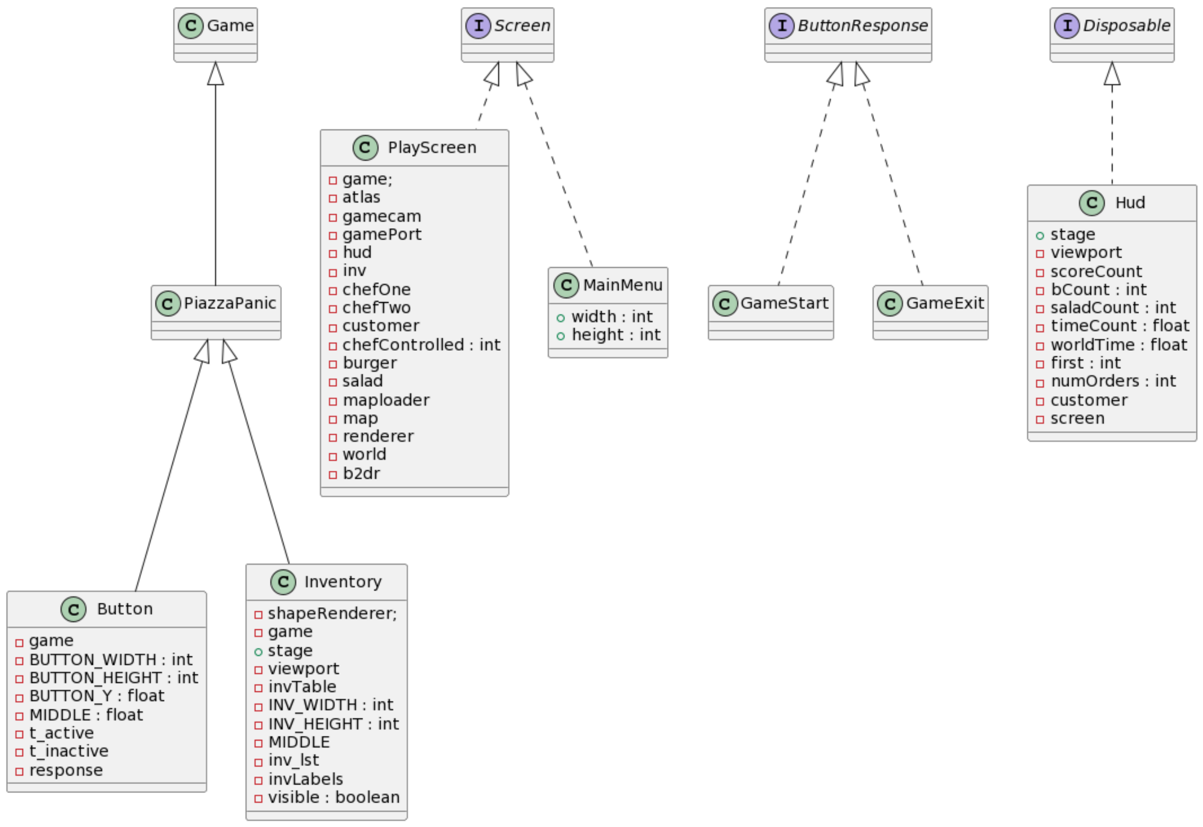
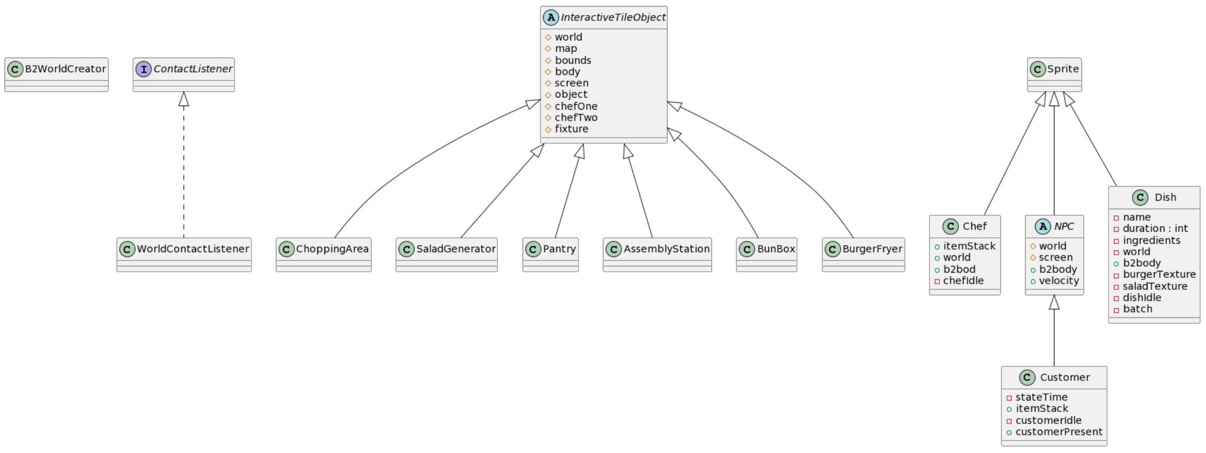
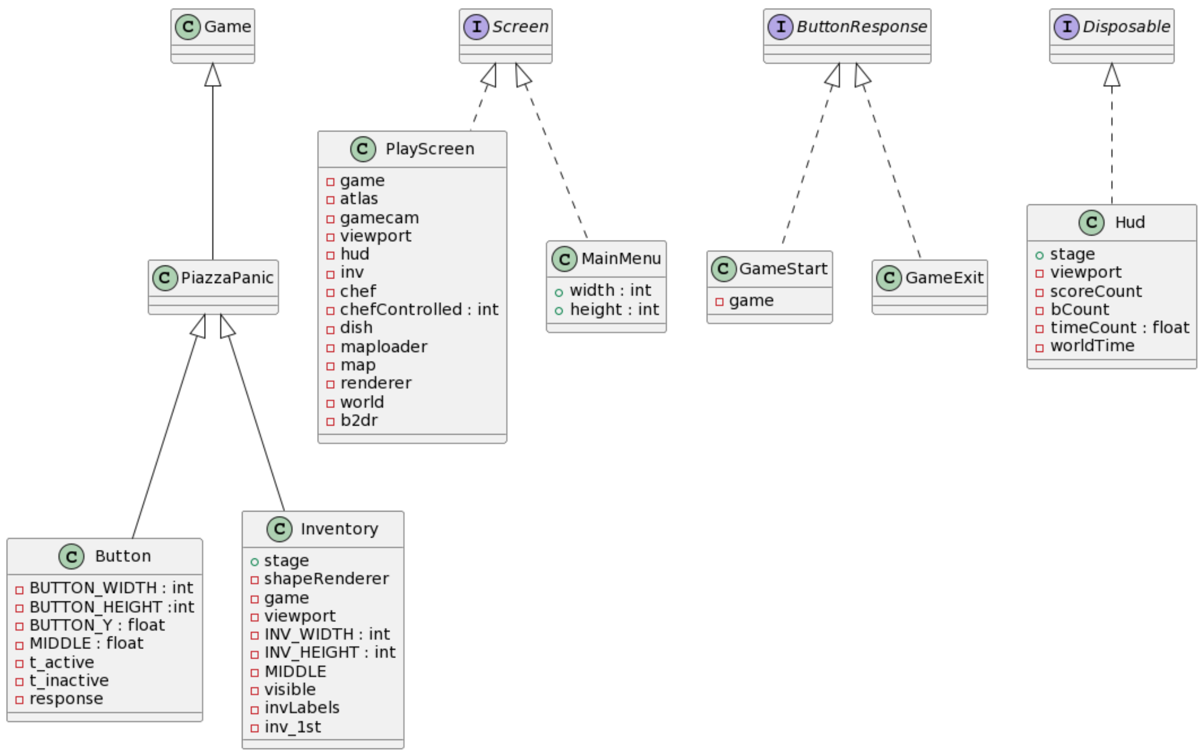
Architecture
3a
For the diagrammatic representations, we have used UML class diagrams to represent the classes, a UML Sequence diagram to show how the user moves through the screens and what information is passed between them. We have used PlantUML to create the Class diagrams because it was more efficient to write the code to create them than to draw them manually in some other application. We created interim versions of the class diagram to help our understanding, you can see how the classes have evolved over the course of development. The other diagrams were created using lucid charts as we have had previous experience in using this tool, and it is quick to create simple diagrams.

