Architecture

Group 28

Piazza Panic
By OuseWorks

Ben Harris
Joshua Gill
Niamh Hanratty
Amy Raymond
Matthew Czyzewski
Matt Rohatynskyj

Diagrammatic Representation

We used diagrams.net [1] on Google Drive [2] to describe the architecture of the code with a UML diagram, as a high-level abstraction of the game's architecture.
We used UML as it provides a non-natural language format that removes ambiguity and increases precision, and ability to portray Java classes. We considered other forms of representing this however we concluded that they were too verbose or imprecise. On top of this, UML is an industry standard and so it is not language or technology dependent.
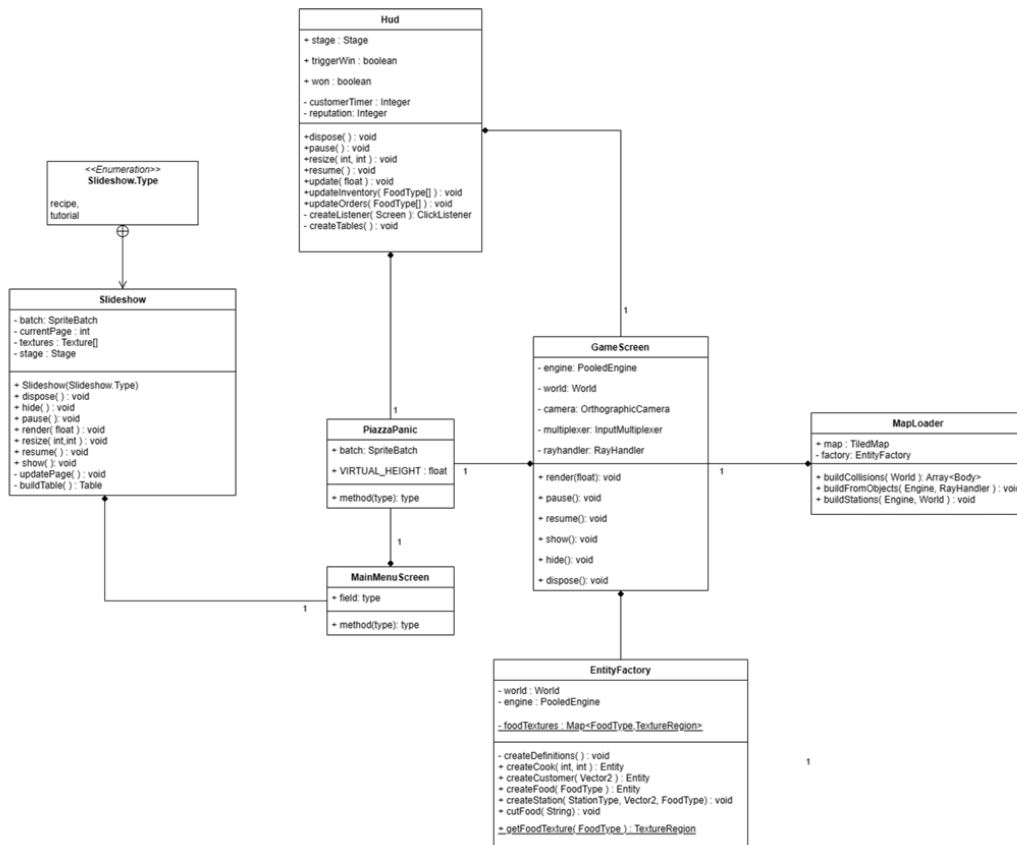
Updated Tools

