Architecture

Group 27 BlackCatStudios
Jack Vickers, Hubert Solecki, Jack Hinton, Sam Toner, Felix Seanor, Azzam Amirul Bahri

Taking over from Group 30 Triple 10
Kelvin Chen, Amy Cross, Amber Gange, Robin Graham, Riko Puusepp, Labib Zabaneh

Design Process

Link to previous Architecture document here.

The first steps of our design process were exploring Team 30's code and understanding how they chose to implement each feature. After that, we decided on a limited set of improvements to radically change the architecture to allow for improved functionality. Due to our previous experience in assessment one, we had very few architectural changes over the course of the project. We based some of our architecture on the previous designs, taking the time to make improvements. These improvements came in the form of reducing the complexity of rendering and allowing for ease of use of various image formats. We made adaptations later in the design process to functions such as Interaction to return a float, this being the time left to lock a chef in place for when using the chopping station and providing ovens with a Consumer<Boolean> to enable potato orders to arrive. We also noticed that our classes tended to fill up with getters and setters for private variables over time due to either testing requirements or other classes needing a reference to particular variables.

Over the course of testing, we also had to alter the architecture to support testing better, such as extracting the machine construction methods into a class called ConstructMachines. While not necessary for gameplay purposes it allowed us to test these methods.

We also noticed that we could perfectly plan for all classes or enums used in the game as we had to add in more data storage classes than we originally had planned such as with LinePoint and more.

Some unusual features in our game are how we designed critical components to be independent of each other, e.g. preventing customer classes from referencing chef ones and vice versa. Because of this design, we have a deficient coupling factor between classes. However, we were then forced to use functional references in the form of Consumers<> or Function<> to allow for communication between separated classes. The low coupling factor was an essential part of our architecture design as it reduced the complexity of adding new features, as you didn't have to worry about changes in a different class breaking the class you were writing.

Classes such as CustomerController, MasterChef and GameScreen directly meet our requirements such as FR_MENU and UR_CHEF_ CONTROLS. However, many of the classes existed solely to support other classes to reach these requirements.

Initial Concepts

We wanted to design our game with ECS in mind due to the need for having many different customers, chefs and stations. For this, we needed an abstraction for classes that can be attached to objects. These objects will need to be represented by a class, in our instance called GameObject. This will need to store:

- 2-dimensional position
- Abstraction for images for (Texture region and Sprites)

We wanted to design the stations to be dynamic and flexible, and easy to extend for this reason we needed an abstract class Station. Containing all the base variables that polymorphed stations used. This extends our ECS system for updating every frame. Our

recipes needed to be dynamic thus we decided to have an abstract step, with a time step and an interaction step inheriting it. Allowing us to construct unique recipes per food or station.

As we use dynamic objects we need to make sure our interactions are dynamic to make map construction as simple as changing the tilemap. For this reason, we use interfaces that can be implemented to define what interactions can be used. The interact function will take in an interaction method and try to find all classes that inherit the interface. It has to determine the closest script via the separating axis theorem due to the different sizes of rectangles.

Our saving system had to take into account not all of our classes are serializable, thus we decided that we will call every core class to save itself into a serializable parameter class first. We chose not to include powerups in saves due to their temporary nature. If the player wants to use a powerup he must use it in one play session. It could appear strange if the player comes back a week later and is significantly faster than they expected as well. If the player is holding a superfood or increased their reputation these will be saved.

CRC Cards

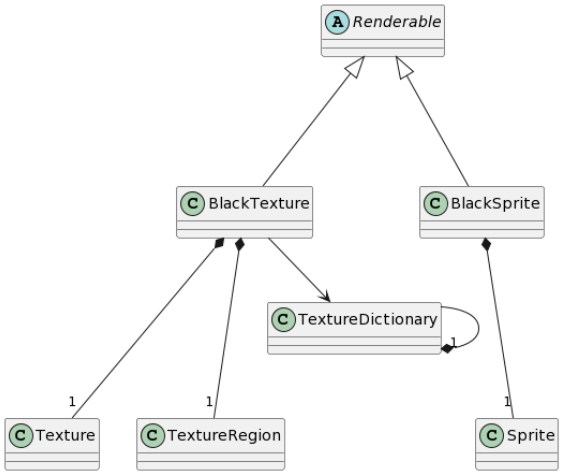
To see our CRC cards look at our website

We used CRC cards to design our classes and how they will interact with each other. This helped us design each system so it could be implemented without many conflicts and rewrites. This follows our initial concepts for our classes with more details.

