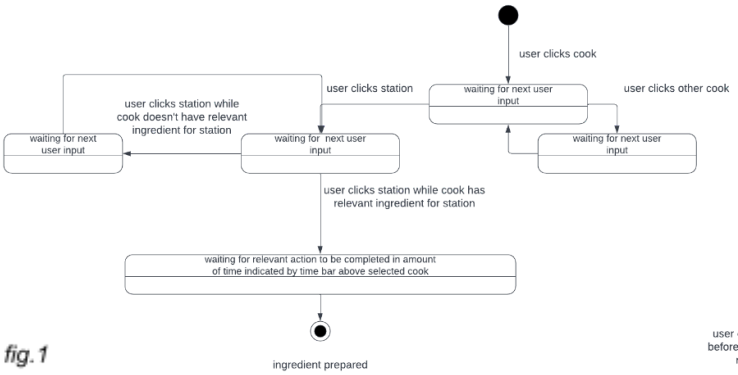

fig.1

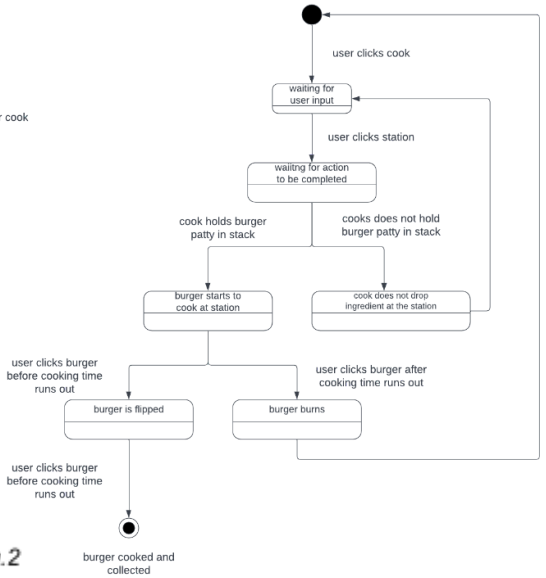

fig.2

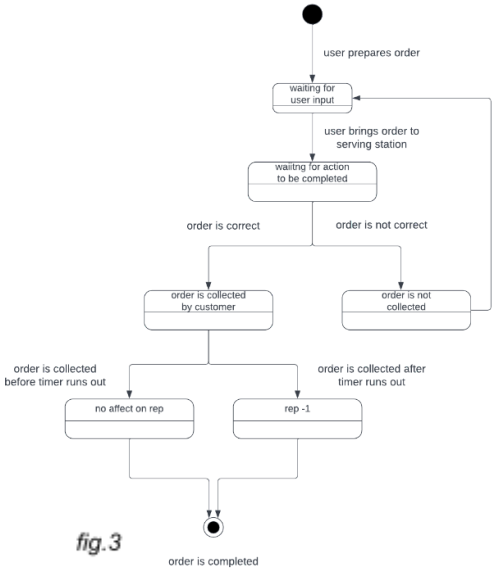

fig.3

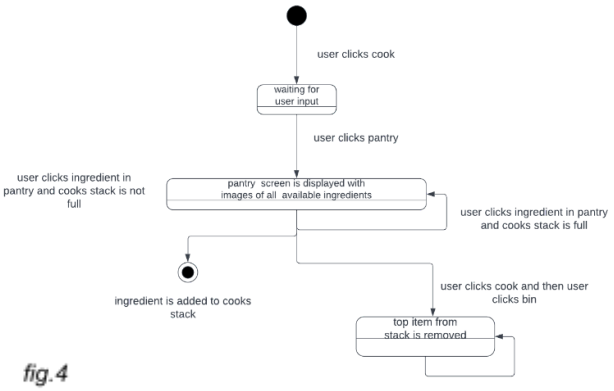

fig.4

user clicks bin

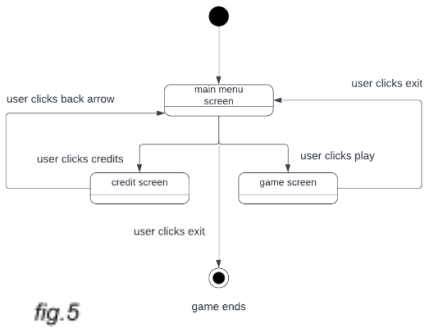

fig.5

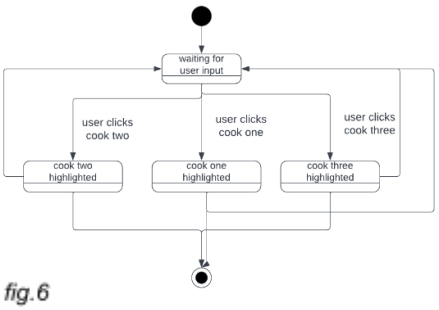

fig.6

