Architecture

Assessment 2

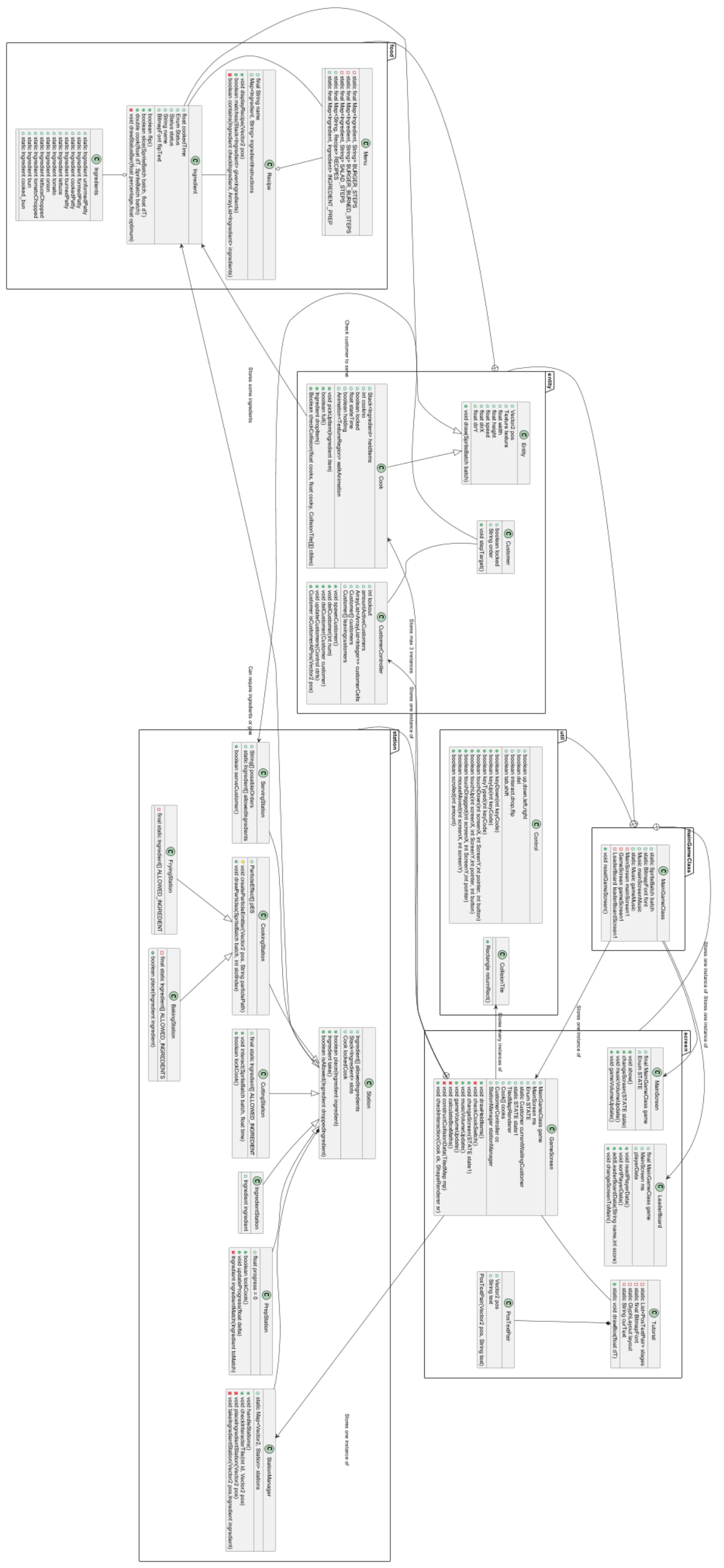
Group 3:

Ben Howard <bh1219@york.ac.uk>
Cai Hughes <cabh500@york.ac.uk>
Harry Richardson <hr1040@york.ac.uk>
Ivan Ndahiro <in597@york.ac.uk>
James Sutton <jis509@york.ac.uk>
Yuzhao Liu <yl5164@york.ac.uk>

Group 6:

Igor Smollinski <is942@york.ac.uk>
Pranshu Dhungana <pd861@york.ac.uk>
Zack Tyler-Kyle <ztk500@york.ac.uk>
Phoebe Russell <pbr508@york.ac.uk>
Sanjna Srinivasan <ss3264@york.ac.uk>
Sam Savery <sgs527@york.ac.uk>
Ewan Hutcheson <@eh1776york.ac.uk>

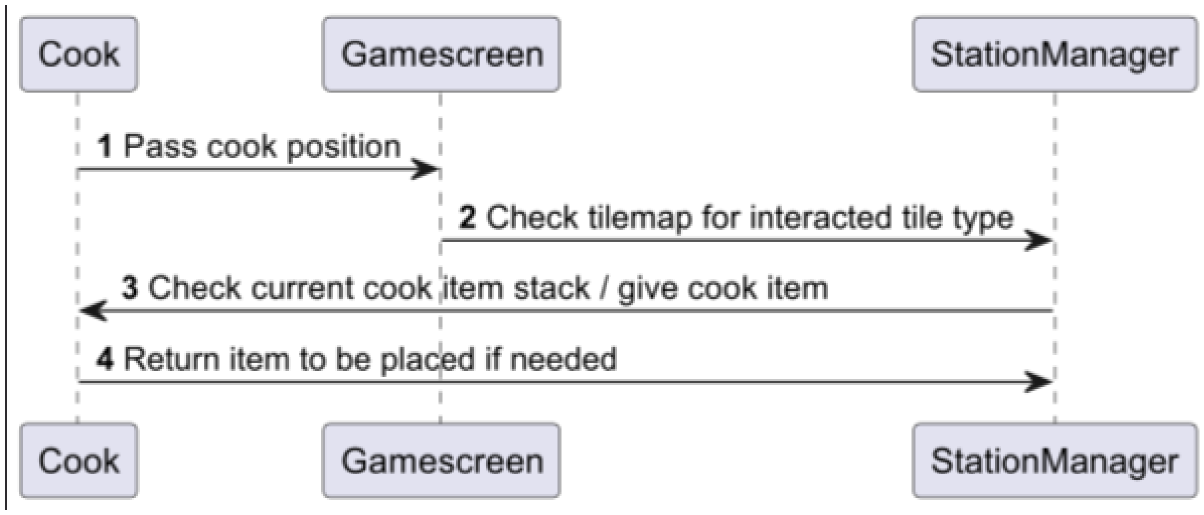
Class structure diagram - Generated using PlantUML



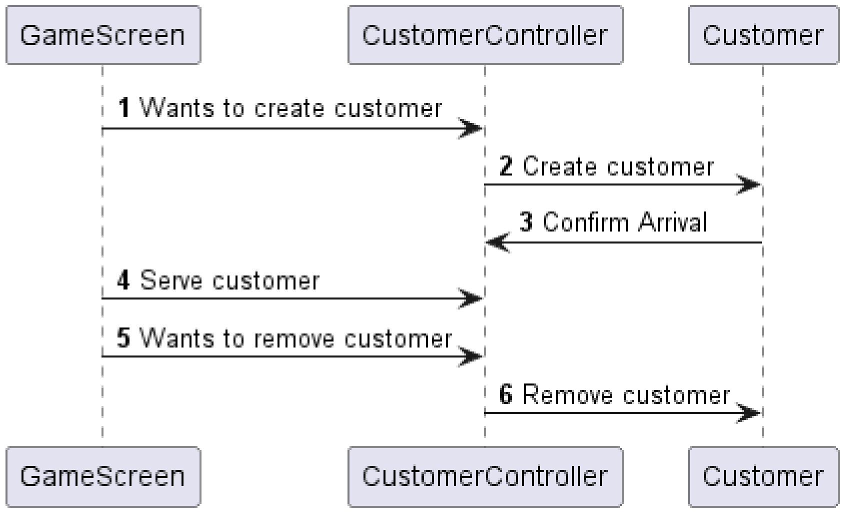
[1]

¹ Full Resolution Image: https://drive.google.com/file/d/1gdHToT4J1ciCABP7xcGOS8B-CiJ0luZ8/view?usp=sharing

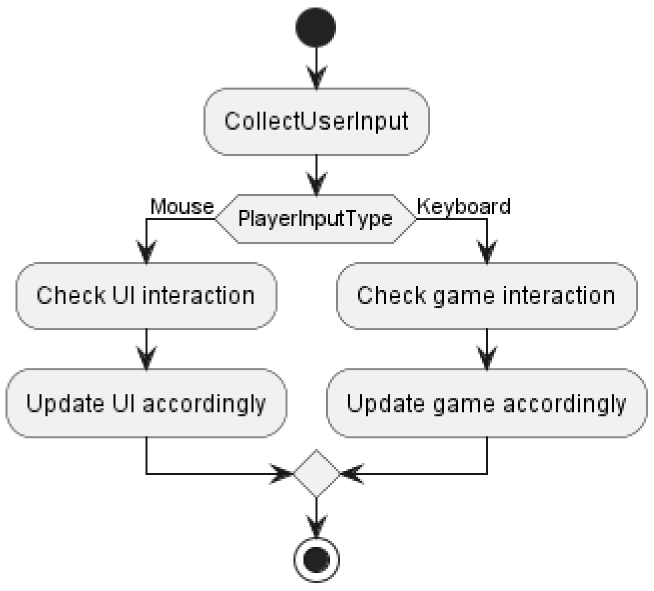
PlantUML sequence diagram for player interactions


Cook
Gamescreen
StationManager
1 Pass cook position
2 Check tilemap for interacted tile type
3 Check current cook item stack / give cook item
4 Return item to be placed if needed
Cook
Gamescreen
StationManager

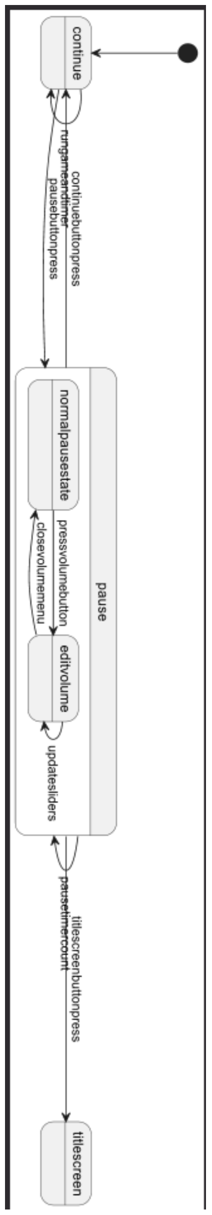
PlantUML sequence diagram for customer interactions

