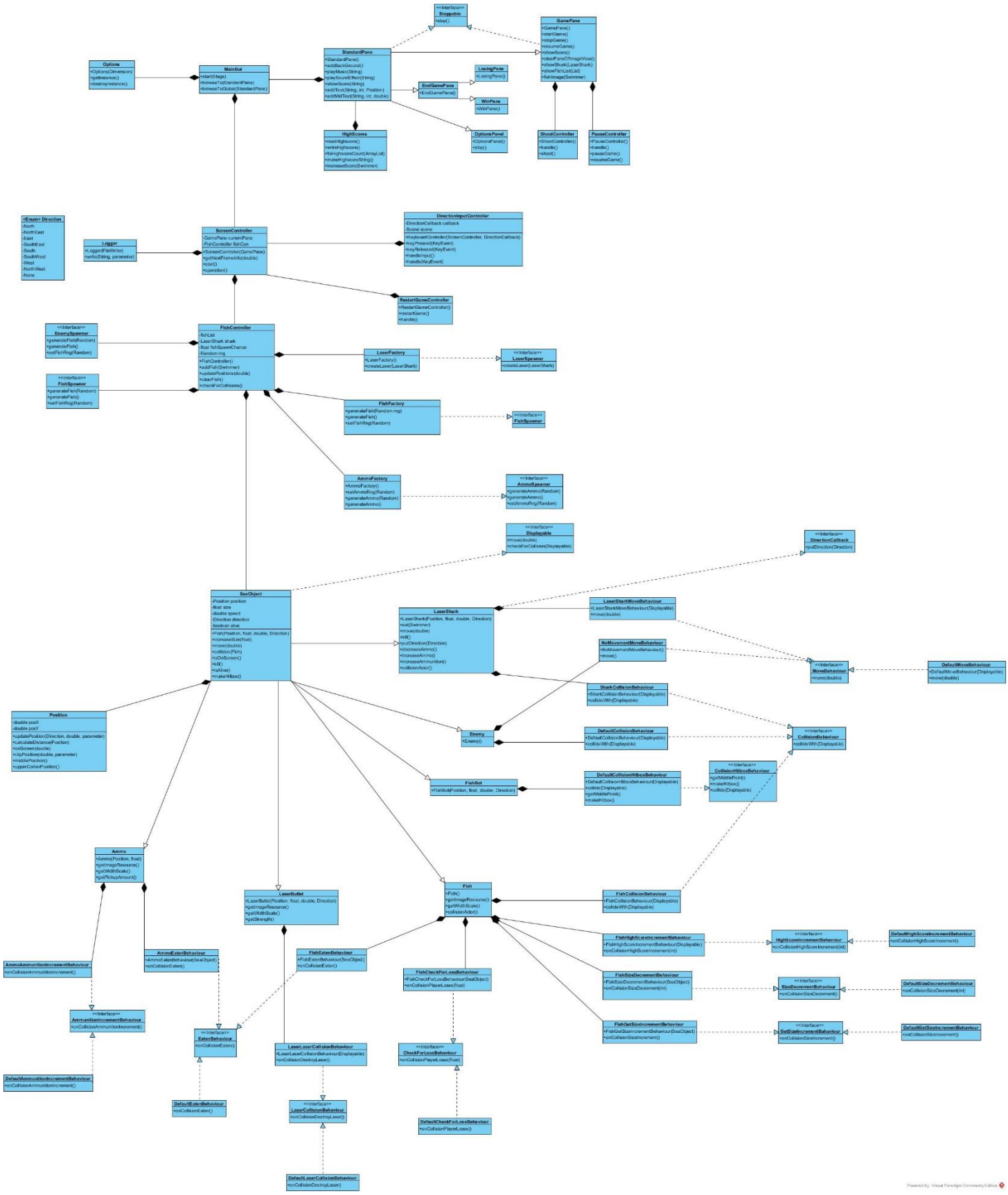
Assignment 5 Group 27

Exercise 1

The updated UML:



One of the main problems that we faced during the span of this project was testing the GUI. At the start we thought testing the GUI was not necessary as we thought testing a GUI is not common practice and will take a lot of time. This meant that our test coverage after the initial version of the game was not very high. With the GUI we got about 60-70% coverage. If we omitted the GUI from that coverage it would go up to about 96%.

