Architecture

Group 26

devCharles

Ross Holmes
Sabid Hossain
Thomas Pauline
Joel Paxman
Andrey Samoilov
Louis Warren

Part a)

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

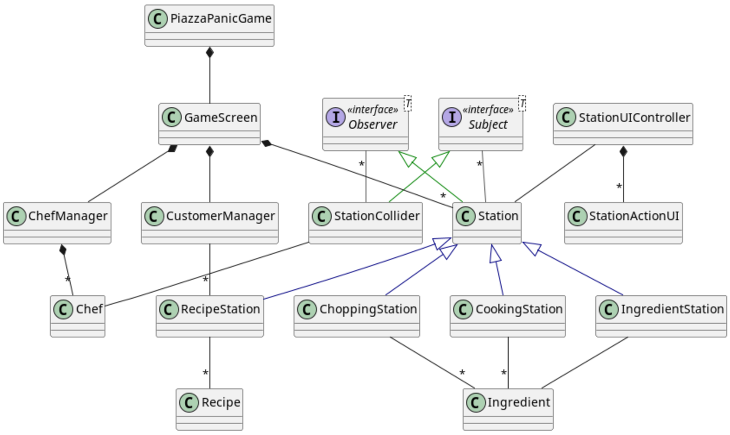


Figure 2: A detailed look at the methods and attributes for GameScreen, Station, ChefManager, CustomerManager and Chef

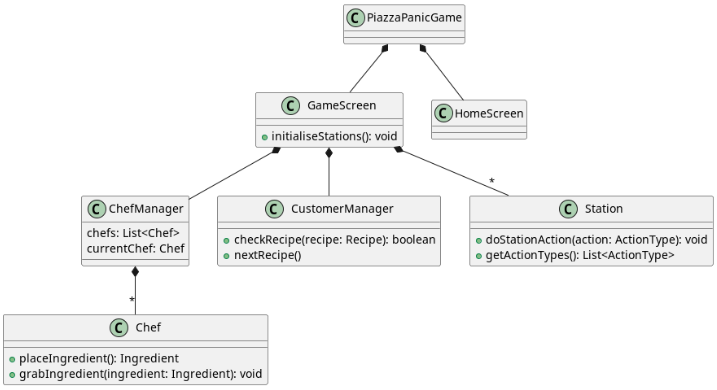


Figure 3: A detailed look at the methods and attributes for ChoppingStation, CookingStation, IngredientStation, RecipeStation, Recipe, Station and Ingredient.

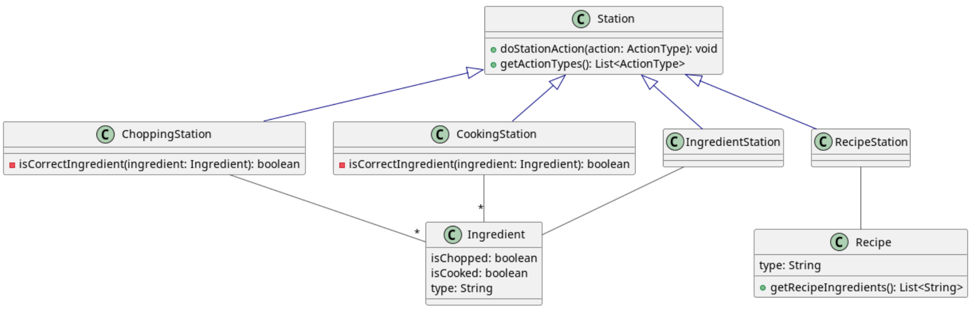


Figure 4: A detailed look at the methods and attributes for Station, GameScreen, UIOverlay, StationUIController, StationsActionUI and Timer

