4. Architecture

Part a)

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

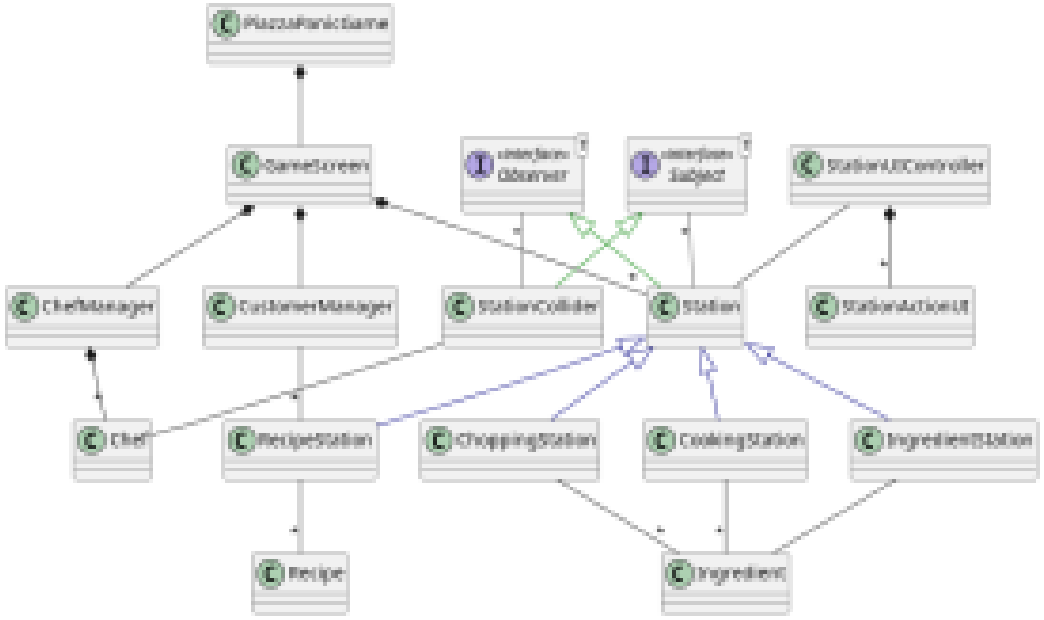


Figure 2: A detailed look at the methods and attributes for GameScreen, Station, ChefManager, CustomerManager and Chef

