3. Architecture

Group Number: **Team 20**

Group Name: **Gourdo Ramsay**

Group Members:

Lauren Waine
Megan Bishop
Tommy George
Bartek Grudzinski
Davron Imamov
Nathan Sweeney

Intro

Architecture and architectural plans are crucial necessities when implementing any system as it provides the fundamental groundworks. These architectural diagrams can be split into two main categories – structural and behavioural. The structural diagrams are important for development and creating a consensus of how the project encompasses the different requirements and tasks it performs. It also focuses on how the sections of the system interact within the overall project. The behavioural diagrams involve the system’s interactions with the user and other components of the system - and how this causes the system state to change; it is a decomposed abstraction of the system’s behaviour.

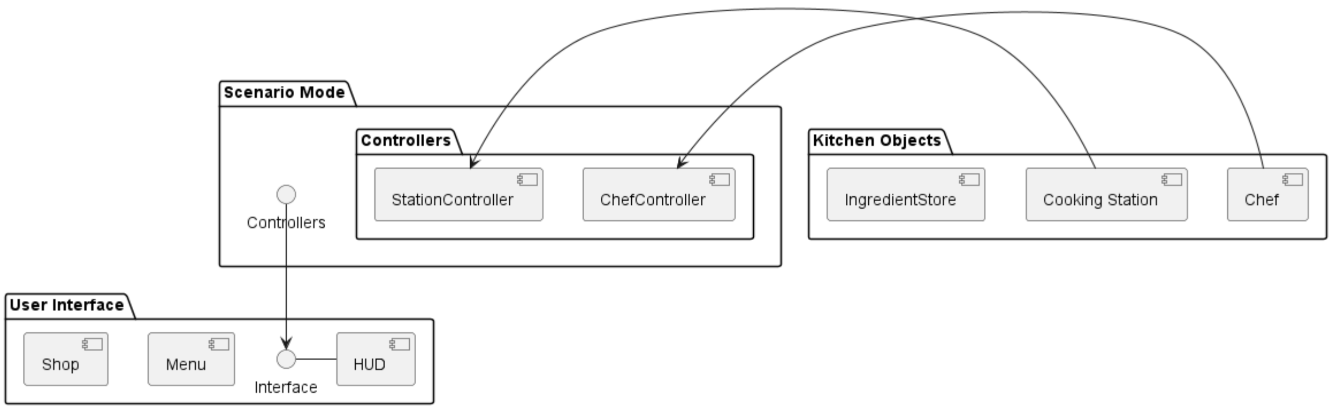
Tool Explanation

The diagrammatic representations have been designed using UML for different model representation with PlantUML as an extension of IntelliJ. This tool has allowed for the diagrams to be created and edited in real-time throughout the process in IntelliJ’s environment. On top of the diagrams needing to be edited in real time, it is also important that they are readily available within the system’s design as this is vital for the implementation of the project.

Structural

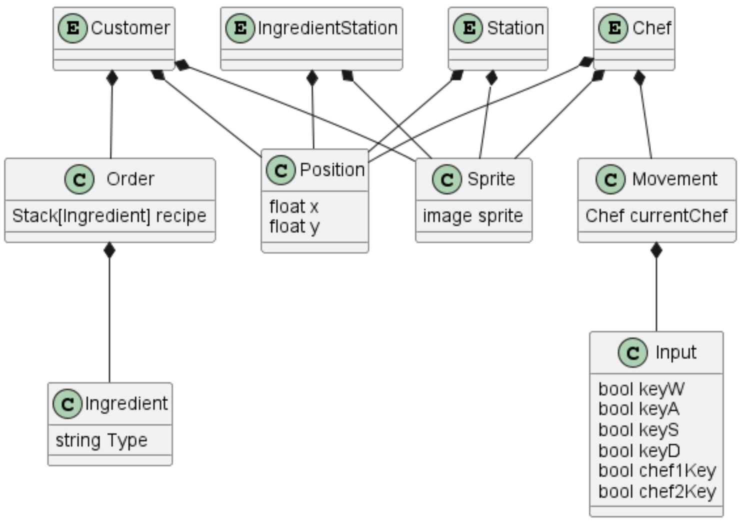
Component Diagram

A Component, or package, Diagram is used to organise different classes into larger type groups of components that refer to the same concept. The purpose of this is to create a level of abstraction when referring to entities or components of the same type for instance, ingredients or the user interface. This results in a larger scale concept to be generated while considering the components involved, but not the specific implementation or methods of them.



