
Royal Institution of Technology
School of Science, Engineering & Technology

PROJECT

WAREHOUSE MANAGEMENT SYSTEM

1. Cao Duy Minh s3818272

Abbreviation

WMS Warehouse Management System

OOP Object Oriented Programming

List of Figures

| | |
|---------------------------------------|----|
| <i>Figure 1 : UML class diagram</i> | 13 |
| <i>Figure 3 : 3-tier architecture</i> | 24 |

List of Tables

| | |
|--|----|
| <i>Table 1 : List of Objects</i> | 9 |
| <i>Table 2 : WareHouse Methods</i> | 10 |
| <i>Table 3 : Employee/Manager Methods</i> | 11 |
| <i>Table 4 : Manager Methods</i> | 11 |
| <i>Table 5 : Item Methods</i> | 11 |
| <i>Table 6 : Department Methods</i> | 12 |
| <i>Table 7 : WareHouse Object</i> | 14 |
| <i>Table 8 : Item Object</i> | 15 |
| <i>Table 9 : Employee Object</i> | 16 |
| <i>Table 10 : Manager Object</i> | 16 |
| <i>Table 11 : Delivery Object</i> | 17 |
| <i>Table 12 : Storage Object</i> | 18 |
| <i>Table 13 : Abstract Class</i> | 19 |
| <i>Table 14 : Package Design</i> | 25 |
| <i>Table 15 : Access Control Modifiers</i> | 27 |

Table of Contents

| | |
|---------------------------------|----|
| I. INTRODUCTION | 7 |
| II. CLASS ANALYSIS | 8 |
| 1. OBJECT | 8 |
| 2. CLASSES | 10 |
| a) Warehouse Class | 10 |
| b) Employee/Manager Class | 10 |
| c) Item Class | 11 |
| d) Department Class | 12 |
| III. CLASS DESIGN | 13 |
| 1. CLASSES | 14 |
| a) Warehouse Class | 14 |
| b) Item Class | 14 |
| c) Equipment/ Refreshment Class | 15 |
| d) Employee Class | 16 |
| e) Manager Class | 16 |
| f) Delivery Class | 17 |
| g) Storage Class | 17 |
| 1.1 Abstract Class: | 18 |
| 2. SOME OOP TECHNIQUES: | 19 |
| A) Overloading method: | 19 |

| | |
|---|----|
| B) Overriding methods: | 22 |
| C) Inheritance | 22 |
| IV. Package Design | 24 |
| V. Interface Design | 26 |
| VI. Access Control | 26 |
| VII. Encapsulation vs Inheritance vs Polymorphism | 27 |
| 1. Encapsulation | 27 |
| 2. Inheritance | 28 |
| 3. Polymorphism | 30 |

I. INTRODUCTION

Warehouse plays an important role in manufacturing and supply chain operations as it contains raw materials as well as finished goods. Because of its major role, warehouse management is extremely important to ensure all warehouse processes run as efficiently and accurately as possible. In this project, we study the business domains as well as business rules of the Warehouse Management System, then build a simple desktop application managing one warehouse.

This project provides some fundamental business functionalities for the stakeholders/users as follow:

- Update warehouse information (retrieve goods and remove goods).
- Organize incoming goods into shelves (locations of goods in the warehouse).
- Provide a report on goods information in the warehouse.
- Notify out-of-stock.
- Checking goods status.
- Manage employees.

To be able to deliver all the functionalities mentioned above, we use an object-oriented paradigm to analyze the real-world problems and implement them in Java object-oriented programming (OOP) concepts. In the real scenario, a warehouse contains various types of goods and has many employees working in it. All of the members in the warehouse and the warehouse itself can be viewed as “objects”, which means that these real-world objects are implemented as “objects” in the OOP concept. With objects having similar states and behaviors, we group them into one Java OOP class. The next session indicates how our team groups those objects as well as designing classes.

Because of many limitations, our project does not capture the whole business domain as well as business rules in the industry. For simplicity, we only investigate some main features of a WHS to demonstrate how basic management functions work. Applying the Java OOP concept, we build a simple, easy-to-understand system implementing the fundamental concepts of a WMS.

II. CLASS ANALYSIS

1. OBJECT

The Warehouse Management System (WMS) should reflect as many real-life actions of the warehouse as possible. Overall, there can be 3 types of activities a warehouse must handle:

- Logistic Activities: this is related to the receiving and delivering process of the warehouse. All of the arrived packages will have to be marked by a unique ID, and be systematically allocated to a place inside the warehouse. Finding and extracting a package is then easier due to indexes.
- Human Activities: Humans are an essential part of any place. Each employee at the warehouse must be managed efficiently.
- Item Management Activities: regular checking, validating activities are required to ensure that each item is in good condition, at the right places, and always ready to be delivered.

