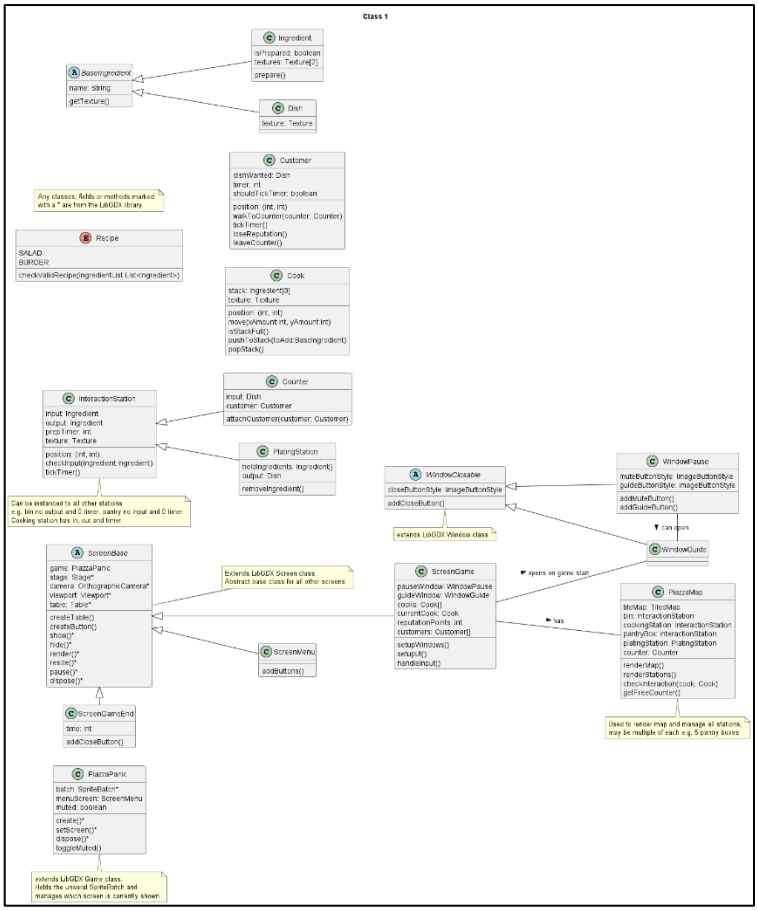
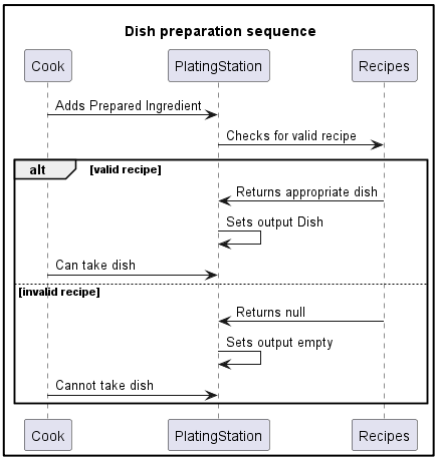
