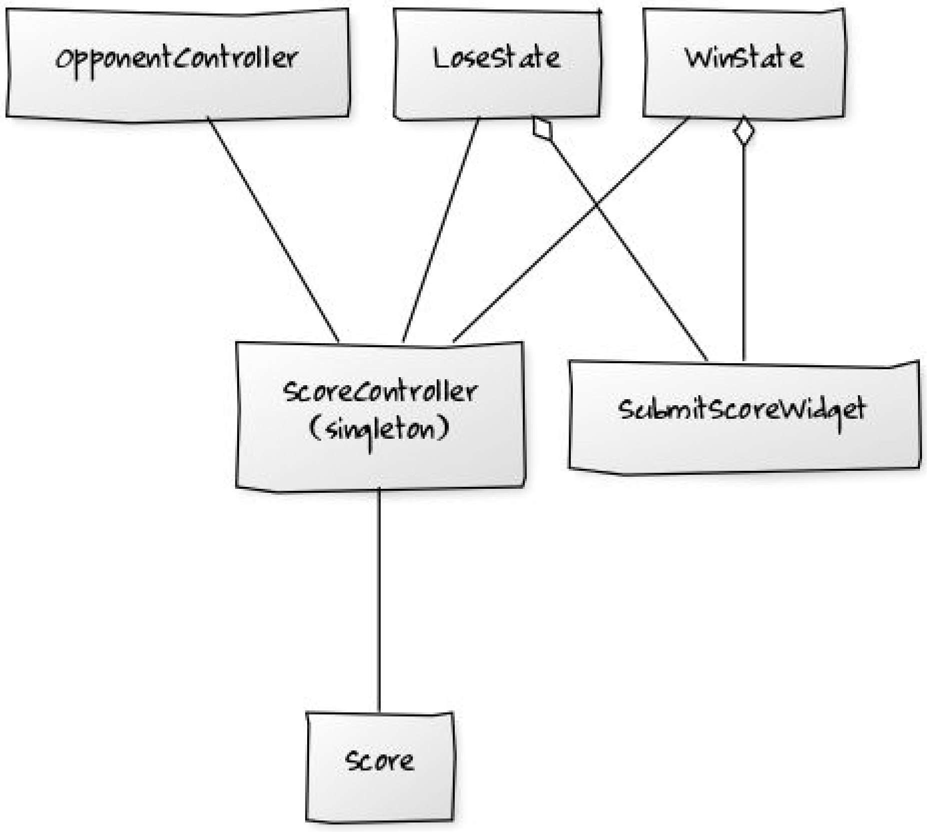
Group 11 Assignment #5

Exercise 1

UML for the high score system



Exercise 2

Factory Pattern

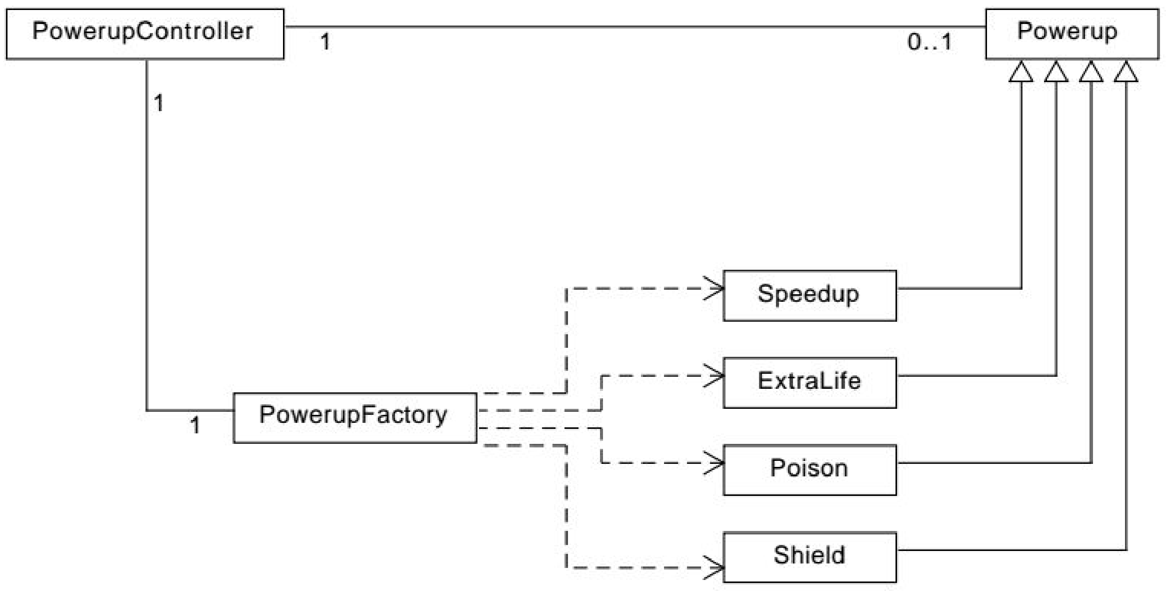
Natural Language Description

We implemented the factory pattern for the power ups because when the spawn method in *PowerupController* was called one of 5 different objects should return, 4 different power ups and a null object.