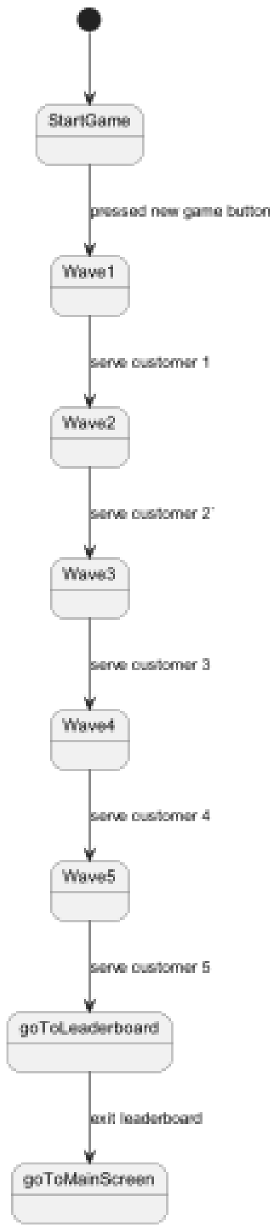


PlantUML Activity diagram for user input handling



PlantUML state diagram for gamewindow + PlantUML state diagram for gameplay state





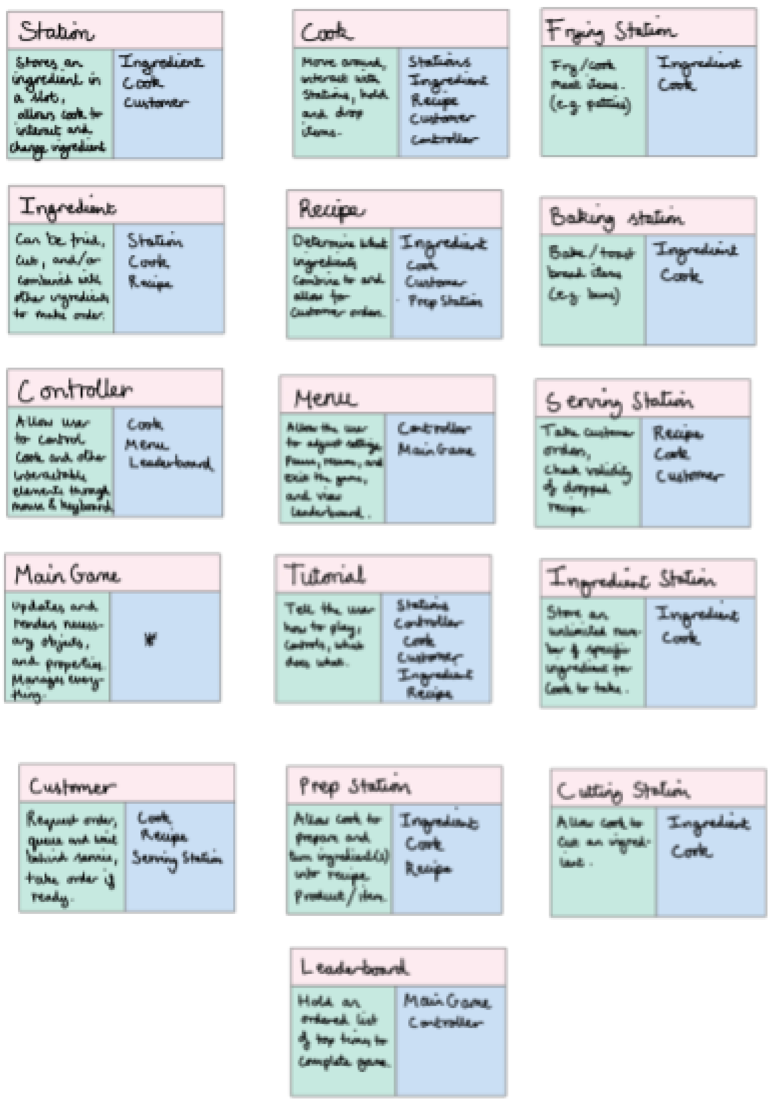
Initially, we looked at the user requirements and from this, we developed a list of potential classes and structures we would need in order to fulfil them. This included linking how the user would interact with the system and how it would respond. For instance, the user would use a keyboard's controls (FR_COOK_CONTROLLER) to move the cooks and perform the various actions. Additionally, they would have to use the mouse cursor to navigate the menu. We made the assumption that most users would be familiar with the typical WASD controls since the majority of games utilise these controls. We also accounted for the closeness of keys, for instance the keys q, e, tab, and shift are quite accessible from the standard position of the left hand on the keyboard. This would meet the UR_CONTROL_SYSTEM, UR_UX user requirement and NFR_OPERABILITY non-functional requirement..
Another consideration that was important to the architecture of our system was similarities between functions. For instance, all stations would behave in a similar manner with the player controlled cook being able to take and drop items when appropriate from these stations. Similarly, connections were made between cooks, customers and ingredients since all would require similar properties of a position, texture, dimensions (width and height), and motion. Consequently, it would make sense for likewise objects to inherit from a shared parent. We developed CRC cards to make this clearer (See Figure 1).

One key functionality which would affect the overall feel to the game would be how actions would be performed on stations and consequently ingredients. We decided that the best course of action would be for all ingredients to have an internal 'cookTime' and 'slices' state which would then be updated if the ingredient was on the appropriate station with the corresponding action being performed. This would meet the FR_COOK_ACTIONS, FR_CUTTING, FR_FRYING functional requirements. Additionally, one gameplay differentiation between the two would be that cooking wouldn't need the cook to be at the station, only there to initiate it, whereas if the user wanted the cook to cut, they would have to lock them at that station. This made more sense since you could quite easily leave something in a pan to cook while slicing is an action directly performed by a person. Additionally, if the user had moved the cook(s) away from the station and for some time, they'd be punished for leaving the ingredient(s) unattended.

In the case of the stations, each would have a number of slots, allowed ingredients, and then appropriate methods to deal with entity interactions. This allowed for flexibility when creating a new station and preventing the user mistakenly placing the wrong ingredient on a station. When the customer arrives at the service station, the user would have to guide one of the cooks to that station to get the order and only when it is placed on the station will the customer leave.

As the project evolved over time, several changes from the initial design became apparent. One such change was in the tutorial menu and the other screens (main menu, pause menu). It felt more natural for the user to be guided through the game, to the exact locations of the stations, when they first played instead of having the tutorial on just another screen with different images. This lets the user understand where the locations are without having to remember each location. Over time, the code in our MainGameClass increased dramatically and it became apparent that we would have to separate certain components and functionalities into different classes. This was especially the case when separating the main game code from the menu and other UI elements.

For creating recipes and menu items, a more modular system felt natural for our implementation. This included individual classes for ingredients and recipes along with a menu and list of ingredients to hold all final recipes (UR_RECIPES) and ingredients respectively. Then, through a station class, we could specify which ingredients the station takes or provides (depending on the type of station - UR_PANTRY_STATION, UR_ITEM_STATION).



Station	
Stores an ingredient in a slot, allows cook to interact and change ingredient	Ingredient Cook Customer

Cook	
Move around, interact with Stations, hold and drop items.	Stations Ingredient Recipe Customer Controller

Frying Station	
Fry/cook meat items. (e.g. patties)	Ingredient Cook

Ingredient	
Can be fried, cut, and/or combined with other ingredients to make order.	Station Cook Recipe

Recipe	
Determine what ingredients combine to and allow for customer orders.	Ingredient Cook Customer Prep Station

Baking station	
Bake/toast bread items (e.g. buns)	Ingredient Cook

Controller	
Allow user to control Cook and other interactable elements through mouse & keyboard.	Cook Menu Leaderboard

Menu	
Allow the user to adjust settings, Pause, resume, and exit the game, and view leaderboard.	Controller Main Game

Serving Station	
Take customer orders, Check validity of dropped recipe.	Recipe Cook Customer

Main Game	
Updates and renders necessary objects, and properties. Manages everything.	*

Tutorial	
Tell the user how to play, Controls, what does what.	Stations Controller Cook Customer Ingredient Recipe

Ingredient Station	
Store an unlimited number of specific ingredient for Cook to take.	Ingredient Cook

Customer	
Request order, queue and wait behind service, take order if ready.	Cook Recipe Serving Station

Prep Station	
Allow cook to prepare and turn ingredient(s) into recipe product/item.	Ingredient Cook Recipe

Cutting Station	
Allow cook to cut an ingredient.	Ingredient Cook

Leaderboard	
Hold an ordered list of top times to complete game.	Main Game Controller

Figure 1.

For some of the user requirements, we developed a table below with a brief description on how they would be implemented:

UR_SCENARIO_MODE	Our customers will arrive one-by-one and the game will finish when all have been served (currentWave >= MAX_WAVE).
UR_CONTROL_SYSTEM	A control class will contain boolean properties for every action which will be true when the key is activated.
UR_ITEMS	The cook will have a stack of held ingredients which can be dropped and added to through station interactions.
UR_DEMANDS	If the service station is interacted with by a customer, a demand will be made (which is an instance of recipe).
UR_NUMBER_OF_CUSTOMERS	There will be a global constant for the max number of customers (like a wave system).
UR_COOKS	Two instances of the cook class shall be created and rendered with a current cook being controlled directly.
UR_TOOLTIP	When approaching an intractable station, the appropriate key will be shown (e.g. 'f' to flip)..
UR_LEADERBORAD	When the game has been finished, the time will be uploaded to a leaderboard and sorted through that class.

