3. Architecture

Part a)

Figure 1: An overview of the Classes involved in the system architecture

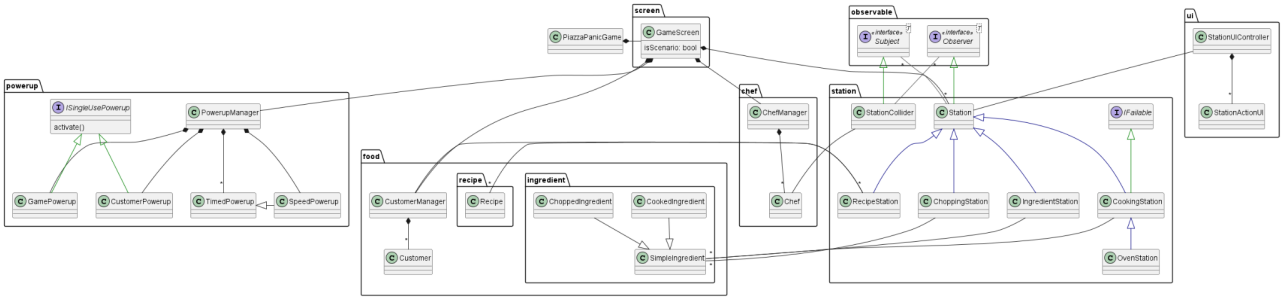


Figure 2: A detailed look at the methods and attributes for GameScreen, Station, ChefManager, CustomerManager and Chef

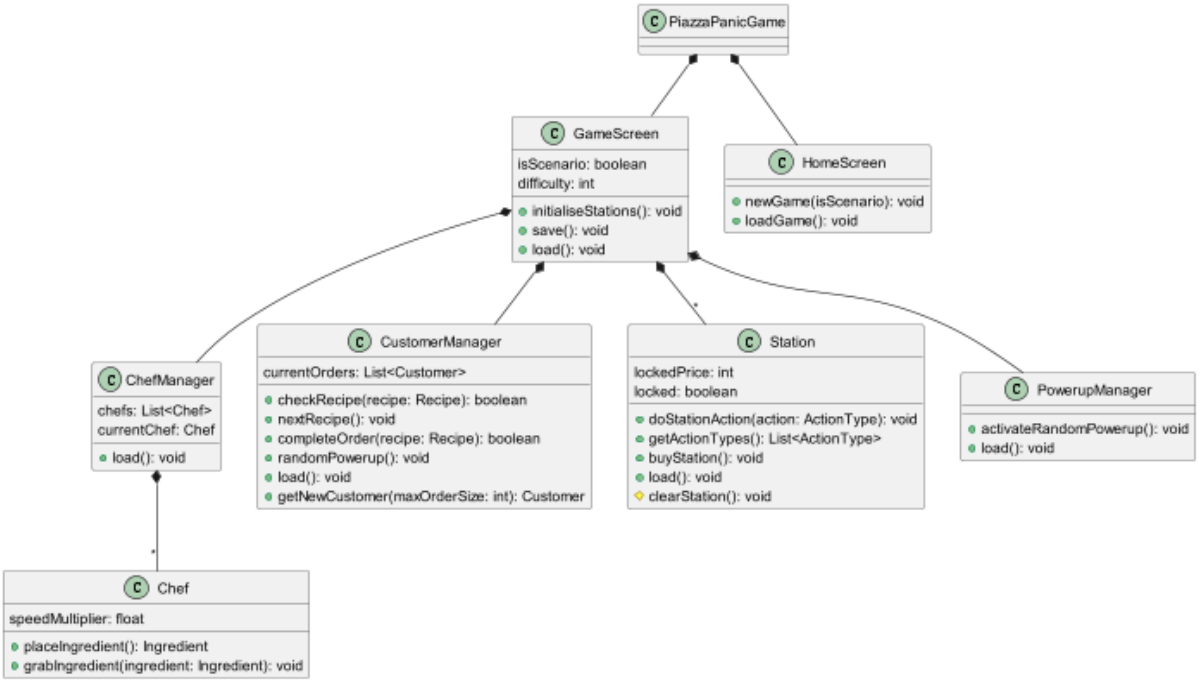


Figure 3: A detailed look at the methods and attributes for ChoppingStation, CookingStation, IngredientStation, RecipeStation, Recipe, Station and Ingredient.