While planning for the functionalities of the warehouse, our team heavily considers the practical aspect of the product. It should be self-explanatory and transparent to whoever is analyzing it. Therefore, our WMS application consists of all the critical functionalities mentioned above. Here is the detailed description of each object:

| No | Object Name | State | Behaviours |
|----|--------------|---|--|
| 1 | Warehouse A | ID: 00001 Name: A Location: Binh Duong | getInfo(), updateInfo(), addEmployee(), removeEmployee(), addItem(), removeItem(),... |
| 2 | Employee B | Name: Nguyen Van B ID: 000001 Role: warehouse employee Salary: 10.000.000VND | calculateSalary(), update(), report(),... |
| 3 | Manager C | Name: Tran Thi C ID: 000002 Role: warehouse manager Salary: 15.000.000VND Bonus: 5.000.000VND | calculateSalary(), update(), report(),... |
| 4 | Department D | Name: department D Employees: employee B, manager C,... | getEmployee(), addPeople(), removePeople(),... |

| | | | |
|-----|--------|--|----------------------------|
| 5 | Item 1 | ID: 1 Name: Coca Cola Price: 150.000VND Location: A001 CheckUpDate: 22/12/2021 | checkUp(), remove(),... |
| ... | ... | ... | ... |

Table 1: List of Objects

The table above lists some warehouse objects along with their states and behaviours. Since there are possibly many physical objects in the warehouse, those with similar states and behaviours are grouped into one class to generalize them with the OOP concept.

2. CLASSES

This section is dedicated to showing detailed information about the classes of each object group mentioned above.

a) WareHouse Class

| Method | Behavior |
|------------------------------|---|
| getInfo | Get the Warehouse information. |
| updateWarehouseInfo | Update the states of object Warehouse. |
| searchItem | Search an item with a specific set of traits (parameters) from the database (all departments) |
| receipt | Import list of items. |
| issue | Move an item to a new department. |
| assignEmployeeToDepartment | Assign an existing employee to a department. |
| removeEmployeeFromDepartment | Remove an existing employee from a department. |
| updateItem | Update the current item information. |
| removeItem | Remove the current item from the warehouse. |

Table 2: WareHouse Method

b) Employee/Manager Class

Employee objects can be categorized into two classes: Employee and Manager which inherits the states and methods of Employee. In this situation, most of the methods in Manager's Class are the override versions from those of Employee.

| Method | Behavior |
|----------------|---|
| viewAssignment | Return the employee Assignment. |
| getInfo | Return the information of the employee. |
| validate | Validate the employee information |

Table 3: Employee/Manager Methods

| Method | Behavior |
|-------------------------------------|--|
| viewAssignment | Return all the employees with their Assignment. |
| @overload viewAssignment(int id) | Return the employee with their Assignment. |
| addEmployee | Add an employee into the Warehouse. This method will create a new instance of the employee object. |
| updateEmployee | Update an existing employee information |
| removeEmployee | Remove an employee from Warehouse by deleting the employee's instance. |
| assignEmployeeToDepartment | Add an employee into the given department. |
| removeEmployeeFromDepartment | Remove an employee from the department. |

Table 4: Manager Methods

c) Item Class

This class is divided into two smaller classes including Equipment and Refreshment. In this case, the Item Class is declared as an abstract class. Two of the subclasses inherit the states and methods of the class Item.

| Method | Behavior |
|-----------|------------------------------------|
| getInfo | Get information about this item. |
| validate | Validate current item information. |
| isExpired | Check if the item is expired. |

Table 5: Item Method

d) Department Class

This class illustrates the department that employees are working for. In this project, we consider two departments which are delivery and warehouse storage. Since they have the same behaviour as Departments but distinct ways of doing things, Delivery Class and Warehouse Storage will fully inherit all the methods of the Department Class. Thus, the Department Class should be an interface. We will discuss more details about this interface in the session Interface Design.

| Method | Behavior |
|--------------------------|--|
| getInfo | Get the department info |
| issue | Export an item from the department. The exported item will be moved to the Delivery Department. |
| @overload issue | Export all items from the department. The exported items will be moved to the Delivery Department. |
| getItem | Get an item with id. |
| @overload getItem | Get all items from the department. |
| getEmployee | Get an employee. |
| @overload getEmployee | Get all employees |
| addEmployee | Add a list of employees into the department. |
| removeEmployee | Remove an employee from the department. |
| addItem | Add items into the current department. |
| removeItem | Remove items from the current department. |

Table 6: Department Methods

III. CLASS DESIGN

This session will provide more detailed information about the class design of WMS. We use UML language as a tool to demonstrate the relationships among the classes of our system. Furthermore, with each class, a table with all the detailed variables and methods is shown.

