Architecture

Group 18

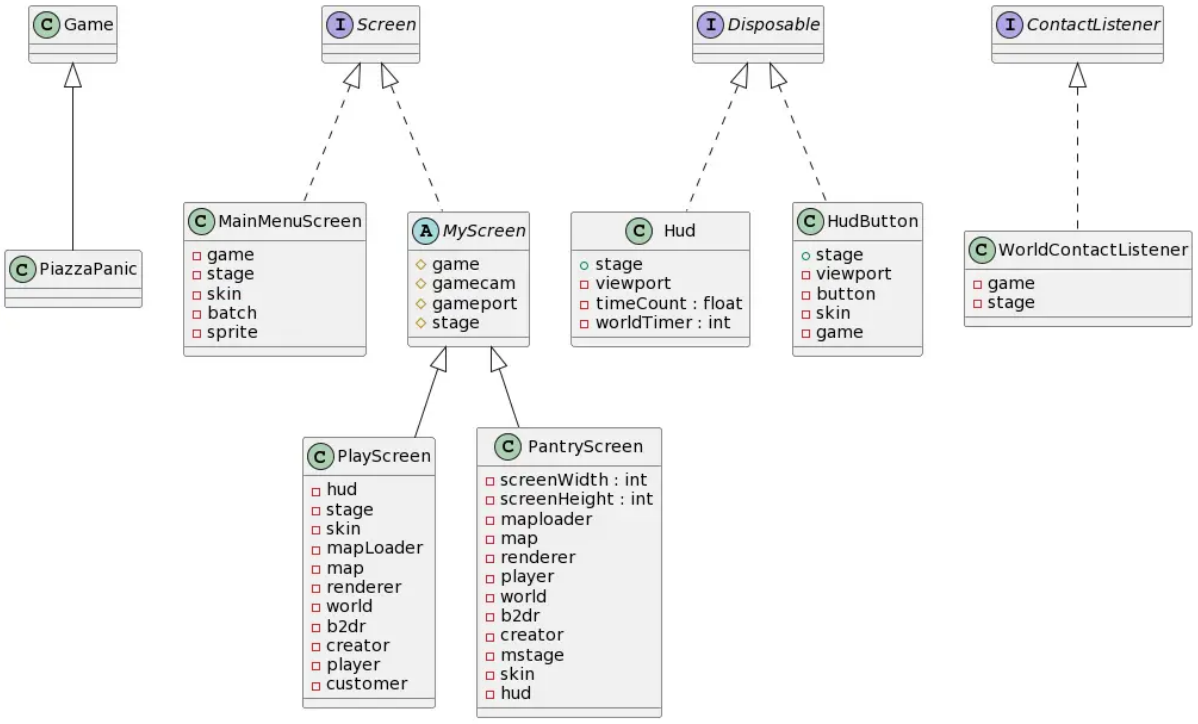
Team B

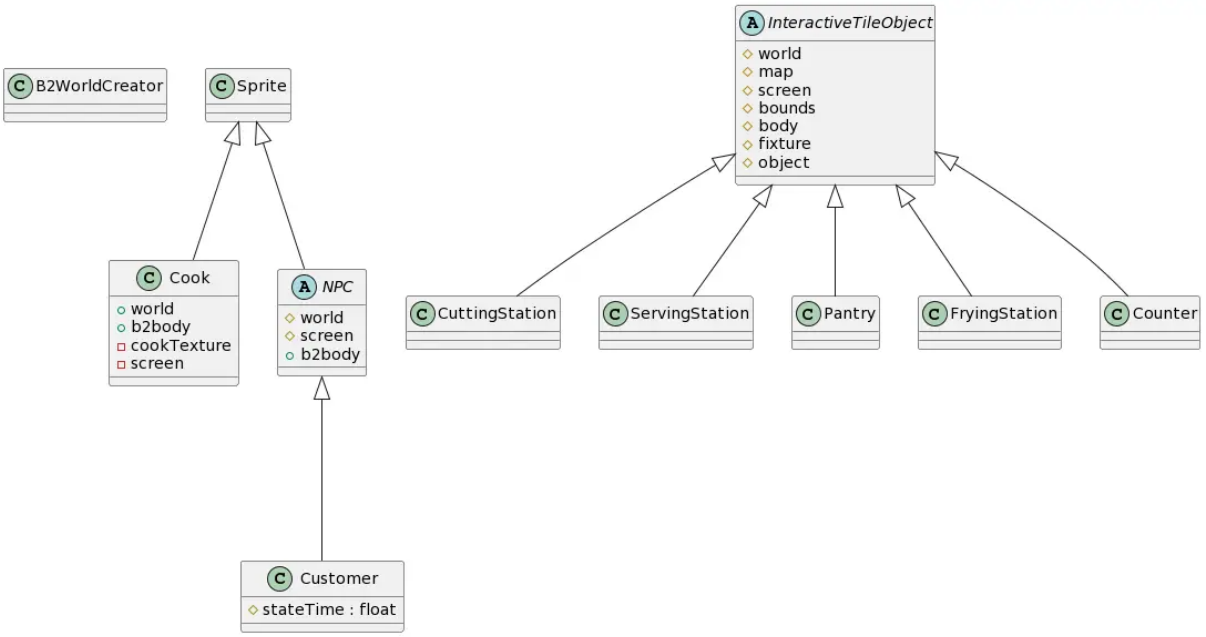
Olivia Betts
Zac Bhumgara
Nursyarmila Ahmad Shukri
Cameron Duncan-Johal
Muaz Waqas
Oliver Northwood
Teddy Seddon

