Architecture

Group 17

Team Loading

Joel Warren
Charlie Wilson
Jake Keast
Hari Bhandari
Ishrit Thakur
Joshua Hills

3. a) Diagrammatic representations

State diagrams

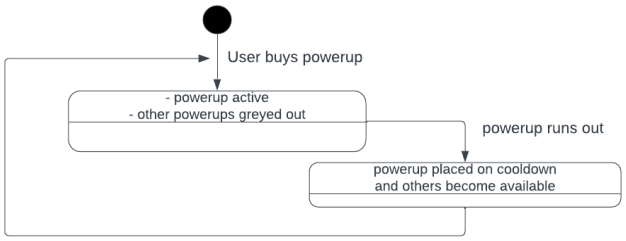


Fig. 7

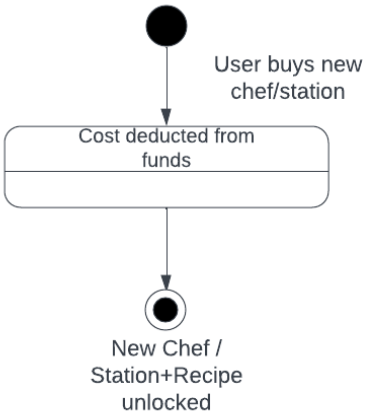


fig. 8

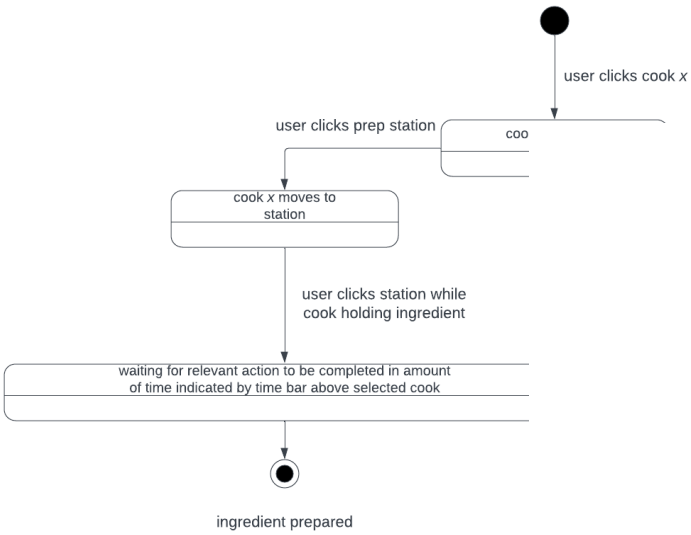


fig.1

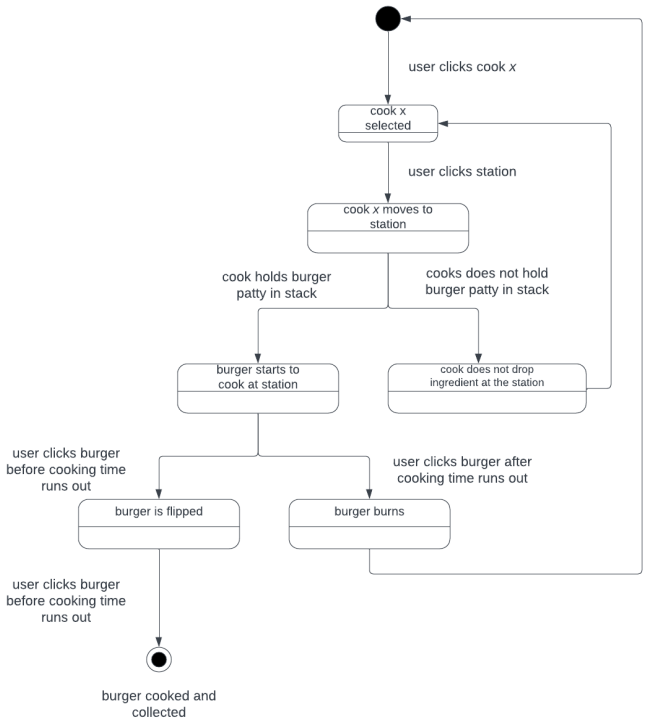
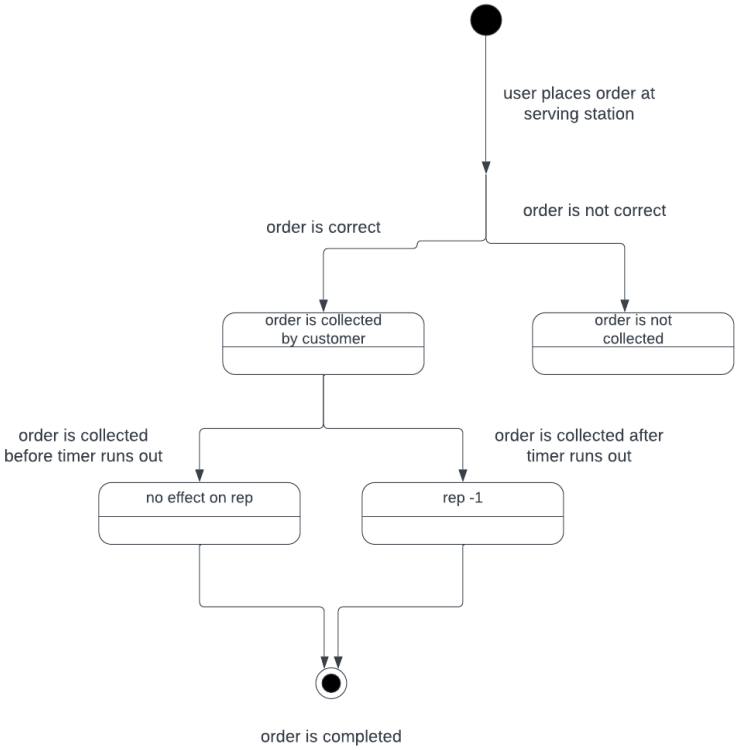


Fig.2

Fig. 3


user places order at serving station
order is correct
order is not correct
order is collected by customer
order is not collected
