only sends *BSDrawable* objects over the network when they have changed.

2. The class diagram for the *Networking Observer* can be found in **Fig. TBD**.

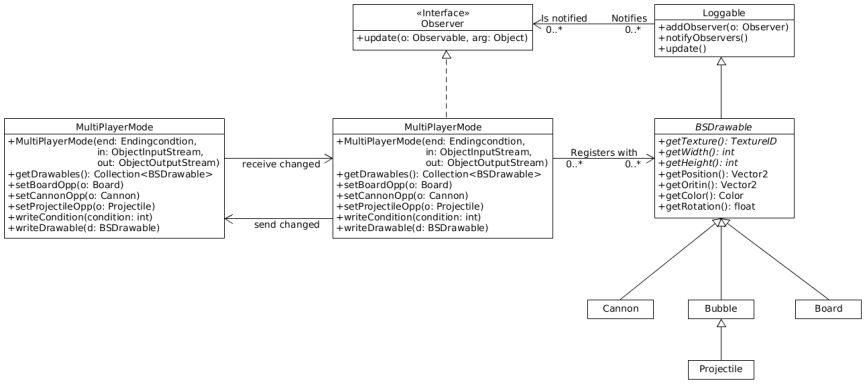

Fig. 7: Networking Observer class diagram

3.