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

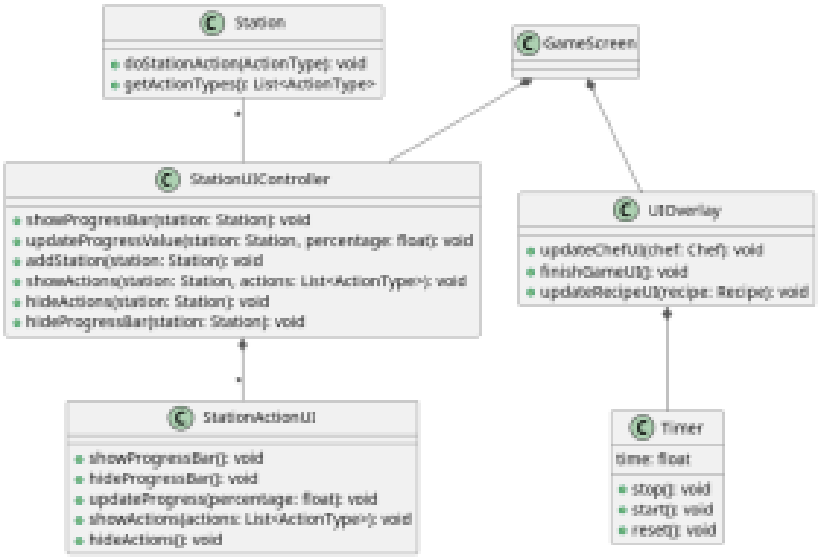


Figure 5: A state diagram for controlling the chefs

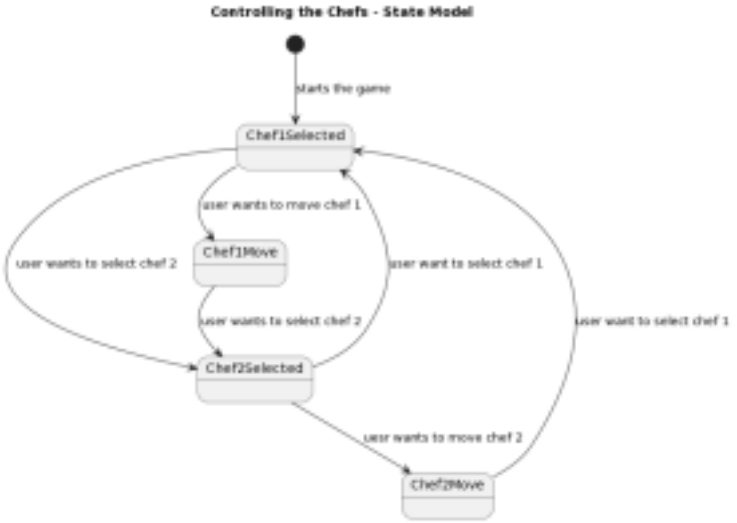


Figure 6: A state diagram for the overall control of the game

Figure 7: A state diagram for navigating through the screens

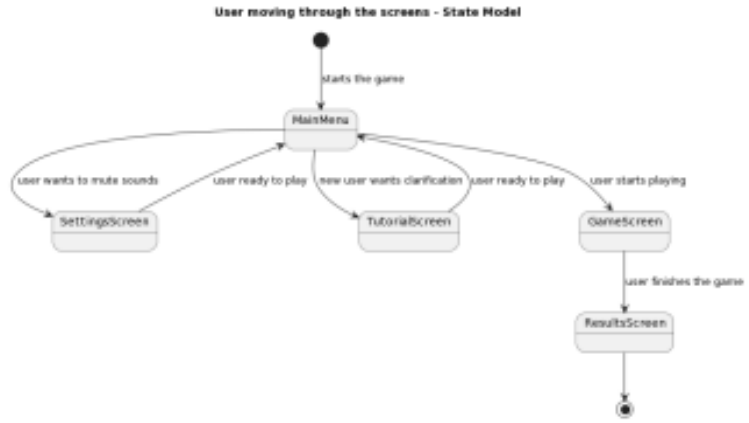


Figure 8: A sequence diagram showing the completed process for the user to create a burger

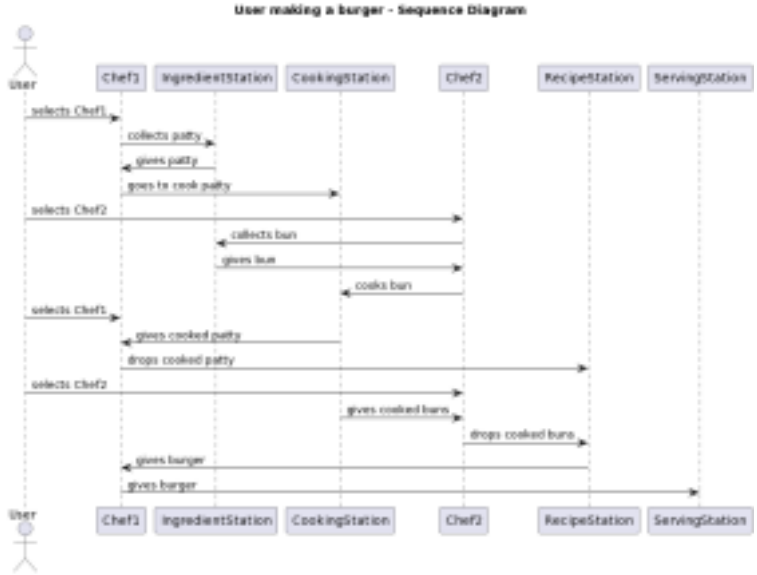
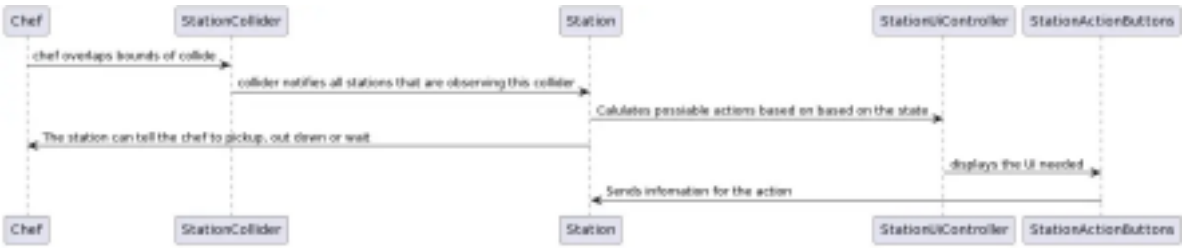


