Architecture

Team 16

CatepillaDevelopment

Yousif Al-Rufaye
Joe Fuller
William Gracie-Langrick
Jack Hardy
Bailey Uniacke
Ben Young

3) a)

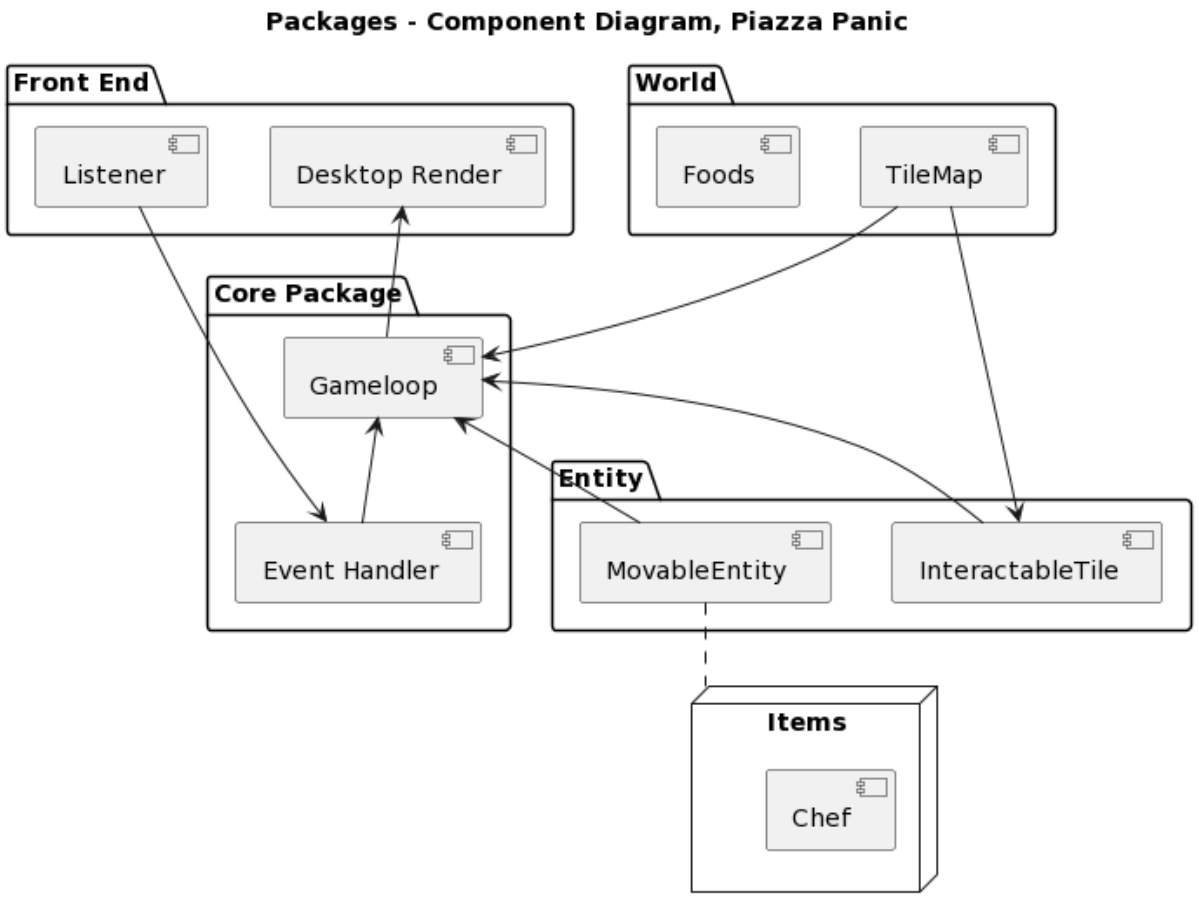


Fig. 1. - Package Diagram v.2

The above is a package diagram of the related overarching unspecific functionalities of our game. This was created in UML on planttext.com, this allows for easy visualisations of the package structure needed.

