# Architecture

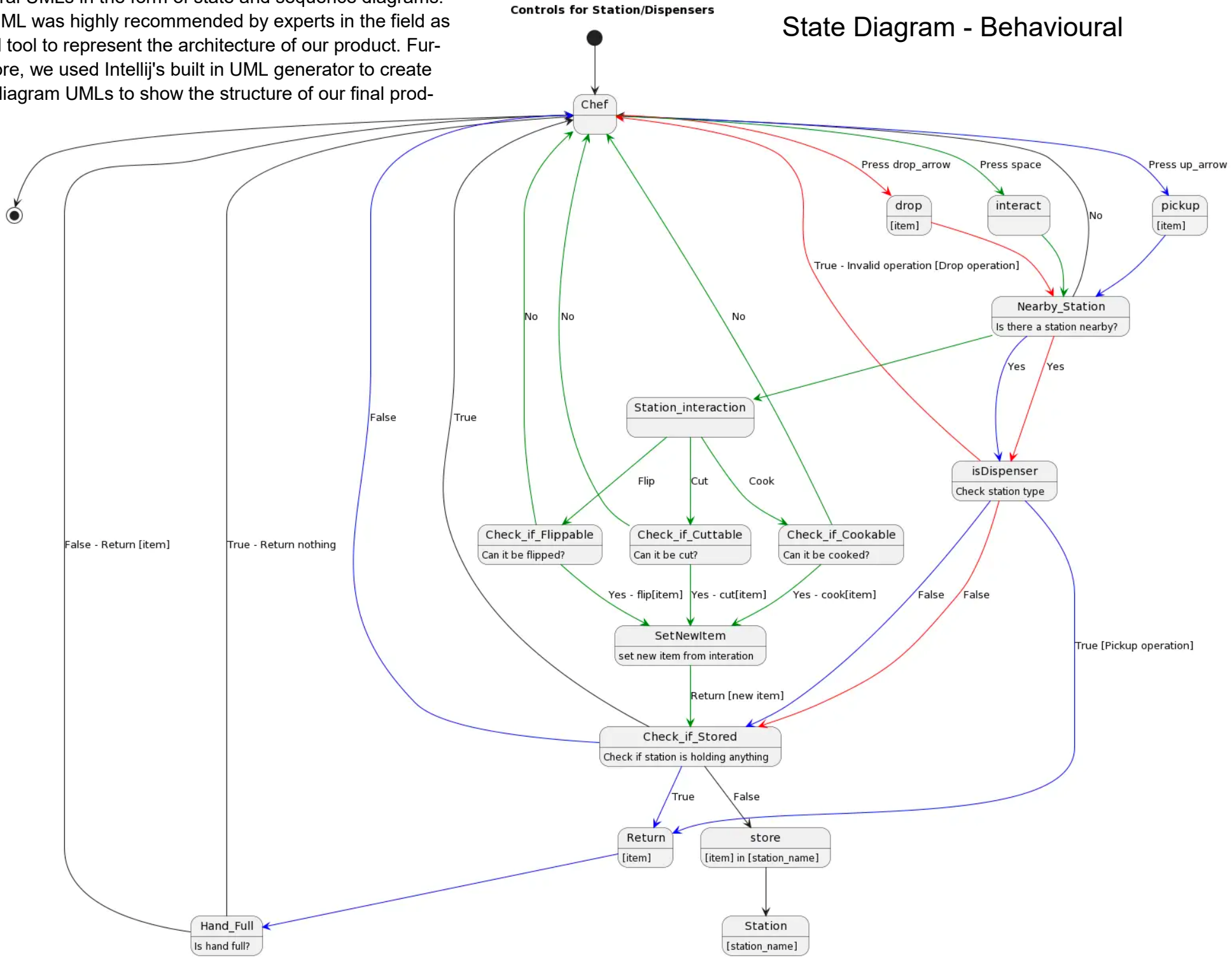
ENG1 Team 7
Broken Designers

Adam Brown
Morgan Francis
Oliver Johnstone
Shabari Jagadeeswaran
Laura Mata Le Bot
Rebecca Stone