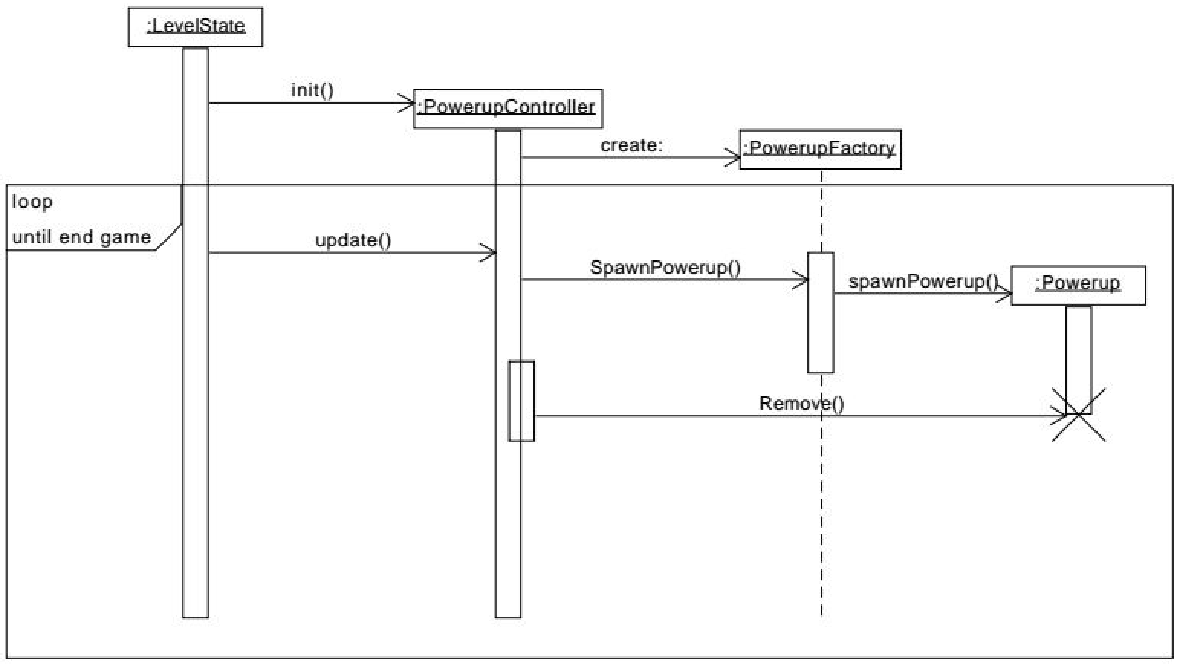
We implemented it so that the *PowerupController* would create a Factory and when needed would call the *spawnPowerup* method. Because we want our power up to be random we don't need to give parameters to the factory for it chooses the power up at random in the spawn method of *PowerupFactory*.

When the factory's spawn method is called I creates 4 different power ups causing these to be random. These power ups are then put in an *arrayList* where a random index is picked. Each power up has its own spawn chance so when the random number between 0-1000 is below this chance, the power up is spawned. Else a null object is returned and nothing is spawned.