Figure 9: The sequence diagram of using chefs and how they interact with the station collider



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.

Phase 2

Going into phase 2, there are not many major changes to the structure of the game though there are some additions made to it. The tool used to create the diagrammatic representations is PlantUML. It is the tool that we have previously used for Assessment 1, so we are already familiar with it. Another alternative that we could use for the representation is LucidChart. However, we only used PlantUML this time as it is more convenient to keep the consistency of the diagrams instead of having to ensure that the diagrams created with different tools are similar and consistent. We used class diagrams for the class representation because it was more efficient to write the code to create them than to draw them manually in some other applications. We also used sequence diagrams for the behavioural structure to show how the user moves through the screens and what information is passed between items. Another example is how to prepare orders that are represented by the sequence diagrams below. We created interim versions of the class diagrams & sequence diagrams on an application on iPad called Notability to help our understanding, you can see how the classes have evolved over the course of development.

Structural Diagrams

Figure 2.1 shows a detailed look at the methods and attributes for ChoppingStation, CookingStation, IngredientStation, RecipeStation, Recipe, Station and Ingredient with new classes added to Recipe.

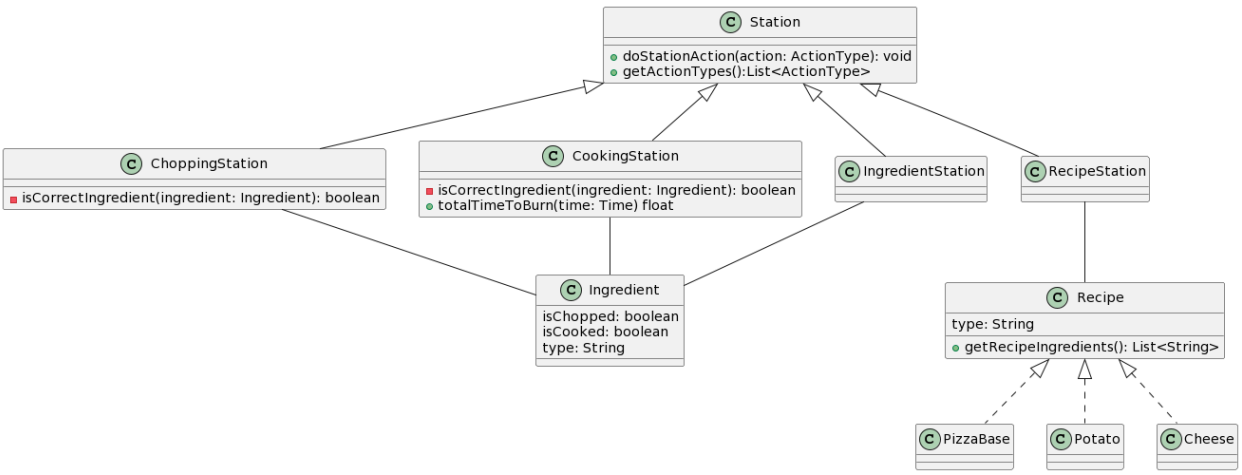


Figure 2.1

Figure 2.2 below shows a detailed look at the methods and attributes for Station, GameScreen, UIOverlay, StationUIController, StationsActionUI and Timer with the newly updated instances in UIOverlay.