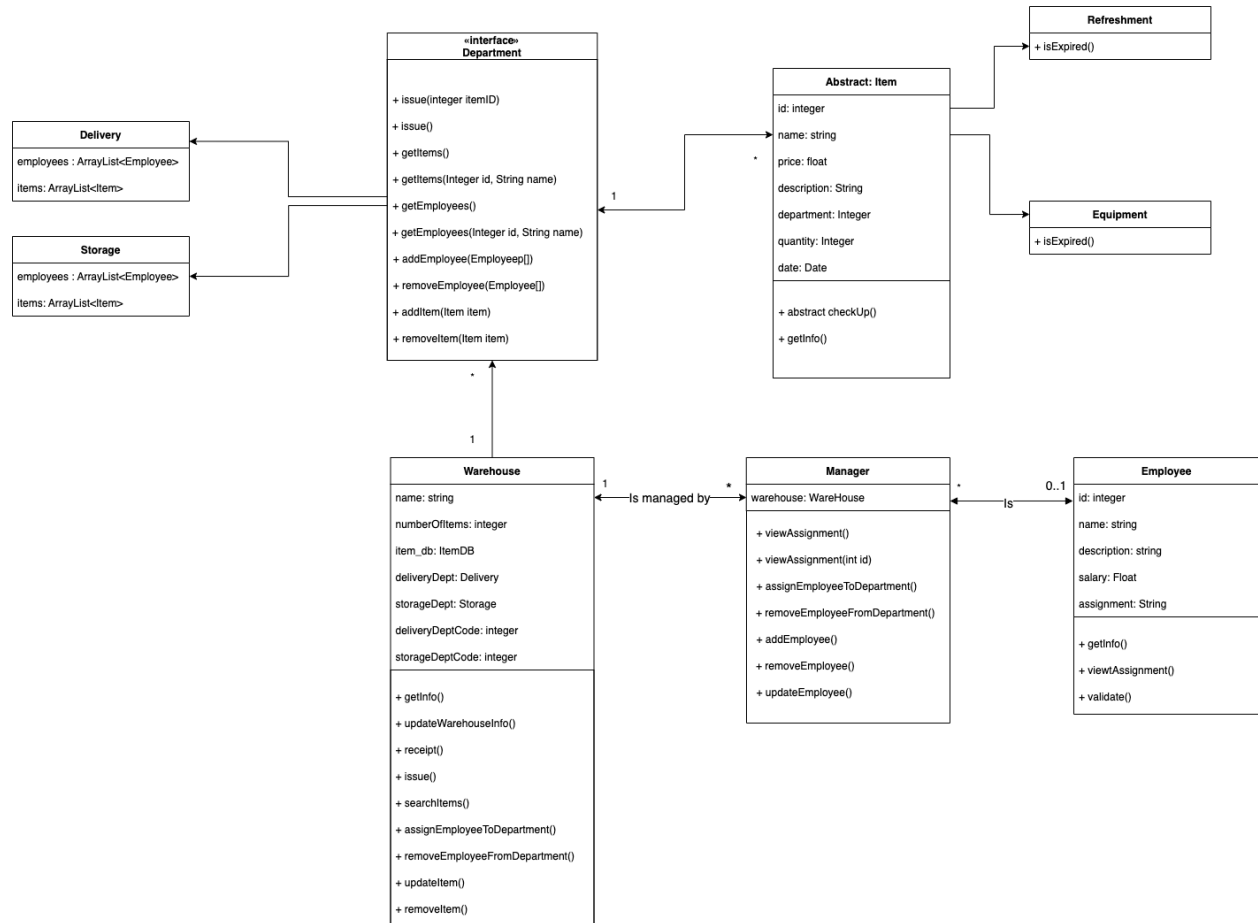


Figure 1: UML class diagram

1. CLASSES

a) Warehouse Class

This object represents the warehouse in practice. It comes with 5 states:

| Variable | Type | Purpose |
|------------------|-----------------|--|
| name | Private String | For differentiating the identity of each warehouse |
| numberOfItems | Private Integer | Number of items in the warehouse. |
| deliveryDept | Delivery | Delivery department of the warehouse. |
| storageDept | Storage | Storage department of the warehouse. |
| deliveryDeptCode | Private Integer | Code of the delivery department. |
| storageDeptCode | Private Integer | Code of the storage department. |

Table 7: Warehouse Object

Warehouse class will be “*public*”: we intended to use GUI to link to this class, therefore having control of the system. Thus, the “public” access modifier should be used to enable the action.

All Warehouse instance variables will be “*private*”: to prevent other classes change the values of variables.

All Warehouse methods will also be “*public*”: allow call of methods from an Warehouse instance anywhere.

b) Item Class

One instance object of this class consists of the basic states for an item stored inside the warehouse. Item is the generalization of three other types, which are Equipment and Refreshment. These objects inherit the states and methods of Items and also have their own ones.