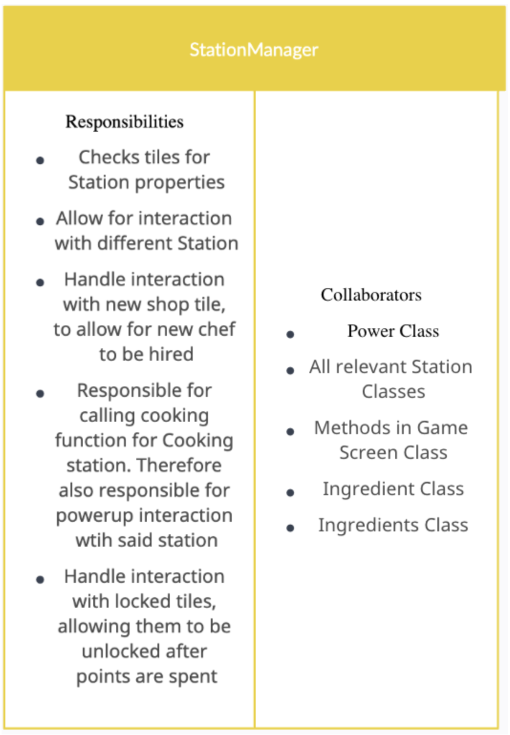
Team 6 Architecture Changes

The architecture has not been changed unless it has been stated below:

CRC cards for stationManager

We kept a similar structure as outlined in the initial architecture documentation. However we started by creating new CRC cards for changes in responsibilities and new classes.

First we started with how StationManager's responsibilities had to be changed. In order to reflect those changes we made a CRC card outlining new responsibilities:



StationManager

Responsibilities	Collaborators
• Checks tiles for Station properties • Allow for interaction with different Station • Handle interaction with new shop tile, to allow for new chef to be hired • Responsible for calling cooking function for Cooking station. Therefore also responsible for powerup interaction wtih said station • Handle interaction with locked tiles, allowing them to be unlocked after points are spent	• Power Class • All relevant Station Classes • Methods in Game Screen Class • Ingredient Class • Ingredients Class

You can see the associations between the collaborators more clearly in this diagram:

https://neves6.github.io/PiazzaPanic2/uml/ManagerPower.png

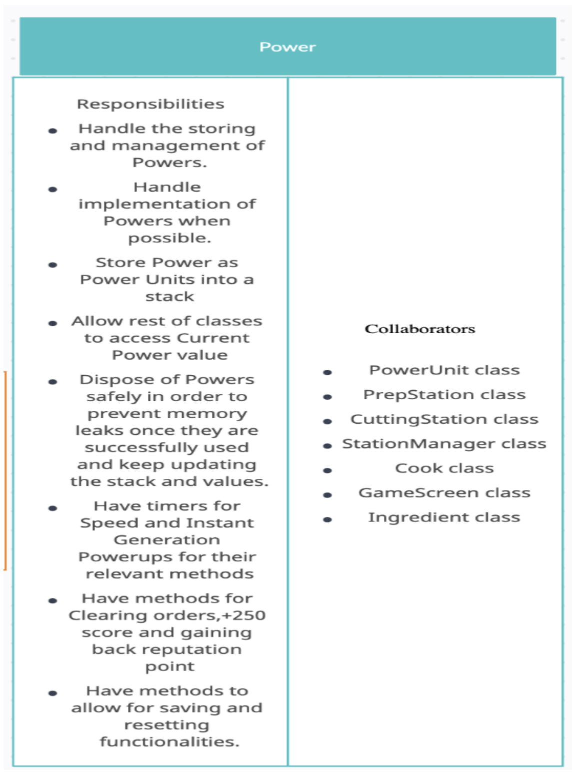
List requirements relating to the locked station and powers

- UR_LOCKED_RESOURCES
- FR_LOCKED_RESOURCE
- UR_POWER_UPS
- FR_POWERUP_GENERATION
- FR_INCREASE_SPEED
- FR_CLEAR_ORDER
- FR_INSTANT_ACTION_GENERATION
- FR_INCREASE_REPUTATION_POINT
- FR_ADD_POINTS
- NFR_POWERUP_TIMER

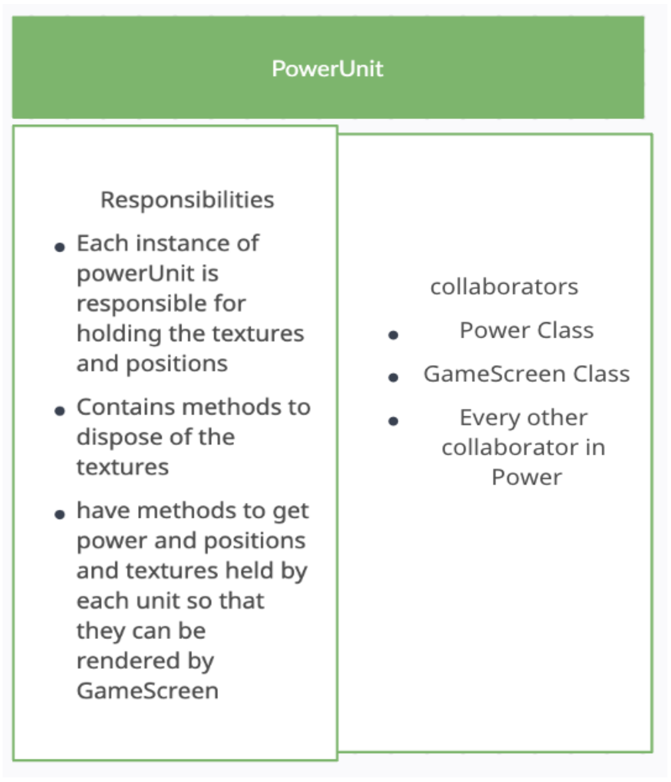
<u>CRC cards for Power and PowerUnit</u>

We had to create CRC cards and try to figure out how to go about implementing the new power up requirements.

We went with a novel approach,in order to make the code easily extensible and interchangeable. However before going into detail and talking about the sequence diagrams for power ups, here are CRC cards for what we wanted.



Power	
Responsibilities • Handle the storing and management of Powers. • Handle implementation of Powers when possible. • Store Power as Power Units into a stack • Allow rest of classes to access Current Power value • Dispose of Powers safely in order to prevent memory leaks once they are successfully used and keep updating the stack and values. • Have timers for Speed and Instant Generation Powerups for their relevant methods • Have methods for Clearing orders,+250 score and gaining back reputation point • Have methods to allow for saving and resetting functionalities.	Collaborators • PowerUnit class • PrepStation class • CuttingStation class • StationManager class • Cook class • GameScreen class • Ingredient class

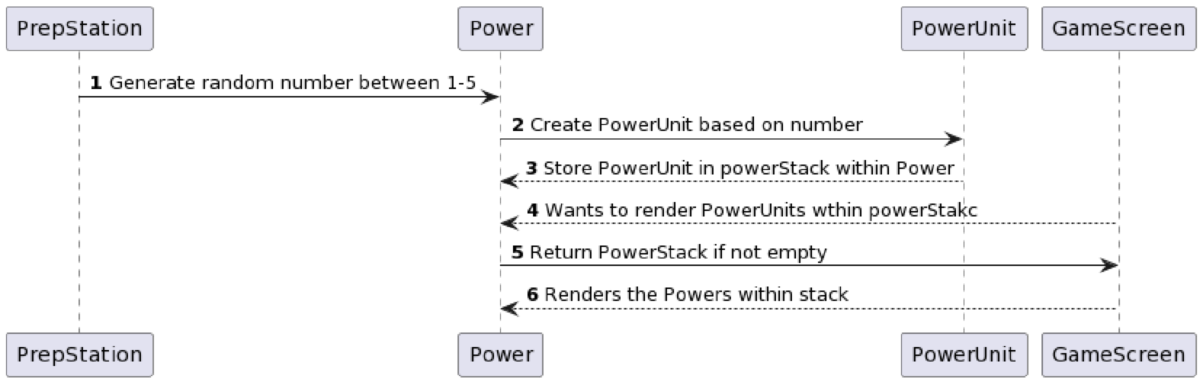


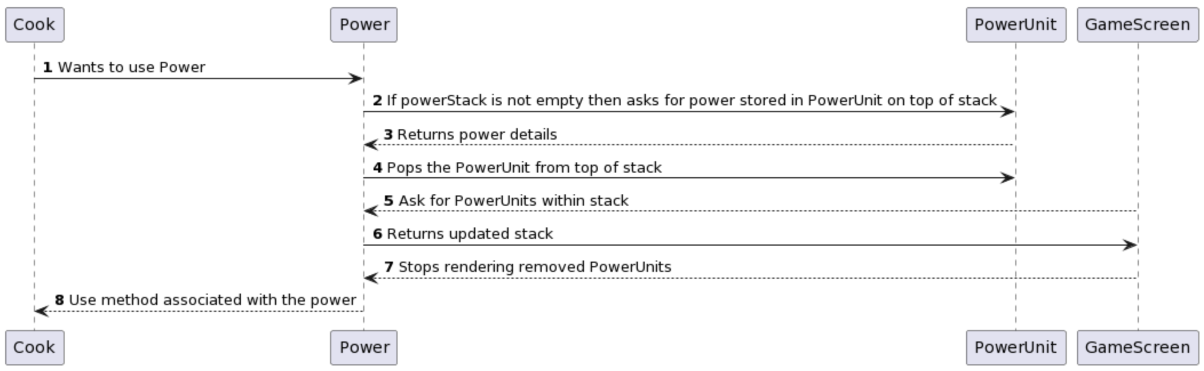
PowerUnit	
Responsibilities • Each instance of powerUnit is responsible for holding the textures and positions • Contains methods to dispose of the textures • have methods to get power and positions and textures held by each unit so that they can be rendered by GameScreen	collaborators • Power Class • GameScreen Class • Every other collaborator in Power

These CRC cards were used to implement the Power up requirements:
UR_POWER_UPS

- UR_POWERUP_GENERATION
- FR_INCREASE_SPEED
- FR_CLEAR_ORDER
- FR_INSTANT_ACTION_GENERATION
- FR_INCREASE_REPUTATION_POINT
- FR_ADD_POINTS