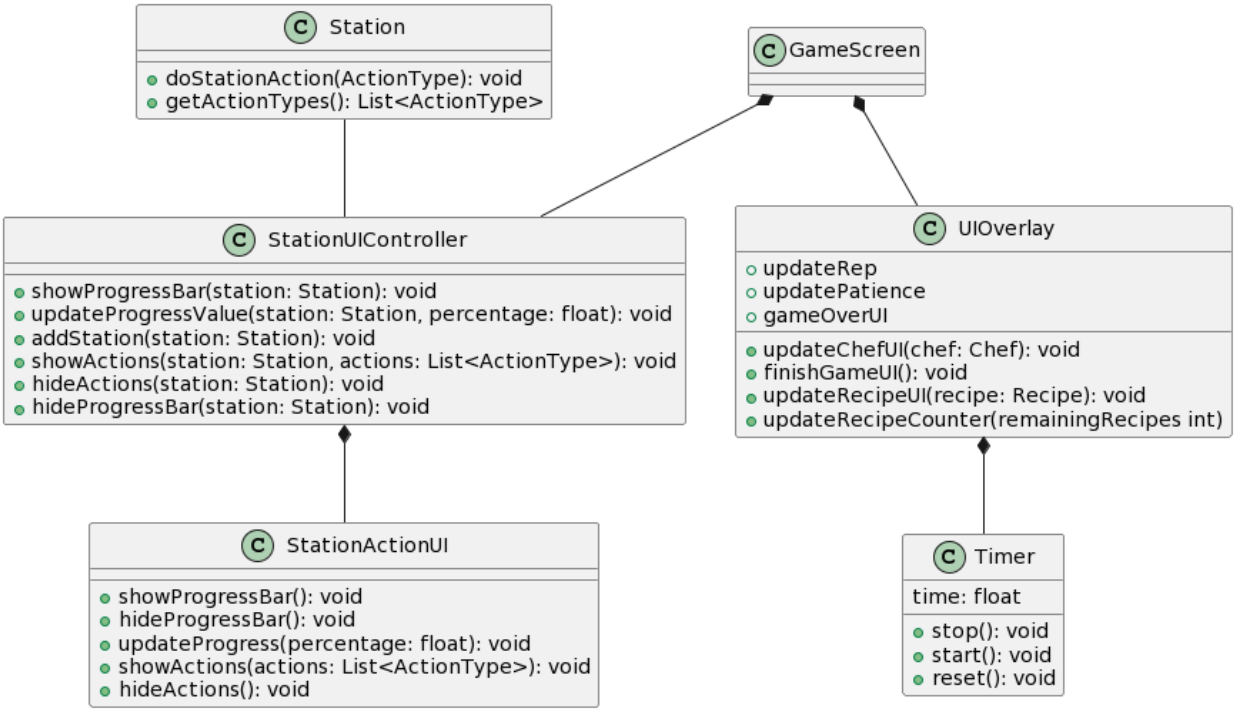


Figure 2.2

Figure 2.3 shows the new class that we created called LongBoiBank for the addition of the money system.

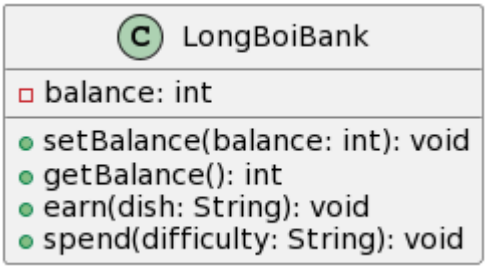


Figure 2.3

Behavioural Diagram

The sequence diagrams below show how to prepare a burger, pizza, jacket potato and salad to ensure each of the items is ready to be served to customers.