After discussing if we should test the GUI as well with both the TA and the group we finally decided to also test the GUI. We set the implementation of the GUI tests as one of the sprint goals for the next week and they were achieved. The GUI was now tested, but there were still problems. The GUI tests lacked consistency. The tests would sometimes randomly fail for no apparent reason and they would also fail on laptops with a lower resolution. To fix these inconsistencies, we made discussed them and made the improvement of the GUI tests a new goal for the next week.

Removing the inconsistencies from the tests was a tedious and time consuming job. The randomness was completely removed from the tests and we thought that the tests worked as they all passed on our machines. We thought we finally had our GUI successfully tested. This was until our TA showed us that the tests did not run on her machine. It seemed that the problem with the resolution had still not be fixed. This is where we decided to completely omit the GUI tests. The tests had taken a lot of our time and to improve them even further to remove all inconsistencies would take even more time. The only thing the tests really offered was an improved test coverage of the code overall, but not much more because we can test the GUI manually to some extend by just running the game.

In conclusion, the GUI tests proved to be too time consuming to make implementing them worthwhile. This why in the end we decided to not make any GUI tests for our project.

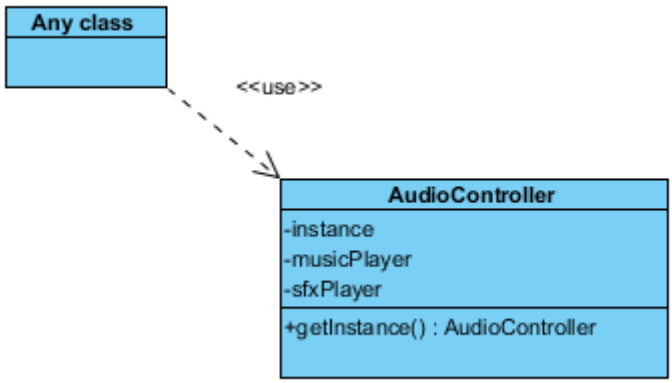
Exercise 2 - Design patterns

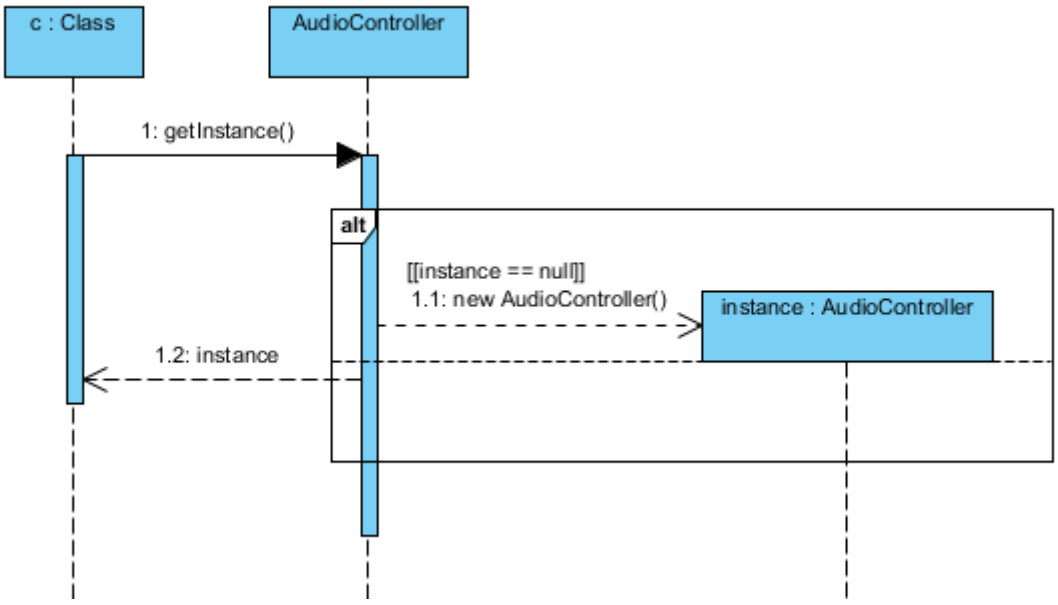
Singleton pattern:

1. Our project has several actions which use a sound to make clear what happened. These were all handled by the class that called them. Therefore we decided it would be a good idea to implement a single location that could handle all audio.