Structural Diagrams

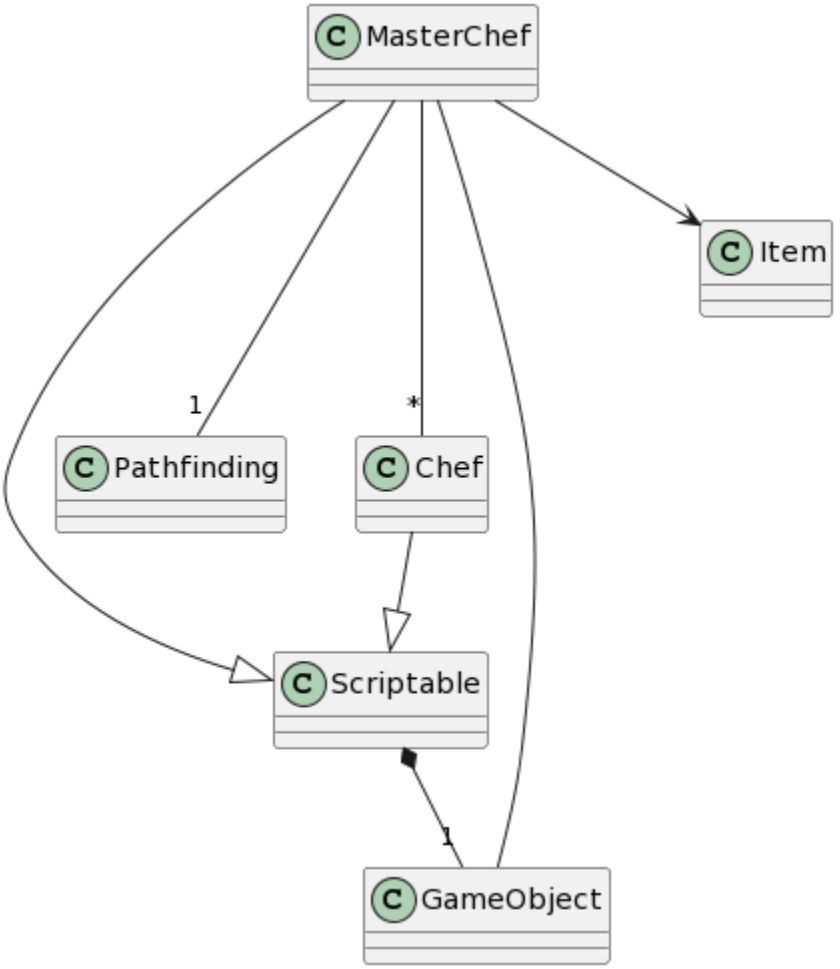
We decided to use PlantUML because it's simple to learn and create complicated UML diagrams that are automatically spaced. With plugins we are able to insert and edit diagrams in Google Docs, allowing us to make changes later and easily add the images into our documents without having to take screenshots or hand draw them. We considered draw.io this would enable us to draw diagrams with various shapes, however, it would require us to manually decide the layout. This would be most appropriate for decision diagrams. However, we generally reused Team 30s decision diagrams. FigJam didn’t seem to produce engineering diagrams easily.
We created a class diagram, with UML Class Diagram using plantUML to consider different components of the program and model the various classes and their relationships. We could decide from each class their main attributes and methods and whether any could be grouped into packages or relate to a more generalised abstract class. The Class diagram used for the game implementation is shown below.