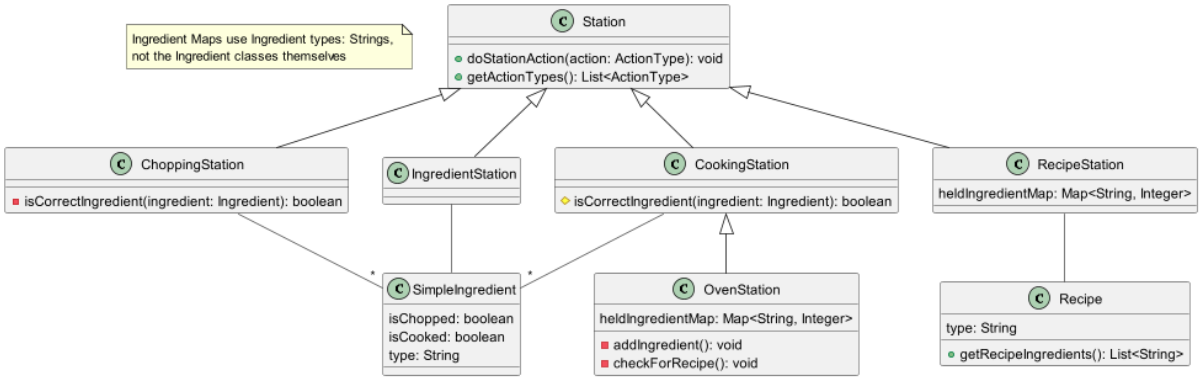


Figure 4: A detailed look at the methods and attributes for Station, GameScreen, UIOverlay, StationUIController, StationsActionUI and Timer

