Architecture

Group 18

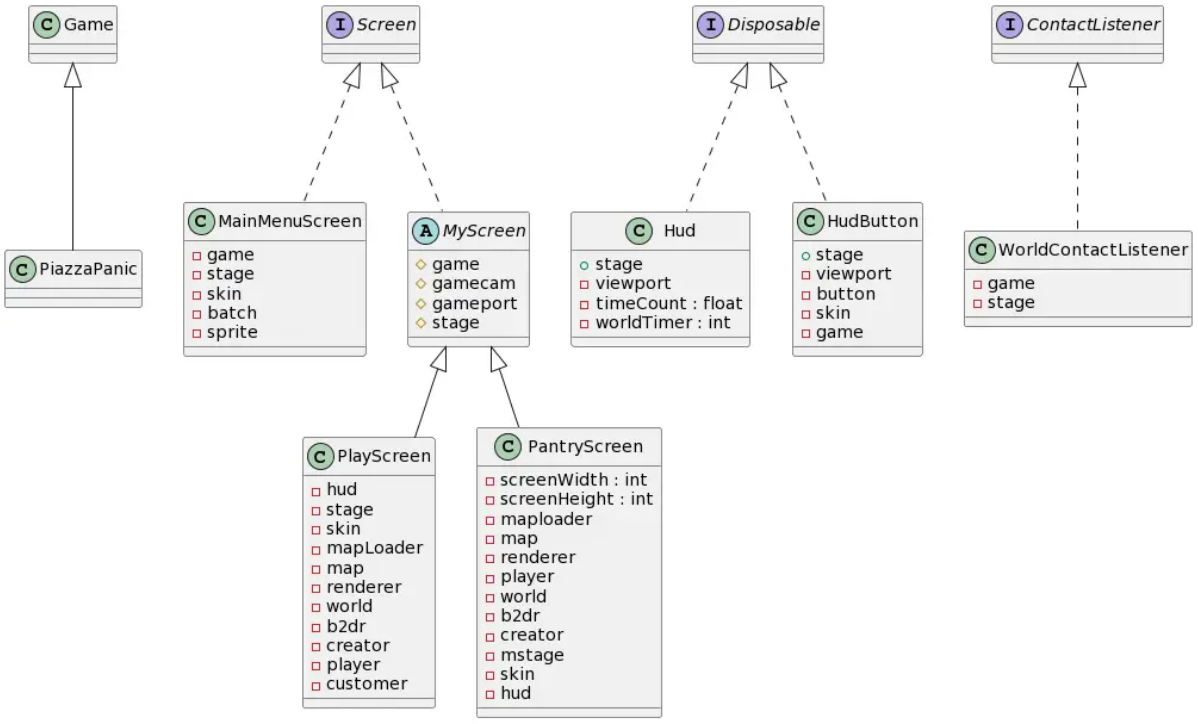
Team B

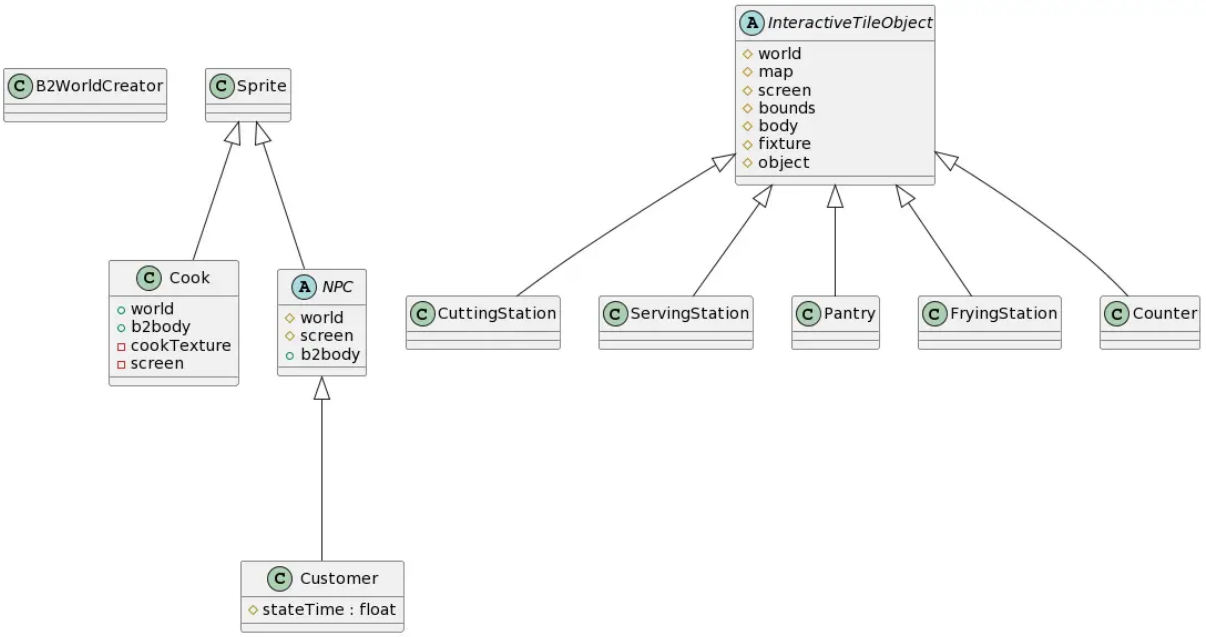
Olivia Betts
Zac Bhumgara
Nursyarmila Ahmad Shukri
Cameron Duncan-Johal
Muaz Waqas
Oliver Northwood
Teddy Seddon