Diagrams

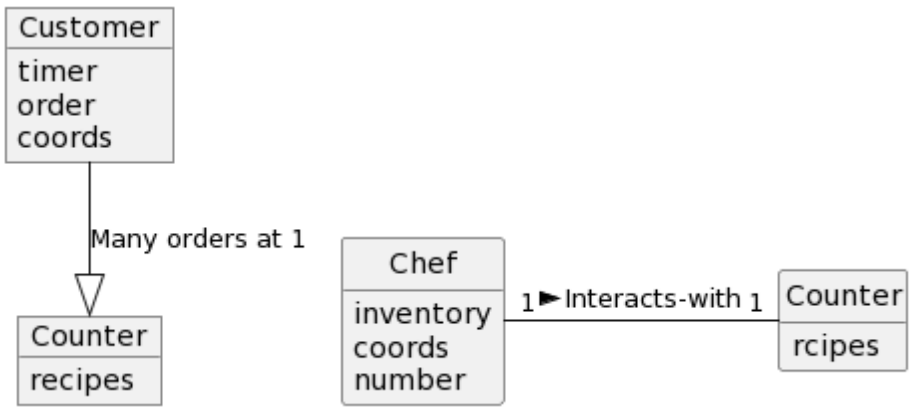
We have used Unified Modelling Language (UML) to model the structure and behaviour of our software system. UML diagrams can be used to illustrate our project before it begins or as project documentation once it has begun. To create the diagrams, we have used an UML extension on Google which is PlantUML Gizmo. Diagrams can be generated from plain text using the open-source tool PlantUML. Some of the diagrams have been split, to make them more readable. The full UML is available on our website.

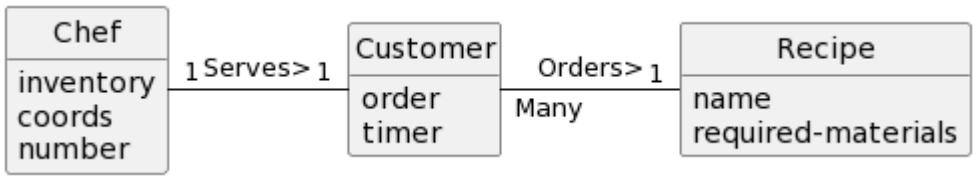
Structural diagrams:





Behavioural diagrams:

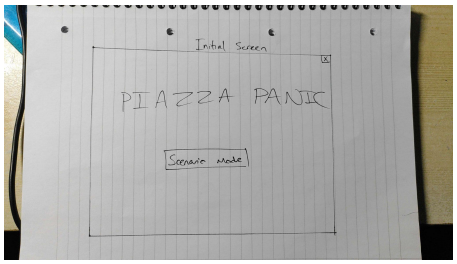


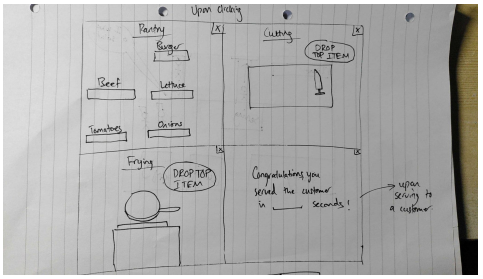


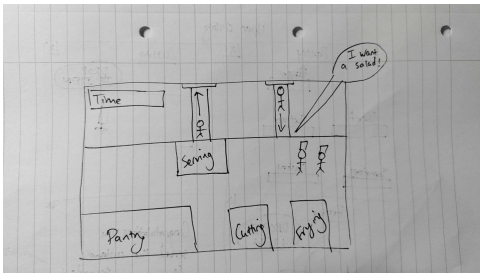
Development Over Time

The first thing to consider in our architecture planning phase was the overall layout of the game. We knew that we needed to make a game which has different entities in different classes, and makes use of object-oriented techniques, such as inheritance. For this reason, we created different classes such as a ‘cook’ class, a ‘customer’ class and a ‘cooking station’ class. The ‘cooking station’ class made use of inheritance and three subclasses (‘cutting station’, ‘frying station’ and ‘serving station’) were the child classes, which all had the same methods as the parent ‘cooking station’ class. We created some initial diagrams to help illustrate the layout of the classes and how they would link together; and some CRC cards were also made to show the different kinds of classes which we had, what their respective responsibilities were and how these responsibilities linked between classes. At this point our architecture was very basic and not fully linked to our requirements. The CRC cards can be found on our website.