Renderable Class



Renderable Class and how it

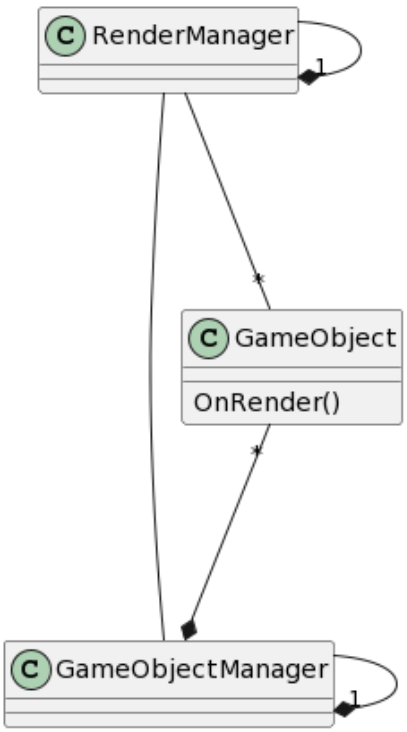
Interacts with other classes

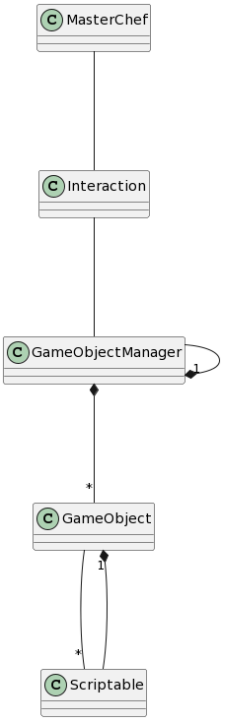


Rendering loop

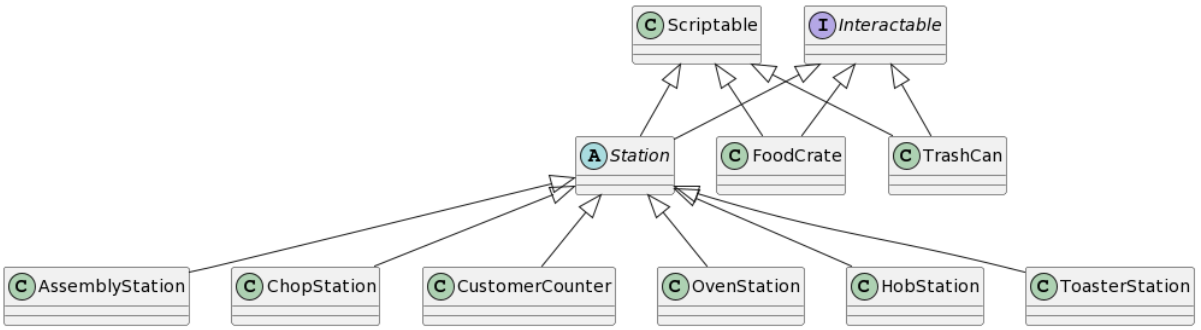
Masterchef and how it

generally interacts with other classes

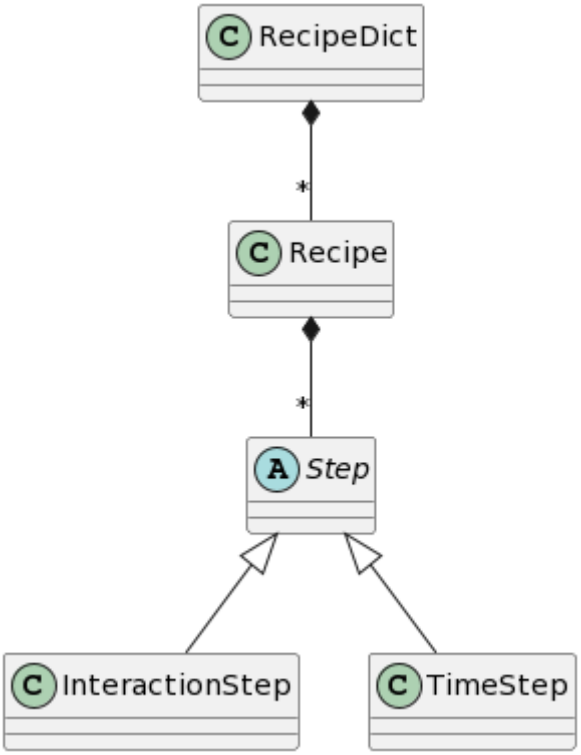




Stations class diagram



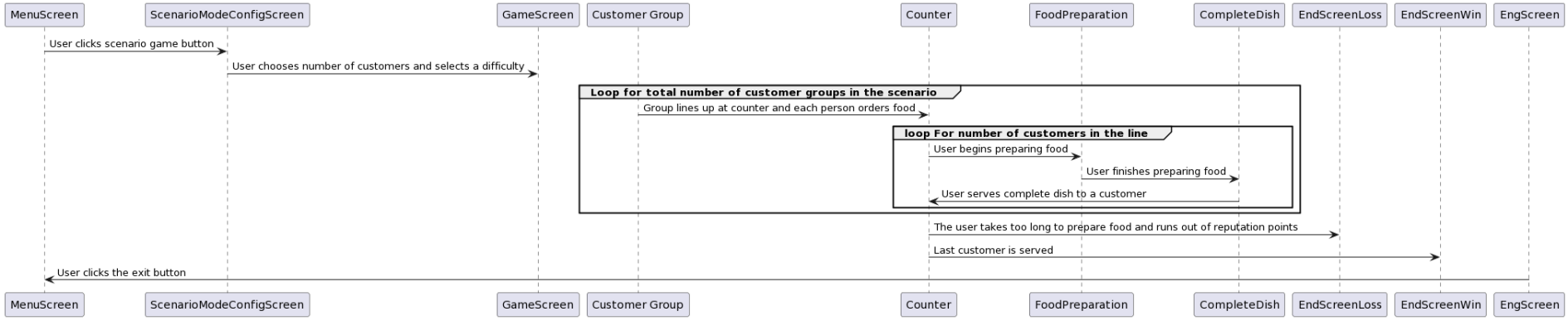
Recipes diagram



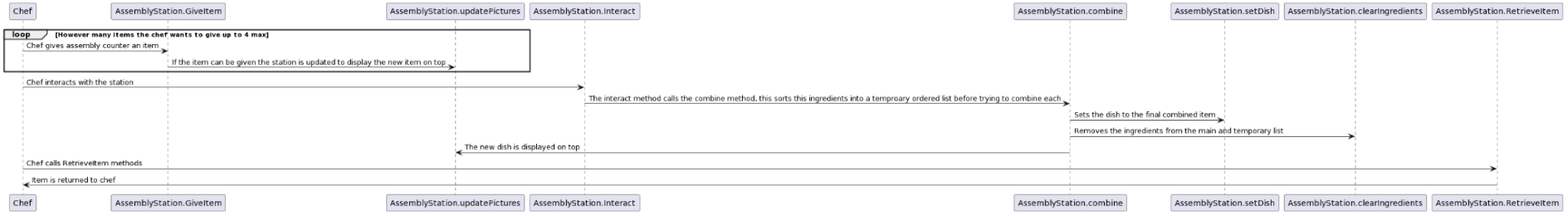
Behavioural Diagrams

Along with our class diagram, it was helpful to consider how the user would interact with this software. As such, we created a sequence diagram to show how objects would exchange messages during a given scenario and an activity diagram to map out the steps taken for the user to complete an action. It was created using UML Sequence Diagrams on PlantUML.