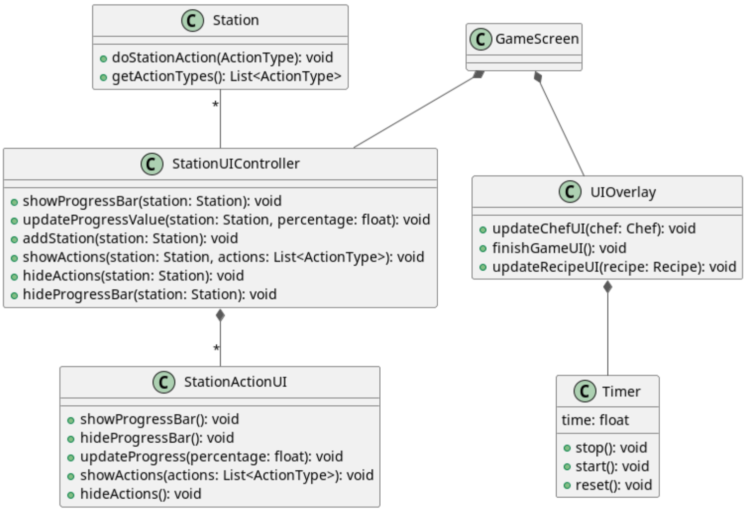


Figure 5: A state diagram for controlling the chefs

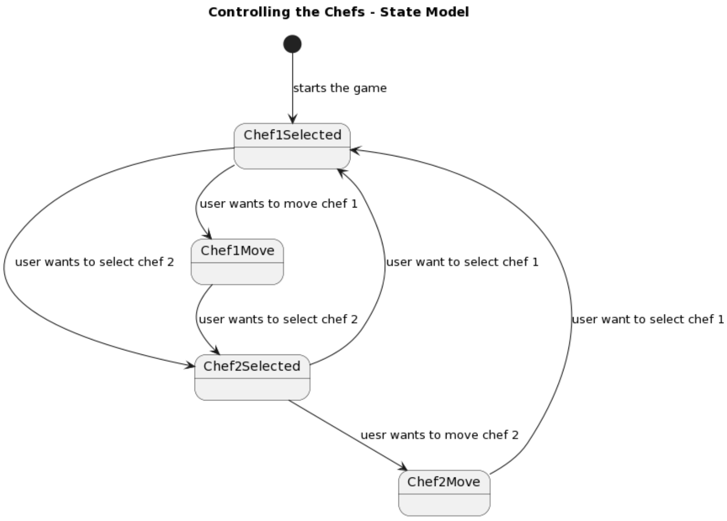


Figure 6: A state diagram for the overall control of the game

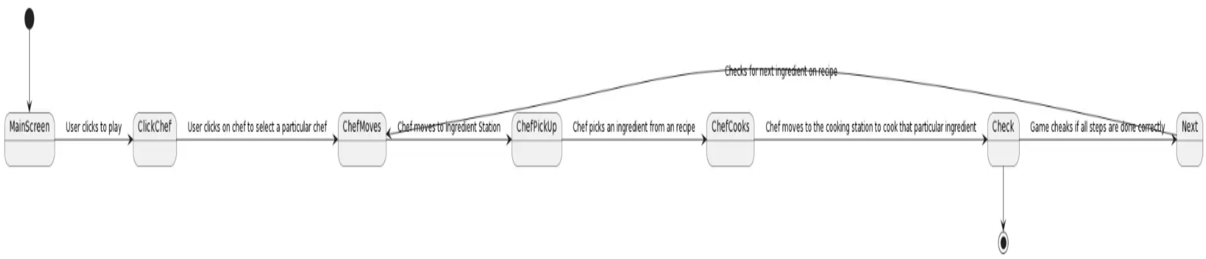


Figure 7: A state diagram for navigating through the screens

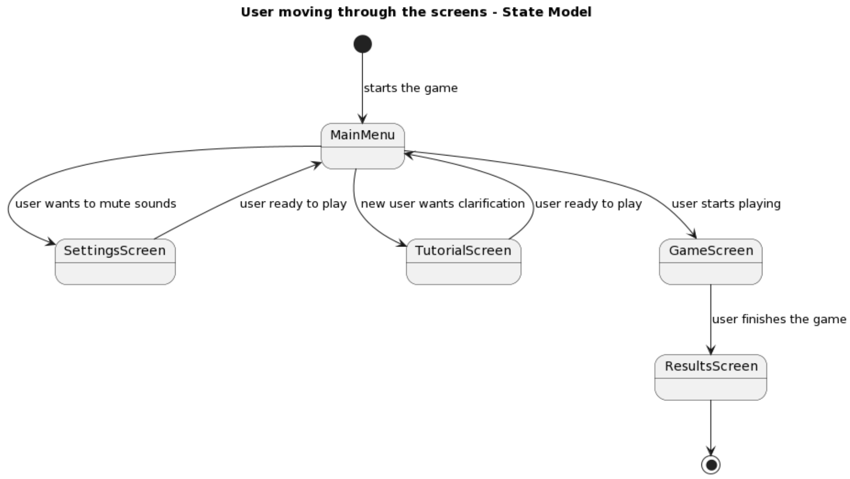


