FINAL PROJECT

Md Ibrahim Ullah – 2295025, Team Leader
Chi-Tao Li - 9730157
Mark Benedict Muyot - 2295022

CONTEXT

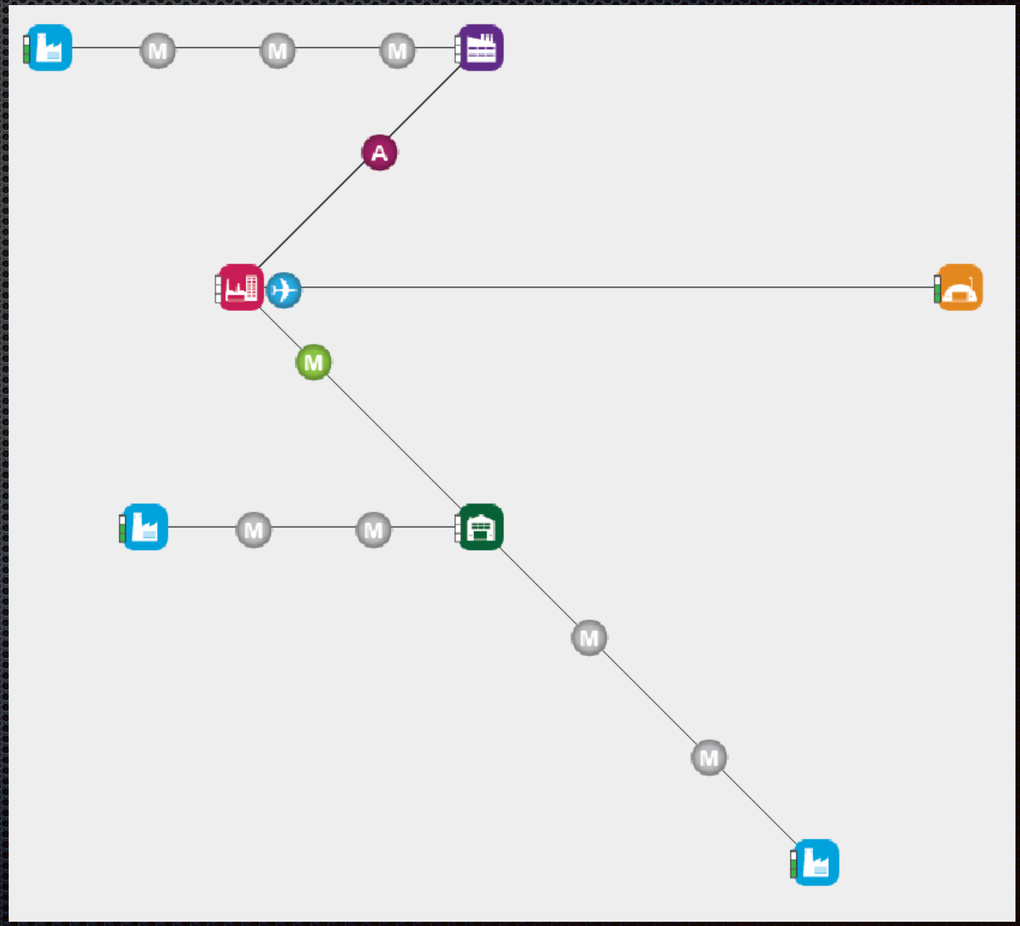
- In industry, the manufacturing of products is done following a series of successive or parallel operations. The organization of these operations constitutes a production chain that the manufacturer generally aims to optimize.
- A production line is made up of several stations, called production plants. Each production plant assembles or produces a specific component.
- To produce a component, a production factory may require components produced by other factories.
- For example, in an aircraft assembly line, one production plant assembles the engine while another plant produces the cockpit. Another production plant assembles the airframe of the aircraft using other components such as the fuselage, wings and landing gear. These components are themselves produced by other factories.

PROJECT DESCRIPTION

- This project consists of designing and implementing an application that simulates a simplified production line of an airplane manufacturer.
- In fact, to maximize production flow and chain reliability, manufacturers use simulation tools.
- A simulation tool makes it possible to model the production line and its various parameters (example: types of equipment, robot or personnel operators, simulation period, etc.).
- The manufacturer can then carry out several simulations with different parameters and thus identify the configuration of the chain which makes it possible to achieve his productivity objectives.