order is collected before timer runs out
order is collected after timer runs out
no effect on rep
rep -1
order is completed

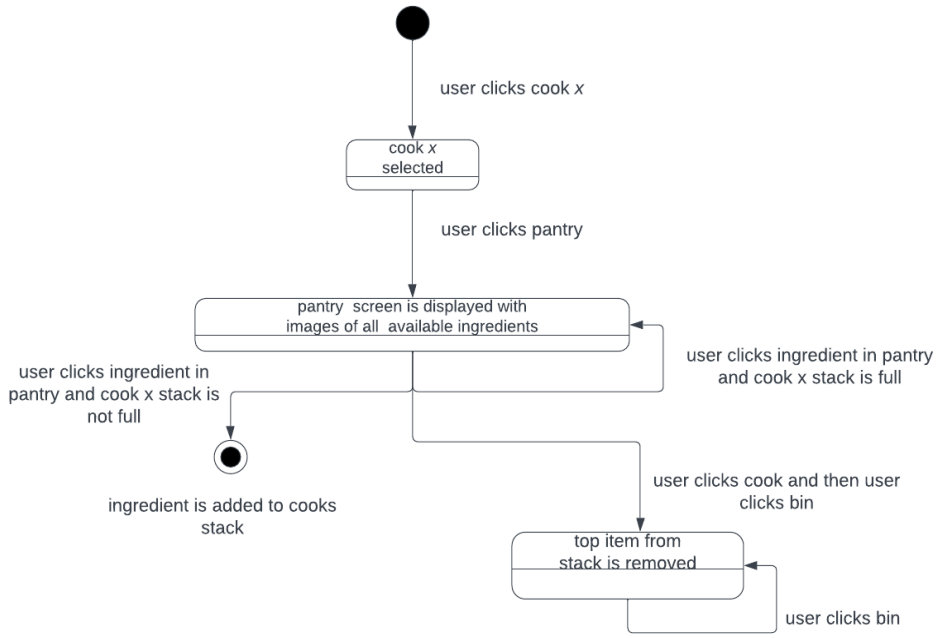

user clicks cook x
cook x selected
user clicks pantry
pantry screen is displayed with images of all available ingredients
user clicks ingredient in pantry and cook x stack is full
user clicks ingredient in pantry and cook x stack is not full
ingredient is added to cooks stack
user clicks cook and then user clicks bin
top item from stack is removed
user clicks bin