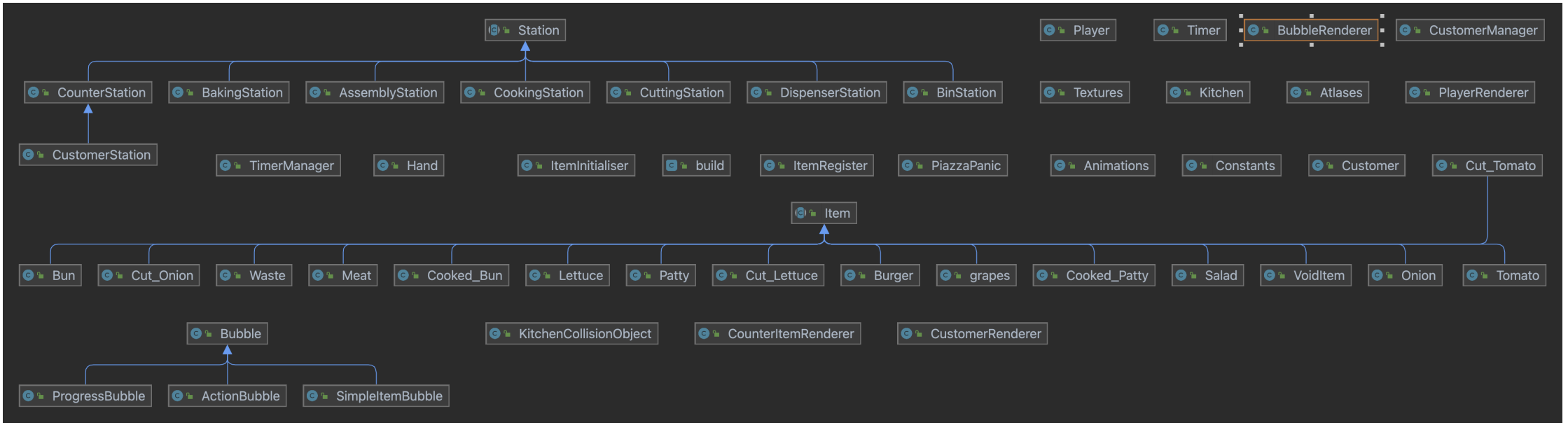
"To make these UMLs I have used PlantUML to create behavioural UMLs in the form of state and sequence diagrams. PlantUML was highly recommended by experts in the field as a good tool to represent the architecture of our product. Furthermore, we used Intellij's built in UML generator to create class diagram UMLs to show the structure of our final product."