PROJECT DESCRIPTION

- For simulation purposes, the production chain will be represented by a network.
- In this network, each node corresponds to a production plant.
- A path connects two production factories where the destination factory (client) consumes components produced by the departure factory (supplier).
- The assembled components are transmitted from one factory to another through the appropriate paths.



PROJECT DESCRIPTION

- In this simulation, we represent 5 types of plants, and 4 types of products:
  - Metal plant 
  - Airfoil plant 
  - Engine plant 
  - Aircraft plant 
  - Warehouse 
- Metal 
- Airfoil 
- Engine 
- Airplane 

PROJECT DESCRIPTION

- Each plant has:
  - Output product type
  - Input product(s) type(s)
  - Input product(s) quantity
  - Production time

#Example: Aircraft plant 

Output type 

Input type 

Input quantity Qty4 + Qty2

Production time 100 time unit

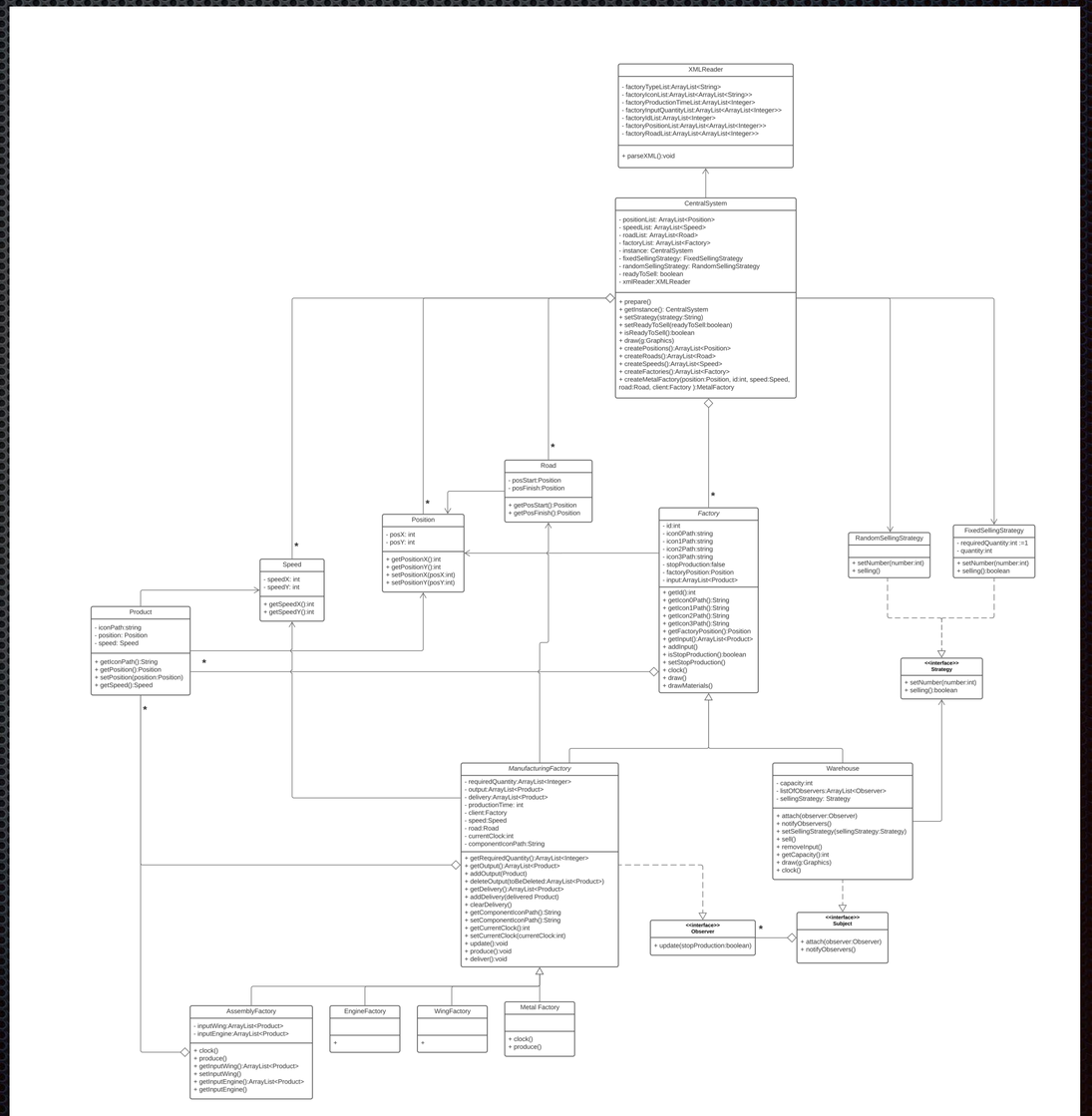
- Each plant has 4 states:
  - Empty 
  - 33% full 
  - 66% full 
  - 100% full 