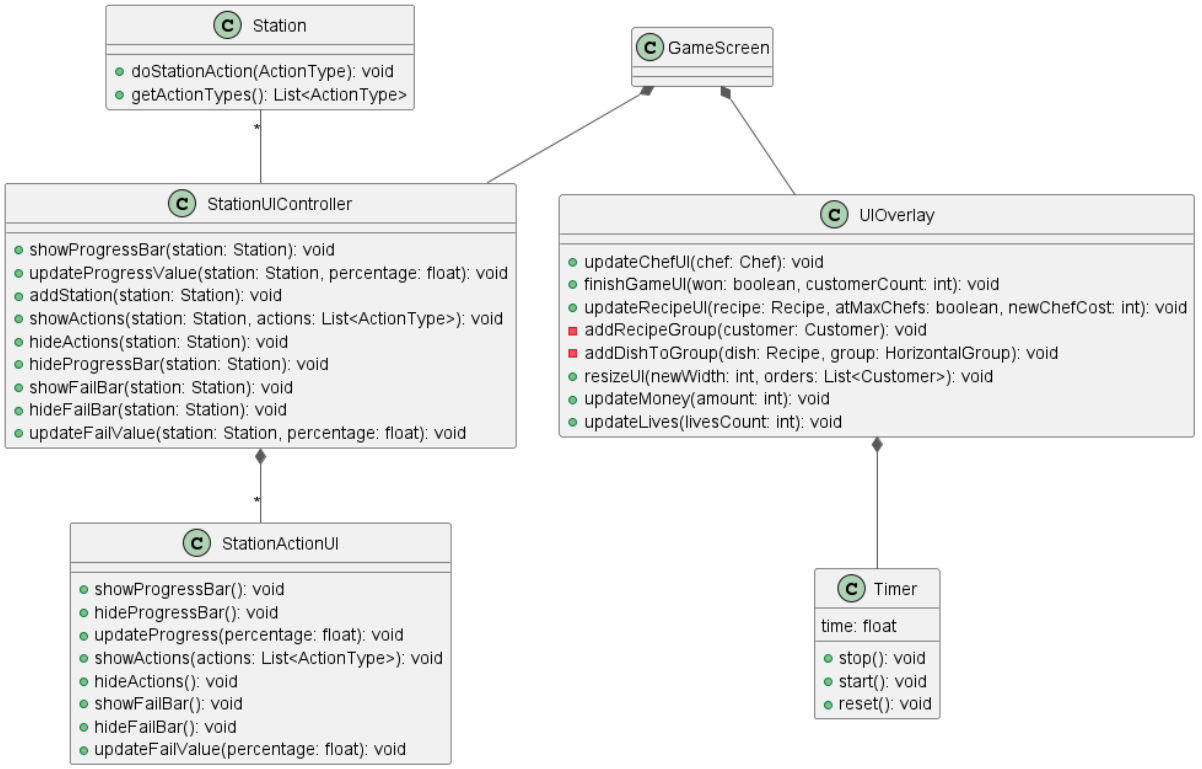


Figure 5: A state diagram for controlling the chefs

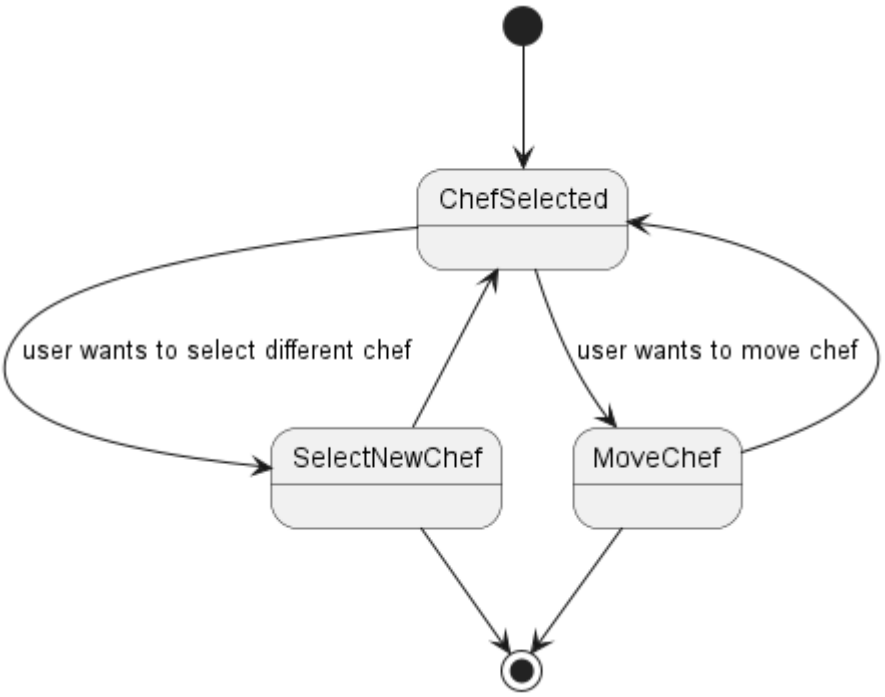


Figure 6: A state diagram for the overall control of the game

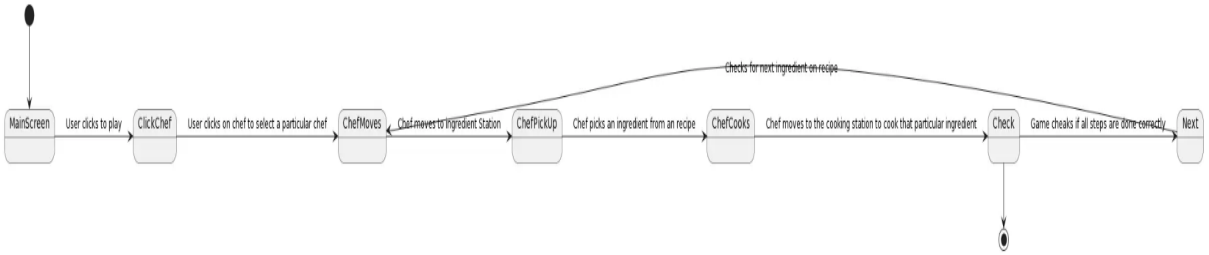