cook selected and ready to act from user input

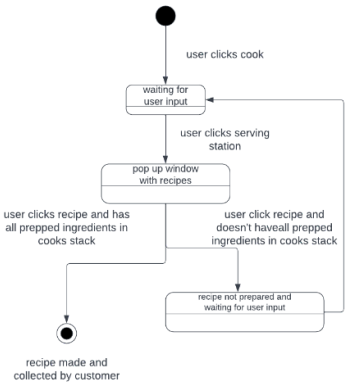

fig.7

Our state diagrams [fig.1-7] and class diagrams [fig.9-10] were made using LucidChart, which was integrated into our shared Google Drive.

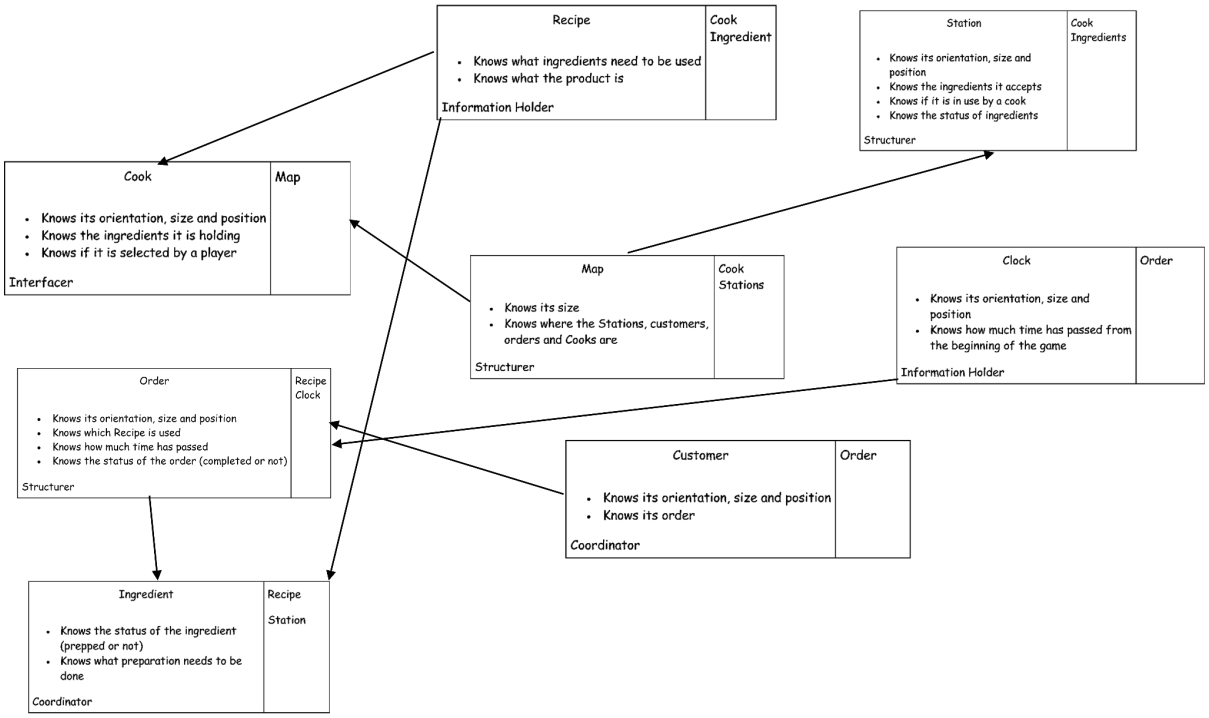

fig.8

Our CRC cards [fig.8] were made in Microsoft Word.

