Requirements:

Team 14

Joshua Beaven (jb2825@york.ac.uk)

Calum Feenan (cf1218@york.ac.uk)

Harry Hillman (hth512@york.ac.uk)

Jonathon Snelgrove (js3336@york.ac.uk)

Omar Alhosani (oaa560@york.ac.uk)

Rayan Butt (rb1770@york.ac.uk)

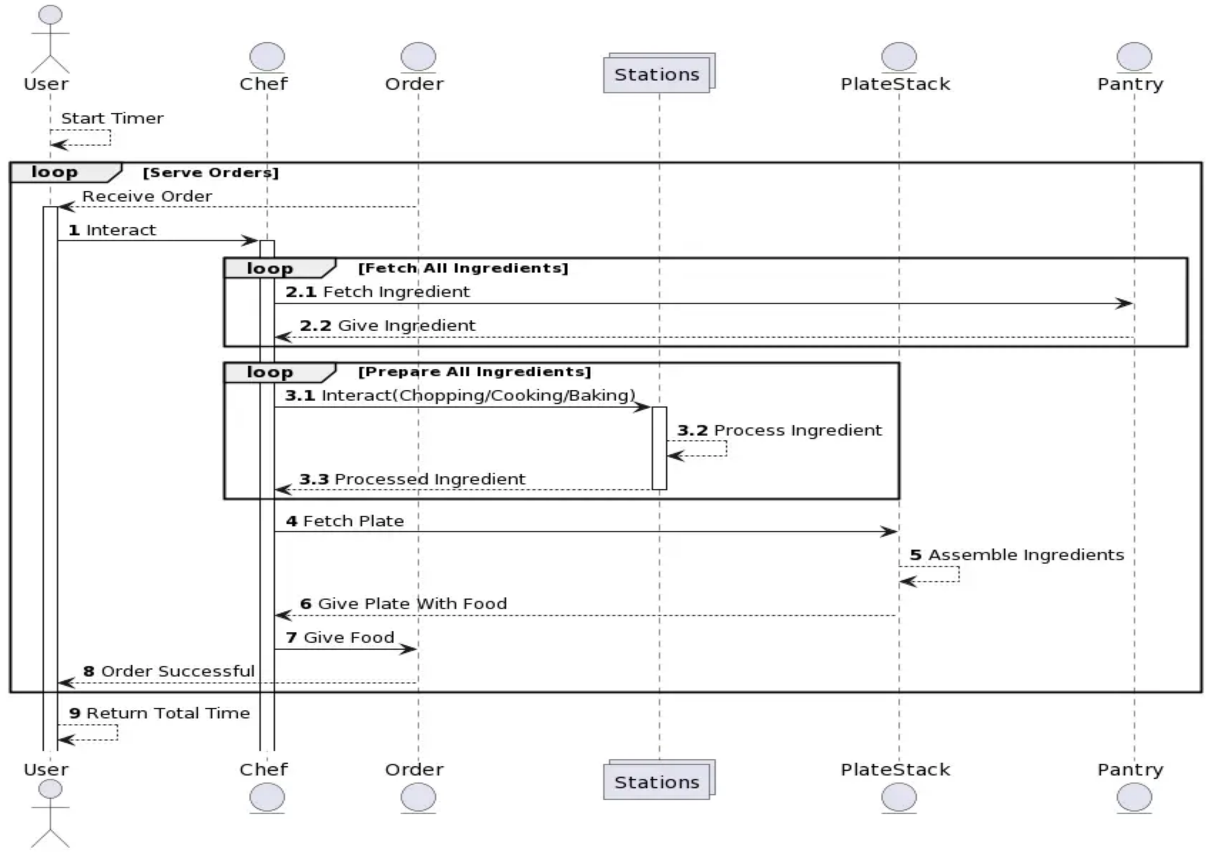
Part A: Structural and Behavioural Diagrams

Since taking an object-oriented system (OOS) approach, as the project is being coded in Java, the Unified Modelling Language (UML) was used for the architectural representation of the product as a language specialised to visually represent OOS. Including its flexible and comprehensive properties, UML has a base foundation for modelling the more basic parts of object-oriented software.

