JPC System Design

# System Structure

The system structure we used for our project is MVC (Model-View-Controller) architecture. Although we had not initially decided on a concrete structure during our design phase, we found this was the model that best fit our code and requirements. This is because it allows for a separation/decoupling of specific elements of the Simulation, allowing us to add new features and methods to classes without heavily impacting others.

As the workload was initially split into Pathfinder and GUI, maintaining the MVC structure meant that while GUI (View) was being changed there was little chance that changes in the Pathfinder (Model) would affect it. To further prevent any overwriting/errors the code was stored on a Git repository where we could pull new code and push when communicated with the team making for swift and efficient amendments.

# File I/O

For the file input and output the application has one example of each, these being importing simulation files into the simulation and exporting the report as a .txt file. Both of the methods involved create a JavaFX FileChooser with file type filters to prevent any formatting and loading issues. Once a file is selected two different methods are called based on input or output.

# Warehouse Controller & Main Class

The application, utilising JavaFX, is run through a Main class which creates the WarehouseController and WarehouseSimulation which allow the application to function. We made the decision to keep the Main class as part of the View of MVC as it does not actually provide control over the application and more accurately sets up the window of the application. Therefore the true Controller is the WarehouseController class, which has access to all important classes such as the simulation and the floor from which it can call methods to affect the Model.

The controller allows for event handling of the GUI that is set up in View as well as the models that are being represented there upon change. For example, once a tick has occurred on a simulation there will be a graphics update from the Controller which calls several update methods in the View based on values set in the Model.

# GUI

The Simulation GUI consists of two main views, these being warehouse and report views which can be toggled by menu items at the top of the application. To meet the requirements of the spec we needed to have a textual and/or graphical representation which is the reason we have these two views. The user also had to be able to perform ticks and generate random Orders to be performed which has been added to the GUI in the form of buttons and a text input field where the user can specify how many ticks they wish to perform at once.

The GUI is part of the View in line with the MVC structure as the necessary classes to simulate the visuals of the Warehouse and report are updated through the WarehouseController taking on and calling methods.

# Model Classes

For our simulation the Model structure represents all of the cells of a warehouse, the actors that are present and the various classes and interfaces that connect everything together and allow them to perform a variety of actions. For the GUI to be created there had to be some basic structure for the model to be projected onto it; so in the initial development phase, skeleton classes were created to allow the two main chunks of code to be constructed. This skeleton was easily made using the initial UML diagrams that had come out of the design phase.

As identified in the design we knew that we needed a generic class to be held in the Cells of the warehouse floor which quickly led to the development of the abstract class Actor. By having Actor as a superclass we negated a large portion of code duplication as we could set up necessary parameters and methods that would be common in all Actors. Similarly we chose to create an Abstract simulation class, which is extended by the WarehouseSimulation class in our application, as this allows for further implementations of the simulator in the future where requirements for the applications use may differ.

We also had interfaces such as GridLocation which is implemented by classes that need to be able to have and output its location as well as handle adding/removing of Actors within itself. By using an interface we are allowing for potential multiple inheritance in the future where a class similar to Cell needs to be able to perform the aforementioned tasks and promotes lower coupling.

Whilst developing the code, encapsulation has been applied to protect specific class data and methods from access of potentially harmful code. By doing this we are allowing important fields to be protected whilst also allowing data manipulation to occur through referenced public methods. Some fields in our classes have a static variable meaning it would be shared and remain the same throughout all instances of a class. This was useful for incrementing UIDs (Unique Identifiers) for mass initialisation of a class object.