Entity Component Diagram

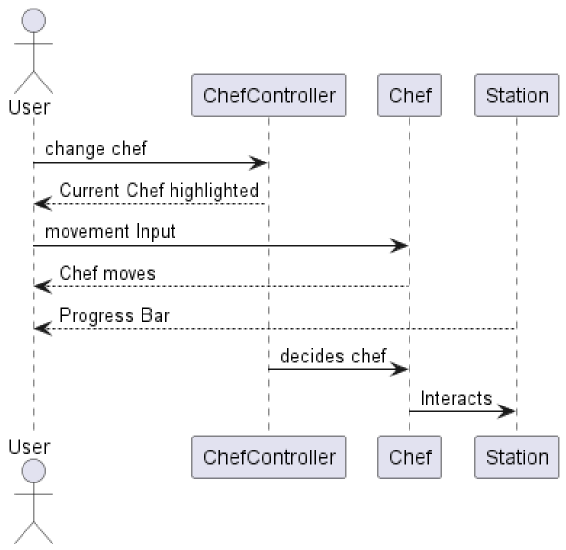
Diagrammatic representation for the overall entity-component structure is crucial for object-oriented programming. This specific UML diagram is used for demonstrating the inheritance and relations between different entities. The importance of this is that the relations between entities must be well documented and suitably abstracted to help with not only development, but also testing and polishing the product – if the structure and relations are well documented, it allows for procedures and dataflow to be traced while keeping encapsulated entities to allow easier troubleshooting, reducing redundancy, through a higher level of abstraction that therefore creates more protection.