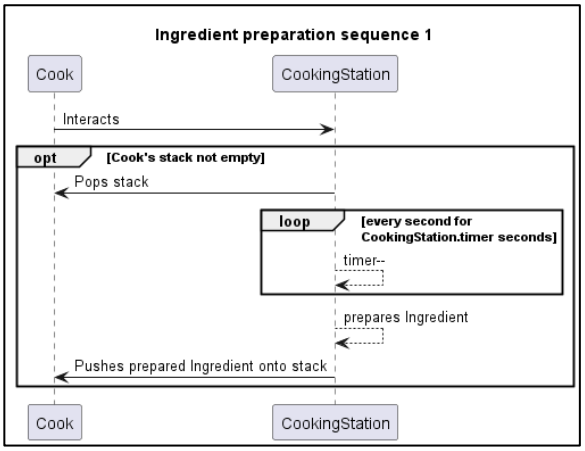
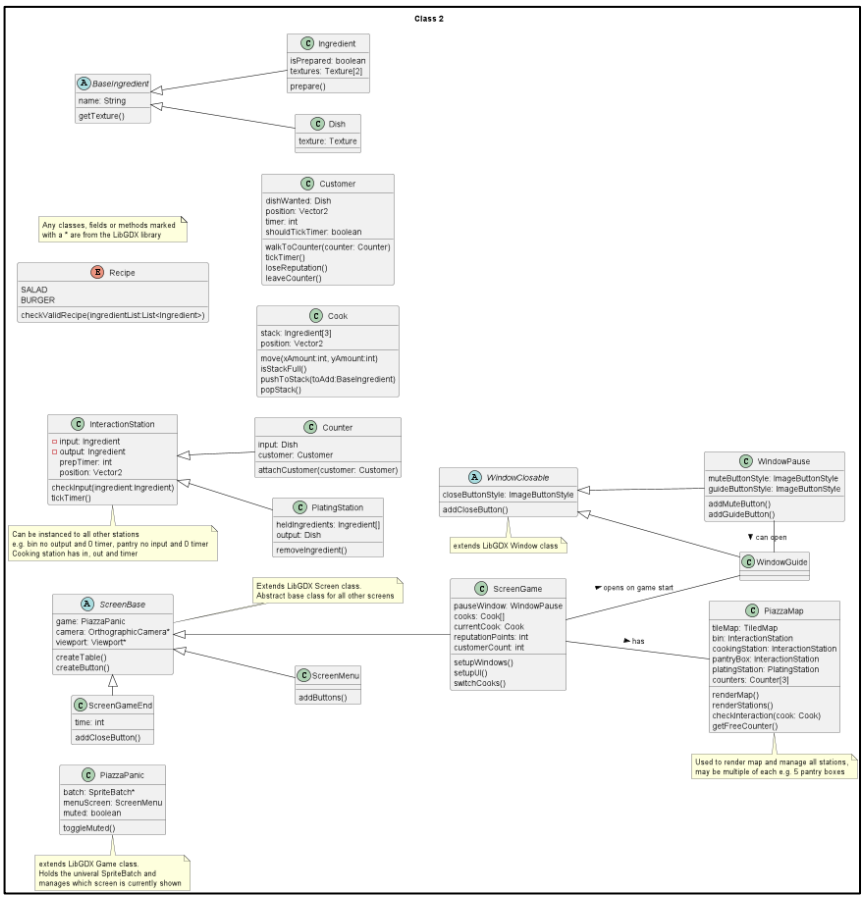
Hand in 3A