Within OuseWorks, we tended not to use IntelliJ Ultimate [3] like devCharles did, as only one member of our team uses IntelliJ so it didn't seem appropriate to have to make changes to our method to account for this. Instead we continued to use the same software as devCharles, diagrams.net, to update their architecture diagrams as this was the easiest way to edit them rather than having to remake them from scratch. For the two new sequence diagrams we added, we used PlantUML [4] as it has a Google Drive plug in called PlantUML Gizmo. We found that this made it easy to make diagrams, leaving the code in the extension in the google doc which means that we could save our progress on the diagram, not having to finish it in one day.
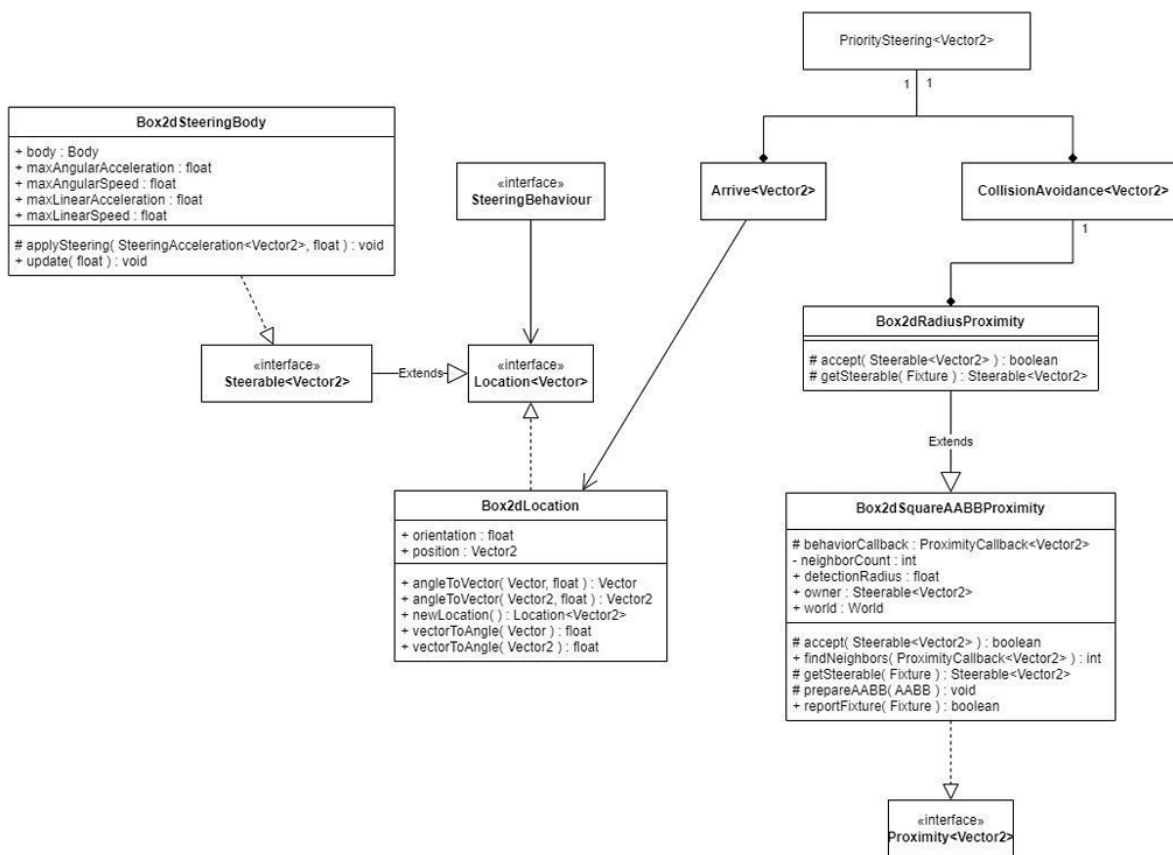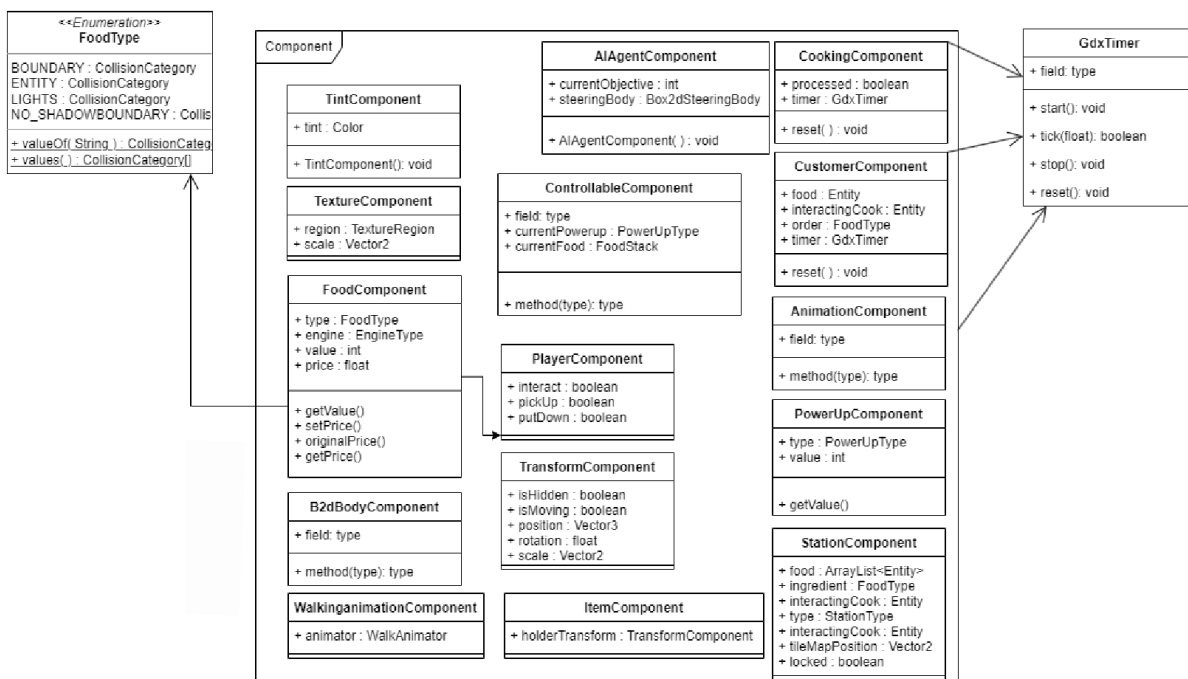
devCharles' diagrams
OuseWorks' updated diagrams
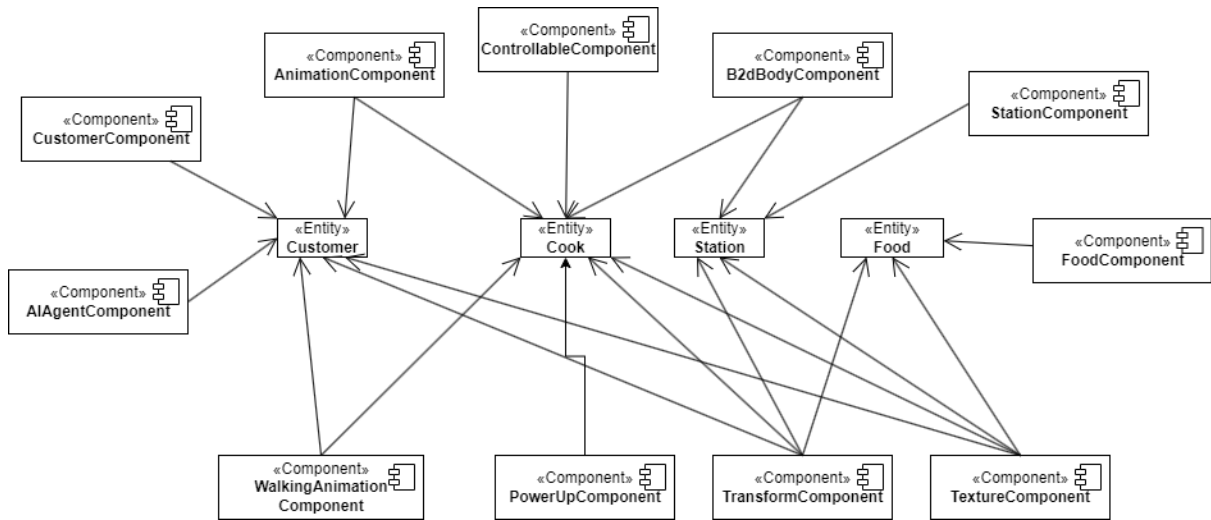
Main UML Diagram excerpt