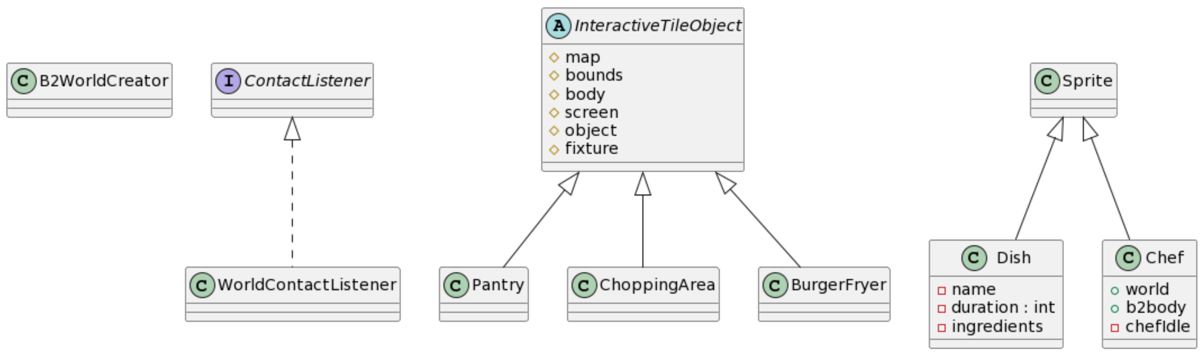
Structural diagrams :