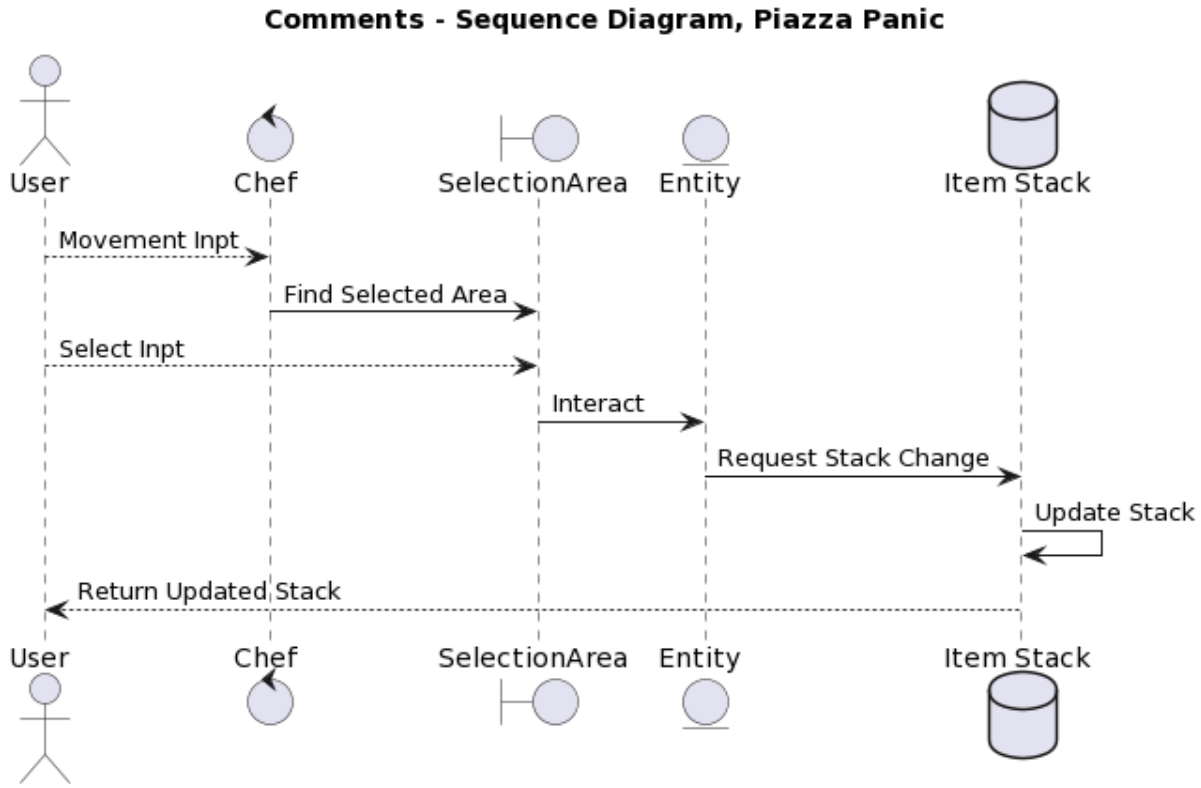


Fig. 2. - Sequence Diagram

This sequence Diagram shows the surface level input and output of a user's interaction with the program. What the user chooses to do and what impact this has on what is stored in the system architecture.