Implementation of gdx-ai

PrioritySteering<Vector2>

Box2dSteeringBody

+ body : Body
+ maxAngularAcceleration : float
+ maxAngularSpeed : float
+ maxLinearAcceleration : float
+ maxLinearSpeed : float

applySteering(SteeringAcceleration<Vector2>, float) : void
+ update(float) : void

«interface»
SteeringBehaviour

Arrive<Vector2>

CollisionAvoidance<Vector2>

«interface»
Steerable<Vector2>

Extends

«interface»
Location<Vector>

Box2dRadiusProximity

accept(Steerable<Vector2>) : boolean
getSteerable(Fixture) : Steerable<Vector2>

Box2dLocation

+ orientation : float
+ position : Vector2

+ angleToVector(Vector, float) : Vector
+ angleToVector(Vector2, float) : Vector2
+ newLocation( ) : Location<Vector2>
+ vectorToAngle(Vector) : float
+ vectorToAngle(Vector2) : float

Extends

Box2dSquareAABBProximity

behaviorCallback : ProximityCallback<Vector2>
- neighborCount : int
+ detectionRadius : float
+ owner : Steerable<Vector2>
+ world : World

accept(Steerable<Vector2>) : boolean
+ findNeighbors(ProximityCallback<Vector2>) : int
# getSteerable(Fixture) : Steerable<Vector2>
# prepareAABB(AABB) : void
+ reportFixture(Fixture) : boolean

«interface»
Proximity<Vector2>

Updated Component Classes UML

«Enumeration»
FoodType

BOUNDARY : CollisionCategory
ENTITY : CollisionCategory
LIGHTS : CollisionCategory
NO_SHADOWBOUNDARY : Collis

+ valueOf(String) : CollisionCateg
+ values( ) : CollisionCategory[]

Component

TintComponent

+ tint : Color

+ TintComponent(): void

TextureComponent

+ region : TextureRegion
+ scale : Vector2

FoodComponent

+ type : FoodType
+ engine : EngineType
+ value : int
+ price : float

+ getValue()
+ setPrice()
+ originalPrice()
+ getPrice()

B2dBodyComponent

+ field: type

+ method(type): type

WalkinganimationComponent

+ animator : WalkAnimator

AIAgentComponent

+ currentObjective : int
+ steeringBody : Box2dSteeringBody

+ AIAgentComponent( ) : void

ControllableComponent

+ field: type
+ currentPowerup : PowerUpType
+ currentFood : FoodStack

+ method(type): type

PlayerComponent

+ interact : boolean
+ pickUp : boolean
+ putDown : boolean

TransformComponent

+ isHidden : boolean
+ isMoving : boolean
+ position : Vector3
+ rotation : float
+ scale : Vector2

ItemComponent

+ holderTransform : TransformComponent

CookingComponent

+ processed : boolean
+ timer : GdxTimer

+ reset( ) : void

CustomerComponent

+ food : Entity
+ interactingCook : Entity
+ order : FoodType
+ timer : GdxTimer

+ reset( ) : void

AnimationComponent

+ field: type

+ method(type): type

PowerUpComponent

+ type : PowerUpType
+ value : int

+ getValue()

StationComponent

+ food : ArrayList<Entity>
+ ingredient : FoodType
+ interactingCook : Entity
+ type : StationType
+ interactingCook : Entity
+ tileMapPosition : Vector2
+ locked : boolean