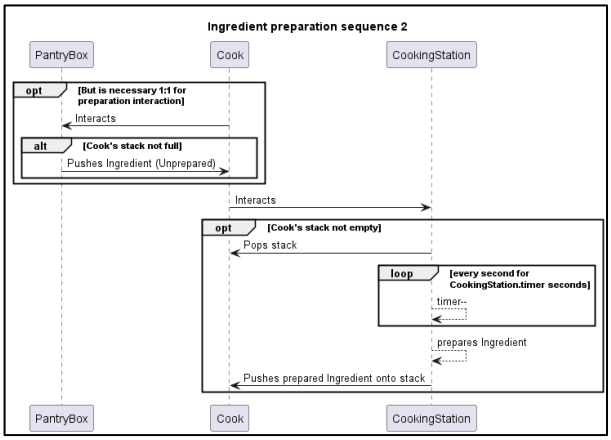
diagrams were upscaled but if they are hard to read they have also been uploaded to the website at https://eng-25.github.io/#UML

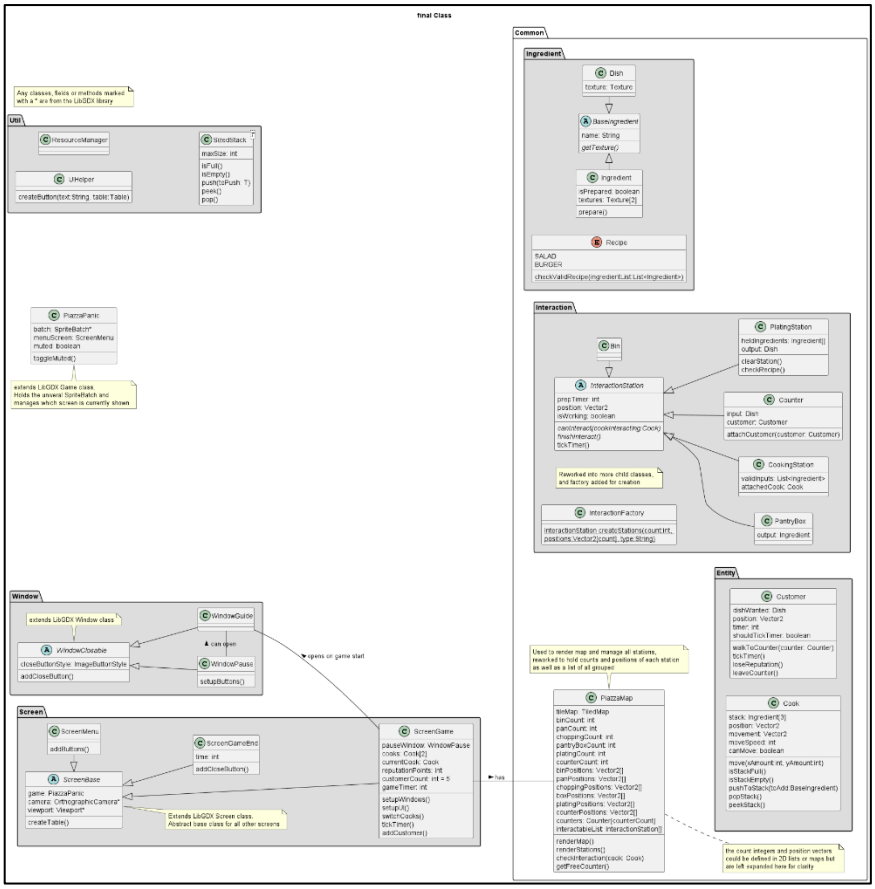


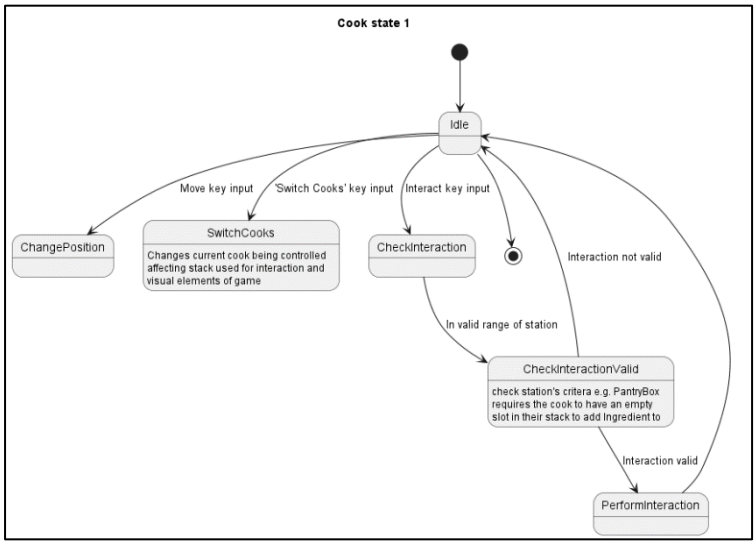


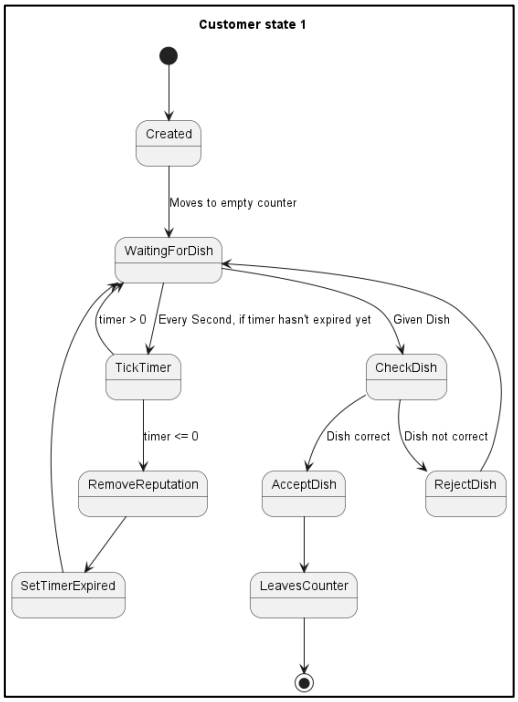


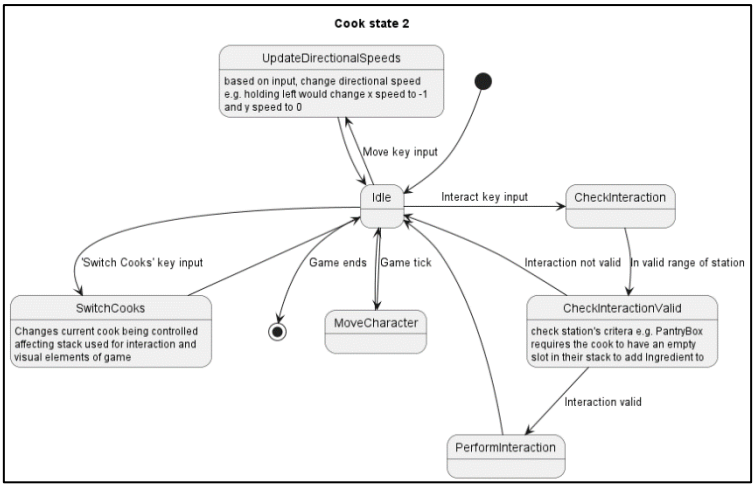


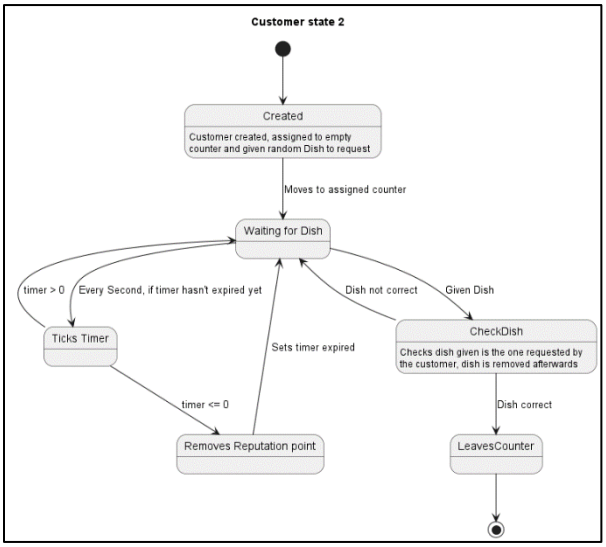



final Class
Common
Ingredient
Dish
texture: Texture
BaseIngredient
name: String
getTexture()
Ingredient
isPrepared: boolean
textures: Texture[3]
prepare()
Recipe
SALAD
BURGER
checkValidRecipe(ingredientList:List<IngredientE>)
Any classes, fields or methods marked with a * are from the LibGDX library
Util
IResourceManager
SimpleStack
maxSize: int
isFull()
isEmpty()
push(toPush: T)
peek()
pop()
UIHelper
createButton(text:String, table:Table)
PizzaPanic
batch: SpriteBatch*
menuScreen: ScreenMenu
muted: boolean
toggleMute()
extends LibGDX Game class.
Holds the shared SpriteBatch and manages which screen is currently shown
Interaction
Bin
PlatingStation
holdIngredients: Ingredient[]
output: Dish
clearStation()
checkRecipe()
InteractionStation
prepTimer: int
position: Vector2
isWorking: boolean
canInteract(cookInteracting:Cook)
finishInteract()
tickTimer()
Counter
input: Dish
customer: Customer
attachCustomer(customer: Customer)
CookingStation
validInputs: List<Ingredient>
attachedCook: Cook
Reworked into more child classes, and factory added for creation
InteractionFactory
interactionStation createStations(count:int, positions:Vector2[count], type:String)
PantryBox
output: Ingredient
Entity
Customer
dishWanted: Dish
position: Vector2
timer: int
shouldTickTimer: boolean
walkToCounter(counter: Counter)
tickTimer()
loseReputation()
leaveCounter()
Cook
stack: Ingredient[3]
position: Vector2
movement: Vector2
moveSpeed: int
canMove: boolean
move(xAmount:int, yAmount:int)
isStackFull()
isStackEmpty()
pushToStack(toAdd:BaseIngredient)
popStack()
peekStack()
Window
extends LibGDX Window class
WindowGuide
WindowClosable
closeButtonStyle: ImageButtonStyle
addCloseButton()
can open
WindowPause
setupButtons()
opens on game start
