3a) A UML class diagram was used to create the diagrammatic representation of the architecture as, being an object-oriented modelling language, it is well suited to representing Object oriented languages. Class diagrams are especially useful as they provide two sections per class for attributes and functions. This makes the diagram more coherent, and makes it easier to see the difference between local attributes and referenced classes. UML also provides a good way to show the visibility of a class, which is useful in seeing how the classes can interact with each other's attributes.

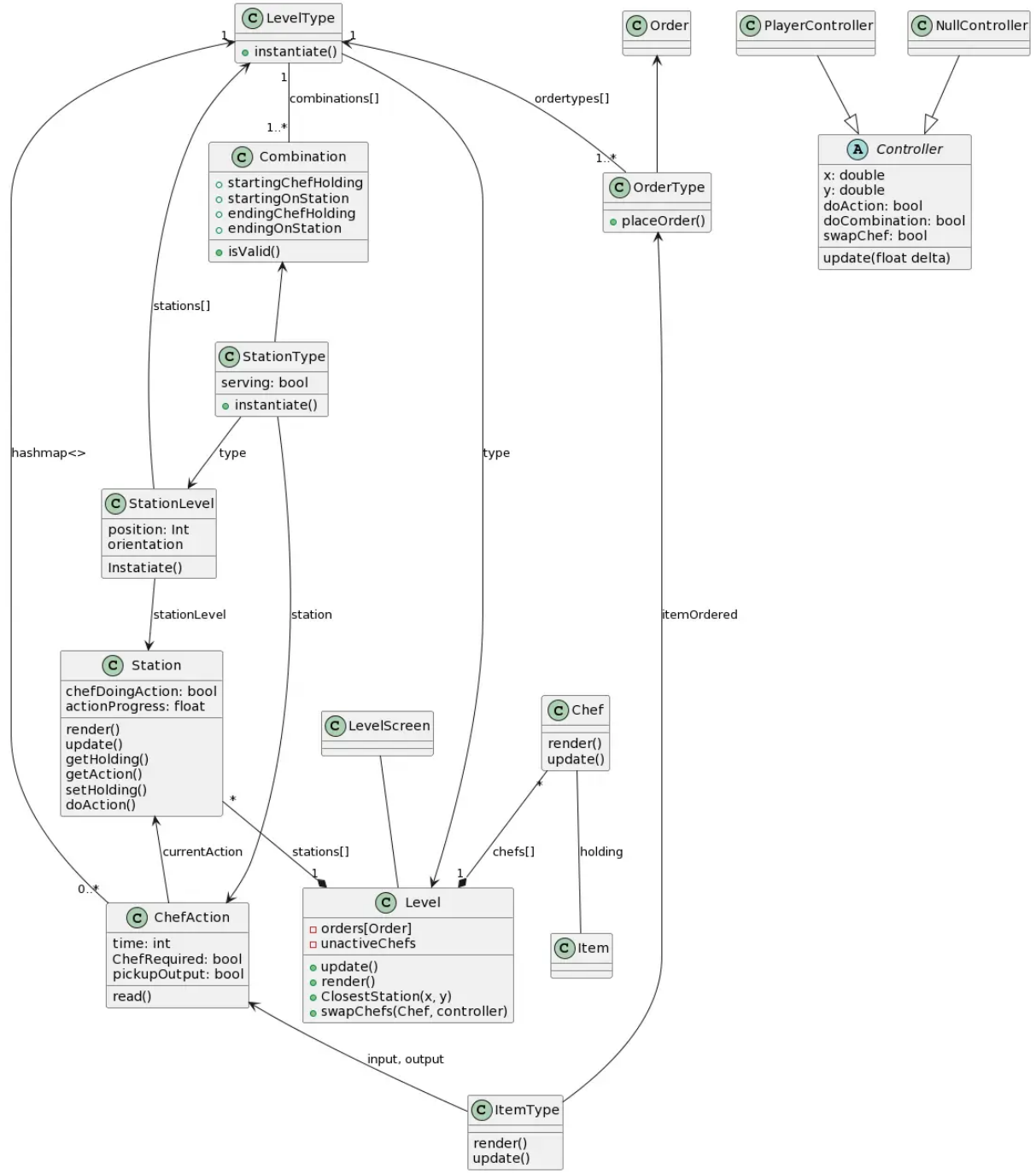
Sequence diagrams represent the flow of data between objects, and as such provides a useful insight into the order of execution of a program.

Another feature that made UML a viable option for us was its ability to represent objects in terms of Java classes. For example we can have abstracts, methods, classes, abstract classes etc.