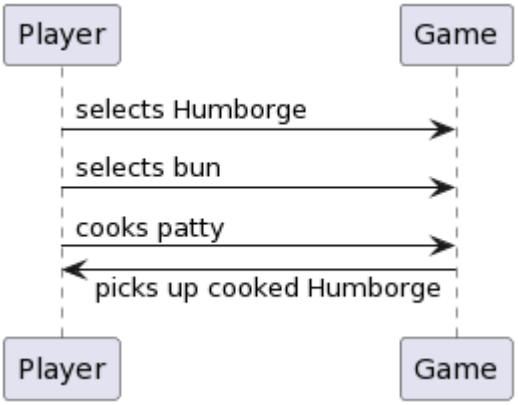


Figure 2.4

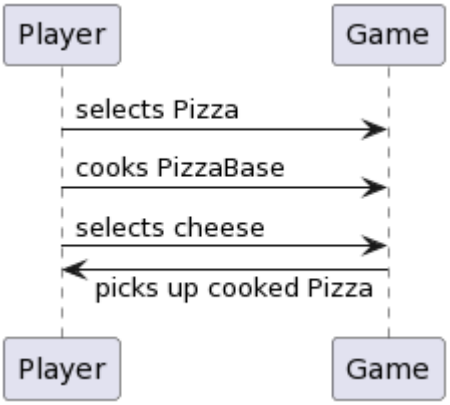


Figure 2.5

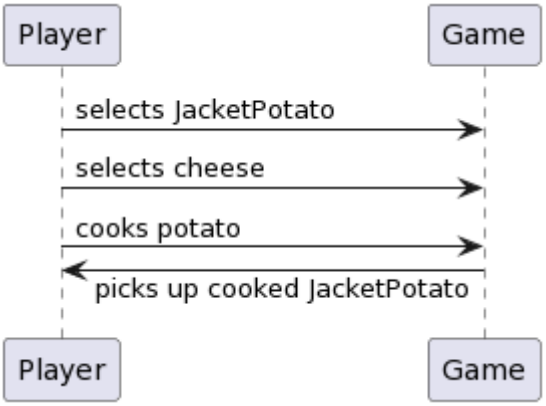


Figure 2.6

Part b)

Architecture is a vital part of designing any program, as it provides a structure of how to code is going to work. Initially, we used the Responsibility Driven Design (RDD) strategy to aid the team to generate the main classes/features/screens that we need to implement. We chose this over Domain-Driven Design since Java is our main programming language, and RDD is best used for object-oriented languages. It allowed us to look at the system as a whole and not get caught up in the specifics just yet.

Before we get ahead of ourselves, it is important to ensure that we are considering the requirements of the project. Relating back to the requirements centres the group ensuring that we remain on task. As a result during the RDD process we ensured that the requirements were in front of us to refer back to them.To begin with, using Jamboard, we noted ideas and principles that we believed to be useful. You can see the result here. Once all the ideas were noted, as a group we removed objects that were similar or not important from the group's perspective.

We each worked on some of the objects and generated CRC cards (seen here). These demonstrated which objects need to interact with each other and the connections between the objects. As a result, it is easier to generate initial UML diagrams with the following CRC diagrams. The RDD process was incredibly useful for the group as it allowed us to think clearer about the intricacies of the project

From completing online research [1], there are various different UML diagrams we can generate, all of them splitting into two categories, structural and behavioural. From there we need to decide which diagrams will better display the overall idea of the program. The diagrams we decided to create were Class diagrams as they provide an overview of the structure of the code and the specific classes we will need in our code. For the behavioural diagrams, we decided that we should complete two types of diagrams since games are user experience based and therefore we should look at the detail of how the game is going to behave.