Used to render map and manage all stations, reworked to hold counts and positions of each station as well as a list of all groups
PlazzaMap
tileMap: TiledMap
binCount: int
panCount: int
choppingCount: int
pantryBoxCount: int
platingCount: int
counterCount: int
panPositions: Vector2[]
choppingPositions: Vector2[]
boxPositions: Vector2[]
platingPositions: Vector2[]
counterPositions: Vector2[]
counters: Counter[counterCount]
interactableList: interactionStation[]
renderMap()
renderStations()
checkInteraction(cook: Cook)
getFreeCounter()
has
the count integers and position vectors could be defined in 2D lists or maps but are left expanded here for clarity
Screen
ScreenMenu
addButtons()
ScreenGameEnd
time: int
addCloseButton()
ScreenBase
game: PizzaPanic
camera: OrthographicCamera*
viewport: Viewport*
createTable()
Extends LibGDX Screen class.
Abstract base class for all other screens
ScreenGame
pauseWindow: WindowPause
cooks: Cook[2]
currentCook: Cook
reputationPoints: int
customerCount: int = 5
gameTimer: int
setupWindows()
setupUI()
switchCooks()
tickTimer()
addCustomer()


Cook state 1
Idle
Move key input
'Switch Cooks' key input
Interact key input
ChangePosition
SwitchCooks
Changes current cook being controlled affecting stack used for interaction and visual elements of game
CheckInteraction
Interaction not valid
In valid range of station
CheckInteractionValid
check station's critera e.g. PantryBox requires the cook to have an empty slot in their stack to add ingredient to
Interaction valid
PerformInteraction


Customer state 1
Created
Moves to empty counter
WaitingForDish
timer > 0
Every Second, if timer hasn't expired yet
Given Dish
TickTimer
CheckDish
timer <= 0
Dish correct
Dish not correct
RemoveReputation
AcceptDish
RejectDish
SetTimerExpired
LeavesCounter





To represent our architecture, we chose to use Unified Modelling Language (UML), as it can be used to represent both the structure and the behaviour of the system. We settled on using class diagrams to represent the entire system's structure, and state and sequence diagrams to represent necessary behaviours. To create the UML diagrams, we chose to use PlantUML because it could be used inside of our code IDEs, as well as in our Google Docs files. This made it easy to create and modify diagrams in familiar environments and as a group in shared files.

