Group 10: Factory Simulation

Devin Shingadia | 170036317

Jacob Williams | 170050173

Mohammed Hamza Zaman | 170107477

Miraj Shah | 160050110

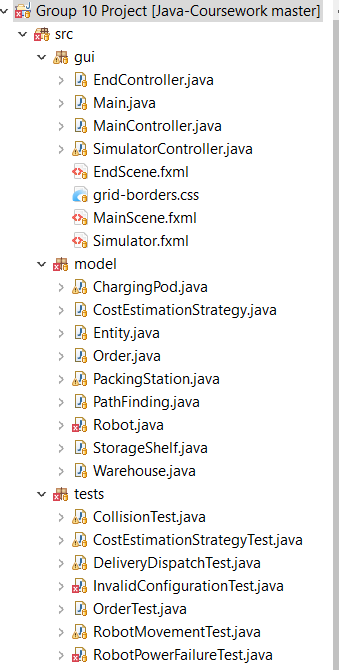
Christos Dolopikos | 170116343

Vivek Bhukhan | 159087932

System structure at the package level

The system is composed of packages, in which there are classes that are loosely coupled together. The usage of packages enforces weak encapsulation as it separates classes based on their functionality. Classes that have similar purposes are put into the same package.

There are three packages that have been created: The GUI, the Model, and the tests.

**Breakdown of the project**

The GUI package is composed of classes that utilise the JavaFX platform for displaying the simulation.

* FXML files for user interface
* Controllers for FXML Files
* CSS File for rendering grid borders

The model package is composed of the underlying logic which controls how the simulation operates.

* Distinct classes for each entity displayed in the simulation
* Classes for Cost estimation and path finding

The testing package is composed of the JUnit test classes which test the logic in the model package.

## Model Package hierarchy

Entity

Cost Estimation Strategy

Pathfinding

ChargingPod

PackingStation

Robot

Order

Storage Shelf

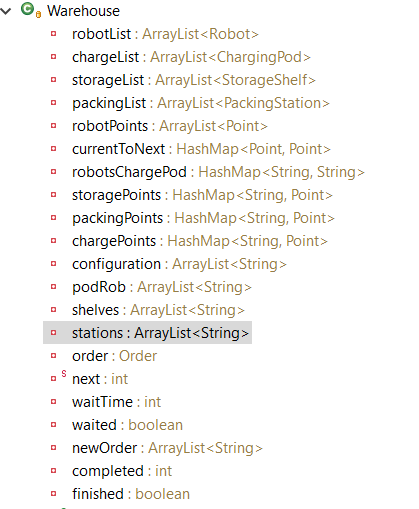
Warehouse

The model package is where the main logic is stored. As shown above, we have decided to make entity an interface class, as it enforces the implementation of the common methods in the entities of the grid.

## Design patterns used

**The immutable pattern**

One of the patterns that was used in the development of the piece of software is the immutable pattern. The immutable pattern ensures that the state of an object, which is often accessed by multiple objects, does not change once it is instantiated. This means that the values of the fields regarding an object that is shared, does not change, meaning that the state of an object can never be affected once It has been instantiated.

* The warehouse is a major component of the simulation, it is where a large portion of the implementing logic is held.
* Once the warehouse is created, we do not want the fields to change often. This design choice means that, without accessor methods, other objects cannot change the attributes in the warehouse class.

As shown in the screenshot, these fields in the warehouse class are private.

**Interface pattern**

Another design pattern that we have decided to use is the interface and abstract pattern. This pattern is used when you need to keep client classes independent of the server classes, and ensure that the behaviour is consistent between server classes. This means that the entity class cannot be instantiated, and that it is composed of abstract methods, forcing each subclass of the entity class to be a concrete class.

We have decided to make the entity class an interface, and the subclasses storage shelf, robot, charging pod, and packing station inherit from entity. As the entity class is an interface, each subclass of this interface is an extension of the superclass, as they implement the common abstract methods, whilst also being a specialisation, as the subclasses apply their own unique parameters and methods which are specific to their purpose- this also enforces cohesion to some extent.

One of the main reasons as to why the interface pattern has been implemented is to ensure that there is no repeated code. The interface pattern is composed of abstract methods. The abstract methods are required to be implemented into every entity that inherits from from the interface class.

The interface class entity ensures that certain methods are implemented, meaning that the entities share common attributes and functionality.

**Cohesion and coupling**

In our project we have decided to keep cohesion high, and coupling low.

High cohesion has been achieved. We have made sure that each class is in charge of its own methods, and that there is no repeated code- no two classes use the same piece of code. This means that each class is unique to some extent.

The data structures that we have used is specific for certain classes, such as the warehouse keeping track of the coordinates of each entity, or the orders class keeping a list of every order. These data structures are not repeated in any other classes, meaning that high cohesion is achieved. For example, the warehouse class is quite cohesive. The warehouse class holds all of the fields and methods that are relevant to the operation of the warehouse itself. For example, it holds the collections for the robots, charging pods and the packing stations.

Coupling has been kept to a minimum. One of the ways in which coupling has been kept to a minimum is by making the design choice to not making fields static. Furthermore, accessor methods have been implemented for the private fields, meaning that the field itself is not being accessed directly.

We have kept coupling to a minimum through keeping cohesion as high as possible. As coupling and cohesion is inversely proportional, it means that as cohesion has been increased, it means that each class is more responsible for its own data. This means that coupling is reduced as classes only use their own data wherever possible.