Exercise 1

Requirements Power Gem

Functional requirements

1.1 Must Haves

1. When a match of 4 jewels is made, a power gem should appear on the board.
2. If the match to create the power gem was vertical, a vertical power gem is made.
3. If the match to create the power gem was horizontal, a horizontal power gem is made.
4. A horizontal and vertical power gem should be distinguishable from normal jewels.
5. The power gem appears on a position contained within the match. (We are still thinking about whether this position is going to be a randomized position of the 4 jewels that created this power gem or not).
6. When a horizontal power gem has a match, the horizontal row will be cleared.
7. When a vertical power gem has a match, the vertical row will be cleared.

1.2 Should Haves

1. Additional points will be added to the score for using a power gem.
2. The vertical and horizontal power gems should be animated.

1.3 Could Haves

1. Create a special T power gem when a T match is made.
2. Create a special L power gem when a L match is made.

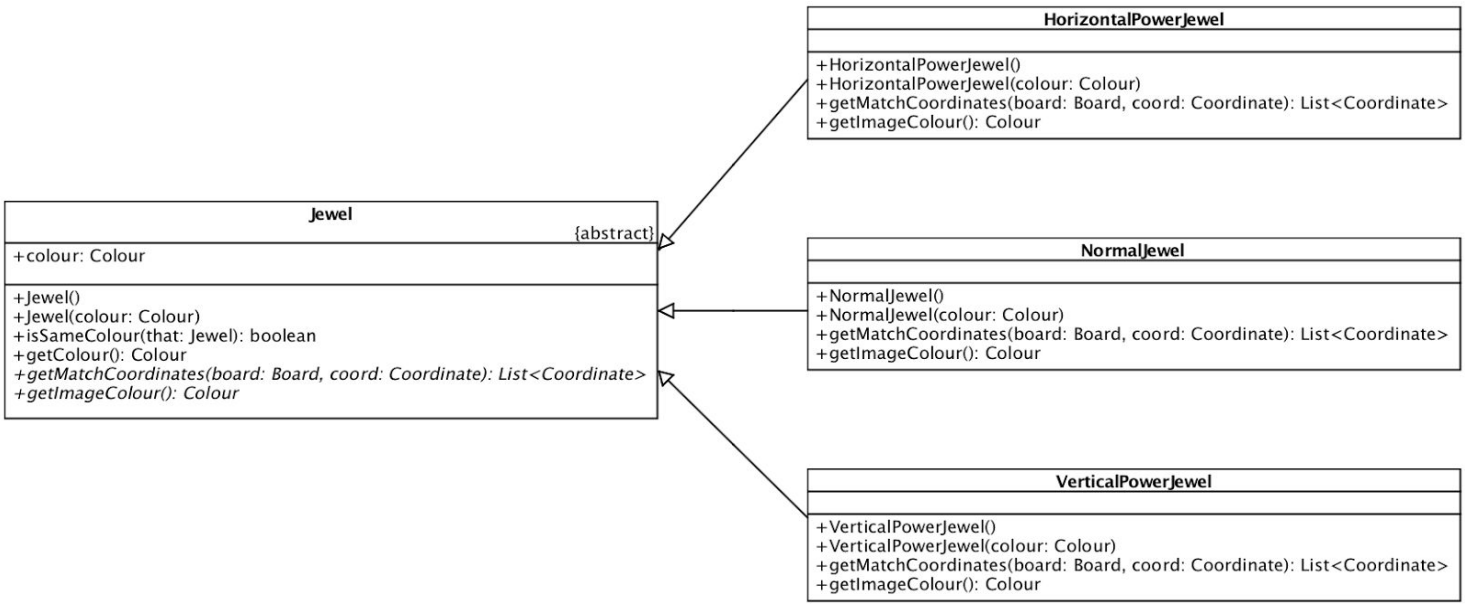
1.4 Won't Haves

1. Special sound when a power gem activates.
2. Special animations when a power gem activates.

Nonfunctional requirements

1. A fully functional version of the power gem function will be delivered on October 27, 2015.
2. The power gem function will be developed using the Scrum software development methodology.
3. The power gem function will be developed using the Responsibility Driven Design technique.
4. The implementation of the power gem shall have at least 75% of meaningful line test coverage (where meaningful means that the tests actually test the functionalities of the powergem and for example do not just execute the methods involved).