Deliverable 3b)

All UML diagrams are on the website: https://eng-25.github.io/#UML. Diagrams are referenced in the document as [DiagramName]

We decided to use a Responsibility-Driven Design process to help bridge the gap between requirements and architecture. Firstly, we used our requirements to create a simple design story.

Design story:

- We are developing a Java cooking game to be run on a desktop that is designed for a University of York Computer Science open day.
- On launch, the player is met with a main menu containing a play button and a mute button in the corner.
- When they start the game, they are presented with a 'how to play' skippable set of pages.
- When the game begins, 5 customers begin to arrive at the counter 1 at a time, with a max of 3 at once and place a random order of either burger or salad with a given completion time.
- The player can move around in the kitchen map as one of two cooks and switch between them at any point. They can pick up ingredients from boxes in the pantry which stack. Ingredients can be placed on certain cooking stations which can be interacted with and take a given amount of time to prepare. Prepared ingredients can then be combined at a plating station (which can be picked back up at any point). If prepared ingredients are combined as a correct recipe, their respective food is created. The food can then be picked up and served at the counter.
- If the correct food is given to a customer they leave. If the customer's timer expires at any time, a reputation point is lost but the customer will not leave until they are served correctly. The player starts with 3 reputation points, if they are all lost the game ends and the player has lost.
- Once all 5 customers have been served without the player losing, the game ends and the player is presented with an end screen displaying their time taken.
- The game can be paused at any point and controls/mute/quit can be accessed.