Fig. 4

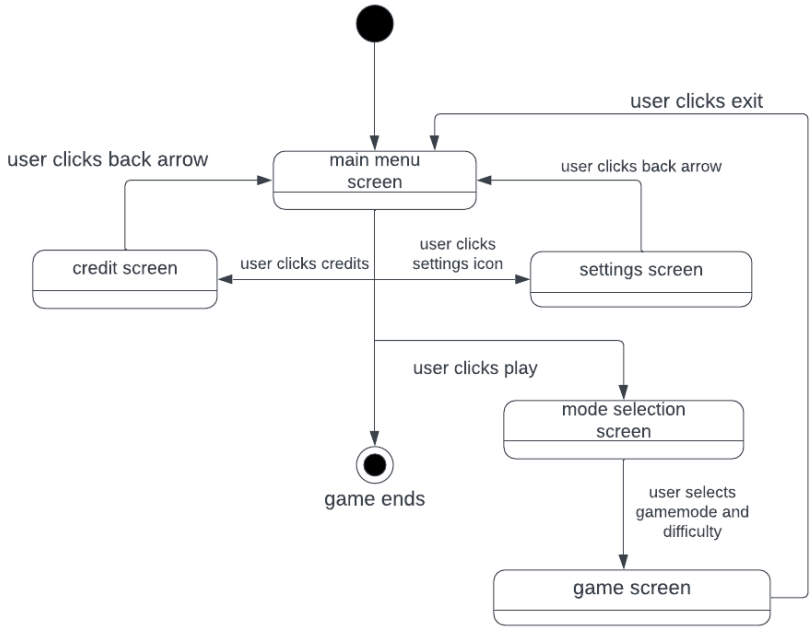

user clicks exit
user clicks back arrow
main menu screen
user clicks back arrow
credit screen
user clicks credits
user clicks settings icon
settings screen
user clicks play
mode selection screen
game ends
user selects gamemode and difficulty
game screen