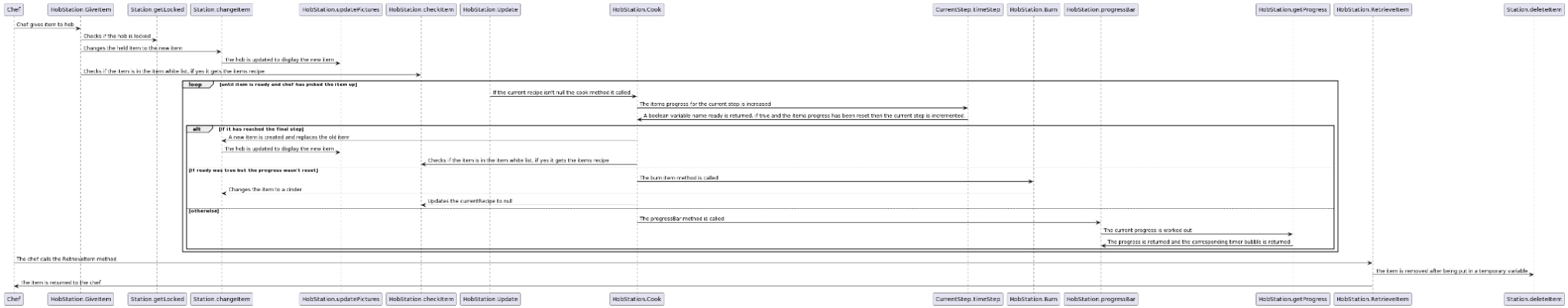
Sequence diagram for the scenario mode



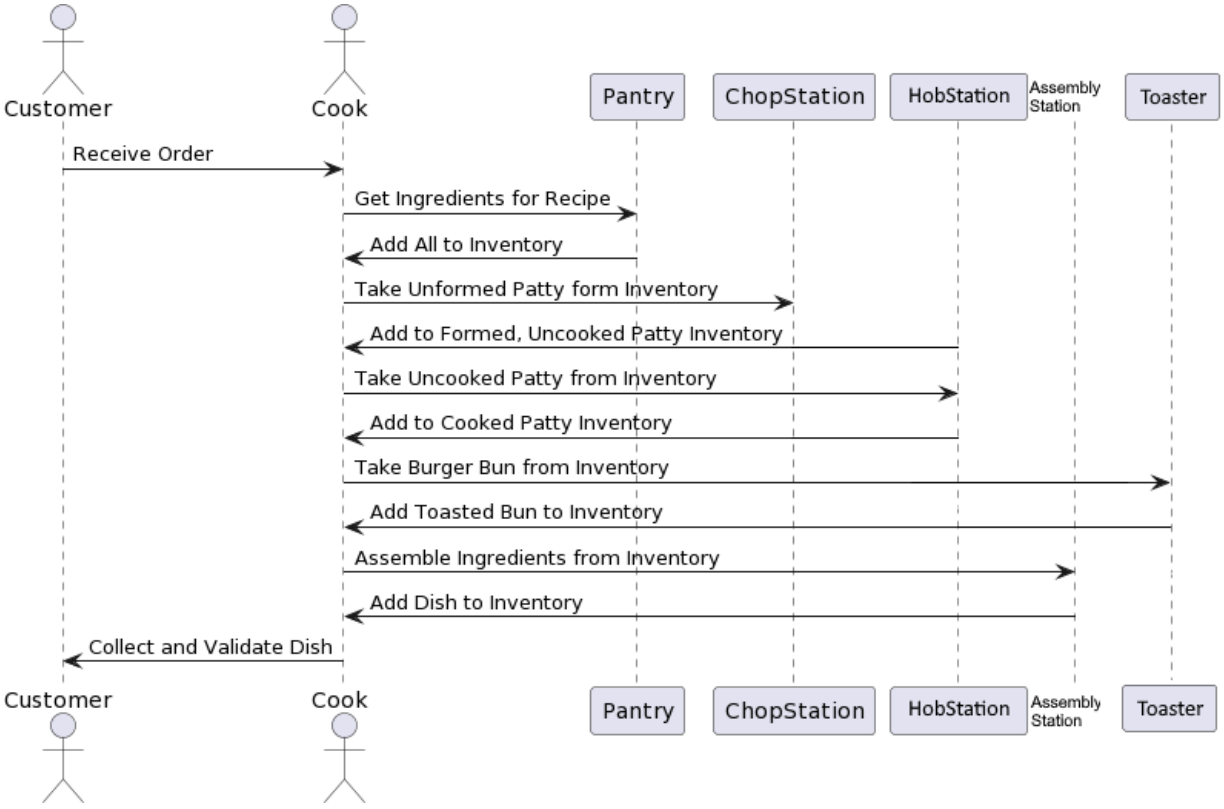
Sequence Diagram for using an assembly station



