Exercise 1

Refactoring the Observer structure for PlayerInput

We made a custom listener/subject interface structure for the PlayerInput.
This listener structure is written to notify about specific events, rather than one general event.
This removes the need for the PlayerInputHandler, and allows the player to directly implement the InputListener (observer) interface.

Bachelli Bubble

The requirements for the bacchelli bubble can be found in the assignment 5 folder in the git repository.

The bacchelli bubble is a special bubble which can't be split and shoots "bullets" (small bubbles) from it's eye. The bacchelli bubble doesn't bounce like a regular bubble but instead just moves from side to side across the screen. The bubble has a fixed amount of health and regenerates health every 2.5 seconds when it hasn't been shot. The bullets the boss shoots don't bounce from left to right but just up and down.
As said before the bacchelli bubble is a boss, and therefore the ball appears alone in the level, in an empty room. The boss level is the fifth level in the regular game, but in order to make it easier to test, the boss level occurs second in the level order.

Removing the bubble factories

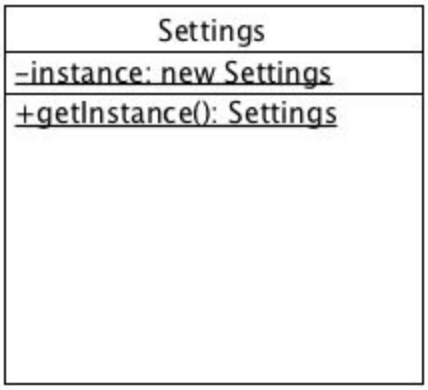
All bubble factories have been removed. They made things look cluttered, because an extra step was required to create bubbles. The bubbles can now be created on its own making things less complicated.

Exercise 2

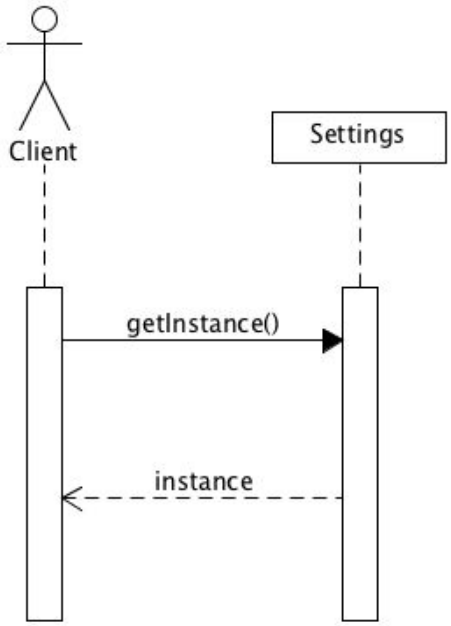
Singleton Pattern

We have used the singleton pattern for our settings class. Before the settings class was never instantiated and all methods which needed the settings were static. Now there is one instance of the settings and all other classes which used the settings now use said instance. This is mainly useful for testing purposes. Because we now have an instance of our settings we can mock it, which means we can test those classes which use it.
The pattern was implemented by making it so all methods were no longer static and passing all classes which need to use them the instance for settings.

Class diagram



Sequence diagram



Decorator Pattern
We implemented the Decorator pattern for the Player Shop Items.
This was beneficial because these items are persistent on Players, so they don't need to be removed like normal powerups.