fig.5

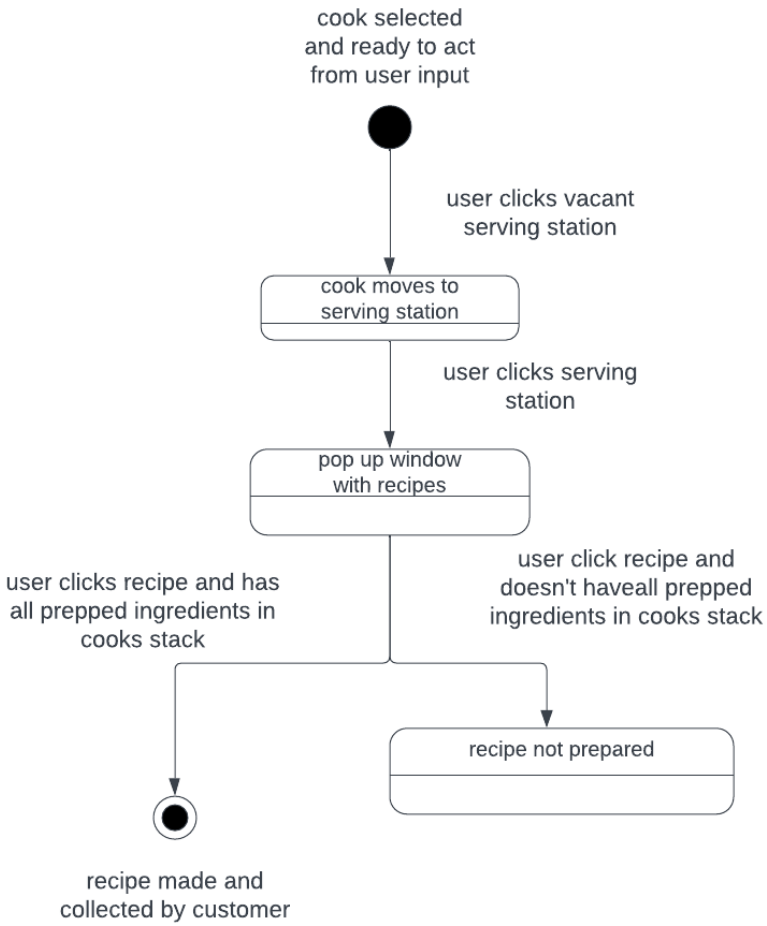
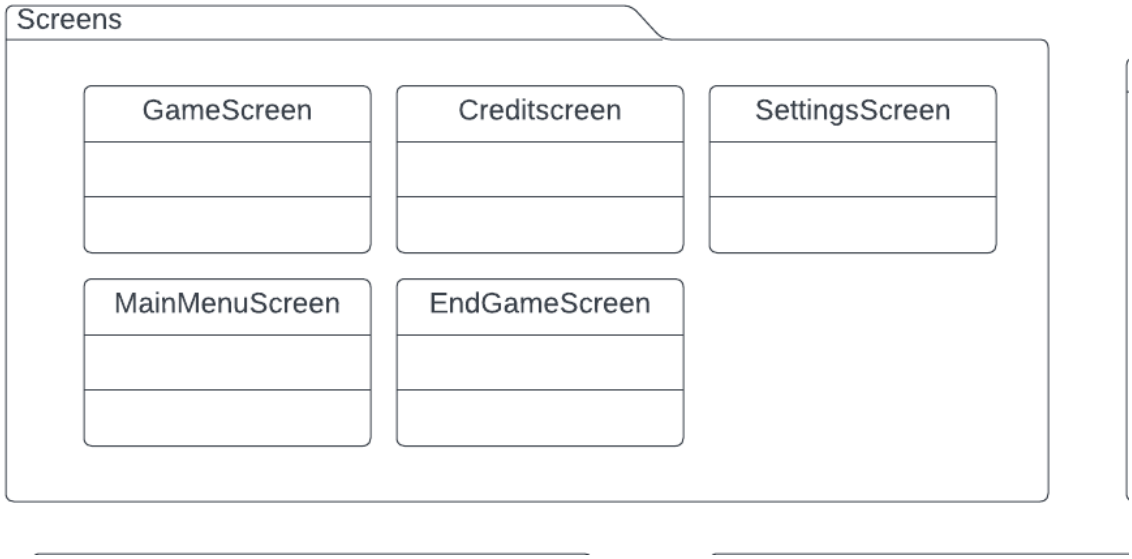
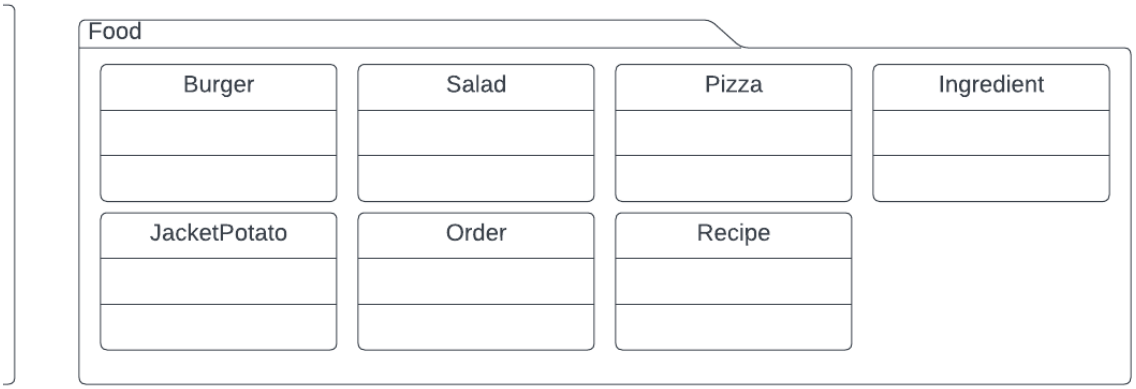
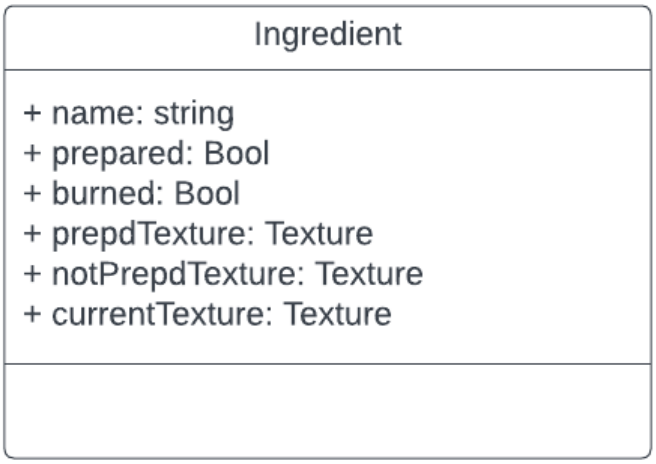