Both structural and behavioural architecture were made using PlantUML, an open-source tool that turns plain text into different types of UML diagrams. PlantUML is supported by different software development environments like IntelliJ and Visual Studio Code.

To visualise the software's flow and behaviour a sequence diagram was used to illustrate a series of sequential steps over time, representing the actual gameplay. Using sequence diagrams reduces confusion and provides a logical representation of the game's intended software.

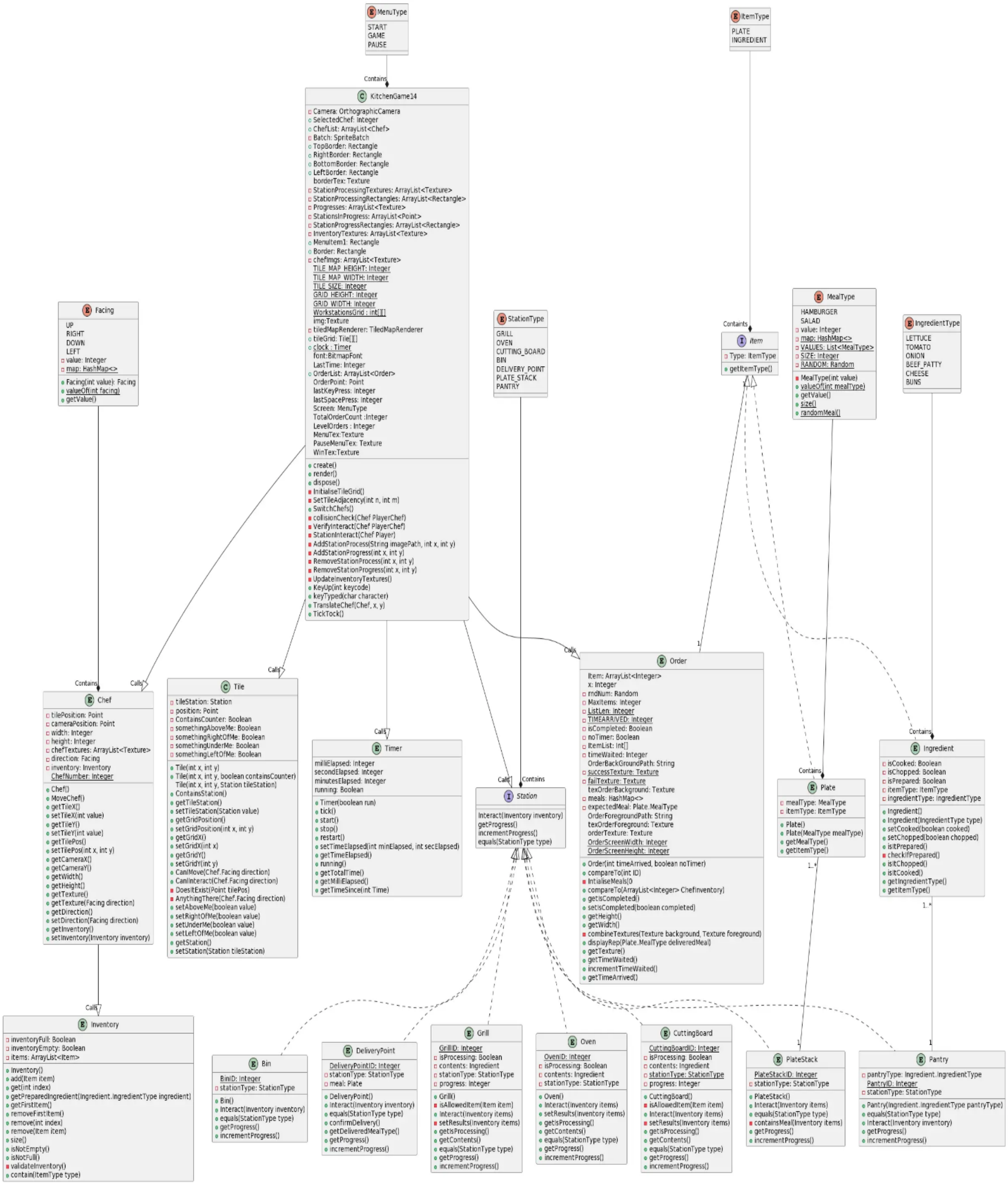
Behavioural Diagram