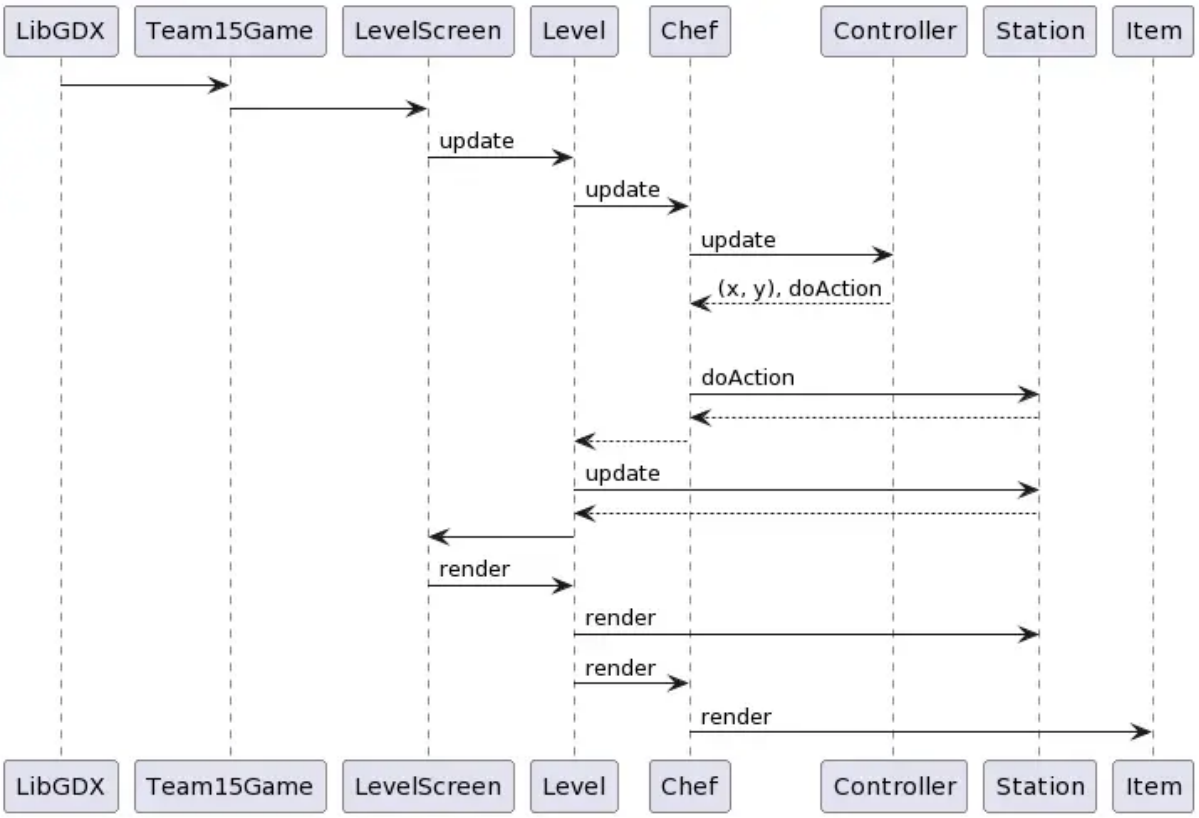
Being 'unified', and industry standard also provides some benefits, as it is likely that other software developers will understand UML and will be able to understand the architecture of our system easier.

The tool we chose to create the UML diagrams was PlantUml, because of its simplicity to use, and the Open-source licence allows us to use the charts without the need to purchase or contact the developers. Having a text-based input allows for much faster diagram creation, but does limit the control that we have over the output, as we cannot rearrange where each object will sit.

Class Diagram



Sequence Diagram



3b)The architecture of the system was based on the requirements that we had previously elicited. These requirements formed the basis of how our game had to function and what it had to achieve in order for it to work successfully. Initially we worked out that the architecture needed to contain a set number of classes to satisfy the requirements. In some cases multiple requirements were satisfied by the creation of a certain singular class. And other times more than one class was needed to satisfy a single requirement. As the project progressed, we realised that in order to achieve all these requirements we would have to change the way in which the architecture was set out, how the classes interacted with each other and create new classes that were able to complete the tasks that other classes were not able to do.

Recipe (D1,D4,D5 on website):
FR_INGERDIENT_COMBINE
NFR_MODULARITY
NFR_READABILITY
The recipe class holds information about the food transformations/reductions. It enables a station to return a new item. This is a necessary class that is needed because it holds information about specific food transformations, which means the code is more modular and readable.

Controller (D2 on website):
UR_MOVEMENT
UR_INTERACTING
UR_SWITCHING
The controller class can find out the current state of the Chef occupying it in the level. Enabling the ability of movement. Without the controller class movement would not be able to occur as the current state and whereabouts of the chef would be unidentifiable.

AIController (D2 on website):
NFR_SCALABILITY
NFR_MODULARITY
FR_SWITCHING
UR_SWITCHING
The AIController class allowed for a version of the chef switching to be implemented, acting similarly to how nullcontrollers do in the current version. This allows for switching to take place within a single class promoting modularity and scalability as the code can be easily modified and multiple chef switches can occur.