Item:

| Variable | Type | Purpose |
|-------------|-------------------|--|
| id | Protected String | For differentiating the identity of each item. |
| type | Protected Integer | Type of the item. |
| name | Protected String | Name of the item. |
| description | Protected String | Description of the item. |
| price | Protected Integer | Price of the item. |
| department | Protected Integer | State the item department |
| quantity | Protected Integer | Quantity of the item |
| date | Protected Date | Date created of the item |

Table 8: Item Object

Item class will be “*public*”: Item class will be called multiple times to access the items’ information.

All instance variables of this class will be “*protected*”: this class is the superclass of Equipment and Refreshment. Thus, the “*protected*” access modifier enables the subclass to inherit the variables.

All class methods will be “*public*”: allow call of methods from an Item instance anywhere.

c) Equipment/ Refreshment Class

Equipment/ Refreshment class will be “*public*” and “*extends*” *Item Class*: This class is the subclass of Item, thus should be able to be called by other classes.

All class methods will be “*public*”: this class owns its methods different from its parent class.

d) Employee Class

One instance object of this class represents an employee working inside the warehouse. The object Manager will inherit the state and method of Employee:

| Variable | Type | Purpose |
|-------------|-------------------|--|
| id | Protected String | For differentiating the identity of each employee. |
| name | Protected String | Name of the employee. |
| description | Protected String | Role of the employee. |
| salary | Protected Integer | Salary of the employee. |
| assignment | Protected String | Assignment of the employee. |
| department | Protected Integer | The department of the employee. |

Table 9: Employee Object

Employee class will be “*public*”: Employee class will be called multiple times to access the employees’ information.

All Employee instance variables are “*protected*”: this class is the superclass of Manager. Thus, the “*protected*” access modifier enables the subclass to inherit the variables.

All Employee methods are “*public*”: allow call of methods from an Employee instance anywhere.

e) Manager Class

| Variable | Type | Purpose |
|-------------|--------------------|--|
| warehouse | Private WareHouse | Controlling warehouse via this instance. |
| employee_db | private EmployeeDB | To access the Database layer in order to manipulate employee data. |

Table 10: Manager Object

Manager class will be “*public*” and “*extends*” **Employee**: this class is the subclass of Employee, thus should be able to be called by other classes.

All variables will be “*private*”: to prevent the personal information from being read or changed by other classes.

All class methods will be “*public*”: this class owns its methods different from its parent class.

f) Delivery Class

One instance object of this class represents the Delivery department in a real life scenario. This object contains a number of items and employees. This department is responsible for transporting items to the destination.

| Variable | Type | Purpose |
|-------------|---------------------------------------|--|
| employees | private static ArrayList<Employee> | All of the employees working in the Delivery department. |
| items | private static ArrayList<Item> | All of the items that the Delivery department manages. |
| item_db | private static ItemDB | To access the Database layer in order to manipulate item data. |
| employee_db | private static EmployeeDB | To access the Database layer in order to manipulate employee data. |

Table 11: Delivery Object

Delivery class will be “*public*”: these classes will be called multiple times to access the department’s information.

All Delivery instance variables are “*private*”: to prevent the information from being changed by other classes.

All Delivery methods are “*public*”: as the Delivery class needs to be accessed by other classes, the “*public*” modifier enables other classes to call Delivery methods.

g) Storage Class

One instance object of this class represents the Storage department in a real life scenario. This object contains a number of items and employees. This department is responsible for storing items in the warehouse.

| Variable | Type | Purpose |
|-------------|--------------------------------|--|
| Employees | Private ArrayList<Employee> | All of the employees working in the Storage department. |
| Items | Private ArrayList<Item> | All of the items that the Storage department manages. |
| item_db | private static ItemDB | To access the Database layer in order to manipulate item data. |
| employee_db | private static EmployeeDB | To access the Database layer in order to manipulate employee data. |

Table 12: Storage Object

Storage class will be “*public*”: these classes will be called multiple times to access the department’s information.

All Storage instance variables are “*private*”: to prevent the information from being changed by other classes.

All Storage methods are “*public*”: as the Delivery class needs to be accessed by other classes, the “*public*” modifier enables other classes to call Delivery methods.

1.1. ABSTRACT CLASSES

| Abstract Class | Abstract methods | Concrete methods | Description |
|----------------|------------------|------------------|---|
| Item | isExpired() | getInfo() | This class is an abstract class that is used by subclasses Equipment and Refreshment. |

Table 13: Abstract Class

2. SOME OOP TECHNIQUES:

A) **Overloading method:**

Overloading is being used extensively in our application. Namely in the Manager Class and the Department Class. Here is the brief description of those use case:

Manager Class: 2 methods **viewAssignment**, but they have different parameters.

```
@Override
public ArrayList<Employee> viewAssignment() {
    ArrayList<Employee> employees = null;
    try {
        employees = employee_DB.getAll();
    } catch (Exception e) {
        e.printStackTrace();
    }
    return employees;
}
public ArrayList<Employee> viewAssignment(Integer employee_id) {
    Employee employee = null;

    try {
        employee = employee_DB.get(employee_id);
    } catch (Exception e) {
        e.printStackTrace();
    }

    ArrayList<Employee> result = new ArrayList<Employee>();
    result.add(employee);

    return result;
}
```