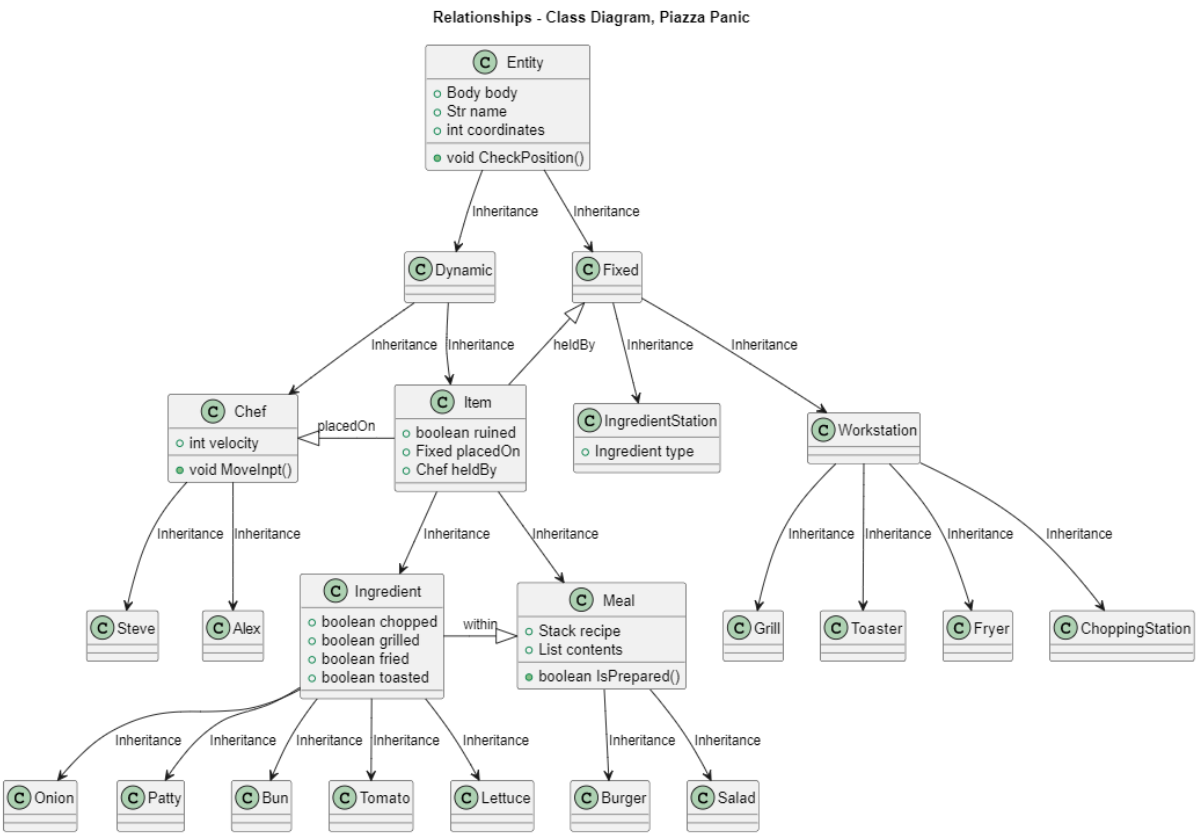


Fig. 3. Relationships Class Diagram, Piazza Panic

This class diagram illustrates the class structure of the project using a UML relationship diagram. Higher level classes are “abstract classes” this has not been illustrated in this version. This tree-like structure made by UML allows us a solid foundation to base our project structure on.

3) b)

The architecture of the project was debated heavily at the beginning of the project and it has been difficult to agree on one path. Several options were discussed and the results shown in part a) are an amalgamation of many different ideas that may not always be consistent between design iterations and implementation as we have had to take some shortcuts to deliver the project in a timely manner.