Power Gem UML



Exercise 2

Strategy Design Pattern

Natural Language Description

For the power-ups, we want to have special forms of Jewels that have different attributes than regular Jewels. We decided to implement this with the help of the Strategy pattern. The Strategy pattern is used when you have a certain object with certain methods, but you want to have the different implementations of certain methods for different variations of the object. This is exactly what we want with our power-ups: it is a variation of Jewel, but when it's cleared, more Jewels are removed.

For the implementation of this, we need to refactor some of our code first. We will first change the name of Jewel class to NormalJewel, which will be a Jewel without powerups. Then we make an interface called Jewel. This class will basically have one method that needs to be implemented by other classes, which is the getMatchCoordinates(Coordinate c). This method should return all the Coordinates which should be cleared when the Jewel on Coordinate c is in a Match.

For the NormalJewel, we just return Coordinate c, which is the Coordinate on which the Jewel is located. Whenever a NormalJewel is cleared, nothing special happens and just the Jewel itself is cleared.

For the implementation of the power-ups, we will add two different Power-ups: HorizontalPowerJewel and VerticalPowerJewel. These will both implement the Jewel interface. Then the getMatchCoordinates(Coordinate c) of HorizontalPowerJewel will return all the Coordinates on the same row and the VerticalPowerJewel returns all Coordinates in the same column.

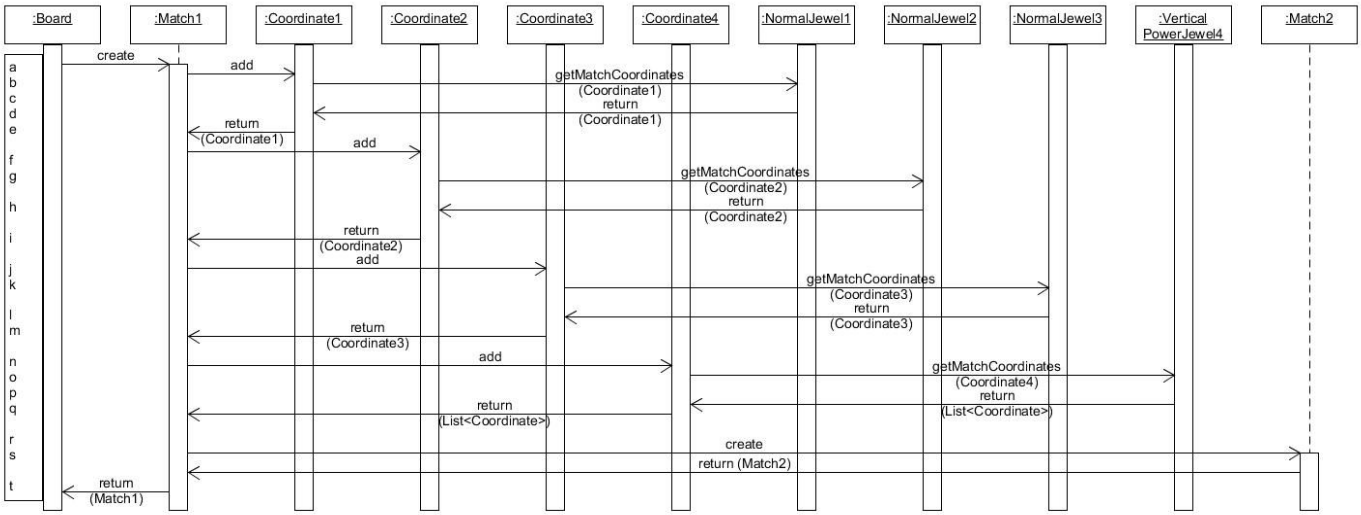
Finally, at the moment a Match is cleared, it will check for the size. If it's 4 and it's vertical, a VerticalPowerJewel is created and if it's horizontal, a HorizontalPowerJewel is created.