GdxTimer

+ field: type

+ start(): void
+ tick(float): boolean
+ stop(): void
+ reset(): void

Updated Entity-System Relationship

«Component»
CustomerComponent

«Component»
AnimationComponent

«Component»
ControllableComponent

«Component»
B2dBodyComponent

«Component»
StationComponent

«Entity»
Customer

«Entity»
Cook

«Entity»
Station

«Entity»
Food

«Component»
FoodComponent

«Component»
AIAgentComponent

«Component»
WalkingAnimation
Component

«Component»
PowerUpComponent

«Component»
TransformComponent

«Component»
TextureComponent

Updated System-Component Relationship

«Component»
ItemComponent

«System»
PhysicsSystem

«System»
RenderingSystem

«System»
CarryItemSystem

«System»
LightingSystem

«Component»
PlayerComponent

«System»
InventoryUpdateSystem

«System»
CustomerAISystem

«Entity»
Customer

«System»
PlayerControlSystem

«Entity»
Cook

«Entity»
Station

«System»
StationSystem

«Entity»
Food

«System»
PowerUpsSystem

«System»
DebugRendererSystem

«Component»
CookingComponent

«Component»
TintComponent

Systematic Justification

To begin with, our product was designed using class inheritance and polymorphism, with our initial plan being developed from our use of CRC cards (updated by OuseWorks), which can be seen [on this page](#). This informed our general inheritance structure, as the groups fell into several broad categories.

Actors (named entities in the CRC cards but actors here for clarity), which was composed mainly of the cooks, and the customers are those who move about the world, with character acting as a superclass for those two.
- This allowed us to satisfy the functional requirements FR_COOK, FR_CUSTOMERS and FR_CUSTOMERS_FLOW.

World is made of the objects that make up the world in which the actors exist. This ranges from concepts like the map, meaning the background and the bounds of the world, to the stations where the cooks create the food, to the storages where the cooks obtain the food from.
- This allowed us to satisfy the functional requirements FR_COOKING_STATIONS, FR_STATION_NUMBERS, and FR_INGREDIENT_STATIONS.

Food is the collection of the items that the cooks use, and the customers take. This is both the food itself and the ingredients which are prepared and combined to make the food. We initially defined ingredients and food as different due to the differing requirements they have, ingredients requiring the ability to be made into food.
- This allowed us to satisfy the functional requirement FR_RECIPES, and aided in satisfying FR_RECIPE_BOOK

GUI which is composed of the UI elements such as menus, the recipe book, the tutorial, customer orders and the win screen.
- This allowed us to satisfy the functional requirements FR_COMPLETION_TIME, FR_MENU, FR_RECIPE_BOOK.
- It also satisfied the user requirement UR_GAME_SIMPLE.

Player exists primarily on its own, as no other features are applicable to this type. This would involve controlling the cooks (switching, movement and interaction) as well as interacting with the GUI.
- This allowed us to satisfy the functional requirement FR_CONTROLS.

This architecture was used for our initial codebase, which followed these general groups when converting it to actual code, whilst further developing the different sections.
- Actors remained largely the same, with both Customer and Cook extending the Entity class which held common features such as the rendering of objects, allowing us to keep the requirements satisfied when creating it in a programming format.
- With the implementation of this section, we were able to additionally satisfy FR_COLLISIONS with the user of Box2d from LibGDX [5].
- World was somewhat separated between the world itself and the objects within the world. The GameWorld class was used to render the map itself, which was created outside the program using Tiled [6], whilst we created a superclass for the cooks' stations that had specific types of stations (e.g. cutting boards, grills) extend from it to allow it to share rendering code but change which food it can interact with.

- Food was also changed from the original CRC card structure, as we were able to have the completed food and ingredients both represented with the same class, simply differing on the methods that were called.
- GUI remained largely unimplemented by the time of this structure.
- Player similarly required little changes from the initial design, as it largely met the requirements specified.

However, as implementation of the customer began, it became apparent that our current structure was unideal to creating them, due to the restrictions that LibGDX and an inheritance structure placed on the movement mapping of customers. When considering our options, we decided to reformat our existing code into an Entity-Component System structure. This had the following benefits:
- It allowed us to implement the customer movement in a way that would be difficult to do with our previous structure, dividing the different parts required for movement (collision, movement, pathfinding) into separate components.
- Using Components allowed us to reduce code bloat as things such as rendering can be shared across all entities that require it. For example, in this structure we can use the TextureComponent and TransformComponents for anything that requires rendering.
- This also enabled easier development, as for the purpose of testing we were able to use components to better understand something during the actual coding of it. For example, we were able to better test customers in the process of creation due to already having the ability to give it a body and texture from the beginning.