B2WorldCreator
ContactListener
WorldContactListener
InteractiveTileObject
world
map
bounds
body
screen
object
chefOne
chefTwo
fixture
ChoppingArea
SaladGenerator
Pantry
AssemblyStation
BunBox
BurgerFryer
Sprite
Chef
itemStack
world
b2bod
chefIdle
NPC
world
screen
b2body
velocity
Dish
name
duration : int
ingredients
world
b2body
burgerTexture
saladTexture
dishIdle
batch
Customer
stateTime
itemStack
customerIdle
customerPresent

Initial diagrams:





3b

We began the design process for the project with an initial discussion of the informal requirements we had laid out after our first meeting with the stakeholder, and how we could apply them to an architecture that would be appropriate to implement. From this discussion, we had a basic idea of how we would implement the game map, how information such as the orders and items the chefs were holding would be displayed, and how interactions would work. We chose to employ an Object Oriented approach to the architecture throughout the software as it lends itself to modularity, which is key given the collaborative nature of the project and means that it will be easier for other teams to understand and extend our code later on. Our initial planning for classes involved physical pen and paper diagrams, such as the CRC cards which are available to view on our website, which we later translated into more detailed UML diagrams as seen in 4a. These CRC cards were useful in establishing the classes on which the rest of the game would be built upon. They were quick to make and easy for everyone to understand, even those who weren't on the implementation team. After this, we created initial classes such as Chef in order to get an understanding for how the IDE and LibGDX framework worked. Our first prototype of the game was a simple kitchen in which two instances of the Chef class could move. From this foundation, we were able to create new classes and add new methods and attributes to the Chef class in order to implement features. This is the principal advantage of OOP, in that it is easy to build from a relatively simple foundation to the final version. This worked well with the Agile methodology we adopted. Since we used object oriented programming, we have been able to create UML diagrams showing the classes and their methods and attributes to better understand the architecture.