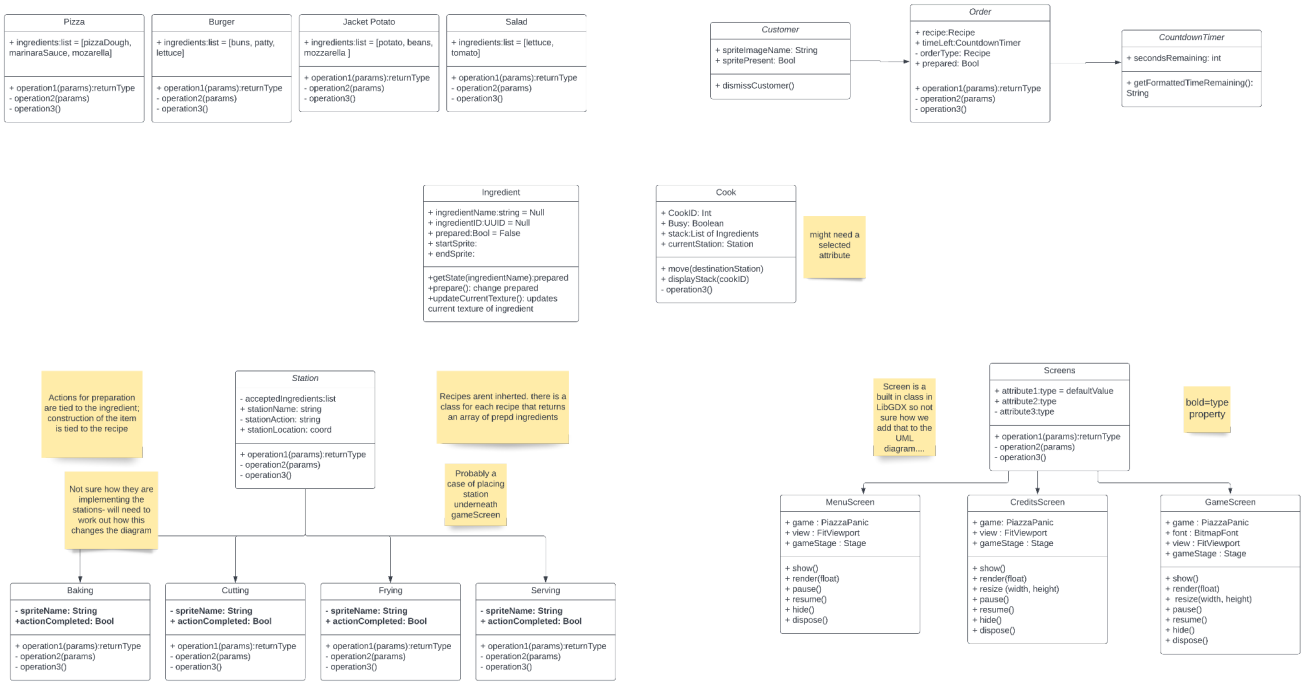

fig.9

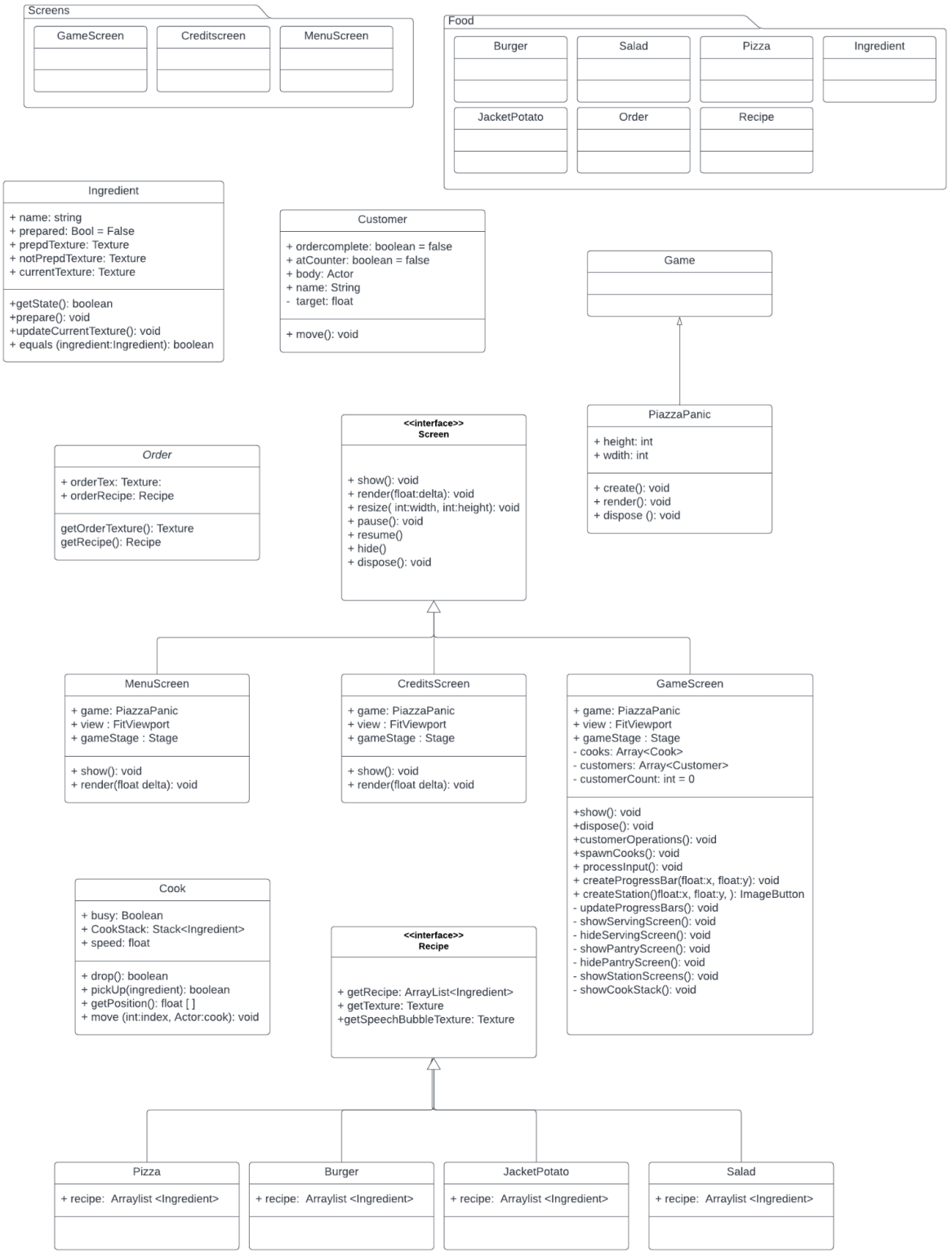
Screens
GameScreen
Creditscreen
MenuScreen
Food
Burger
Salad
Pizza
Ingredient
JacketPotato
Order
Recipe
Ingredient
+ name: string
+ prepared: Bool = False
+ prepdTexture: Texture
+ notPrepdTexture: Texture
+ currentTexture: Texture
+getState(): boolean
+prepare(): void
+updateCurrentTexture(): void
+ equals (ingredient:Ingredient): boolean
Customer
+ ordercomplete: boolean = false
+ atCounter: boolean = false
+ body: Actor
+ name: String
- target: float
+ move(): void
Game
PiazzaPanic
+ height: int
+ wdith: int
+ create(): void
+ render(): void
+ dispose (): void
Order
+ orderTex: Texture:
+ orderRecipe: Recipe
getOrderTexture(): Texture
getRecipe(): Recipe
<<interface>>
Screen
+ show(): void
+ render(float:delta): void
+ resize(int:width, int:height): void
+ pause(): void
+ resume()
+ hide()
+ dispose(): void
MenuScreen
+ game: PiazzaPanic
+ view : FitViewport
+ gameStage : Stage
+ show(): void
+ render(float delta): void
CreditsScreen
+ game: PiazzaPanic
+ view : FitViewport
+ gameStage : Stage
+ show(): void
+ render(float delta): void
GameScreen
+ game: PiazzaPanic
+ view : FitViewport
+ gameStage : Stage
- cooks: Array<Cook>
- customers: Array<Customer>
- customerCount: int = 0
+show(): void
+dispose(): void
+customerOperations(): void
+spawnCooks(): void
+ processInput(): void
+ createProgressBar(float:x, float:y): void
+ createStation()float:x, float:y,): ImageButton
- updateProgressBars(): void
- showServingScreen(): void