Figure 8: A sequence diagram showing the completed process for the user to create a burger

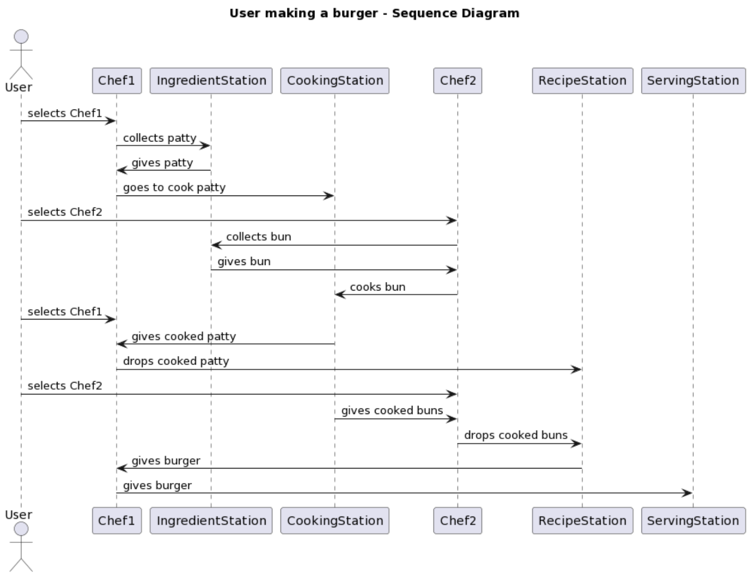


Figure 9: The sequence diagram of using chefs and how they interact with the station collider

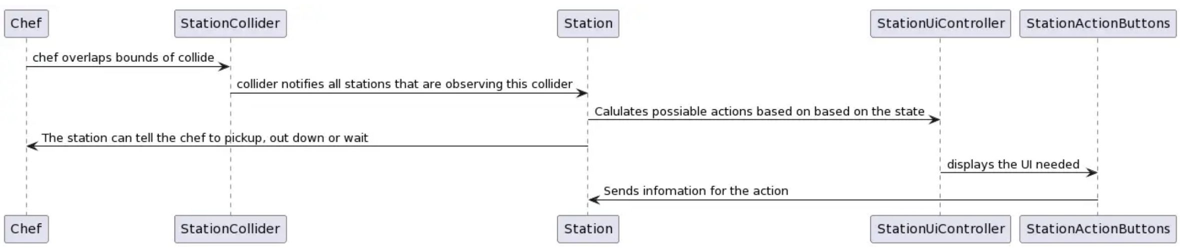


Figure 10: A diagram representing the new structure of the customer architecture, CustomerManager and Customer comprise the customer package within our code.