   Since this would be some sort of audio controller we thought that it would benefit from a singleton design pattern, because that way all classes had access to the same media players. Having access to these media players helps because it means we can modify it from anywhere within our code without having to go through another class.

   This design pattern required some additional actions to be called whenever we might need to change to a different audio controller since we don't want the music from that controller to keep playing in the background.

2. 

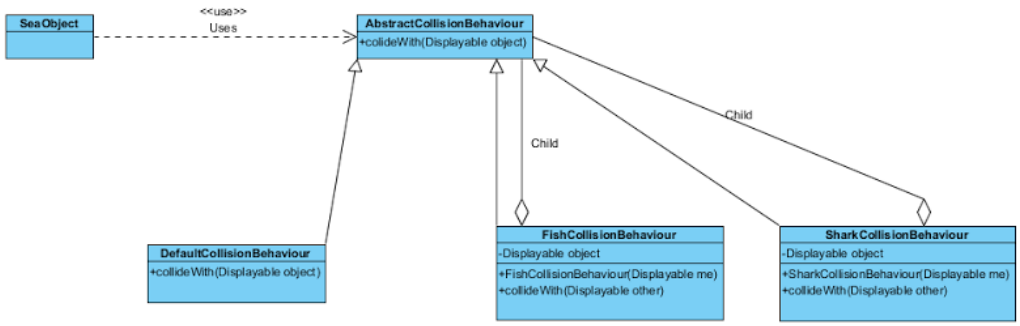
3. 

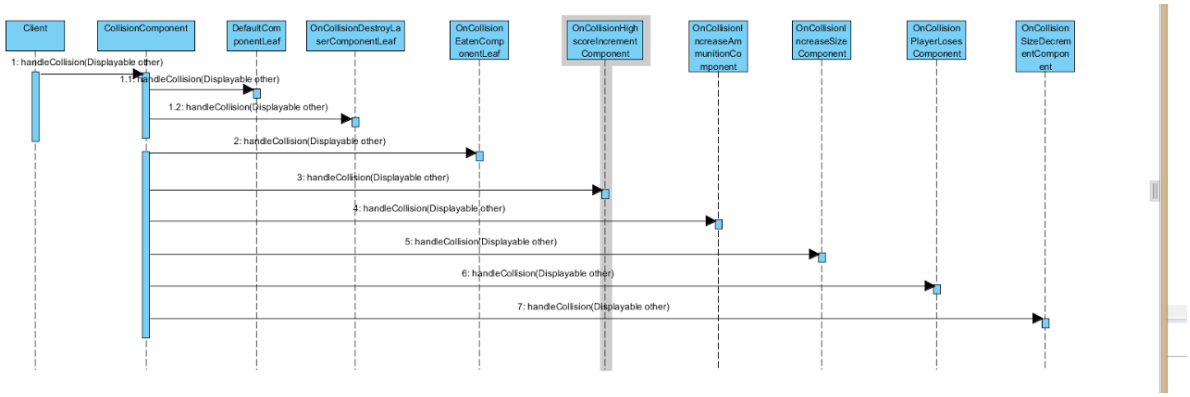
Composite pattern:

1. Our project contains a lot of collision behavior classes. This is because there are multiple types of SeaObjects that are able to collide with each other. Each of these collisions have to be handled in a different manner and thus need a different effect of the collision. This is where the multiple collision behavior classes come from.

   Before this exercise the collision behavior classes did not show any sign of cohesion with each other. They were all just different classes. This is why a composite pattern was needed. The composite design pattern now lets us selectively treat this group of objects that are part of a hierarchy as "the same" when they are in fact different.