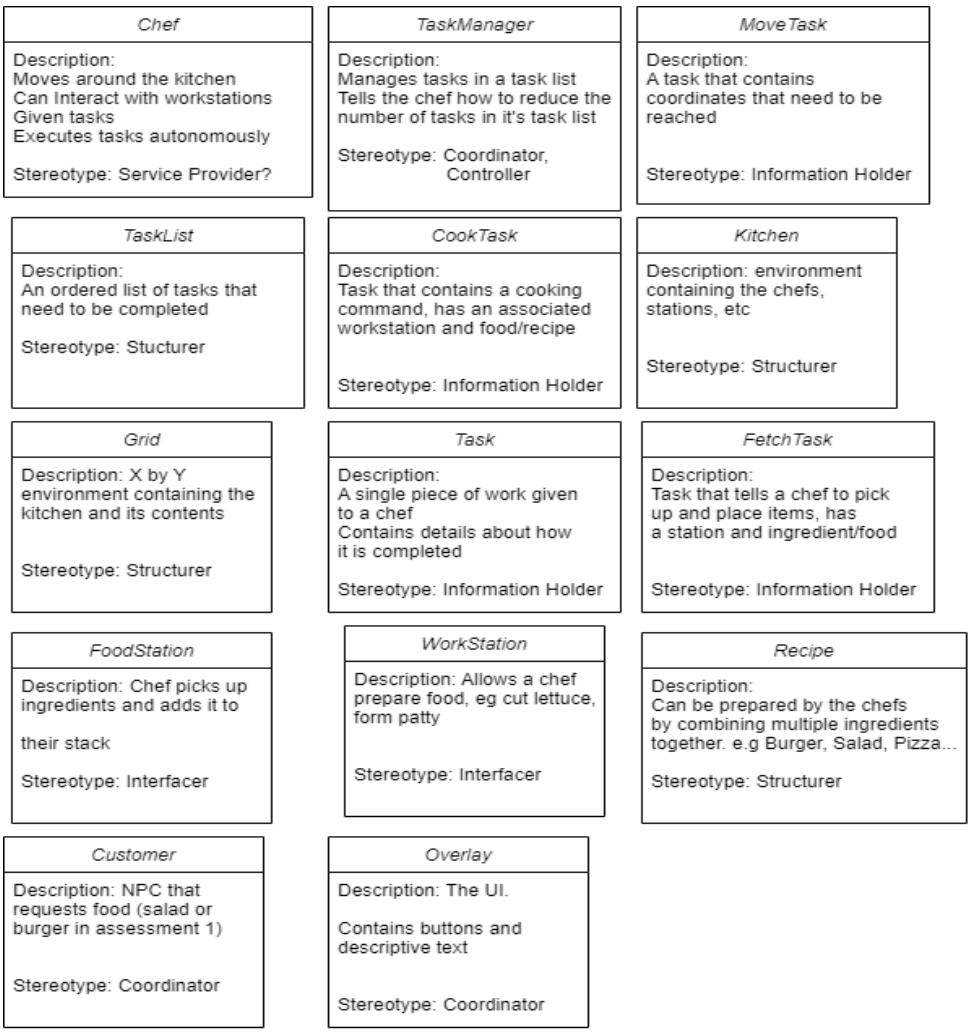


Fig. 4. - Stereotypes/Entity Uses.

Above, is the basis of our group discussion of architecture abstraction. These are a combination of CRC cards where we are looking at what elements can be abstracted to classes and stereotypes which focus on what areas of the system elements will need to be implemented in.

These cards were designed as the basis of the class structure architecture for our game, however after deciding on our library, LibGDX, which was structured based on an entity component system. We initially used the ideas created and solidified them as an entity component diagram. However, we found this structure too difficult to work from. As none of our team members had any experience with this structure before, and the investigative coding we had done to build the foundations of our game did not suit this structure. This led us to use a Class based structure; as shown in figure 3.

The class structure has a tree like structure, which inherits from the Entity class, these high level classes are abstract and only exist to give our format structure. The class diagram shows the relationships between objects, describes both the values that the objects have and the values they hold and the functions/procedures that they can run.

Our architecture has three main classes that inherit from the Entity class; Chef, Item and WorkStation. Chef and Item are dynamic and can change states, for Chef this is the velocity and activity. And for Item this is the location, either held by or placed on (Chef and Counter respectively). The workstation does not change state. It just exists as an entity in order to be interacted with. The different types of Workstation all inherit from the WorkStation class. Though they have different names they all have the same basic functionality. Similarly to Chef, where multiple entities are created so they can have different controls. (This has not yet been implemented). Items can come in the form of an Ingredient which can either be fried or chopped etc. These Ingredients combine to make a meal, a meal is also treated as an item that can be placed or held.

This Class Architecture allows us to illustrate all the functionalities needed.

The sequence diagram aims to highlight the very simple input system that the game will incorporate. There are only two input loops; movement and selection. This reduces the amount the user must hold in their mind to make decisions. Many of the checks are done automatically, including checking what is selected, whether the stack is full/contains the ingredients for a recipe. Which makes these simple controls usable. The Listener is perpetually checking for input from the user, there is only one state that the system can be in, checking for inputs. All inputs have instant change and feedback.

The package diagram illustrates how the more abstract entity diagram translates into a file structure that we can apply. The structure is based around the central node Game loop with several branches coming off it as it produces results. And nodes going into it with inputs for it to process. This structure has low cohesion and high coupling, which does not make it robust to changes, but this was the way we thought would be easiest to implement. Especially given our limited time.

Changes over time.