Figure 7: A state diagram for navigating through the screens

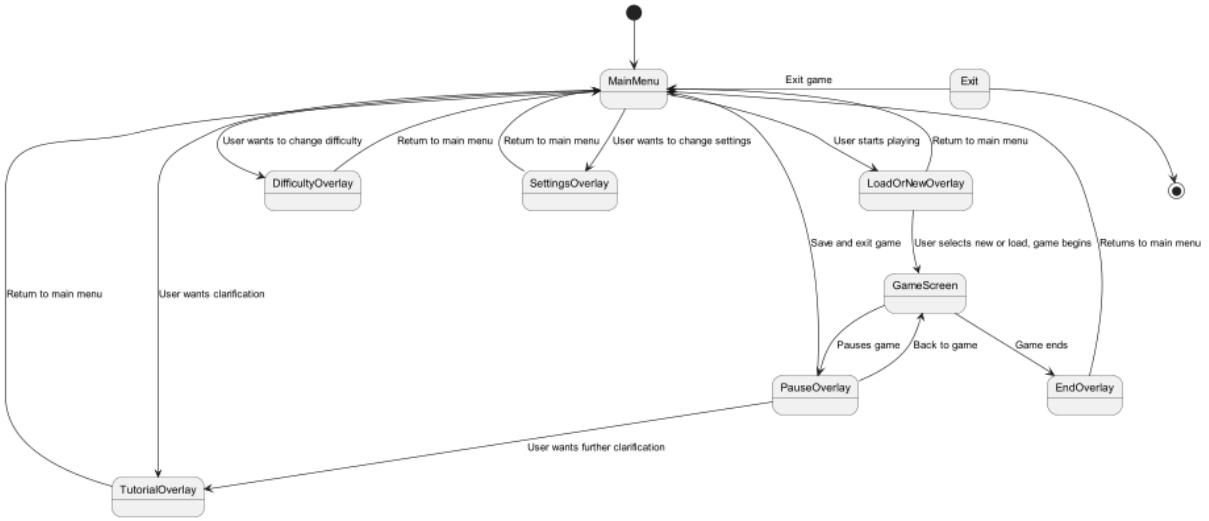


Figure 8: A sequence diagram showing the completed process for the user to create a burger

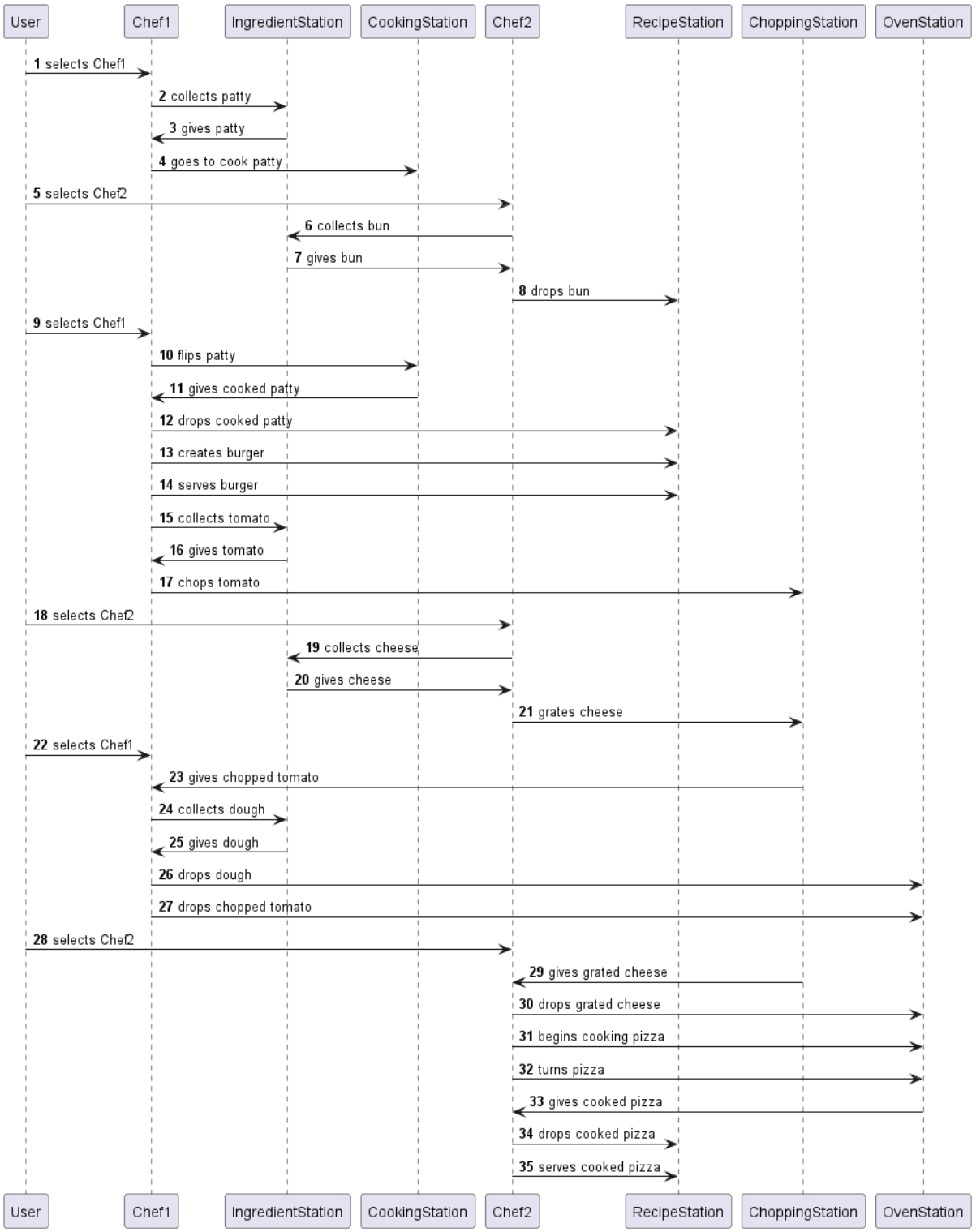
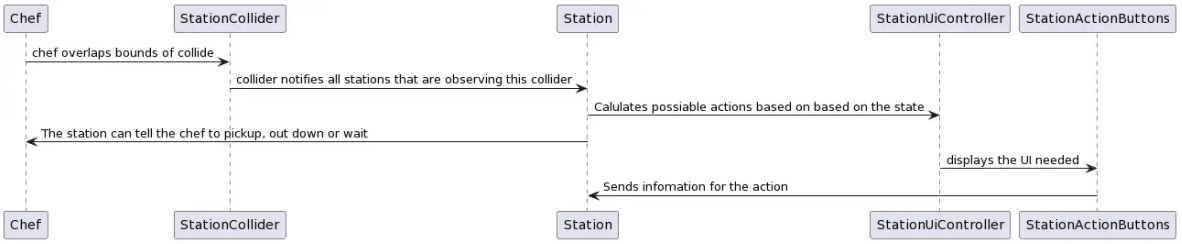