Therefore, we chose to restructure the project to work with the ECS in order to fully meet the requirements, in addition with making it easier to meet more requirements such as UR_SYS_REQ and aid in meeting UR_DOC due to clarity of code.

The UML diagram for the ECS and final structure of this project can be found on our website with a diagram detailing the relationship between the entities, components and systems found above. The ECS diagram shows the entities we have, what components make them up and the systems that use them. An explanation of the systems and how they allow us to meet our requirements is below:

Cook

The cook entity is the entity controlled by the player that is able to interact with the stations and give food to customers. It has the following components:
- AnimationComponent: This is primarily used as a flag component to signal when an entity has a component. This has a field that is unused as we do not have any non-walking animations in the game. This component is used to meet requirement UR_GRAPHICS, as it allows clear distinction between the usable and relevant assets.
- B2dBodyComponent: This component stores the Box2d body of any object that requires collision, which is required for FR_COLLISIONS.
- ControllableComponent: This is the component that marks an entity as a cook, as they are the only entities the user is able to control. Hence, this is used to meet FR_CONTROLS and FR_COOKS. This also stores the food a player will hold, thus meaning it
- TextureComponent: This component holds the textures of an entity as well as the default scale of the texture. This helps to meet UR_GRAPHICS.
- TransformComponent: This component is used to define how an entity exists in

the world, and as such contains such things as its world position, its rotation and whether the entity is visible.
- WalkingAnimationComponent: This is a component used for flagging entities with a walk animation and stores an instance of the WalkAnimator class. This is used for UR_GRAPHICS, like AnimationComponent.
- PlayerComponent: This component is assigned within the PlayerControlSystem, and flags which cook entity is currently being controlled by the player. Hence, like ControllableComponent, this fulfils the requirements FR_CONTROLS as well as FR_COOKS (as it allows switching between cooks).

Food

The food entity is the entity that Customers and Cooks are able to hold, and can change between different forms of food. It has the following components.
- FoodComponent: This component holds the details of the food, stored as an enum of each possible type of food. Therefore, this allows FR_RECIPES, FR_COOKING_STATION and FR_INGREDIENT STATION to be completed
- TextureComponent and TransformComponent are used for the same purpose as in a cook entity.
- ItemComponent: This is assigned to the entity when they are held by a customer or cook entity, and it holds the TransformComponent of the holder to allow it to move with the holder within the world.
- CookingComponent: This is assigned to the entity when it is being cooked on a station to mark when it is in progress, and is removed when the time has passed.
- TintComponent: This is assigned to the entity when it must be interacted with whilst processing on a station.

Customer

This entity is very similar to the cook entity in its basics, but has a number of different components to allow it to move uncontrolled by a player. It has the following components:
- TextureComponent, TransformComponent, AnimationComponent, WalkingAnimationComponent, B2dBodyComponent are used for the same purpose as in a cook entity.
- CustomerComponent: This component is assigned to designate an entity as a customer. It is used to store aspects required of a customer such as ordering and a time limit to avoid reputation loss. Therefore, this is used to meet requirements FR_COMPLETION_TIME_LIMIT, FR_CUSTOMERS, FR_CUSTOMERS_FLOW and FR_TIMING.
- AIAgentComponent: This component is used for a customer to be able to path find to specific objective points. This is used to meet requirement FR_CUSTOMER_FLOW.

Station

This entity is used to provide functionality to the stations that are rendered through the tilemap.
- TextureComponent, TransformComponent, B2dBodyComponent are used for the same purpose as in a cook entity, though the values are given from the tilemap as opposed to from our assets.