Candidates:

- MainMenuScreen, PauseScreen, InstructionScreen, GameScreen, GameEndScreen
- Cook, Customer
- CookingStation, PlatingStation
- Bin, Counter
- Player
- Kitchen (map)
- PantryBox
- Ingredients (prepared and unprepared state)
- Dish (finished food)
- InputHandler

From this, we developed a list of 'candidates', and grouped them. We then developed a CRC card for each, giving a brief description, adding responsibilities and labelling each with their stereotypes. Finally, we grouped cards again and went back over our design story using the candidates, determining if we needed all of them or not. For example, we ended up removing the InputHandler. We did not add collaborators to each card as this was a missed step at the time. This process was done on paper but the cards have been uploaded digitally. The cards can be seen on the website here: https://eng-25.github.io/#CRC

From this, the task of developing initial UML diagrams was given to Faran. This resulted in an overall class diagram, Customer and Cook state diagrams, as well as Ingredient and Dish preparation sequence diagrams. [Class1, CustomerState1, CookState1, IngredientPreparationSequence1, DishPreparationSequence1]

These UMLs were then reviewed as a group, and the following changes were made:

- For the class diagram, we immediately discussed removing any fields/methods which were part of third-party libraries. This was done in most places, however, we kept in a few fields and methods which we felt were necessary to their classes' functionalities, justified as follows. BaseIngredient and its inheritors' sole purpose would be to store a name String and a Texture to be drawn. Although Texture is part of LibGDX, we felt it was necessary to explain the BaseIngredient classes' functionalities. Furthermore, all screens (extending ScreenBase) would need a viewport and a camera to function as a screen, justifying leaving these fields in. The main PiazzaPanic class kept the LibGDX SpriteBatch which would be used everywhere to render textures, making it important enough of a detail to leave in. We also changed all integer tuples used for positioning to 'Vector2' type. [Class2]

- In the customer state diagram, we removed redundant transitions and added more detail. We determined that when a Dish was given to a customer, it would always be lost even if incorrect. Therefore, Dish's outcome is not affected by whether it's correct or not, so AcceptDish and RejectDish states were not needed as no logic would happen in them. We also removed the redundant SetTimerExpired state. In the cook state diagram, we reworked the movement key input to affect the cook's speed, which is then used in the calculation to move the cook every game tick. We figured the player's input affecting the speed and not directly causing the input could reduce movement delays as movement would now happen every tick and did not directly rely on key input. We also thought it might create opportunity for speeding up/slowing down mechanics using the

cook's speed. We had also forgotten a number of loop-back transitions to the Idle state so these were added. [CookState2]

- We decided to add a game state diagram, to get a better idea of how the game would change over time, due to certain events. [GameState]

- The Dish preparation sequence diagram did not elicit any changes. For the ingredient preparation sequence diagram, we felt it was important to keep the PantryBox interaction as this was how Ingredients would be obtained (necessary for any preparation interaction to take place), but agreed it was not directly necessary for that interaction to happen immediately before the preparation interaction. So, we made it an optional in the diagram, noting that it was a 1-to-1 necessity, but not strict in sequence. [IngredientPreparationSequence2]

Later, after a little time implementing, we further discussed any changes to make to the diagrams.