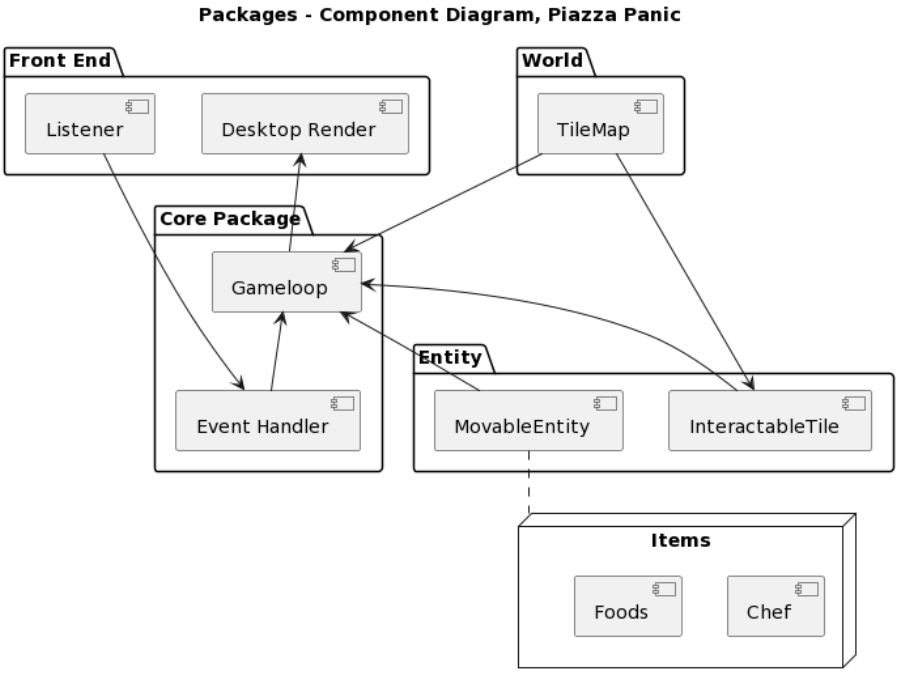
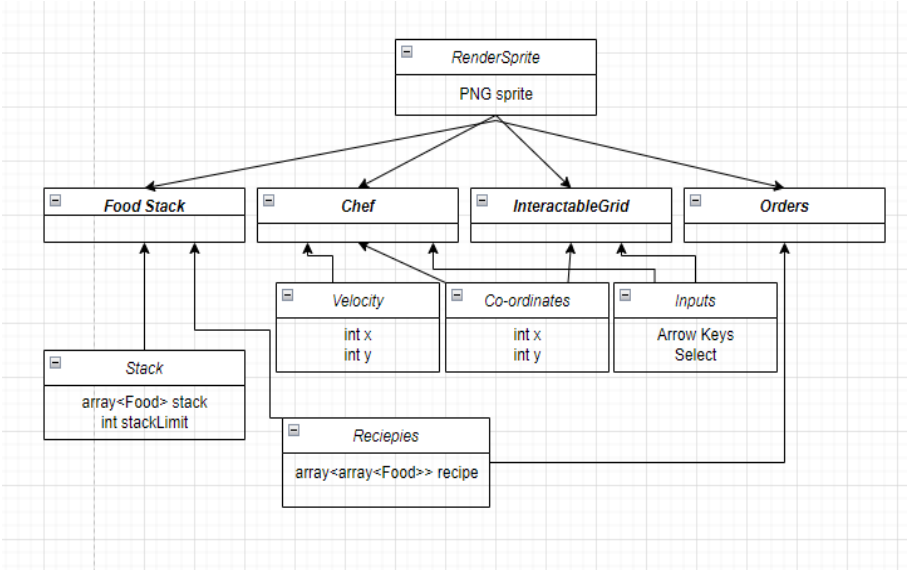


Fig. 5. - Package Diagram v.1

As we began to implement our game we realised how difficult it would be to implement each Food item as its own entity.

This is illustrated in the diagram beside, where we included foods in the MovableEntity diagram. We would need to calculate collision and velocity etc. for the Food items and this would be extremely complicated; especially with the limitations of LibGDX. Instead we decided to treat Food items as a value that can be transferred between different stacks. i.e the Chef stack, or placed on an interactable tile. This can be implemented much more easily, and the food can be simply represented in the abstract form rather than having to be entities.

Fig. 6. - Entity Component Diagram



An Entity diagram is used to represent the Entity component system architecture of the project. This diagram was created in Draw io. Which allowed for flexibility as UML struggles to create a coherent diagram that is this complicated. As explained above, this structure was not fit for purpose and had to be scrapped.