PROJECT DESCRIPTION

- Notes:
  - In order to produce metal, the metal plant does not require input products but only the necessary production time.
  - The warehouse has no production. The aircraft produced by the production chain are stored there. Therefore, the warehouse has no output components or production time. However, the warehouse has a storage capacity limit that it cannot exceed.
  - When the warehouse is full, it should notify all the other plants to stop manufacturing immediately (*Observer pattern*).
  - The number of aircrafts in the warehouse decreases when a sale takes place. Different sales strategies can be used and the flow of the production chain must be managed accordingly. In our simulation, we will consider two strategies (*Strategy pattern*).

PROJECT ARCHITECTURE

- To better describe the project architecture, a PDF file containing the UML Class Diagram of the project will be provided



DESIGN REQUIREMENTS

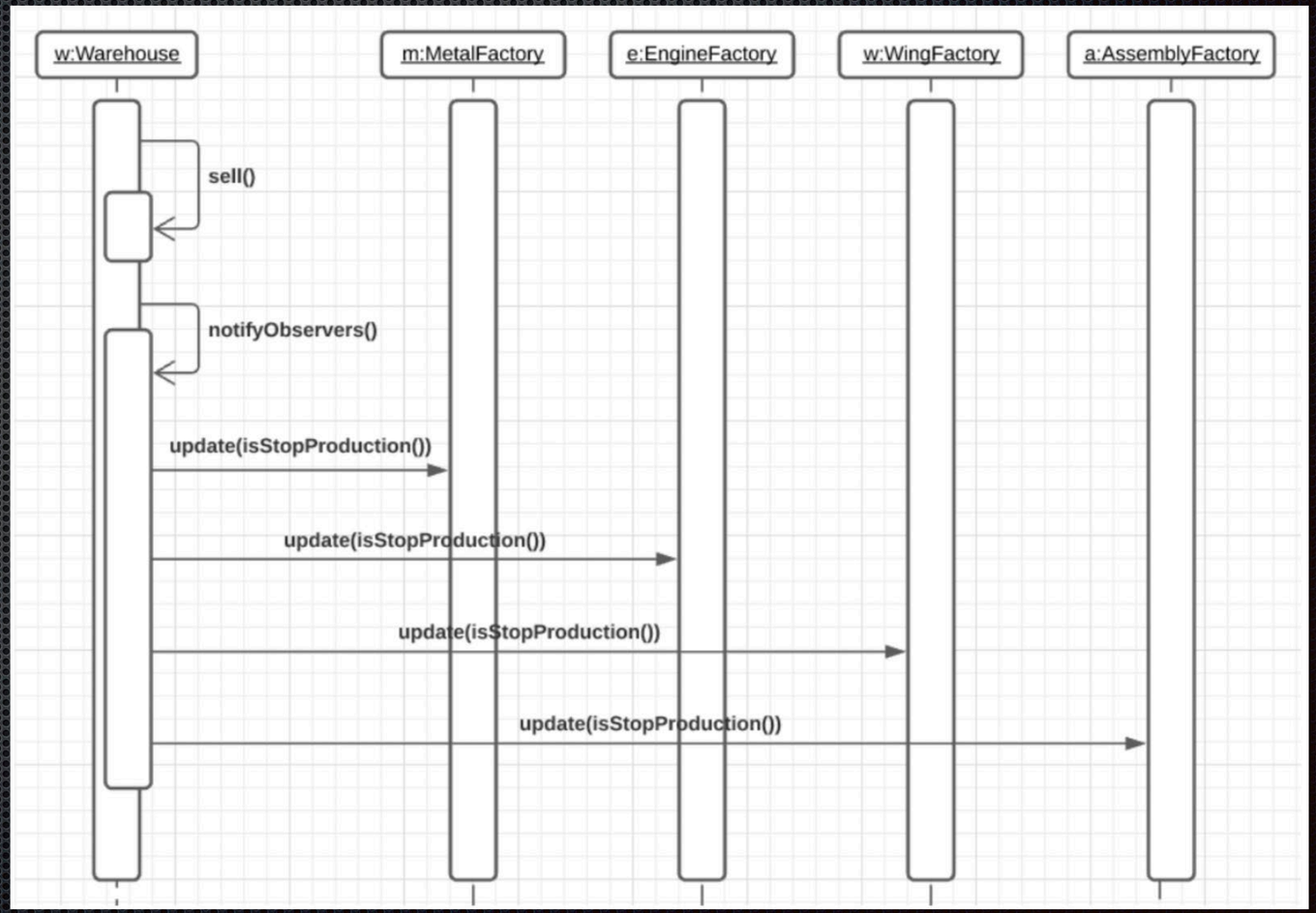
- In this project:
  - We designed and implemented several object-oriented concepts
    - Encapsulation, Inheritance, Polymorphisme…
  - We analyzed and extracted data from an XML file, which represents our data file
    - We used the concepts of Node, Tag Element…
  - We used various design patterns
    - Observer design pattern, Strategy design pattern, Singleton design pattern
  - We implemented different data structures and used Stream processing
    - Lists, ArrayLists, HashMaps, Lambda expressions…
  - Finally, we provide a clear, precise and easy to use Java Swing interface