Strategy Design Pattern Class Diagram
(Check the previous page for the UML for our Power Gem. It is a class diagram of the Strategy Design Patter).

Strategy Design Pattern Sequence Diagram

Some information beforehand:

- This sequence starts during the checkMatches method, when it finds a 4-Match with a VerticalPowerGem in it.
- For each of the Jewel X in the diagram, it is located at Coordinate X.



Explanation with some of the steps:

b-e: This is what happens when you add a Coordinate with a NormalJewel in it. It returns a single Coordinate (in a List) It adds just that Coordinate.
f-i: Same as (b-e)
j-m: Same as (b-e)
n-o: This is the Coordinate with a PowerJewel in it. The same happens here as with a NormalJewel, you execute the getMatchCoordinate(Coordinate 4) method.
p: However, the method will not return a single Coordinate, but 8 Coordinates. That's because the getMatchCoordinate method is implemented differently for NormalJewel and VerticalPowerJewel. This difference shows the Strategy pattern in the Jewel.
q: Returning of the list with 8 Coordinates.
r-s: Since the size of the list is bigger than 8, a new Match is created, in which these 8 are added. Then this second Match is added as a MatchComponent to the first Match.

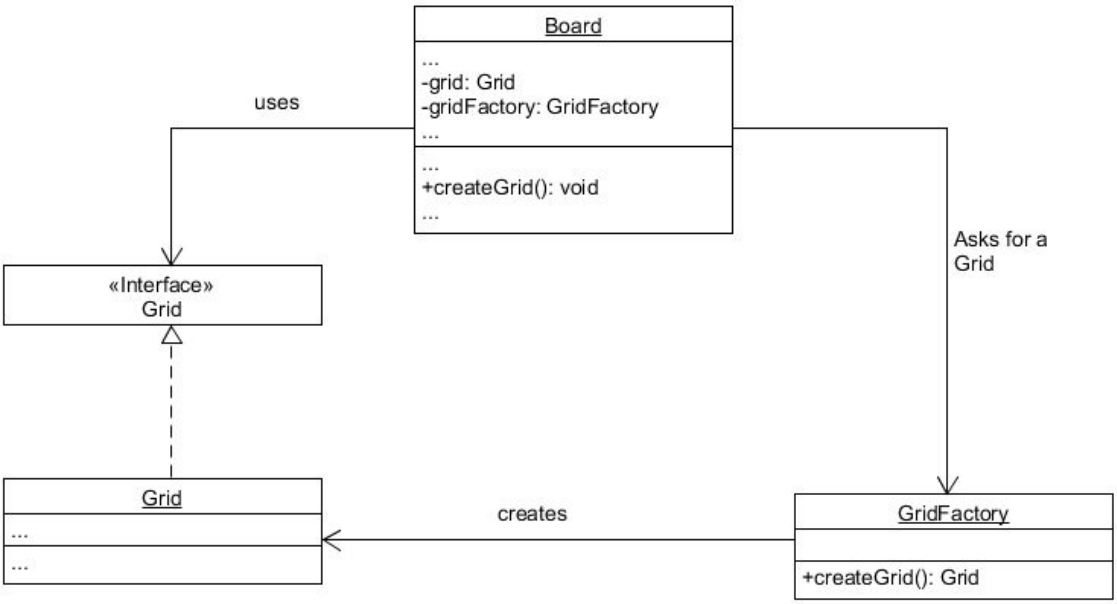
Explanation for lack of another applied Design Pattern