<u>Behavioural</u>

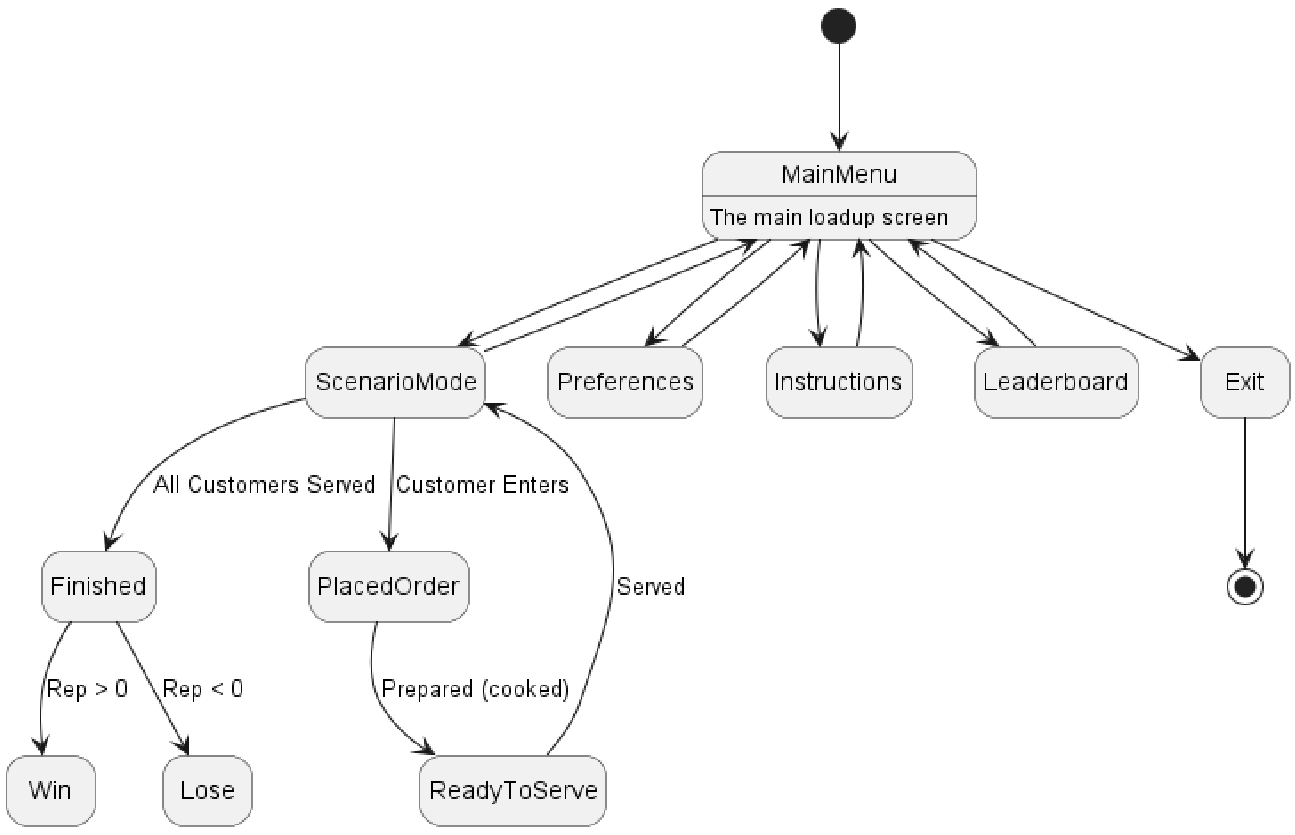
<u>Sequence Diagram:</u>

A sequence Diagram is another key behavioural UML diagram. It gives a brief insight into the entities within the system sequentially. One of the main reasons for this type of diagram is it allows the individual interactions between specific entities to be seen which means that expected interactions and data can be known before implementation; this also helps with the encapsulation. Another reason for their importance is it allows alternative scenarios that vary to be modelled – for instance the interaction of the food entities when serving each meal, or incorrectly chosen.



State Diagram:

State Diagrams are vital for demonstrating the behaviour and flow of a system. This is due to the fact it portrays the states the system can be in, and the transitions required to move between these available states. The importance of this is it allows developers to be aware of key procedures that will occur throughout the execution of the system and how they link together with a larger level of abstraction so that specific implementation is ignored – leaving the overall interactions within the system.



B.

The original stages of the architectural design were through prescriptive models that took the form of rudimentary sketches of the overall concept and of each component. This was achieved by analysing the user's requirements for the project and discussing potential solutions to form a consensus on the best design ideas. The main requirements that informed the initial sketches were UR_ARCADE_GAME and UR_IMAGES as this exclusively defines how the architecture should be perceived to the end user. We also considered the requirements UR_INGREDIENT_STATION, UR_COOKING_STATION and UR_CUSTOMER_WANTS when designing the scene to ensure we had every aspect of the game that would be required. This resulted in an open layout (seen in Figure 1 under Diagrams on the website) for the scene (kitchen) to allow the user easy and effortless movement between the stations and ingredients, and to serve the customers. After developing the original style of the game and overall layout, a storyboard (Figure 2) was generated to allow a precursive concept for the flow of the game and how the user will navigate through it. This is important as it allows the progression to be concentrated on specific areas while providing an abstracted and simplistic architectural view.

Once the design of the overall architecture was developed, the interactions of the components needed to be discussed. The original theoretical architecture was through a range of self contained classes (Figure 3) that would pass data between one another and encapsulate each component so that the data is only accessed by relevant parts. This would increase security and protection of the system which is important when producing a game to avoid bugs that may affect user experience (in a minor or major way). This however is easy to plan architecturally, but results in a high quantity of refactors as issues arise throughout the development process. Therefore a more largely abstracted model was generated through CRC cards (seen below). These are valuable for responsibility-driven design where flexibility is preserved by trying to maximise abstraction to allow development to evolve and not be limited. The original CRC cards were developed by conceptualising the key sections of the system and briefly describing their responsibilities, i.e. the actions and interactions that they encompass. Finally, the different sections they interact with to perform these functions were noted as their collaborators. This is valuable as it further helps map the interlinking parts and what is required for each section to be implemented.

