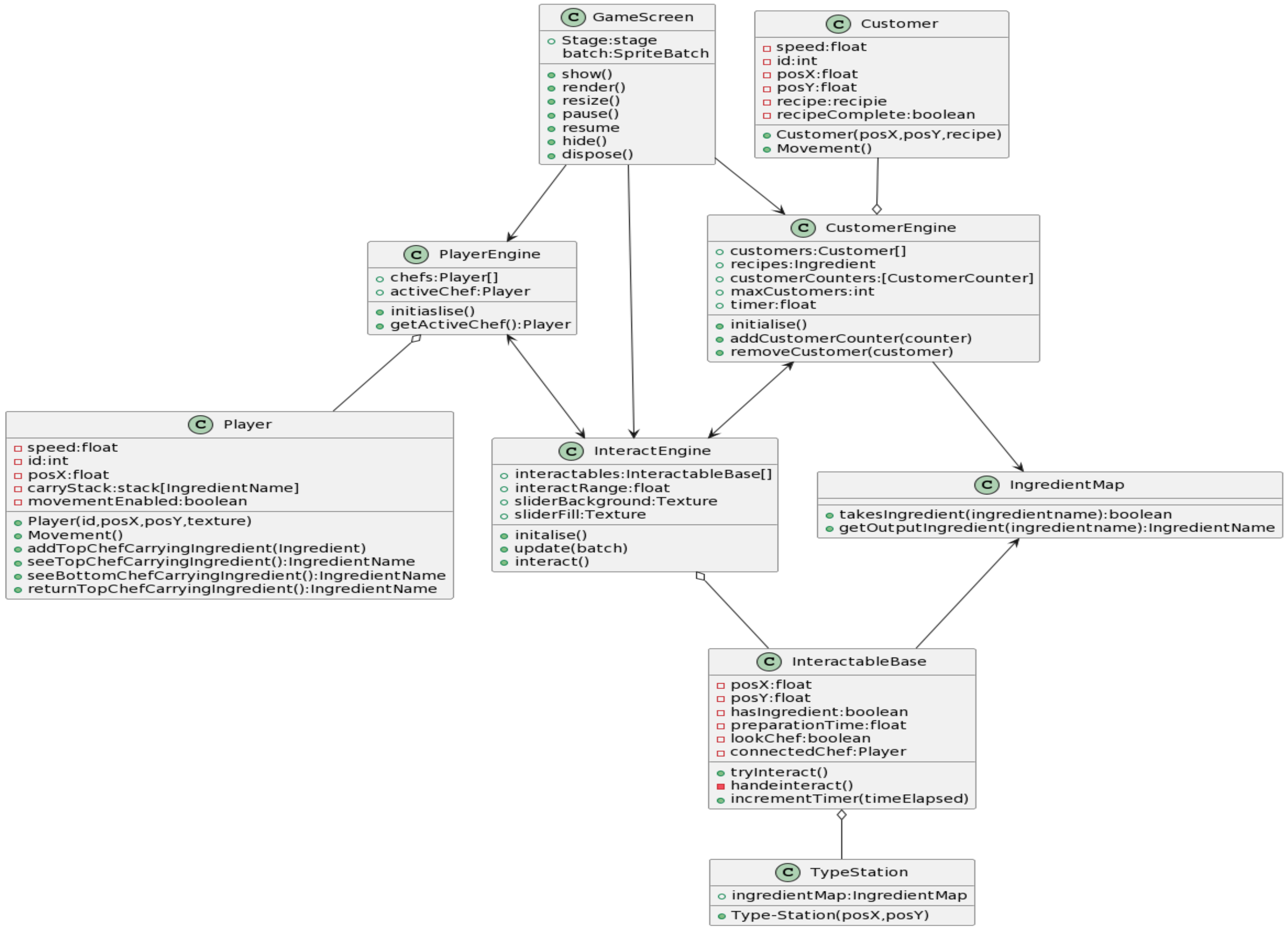
<u>Architecture</u>

“Lucky” Team 13

Team 13

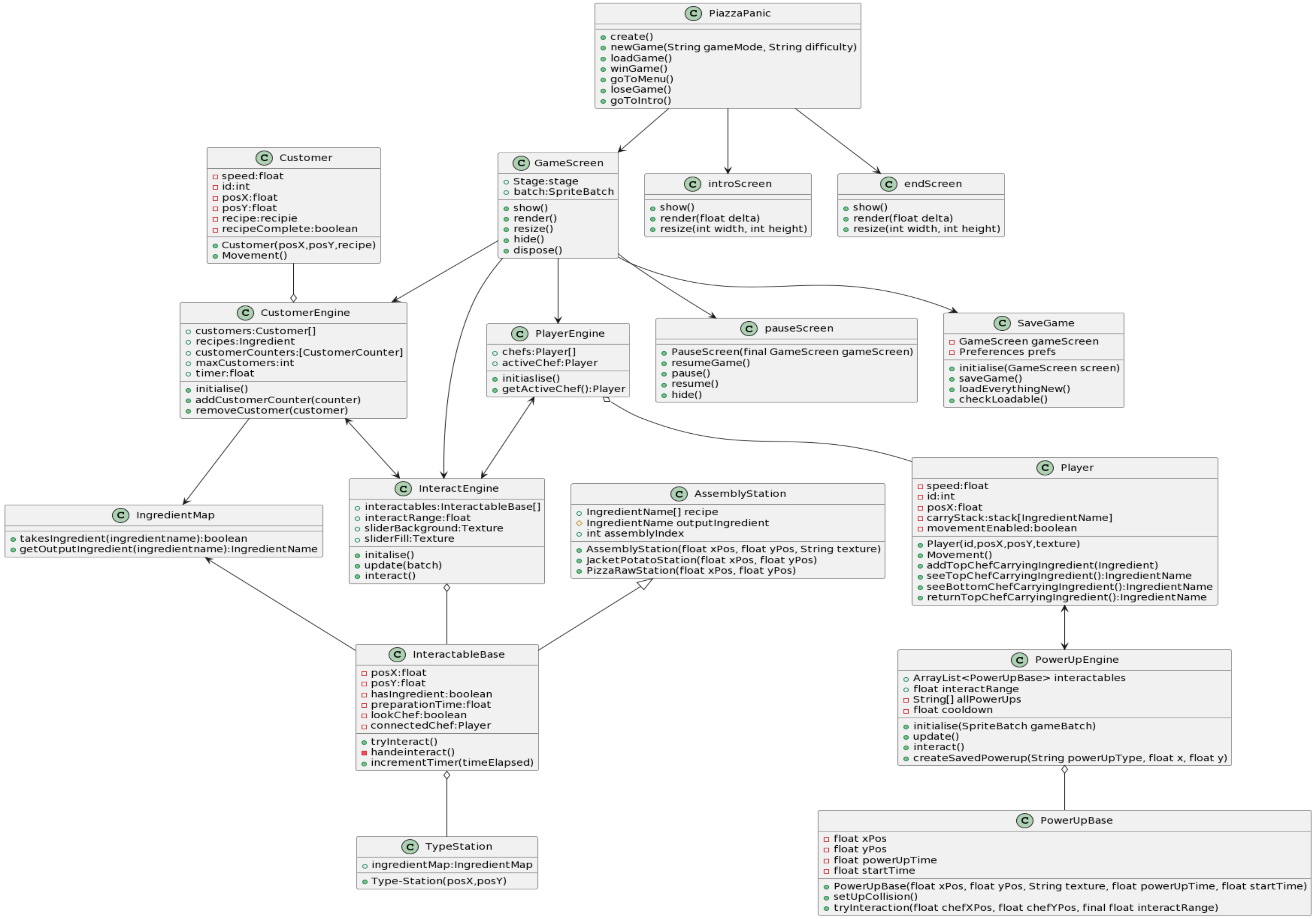
Yuxin Wu
Bailey Findlay
Elizabeth Edwards
Chenxi Wu
Alex McRobie
Yihong Zhao

Class Diagram version 1:



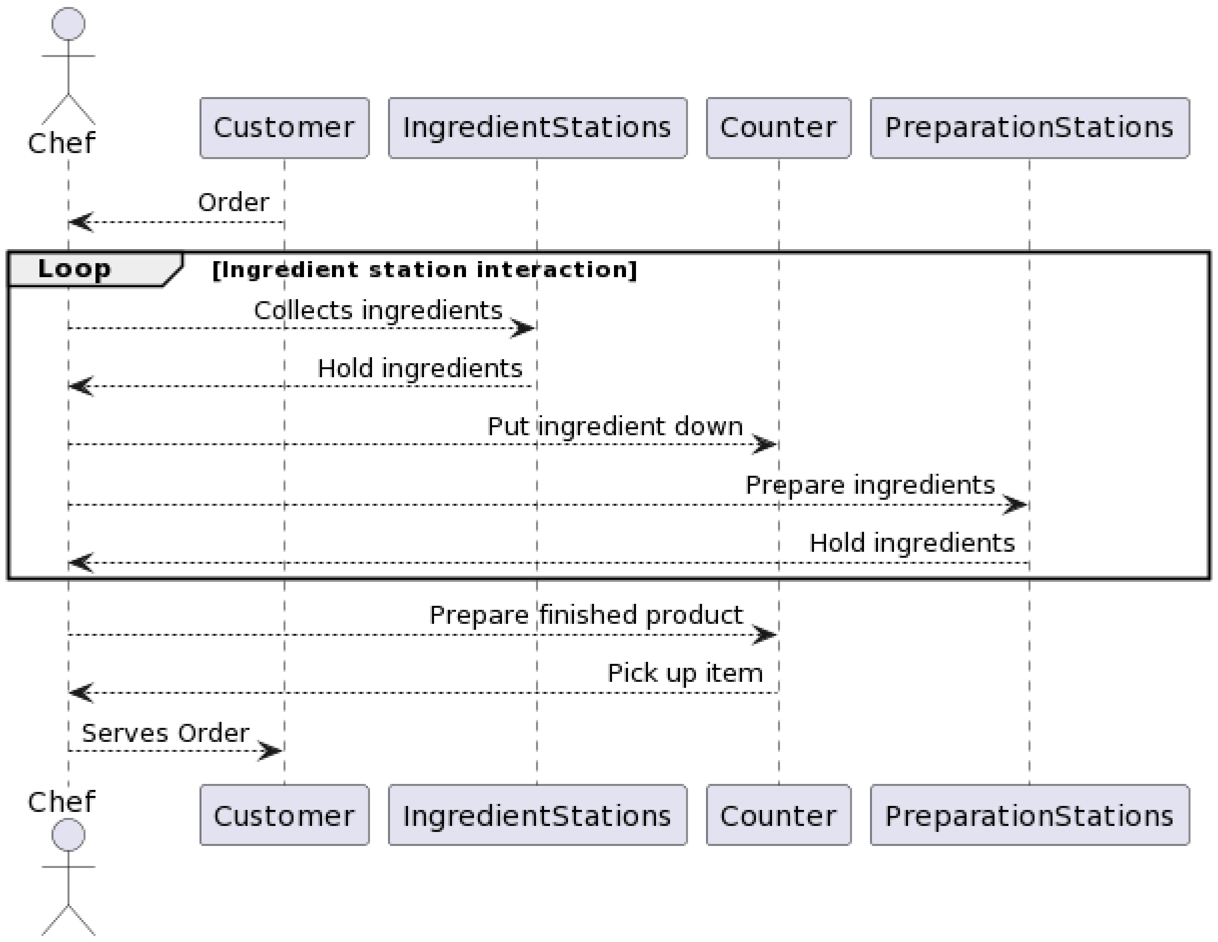
A UML class diagram was created using plantUML.