- hideServingScreen(): void
- showPantryScreen(): void
- hidePantryScreen(): void
- showStationScreens(): void
- showCookStack(): void
Cook
+ busy: Boolean
+ CookStack: Stack<Ingredient>
+ speed: float
+ drop(): boolean
+ pickUp(ingredient): boolean
+ getPosition(): float [ ]
+ move (int:index, Actor:cook): void
<<interface>>
Recipe
+ getRecipe: ArrayList<Ingredient>
+ getTexture: Texture
+getSpeechBubbleTexture: Texture
Pizza
+ recipe: Arraylist <Ingredient>
Burger
+ recipe: Arraylist <Ingredient>
JacketPotato
+ recipe: Arraylist <Ingredient>
Salad
+ recipe: Arraylist <Ingredient>

fig.10

Overall we have used an Object-Oriented Architecture; this is what worked best for our team as we had a limited time scale to try and organise the project which can require a fair amount of overhead. It also allowed us to code and design semi-concurrently as the programming team only needed a limited amount of documentation to get started.
The nature of object-oriented architecture meant our team could program different classes separately without much need for discussion between team members whilst programming was happening. The team would then regularly discuss how various features had been implemented to ensure that it didn't impact any other classes or functions. This meant that our diagrams had to be quite flexible, which did make accurate documentation harder, as the implementation was open to change. However, the size of our team made this reasonably easy to handle, and we used systems such as messaging and post-it notes to keep each other up to date.