For this assignment, it was asked that we implement two more design patterns. But other than the strategy pattern that we implemented, there did not appear to be any beneficial way to implement any of the other design patterns.
This is because the strategy, observer, and composite, design patterns were already implemented;
The factory and abstract factory method are not useful to us because we do not create anything other than the same grid every single time;
The singleton pattern is too simple to justify using it as our assignment;
The adapter pattern is not useful to us because we are not using anything that would need to be adapted to out code (I suppose we could steal the code from another group and apply the adapter pattern to that, but somehow I think that would be frowned upon.);
The iterator pattern is also not useful to us because we do not use anything that mixes data structures in a way that would need an iterator;
The state pattern is not very useful for us because we only have one state for the game, and that is playing (It would be possible to apply it to our project, but there would be no reason too, and just like with the singleton pattern, it would be mind blowingly easy);
The command design pattern is also not very useful for us because, in our project, it would only add unneeded complexity to our project (We also already have the observer pattern, and while there are instances where they can work together, in the scope of our project, they would conflict a little bit.);
Lastly, the model view controller has the same problem as the command patter, it would be possible to implement it, but it would be hard to do and would only serve to clutter the project with unneeded trash.

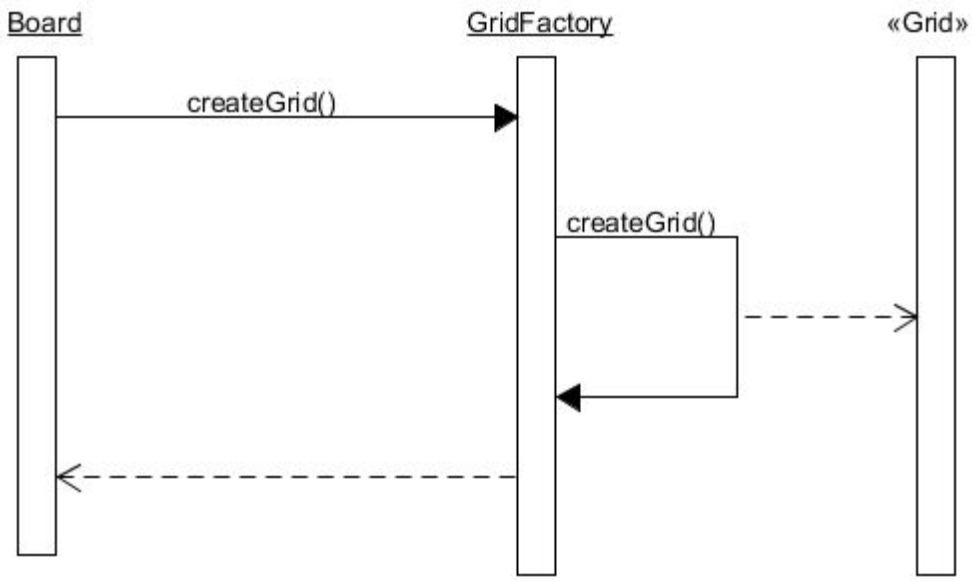
Overall, the majority of the design patterns was not rejected because they were impossible to implement, but because they would not fit into our project. We can see why these design patterns would be useful and even needed in other projects that were perhaps bigger and harder to manage than ours, but for our fairly small and simple project, they are not needed. After looking at them all more and trying to find which one would be the best fit for our project, we decided that the factory design pattern would be the best fit for our project, but it would have to come with a new implementation where the grids from new games and levels would be randomly generated.
The reason why the factory design pattern would be the best design pattern to add to our project if we were going to add it, is because it would be needed. If we were to implement a map creator that creates a map whenever it is needed, it would be a pretty intensive class because it would have to make sure that the map doesn't start with any matches, starts with possible matches, and isn't impossible for the player to complete.

To best do this, it would be best to implement the factory pattern and create something like a mapFactory that would handle the creation of the maps and possibly a jewelFactory to handle the placement of the jewels if it were not handled already by the mapFactory.

Factory Design Pattern Class Diagram



Factory Design Pattern Sequence Diagram



Exercise 3 - Reflection

Seven weeks ago, the five of us first convened to start our work on the Software Engineering Methods practical assignments. Though we were acquainted, we had no working experience together, thus we had no real idea of everyone's strengths and weaknesses. We simply starting working on our game and discussed problems we stumbled upon with the others, but we didn't follow any concrete plan (such as a sprint plan). Everyone made bits and pieces of the game and by using Git, we merged the code accordingly, resulting in our game's first working version.