# UML Diagrams

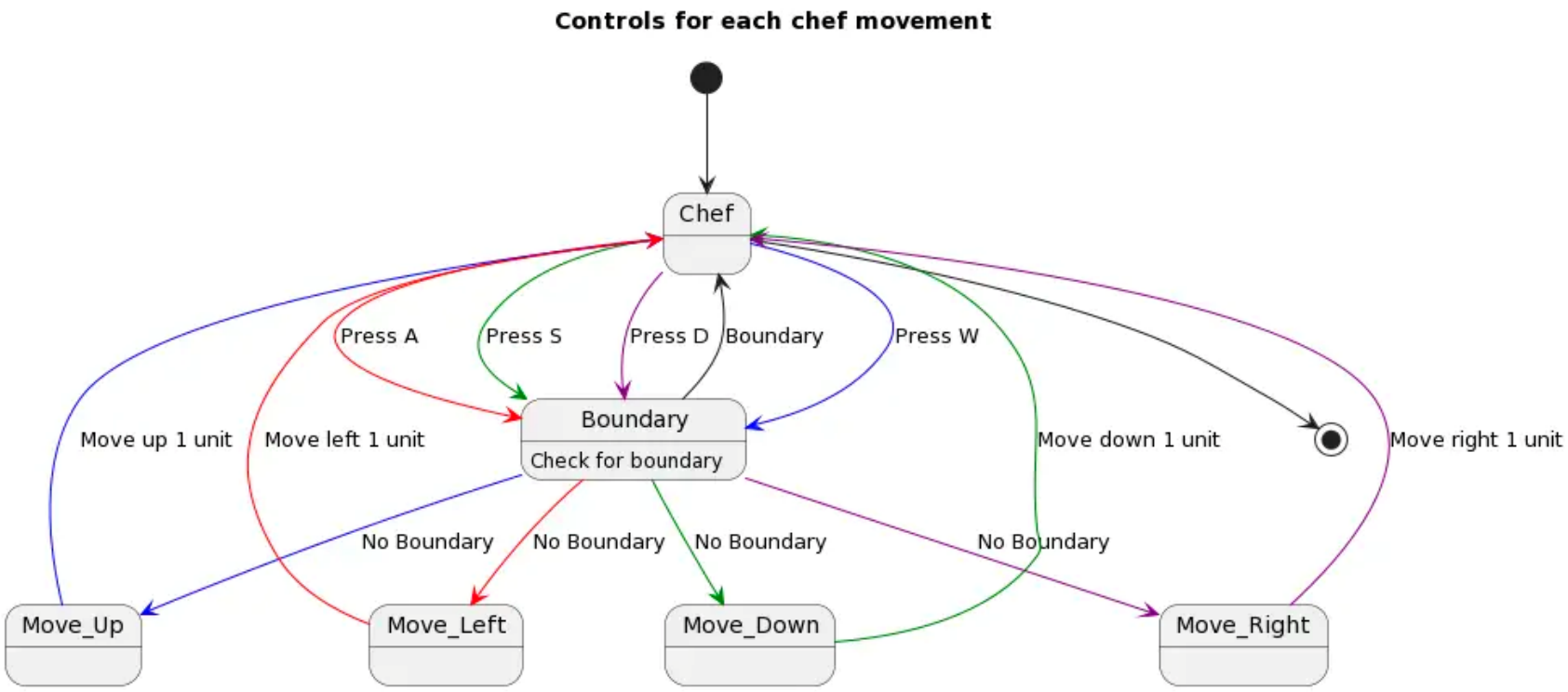
## State Diagram - Behavioural

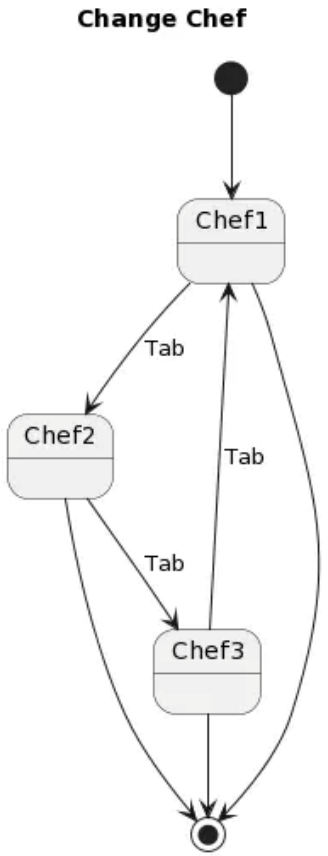


# Class Diagram - Structural

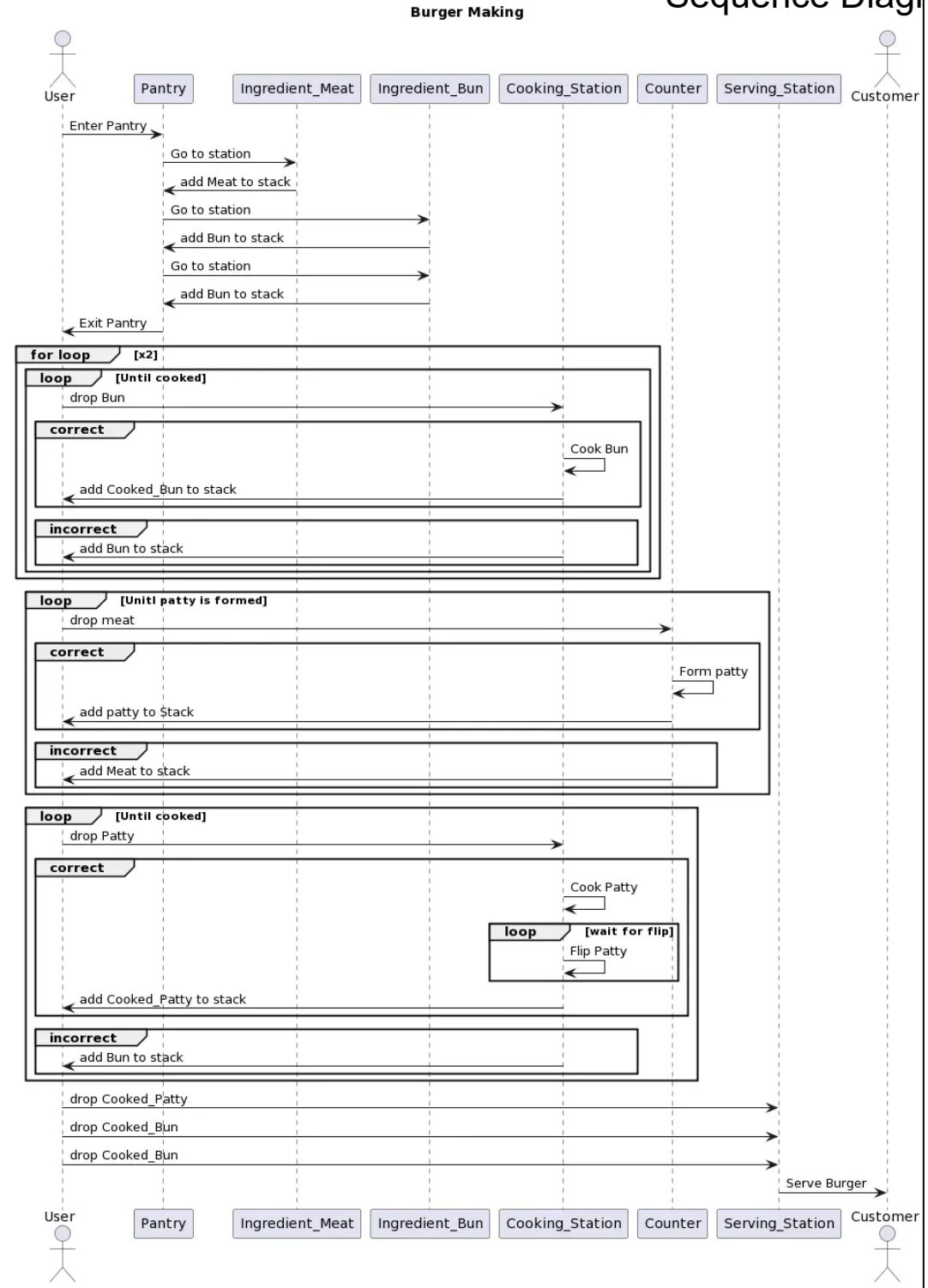


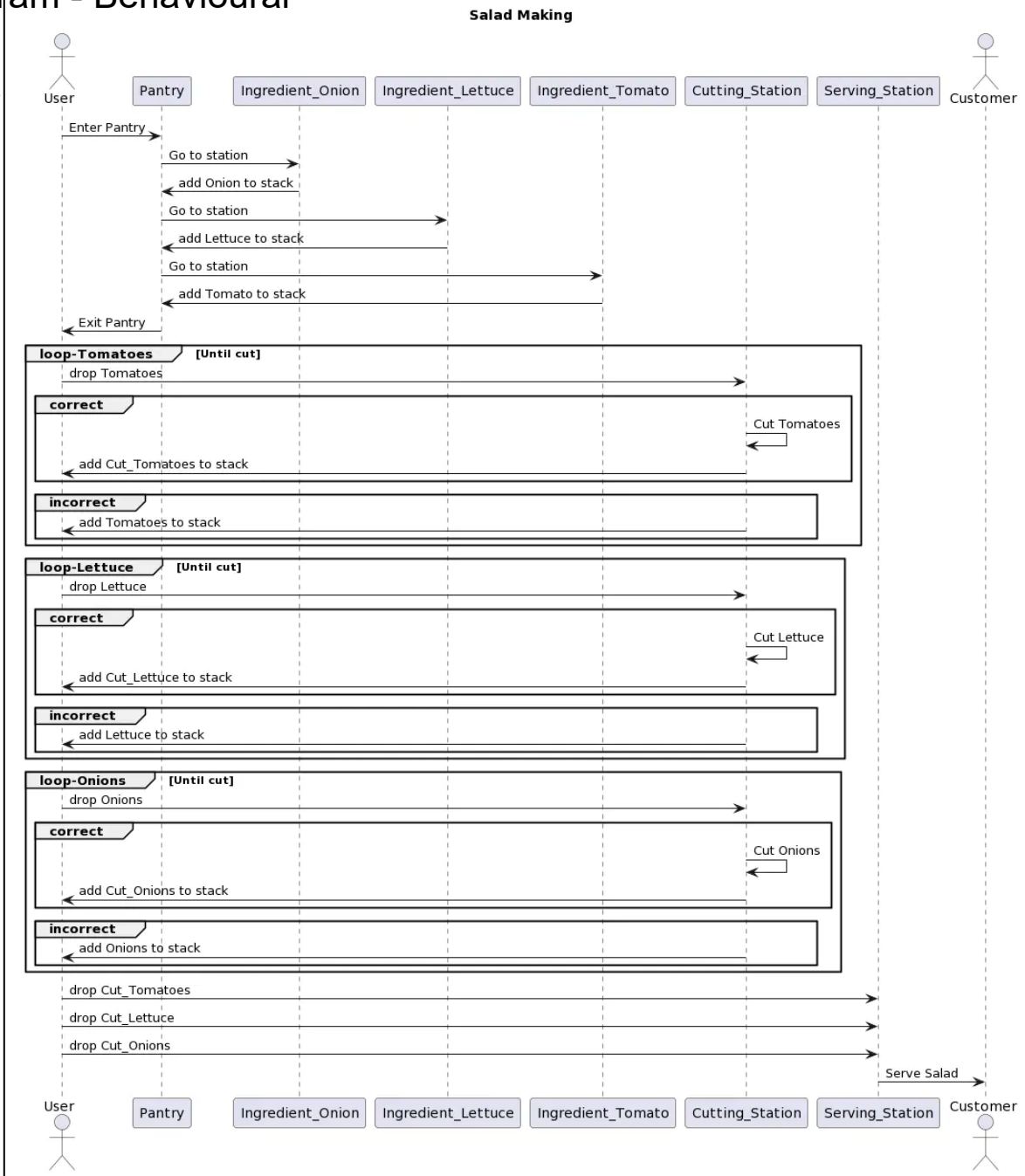

# State Diagrams - Behavioural



# Sequence Diagram - Behavioural



# Architecture report

The architecture design loosely follows the entity-component-system style [1]. The most beneficial element of this approach is modularity. Modularity involves taking the individual parts of a system and breaking them down into smaller functionalities that can be reused throughout the system.

There are several different people contributing to the design and implementation of the project. The modularity of the entity-component structure makes the code easy to read and extend, allowing developers that are unfamiliar with the project to understand how to use and extend the design. Additionally, modularity provided us with consistency in the behaviour of several of the classes.

The entity-component-system architecture encourages the use of abstract classes that can be reused and refactored into many different entities. This structure improves the robustness and quality of the end-product. This is because issues in the code only need to be fixed in one class instead of over an entire system, reducing the likelihood of errors being missed. For example, all stations inherit the ability to pick up and put down items on them from the component "*station*". Components of the game like the food items or different stations have similar requirements to each other, which made selecting this structure beneficial for both our timescale and for the quality of the project as a whole.