Sequence diagram for using the hob station



Sequence diagram for the process of cooking a burger patty



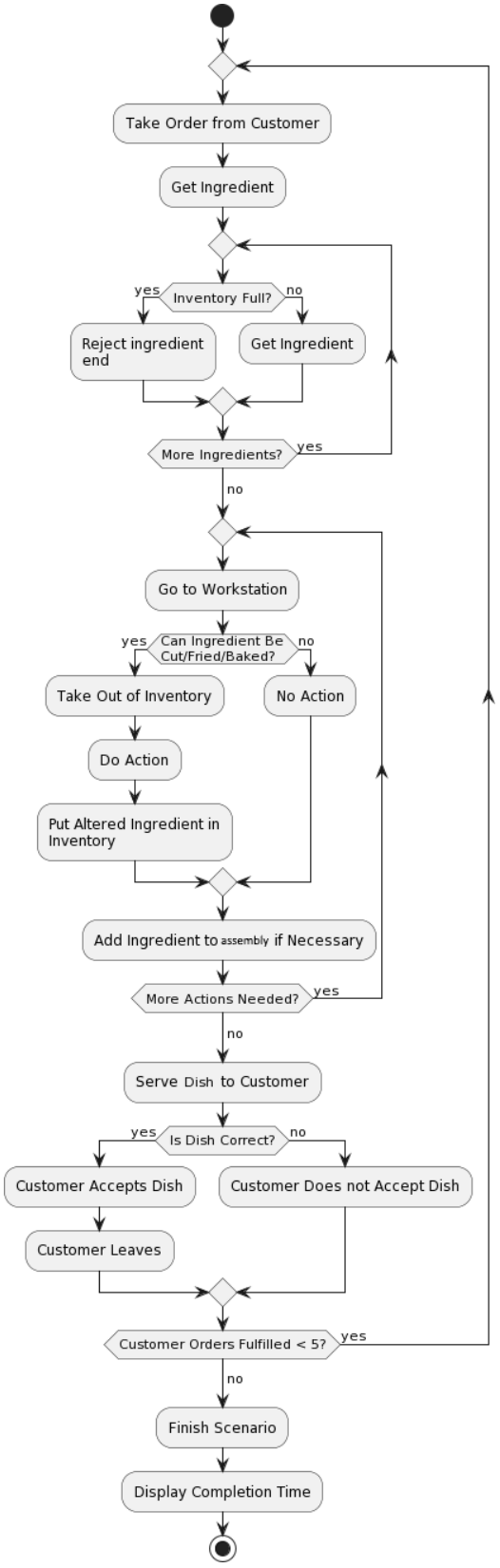
We decided to modify this diagram as it is very similar to ours. The only difference is that assembly stations are used to create food and not plates. They also had an error that the burger was fried on the ChopStation. The toaster class was also added in to toast the bun.

Following this diagram, we could expand on each action to show what steps must be taken in the game. For example, when adding ingredients to inventory, the system must decide whether this is allowed depending on whether the inventory is full. For this, there is an activity diagram, as detailed on the next page.

In this diagram, we separated the stations to clarify which station each action is to take place in and described what each activity contains. The design hasn't changed much for this diagram, as the product brief described the actions that needed to take place, so they cannot change during development. Many actions in the sequence diagram directly relate to our defined requirements. The first action defined where a chef receives the order from a

Customer and directly relates to UR_CUSTOMER_ORDER. Specific requirements also relate to the areas illustrated in the sequence diagram, like Pantry matching with UR_PANTRY.

Activity Diagram



Triple10 created an Activity Diagram using UML (plant UML) to show the steps a user takes in the game, from taking an order to displaying the completion. This diagram includes decisions. The diagram shows a brief overview of the actions taken between the customer first making an order and when they receive and validate the dish and its steps based on the sequence diagram. We have used this diagram to build up more detail in each game process to ensure we follow the requirements we have outlined and that they are correctly implemented. In the sequence diagram, the user simply got the ingredients from the pantry and was put into their inventory, which has been expanded to include a check that the inventory is not full. Some tasks have been generalised. For example, all workstation tasks have been combined to reduce overlap and confusion. Requirements and how they relate to the Activity Diagram:

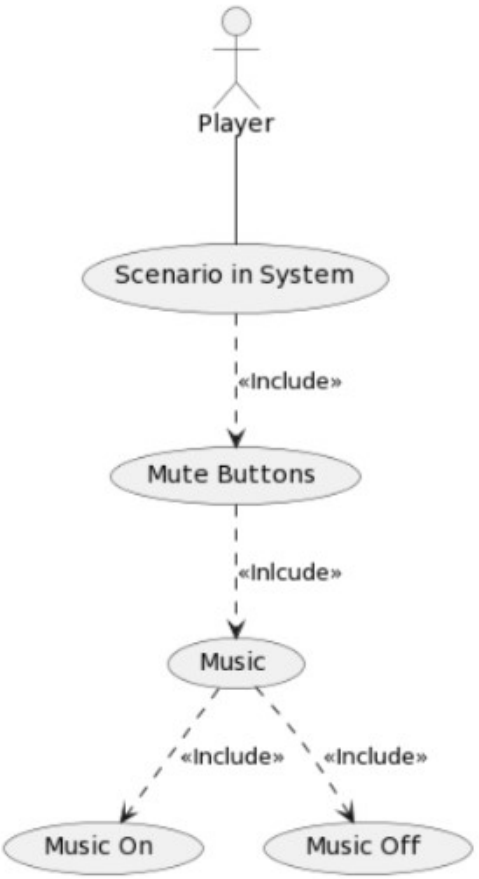
1. Receive Order → FR_ORDER
2. Get Ingredients for Recipe → FR_PANTRY
3. Interactions with Workstations → FR_MULTI_ITEM, FR_INGREDIENTS, FR_STEP_VALIDATION, FR_COOK_BUSY
4. Collect and Validate Dish → FR_DISH_VALIDATION

The activity diagram is very similar to our game so we decided to only modify it. We had to make very few changes to the flow diagrams for the scenario mode as we only had to change the <5 to be <n as scenario mode can have almost any number of customers. We also had to add in logic that if reputation <= 0 then it should end.

Use Case Diagrams

We decided to reuse some of our use case diagrams from our past assessment as fundamentally the systems are not that different, instead editing some of the scenarios and postconditions to adjust for new factors.

Use Case: Mute Sound and Music
Primary Actor: User/Player
Supporting Actors: N/A
Precondition: Game has started
Trigger: The user does not want to hear the music or sounds within the game



Main Success Scenario:

- During the game, the user easily located the buttons in the top corners of the screen that switch off game music and sounds
- The user points the mouse at the buttons and presses them
- The sounds and music switch off and the display of the button changes to ensure the user knows the buttons have been pressed

Secondary Scenarios:

- The user cannot find the clearly displayed buttons and continues to play the game with music and sounds

Success Postcondition: The sound and music will stop while the game is playing as to not irritate the user if they have decided to turn them off; the game will continue without any implications

Minimal Postcondition: The sound and music will continue to play while the user is playing the game, which may irritate the user or entertain them

Use Case: Cook Food
Primary Actor: Cook (Player)
Supporting Actors: Customers (Non-playable character)
Precondition: Game has started
Trigger: The cook has received orders from customers
Main Success Scenario:

- The cook collects ingredients from the pantry
- The cook brings ingredients to designated workstations
- The cook cooks the food on time at designated workstations
- The cook assembles ingredients before serving food to customers
- The cook serves food to customers

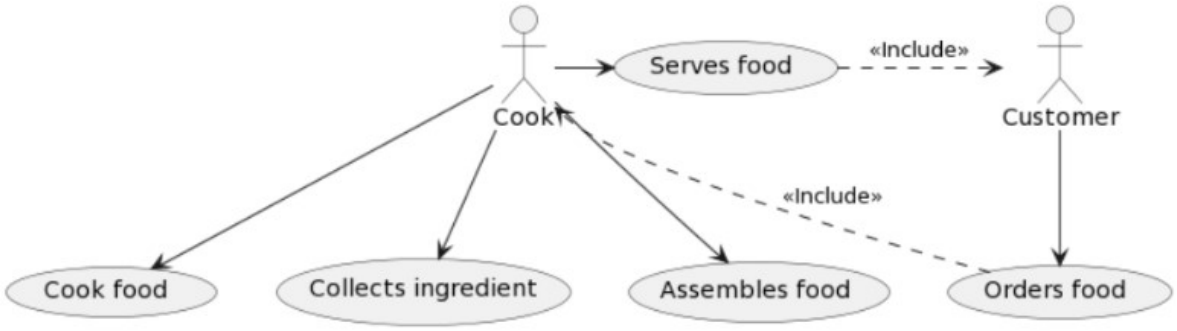
Secondary Scenarios:

- 1.1 The cook does not collect the ingredients before going to workstations
- 1.1.1 An instruction window is readily available for the player to view

- 1.2 The cook collects the wrong ingredients
- 1.2.1 The correct ingredients are clearly shown if the user clicks on the dish they're trying to make
- 2.1 The cook brings the ingredients to wrong workstations
- 2.1.1 The player won't be able to do anything with the ingredient on the station so will have to remove it
- 3.1 The cook takes too long to cook food
- 3.1.1 The cook has to redo the process from the start eg: collect ingredients, bring the ingredients to the workstation, etc.
- 4.1 The cook does not assemble the ingredients before serving food to customers
- 4.1.1 The food will not be served to the customer implying it's incorrect

Success Postcondition: The food served to the customer is the right order and the customer is happy and leaves the restaurant