DESIGN REQUIREMENTS - EXAMPLES

- *Inheritance and Polymorphism:*
  - In order to represent the 4 factories + the warehouse, we had 2 solutions:
    - Solution 1:
      - Design 2 abstract classes : Factory & ManufacturingFactory where ManufacturingFactory extends the Factory
      - Add 5 concrete classes : 4 types of factories + 1 warehouse
      - The warehouse extends the Factory class
      - The 4 factories extend the ManufacturingFactory class
    - Solution 2:
      - Design only 5 concrete classes for each factory
  - We chose the first solution for better cohesion and less dependency. Without this solution, all the methods and attributes in the classes Factory and ManufacturingFactory would be duplicated 5 times over each factory class. Furthermore, we would not be able to use polymorphism in method parameters or return types.

DESIGN REQUIREMENTS - EXAMPLES

- *Design Patterns:*
  - We used the observer design pattern in order to achieve the following:
    - Once the warehouse is full, all factories should stop production immediately
    - Once the warehouse has storage capacity, all factories should resume production immediately



DESIGN REQUIREMENTS - EXAMPLES

- *Design*
  (Code snapshots)

```java
package ObserverPattern;

/**
 * Subject Interface - Observer Design Pattern
 */
public interface Subject {
    /**
     * Attach an observer to the list of observers
     * @param observer factory observing the subject
     */
    1 usage   1 implementation
    void attach(Observer observer);

    /**
     * Notify the list of observers when the subject's state has changed
     */
    2 usages   1 implementation
    void notifyObservers();
    // detach could be added, but it is not used in this project
    //void detach(Observer observer);
}
```

```java
public class Warehouse extends Factory implements Subject {

    /**
     * Attach observer to the list of observers
     * @param observer factory observing the warehouse
     */
    1 usage
    @Override
    public void attach(Observer observer) {listOfObservers.add(observer);}

    /**
     * Notify observers to update their status
     */
    2 usages
    @Override
    public void notifyObservers() {
        for (int i=0; i<listOfObservers.size();i++){
            listOfObservers.get(i).update(isStopProduction());
        }
    }
```

```java
package ObserverPattern;

/**
 * Observer Interface - Observer Design Pattern
 */
public interface Observer {
    /**
     * Update observer state based on the boolean parameter stopProduction
     * @param stopProduction true when the warehouse is full, and false otherwise
     */
    1 usage   1 implementation
    void update(boolean stopProduction);
}
```

```java
public abstract class ManufacturingFactory extends Factory implements Observer {

    @Override
    public void update(boolean stopProduction){setStopProduction(stopProduction);}
```