To create the structural representation of the software, a class diagram is used to expand on the software's elements and their relations. Class diagrams can visualise the logical structure of an OOS, defining its elements, classes, attributes, and the relationships among them.

PlantUML provides icons and a proper visual representation of the software, thus giving future teams a better understanding of the code and reducing the amount of time wasted in the process of transitioning to this project.

Structural Diagram



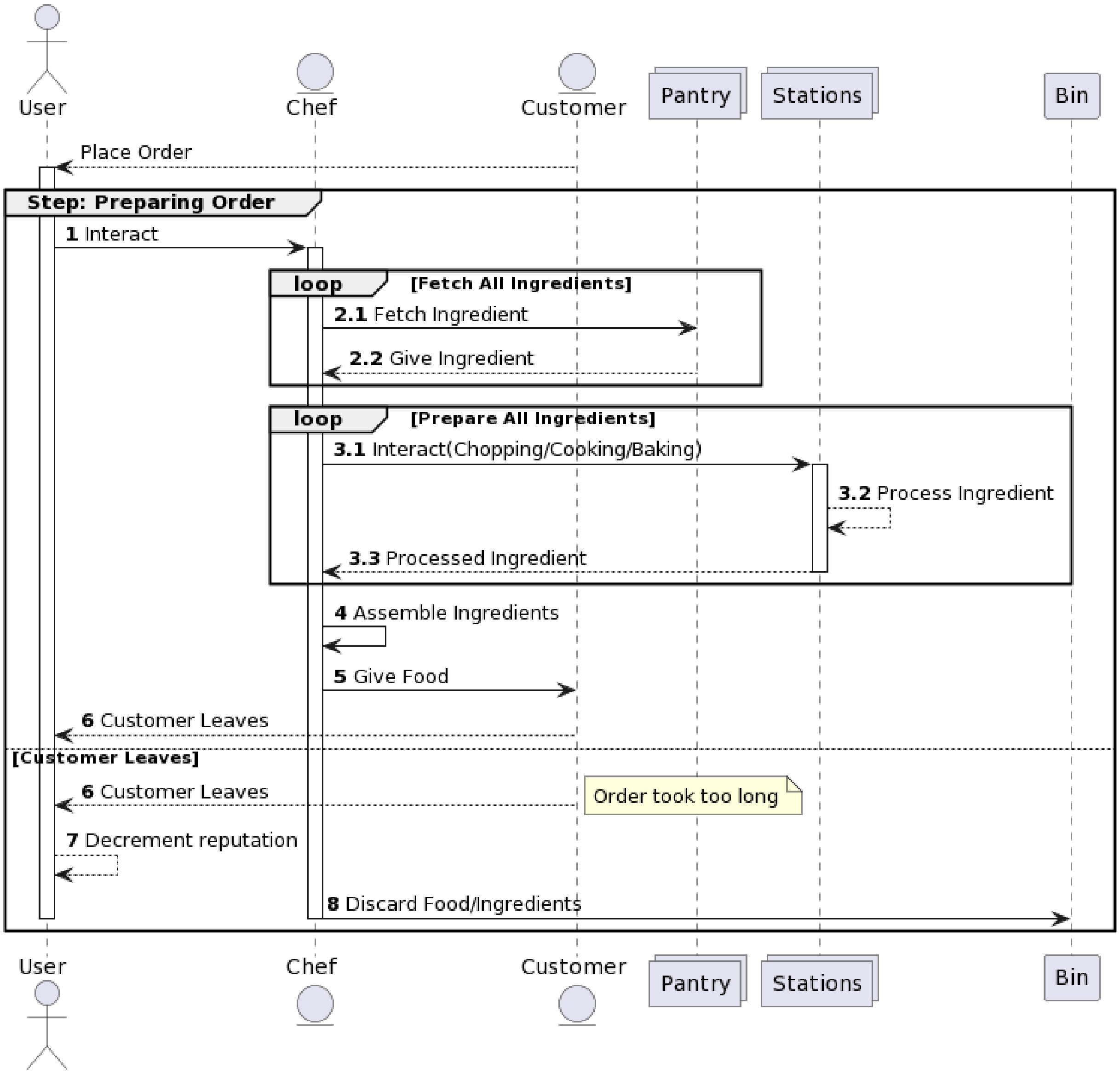
Part B: Systemic Justification of Architecture