We also considered other architecture styles, but these did not fit our product well. For example, pipeline architecture is better suited to large scale projects like real time message passing platforms. Architectures like this were ruled out due to the difficulty of applying them to the project and also the time that would have been needed to maintain them. The features upheld by the entity-component system architectural style will help other teams expand on the product and allow us to organise our code into a readable format.

However, the entity-component-system model was not fully implemented. In this architectural style, entities only contain a set of data, and components only contain a set of entities which are then used and managed by the system.This was not followed in our final design. In our architecture, we have used the basic idea of entities and components, but allowed entities and components to contain logic that would normally be handled by the system. The decision to deviate from the model was done largely because Java does not support multiple inheritance. In a traditional entity-component-system style, many different components would be used by a system. Since this was not possible in our architecture, we decided to deviate from the model. The adapted version of the model simplifies the code, leading to less cluttered architecture which in turn will make the code easier to expand upon.

Initially, we focused only on the entity-component-system architecture, but this design evolved over the course of the project. Our earliest design of the architecture focused on creating generalised components that many entities could use. Although the game needed many different food items and work stations, a lot of these requirements would share similar properties. An example of this is that all workstations should allow the player to pick up and drop items on them. Therefore, implementing a set of generalised abstract classes would reduce the amount of repeated code. Our earliest model included abstract entities such as an abstract "*item*" class from which all food could inherit their properties, as well as classes such as "*station*" which workstations would use. At the time of initial design, we did not have a specific idea on which architectural pattern we would follow. However, after the first iteration of reviewing the key classes, we identified that utilising reusable components would greatly reduce the work required. From this discovery, we focused on designing mock diagrams of classes that would follow the entity-component-system style in order to meet this requirement. This was done through brainstorming a group class diagram on a whiteboard, and this would be the basis for our initial architecture. Sessions like these guided our initial creation and subsequent evolution of the architecture by allowing all

members of the team to contribute ideas and understand the methods by which we would design the game.