DESIGN REQUIREMENTS - EXAMPLES

- *Data structures & Stream processing(Lambda expressions)*

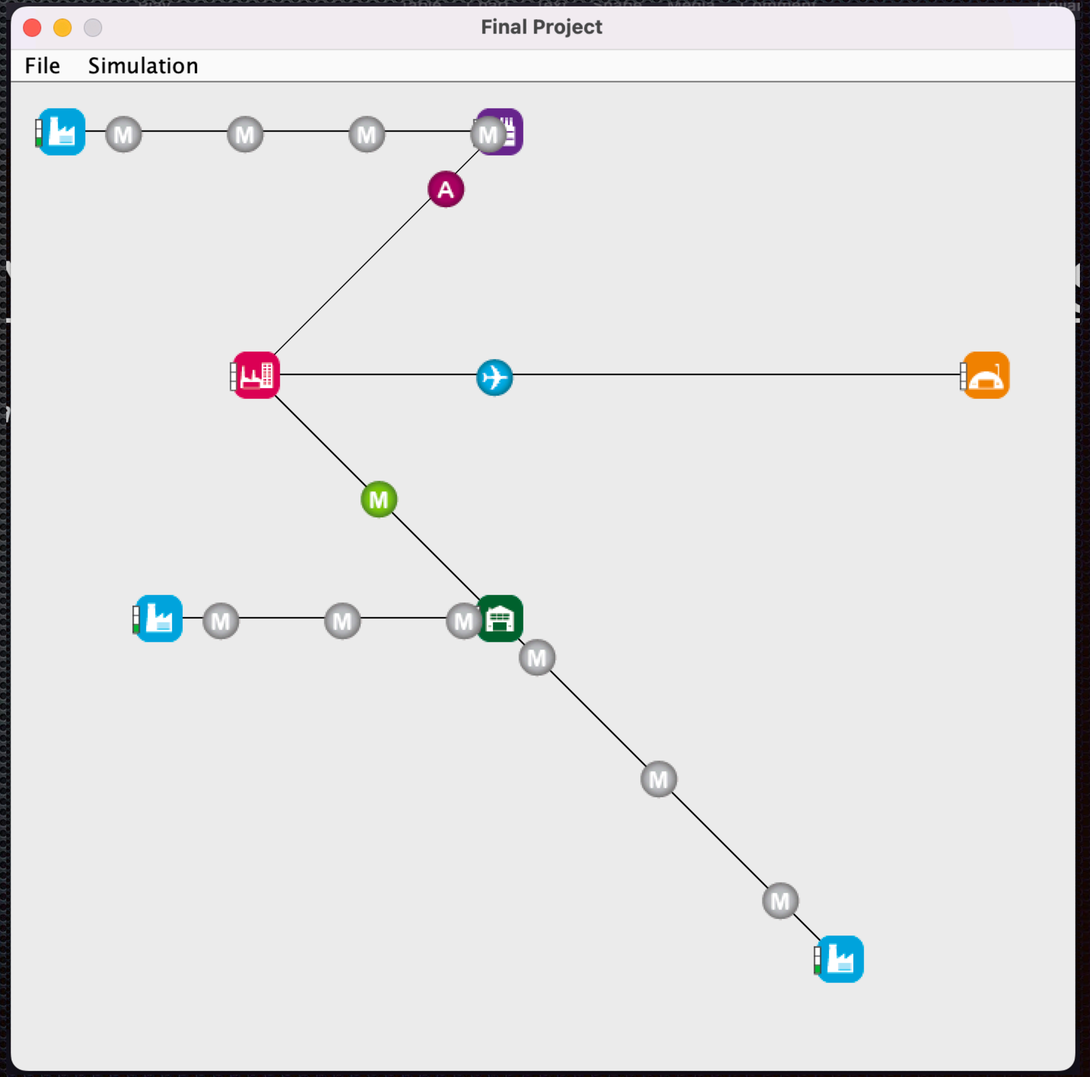
```
//Hashmap to get a Position from an ID, useful for creating roads
HashMap<Integer, Position> idToPosition = new HashMap<>();
```

```
/**
 * The respective required quantities of input components necessary to manufacture a product
 */
2 usages
private ArrayList<Integer> requiredQuantity;
```

```
//Lambda example
confirmButton.addActionListener((ActionEvent e) -> {
    String selectedButtonText =getSelectedButtonText(buttonGroup);
    this.strategy=selectedButtonText;
    ControlSystem.getInstance().setStrategy(this.strategy);
    SwingUtilities.getWindowAncestor((Component) e.getSource()).dispose();
});
```