During the development phase of the architecture, the behavioural and structural diagrams were developed based on the Requirements section. This was carried out by:

- Focusing on satisfying all user requirements first.
- Discarding planned architectural developments that would hinder progress.
- Discarding architecture that is out of bound of user requirements.

The first iteration of the behavioural diagram did not receive any drastic changes to its structure, however, elements like "PlateStack" were introduced to facilitate problems like the congestion of items in the chef's inventory or the method of serving the finished food items. The failure condition was removed as this version of the game does not have a failure condition, and the Step "Preparing Order" is replaced with the loop "Serve Orders".

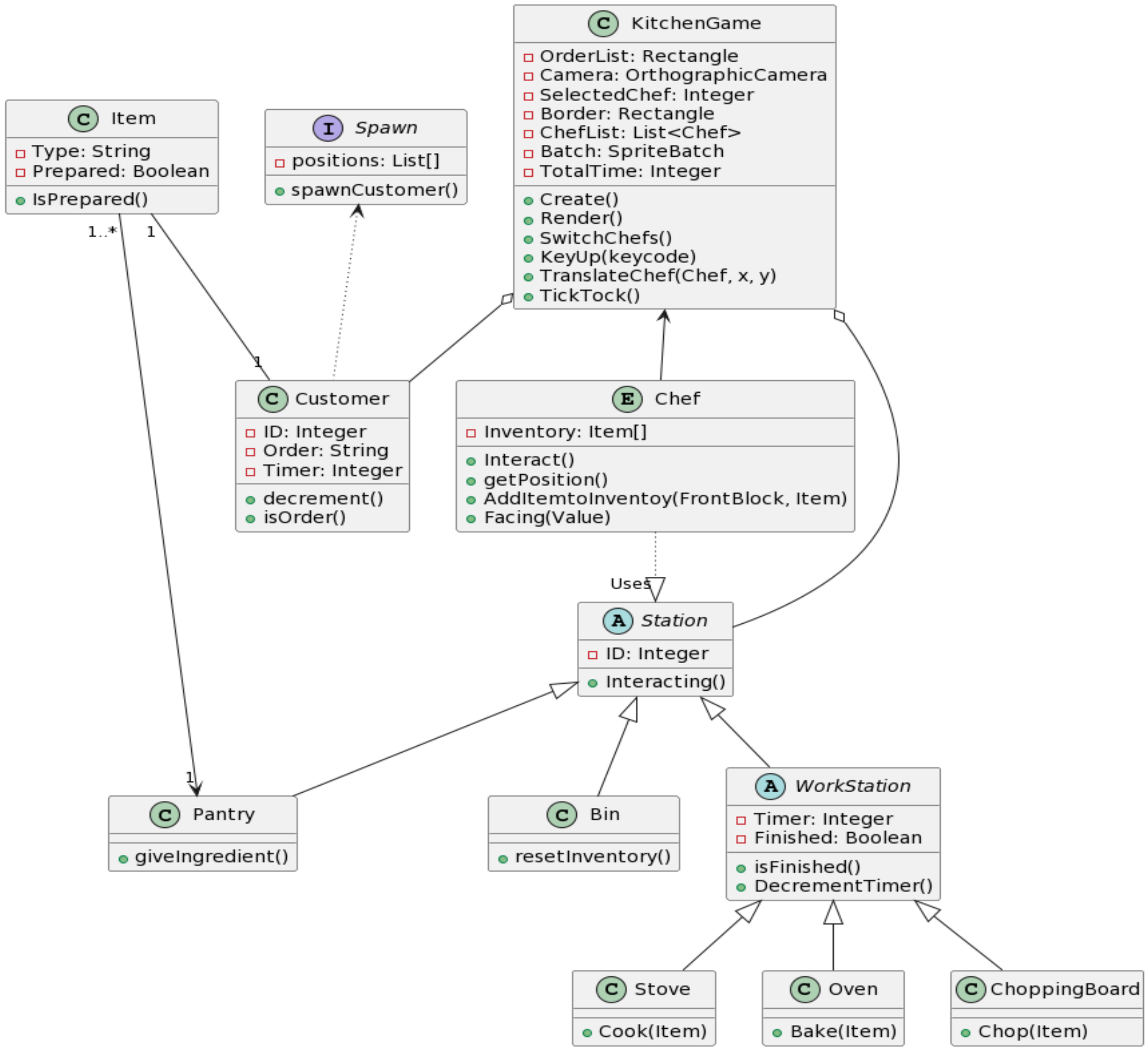
Behavioural early iteration



In earlier iterations of the structural diagram, elements such as “Customer” and “Spawn” were introduced to develop the basis of what the ordering and reputation systems were meant to become. However, in future iterations the idea that a “customer” must be created and introduced along the “spawn” interface was deemed unnecessary, causing them to be discarded for the “order” entity that provided a simpler and more direct approach.

Elements like the abstract classes “Workstation” and “Station” would eventually be replaced by an interface “Station” to facilitate future teams’ implementation of the monetary system which allows players to earn and spend money on workstations.

Structure early iteration



Java Class	Requirements	Relation
Bin	UR_ClearInvetory	As the chef throws what they're holding into the bin, they empty their inventory stack.
Chef	UR_CookMove UR_ChefControl	Class to describe the current chefs position, movement, inventory and interactions.