Class Diagram version 2:



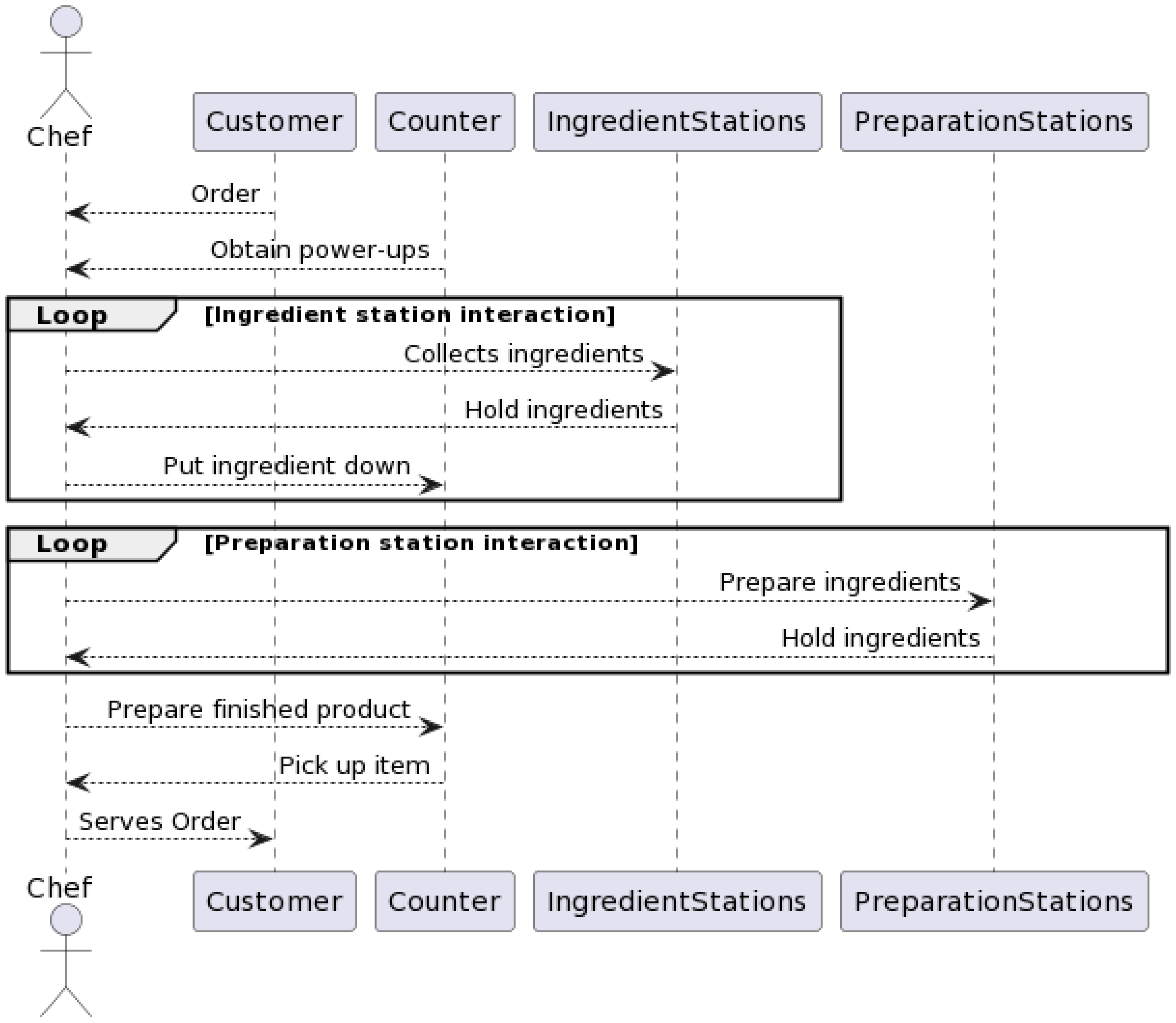
A UML class diagram was created using plantUML

Sequence Diagram version 1:



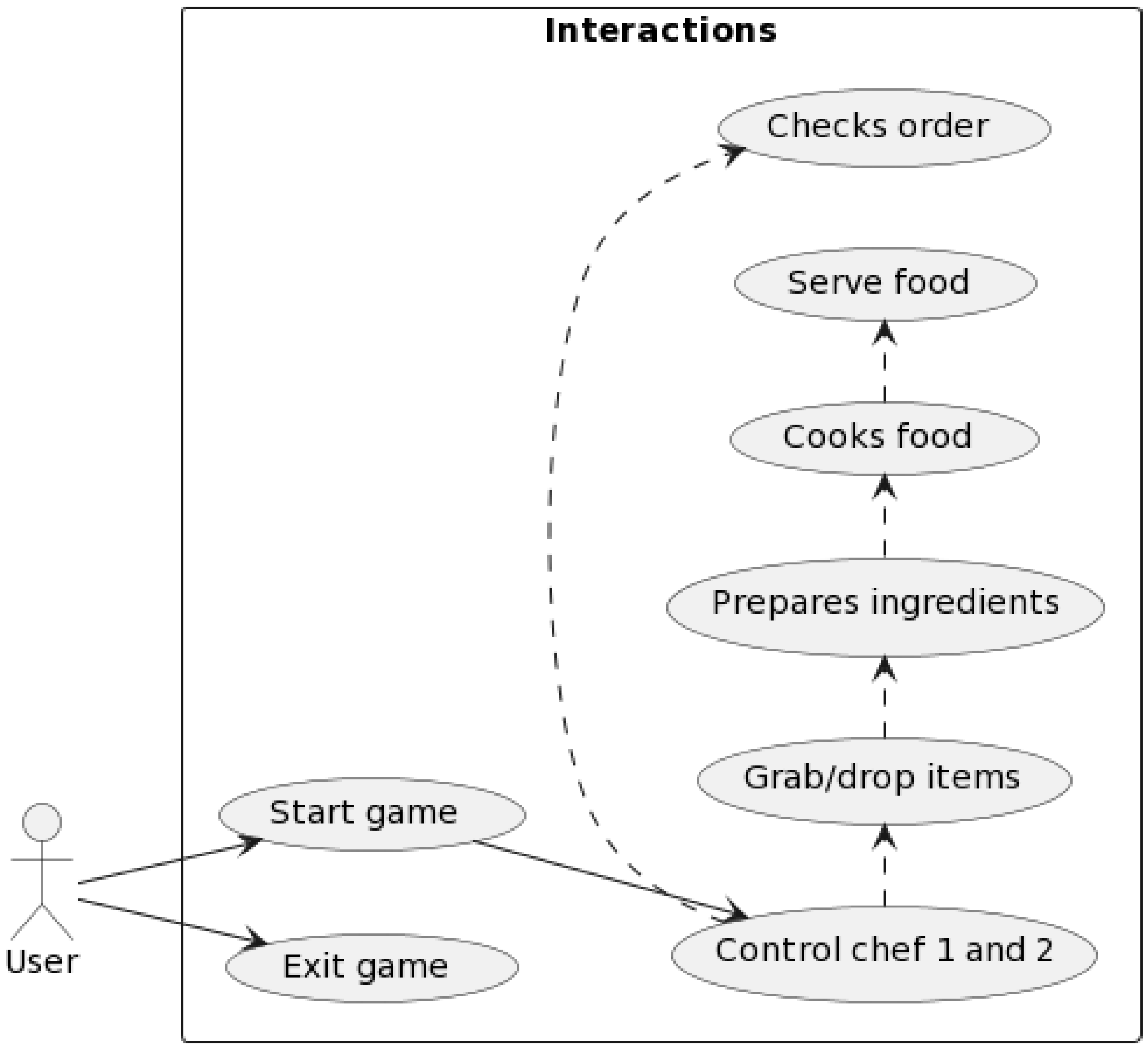
A UML sequence diagram was created using plantUML

Sequence Diagram version 2:



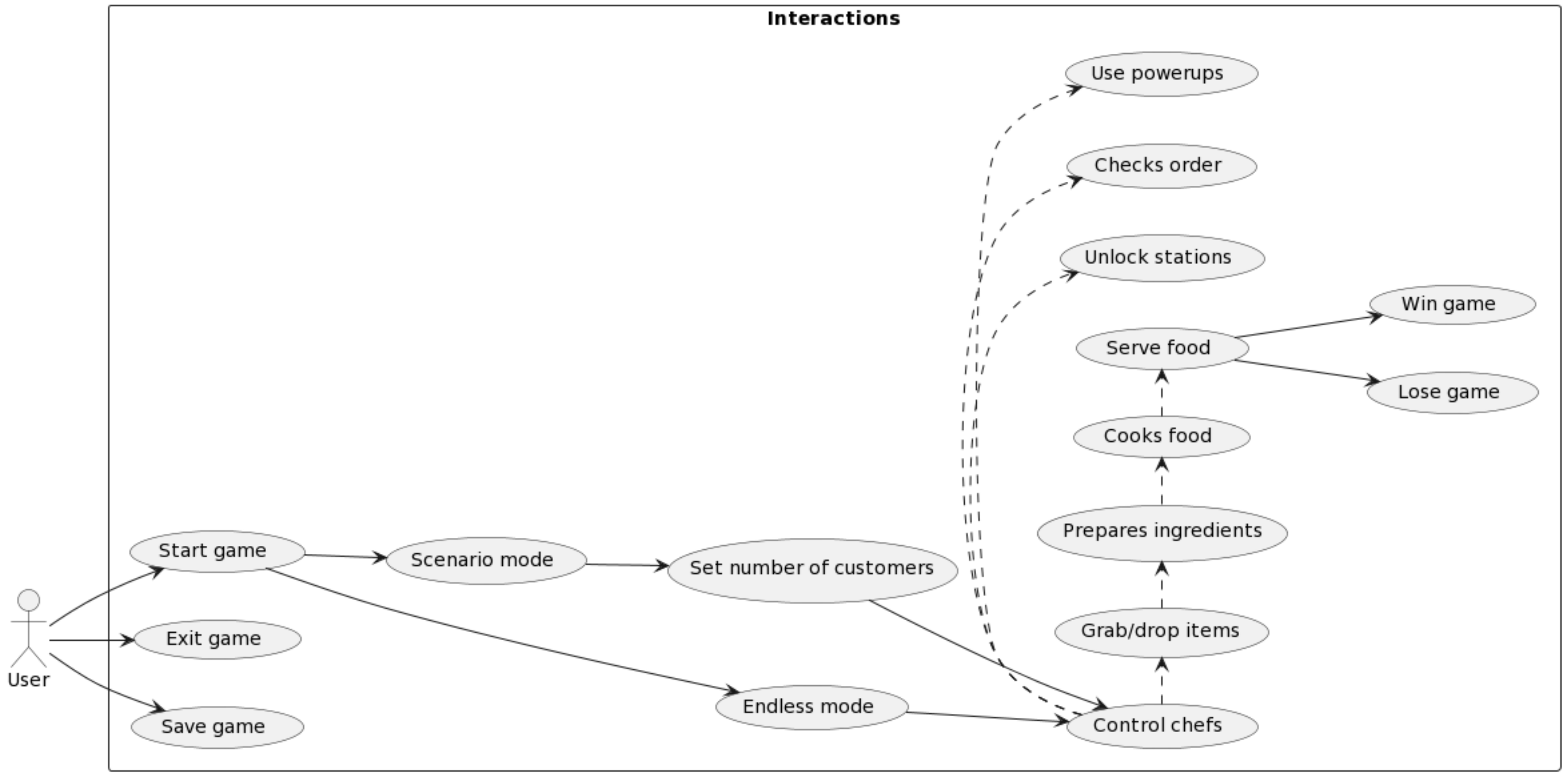
A UML sequence diagram was created using plantUML

Use Case Diagram version 1:



A UML use case diagram was created using plantUML

Use Case Diagram version 2:



A UML use case diagram was created using plantUML

All of the diagrams named below can be found in the Architecture Diagrams folder on the website.

The initial outline of how the game was to be designed was based on a simple outline of how the classes were to be organised and work together [ArchitectureRoughDiagram.png]. This was built with the initial understanding of how the LibGDX library worked.

The game is all held within a core folder, with a desktop folder that sorts out platform specifics. After the discussion with the customer, it was decided that it would not need to run on any mobile devices, so only the desktop features were implemented. All of the architecture diagrams were made for use within the core directory of the game.