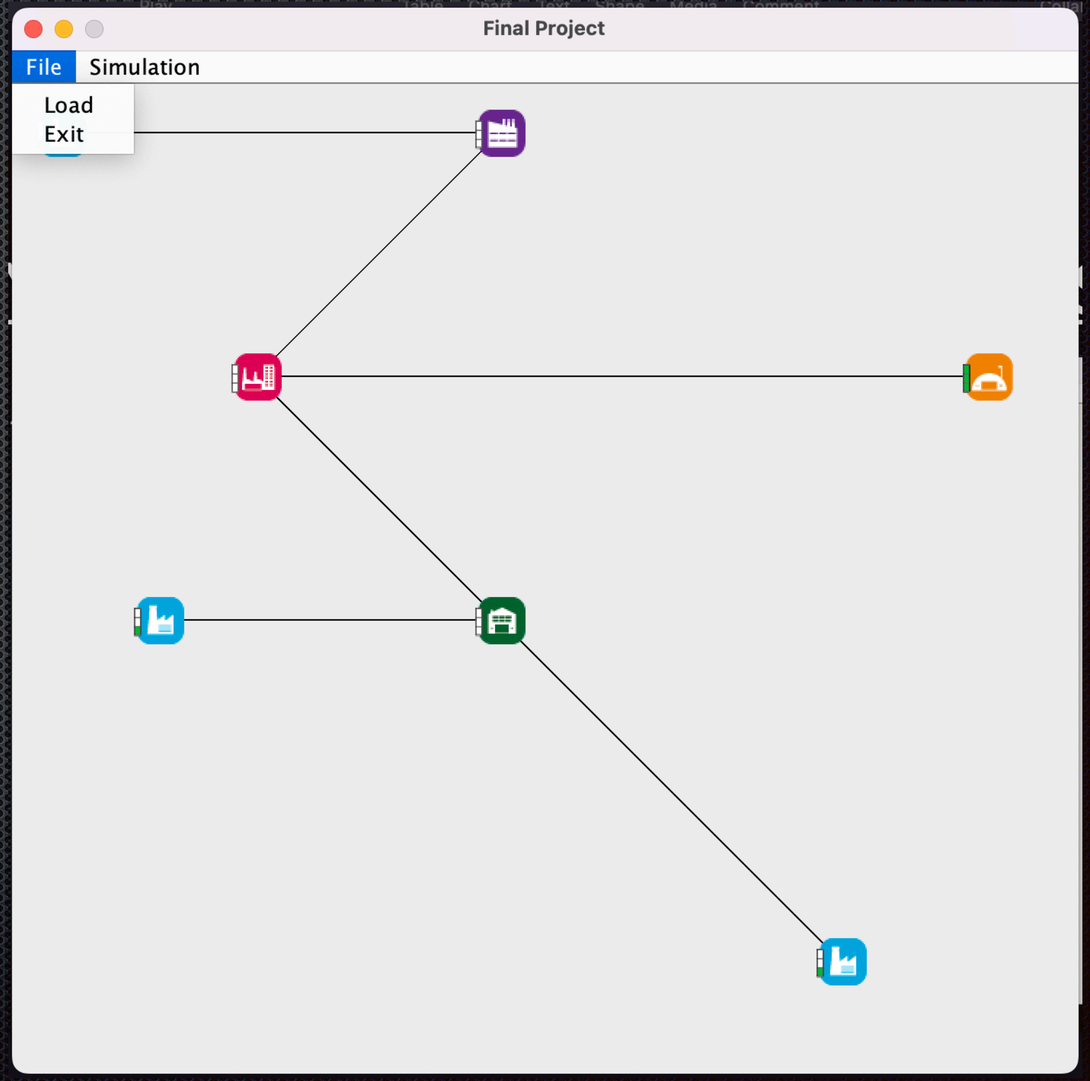
DESIGN REQUIREMENTS - EXAMPLES

- *An easy to use Java Swing interface*



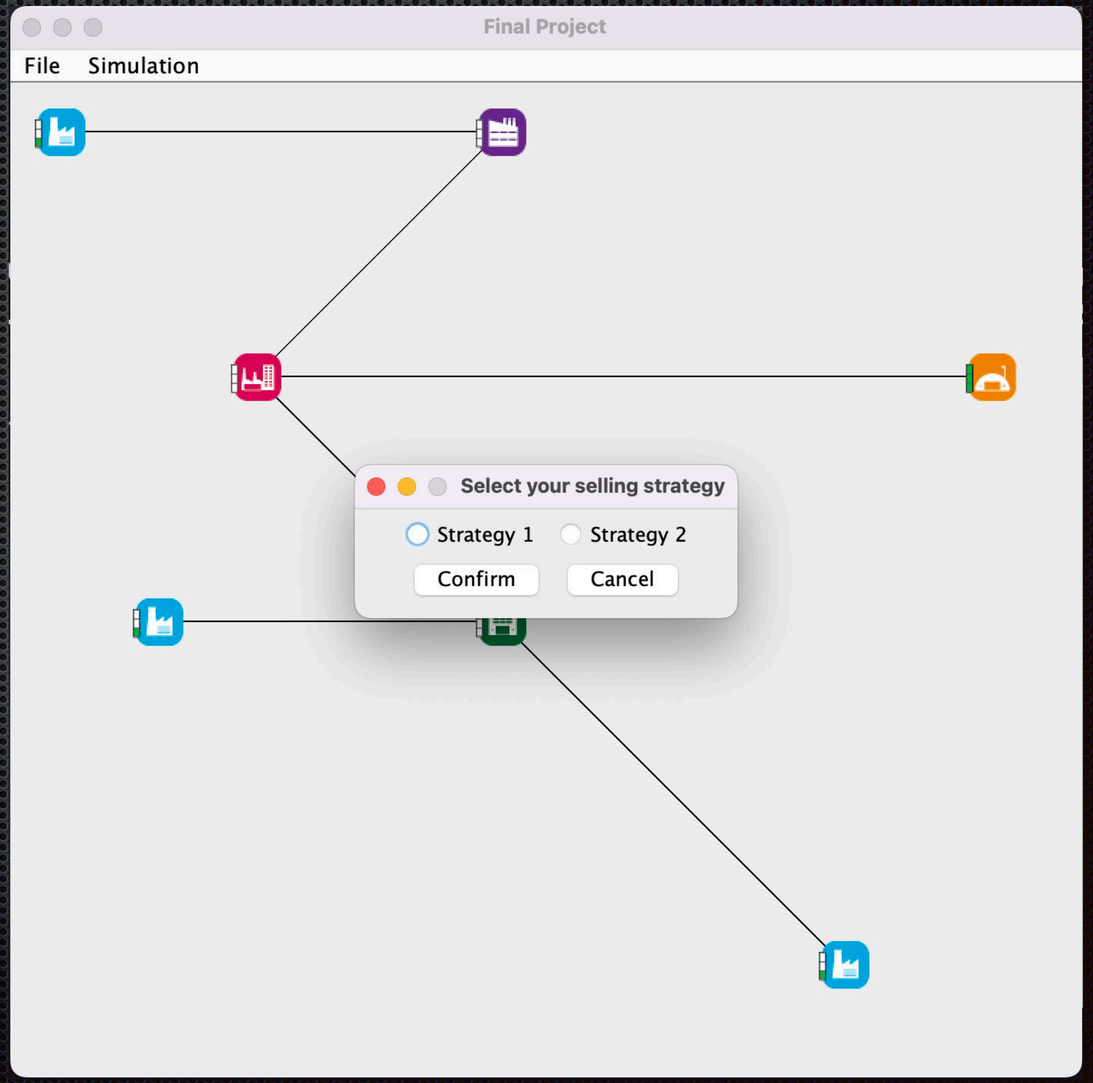
DESIGN REQUIREMENTS - EXAMPLES

- *By clicking on the File button, we can load the .xml file containing the necessary data for our simulation*



DESIGN REQUIREMENTS - EXAMPLES

- *By clicking on the Simulation button, we can choose between two selling strategies for the warehouse:*
  - *Strategy1: fixed-selling strategy Meaning that we sell an aircraft in a predefined frequency (e.g., every two seconds)*
  - *Strategy2: random-selling strategy Where the sale is performed randomly*



CONCLUSION

- In this project, we applied the different concepts of Object-Oriented Programming. We also better understood the importance of design patterns. And we used various data structures and stream processing.
- More importantly, we learned how to collaborate in a team and work on a large project, breaking down each problem into smaller tasks and setting dates for milestone events to complete the project within the predefined deadline.