KeyboardController (D2 on website):
N/A(without this most requirements would not be able to take place.)
The KeyboardController class allowed for keyboard inputs to be returned. Without the keyboard controller there would be no user interaction with the system so most, if not all of the requirements rely on it.

Order (D3 on website):
UR_ORDER_SEND
UR_ORDER_TAKE
FR_CUSTOMER_INTERACTION
FR_ORDER_COMPLETION
The order class contained the items that needed to be returned to the customer. Without this class the ability to fulfil the requirements listed above would be impossible.

Chef (D1, D2 on website):
UR_CARRY

FR_TAKE_ITEM
FR_PUT_ITEM
FR_MOVEMENT
FR_INGREDIENT_COMBINE
The chef class contains functions that allow it to interact with the surrounding level in order to move. It also has the ability to interact with stations. This is a vital class as it enables all of the chef functions and attributes for its character to be modularised and allows the requirements listed above to be reached.

Station(D1, D4 on website):
FR_TAKE_ITEM
FR_PUT_ITEM
FR_INGREDIENT_COMBINE
FR_INTERACTION_CONFIRMATION
FR_INTERACTION_COMPLETION
FR_INTERACTION_FAIL
UR_FAIL
The station class is the refreshable instance of a station. It contains the item it's currently holding, the chef action being performed, and the progress of the action. It is responsible for the update and rendering of the image for each tick.

Level(D1 on website):
UR_INTERACTING
FR_INTERACTION_CONFIRMATION
FR_INTERACTION_COMPLETION
FR_INTERACTION_FAIL
The level class is vital to having a functional game and reaching all the necessary requirements. The ability to interact is governed by the level along with the current layout of the system. It is necessary to have the level class as it facilitates all of the requirements above.

By using these classes we improve the readability of our code, in addition to this we also add an idea of future proofing, in the sense that we can now add multiple of these classes, whilst maintaining the intended functionality.

We created a number of classes to help with abstraction of code; these included Entity, EntityType, StationType, ItemType, OrderType, LevelType, ItemController, ItemCarrier. These classes helped to:

- Improve maintainability: by presenting a simplified version of the Classes that are instantiated, the code is easier to understand, maintain and update as the project progresses. This can in turn help with preventing bugs from appearing in the code.
- Improve reusability: The code in these classes can be used in multiple places, meaning that code does not have to be repeated in places where it does not need to be. Code can be easily reused across the whole project, in different areas.
- Improve readability: this process means that the code is less complex and long in these separate classes which means that the code is more easily readable which reduces the time that it takes to understand and modify the code, in context's like debugging.
- Improve modularity: The code is split up into a lot more sections which means that it is easier to modify different parts of the code without affecting the rest of the code that may be unrelated. It reduces the likelihood of game breaking errors when adding to the project.

This helped to achieve the requirements of NFR_READABILITY. NFR_MAINTAINABILITY

As the project developed we realised that the initial architecture layout wasn't as suited for our project as we hoped. There were a few major changes as the implementation went further and we realised that some classes were not needed and others were.

Item and ItemType were not necessary, since item did not have a specific location on the level there was no need for both classes: ItemType contained all the values for Item. So we removed the class as it was wasted space and decreased how well we approached the NFR_READABILTY and NFR_SCALABILTY requirements.

Stations need to be able to access the list of StationLevels. We added a new class StationLevel that contains position data for each station within a level
A new StationLevel is created for each station in a LevelType. This has the advantage over using stations as stations have attributes that are not applicable to LevelTypes.

A notable design feature of our project was the incorporation of JSON to retrieve game data, such as items, stations and combinations. We realised that generating most of our game from JSON makes it easier to modify and add new levels to the game and makes the source code more Coherent. The code contains a general case but all the specific items, recipes and combinations are held within the JSON file. This also allows for fewer objects in the code, linking to NFR_READABILITY. The bulk of the JSON processing is performed by the LoadJSON function in the Team15Game class, which loads each file into a respective hashmap. For example, Actions are loaded into a hashmap that maps a string to the corresponding ChefAction. The list of ChefActions valid in the level is processed, using level.json to decide which actions are valid. The valid actions are then passed to the Level class. This means the actions that are valid in each level can be determined by the json file, which allows for easy editing of the game space. This is true for the rest of the objects loaded from JSON (items, orders, levels, combinations, stations), however some are mapped from a string to a list of types. E.g. orders will be entered into a hashmap from {'order-x' : ['way of completing 1', 'way of completing 2']}. The combinations in the JSON file were implemented as a way of combining ingredients to make the recipes, as well as plating certain items. This satisfies the requirement FR_INGREDIENT_COMBINE. Combinations was designed with attributes for what the chef and station are holding at the beginning and end of the combination, which removes the need for processing in the program itself. This satisfies the NFR_RELIABILITY as the processing load is reduced. Because combinations.json changes the items being held by the chef and the station, it also acts as an easy way to pick up and put down items without combining them, meaning there is no functional difference between combining items and placing items, which cuts down on code.