Department is an interface, which includes the following overloading methods.

```
public String issue(Integer itemID);  
public String issue();  
public ArrayList<Item> getItems();  
public ArrayList<Item> getItems(Integer id, String name);  
public ArrayList<Employee> getEmployees();  
public ArrayList<Employee> getEmployees(Integer id, String name);  
public void AddEmployees(Employee[] employees);  
public void RemoveEmployee(Employee[] employee);  
public Department getInfo();
```

Delivery and Storage implement the Department interface. Thus, they have implemented the body of the function. This is the sample of class Storage, which has the mission of storing items while waiting for a delivery session.

```
        public String issue(Integer itemID) {
            StringBuilder sb = new StringBuilder();
            sb.append("The following items are being exported to customer: \n");
            for (Item item : items) {
                if (item.getID() == itemID) {
                    try { item_db.remove(item.getID());
                        catch (Exception e) { e.printStackTrace();}
                    sb.append(item.getName() + ", ");
                    items.remove(item);
                    break;
                }
            }
            return sb.toString();
        }
        public String issue() {
            StringBuilder sb = new StringBuilder();
            for (Item item : items) {
                try { item_db.remove(item.getID());
                    catch (Exception e) {e.printStackTrace();}
                sb.append("The item is being exported to customer: \n");
                sb.append(item.getName() + ", ");
                items.remove(item);
                break;
            }
            return sb.toString();
        }
        public ArrayList<Item> getItems() {return items;}
        public ArrayList<Item> getItems(Integer id, String name) {
            ArrayList<Item> items = new ArrayList<Item>();
            for (Item item : this.items) {
                if (item.getID() == id || item.getName().equals(name)) {
                    items.add(item);}}return items;
        }
        public ArrayList<Employee> getEmployees() {return employees;}
        public ArrayList<Employee> getEmployees(Integer id, String name) {
            ArrayList<Employee> employees = new ArrayList<Employee>();
            for (Employee employee : this.employees) {
                if (employee.getID() == id || employee.getName().equals(name)) {
                    employees.add(employee);
                }
            }
            return employees;
        }
    }
```

B) Overriding methods:

Manager Class: viewAssignment overrides its of the parent class.

```
@Override
public ArrayList<Employee> viewAssignment() {
    ArrayList<Employee> employees = null;

    try {
        employees = employee_DB.getAll();
    } catch (Exception e) {
        e.printStackTrace();
    }

    return employees;
}
```

C) Inheritance:

Inheritance gives us the flexibility of taking the parent's class methods, while allowing us to change the behavior of the subclasses' methods. In our application, inheritance is being implemented in:

- a. Item class: all the concrete functions of Item class such as getters and setters, display information about the class, v.v are all shared between its subclasses, Equipment and Refreshment. Moreover, **protected** states are also included, so that the subclass can inherit all the states.

```
public abstract class Item {
    protected Integer id;
    protected String name;
    protected String description;
    protected double price;
    protected Integer type;
    protected Integer department;
    protected Integer quantity;
    protected Date date;
    public String getName() {return this.name;}
    public Integer getID() {return this.id;}
        public void setID(Integer id) {this.id = id;}
        public double getCost() {return this.cost;}
    public abstract boolean checkUp();
}
```

-
- b. Employee class: this class is also designed to be inherited by the subclass Manager. However, the Employee is not an abstract class, because in common sense, the employees have far lower functionality compared to managers. Thus, methods that belong to Manager Class should be implemented there. This code shows how we design the Employee Class with **protected** keywords for inheritance.

```
public class Employee {
    protected Integer id;
    protected String name;
    protected String description;
    protected Float salary;
    protected Integer department;
    protected String assignment;
    public Employee(int id, String name, String description, Float salary,
int    department, String assignment) {
        this.id = id;
        this.name = name;
        this.description = description;
        this.salary = salary;
        this.department = 0;
        this.assignment = assignment;
    }

    public Employee getInfo() {return this;}

    public Integer getID() {
        return this.id;
    }

    ...
}
```