Initially, the Class diagram was designed like this (found under first draft). It was clear early on that all the stations had similar attributes but different actions, as a result we designed the architecture such that there would be a stations class in which specific stations (cutting, cooking, serving and so on) would inherit from. The difference from all the stations being the action they complete. This description of the Class diagram represents the requirement FR_FLIP_AND_CHOP. Further the requirement FR_TIMER has been demonstrated in this diagram through the inclusion of the timer attribute in the GameState which records the time elapsed playing the game. It is best to include this in the GameState since it controls when the user decides to pause, start and end. Therefore it is easier to code to identify when the timer needs to start and stop. Moreover, the requirement FR_GRAB_ITEMS is satisfied by the inclusion of the Ingredient station where it allows the chef (aka the user) to pick up ingredients for the recipe they are making.

Upon reflection we were missing a lot of information and the Class diagram needed more detail thus resulting in this diagram (found under Second Draft section, titled “Piazza Panic v2 – Class Diagram”). Here there are classes such as the customer that were missing from the original. Also during the discussion of the generation of this diagram, we wanted to clear some terminology up for consistency e.g. Chef instead of Cook, Chopping instead of Cutting and so on. This is to ensure that there is no confusion later on in the project. The new diagram satisfies the requirement UR_CUSTOMER and FR_SERVE_CUSTOMER by including the customer class and the customerServed() method, along with the order

variable. This is a very important part of the project that was neglected in the original Class diagram.
Additionally, FR_PLACE_ITEMS requirement is fulfilled by the addition of the counter stations that will allow the user to place items on the top of the chef stack onto the counter. This station was not included in the product brief however we felt that it was necessary to include this into the project as it gives the user more freedom as they might want to cook the second ingredient in the stack and cannot access it. The counter station mitigates this problem.

Further to this, we include a bin station where the user can “bin” the top ingredient of the chef's stack, this achieves the requirement FR_REMOVE_ITEMS. Again this station is not mentioned in the product brief, but without the inclusion of this feature, the player might accidentally pick up ingredients and then have nowhere to get rid of it.

Other diagrams were created on top of the class diagram to represent the behaviour of the game and specifically how the user will interact with it. Thus, research was conducted into displaying sequence and state diagrams correctly [2] [3]. This is an important part of the design to demonstrate as there are several ways to implement a game, like will the chef be controlled by the WASD method or will they be controlled using the arrows on the keyboard or using the mouse and so on. Thus we need to clearly outline how the user interacts with the software.

In figure 6 (also found here under overall diagram) it demonstrates an overview of the system as a whole in a simplistic view. This view is important as it depicts the general use of the system that the user will go through. However, it is vital to consider some specifics of the system like how the chefs are controlled.

This is shown in the state diagram in figure 5 which demonstrates the user controlling a chef. In our game, we decided that the user will use the mouse to select the chef that they would like to control and then use the arrow keys on the keyboard to control the movement. This simple state diagram demonstrates the FR_CHANGE_PLAYABLE_CHARACTER and FR_MOVE_PLAYABLE_CHARACTER requirements.

Additionally, the state diagram in figure 7 illustrates the requirements FR_MUTE_SFX and FR_GUIDE_USER, since it allows the user to move in between screens that allow the performance of both requirements. This is quite a simplistic view of the system as it only looks at the way that the user navigates the system and not how the user will interact with the game contents, therefore an additional behavioural diagram needs to be generated.

Next we created a sequence diagram that shows how a user will make a burger and interact with the stations and the different chefs. This is shown in here (the very last diagram). This diagram is a step by step of how the user could make a burger in full, where the ingredients include: cooked bun and cooked patty. It is important to note that this is only one way it could be done, for example the user could cook the bun first, before cooking the patty. As a whole the diagram represents FR_SERVE_CUSTOMER, UR_SERVE_FOOD and UR_COOK_FOOD.

Generating this diagram, made the group realise that we have not accounted for combining the ingredients together to create the final product. Thus we discussed including a recipe station for that exact purpose, where the chef drops off the ingredient and then once all the ingredients are present it will output a completed recipe. As a result of this the Class diagram needed updating to include the missing station. This new version can be seen here under Second Draft titled “Piazza Panic v2.1 – Class Diagram”.