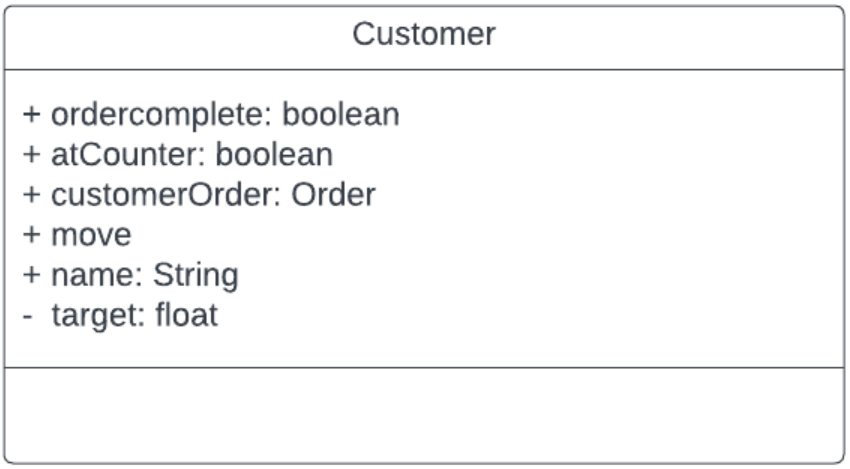
fig.6

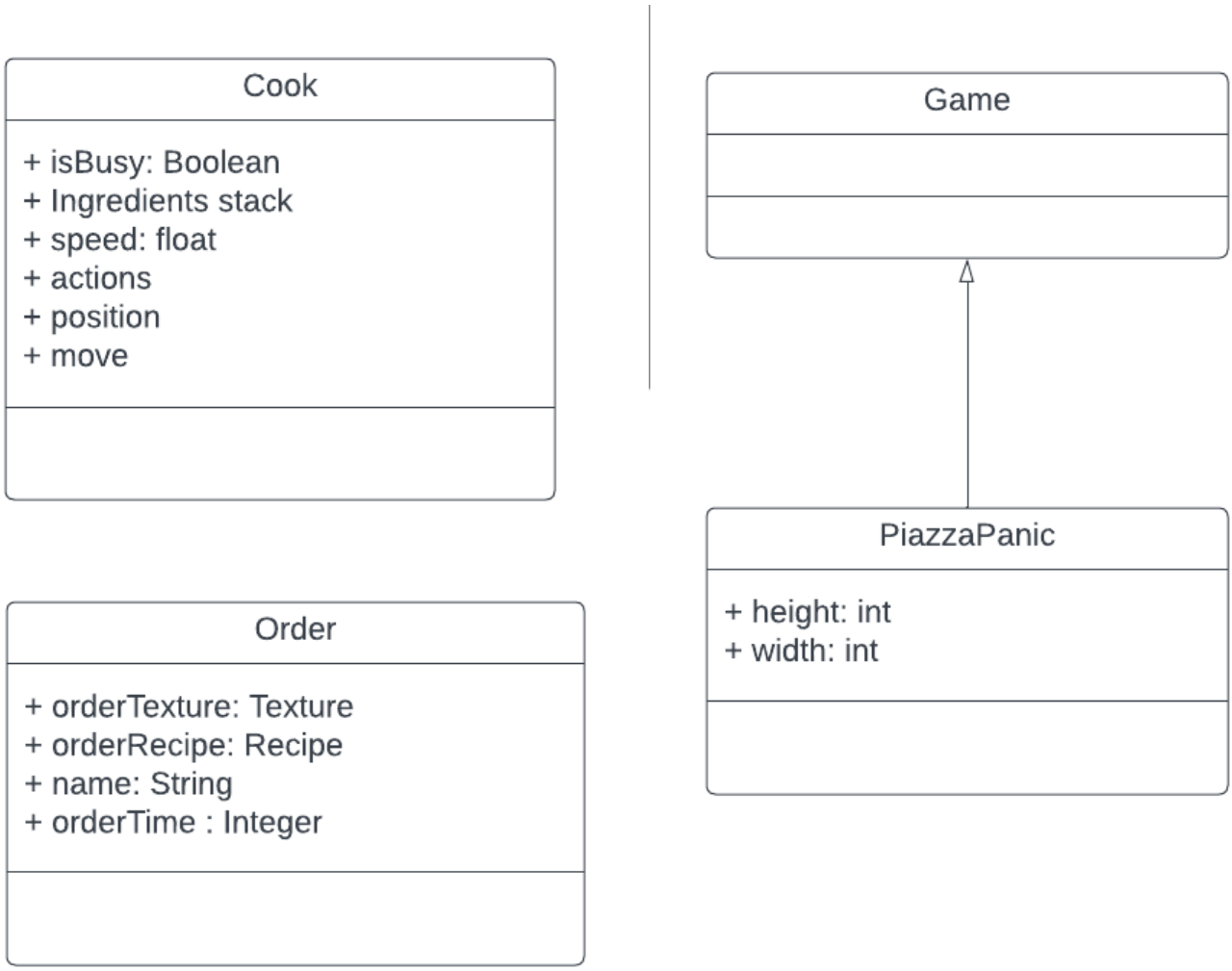
Class diagrams

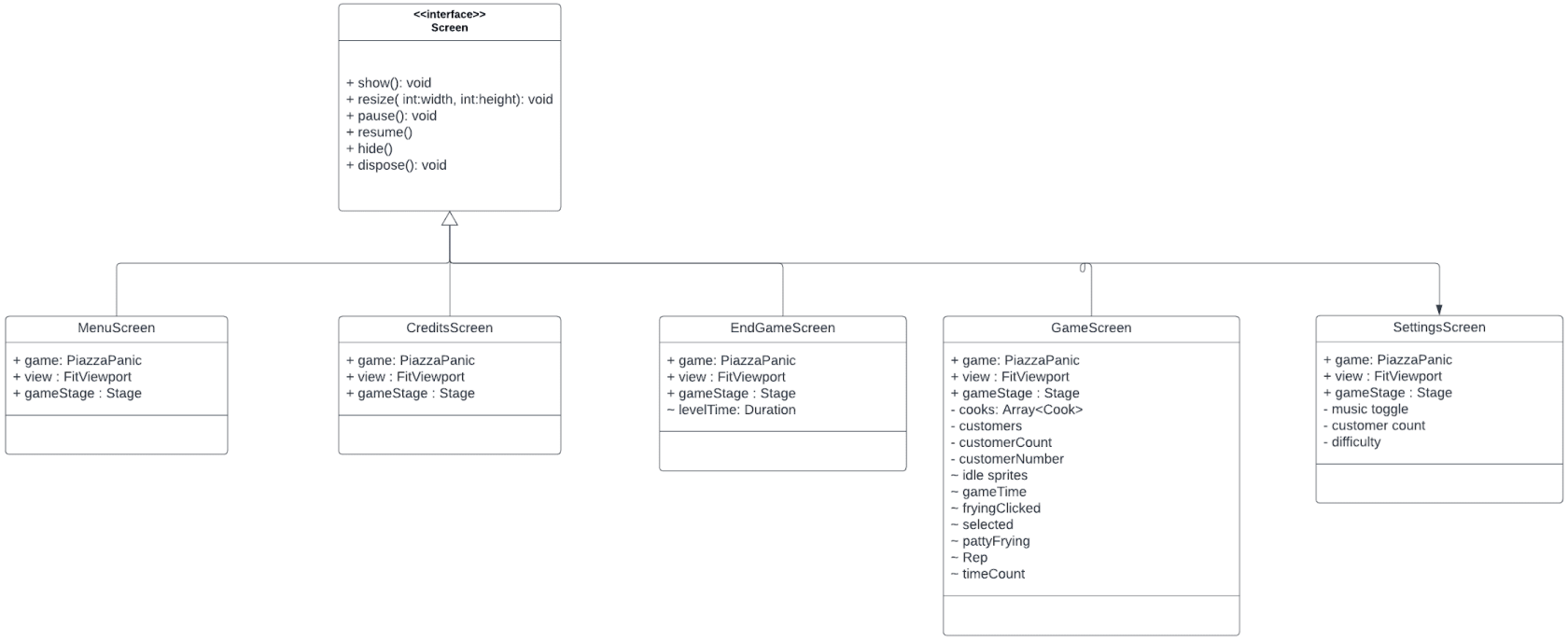

Screens
GameScreen
Creditscreen
SettingsScreen
MainMenuScreen
EndGameScreen


Food
Burger
Salad
Pizza
Ingredient
JacketPotato
Order
Recipe


Ingredient
+ name: string
+ prepared: Bool
+ burned: Bool
+ prepdTexture: Texture
+ notPrepdTexture: Texture
+ currentTexture: Texture


Customer
+ ordercomplete: boolean
+ atCounter: boolean
+ customerOrder: Order
+ move
+ name: String