IV. Package Design

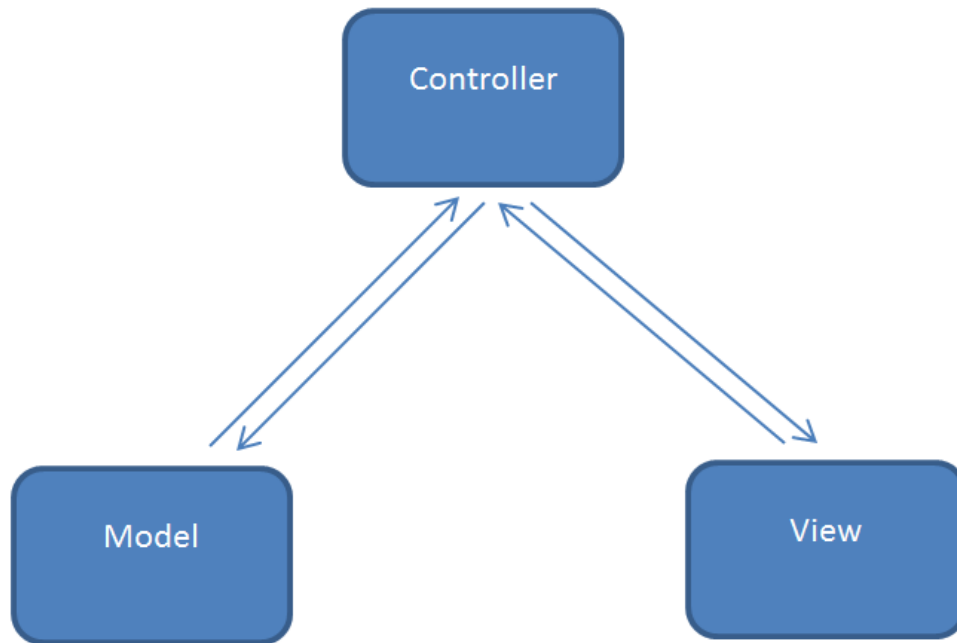


Figure 3: 3-tier architecture

Our application is build based on the 3 tier architecture:

1. Model: the layer where database activity is implemented. Here, we have classes and functionality to support database communication with all Classes on Controller.
2. Controller: the layer where the use cases of the application are implemented. All the classes such as Items, Employees, WareHouse are written here. Thus, this middle-layer has the responsibility of pushing information to the view layers, and saving data using model-layer communication.
3. View: the user interface of the application, main calling information from tier 2.

In each of the layers, we organize the packages by its usa case. This report is dedicated to only layer Controllers, therefore only the controller layout will be given.

Inside the controller layer, we have 4 packages:

| Package Name | Classes | Description |
|---------------------|-------------------------------|--|
| department | Delivery, Department, Storage | Department is the interface, and the two other classes implement it. Thus, it is relevant to put inside 1 package. |
| employees | Employee, Manager | Manager extends Employee class. |
| items | Item, Equipment, Refreshment | Item is the abstract class, while others extend it. |
| warehouse | WareHouse | |

Table 14: Package Design

V. Interface Design

An interface is a completely “abstract class” that is used to group related methods with empty bodies. All of its methods must be overridden by those classes that implement that interface. As mentioned before, Department is the interface for Delivery and Storage to implement it.

```
public interface Department {  
    public String issue(Integer itemID);  
    public String issue();  
    public ArrayList<Item> getItems();  
    public ArrayList<Item> getItems(Integer id, String name);  
    public ArrayList<Employee> getEmployees();  
    public ArrayList<Employee> getEmployees(Integer id, String name);  
    public void addEmployees(Employee[] employees);  
    public void removeEmployee(Employee[] employee);  
    public void addItem(Item item);  
    public void removeItem(Item item);  
}
```

VI. Access Control

The details of access controls of each class have been provided in the session Class Design, this session only gives a summary on the access control modifiers of all classes, all methods as well as all variables in this application.

| Class Name | Class Modifier | Variable Modifier | Method Modifier |
|---------------------------|-----------------------|--------------------------|------------------------|
| WareHouse | Public | Private | Public |
| Item | Public | Protected | Public |
| Equipment/ Refreshment | Public | Private | Public |
| Employee | Public | Protected | Public |
| Manager | Public | Private | Public |
| Department | Public (interface) | | Public |
| Delivery/ Storage | Public | Private | Public |

Table 15: Access Control Modifiers

VII. Encapsulation vs Inheritance vs Polymorphism

1. Encapsulation

Encapsulation means that we must wrap the data types and the functions which are related to an object to a single unit. In this way, our code is **clean** (as every function inside a class is related to the object) and **organized** (other classes cannot access the data in the class normally).

```
public class WareHouse {
    private static String name;
    private static Double cash;
    private static Integer numberOfItems;
    private static ItemDB item_db;
    private static Delivery deliveryDept;
    private static Storage storageDept;
    private static final Integer deliveryDeptCode = 100;
    private static final Integer storageDeptCode = 200;
    public WareHouse() {
        name = "Warehouse System";
        item_db = new ItemDB();
        deliveryDept = new Delivery();
        storageDept = new Storage();
        ArrayList<Item> items = null;
        try { items = item_db.getAllItems(); }
            catch (Exception e) { e.printStackTrace(); }
        numberOfItems = items.size();
    }
    public void receipt(List<Item> items, Integer departmentCode) {
        for (Iterator<Item> i = items.iterator(); i.hasNext();) {
            Item item = i.next();
            try { int item_id = item_db.add(item); }
                catch (Exception e) { e.printStackTrace(); }
            numberOfItems++;
        }
    }
}
```

For example, by encapsulating the datatype of Warehouse class above to be private, no other class can call the data type without accessing the class' getters and setters. Moreover, the function **receipt** implemented inside the class means that the function can only be called by an instance of Warehouse. Thus, called the methods of class.