-For our class diagram, we thought it would be appropriate to plan how to package classes so we all agreed on the project's structure. Furthermore, we discussed reworking the Interaction class system. Our initial idea was to use the InteractionStation class for most interactable parts of the system, but we realised our idea would result in lots of null value usage and was not going to work how we had envisioned, so it was split into more child classes and an abstract base class. We also saw the opportunity to rework the creation of those interactable classes to be more efficient and more future-proof (as there would be a lot of them) so we implemented the InteractionFactory class. In our Window classes, we removed the LibGDX ImageButtonStyles we had left in, and reworked our button creation methods into a single method. Upon further discussing the button creation, we realised the createButton method in our ScreenBase class could be re-used, so we decided to move it to a helper class for global use. We had also at this point created our ResourceManager class in code, used to manage and load all assets, as well as a SizedStack for the Cook to use (a stack with a max size allowed in order to fulfil NFR_MAX_INGREDIENTS_IN_STACK requirement) so we added that to the diagram. We realised our ScreenGame class had no timer or customer logic yet, which would be needed soon for implementation, so we added the appropriate fields and methods. The movement system discussed previously and changed in the cook's state diagram during our previous discussion was not updated in the class diagram yet, so we reworked the Cook class to account for that. We had also realised the Cook would need an extra Boolean check for movement, to account for things like interacting with stations. [FinalClass]

Below are links to requirements from our architectural diagrams not mentioned in changes above:

- UR_COOKS: Cook class and Cook[2] in ScreenGame class.

- UR_SCENARIO_BASED_MODE: ScreenGame customerCount, Customers served=5 transition in gameState diagram.

- UR_RECIPES, FR_RECIPES, NFR_SALAD, NFR_BURGER: Recipes enum.

- UR_COOKING_STATIONS: CookingStation class.

- UR_REPUTATION_POINTS: ScreenGame reputationPoints field.

- UR_PANTRY, FR_PANTRY, NFR_RUNNING_OUT_OF_INGREDIENTS: PantryBox class, each only has a single Ingredient type.

- UR_TIME_AT_END, NFR_TIME_TAKEN: ScreenGame gameTimer and ScreenEndGame time fields.

- UR_LEARNABILITY, FR_TUTORIAL, FR_CONTROLS: WindowGuide class.

- UR_MAIN_MENU: ScreenMenu class.

- UR_PAUSE: WindowPause class.

- FR_COOK_SWITCH, NFR_MOVING_COOKS: switchCooks() methods in both Cook class and Cook state diagram.

- FR_COOK_MOVE: move() method in Cook, MoveCharacter state in Cook state diagram.

- FR_COOK_INTERACT, FR_COOKING_STATION, FR_RECEIVING_INGREDIENTS, NFR_RANGE_OF_COOKING_STATION, NFR_DISTANCE_TO_PANTRY: canInteract() method with Cook parameter, CheckInteraction state in cook state diagram.

- FR_INGREDIENT_STACK: Cook stack field.

- FR_ORDER_OPTIONS: Recipes enum

- FR_TIMER, NFR_CUSTOMERS_WAITING: Customer timer field

- FR_LEAVE: Customer leaveCounter() method, LeavesCounter in customer state diagram

- FR_START_REPUTATION_POINTS: ScreenGame reputationPoints

- FR_LOSING_REPUTATION_POINTS: Customer lostReputation() method, RemovesReputation state in customer state diagram.

- FR_ZERO_REPUTATION_POINTS: Implemented in class3 changes as the gameEnd() method was added to the ScreenGame class. Shown as LoseGame state in game state diagram.

- FR_PANTRY: PantryBox output field of type Ingredient

- FR_CLOSING_THE_TUTORIAL: WindowClosable has a close button.

- FR_ON_MAIN_MENU, FR_MUTE_BUTTON_ON_MAIN_MENU: addButtons() method in ScreenMenu.

- FR_PAUSE_BUTTON: setupUI() method in ScreenGame class.

- FR_PAUSE_MENU: pauseWindow field in ScreenGame class.

- NFR_MOVEMENT_REQUIREMENTS: moveSpeed field in Cook class.

- NFR_MAX_CUSTOMERS_WAITING: class3 change - PiazzaMap counterCount field, allowing for 3 Customers at one time.

- NFR_SCALABILITY: All screens have a viewport, shown in ScreenBase, allowing for easy scalability

- NFR_ACCESSIBLE_TUTORIAL: WindowPause instances and can open WindowGuide.

- NFR_LENGTH_OF_GAME, NFR_TIME_TO_PREPARE: InteractionStation prepTimer and customer waitTimer fields result in an easily adjustable approximate game time.