Initial diagrams:







Once the user requirements were complete, then we began to link our architecture more specifically to the user requirements. In particular, we made use of the functional system requirements, which would explain how the game itself works.

From the start, we had a top-down approach to the design of the architecture. We started by breaking down the functionality of the product into small classes, such as recipe, ingredients, and pantry. However, as we progressed with the development of the product, we realised that this approach led to a lot of duplicated code and increased complexity. As the problem had been over-simplified, we decided to scale back and refactor the architecture. For the example mentioned before, all the oversimplified classes were made into a single class called pantry which inherited information from all the other small classes. This approach simplified the architecture and made it more maintainable.

Alongside the implementation of the project, we started developing a final representation of the software architecture. This would be composed of a UML structural diagram, representing the different entities and their relationships in the game. In this document we have justified how the different entities within the diagram relate to the requirements.

To do this, we used the UML Maker tool provided by IntelliJ Idea. This was useful as it allowed us to automatically create the proper layout for the UML, complete with all the entities. After some editing to the diagram, the diagram created a link between the requirements and the implementation.

During the implementation of the project, we created some new classes to make coding the game easier. For example, we created a MyScreen abstract class which extends into the separate PantryScreen and the StationScreens. This was useful because it meant that the child classes inherited all of those characteristics from MyScreen. Along with that, we created a new HudButton class to allow functionality on the separate screens.

Towards the end of the implementation, we added these into our UML diagrams.

Linking Architecture and Requirements in full:

MainMenuScreen: this is where **FR_GAMEMODES** is fulfilled. The main menu screen allows the user to switch between two game modes, scenario and endless mode (although the endless mode button is absent as that was not a requirement for Assessment 1). This class was not originally in our designs, so we added it to the final architectural model.

PlayScreen: this is where **FR_CONTROLS** is fulfilled. The functionality allows the cook to move around the kitchen using the arrow keys. **FR_CUSTOMERS** is also fulfilled as the customers are also created and rendered in this class.

MyScreen: this is an abstract class which is the parent class of the following classes - ***PantryScreen, CuttingScreen, FryingScreen*** and ***ServingScreen***. It fulfils **FR_PREPARE** as it takes the player to a separate screen, where they can then take the steps to prepare the dish.

WorldContactListener: we needed a method to actually be able to detect when the cook collides with the Interactive Objects. This is done with this class. This helps to fulfil the **FR_RECIPES** and **FR_PREPARE** as it allows the cook to go into those separate screens to try to create the dish.

InteractiveTileObject: Fulfils the same requirements as above. Rendering objects on the map as InteractiveTileObjects allowed the WorldContactListener to detect collisions.

Pantry, CuttingStation, FryingStation, ServingStation: these are all child classes of ***InteractiveTileObject***. Again they fulfil the same requirements.

Buns, Lettuce, Onion, Patty, Tomato, Knife, Pan, Plate: these are also child classes of ***InteractiveTileObject.*** Again they fulfil **FR_PREPARE** only, as they allow the actual ingredient to be chopped, fried or served.

Hud, HudButton: these two classes allow for navigation between the screens and a timer and reputation points. They fulfil **FR_REPUTATION_POINTS.** We also realised that the time taken to complete the scenario should be displayed. For this reason we added a timer method to the customer class.

Cook: this class is meant to fulfil **FR_MULTI_COOKS** and **FR_COLLISION**. We were unable to implement these in the actual game. It also fulfils **FR_ITEM_INTERACTION** which we implemented, but did not include the upper limit.

Customer: this class helps to implement **FR_CUSTOMERS** and it is a child class of ***NPC***.

B2WorldCreator: this class creates all the objects on our game map. Therefore it is crucial for the entire running of the game. It helps to fulfil **FR_RECIPES** and **FR_PREPARE**.

Bearing all these requirements in mind, we made noticeable changes to the architecture of our game. The most evident of these was the addition of new classes to represent the different screens which we will have. We decided we needed a main menu screen, which will have a button to start the scenario mode. After starting the game, the game screen would be loaded in, which is a 2d view of a kitchen, containing cooks, a counter, a pantry, and the cooking stations.

There are some requirements which we have not implemented as of this moment. These are all requirements which we plan to implement in our next project, and include **FR_FAIL_STEP**, **FR_INVEST**, **FR_REPUTATION_POINTS**, **FR_DIFF_INCREASE** and endless mode.

Some of our final UML is pictured below. Due to the increase in size of the project, PlantUML was unable to fully render it, which was a drawback we hadn't considered when choosing the program in the beginning.