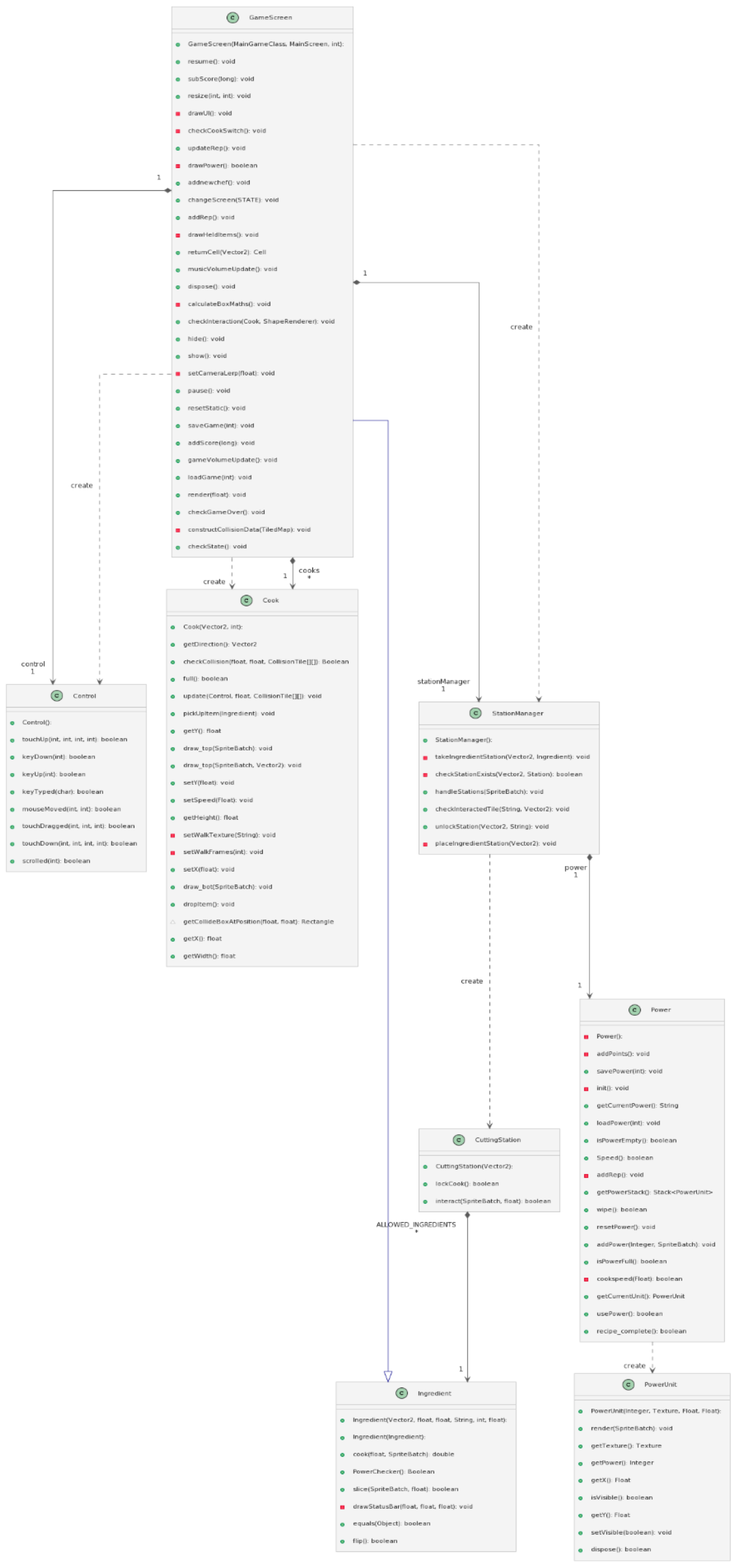
Sequence diagrams for Power up functionality


PrepStation
Power
PowerUnit
GameScreen
1 Generate random number between 1-5
2 Create PowerUnit based on number
3 Store PowerUnit in powerStack within Power
4 Wants to render PowerUnits wthin powerStakc
5 Return PowerStack if not empty
6 Renders the Powers within stack
PrepStation
Power
PowerUnit
GameScreen


Cook
Power
PowerUnit
GameScreen
1 Wants to use Power
2 If powerStack is not empty then asks for power stored in PowerUnit on top of stack
3 Returns power details
4 Pops the PowerUnit from top of stack
5 Ask for PowerUnits within stack
6 Returns updated stack
7 Stops rendering removed PowerUnits
8 Use method associated with the power
Cook
Power
PowerUnit
GameScreen

The implementations of the power varied depending on its function. We added methods into classes like cook to allow for speed to be changed and while some required just one method call to change something, others like Speed and Instant Generation required Timers and, Instant Generation was essentially just setting the CurrentPower variable to Instant and using that as a flag which was checked by Ingredient class with the powerChecker method.Where another flag would be set to skip the cooking process, since the Ingredient Class handled the methods pertaining to the cooking/cutting process. You can see these methods on the UML diagram.

<u>UML relationship diagram for Power and relevant collaborators after implementation</u>

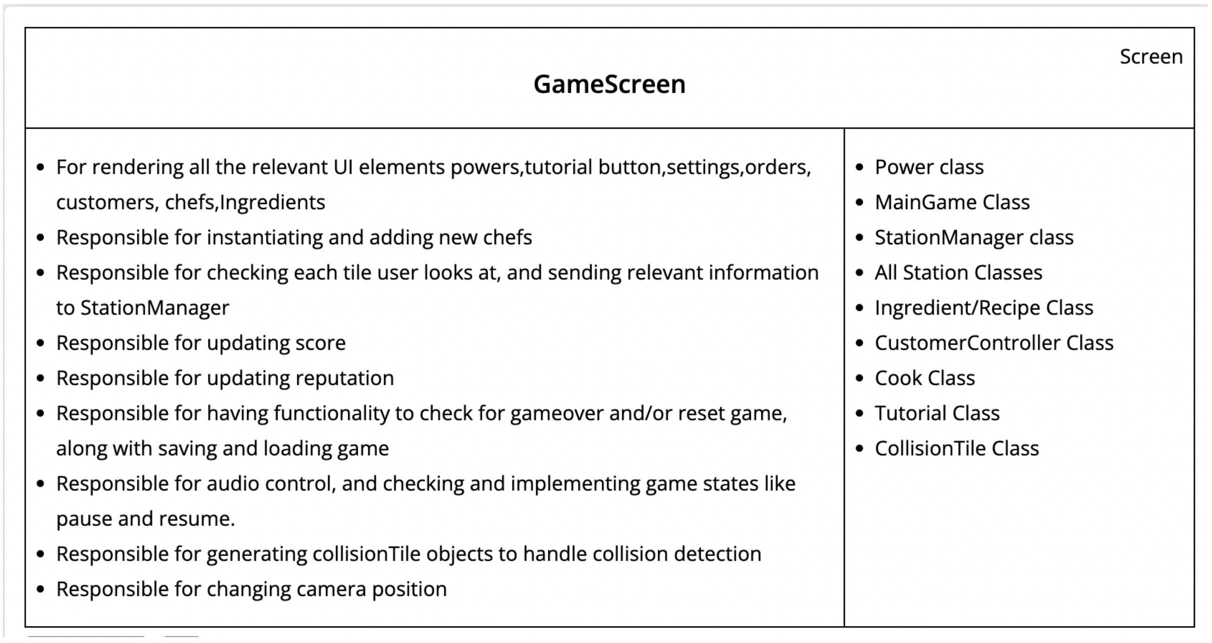

GameScreen
GameScreen(MainGameClass, MainScreen, int):
resume(): void
subScore(long): void
resize(int, int): void
drawUI(): void
checkCookSwitch(): void
updateRep(): void
drawPower(): boolean
addnewchef(): void
changeScreen(STATE): void
addRep(): void
drawHeldItems(): void
returnCell(Vector2): Cell
musicVolumeUpdate(): void
dispose(): void
calculateBoxMaths(): void
checkInteraction(Cook, ShapeRenderer): void
hide(): void
show(): void
setCameraLerp(float): void
pause(): void
resetStatic(): void
saveGame(int): void
addScore(long): void
gameVolumeUpdate(): void
loadGame(int): void
render(float): void
checkGameOver(): void
constructCollisionData(TiledMap): void
checkState(): void
1
1
create
create
create
1
cooks
*
control
1
Control
Control():
touchUp(int, int, int, int): boolean
keyDown(int): boolean
keyUp(int): boolean
keyTyped(char): boolean
mouseMoved(int, int): boolean
touchDragged(int, int, int): boolean
touchDown(int, int, int, int): boolean
scrolled(int): boolean
Cook
Cook(Vector2, int):
getDirection(): Vector2
checkCollision(float, float, CollisionTile[][]): Boolean
full(): boolean
update(Control, float, CollisionTile[][]): void
pickUpItem(Ingredient): void
getY(): float
draw_top(SpriteBatch): void
draw_top(SpriteBatch, Vector2): void
setY(float): void
setSpeed(Float): void
getHeight(): float
setWalkTexture(String): void
setWalkFrames(int): void
setX(float): void
draw_bot(SpriteBatch): void
dropItem(): void
getCollideBoxAtPosition(float, float): Rectangle
getX(): float
getWidth(): float
stationManager
1
StationManager
StationManager():
takeIngredientStation(Vector2, Ingredient): void
checkStationExists(Vector2, Station): boolean
handleStations(SpriteBatch): void
checkInteractedTile(String, Vector2): void
unlockStation(Vector2, String): void
placeIngredientStation(Vector2): void
power
1
create
1
Power
Power():
addPoints(): void
savePower(int): void
init(): void
getCurrentPower(): String
loadPower(int): void
isPowerEmpty(): boolean
Speed(): boolean
addRep(): void
getPowerStack(): Stack<PowerUnit>
wipe(): boolean
resetPower(): void
addPower(Integer, SpriteBatch): void
isPowerFull(): boolean
cookspeed(Float): boolean
getCurrentUnit(): PowerUnit
usePower(): boolean
recipe_complete(): boolean
CuttingStation
CuttingStation(Vector2):
lockCook(): boolean
interact(SpriteBatch, float): boolean
ALLOWED_INGREDIENTS
*
1
create
Ingredient
Ingredient(Vector2, float, float, String, int, float):
Ingredient(Ingredient):
cook(float, SpriteBatch): double
PowerChecker(): Boolean
slice(SpriteBatch, float): boolean
drawStatusBar(float, float, float): void
equals(Object): boolean
flip(): boolean
PowerUnit
PowerUnit(Integer, Texture, Float, Float):
render(SpriteBatch): void
getTexture(): Texture
getPower(): Integer
getX(): Float
isVisible(): boolean
getY(): Float
setVisible(boolean): void
dispose(): boolean

Locked Stations and Hiring chef

We choose to implement the locked station and hiring shop as just additional tiles with different properties. Only GameScreen and StationManager were responsible for handling this requirement. The previous architecture design seemed to make GameScreen have a lot of responsibilities. So we just decided to build on top of it's responsibilities to add more.
This is the CRC diagram for GameScreen that we used to develop what the requirements needed.