We will now provide a systematic breakdown of the architecture of the code, and how it relates to the requirements that have been set out in the Requirements deliverable.

User Requirement **1.1** concerns the movement and control of the chefs. The requirement **1.1.1** and **1.1.2**, about interacting with the stations is handled on a case by case basis by the respective station classes within the code. Requirement **1.1.3** (concerning the chef picking up items) is implemented as an inventory system which is coded in the *Chef* class as a method, with further implementation details in the *Hud* class, which handles other interface related tasks and will be mentioned again in this document. Likewise, requirement **1.1.4**, concerning the interaction between a station and the items the chef using it is carrying is also implemented in the *Chef* and appropriate station class, *ChoppingArea* for example. The requirement **1.1.5** describes the movement of the chef. Movement of the chef is implemented within the *PlayScreen* class, which continually takes inputs and adjusts the x and y coordinate attributes of the appropriate instance of the *Chef* class accordingly.
User requirement **1.2** describes the behaviour of the customers and defines the rules for how they should order items in the game. We have implemented the customers - thus fulfilling requirement **1.2.1** - in the class *Customer*, which interacts with the *PlayScreen* class in order to display their sprites. Code regarding the behaviour of the customer, such as when it appears and randomly selecting their order - is implemented within the *Customer* class itself. *Customer* inherits from the class *NPC*, which we created in order to define the sprite.
User requirement **1.3** details the pantry, which is implemented like all other stations as another class called *Pantry*, with its behaviour (requirements **1.3.1** and **1.3.2**) and rules for collision implemented within it. All stations inherit from the *InteractiveTileObject* class.

User requirement **1.4** and its subclasses concern another station: The cutting station. This is implemented as a class *ChoppingArea*. The fixed amount of time described in **1.4.1** is implemented within the *Dish* class, which specifies the time for individual ingredients on a per recipe basis.
1.5 is similar in which it details the frying station for grilling burgers. It is implemented as a class *BurgerFryer*. Again, **1.5.1** - the fixed amount of time - is implemented as a variable in *Dish*. Validation specified in **1.5.4** and **1.5.5** is provided in the code for the *BurgerFryer* class.
Again, similarly, requirement **1.6** describes the toasting station, which is implemented as the *BunBox* class which extends *InteractiveTileObject*. Implementation again is the same, extending *InteractiveTileObject*, with the time in **1.6.1** implemented in *Dish* and the logic for validation in *BunBox* itself.
Finally in this vein, there is a station for assembling ingredients into dishes, implemented with the class *AssemblyStation*. This satisfies requirement **1.7** and its sub requirements concerning its behaviour. Again, this class inherits from the *InteractiveTileObject* class. As mentioned in **1.7.3**, this station's behaviour should be implemented as a stack, which you can see.
User requirement **1.8** moves on from the stations and begins to describe the behaviour of the recipes which the customers will order. Each recipe is implemented as an instance of the *Dish* class, which has variables for defining the name, ingredients and time it will take to make the recipe. It does this using elements of JSON.
The requirement **1.9** concerns the ending screen, which is implemented as another class *EndScreen*. The sub-requirement **1.9.1** states that the timer must end and its result be displayed. This timer is implemented within the *Hud* class, and its value is passed from this class to *EndScreen* to be output, satisfying **1.9.2**. Finally, the two buttons described in 1.9.3, exit and restart, are implemented as instances of the *Button* class.
The final user requirement **1.10** describes the homepage that is displayed when the game begins. This was implemented as a class *MainMenu*. As per the requirements **1.10.1** and **1.10.2**, it contains 2 buttons to begin the game and exit, which are implemented as instances of the *Button* class.