The game is drawn to the screen using the GameScreen class, and that class should avoid handling anything unnecessary such as any game logic. The focus of that class should be on rendering the game and that only, to avoid slowing down the game. This is in accordance with the non-functional requirements NFR_lag and NFR_loadtime.

The engine classes would handle user inputs, and allow for interaction between the player classes, and the different station classes. By using engine classes it provides modularity to the system, as we have multiple chefs, and the ability to interact with different stations. The engines also handle rendering for each player or station instance, to stop cluttering and size of the GameScreen class. Allowing the engines to control the inputs and have the knowledge of what each station/player is doing means that the only communication is between the engines. The UR_Chef_Swap requirement means that each does its own actions, even if not controlled, and the engine separation makes this easy, as it holds control over all chefs at all times.

Note at this point no customer class had been decided on yet, as it was initially planned that the IngredientMap convenience class would hold recipes for food that would then be assigned as active to the player.

From the initial outline of the classes, some class responsibility collaboration cards were made [classcards1.png] and [classcards2.png]. This helps to visualise the tasks that each class will be required to carry out, and the different classes that they will interact with.

These cards were useful as they allow for vagueness with the description of each class's responsibilities to allow for you to assign their responsibilities without a full understanding of how the library or each class will work on a code level.

For rendering the game, it was decided that each instance of player and interactable would load its sprite. This would then be added to the sprite batch in each of the engine classes that would then finally be loaded in the GameScreen class. This is with the non-functional requirement NF_lag in mind, as this method of batching will help to reduce the lag and load time between frames. Each engine would hold a list of its elements, e.g. PlayerEngine would hold a list of Player classes, one for each chef. In accordance with the user requirement UR_Chef_Move, each Player class would hold its relative position on the screen and have the function to be moved. The actual movement would in the end be called by the PlayerEngine class.

Here it was decided that each type of station, be it baking, frying or so on will inherit from the class InteractableBase. Each of these bases will control its own interaction timing and what ingredients it will accept or give out. Different constructors can be used depending on if it is a station for turning ingredients into new ones, or if they are stations for only taking items or giving items. This means that new types of stations can easily be created when they are needed, such as bins or different types of ingredients. This also means that each station can deny interaction based on what the Player interacting with it is holding, fulfilling the functional requirement FR_Invalid_Interact.

With these cards, the customer class was created, which would allow customers to come to the restaurant as actually rendered sprites. This also means they can interact with a counter to allow them to be served, as required by the user requirement UR_Customer_Counter. It also means that there are customers to serve meals to, and when you run out of them you can complete the win condition as stated in the user requirement UR_Win.

Behavioural diagrams were created next, as use-case diagrams, to model the behaviour of the system and help capture the requirements of the system as well as identifying the different interactions between the system and its actors. In the first version of the diagram, the player who is represented as the actor, is the main stakeholder of the system. This allows the diagram to show the different steps the player can do to navigate through the different scenarios presented to them. For example, in the very first basic version, the actor is linked to start game and exit game. When starting the game, the user would be able to navigate through the different actions needed to then complete the game, such as controlling the different chefs.

In the second updated version of the use-case diagram, we implemented a new gamemode: endless mode. This can be shown through the requirements of UR_Gamemodes, which explains the different game modes the user can choose at the start of the game. This meant that instead of having five customers coming to order, it would continue infinitely. In the first version there was no possible way for the player to lose, therefore in the new endless gamemode, we have implemented a loss case. This was implemented through the idea of reputation points, starting with 3 points and having one taken away each time the player would not manage to complete an order within a given timeframe. This can be seen through the new user requirements of UR_Endless, UR_Loss and _Reputation.

As well as the endless mode, we have also implemented a save game function which is exactly what the name suggests. This would allow the user to save their process throughout the game, without losing any progress. This is in accordance with the user requirement of UR_Save.