Our CRC cards [fig.8] were useful in laying the groundwork for our first class diagrams [imozwastaken.github.io/architecture.html#class]. Our CRC cards layout our user requirements well, making sure our class diagrams do the same. After we made good progress with the CRC cards, we translated them into class diagrams [fig.9,10]. This took effort from the whole team as this is when we started to make concrete implementation decisions. This means everyone had to understand how we organised the objects and why. Even though our class diagrams went through many changes, it was very beneficial to have a vague starting point.

One example of a change is the Screen interface. This was a feature of LibGDX that the design team were not aware of, we thought we would have to make our own Screen Class [fig.9]. Instead, our three main screens (MenuScreen, CreditsScreen, GameScreen) implement the Screen interface.

It also took a lot of consideration whether or not the separate stations would be their own classes, or attributes within the GameScreen class. We had them in separate files [fig.9] as we underestimated the reliance of the stations on the cooks. Having them as attributes [fig.10] meant that the stations could always know the state of the cooks. This is important as the stations react to the cook's interactions with it. This meant we could more easily implement the requirements: UR_COOK_ACTION, FR_MOVE_COOK, FR_USE_STATION. As without this the user would not be able to move the cook in appropriate ways. Cook and station interactions are also shown in our state diagram [fig.1], here it demonstrates UR_STATION_ACTION and FR_DROP_RESTRICTION, as the ingredients are prepared only if the cook has the right ingredients in their stack.

Another example would be in relation to the recipes. Our second design [fig.9] shows that the recipe classes would be independent of each other and not extend or implement from anywhere else. We soon realised it would be beneficial to include a Recipe interface [fig.10] into the design. This meant we could loosely link the recipes together and still have flexibility as our code was still changing a lot at this point. Without the recipe interface it would be hard to complete FR_SERVE_DISH and UR_CUSTOMER_VIEW. FR_SERVE_DISH is shown in fig.7 with FR_SERVING_STATION. A window pops up when the serving station is selected, and the recipe is made if and only if the needed prepped ingredients are in the cook's stack. This is only possible because we made the Recipes use the type Ingredient.

There was also much discussion regarding class relationships, especially between Order and Customer. Initially we thought that order would extend Customer, but as the order system was implemented we knew that this was incorrect. The Order class is about the object on the screen and therefore does not need to inherit from Customer. Customer uses the Order type to instantiate a customer's order.

Our user requirements, UR_SWITCHING_COOKS, FR_SWITCHING_COOKS, are shown in fig.6. This diagram shows how the user can select a cook, done in the game by pressing "1" or "2", and then move it to another area on the screen. This is done by clicking on the desired location (UR_MOVING_COOK). Our architecture supports this as we have a class Cook which has an attribute which controls the state of the cook regarding if it is in use or not. Without this the user could move the cook at any point.

UR_COOK_STACK is shown in fig.4, with the stack constraint. Fig.4 also shows how the user can remove things from the cook's stack by clicking the bin. This is different from our system requirement FR_VIEW_PANTRY as this says the bin icon should be in the pantry window. However, after consideration, we decided that this icon would be too small and there would be too many functionalities in one space.

UR_REPUTATION and UR_TIME_CUSTOMERS are represented in fig.3. An order is only completed if the customer receives the correct order. The user's reputation is affected by whether or not the order is completed within the given time.

These discussions have led to many changes in our Class diagrams [fig.9], which is why our agile method has been so helpful. During scrummages, we have been able to update the different teams and documentation on the changes, as well as have discussions about which changes are necessary and which implementations will be most effective. A lot of the time, we found that the programming team could be a little short-sighted in the game implementation, focusing just on the bits they needed to get done; however, the object-oriented structure allowed the other teams to draw their attention to future changes that will need to be made and how to implement the current program to make that as easy as possible.

We struggled to pick a specific architecture and instead stuck with Object-Oriented because we knew we would be handing the project over to another group. We know that all the other students are familiar with OOP. It is a very modular programming style allowing for different functions and methods to be added and changed without impacting the program's functionality too much. It also lends itself to accurate and in-depth code documentation as you can include JavaDocs for each Class, each Instance of those classes, each Method within the class and so on. This allows for a much easier handover as other programmers can easily see what has been implemented and how.