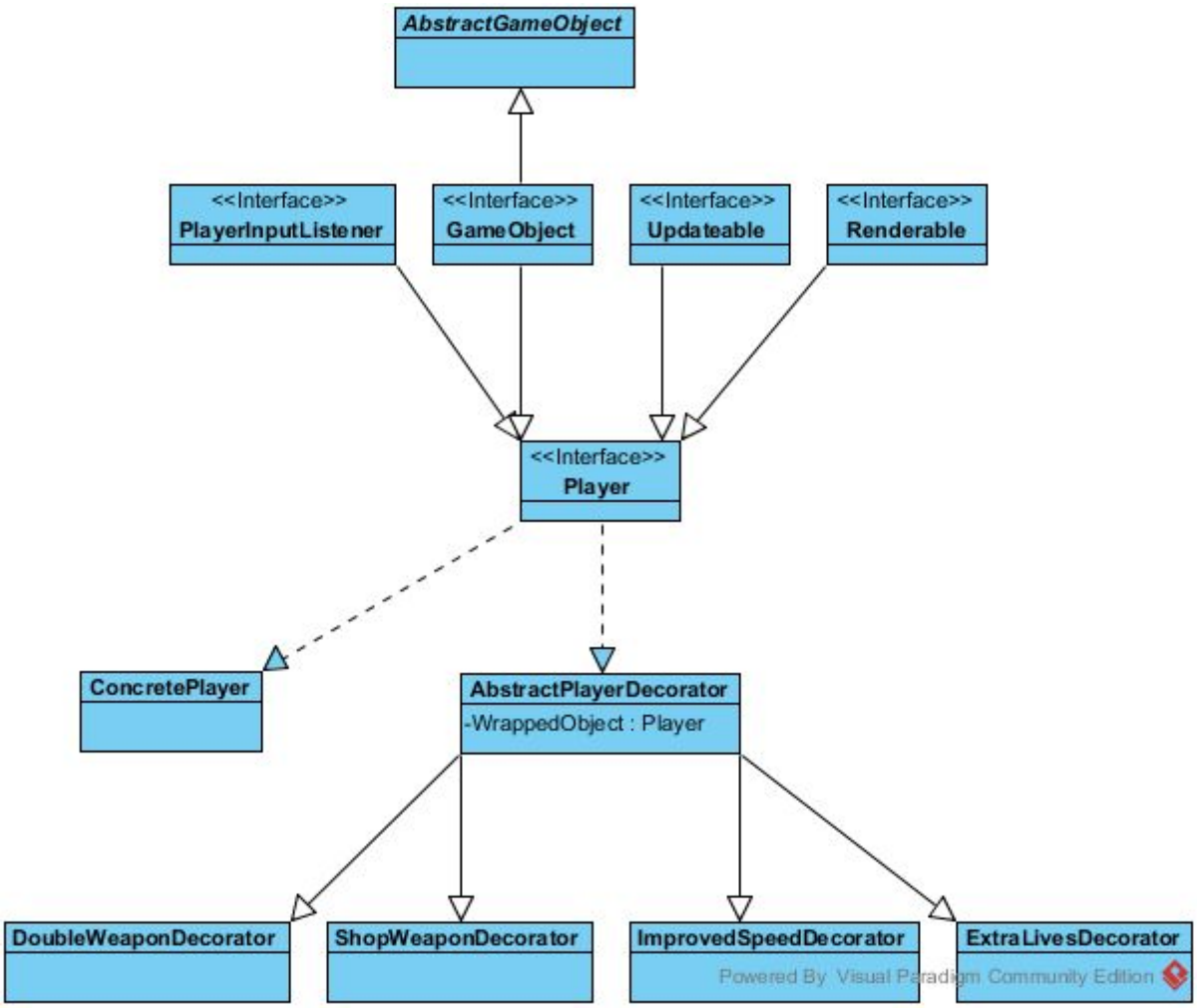
Because of the decorator pattern, instead of keeping many states in Player, we can simply override the functionality of certain methods.
This makes Player cleaner and makes the amount of ShopItems a lot more extendable.

The Pattern is implemented by creating a Player interface, which describes all functionality (combined from all other interfaces) of Player.
A ConcretePlayer is our normal, undecorated Player instance, with normal functionality.

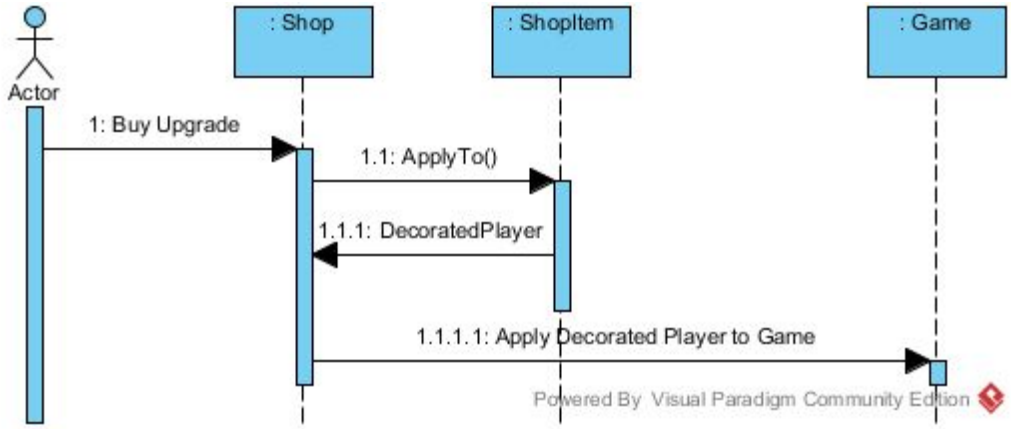
The AbstractPlayerDecorator implements the Player interface and provides default implementations for the (many) player methods, to call its wrappedObject.