2. Inheritance

Inheritance in Java is a mechanism in which one object acquires all the properties and behaviors of a parent object. In our application, we want the code to be as clean and simple as possible. That is why we heavily rely on it when building classes such as Employee.

```
public class Employee {
    protected Integer    id;
    protected String     name;
    protected String     description;
    protected Float      salary;
    protected Integer     department;
    protected String     assignment;
    public Employee() {}
    public Employee(int id, String name, String description, Float salary, int department, String
assignment)
    {
        this.id = id;
        this.name = name;
        this.description = description;
        this.salary = salary;
        this.department = department;
        this.assignment = assignment;
    }
    public Employee getInfo() { return this; }
    public Integer getID() { return this.id; }
    public String getName() { return this.name; }
    public String getDescription() { return this.description; }
    public int getDepartment() { return this.department; }
    public Float getSalary() { return this.salary; }
    public String getAssignment() { return this.assignment; }
    public void setDepartment(int code) { this.department = code; }
    public void setName(String name) { this.name = name; }
    public void setAssignment(String assignment) { this.assignment = assignment; }
    public void setSalary(float salary) { this.salary = salary; }
    public void setDescription(String description) { this.description = description; }
    public boolean validate()
    {
        if (this.name == null || this.name.isEmpty()) {
            return false;
        }
        if (this.description == null || this.description.isEmpty()) {
            return false;
        }
        if (this.salary == null) {
            return false;
        }
        if (this.assignment == null || this.assignment.isEmpty()) {
            return false;
        }
        return true;
    }
    public ArrayList<Employee> viewAssignment() {
        ArrayList<Employee> employees = new ArrayList<Employee>();
        employees.add(this);
        return employees;
    }
}
```

Our Employee Class, because of the Encapsulation feature, is filled with constructors, getters and setters. The effort spent to write those methods are gigantic, with just only a small number of variables. On the other hand, Manager in common sense is also an Employee, with the same number of variables and methods. In this way, we let the Manager extend the Employee, thus get all the variables and methods without the need to recreate.

3. Polymorphism

Polymorphism, in my opinion, helps the application to use Inheritance more flexibly. The reason is that, while inheritance helps subclasses to use the parent's class methods, it also creates the problem of flexibility: many classes can only use 1 method with the same behavior while in reality, the methods should perform the behavior with regard to the object called it.

Polymorphism supports Overriding, which is done so that a child class can give its own implementation to a method which is already provided by the parent class. For example, from the above Employee class, you can see the method **viewAssignment**, which returns Employees with their assignments. However, the Manager should be able to view all employees assignment. Therefore, we override the method **viewAssignment**:

```
public class Manager extends Employee {
    private WareHouse warehouse;
    public Manager(String name, Integer id, String role, Float salary) {
        super(name, id, role, salary);
        this.warehouse = new WareHouse();
    }
    @Override
    public ArrayList<Employee> viewAssignment() {
        ArrayList<Employee> employees = null;
        try {
            employees = employee_DB.getAll();
        } catch (Exception e) {
            e.printStackTrace();
        }
        return employees;
    }
    public ArrayList<Employee> viewAssignment(Integer employee_id) {
        Employee employee = null;
        try {
            employee = employee_DB.get(employee_id);
        } catch (Exception e) {
            e.printStackTrace();
        }
        ArrayList<Employee> result = new ArrayList<Employee>();
        result.add(employee);
        return result;
    }
}
```

Moreover, we also apply Method Overloading from polymorphism, that allows a class to have more than one method having the same name, if their argument lists are different. Another method **viewAssignment** but now with the parameter *employee_id* allows Manager to see the given employee's assignment.