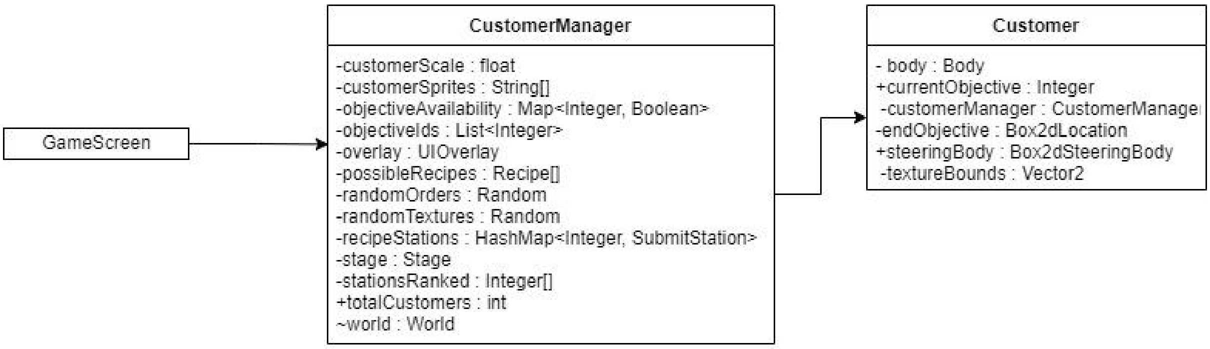


Figure 11: A diagram representing the classes related to saving, these functions make up the utility.saving package within our code.

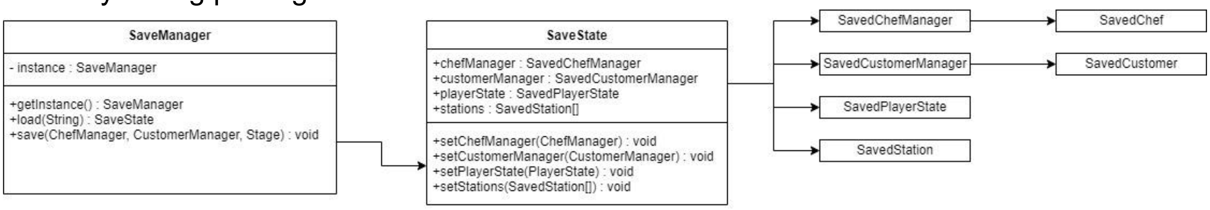


Figure 12: A sequence diagram represents our saving method, which creates saveable instances of the relevant classes, which hold the required values for saving. (E.g. a SavedStation stores the food that Station might hold).