The prior Class diagrams were interim designs and were never intended to be the official diagrams, since they lack some detail and are not in the correct format. Therefore, research [4] was conducted to ensure that the diagrams were technically correct. Furthermore, looking more closely at the programming language and LibGDX we notice there were some things we were missing from the originals that allow for collisions and the movement of the chefs. Thus more Class diagrams were created, these are figure 1, figure 2, figure 3 and figure 4. They can als be found here under the heading Final Draft.

There is one class diagram that solely represents the structure of the classes, missing the details of the methods and variables that we need. This diagram is in figure 1. The reason that we included this diagram is to show how each part of the code interacts with each other, showing any inheritance and dependencies. However we would need smaller in depth architecture that looks at the methods.

Looking at the second iteration we decided that there were too many stations and that violated the NFR_OPERATABILITY requirement and it needed to be simplified. As a result, we decided that instead of the bin station, we would include the action to remove the top item of the stack in the ingredient station. This meant that the FR_REMOVE_ITEMS remained satisfied as well as not violating the NFR_OPERATABILITY requirement. Further to this we discovered that the inclusion of the recipe station meant that there was no longer a need for a counter station since finished items can be placed in the recipe station instead. This is shown in figure 3.

Additionally, when removing stations we had to update previous diagrams such as the sequence diagram here (the last diagram on the page). We also decided to lessen the steps in order to make a burger to maintain NFR_OPERATABILITY. The new sequence diagram is seen in figure 8, which still satisfies the requirements FR_SERVE_CUSTOMER, UR_SERVE_FOOD and UR_COOK_FOOD.

When thinking about how the chef is going to complete an action (e.g. cooking, baking and so on) we decided that when the chef collides with a station then a list of actions they can complete will appear. This is demonstrated in figure 4, where the Station deals with the ingredients and chef. There is one StationActionUI per Station which has the buttons and progress. The StationUIController allows stations to find the corresponding StationActionUI. This diagram fully represents the requirement FR_FLIP_AND_CHOP since it includes exactly how that will be achieved in the code. Also it satisfies the requirement FR_NOT_OVERCOOKING such that the progress does not result in any overcooking of sorts.

Finally, in figure 2 it describes how the chef is going to interact with the game and therefore the stations. This describes the requirement UR_CONTROL_CHEFS. This is because of the inclusion of the chef manager which deals with when the user selects a different chef. This is further represented by the state diagram in figure 5.

During the coding it is important to ensure that every part of the architecture is carefully included into the project and that no code clashes with each other. Therefore it is important to start with a plan, where someone sets up the environment and then others can add on their code when needed. This was demonstrated when AF committed the initial commit to generate LibGDX game and then to follow up he created a scaffolding of the code, for all the classes that we had in the Class diagram.

The next step is to add the constructors to the relevant classes. This was demonstrated by

the commit, made by MF. Parallel to that, the tilemap can be rendered that allows objects to move around the map. From here, the functionality for each part can be coded simultaneously, provided that no coding tasks clash with each other.

Phase 2:

Going into phase two, we aimed to implement the small features first, ones which would only require minor changes to the architecture. For example, we implemented the burning mechanic as described in FR_OVERCOOKING, by creating a new class BurningUI and altering the already existing CookingStation class with a new value totalTimeToBurn and the associated logic.

Another minor change we made was to correct the UR_CONTROL_CHEFS requirement, which had changed from 2 to 3 when the requirements were updated early in the second phase. This was simple to implement with some minor modifications to the ChefManager class. Since the previous team's architecture accounted for the expansion of the game, adding a third chef was as simple as adding the new texture into an array. This change required little change to the architecture, it remains the same as was first illustrated in figure 5

A slightly larger, but still trivial change was to introduce the new recipes, as described in the altered UR_COOK_FOOD and UR_SERVE_FOOD requirements. We created three new classes, PizzaBase, Potato and Cheese which are needed to create the new recipes. Creating the new recipes themselves was done using two new classes Pizza and JacketPotato, which both extend the Recipe class implemented by the previous team. All logic was therefore already in place, making this a quick and simple change. Again, this required little change to the underlying architecture - only a few new simple classes - and as such figure 3 remains largely unchanged. However, behaviour for these new recipes can be seen in our newly created behavioural diagrams, figure 2.5 and figure 2.6

One of the first major new features we implemented was the addition of the money system, visible with the LongBoiBank class you can see in our new diagrams. This fulfils the new UR_CURRENCY requirement and corresponding functionality. The diagram figure 2.3 illustrates this new class. It is largely standalone.