Minimal Postcondition: The customer will not be served and will not leave the restaurant until the correct order of food is served



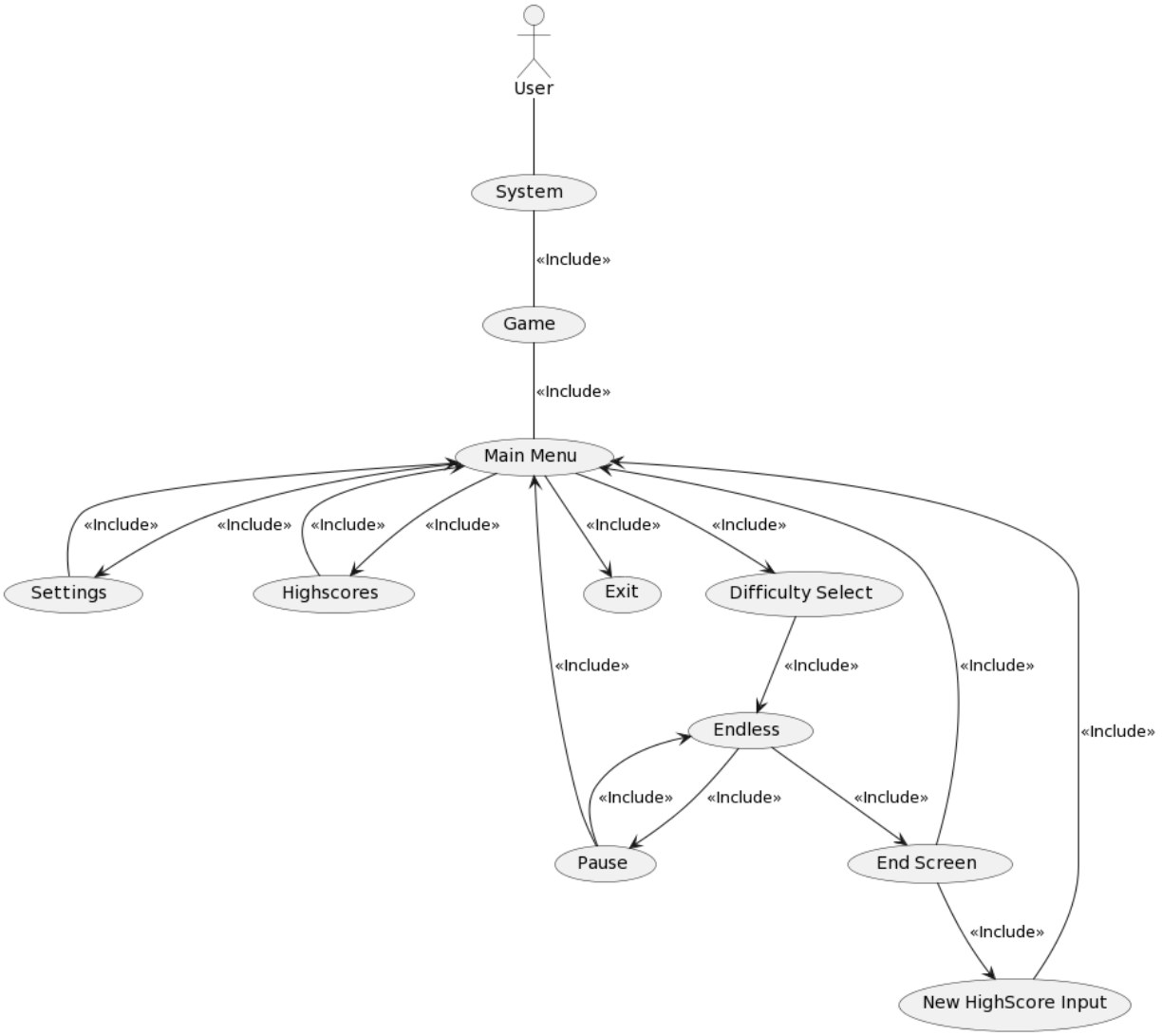
Use Case: Play Endless Game
Primary Actor: User/Player
Supporting Actors: N/A
Precondition: Player is ready to play the game and has ran it on the specified system; the system is displaying the main menu and they have no save files
Trigger: The player presses the start button which triggers the scenario to run and the game to play out

Main Success Scenario:
- Player selects difficulty
- Player begins the game
- Player plays the game and clears all customers
- The player is presented with an end screen that displays their score and asks whether they want their score to be saved on the leaderboard
- If the player decides to save their score in the leaderboard, the leaderboard will be updated and displayed to the user; otherwise, the player will be displayed the leaderboard compared to their score
- Player is redirected to the main menu of the game