Diagrams

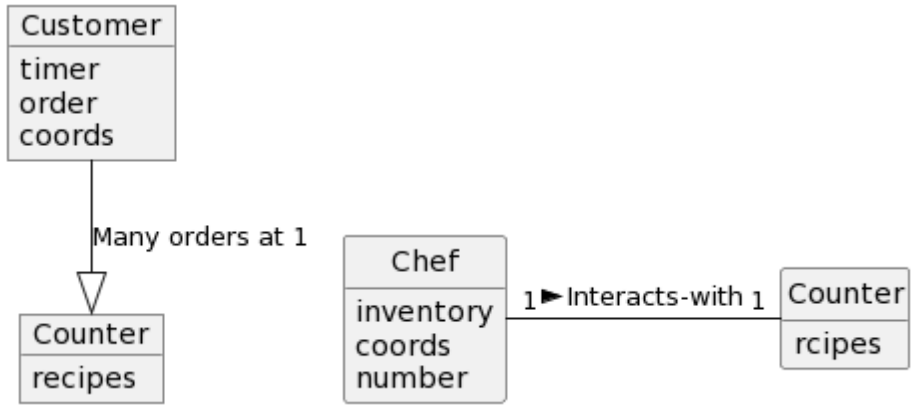
We have used Unified Modelling Language (UML) to model the structure and behaviour of our software system. UML diagrams can be used to illustrate our project before it begins or as project documentation once it has begun. To create the diagrams, we have used an UML extension on Google which is PlantUML 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.

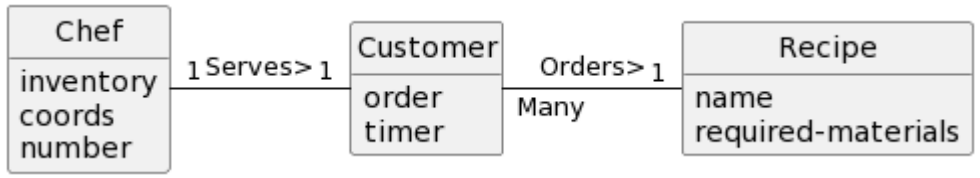
Structural diagrams:





Behavioural diagrams:

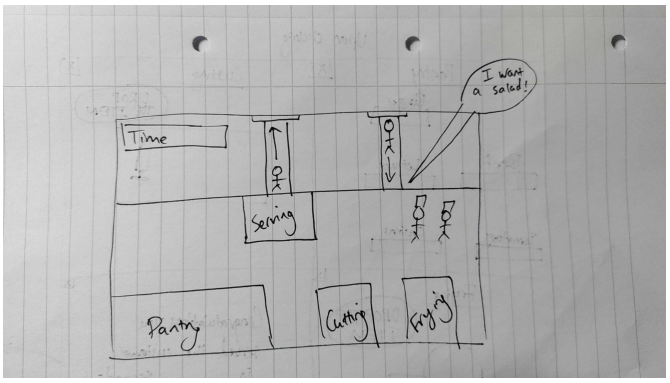


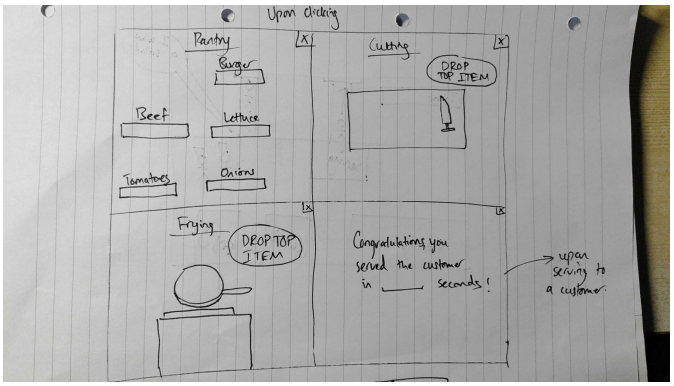


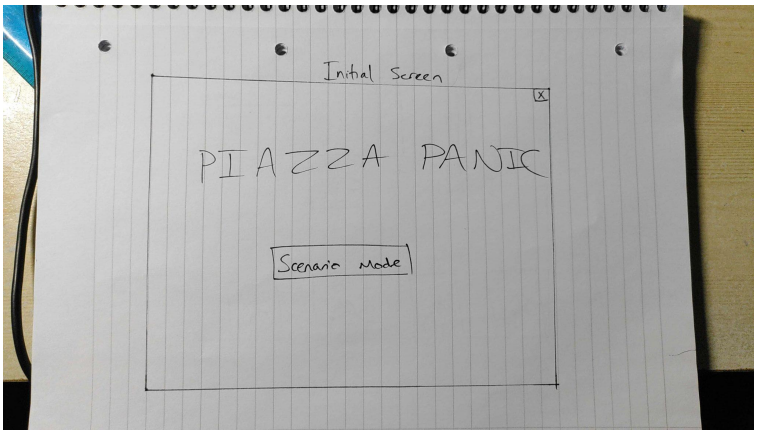
Development Over Time

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.

Initial diagrams:







Some CRC cards:

Class:
Cook
Responsibility:
Should be able to switch between the cooks Should be able to move the cooks Cooks should be able to complete an action with what is in front of them If cooks bump into each other nothing happens Cooks can hold two things at a time but cannot complete any actions during this
Collaboration:
Cooking Pantry

Class:
Recipe
Responsibility:
Store the recipe which is needed for the customer
Collaboration:
Ingredient/Utensil

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.

The first requirement which we met was that the game should be able to be played in scenario mode or infinite mode (**FR_GAMEMODES**). For this to happen we would need a menu screen, which allows the user to switch between the two game modes. We decided to implement a main menu screen when our game loads up – this required us to create a new class which was not initially in our architectural designs.

Another requirement was that there should be customers arriving at intervals and waiting at a counter (**FR_CUSTOMERS**). From here, we made a customer class, including methods such as having a set route out.

The requirements stated that a player should be able to switch between three different cooks (**FR_MULT_COOKS**) and that if the cooks bump into each other, nothing happens (**FR_COLLISION**). The ability to switch cooks and the collision prevention were also methods to add to the cook class. The cook can also hold a specific number of items (**FR_ITEM_INTERACTION**) and we represented this using a stack. We also added the ability for a cook to be able to prepare ingredients at stations (**FR_PREPARE**).

In terms of the recipes, each recipe has multiple steps which must be carried out before the dish can be served (**FR_RECIPES**). There are different recipes for salad, burgers, pizza, and jacket potatoes, although only salad and burgers need to be implemented for Assessement1.

The final requirement which we implemented in this project was the fact that the game will use a mouse and a keyboard to control cooks (**FR_CONTROLS**). We realised that we would create a ‘user input’ class, which processes the instructions which the user gives to the cook and subsequently moves the cook sprite around. Our rough plan was to use the arrow keys to move and click on the mouse to interact with something (such as the pantry/cooking station).

Another requirement was that the time taken to serve each customer should be displayed after they have been served. For this reason, we added a timer method to the customer class.

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.