As the weeks progressed, we got into a certain rhythm of weekly deliveries. Each week we became more accustomed to applying responsibility driven design, to estimate the workload for each aspect of a delivery, to program code while keeping our teaching assistant's feedback in mind, to use the information from programming tools (like Travis-CI, Maven, checkstyle etc.) to our advantage and generally to work together. We learned to apply a few key activities, which we will discuss now.

Every week on tuesday we would make a requirements document. We would do this together with the whole group, since it is easier to come up with features when working with 5 different viewpoints. Making this document helped us break down the problem into smaller components and get a better overview of the problem at hand. We learned that this is better than to just start working on one big problem individually and see where it leads.

Afterwards we would make a sprint plan, in which we divided the subproblems derived from the requirements document amongst the team members. We used Planning Poker to aid us a few times, but we did not consider this an indispensable activity. We learned that time spent on Planning Poker could sometimes be better spent on discussing ideas for and problems with the next assignment. We would decide along the way if some problems required the aid of an additional team member. When a certain activity was finished, we decided that the responsible team member should mark this activity with a green colour on the sprint plan spreadsheet on Google Docs, so that the whole team was aware of the actual status of the assignment.

After these preparatory activities, if our assignment included the implementation of a new feature, we would work on the design of said feature. To better understand the architecture of our code we would make UML documents, both before and after implementation. We learned that UML can help explain ideas for designs to other team members, spot possible weaknesses in our code, such as duplicate responsibilities for certain classes, and spot opportunities for improvement, such as that the application of a certain design pattern that could improve maintainability and efficiency.

What we have learned is reflected in our code. It is now more logically structured and understandable compared to when we released our first working version. This is partly due to the above mentioned preparatory activities and feedback from our teaching assistant.

Of the design patterns, we can honestly say that the Observer pattern proved to be most helpful in the end. The Strategy design pattern gave us some advantages with clearing jewels, but disadvantages with creating new jewels. The Composite design pattern did not help a lot in the end.

Don’t mistake this for us thinking that design patterns are useless, however. We realize that when one or more design patterns are applied (it depends which pattern is be applied in which situation, if at all), they can make the code more understandable and more efficient. We just wanted to gain experience by practicing with the application of design patterns for future projects, regardless of whether it would really improve the current project.

Some tools that helped us during the development process were Maven, Travis and Git. Maven and Travis would consistently test our software. By regularly reading the reports generated by these tools and correcting the errors and warnings stated there, we were likely able to prevent future bugs. Moreover, seeing our builds pass created a sense of confidence in our code. As for Git, it was yet again an indispensable tool for group programming, because it allowed for easy code sharing between team members, the use of branches for working on different implementations and checking code differences between commits. Furthermore, we learned to make better use of pull requests. Before, we discussed everything code-related during meetings or on whatsapp, but pull requests allowed us to discuss new features separately, making the discussion progress more organized.

We learned as a team that the application of the above mentioned activities and tools, though to a certain degree depending on the situation, would have a positive effect on future projects. If we were to start another project, we would apply what we have learned from the start. Before recklessly starting to write code, we would first think hard about the structure (make UML diagrams or use cards on which to write classes, sub- and superclasses and collaborators if necessary), because we noticed that a lot of refactoring was needed to implement new features into our old code. We believe this can be prevented by making a strong basis from the start.

We also learned to better communicate with each other. At the start of this project, some members were reluctant to express their concerns about certain code designs and implementations. But by getting to know each other over the many meetings, we learned to both give and receive criticism better. Some of us also taught the rest that it is good to be critical of your superiors. Even though it is a virtue to listen closely to the feedback of the teaching assistants, It is also important to give them feedback yourself, if necessary.

All in all, the practical assignments were a valuable learning experience. At the start of this project, we didn’t really know where to start and making a game in under 2 weeks felt like we were being thrown in at the deep end. Now we are more knowledgeable about all the activities software engineering encompasses and more confident about starting up projects, designing products and being able to deliver products on time.