GameScreen CRC card

Ended up using a different website to generate these cards since the previous website since Lucid charts ended up requiring additional payment to create a lot more cards. The website I am using for generating CRC cards is now: https://echeung.me/crcmaker/ . It's more efficient since I do not need to worry about managing text boxes and shapes.



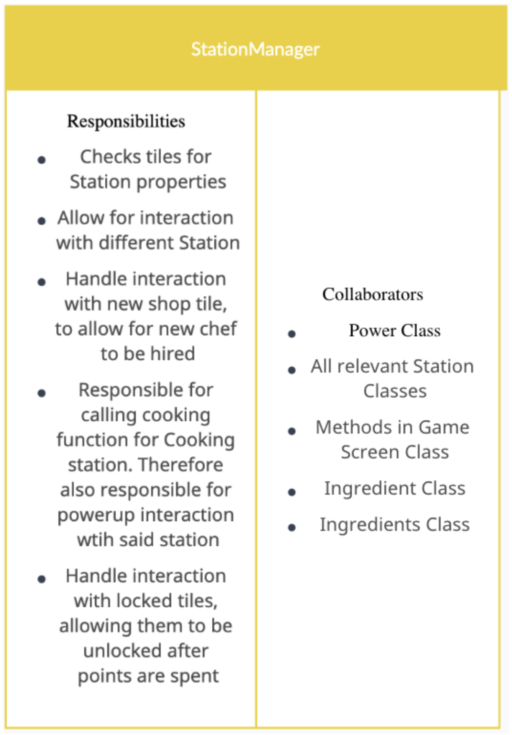
GameScreen	Screen
• For rendering all the relevant UI elements powers,tutorial button,settings,orders, customers, chefs,Ingredients • Responsible for instantiating and adding new chefs • Responsible for checking each tile user looks at, and sending relevant information to StationManager • Responsible for updating score • Responsible for updating reputation • Responsible for having functionality to check for gameover and/or reset game, along with saving and loading game • Responsible for audio control, and checking and implementing game states like pause and resume. • Responsible for generating collisionTile objects to handle collision detection • Responsible for changing camera position	• Power class • MainGame Class • StationManager class • All Station Classes • Ingredient/Recipe Class • CustomerController Class • Cook Class • Tutorial Class • CollisionTile Class

This was done in order to implement the following requirements:

- FR_POWERUP_GENERATION
- FR_INCREASE_SPEED
- FR_CLEAR_ORDER
- FR_INSTANT_ACTION_GENERATION
- FR_INCREASE_REPUTATION_POINT
- FR_ADD_POINTS
- UR_SUPPORT_DIFFICULTY
- FR_EASY_MODE

- FR_MODERATE_MODE
- FR_HARD_MODE
- UR_SAVE_STATE
- FR_SAVE_STATE

You can see how station manager's responsibilities were related to GameScreens:

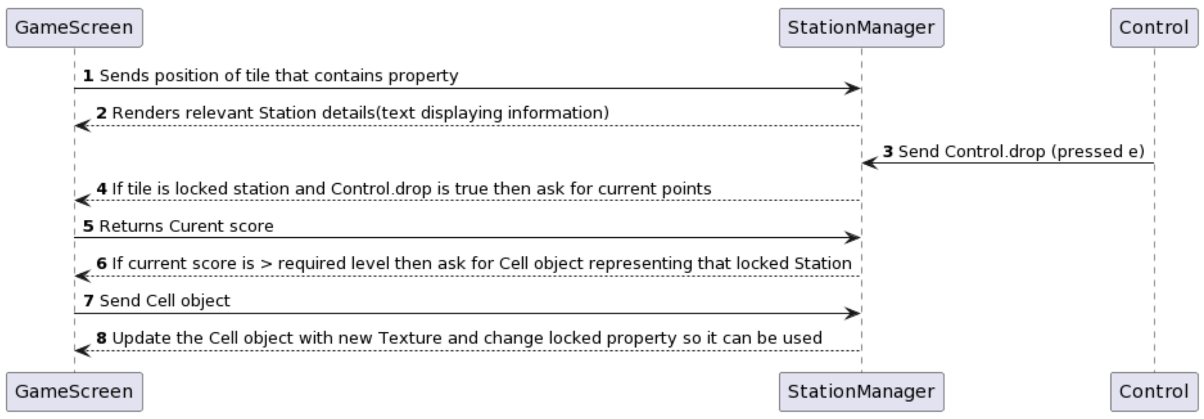


StationManager	
Responsibilities • Checks tiles for Station properties • Allow for interaction with different Station • Handle interaction with new shop tile, to allow for new chef to be hired • Responsible for calling cooking function for Cooking station. Therefore also responsible for powerup interaction wtih said station • Handle interaction with locked tiles, allowing them to be unlocked after points are spent	**Collaborators** • Power Class • All relevant Station Classes • Methods in Game Screen Class • Ingredient Class • Ingredients Class

In this case just focusing on interacting with the shop tile and locked station tiles. Along with enabling the user to spend points.

Sequence diagram for locked stations

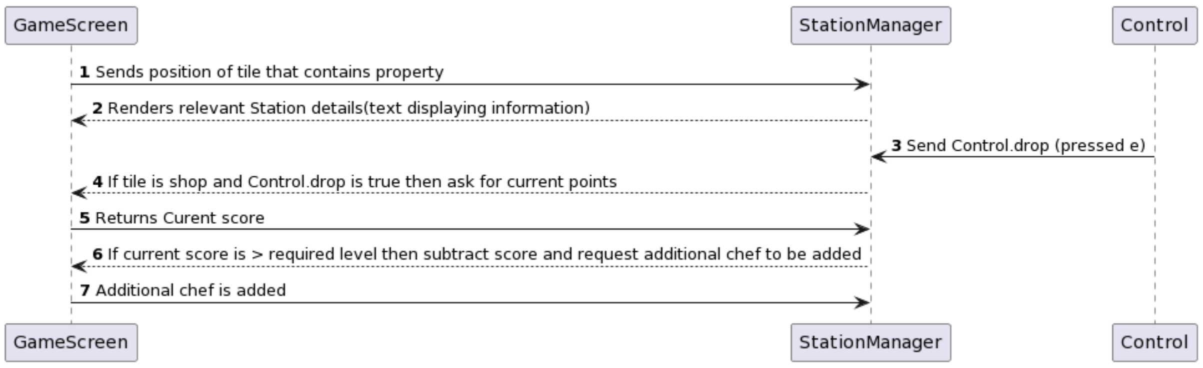
Sequence diagrams are generated using plantuml, We found this to be the simplest language to use available. We did not wish to do anything particularly complicated and this allowed us to just use plain text to generate sequence diagrams without worrying about layouts and shapes.


GameScreen
StationManager
Control
1 Sends position of tile that contains property
2 Renders relevant Station details(text displaying information)
3 Send Control.drop (pressed e)
4 If tile is locked station and Control.drop is true then ask for current points
5 Returns Curent score
6 If current score is > required level then ask for Cell object representing that locked Station
7 Send Cell object
8 Update the Cell object with new Texture and change locked property so it can be used
GameScreen
StationManager
Control

As can be seen there is a lot of communication between the two classes.Which we used to later implement our approach. After further research we had to implement the sequence diagram slightly differently in terms of updating the Cell object. We used an array of textures, that stored relevant station textures when they are not grey. We created tiles where the stations were greyed out to represent the locked stations. We just added a property to these stations in a similar way to the rest of the code, so the property was something like {Station:locked_frying}. The Cell objects and position were sent by GameScreen by using the checkInteractedTile method of StationManager class and StationManager would use those properties to implement the locked stations.Displaying a text with instruction on cost and to press e in order to unlock Station.

If e is pressed and Control.drop is true and GameScreen.score is above a certain threshold then we buy station using the unlockStation method in station manager, which would use the gameScreen.returnCell method to return cell at the position where interaction is occurring.Then update Cell using hashmap of texture, depending on station property we want e.g Station:Frying instead Station:locked_frying.