The requirement FR_INVEST_EARNINGS was implemented by building upon the previous team's architecture surrounding stations and the StationAction class, in conjunction with the new currency feature we had added. At the beginning of the game, we altered some of the stations to be unavailable, and created the new action Unlock, available when the user walks up to the station. The StationAction class is abstract and very easy to use in order to implement this feature. As discussed in more detail later, the cost to unlock stations is varied by the selected difficulty, relating to UR_DIFFICULTY. Additionally, we made it so that the 3rd chef must be unlocked, providing the user with another incentive to earn coins while still fulfilling UR_CONTROL_CHEFS.

Given the high quality architecture of the original game, many new features could be added by altering preexisting classes. For example, we were able to implement the new endless mode feature, as per the requirement UR_MODES, by editing the UIOverlay.java file. All relevant methods were edited to take into account the boolean value isEndless, which alters

aspects of game play. We edited the HomeScreen class in order to allow the user a choice between endless (FR_ENDLESS_MODE) and scenario mode (FR_SCENARIO_MODE).

Similarly, we implemented the new difficulty modes in this way. We took the design decision to implement a new feature outside of the brief called Reputation, which is similar to a score system: serving customers increases reputation, and taking too long with their orders causes reputation to go down. If it hits 0, the game is over in endless mode. Reputation and other factors such as food prices and time to burn are affected when the difficulty level is changed. Our changes across the entire codebase fulfil the UR_DIFFICULTY requirement. The choice between difficulty levels is given to users within the difficultyOverlay, created in the HomeScreen class which is instantiated when the game is first played.

To fulfil UR_SAVE_GAME, which is linked closely to FR_SAVE_GAME_STATE, we had to significantly refactor the code. The brief states that the user shall be able to save “at any point”, in the game and as such we had to introduce saving features for both scenario mode (FR_SCENARIO_MODE) and the newly introduced endless mode (FR_ENDLESS_MODE). To implement saving, we first created a new class Save, which uses some JSON code in order to save details of the game's current state to a text file. Now, a user is presented with the opportunity to load or start a new game when they open the game. If the load option is selected, then the previous game state is restored. If not, then the usual process occurs: the user selects their mode and difficulty and a new instance of the game is created.

The final main feature of the game that we have added is power-ups. This fulfils the requirement UR_POWERUPS. We implemented this feature by creating a new class PowerupStation, which extends Station. One benefit of this is that it improves test coverage, as we have already been able to test the underlying methods from the Station class, giving us more confidence in the correctness of the PowerupStation. We decided that powerup stations should dispense random power ups at a given interval, which varies by difficulty. This interval fulfils FR_POWERUP_COOLDOWN. The powerup station creates a new StationAction ,GET_POWERUP, which is used to give the user its benefit, this fulfils FR_POWERUP_ACTIVATION. The requirement FR_POWERUP_ABILITIES states that “The system shall provide 5 powerups with distinct abilities”. We have met this requirement by implementing the 5 random power ups as: increasing the chef speed, reducing the time to cook, increasing the time to burn, decreasing the time to chop, and earning a set number of coins. The benefits gained from the first 4 power ups are all time limited, and the effect wears off after a given interval, further meeting FR_POWERUP_COOLDOWN.

Additionally, we have made some other minor tweaks to the architecture over the course of this phase of the assessment. However, they do not relate to any specific requirements and are just “quality of life” improvements, usually only a line or two. We have also made some edits to the code to refactor it to make it more suitable for testing. Relating to testing, one of the largest changes you will notice is the addition of our “test” folder, where all of our unit tests are located.