Further into development, we made sequence diagrams in order to better understand the interactions that should happen in the game. The use of sequence diagrams enabled us to mature our requirements around several key interaction components of the game. For example, one of our sequence diagrams, which is about the creation of a salad (refer to the UML section at https://ajbrown-york.github.io/html/umls.html), allowed us to see which aspects of the design would need to be completed in order to allow this requirement to be implemented. This model was updated throughout the project to fit the changing environment of the game. The sequence diagrams also shed light on the construction of several key components of different interactions. These models were updated as we created more parts of our game, as they provided insight into intricate details we would need to be careful of while constructing our implementation of the game. For example, the control sequence diagram was updated to include collision when we were working on the implementation of that section into the architecture.

The implementation of parts of our architecture was significantly reconsidered after we created state diagrams of various aspects of the game (refer to the UML section at https://ajbrown-york.github.io/html/umls.html). These allowed us to see individual issues that could arise if players did actions that were unconventional, such as picking up an item when their hands were already full. Our initial diagrams failed to implement methods to prevent these actions, so had to be updated to accommodate this discovery. The issues highlighted in this manner caused us to reconsider entities that we had already created, requiring us to put checks in place to catch unexpected behaviour the user could take. The lessons learned from the use of stage diagrams were used when creating new entities and components, which benefited the robustness of the architecture as it evolved. At this point in the design process, we decided that state and sequence diagrams would help the design process the most, so we focused on designing diagrams in these styles.

Our final architecture stuck to the entity-component-system style as much as possible. As mentioned previously, we chose to persist with the principle of the model by having components that entities inherited from, but allowed entities and components to hold more than the model suggested. There are also areas that do not follow the architectural style at all, though these are few in number. An example of this is the "*CounterStation*" class that both inherits and also inherited from by other objects. This is a rare exception to the general architectural pattern we worked by. This was done as this class largely inherits from the "*Station*" class, but is also used to make the counters used by the player to serve customers. Therefore, in order to prevent redundant code, the class that handles the customer counter, "*CustomerStation*", inherits from "*CounterStation*" which itself inherits from "*Station*". In general, we have managed to stick to the general architectural style we planned from the start, changing it slightly as we started to implement the program due to the limitations of the language and to improve the general clarity of the classes in our system from an outside perspective.

The implementation of the architecture was designed with our requirements in mind. All of these are based on functional requirements.

The FR_CHEF_INFO requirement states that information should be stored about the chef's ID, their selection status (are they currently selected), and their hand. The class Player stores this information and references a subclass hand. The hand class stores information about the three items on the chef's stack. The FR_CHEF_SWITCHING and FR_CHEF_MOVEMENT requirements are also fulfilled by the Player class alongside the Piazza Panic class.

This was really useful as we could move the chefs in whichever direction we wanted, be it horizontally or vertically across the stations.

The FR_ITEM_REGISTER requirement states the system must store an item register with a string ID for each item, which is implemented using the class ItemRegister and ItemInitialiser. Any place that handles the picking up and dropping off items references the ItemRegister class.

In addition to this, the FR_ITEM_INTERACTION requirement was for the purpose of interacting between chefs and items. An instance of Player initialises the interaction to be handled by the Station. Furthermore, the FR_COOKING_STATION requirement was needed for many activities like chopping, frying, baking, and assembly. To implement these, the abstract Station class and its child class are used.

Another integral part of the game was to serve systematically, as covered by the FR_SERVING_STATIONS requirement. This is implemented by the child class of Station, CustomerStation and also the Customer class. The FR_ASSEMBLY_STATION requirement states that different prepared ingredients of a recipe must be taken to the assembly station to determine if a recipe was followed and then prepare the final dish. This is implemented by the AssemblyStation class.

## Bibliography

https://ajbrown-york.github.io/html/bibliography.html