CuttingBoard	FR_Recipe	Class used to change an ingredient, such as "Cheese" to "Chopped Cheese" from the top of the Chefs inventory stack.
DeliveryPoint	UR_Counter, FR_CustomerCounter	Class used as a space for the order given and the order from the Chefs inventory to each other.
Grill	FR_Recipe	Child class of Station to change an ingredient.
Ingredient	UR_Recipe, UR_Pantry, UR_CookPantry, FR_GettingIngrediants	Parent class for each ingredient used in a recipe.
Inventory	UR_ClearInventory, UR_Pantry, UR_ClearInventory, FR_GettingIngrediants	Parent Class to represent Ingredient/Plate and what the chef is carrying.
Item	FR_GettingIngrediants	A parent class to Inventory.
KitchenGame14	UR_Gamedisplay NFR_FrameRate NFR_NoCrashing	The main class - used for creating and rendering objects.
Order	UR_CustomerOrder	Class which randomly generates an order from a customer, to be completed by the player/chefs.
Oven	UR_FlipPatties	A child class of Station, this can be used to cook an ingredient.
Pantry	UR_Pantry, FR_GettingIngrediants	Child class of Station. Used as a place to collect an ingredient from.
Plate	UR_Plate	Used to show that an order is completed and being carried by the chef.
PlateStack	UR_Plate	A Class which converts the chef inventory - which contains the valid and necessary ingredients of a recipe - to a single item of a "Plate" which has the order on it.
Station	UR_Interact	A Parent class for Pantry, Oven, Grill.
Tile	UR_CookMove UR_Contollers	A class used to represent a section in the "game grid". Each element is a tile, whether it be a Countertop, Floor a tile which has a function such as Pantry.
Timer	UR_PlaygameTime, FR_CustomerTimeWaiting	A class used to represent time using the amount of frames which have passed. Also used throughout the program to add time delays to cooking processes, to add an elapsed time timer.