Sequence Diagrams for hire chef

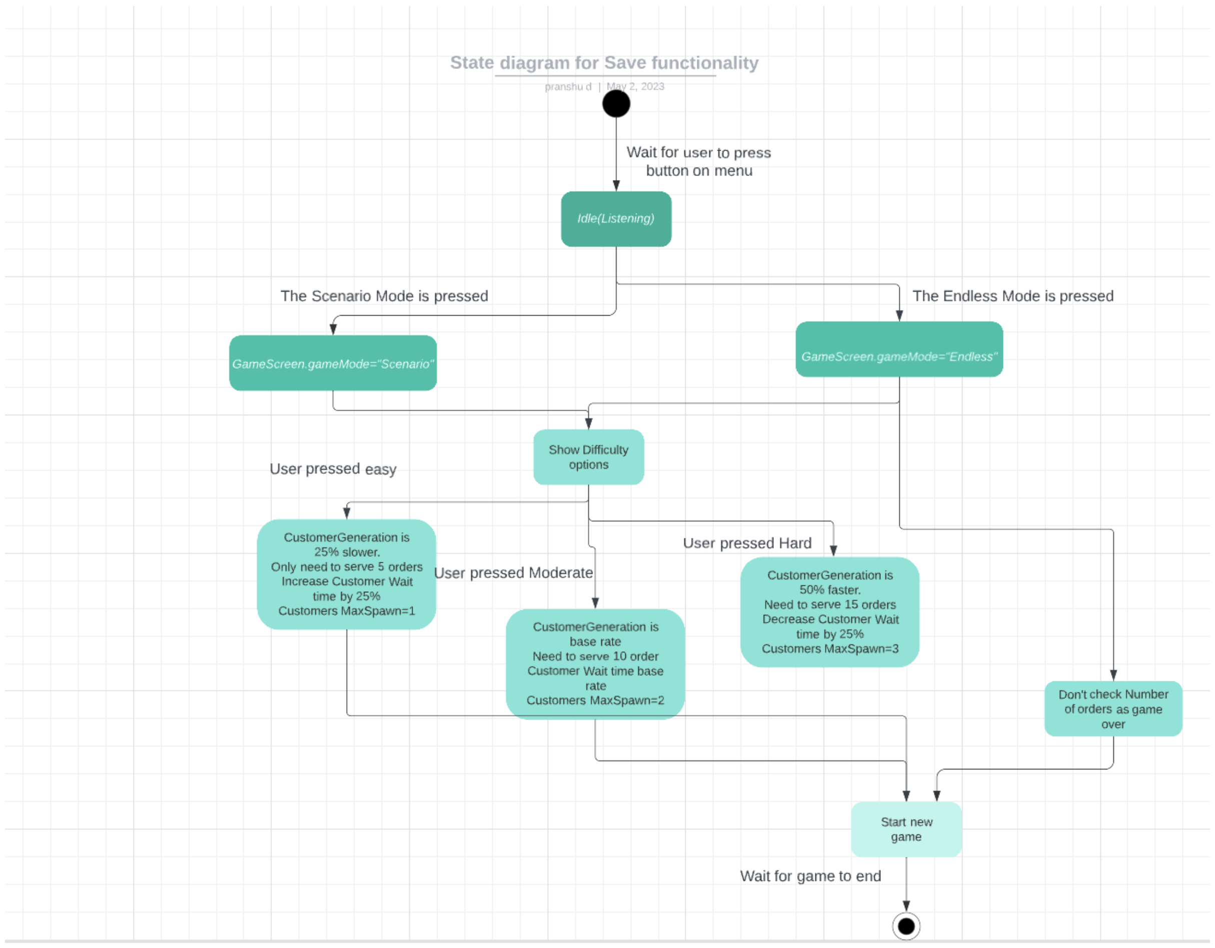

GameScreen
StationManager
Control
1 Sends position of tile that contains property
2 Renders relevant Station details(text displaying information)
3 Send Control.drop (pressed e)
4 If tile is shop and Control.drop is true then ask for current points
5 Returns Curent score
6 If current score is > required level then subtract score and request additional chef to be added
7 Additional chef is added
GameScreen
StationManager
Control

This sequence diagram was implemented by using the addChef method in GameScreen which is pretty much the main difference.

UML relationship diagram showing relationships between GameScreen and other relevant classes and collaborators

https://neves6.github.io/PiazzaPanic2/uml/GameScreenBig.png

State Diagrams for difficulty modes and different modes

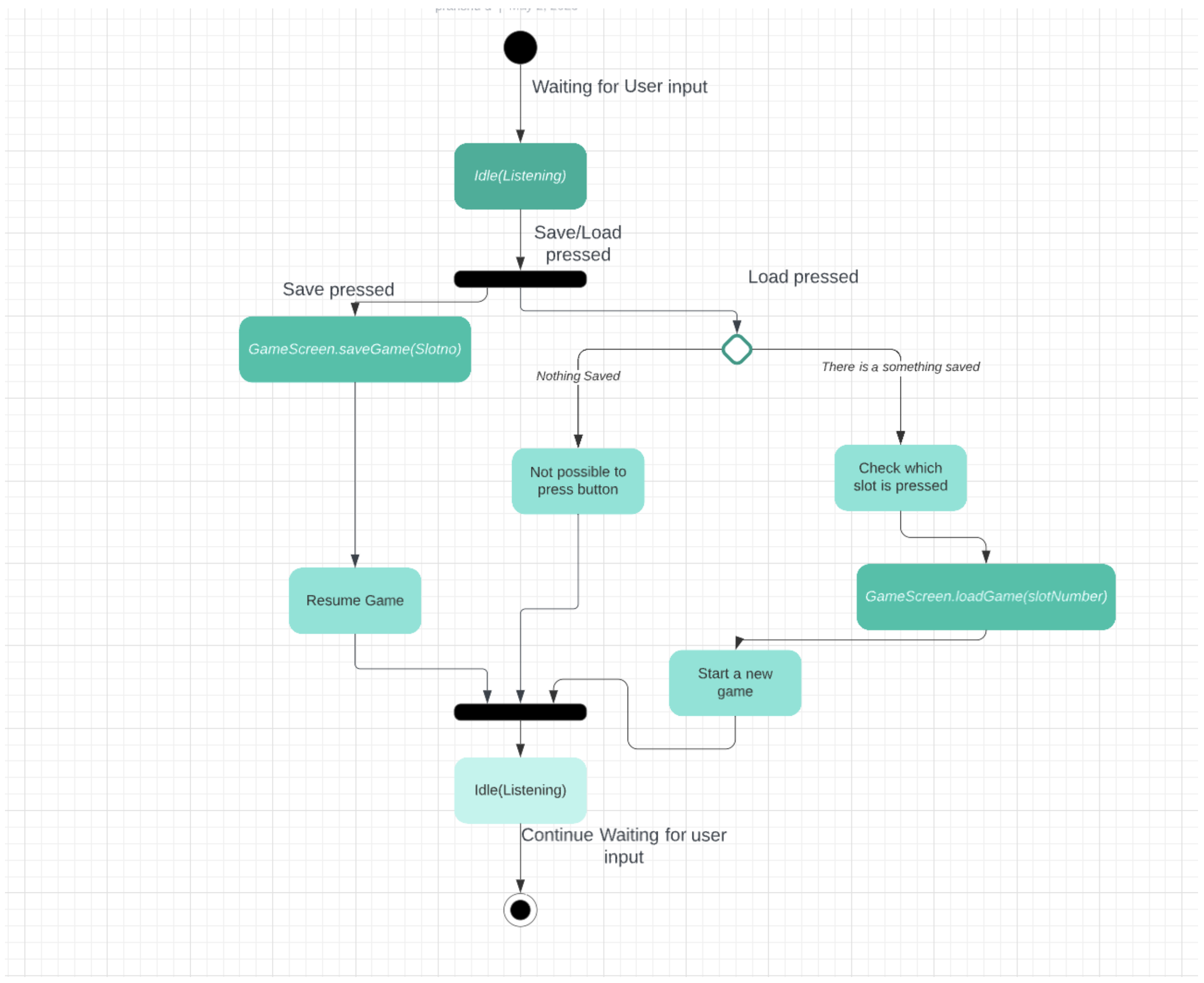

State diagram for Save functionality
pranshu d | May 2, 2023
Wait for user to press button on menu
Idle(Listening)
The Scenario Mode is pressed
The Endless Mode is pressed
GameScreen.gameMode="Scenario"
GameScreen.gameMode="Endless"
Show Difficulty options
User pressed easy
User pressed Moderate
User pressed Hard
CustomerGeneration is 25% slower. Only need to serve 5 orders Increase Customer Wait time by 25% Customers MaxSpawn=1
CustomerGeneration is base rate Need to serve 10 order Customer Wait time base rate Customers MaxSpawn=2
CustomerGeneration is 50% faster. Need to serve 15 orders Decrease Customer Wait time by 25% Customers MaxSpawn=3
Don't check Number of orders as game over
Start new game
Wait for game to end

This state diagram represents what occurs when User is on the Main Menu screen and chooses to start a game.All these buttons are going to be listeners waiting for input. We decided to just apply the same difference in configuration between easy , moderate and hard mode with endless mode as well. The only difference being , which can be seen on the State Diagram that the Endless mode won't check for Number of Orders as game over state.

Requirements that this is linked towards:

-UR_ENDLESS_MODE
-UR_SCENARIO_MODE
-UR_SUPPORT_DIFFICULTY
-FR_EASY_MODE
-FR_MODERATE_MODE
-FR_HARD_MODE

State Diagram for save/loading


Waiting for User input
Idle(Listening)
Save/Load pressed
Save pressed
Load pressed
GameScreen.saveGame(Slotno)
Nothing Saved
There is a something saved
Not possible to press button
Check which slot is pressed
Resume Game
GameScreen.loadGame(slotNumber)
Start a new game
Idle(Listening)
Continue Waiting for user input

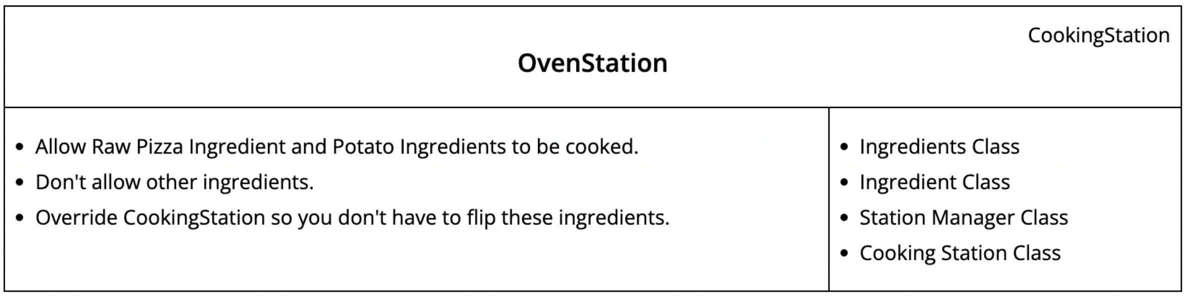
We chose to implement this functionality in the end through using listeners with our buttons that get rendered by GameScreen and creating save methods on classes storing values outside the GameScreen class. Such as the Power and CustomerController class which have methods to save their respective values.Also this state diagram is referring to within Game Screen. Menu will just have the load buttons.