- target: float


Cook
+ isBusy: Boolean
+ Ingredients stack
+ speed: float
+ actions
+ position
+ move
Game
Order
+ orderTexture: Texture
+ orderRecipe: Recipe
+ name: String
+ orderTime : Integer
PiazzaPanic
+ height: int
+ width: int


<<interface>>
Screen
+ show(): void
+ resize(int:width, int:height): void
+ pause(): void
+ resume()
+ hide()
+ dispose(): void
MenuScreen
+ game: PiazzaPanic
+ view : FitViewport
+ gameStage : Stage
CreditsScreen
+ game: PiazzaPanic
+ view : FitViewport
+ gameStage : Stage
EndGameScreen
+ game: PiazzaPanic
+ view : FitViewport
+ gameStage : Stage
~ levelTime: Duration
GameScreen
+ game: PiazzaPanic
+ view : FitViewport
+ gameStage : Stage
- cooks: Array<Cook>
- customers
- customerCount
- customerNumber
~ idle sprites
~ gameTime
~ fryingClicked
~ selected
~ pattyFrying
~ Rep
~ timeCount
SettingsScreen
+ game: PiazzaPanic
+ view : FitViewport
+ gameStage : Stage
- music toggle
- customer count
- difficulty

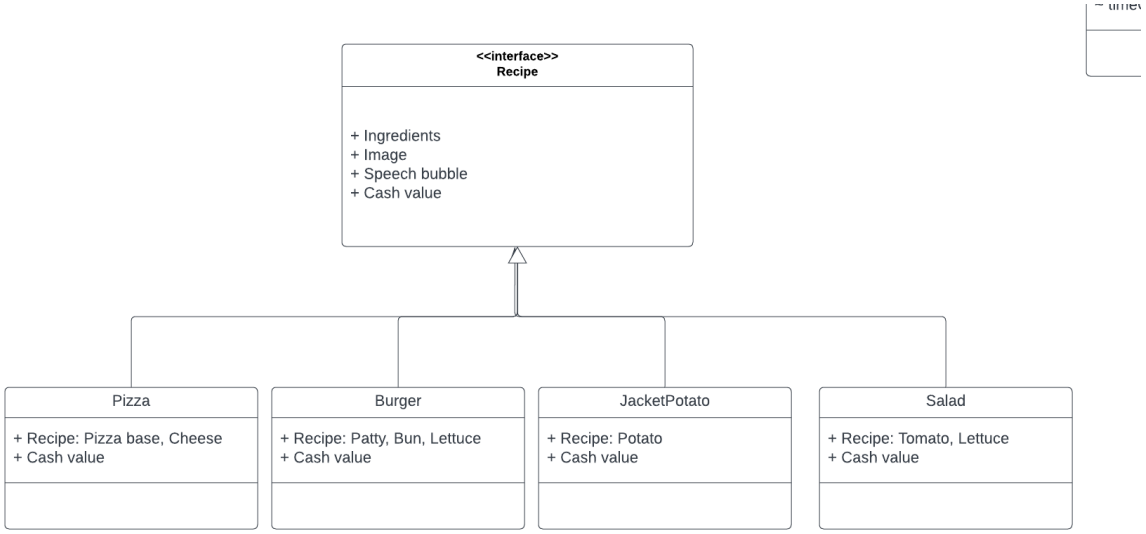


Fig.10 (all class diagrams)