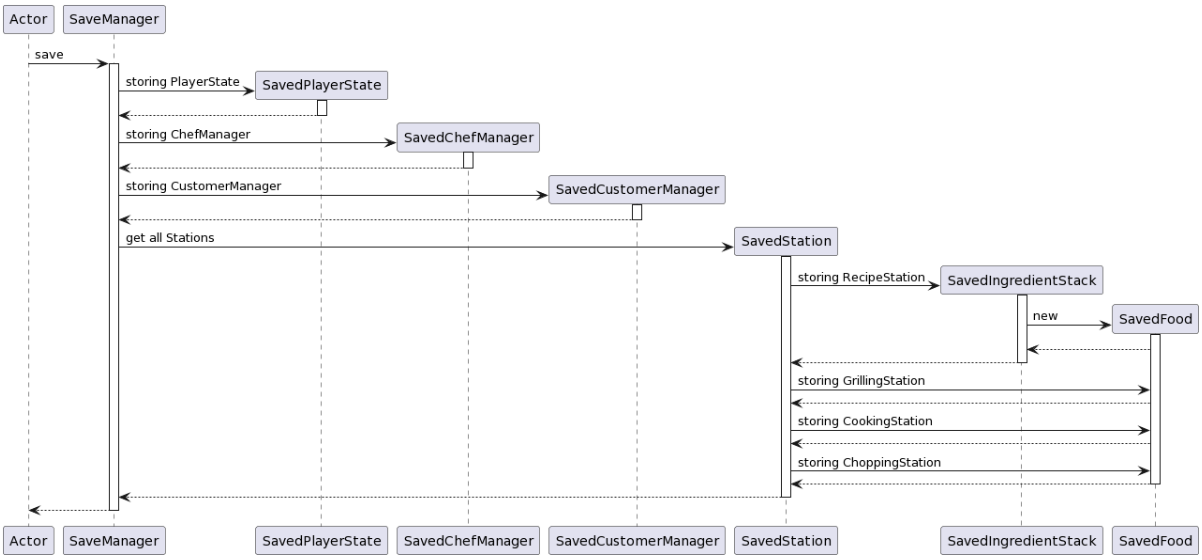
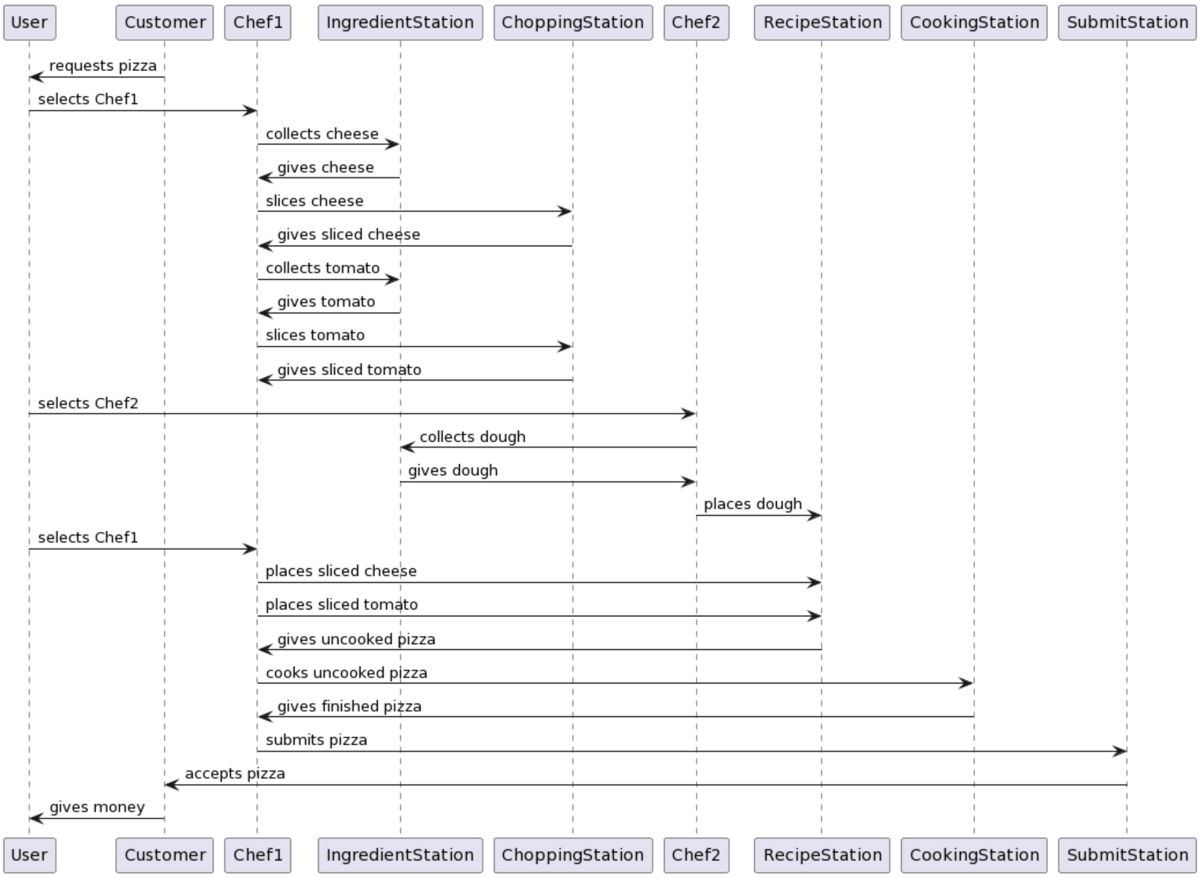


Figure 13: A sequence diagram depicting the process of making a pizza, including the required stations and food items.