System requirement **2.1** describes in more detail the movement and behaviour of the chef sprites. **2.1.1** considers the hit collection, which we implemented using the LibGDX *contactlistener*. It checks for contact between fixtures of the chef sprite and the counters and stations. As previously mentioned, the WASD movement (**2.1.3**) was implemented within the class *PlayScreen* (of which the game itself is an instance of). **2.1.2** has been handled naturally from the construction of the code - invalid inputs are ignored.
2.2 and its sub requirements detail the random nature of the customers orders. We implemented counters which are described in **2.2.1**, and their behaviour described in **2.2.1.1** and **2.2.1.2**, as labels within the *Hud* class - an overlay that is displayed over the game screen. The logic for the order generation is split between the *Customer* class and the *PlayScreen* class. The random element of the orders, detailed in **2.2.2** and **2.2.3**, was provided through the *addOrder()* method in the *Hud* class, which randomly increments the burger or salad counter when an instance of *Customer* calls this method (when they arrive).
2.3, again this system requirement, like its user counterpart, concerns the handling of items that the chef is holding. This requirement specifies that it should be implemented as a stack (**2.3.1**). This stack is implemented within the *Chef* class, and communicates with other classes when the chef interacts with stations, which mirrors changes within the *Hud* class as

specified in **2.3.2** and **2.3.3**. Like WASD movement, the keyboard controls to display the inventory have been coded in *PlayScreen*.
System requirement **2.4** goes into more detail on the hit detection, which as previously mentioned, was implemented using the predefined *contactlistener* feature of LibGDX. The basic logic of collision (**2.4.1** and **2.4.2**) were defined in the *PlayScreen* class. The “fixtures” that make up the chefs were defined within the *Chef* class and mean that hit detection is handled neatly and simply, with the slight overlap with counters as detailed in **2.4.1.1**
2.5 concerns the use of items in preparing recipes. This is the core mechanic of the game and as such the underlying architecture is perhaps the most complex. All stations which inherit the *InteractiveTileObject* class play some role in preparing the ingredients for use in dishes, which are finally assembled at the assembly station, implemented as the *AssemblyStation* class. This checks that all necessary items are present in the users inventory and produces the correct instance of the *Dish* class.
System Requirement **2.6** talks again about the implementation details of the main menu as previously mentioned in this document when we talked about user requirement **1.10**. It also notes the non functional requirement **2.6.1** which specifies that the layout should be as simple as possible, which we tried to keep in mind when designing the interface for *MainMenu*.
2.7 is a more general requirement concerning the performance of the game, which we tried to heed by ensuring logic was as simple as possible, minimising the amount of loops or processing heavy tasks in code. This was of particular importance in coding the behaviour for the *Chef* class and the classes for stations, and the interactions between them.
Similarly, the system requirement **2.8** and its sub requirements also focuses on the performance and scalability of the program, which we tried to account for in a similar way, also paying extra attention to the UI again, as mentioned in **2.8.3**.
Focusing more on the UI, **2.9** gives more detail on how we should improve the UI, given the context of where the game will be played, and also considering those users with visual impairments. You can best see this in the *MainMenu* class and its interface - we used simple colours and high contrast, which links to the requirements **2.9.1.1** and **2.9.1.2**. Additionally, you can see in *PlayScreen* how we have tried to display information graphically (**2.9.2**)
2.10 concerns the end screen, again implemented as a class *EndScreen*, with the two buttons (**2.10.2** and **2.10.3**) implemented as instances of the *Button* class. The output of the timer value is handled by logic within *PlayScreen*.
Finally, the system requirements **2.11** and **2.12** focus on the code itself - the quality of it and the style that it is written. Hopefully you can see throughout our code that we have paid attention to ensuring that it is clear and maintainable (**2.11.2**) and has comments throughout as described in **2.11.1**. We also laid out in requirement **2.12** the details of OOP that we wished to employ. It is clear that we have made frequent use of classes, such as *Chef* where each individual chef is an instance of this, which fulfils requirement **2.12.1**. We also used encapsulation throughout the code (**2.12.2**), wherever you see a private attribute, this is where we have used encapsulation for robustness. Finally, requirement **2.12.3** has been met, wherever the “extends” keyword is used within the code, showing a subclass inheriting from its parent class, such as *BurgerFryer* inheriting from *InteractiveTileObject*.