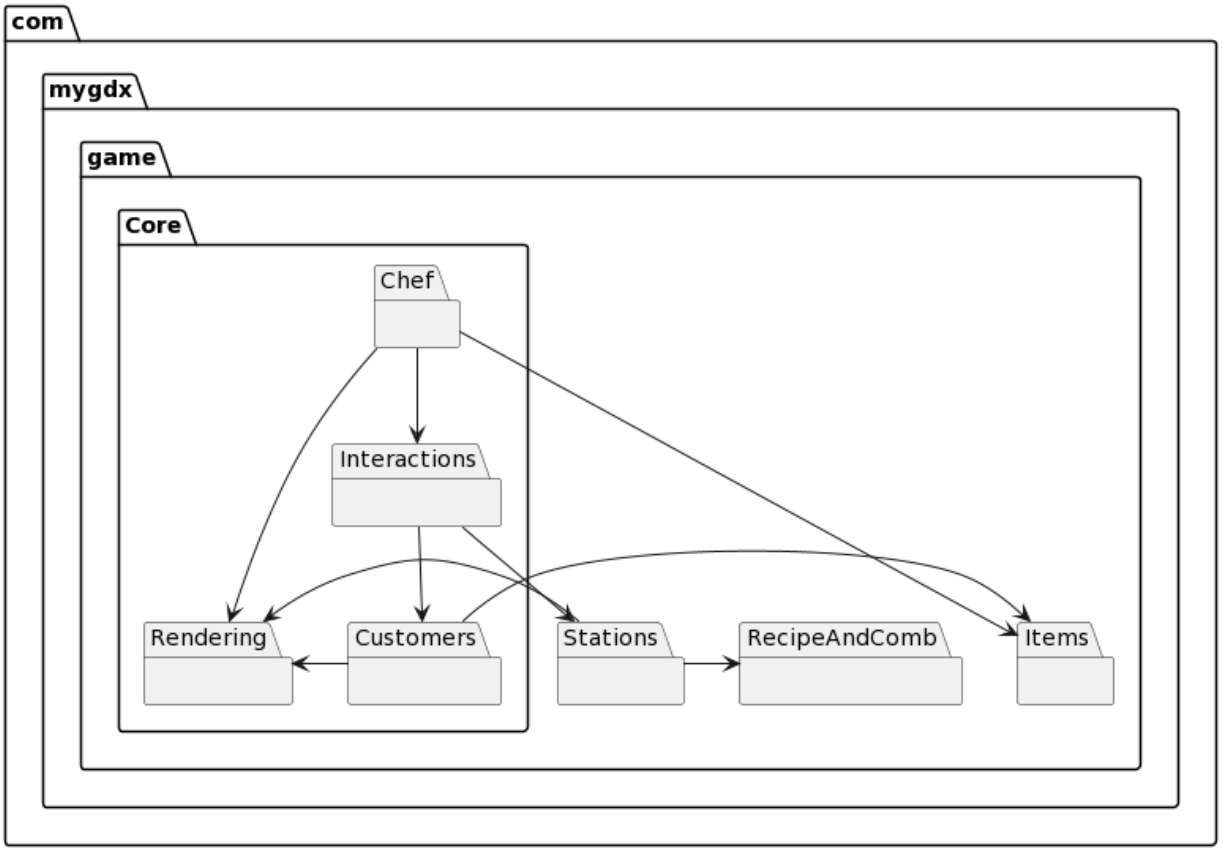
Secondary Scenarios:
- 1.1 The player does not know the instructions
- 1.1.1 The player can open the instructions menu to read how to play the game

- 2.1 The game is too hard for the player and they lose
- 2.1.1 The player is asked to input their score into the leaderboard
- 2.1.2 They can change the difficulty of the game if it not already at the lowest option when they return to the main menu and start a new game
- 3.1 Player no longer wishes to play
- 3.1.1 Player presses pause button and quits

Postcondition:
The player has lost all reputation points or surrendered and receives a score that qualifies for the leaderboard
They either decide to integrate their score into the leaderboard, at which point the leaderboard is updated and displayed to the user;
Or, the user decides not to integrate their score into the leaderboard and their score is displayed alongside the leaderboard
The player is redirected to the main menu



Package Diagrams



This is our first iteration of the redesign in package diagram format, it has been simplified and some newer packages have been omitted such as the new requirements. We attempted to create an updated package diagram upon project completion however due to the nature of a game based on ECS there are too many packages and classes to be able to keep it simple as requested in our previous assessment feedback.

Leaderboard file format
We save our leaderboard code in a bespoke language called fson. This follows the format of “<name,score> “. New values are appended to the end of the last element. We use regex to read the data into classes that represent the data in code. We use this language as the leaderboard didn't require a complicated storage type such as json, and didn't need to be compact such as storing as bytes in a file. Thus a regex solution fulfilled our needs.

Adjustments for Difficulty
We adjust the game to be more difficult statistically when the player sets a higher difficulty by making the chefs slower, the customer loses patience faster, changes the money gained and more. We also increase the cook speed of items to make sure the game is possible on harder difficulties. We prevent certain foods from being ordered in lower difficulties to make them easier. On harder difficulties you have less frustration points.

Customer Ordering
Customers use a pseudo random algorithm that keeps track of previous orders to ensure that the customer orders seem fairly random, but biassed towards having various orders.