The tool we used to extend architecture was Lucidchart. This allowed for easier extension and continuous diagramming since this was the tool used to create the assessment 1 architecture by the previous team. However, this did take some time to adjust from our previous experience with plantUML.

3. b) Justification, design evolution and requirements traceability for architecture

Justification

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.

Class diagram provides the basic structure for the implementation of our code. This diagram would give an ideal starting point and class hierarchy needed to base our code on.

The state diagrams show various "phases" that each of the actors on screen could be in. Each of these phases provide a different behaviour for the actor and inputs and processes should differ based upon these phases. This allows those working on implementation (Charlie and Jake) to ensure that nothing is unaccounted for and that each phase behaves correctly.

Justification for design process

The previous team's architecture was derived from the requirements, because this was the recommended process described in lectures and external literature. This allowed them to create accurate diagrams which sufficiently reflected the user requirements and thus correctly representing the game's components. This did mean that requirements had to be finished before architecture could be worked on.

The design process we used to extend architecture ensured that all new and existing requirements were properly represented, and ensured correct simplicity for the architecture by removing unnecessary features/java jargon.

Design evolution

The 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.

From this point, team ownership changed, with our team (team 17) taking ownership of the project.

Since taking ownership, assessment 2 introduced new requirements, as well as new requirements from the customer. This meant that the statement of user and system requirements had to be updated. Then, we could analyse the requirements against the existing architecture to highlight missing requirements (mainly assessment 2). This allowed us to extend existing architecture, add new architecture as well as remove unnecessary architecture.

This resulted in further evolution of the architecture. Existing diagrams were extended to include new requirements, such as the screen diagram now including settings screen. Changes were also made to existing architecture, such as most of the class diagrams were made simpler by removing technical java jargon. This is because the architecture should be simple and more high level general concepts.

Lastly, when we created and extended these architectures, they then did not change majorly at all, since when we created them we could do so accurately as we already had a good understanding how the game worked and functioned, and of the new requirements for assessment 2.

Requirements traceability

Architecture	**Related Requirements**
Class GameScreen	● UR_SWITCHING_COOKS ● UR_COOK_ACTION ● UR_STATION_ACTION ● UR_MOVING_COOK ● UR_CUSTOMER_VIEW ● UR_TIME_CUSTOMERS ● UR_REPUTATION ● UR_COOK_STACK ● UR_MOVEMENT_WASD ● UR_PURCHASE_COOK ● UR_POWERUP ● UR_RECIPES ● UR_STATION_ACTION ● FR_DIFFERENT_COOKS ● FR_DESTINATIONS ● FR_USE_STATION ● FR_DROP_RESTRICTION ● FR_VIEW_PANTRY ● FR_EXIT_PANTRY ● FR_SERVING_STATION ● FR_EXIT_SERVING_STATION ● FR_REPUTATION ● FR_BURGER_BURN

	• FR_CUSTOMER_GROUPS • FR_CUSTOMER_LEAVE
Class Cook	• UR_COOK_STACK • UR_MOVING_COOK • FR_COOK_RESTRICTIONS • UR_MOVEMENT_WASD
Class Ingredient	• FR_TAKE_PREPPED_INGREDIENT • UR_RECIPES
Class Customer	• UR_CUSTOMER_VIEW • FR_CUSTOMER_GROUPS • FR_CUSTOMER_LEAVE
Class EndGameScreen	• UR_SCENARIO_TIME • UR_MAX_SERVE
Class SettingsScreen	• UR_MOVEMENT_WASD • UR_DIFFICULTY_SELECT • FR_DIFFICULTY