DC2300 Coursework Report

TEAM 1: WILLIAM CREWE, COLM MALLON, MHAIRI MCDONALD

1 ASSESSING REQUIREMENTS

Our work began by examining the specification using the noun/adjective method; entities and behaviours were identified by extracting details from the problem description to formulate a basic idea of what needed to be modelled, and the relationship between different parts of the system. This allowed us to understand where the bulk of the work lay, and it meant we could assess how to appropriately delegate tasks according to personal strengths and areas of weakness.

We saw a clear split between modelling the class entities and making a start on the GUI; Will volunteered to do the latter whilst Colm and Mhairi began mapping out the different classes required and how they would interact with each in the best way possible, based on our Noun/Adjective analysis. The importance of fully understanding the problem brief and designing a structure in the initial stages was understood, and it meant we could best get a clear picture of how the entire system would develop as we began implementation.

2 UML DESIGN

It was fairly simple to grasp the individual parts of the system, i.e. the model entities, the GUI, the path finding and cost estimation strategies – but we could see the challenge of how these individual subsystems would work together to produce the final implementation. We broke our system down into manageable chunks:

- File I/O
- GUI
- Model entities
- Path Finding/Cost Estimation Strategies

2.1 FILE I/O

Our warehouse configuration allows for manual setting of the various simulation parameters; a text file is loaded into the program in the .sim format and each parameter is handled accordingly via a FileLoader class and various helper classes. FileLoader parses the File given by the program, and pieces it based of parameter type, i.e. information pertinent to a Robot is added to an ArrayList, which is then passed to a ConfigRobot to ‘set up’ the Robot object ready to be used by the Simulation/GUI. This also applies to the other actors in the program, such as PackingStation and StorageShelf. It allows for proper separation of functionality between handling the input file, parsing

the information, and using that information to construct the objects before passing them to the simulation.

Class ActorConfigConversion allows for appropriate conversion from Config<Object> to <Object> and vice versa to facilitate both the information set up from the input file, as well as outputting parameter information as Strings for the generated output report.

The design for File I/O is demonstrated in the warehouse.io/config UML.

2.2 GUI

The main simulation is initiated by the Main class which calls the WarehouseScene.fxml resource and initialises a WarehouseController; this is where the bulk of the GUI 'work' takes place.

The Warehouse is created based on the parameters passed in by the ConfigFile, however, checks are in place to ensure the input file is in a valid format. If it isn't the user is alerted and asked to provide a file in the correct format. This means that incorrectly formatted input files won't crash the program without the user understanding the issue. The public static file primaryConfigFile controls the initial state of the GridMap created by the WarehouseController.

As the simulation runs, the grid (and the Actors on it) are updated via the Simulation class; Simulation sends Actor behaviour updates (i.e. ticks) to an updated ConfigFile which then overwrites the state of the original map configuration. This occurs with every iteration of the simulation to update the map and the actors' positions and actions within the simulation.

2.3 Model Entities

The Actor Interface allows the definition of generic functionality amongst all the actors in the simulation. Robots, packing stations, charging pods, and storage shelves all 'tick', so it made sense to define this in a common interface, and have each entity override with their own specific behaviour accordingly.

Location models the simulation grid position of each actor, and Warehouse models the wider picture, i.e. placement of the objects in their locations on the map.

The Simulation class models much of the program functionality and is included in the warehouse.model UML to demonstrate its relationship with how the Warehouse is controlled, and with its list of Actors.

The Order class contains information relevant to each order passed in by the .sim file (via FileReader/ConfigFile/ConfigOrder) and this is populated as a list within the Simulation.

2.4 Path Finding/Cost Estimation Strategies

The ManhattanCostEstimator class calculates the cost associated with moving to each position on the map. The overall view of each cell's cost is modelled via a HashMap containing the Location as a key, and an Integer as the value representing the cost. A formula representing this is used to create a 'cost map' which can then be used by the PathMapper class to determine a path for the Robot to take.

PathMapper uses the current location of the Robot to check the cost of the adjacent locations based of the aforementioned ‘cost map’. Once the cell with the lowest value, and therefore cost, is found, the ‘bestLocation’ is returned for the Robot to move into.

This class also handled conflicting paths with other Robots by setting ‘obstructions’ on the map – this is done by iteration over all the locations in the currentMap and setting position containing other Robots to ‘obstructed’ i.e. unavailable for the Robot to move into.

3 DESIGN CONSIDERATIONS

In the design for this simulation, we aimed to keep cohesion high by ensuring classes and methods fulfilled specific tasks, and that we didn’t overload classes by piling in functionality. For example, the way I/O has been handled means the file reading, parsing, object construction, and simulation map creating associated with the input file are separated into different classes dealing with the information in an intermediary way – specifically the flow from FileLoader, ConfigFile, Config<Object>, Simulation etc. This modular design allows us to adjust different aspects of the program without having a drastic effect on how each of these entities/functions operate.

We have also adhered to good practice by keeping classes loosely coupled. All fields and methods are declared with appropriate access modifiers to prevent direct access of attributes between classes. Information such as this is accessed properly via get/set methods across the entire program.