Architecture

For this Assessment we built the architecture diagram using the IntelliJ built in diagram functionality. Whilst we planned the new architecture on paper it was far easier generate a diagram from the code than try to reorganise our current diagram, which is very large, on lucidchart. Our Final architecture is available at https://teamfractal.github.io/assessment4/diagram.png.

For assessment 4 we introduced a number of changes to the architecture, including adding a number of classes.

We created the `PlayerSelectScreen` class and the PlayerSelectActors class, this allowed us to build a screen on which a player can choose the number of players they want to play with. We separate the screen from the actors to allow us to keep the logic of the buttons in one class and then focus on where they sit on the screen in another.

The `PlayerSelectScreen` and actors class communicate to allow us to place buttons and labels on the screen. The screen is initialised when the player presses the new game button on the main menu screen and is stored in the `RoboticonQuest` engine class, along with the other screens. When the player confirms their player selection the game engine will be called to enter the main game.

To implement the *Capture the Chancellor* mode, we took advantage of the `gson` module to transform JSON configurations into our predefined class, `CaptureData`. The advantage of separating the code and the data is to enable a flexible way, to modify the game's behaviours based on the pre-defined data input, any maintainer can adjust values accordingly without the need to figure out where the data is embedded inside the large code base.

We have 3 animation classes

- TypeAnimation - to display text
- WhiteStrip - to show the white stripes on screen
- WildChancellorAppear - use to show the chancellor and show him being captured

TypeAnimation and WhiteStrip both communicate with the WildChancellorAppear to send the details of the part of the animation that they deal with to the animation where it will be displayed. All three classes extend the AbstarctAnimation Class and implement the IAnimation class allowing us to easily build an animation with prebuilt classes.

During gameplay the engine handles the capture the chancellor game. After installation of roboticons the engine will generate a new instance of WildChancellorAppear which handles the rest of the minigame by generating the other animation classes and data classes. This means the engine is not too involved in the minigame and allows us to separate it from the rest of the game.

We also changed how the screens were generated. When we inherited the code the main game screen was created by the main menu screen, however by inserting the new screen to choose the number of players this no longer worked. We instead moved the screens to the engine as attributes. This is a far better system because it places all of the screens in one place and allows any class that knows about the engine to manipulate the screens.

Traceability

To ensure traceability, we focused on bringing forward the ideas about the game during the initial formation of requirements to the whole project, all the way through to the implementation.

The main way in which we did this was by presenting our requirements as a series of testable features. This meant that when we had implemented the game we could easily go back to our initial requirement documentation and go through each one of our requirements tests. If any of these tests did not pass we would know what we still had to implement.

We also used the git version control system, this allows us to easily to locate the history of our product. From this history we can get an idea of how the game progressed, we can see the order in which classes were made and how the relationships between classes has developed and evolved. This has been useful as it has allowed us to understand the relationships between classes, and made the processes of modifying the architecture a lot easier.