The Concrete decorators override the required methods to provide new functionality.

Class Diagram



Sequence Diagram



Exercise 3

Reflection on Project development and team progress
Having done this project in scrum-like work style definitely had a large influence in the way the project progressed. The weekly iteration, because of getting a new assignment each week, made it so the work flow was steady. This also makes development and progress easy to track, which in turn makes assessment and making changes less difficult.

A big boost in efficiency when dealing with scrum is having a capable scrum master. Preferably one who makes clear what needs to be done, but maybe even more importantly, what went wrong in the last iteration. This was always discussed at the beginning of a meeting. One recurring issue on this topic was time management. The deadline and meeting were only 3 days apart, and in some weeks more than others, a lot of work had to be done. Workload had to be distributed accordingly and time had to be managed well. About 2 times there was quite a bit of stress on the Friday evening before the deadline. The next meeting we communicated more about when someone was able to work on the project and when not, so some parts of the project could be done sooner in the iteration. This not only helped with stress relief, but also with making sure all features planned for that iteration were implemented and things like testing were not skimped on. As time went on, we became better in judging how much time certain additions to the project took, and therefore better able to manage time.

Another important part of any project, software related or not, is communication. Besides the weekly meetings, we also skyped occasionally when important decisions were made, or people needed to be brought up to date. Whatsapp and real life interaction were the most important forms of communication though. Only minor communication issues occurred, which were quickly pointed out, resolved and reflected upon to avoid recurrence.

All in all the team cooperated very well. Every week the project gained progress and the team became more accustomed to the workflow and steady heartbeat of the project.

Elaborate Problems encountered and solutions
Maven not building in clean environment

At the start of our project we decided to use Slick2d as library to build our game. The library makes use of some native libraries, which posed some problems for our maven environment. Since maven does not have any real solution to native libraries, our project would not build in a clean environment. We solved this by using a plugin called Maven Natives. This plugin was able to resolve the native libraries and build the project.

Testing slick on travis

With slick2d the class-structure is not really clean, unless explicitly taking care of your structure. This resulted in a set of classes (in slick GameStates) that were not easily tested, without providing a functional display. Slick2d has no option to either mock or stub its OpenGL context. This meant that you really needed to be able to actually run slick2d application on a display, or don’t touch any graphics related code at all. This caused a lot of

trouble for Travis, as is it run in a headless environment. Our solution first was pretty obvious and suggested by the documentation of Travis themselves. Their solution was to use a virtual framebuffer and redirect maven to use that. Oddly enough, it didn't work. We ended up refactoring the classes in such a way that we decoupled logic from anything graphics related. This meant that we could mock any code that touches the OpenGL context ourselves and resulted in cleaner code overall.

Learning points and future adjustments

The main thing we learned during this project was the usefulness of the single responsibility principle and how it can be applied to a project. For the first release we hacked our game together, which meant we had to refactor it in its entirety to make it even somewhat maintainable. The further we progressed in doing so, applying appropriate design patterns when applicable, the more fun the project became. Bugs became easier to find and fix, as everything was contained in their own compact classes. This also meant the code became easier to test. However, unfortunately, it took us a while to come to a point where this was the case. Our code looked relatively horrible and was difficult to work with until one of the last sprints.

The single responsibility principle isn't the only SOLID principle which we have come to appreciate. Both the interface segregation principle and the dependency inversion principle have become staples in our work flow. Without them our code would have been a lot more difficult to work with, and without this course/project we probably wouldn't have come to know them until much later.

The main thing we would do differently would be to start testing right away. If your code is difficult to test it means your code is too complex and should be refactored. Had we done so we would have had an easier time with the project from the beginning, instead of from the end when we were almost done.

We would also follow the SOLID principles more closely than we had now. Some of them we had already partly followed in our initial version, but have been greatly improved in the further iterations. Had we followed them from the beginning we would have been a lot more agile and it would have been a lot easier for us to adapt to our TAs requests.