User
Chef1
IngredientStation
CookingStation
Chef2
RecipeStation
ChoppingStation
OvenStation
1 selects Chef1
2 collects patty
3 gives patty
4 goes to cook patty
5 selects Chef2
6 collects bun
7 gives bun
8 drops bun
9 selects Chef1
10 flips patty
11 gives cooked patty
12 drops cooked patty
13 creates burger
14 serves burger
15 collects tomato
16 gives tomato
17 chops tomato
18 selects Chef2
19 collects cheese
20 gives cheese
21 grates cheese
22 selects Chef1
23 gives chopped tomato
24 collects dough
25 gives dough
26 drops dough
27 drops chopped tomato
28 selects Chef2
29 gives grated cheese
30 drops grated cheese
31 begins cooking pizza
32 turns pizza
33 gives cooked pizza
34 drops cooked pizza
35 serves cooked pizza
User
Chef1
IngredientStation
CookingStation
Chef2
RecipeStation
ChoppingStation
OvenStation

Figure 9: The sequence diagram of using chefs and how they interact with the station collider



The tools used to create the diagrams to represent the architecture was PlantUML. We used PlantUML for its easy to learn and understandable syntax. PlantUML has support for making a variety of diagrams such as UML, State and sequence diagrams. PlantUML was useful to produce these diagrams in different formats that are simple to understand like PNG and Pdf files. We used class diagrams as it is most suited to represent an object oriented programming language such as Java.

Part b)

Architecture is a vital part of designing any program, as it provides a structure of how to code is going to work.

The diagrams we decided to continue working on from the ones inherited included all class diagrams - representing an overview of the structure of the code and specific classes needed, aside from the Observer/Subject interface diagram as we would not be making any changes to it, as well as all behavioural diagrams - to show various flows of the program, aside from the chef controls as they would also remain the same.

The diagrams already inherited covered most of the requirements we would be keeping, hence this document will mostly explain how the changed architecture covers new requirements. In their final class diagram, the previous unique Station classes allowed for FR_FLIP_AND_CHOP, with the IngredientStation covering FR_GRAB_ITEMS. The FR_TIMER was covered by the Timer class, implemented in the UIOverlay. The CustomerManager and customerServed() - now completeOrder() - methods covered UR_CUSTOMERS and UR_SERVE_FOOD. FR_PLACE_ITEMS was fulfilled by the IngredientStation class.

The previous state diagram for controlling a chef already covered FR_CHANGE_PLAYABLE_CHARACTER and FR_MOVE_PLAYABLE_CHARACTER. Their initial sequence diagram for burgers already fulfilled FR_SERVE_CUSTOMER, UR_SERVE_FOOD and UR_COOK_FOOD

After eliciting new requirements, we turned our focus into changing architecture to prepare for implementation. Firstly, we examined the code we had been given in an attempt to correct any disparities between the previous architecture and actual code. For example, we added packages to the overall class diagram and updated the class names in our final cooking sequence diagram.