- StationComponent: This component is used to mark an entity as a station, and stores the values required to give it functionality, such as storing the food and the cook who is using the station. Therefore it meets FR_COOKING_STATIONS, FR_INGREDIENT_STATIONS and FR_RECIPES.
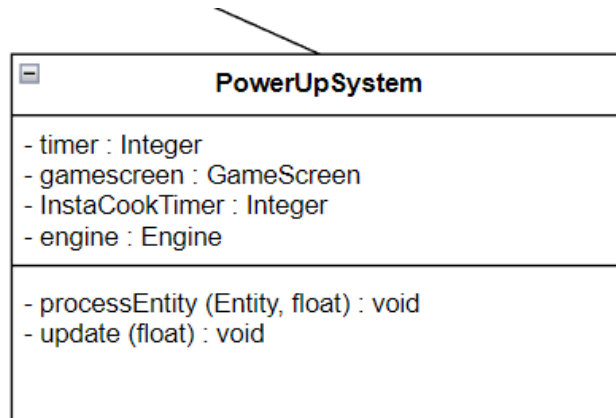
Relevant Classes

The following classes are particularly relevant for meeting our designated requirements:

- PlayerControlSystem: Assigns the PlayerComponent to a new cook when switching active cook and allows the user to interact with stations and customers, meaning it meets FR_CONTROLS and FR_RECIPES.
- Hud, Slideshow: Allows the user to view the recipe book, tutorial, pause screen as well as measuring the timer, meeting FR_RECIPE_BOOK and FR_COMPLETION_TIMER.
- MainMenuScreen: Allows the user to navigate the main menu, meeting FR_MENU.
- CustomerAISystem: Controls the customer, tracks how many have been served (by storing currently active customers), dictates how many customers can be spawned and how often they are spawned. This means this meets FR_COMPLETION, FR_CUSTOMERS, FR_CUSTOMERS_FLOW, FR_COMPLETION_LIMIT.
- RenderingSystem: allows the various entities to be rendered, therefore meeting UR_GRAPHICS.

**Updates to existing architecture:**

PowerUpSystem Classes UML

```
┌─────────────────────────────────────────┐
│ ⊟           PowerUpSystem                │
├─────────────────────────────────────────┤
│ - timer : Integer                        │
│ - gamescreen : GameScreen                │
│ - InstaCookTimer : Integer               │
│ - engine : Engine                        │
├─────────────────────────────────────────┤
│ - processEntity (Entity, float) : void   │
│ - update (float) : void                  │
│                                          │
│                                          │
└─────────────────────────────────────────┘
```

Components:
- FoodComponent has been given more fields and methods
    - This is to account for the added recipes for assessment 2 and for implementing the pricing aspect of the food and checking up on the food
    - Fulfills requirement FR_DISH_PRICES
- ControllableComponent has been given more fields
    - This is to account for the new power-ups added for the endless mode
    - Fulfills FR_POWER_UP.
- PowerUpComponent was added
    - FR_POWER_UP
- StationComponent has been given more fields
    - This has been added to fulfill FR_UNLOCK_STATIONS

System Components:
- CustomerAISystem has been given a new method
    - timeFreeze is one of the powerups, FR_POWER_UP
    - getRandomCustomerGroupSize, FR_CUSTOMER_FLOW
    - GetRandomCustomerTip, FR_CUSTOMER_TIPS
- PlayerControlSystem has been given more fields
    - Timer, FR_COMPLETION_TIME_LIMIT, FR_COMPLETION_TIME
    - speedMultiplayer was added as one of the power ups
- PowerUpSystem was added from scratch
    - FR_POWER_UP
- RenderingSystem has been given a new method
- StationSystem given more methods and fields
- Hud added more fields.