Station	
Responsibilities	**Collaborators:**
Change Food State	Station
Specific Station Task	Ingredient

Ingredient	
Responsibilities	**Collaborators:**
Type	Station
State (prepped/not)	Chef

Chef	
Responsibilities	**Collaborators:**
Chef Movement	Station
Chef ID	Ingredient
Chef Sprite + Animation	

UI	
Responsibilities	**Collaborators:**
Interface to User	Station
Show movements	Ingredient
Reflect input data	Chef
Provide coherent experience	

To also aid development, additional UML diagrams were designed to allow a carefully planned out system and interactions. One example of these UML diagrams is a sequence diagram (Figure 4). This prompted the interactions between different systems and the user so that these could be planned and implemented accordingly. This however was updated in the final version seen in the previous section as it did not include the feedback from the system to the user - a key part to any system, especially a game where constant information to the user is valuable to enable a seamless gaming experience.

In actuality to simplify development and allow for a deeper understanding of how our chosen platform - LibGDX - functioned, the architectural development took place within one class in the main java file. This class extended the 'ApplicationAdapter' - the main class which provides the functionality for implementing a user interface. This however, was not efficient in practice as it did not allow for easy collaborative work due to the potential risk of overriding previous code; this may also struggle to be removed even with version control. Furthermore, it also resulted in redundancy and repeated code - which in turn will have a detrimental impact on the overall performance of the system and any further development from a different developer. If further development occurred with this flawed architecture, it would have resulted in catastrophic errors with the code and overwriting issues.

Due to these issues with the current architecture, it was altered to be structured as a more modular and object oriented approach. This was achieved by changing the same main class to become a controller for initialising the game and by extending 'Game' instead. This allows the use of screens which are self contained classes for each screen present in the game seen in the aforementioned storyboard (Figure 1). When designing the separate scenes we settled on the idea of an instruction screen as to satisfy the requirement NFR_FOOD_PREP so that users can reference the recipes and controls whenever they like should they get confused or forget something. This is highly relevant as it creates a simpler design that is easier for the user to understand and increases the accessibility of the game - allowing it to appeal to a wider audience, a vital part of the brief's target user. Even more so to further meet the user requirements, the storyboard was updated (Figure 5) to include a leaderboard, and winning and losing screens - along with a few further alterations. This results in the new architectural designs to meet UR_LOSE_GAME, UR_WIN_GAME and UR_LEADERBOARD. Each screen is developed as a separate class that implements the screen package causing the object to only be instantiated within a given screen. The large benefit of this is it begins to encapsulate the components and allows the code to not only run more efficiently, but also allows easier collaboration when working.

To further encapsulate the components and progress the system's modularity, separate classes were used for objects that will be repeated in multiple scenes or multiple times; these objects consist of the chefs, stations, ingredients, and customers. It is important that the architecture becomes fully object oriented and modular as it not only permits simpler testing and error resolution, but also allows further collaboration support, especially with any new development teams. Additionally, this approach is perfect for expanding the game and limiting aspects - for instance when attempting to meet requirements FR_SWITCH_COOKS and UR_EARN_MONEY. It is vital for these requirements as when switching and controlling separate cooks, if they are created as separately managed objects, these controls can easily be implemented and the objects simply made idle. It is important for UR_EARN_MONEY, as the requirements state that the user should be able to buy new chefs or stations from the shop with this money - which can efficiently be done through an object-oriented approach due to only a new instance having to be instantiated to apply these alterations.

Once the components are encapsulated and converted into separate classes, the architecture can be further developed using inheritance. This helps to continue to reduce redundancies by allowing generic methods and attributes to be centrally managed from a parent superclass. This superclass stores overridable data and methods - allowing the differences between its subclasses to occur while still maintaining a layer of generality. For instance, this can be applied to the ingredients as all will require methods to be interacted with and prepared, but this will differ between each subclass (ingredient type). On top of reducing redundancies, this architectural method will allow for easy expansion of the game by introducing new recipes and ingredients as the game progresses to introduce a level of variation within the gameplay.