The last features that we have implemented are the use of power-ups and station unlocking/chef unlocking in this version of the game. The use of power-ups, which is directly related to the functional requirement for FR_Power_Ups, were the ideas of items that the user could pick up to give them a more advantageous experience, such as a speed increase and a few more. Station unlocking and chef unlocking is available by being purchased by in-game currency which can be earned from completing orders set by the customers, seen through the requirements of UR_Investment, which basically means users would be able to invest their earnings into unlocking chefs/stations. These can then be used to give the user an advantage in order to traverse through the game easier. All of these new cases introduced in the second version of the use-case diagrams give a wider and better representation of the behaviour of the system and its interaction with its actors.

The sequence diagram [SequenceDiagram.png] shows the process that the chef will undergo when completing the task of creating a meal for a customer. It gives a general flow that the player controlling the chef will follow, which creates the core system of gameplay.

The player will systematically pick up the needed raw ingredients to create the requested meal for the customer, and then interact with the corresponding preparation stations to make each final ingredient for the meal. By putting all the ingredients down on the final counter that the customer is waiting at, you then prepare the finished product for the customer and complete the main objective which is to serve the customer. This is then repeated for each customer and in turn, achieves the requirement UR_Win that they will win by serving all customers.

In our updated sequence diagram, we made slight changes to the original diagram as not a lot needed to be changed even with the new requirements. The main change that was added was separating the loop of preparing ingredients into ingredient station interaction and preparation station interaction. This makes it clearer and gives a better understanding of the sequential order of interactions within the game by denoting that one will necessarily involve the other in order every time.

We have also added the use of power-ups, which can be obtained by interacting with the symbols on top of work counters, to give users an advantage when playing the game in order to get a higher score or a faster completion time. This could be auto-completing dishes, anti-burning, speed increases or increasing customer patience.

With these rough outlines of classes, the full UML class diagram was created [ClassDiagram1(old).png]. This is a fully fleshed-out version of the class structure our game uses, with a change made regarding the Recipe class later on as it was mostly a placeholder for a concept at this stage.

At this point, the implementation had just begun and it was clear how the different engine classes would be structured and how the GameScreen should be done. The GameScreen will interact with each of the engines, and the engines only, while the engines won’t interact back with the GameScreen. This is just the GameScreen updating its batch for each frame to then render.

Each engine class will be initialised during the calling of the show function of the GameScreen. This is the creation of the window where the game will be rendered, by the continuous calling of the render function in GameScreen.

For the whole game, the only inheritance is from the different stations inheriting from InteractableBase. This is because all stations in essence give or take different ingredients at different times. By using different constructors for InteractableBase, we can differentiate between preparation stations, ingredient stations, assembly stations, and bins. Each of these can then be created with its own ingredientMap which will tell the game what it will give or take from the chef when interacted with.

Finally, further into the implementation stage, the class diagram was updated to [ClassDiagram2.png] to replace the Recipe class with the correct IngredientMap class. This is a convenience class that holds a list of the ingredients in a recipe and can output the created meal from those ingredients. This means that new recipes can be easily created by just creating a new IngredientMap. This system completes the user requirement UR_Recipe which requires the user to be able to make different recipes.

After taking over the project, we have decided to update the class diagram to make it simpler and not include as many attributes and methods as well as directly including all java classes. For example, in the original class diagram in the player class, player X and Y position were both included as a method as well as move(). Therefore in the updated version these were removed and replaced with just move().

Other main changes in the class diagram were some attribute and method names, such as pushIngredient, peekIngredient etc, as these would be too difficult to understand for users who do not come from a coding background. Therefore, these were changed to names that are easier to understand for everyone. For example, peekIngredient has been changed to SeeTopChefCarryingIngredient.

Our second version of the class diagram introduced new classes and attributes such as PowerUpEngine, PowerUpBase, assemblyStation and a few more, which are all related to the requirements given. PowerUpEngine and PowerUpBase are extended from the player class which shows how powerups extends the relationships between it and the player class. These new classes gives a basic understanding of the important classes, attributes and methods used in the implementation. As well as powerups we have added the features of game pausing and saving etc. The game save feature is directly related to the user requirement of UR_Save.

Overall, in the new updated version of the class diagram with updated classes, all of the methods and attributes are used in the final implementation, which shows the consistent naming of constructs.