Power Ups Sequence Diagram
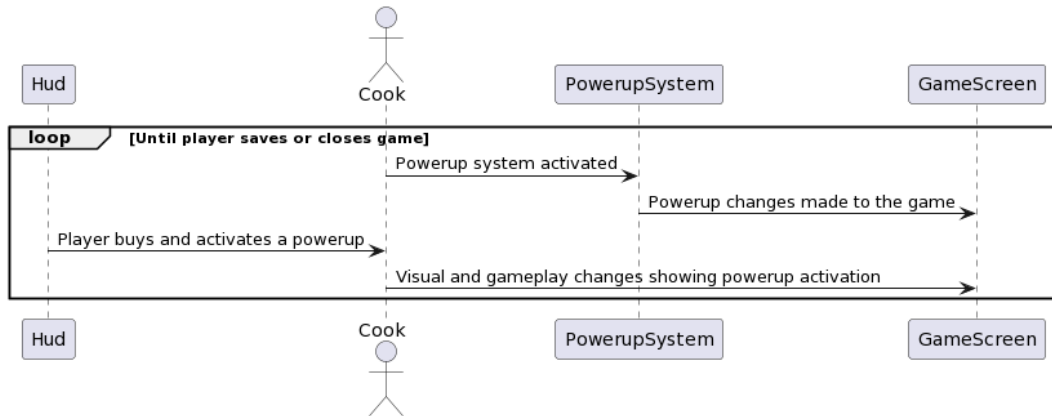
This system is necessary to satisfy the following requirements:
- FR_POWER_UP allowing for a player to unlock and activate in-game power ups
- FR_UNLOCK_STATIONS, FR_MONEY and FR_DISH_PRICES. This power up system enables implementation of in game currency with which the player can unlock new stations and chefs as stated in the client brief.

Saving the Game
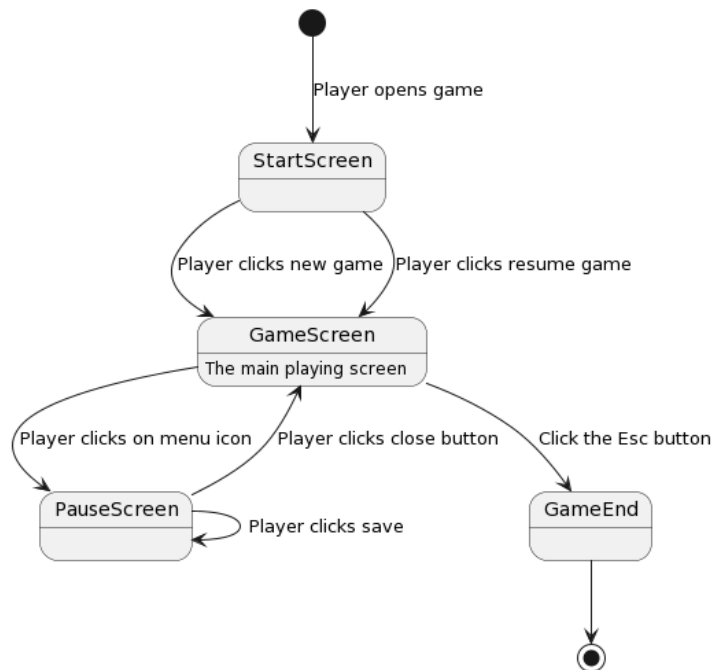
This state diagram above shows version 2 (final version) of our state diagram showing how we might implement the saving game. Our initial version (on website) explores the idea of being able to resume and overwrite previous saved games, however we realised that this is not necessary as it is not a requirement for assessment 2. The requirement that this diagram covers is UR_SAVE.

References:

[1] *Security-first diagramming for teams. Diagram Software and Flowchart Maker*. Available at: https://www.diagrams.net/

[2] *Personal Cloud Storage & File sharing platform* (no date) *Google*. Google. Available at: https://www.google.com/drive/.

[3] IntelliJ Ultimate https://www.jetbrains.com/idea/

[4] *Open-source tool that uses simple textual descriptions to draw beautiful UML diagrams.* (no date) *PlantUML.com*. Available at: https://plantuml.com/.

[5] *LibGDX* (2022) *libGDX*. Available at: https://libgdx.com/.

[6] (no date) *Tiled*. Available at: https://www.mapeditor.org/.