   We have implemented the composite design pattern in a very simple way. Because all of the collision behavior classes contained the same type of method, the method that handles the result of the collision. This is the method handleCollision that is in the leafs. All the leafs on their part implement the CollisionComponent interface.

2. 

3. 

Exercise 3 - Wrap up – Reflection

Our main goal of the first version was making a working version of the game with all the must haves of the requirements. We succeeded in making a first working version with all the must haves of the requirements, but we were not completely satisfied with the implementation. Due to the fact that we only had two weeks to make a working version, we split the work in separate parts without a good communication about how and who had to implement what. This resulted in some messy parts in the code and some duplicated work.

After the discussion about the messy parts and duplicated work, we tried to make clear agreements about classes, responsibilities and collaborations following the Responsibility Driven Design. Besides the exercises in each iteration, we assigned a feature or improvement in the code to every team member to make the game better playable or the code more structured after each iteration.

In the first iteration we were more focussed on the already mentioned agreement about classes, responsibilities and collaborations. Making use of tools like CRC cards and UML diagrams, it was easier to get a clear view of how the structure of the implementation looks like. In this iteration, the code was improved with some features like restarting the game, adjusting the screen resolution, a score system was implemented and a logger was added to the code.

In the second iteration we did a lot of restyle of the code. The GUI class was divided into separate classes which resulted in a clearly cleaner code of the GUI, the highScore class was restyled and we removed the unnecessary classes. Knowing that we had to implement design patterns in the next iterations, our main focus was not already on design patterns, because that would give us duplicated work in the next iteration. In this iteration we also added some new features like sound effects, different fishes and we made more tests to get a better coverage.

The third iteration was mainly focused on design patterns, we implemented two design patterns to get a better structure in the code. The first design pattern to improve the code was the factory design pattern for creating fishes. The second design pattern was the strategy design pattern for our fishes. In this iteration we did not add a lot of features, but we did a lot of testing, restyling and fixing bugs.

The fourth iteration was about software metrics and adding new features. The inCode tool uses software metrics to detect a number of design flaws, the number of design flaws that was detected in our code was 2. There was a duplication between the LosingPane and WinningPane class and there was an internal duplication in the DirectionInputController class. After fixing these two design flaws, we also fixed a flaw that was not shown in the inCode tool. We had implemented 12 enemy classes. Using a factory design pattern we reduced these 12 classes to one general fish class. The feature we added in this iteration was making the shark able to shoot a laser. When the laser collide with a fish, the fish must shrink. We also made the shark able to gather ammo that appears on the screen.

In the fifth iteration we did a lot of restyling and added two new design patterns. The restyle was especially about the collision behaviour. The two new design patterns we added were the singleton pattern and the composite pattern. We also added a new feature, which was a pause screen. With this feature, the player is able to choose certain settings in the pause screen. In this fifth iteration we decided to remove the already implemented GUI tests, because they were giving errors on some devices. We did not know the reasons for these errors and we did not have the right devices to test if

the GUI tests were working right on different kind of devices. For these reasons, we decided to not fix the error is in the GUI test.

To reflect on our progress throughout the iterations, we made a lot of improvements if we compare the version we have now with the first version. In the first version we did not use design patterns, while in the final version, we have used design patterns almost all over the code. This results in a better structure and a cleaner code. We also added more features to the game then we thought of in the beginning of the project. This results in a better playable and more challenging game.

To reflect on our team of programmers, there is still some improvement possible. We sometimes forget to assign a specific task to a team member, like handing in the requirements, which is an important task. There were also some misunderstandings between each other in the way of implementing some features. We think the reason for this is lack of communication or differences in experience in programming between the team members.

What we have learned and what we will use in the future to design and implement software systems, is implementing in a clean and structured way from the start of the project. We spent a lot of time in restyling of the code and fixing messy code, this time could be better spent for example on new features, to make the game better playable. If we start from the beginning of a project with already implementing in a clean way using design patterns and following the Responsibility Driven Design, it would reduce a lot of time spent on restyling.