This helped implement these save/load requirements:

-UR_SAVE_STATE
-FR_SAVE_STATE
-UR_LOAD_STATE
-FR_LOAD_STATE

<u>**Rest of UML/CRC diagrams:**</u>

We need to add an Oven station to facilitate the cooking of Pizza and Jacket Potato Recipes.Heres a CRC :



OvenStation	CookingStation
• Allow Raw Pizza Ingredient and Potato Ingredients to be cooked. • Don't allow other ingredients. • Override CookingStation so you don't have to flip these ingredients.	• Ingredients Class • Ingredient Class • Station Manager Class • Cooking Station Class

It used functionality already within the code and architecture. Simply inherited from CookingStation and, the Station Manager used the Cook method within the Ingredient Class.
It used functionality that was already there since Stage 1, and so there was not much need to create a sequence diagram.
<u>This links towards these Requirements:</u>

-UR_COOKING_STATIONS
-UR_ITEM_STATION
-UR_PANTRY_STATION
-UR_COUNTER
-FR_COOK_ACTIONS
-FR_ACTION
-FR_PIZZA
-FR_BURGER
-FR_SALAD
-FR_JACKET_POTATO

OrderCard CRC:

We ended up using OrderCards because previously each order was tied to the customer.This was causing it hard to generate a lot of customers.We decided to refactor it and moreover we made it easier to implement the reputation system since each OrderCard unit contains various details about the order and texture and the expiry which ties into the reputation system.

OrderCard	
• Have information about the texture and positions of orders so that orders can be displayed by GameScreen class • Have details about the expiry time for an order • Handling determining whether the order has expired in order to result in lower reputation points by GameScreen Class	• GameScreen Class • ServingStation Class

Food UML Relationship Diagram and GameScreen

https://neves6.github.io/PiazzaPanic2/uml/FoodGame.png

We didn’t really change the underlying architecture except for the OrderCard.Just adding new recipes and added new ingredients to facilitate that. There was not much need to create a CRC card for the changes we planned to do here. Since there was not really much change to overarching responsibilities. Menu still helped facilitate creation of Recipes and transformation of ingredients with PrepStation. Ingredients still just contained Ingredient objects that were used within the Recipes.

The only thing was adding new Recipes and ingredients for Jacket Potato and Pizza.

Stations CRC cards:

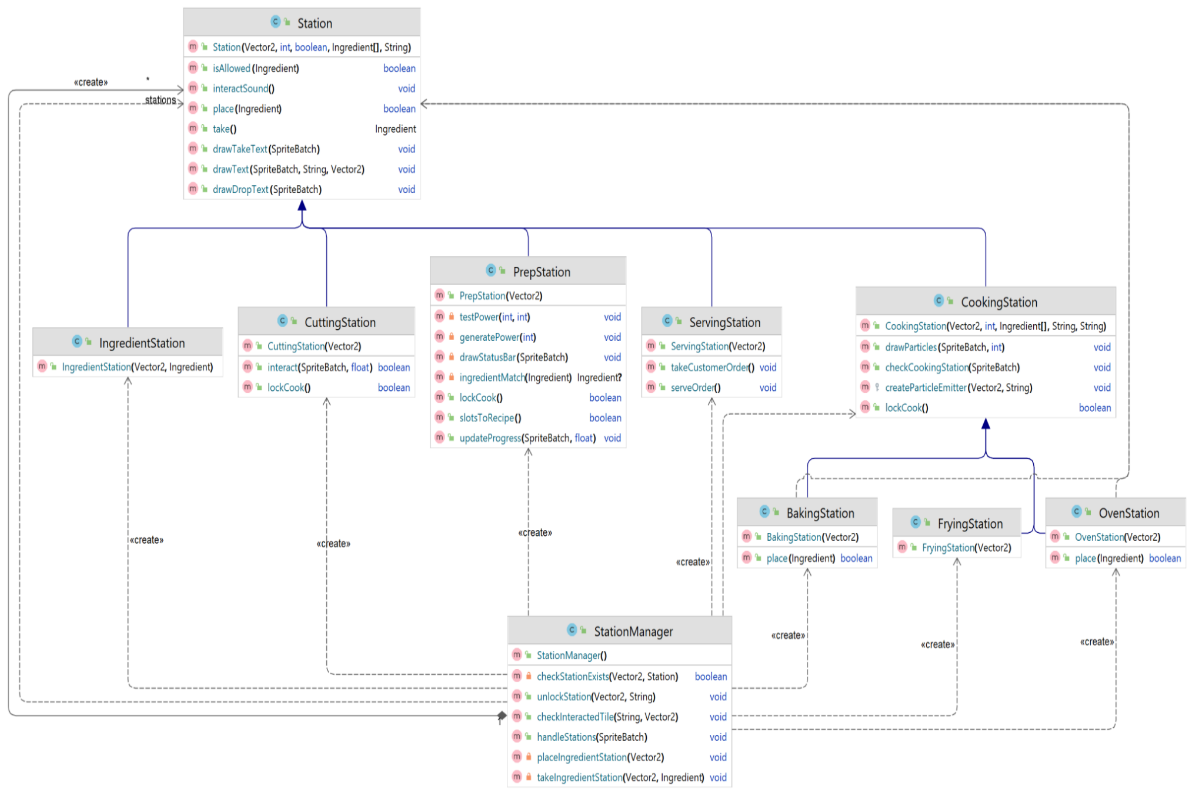
There was a significant change to some of the Stations here.We need to score points based on serving dishes, and generate new powers which we did as shown in Sequence diagram through the PrepStation.Therefore needed to create new crc cards for the Serving Station and PrepStation.

PrepStation	Station
• Allow transformation of Ingredients and formation of Recipes • Generate powers randomly with different percentages	• StationManager Class • Power Class • GameScreen Class • Menu Class

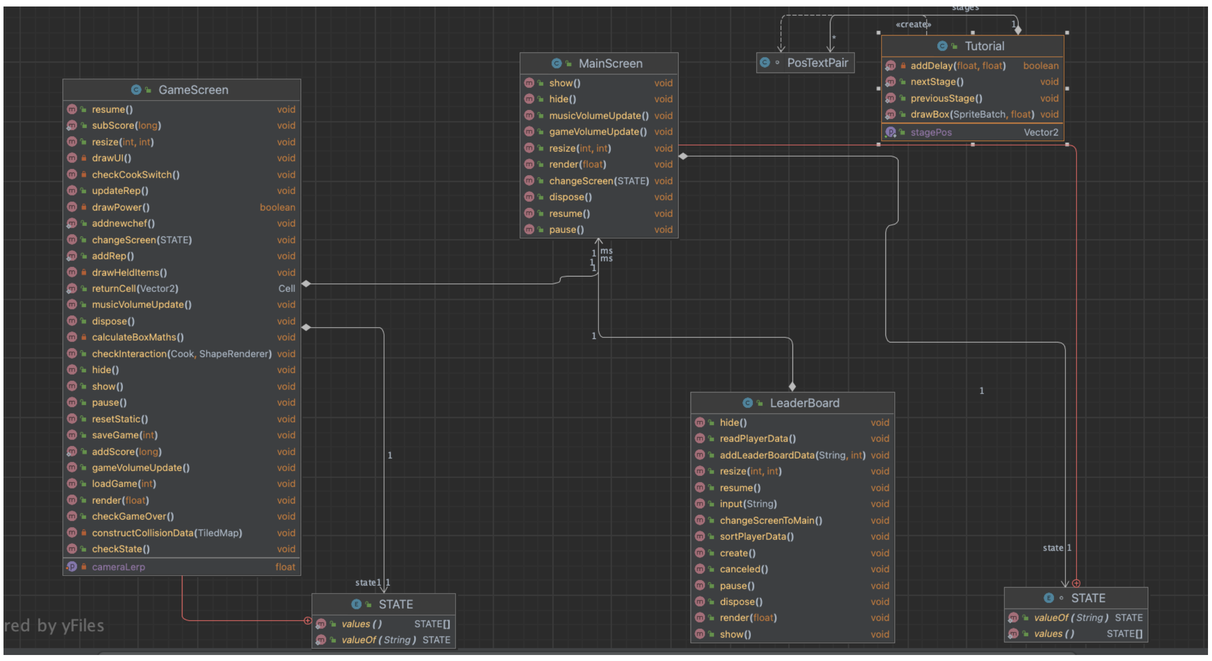
ServingStation	Station
• Add score for each order successfully served • Keep track of waiting Customers • Keep track of what order needs to be served where • Add new orders to GameScreen.orderCards	• GameScreen class • Menu class • Recipe and Ingredients class • StationManager class

We did not find it particularly necessary to use a sequence diagram for the new functionalities as well since it’s fairly straightforward. Most of their functionalities are done with methods within the class i.e PrepStation generates random power using its own methods. Which can be seen in the final Stations package UML diagram. This functionality was already showcased within earlier sequence diagrams.