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. Additionally, for assessment 2, we used draw.io to create simple class diagrams (figures 10 and 11).

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.

During the implementation of the Assessment 2 requirements, it became apparent that the current implementation of ingredients and recipes, in conjunction with the way the stations were implemented, was not suited for recipes that are not finished at the recipe station. For example, the pizza is completed in the oven rather than at a recipe station. For this reason, we refactored the implementation of ingredients. To begin with, both ingredients and recipes now implement a base Holdable interface, which is done to allow the chef to hold both recipes and ingredients. Then, ingredients now inherit from a basic class that determines what actions they can do (such as BasicCookable, or BasicChoppable). Finally, we introduced a new station to separate grilling (for the patty) and cooking (for the pizza and potato), along with returning to a separate submit station from the recipe station. This allows us to more accurately fulfil FR_CUSTOMERS. Using these basic food classes, we were also able to implement FR_TIME_FAILURE, as the timers for how long food takes to cook is now within those classes, and so we could cause the food to become unusable when it is left on the station after processing for too long.

Additionally, in order to further fulfil FR_CUSTOMERS we are required to represent the Customer on the screen, which resulted in us creating a Customer package at the same level as the Chef package, moving CustomerManager into that class along with a new class Customer, represented in figure 10. The implementation of this requirement also led to us creating a Box2D package, which contains our classes used to control a customer's movement. Within the existing file structure, it would have been impossible to implement customers in such a way due to them only existing as orders within CustomerManager.

Furthermore, creating customers as an individual class also aids in the implementation of reputation loss, as a timer for this must be unique to a customer.

In the process of our changes to Customer and CustomerManager, we also implemented the endless and variable scenario modes, as well as the assessment 1 requirement of a customer spawning on a timer as opposed to once their order has been fulfilled, meeting FR_ENDLESS_CUSTOMERS.

Initially, we had planned to complete the power ups aspect of FR_SHOP through a PowerUps object that would be given to the relevant classes that needed to be affected by it. The power ups we chose (double chef speed, double preparation speed, no reputation loss, no failed preparation steps and double earnings) meant that it would have to be accessed in very separate parts of the code, being the food classes, the customer, the chef and the function to earn money. However, due to the bloat this would cause we instead chose to implement the power ups as part of a singleton class, PlayerState. This class also allowed us to easily implement the other aspects of the shop like storing earned money, hiring chefs, unlocking more stations and a difficulty mode. As a singleton class, it allowed us to easily have values be accessed in their relevant places across the program, which allowed us to meet FR_SPENDING, FR_SHOP, and FR_DIFFICULTY, with the values for these being set through our UI classes.

In order to meet our final new requirement of saving and loading (FR_SAVE and FR_LOAD), we were required to create a number of new classes, demonstrated in figure 11. We decided to store our relevant values in a class named SaveState, which holds instances of the other classes that we need to save, those being: ChefManager (to save things like the number of active chefs and their inventories), CustomerManager (to save things like how many customers there are at present, along with their orders and time remaining before they deduct reputation), PlayerState (to store the player’s cash and progress with unlocking stations and more chefs) and all Stations that aren’t ingredient or submit stations, as those are derived from the map and CustomerManager respectively. We created saveable versions of these classes (and the relevant subclasses like Chef and Customer) so that they can be stored as a json file.