Class Diagram



Sequence Diagram

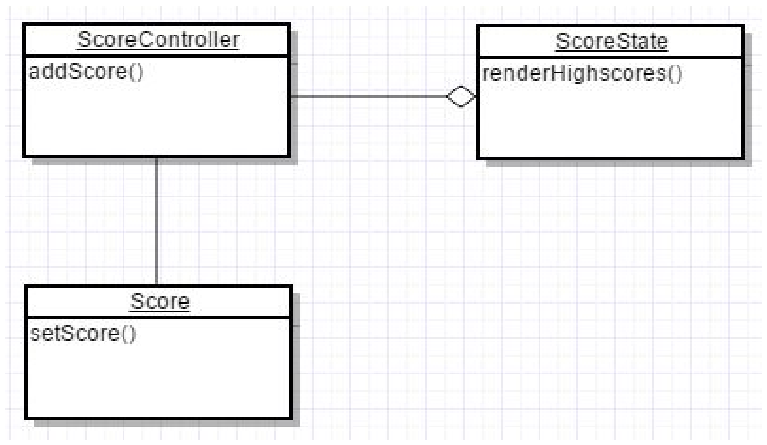


Observer Pattern

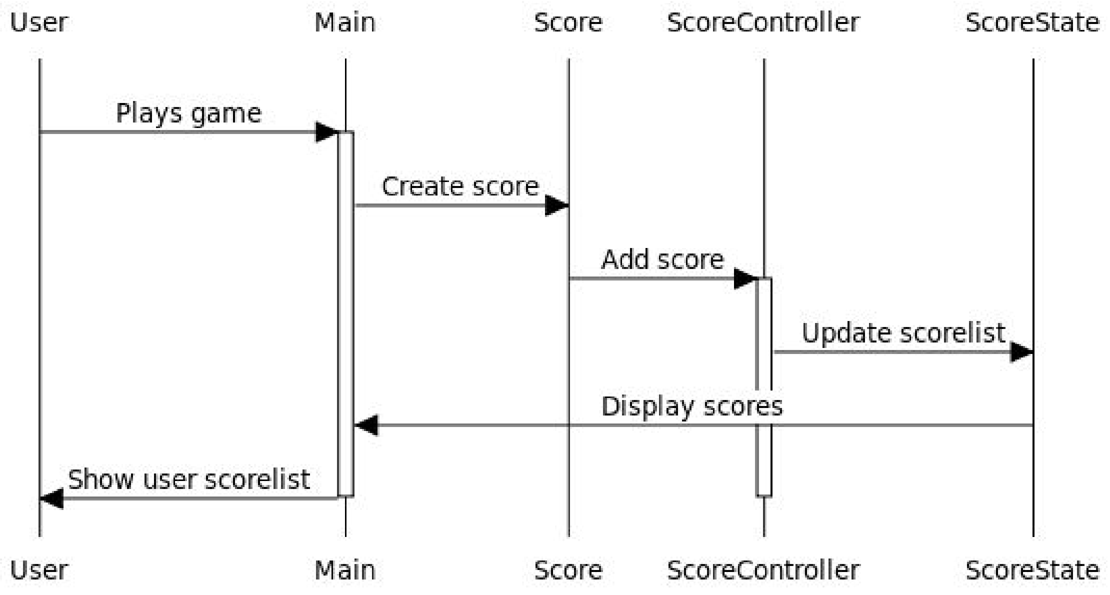
Natural Language Description

The observer design pattern is very useful to send notification when a state of something changes -- for example, the high score. This is where we have implemented the observer design pattern. When a new high score is reached, the subject class is notified and passes this information through to the rest of the program to ensure that the high score list is always up-to-date. We chose this method so that if we were to ever expand functionality with score, such as different levels, it would be easier to implement those new features. The observer design pattern is perfect for this as it defines a one-to-many dependency between objects so that when one object changes state, all of its dependents are notified and updated automatically, even though the 'many' is not really present in our project yet.

Class Diagram



Sequence Diagram



Exercise 3

Reflection

To see our process during the weeks of our project, we took a closer look at the versions of our project we delivered every deadline and run them through InCode. When looking at the first version we handed in, which is the first working version, we see that we had some problems with the Entity Class. The Entity Class was a data class, caused by the fact that the class had a lot of public attributes and a lot of accessor methods (even accessor methods for public variables. There were also some methods in the class that were not even used. When we look at the versions we submitted for the second and third iteration, InCode did not detect any design flaws. The version we submitted for the fourth iteration contained 12 Data Clumps. This was again because of the Entity class, which was extended by all the Player and Opponent classes. The constructor method for an Entity had a long parameter list, containing the position, dimensions, velocity and acceleration of the Entity. InCode suggested that we should extract this group of parameters into a new class. Thus we moved these parameters to a new class called Moveable. There were also no design flaws in the last version we submitted, this because of the exercise where we had to eliminate the design flaws.

The use of InCode to see the problems in code is really nice and we will definitely use the tool again. It is really easy to see what the problems are in your code and the tool even suggests how to fix the problems.

We learned a lot about ourselves as programmers. In the beginning of the project, we were very bad at planning our tasks and we could not really tell how much time a certain task would probably take. Because of this we had some chaotic Friday-evenings where all of a sudden everything had to be merged and there were conflicts, and certain things were not done because other parts of the assignment had taken a lot longer than anticipated. We learned that we had to divide our assignments into smaller assignments and thus could keep track of how far along we were with the assignment. By doing this we knew better how much time each task would take and thus we could have multiple people working on one big task. This resulted into our tasks being done in time and us having enough time to merge everything and not needing to rush before the deadline on Friday evening.

Another thing we learned a lot more about was the use of Git, namely branching and using pull requests. During the OOP Project and the MAS Project we did use git, but in those projects we mostly all worked on the same (master) branch and just committed every change we made. We never used pull requests and rarely made a branch in which we developed some feature of our project. Thus we ended up with a lot of conflicts every time someone would commit a change and we regularly had a non-working master branch. During this course we learned a lot more on how to actually use Git properly and you basically do all the developing in different branches and as soon as you’re happy with the changes you’ve made, you let other people check your work. Because of this, all of us knew about all the code, instead of knowing only about the parts we made ourselves.

Most of us also had never heard about Design Patterns and what they are used for before the lectures. Because of the easy solutions to problems we encountered, we ended up using a few Design Patterns in our own project and really liked the outlines of certain patterns. If we encountered another situation in which the use of a certain design pattern would benefit our code, we would definitely use a design pattern again.

To us, the project was a lot of fun. This because it combined the fun part of making a game with the use of things we would probably need later on, like the design patterns and flaws. We had a lot of fun thinking about elements that could make our game better and the testing of the game to make sure it was functioning properly.