Then, we mapped out the new requirements and any changed old ones, and planned how we could implement them into the already existing system, attempting to follow similar design principles if we had to add any new architecture. For example, we used a PowerupManager to handle all Powerups classes centrally.

After initial designs and changes, we went away to further analyse the code and begin implementing. Later, we updated our architecture in relation to any issues we may have come across or any new discoveries we had made relating to the code already in place. e.g., in our 2nd class diagram we realised that the GameScreen class used a stage to manage most game logic each render, so having multiple new GameScreen classes would prove inefficient.

To improve on UR_UX, we wanted to ensure the game was playable properly in all sizes comfortable. Particularly in the UI classes, we introduced new resize methods to ensure all new and old UI elements would scale properly and that the game was playable on both bigger and smaller displays.

With bigger order sizes and more than one order now being possible in the game due to the changes in CustomerManager in order to meet the FR_CUSTOMER_ENDLESS requirement, we also had to rework the order UI. Previously, the UI would show 1 order with its entire recipe process on the screen. If we maintained this, the screen would become very overcrowded very quickly. So we introduced the addDishToGroup and addRecipeGroup methods in UIOverlay which would display just the dishes ordered, grouped neatly down the side of the screen. To account for this change, and ensure that we were still meeting the UR_INSTRUCTIONS and NFR_DOCUMENTATION requirement, we updated the tutorial to show recipes along with game instructions.

To meet requirements of NFR_OPERATABILITY and NFR_DIFFICULTY, we implemented difficulty options into the DifficultyOverlay as shown in the menus state diagram. This is also represented in the GameScreen's new difficulty field.

To implement a new powerup system and meet the UR_POWERUPS and FR_POWERUP_SPAWN_ENDLESS requirements, we added the PowerupManager class and an initial abstract Powerup class. After some time implementing, we realised most powerups would have unique logic and there was not much in a base class. However, all powerups could benefit from some shared functions such as activate() function, which we implemented using the ISingleUsePowerup interface. As well as this, the invulnerability and chef speed powerups we planned to add could both use the TimedPowerup class, implementing a simple timer and also extending libgdx's Actor class, allowing it to tick the timer each act() method call. These changes are shown in the 2nd class diagram.

To account for new stations in FR_NEW_STATIONS_ENDLESS, we introduced the new OvenStation class, and decided other recipe aspects could be implemented with already existing classes such as IngredientStation for new pantry boxes of new Ingredients.

To meet the new fail steps for recipes (FR_RUIN_FOOD_ENDLESS), we added the IFailable interface for CookingStation and OvenStation classes, which also added functionality for a fail bar to render on the station.

We also had to make architectural changes affecting the original game to account for the new gamemode. For example in the CustomerManager class, orders were now using our new Customer class and not the old Recipe class. To still account for FR_CUSTOMER_SCENARIO, we had to ensure that a Customer class's order field (List<Recipe>) was only ever of size 1 in scenario mode, but it allowed us to easily scale having grouped orders in order to meet endless mode requirements – FR_CUSTOMER_ENDLESS in this example.

To cover UR_REPUTATION, we had to introduce the reputation field as seen in the GameScreen class. This reputation is in turn affected by the CustomerManager's tick

method, which returns an integer to represent how many reputation points should be lost from that call.

For UR_MONEY, we introduced a money field in the GameScreen class.

For UR_EXPAND and FR_BUY_STATIONS_ENDLESS, we added the locked fields into stations, as well as the buy methods.

To implement saving and loading into the system, also covering FR_SAVE_CHANGES, we implemented save and load methods into the game screen, as well as loading methods for all of the manager and station classes.

Finally, for FR_COOK_FOOD_ENDLESS, we had to create the new recipes and ingredients for them. When doing this, we realised how obsolete their class system was for Ingredients and Recipes. Some like Bun just called super in their constructor with a string. To solve this, we made all Recipes instances of the Recipe class, and reworked the Ingredient class into SimpleIngredient, the Tomato and Lettuce classes into the ChoppedIngredient class and the Patty class into CookedIngredient. This allowed us to easily add our new ingredients: Dough and Potato as SimpleIngredient, beans, raw pizza and raw jacket potato as CookedIngredient, and cheese as a ChoppedIngredient.