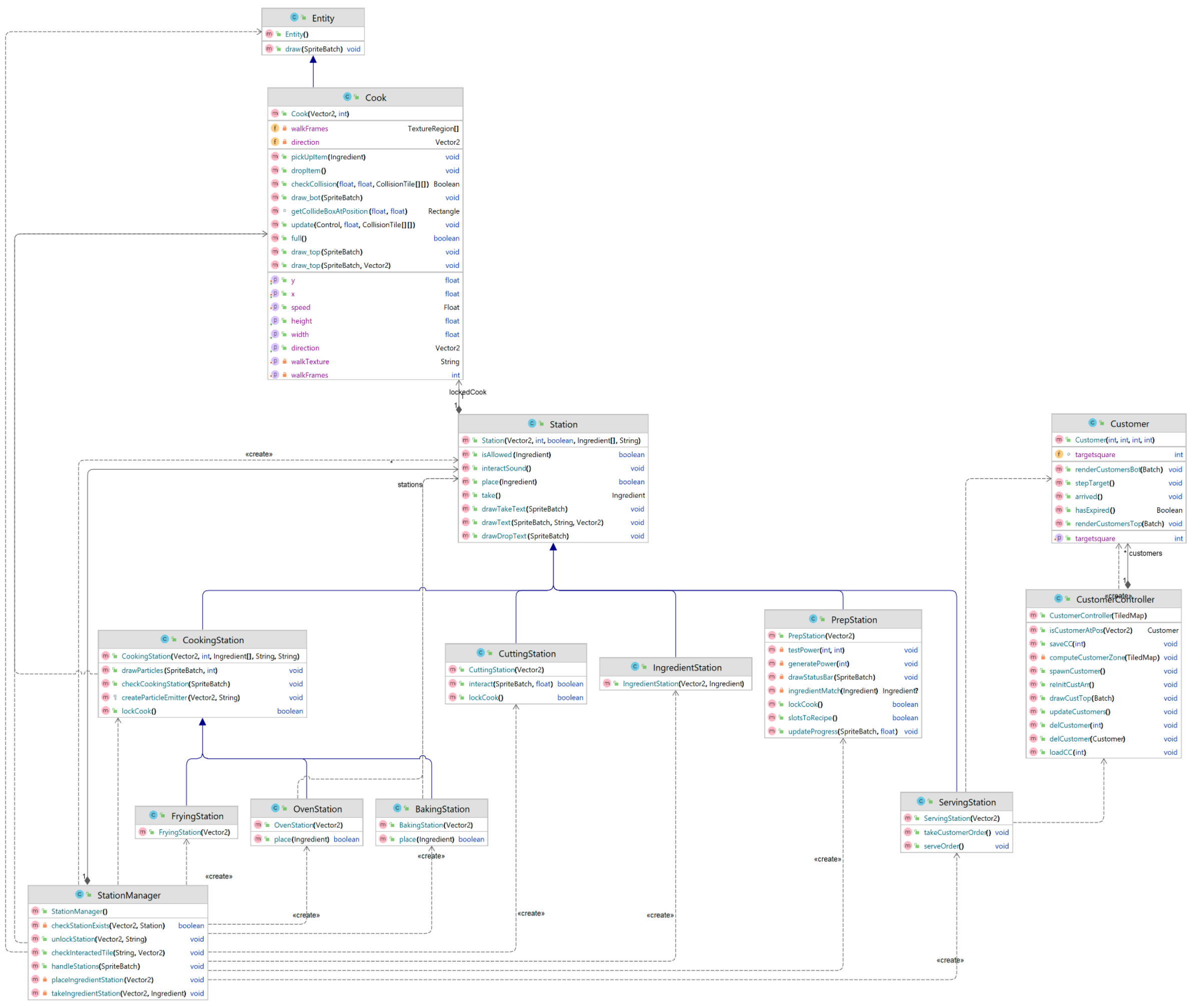
Stations package UML diagram


Station
Station(Vector2, int, boolean, Ingredient[], String)
isAllowed(Ingredient) boolean
interactSound() void
place(Ingredient) boolean
take() Ingredient
drawTakeText(SpriteBatch) void
drawText(SpriteBatch, String, Vector2) void
drawDropText(SpriteBatch) void
«create»
*
stations
IngredientStation
IngredientStation(Vector2, Ingredient)
CuttingStation
CuttingStation(Vector2)
interact(SpriteBatch, float) boolean
lockCook() boolean
PrepStation
PrepStation(Vector2)
testPower(int, int) void
generatePower(int) void
drawStatusBar(SpriteBatch) void
ingredientMatch(Ingredient) Ingredient?
lockCook() boolean
slotsToRecipe() boolean
updateProgress(SpriteBatch, float) void
ServingStation
ServingStation(Vector2)
takeCustomerOrder() void
serveOrder() void
CookingStation
CookingStation(Vector2, int, Ingredient[], String, String)
drawParticles(SpriteBatch, int) void
checkCookingStation(SpriteBatch) void
createParticleEmitter(Vector2, String) void
lockCook() boolean
BakingStation
BakingStation(Vector2)
place(Ingredient) boolean
FryingStation
FryingStation(Vector2)
OvenStation
OvenStation(Vector2)
place(Ingredient) boolean
StationManager
StationManager()
checkStationExists(Vector2, Station) boolean
unlockStation(Vector2, String) void
checkInteractedTile(String, Vector2) void
handleStations(SpriteBatch) void
placeIngredientStation(Vector2) void
takeIngredientStation(Vector2, Ingredient) void
«create»
«create»
«create»
«create»
«create»
«create»
«create»

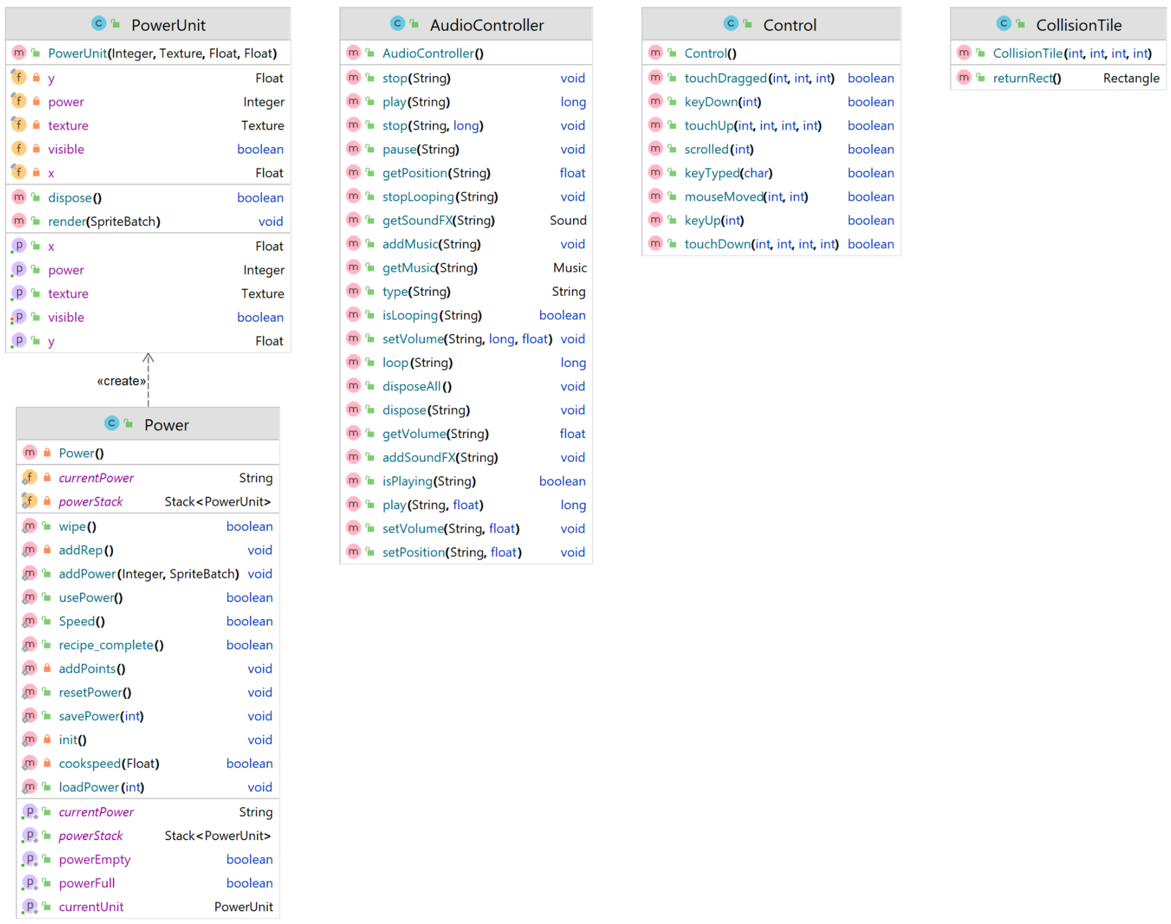
Screens Relationship UML diagram



Entity Relationships UML diagram


Entity
Entity()
draw(SpriteBatch) void
Cook
Cook(Vector2, int)
walkFrames TextureRegion[]
direction Vector2
pickUpItem(Ingredient) void
dropItem() void
checkCollision(float, float, CollisionTile[][]) Boolean
draw_bot(SpriteBatch) void
getCollideBoxAtPosition(float, float) Rectangle
update(Control, float, CollisionTile[][]) void
full() boolean
draw_top(SpriteBatch) void
draw_top(SpriteBatch, Vector2) void
y float
x float
speed Float
height float
width float
direction Vector2
walkTexture String
walkFrames int
lockedCook
1
Station
Station(Vector2, int, boolean, Ingredient[], String)
isAllowed(Ingredient) boolean
interactSound() void
place(Ingredient) boolean
take() Ingredient
drawTakeText(SpriteBatch) void
drawText(SpriteBatch, String, Vector2) void
drawDropText(SpriteBatch) void
«create»
*
stations
Customer
Customer(int, int, int, int)
targetsquare int
renderCustomersBot(Batch) void
stepTarget() void
arrived() void
hasExpired() Boolean
renderCustomersTop(Batch) void
targetsquare int
*
customers
1
«create»
CustomerController
CustomerController(TiledMap)
isCustomerAtPos(Vector2) Customer
saveCC(int) void
computeCustomerZone(TiledMap) void
spawnCustomer() void
reInitCustArr() void
drawCustTop(Batch) void
updateCustomers() void
delCustomer(int) void
delCustomer(Customer) void
loadCC(int) void
CookingStation
CookingStation(Vector2, int, Ingredient[], String, String)
drawParticles(SpriteBatch, int) void
checkCookingStation(SpriteBatch) void
createParticleEmitter(Vector2, String) void
lockCook() boolean
CuttingStation
CuttingStation(Vector2)
interact(SpriteBatch, float) boolean
lockCook() boolean
IngredientStation
IngredientStation(Vector2, Ingredient)
PrepStation
PrepStation(Vector2)
testPower(int, int) void
generatePower(int) void
drawStatusBar(SpriteBatch) void
ingredientMatch(Ingredient) Ingredient?
lockCook() boolean
slotsToRecipe() boolean
updateProgress(SpriteBatch, float) void
FryingStation
FryingStation(Vector2)
OvenStation
OvenStation(Vector2)
place(Ingredient) boolean
BakingStation
BakingStation(Vector2)
place(Ingredient) boolean
«create»
ServingStation
ServingStation(Vector2)
takeCustomerOrder() void
serveOrder() void
«create»
1
StationManager
StationManager()
checkStationExists(Vector2, Station) boolean
unlockStation(Vector2, String) void
checkInteractedTile(String, Vector2) void
handleStations(SpriteBatch) void
placeIngredientStation(Vector2) void
takeIngredientStation(Vector2, Ingredient) void
«create»
«create»
«create»
«create»
«create»

Utility Relationships diagram:



Full Uml Diagram:

https://neves6.github.io/PiazzaPanic2/uml/FullUML.png

All the final UML relationships diagrams were generated using Intellij and formatted further using plantUML.This was the most efficient way we could do it.