VIII. Experiment

1. Environment and Tools

a. Environment:

- Operating System: Windows 10 Home 64-bit (10.0, Build 19042) (19041.vb_release.191206-1406)
- System Model: Surface Pro 6
- Processor: Intel(R) Core(TM) i5-8250U CPU @ 1.60GHz (8 CPUs), ~1.8GHz
- Memory: 8192MB RAM
- Available OS Memory: 8108MB RAM
- DirectX Version: DirectX 12
- DirectX Database Version: 1.0.8

b. Tools:

MySQL Connector/J is the official JDBC driver for MySQL

2. Functions

Our project simulates a virtual warehouse and allows users (in the role of warehouse manager) monitoring some warehouse activities. Those functions are specified in the table below.

| No | Function | Method | Class |
|----|---|---|---------|
| 1 | To view all the employees working in the warehouse. | public Employee[] viewAssignment() | Manager |
| 2 | To get employee | public Employee[] viewAssignment(Integer employee_id) | Manager |

| | | | |
|----|--|---|----------------------|
| 3 | To add an employee. | public Integer addEmployee(String name, String description, Float salary) | Manager |
| 4 | To remove an employee. | public void removeEmployee (Employee employee) | Manager |
| 5 | To add an employee to the department. | public void assignEmployeeToDepartment(Employee employee, Integer deptCode) | Manager |
| 6 | To remove an employee from the department. | public void removeEmployeeFromDept(Employee employee, Integer deptCode) | Manager |
| 7 | To add a list of items to the warehouse. | public void receipt(List<Item> items, Integer departmentCode) | WareHouse |
| 8 | To update an existing item | public String updateItem(Item item) | WareHouse |
| 9 | To get the list of items in each department. | public ArrayList<Item> getItems() | Storage/ Delivery |
| 10 | To get the list of employees working in each department. | public ArrayList<Employee> getEmployees() | Storage/ Delivery |
| 11 | To issue an item. | public double issue(Integer itemID) | Storage/ Delivery |
| 12 | To issue all items. | public double issue() | Storage/ Delivery |

3. Database

For the database layer, we have been inspired to use MySQL for its convenience and user-friendly interface (from MySQL Workbench). Here is the Entity Relationship that defined our database:

| items | |
|-------|--|
| PK | id: int |
| | name: varchar(255) description: varchar(255) price: float type: int department: int quantity: int |

| employee | |
|----------|---|
| PK | id: int |
| | name: varchar(255) description: varchar(255) salary: float department: int |

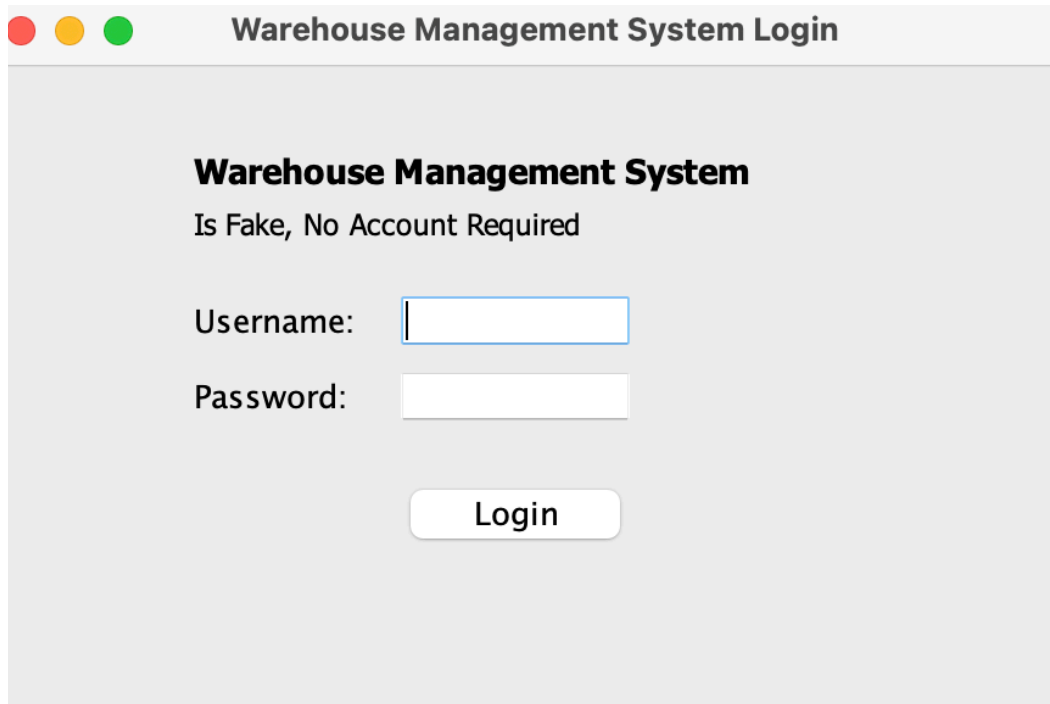
4. GUI

We use Java Swing to build the graphical user interface for this application. Also, we take advantage of the WindowBuilder which is a plugin of the IDE Eclipse that is dedicated to designing GUI. This plugin provides a better visual implementation without spending a lot of time writing code. To achieve the goal of providing all the functionality that is mentioned in the previous session, many Swing classes are created to support those functionalities with user-friendly interfaces. The table below provides the information on the names and orders with short descriptions of those classes.

| N | Name | Description |
|---|--------------|--|
| 1 | Login | This is the first frame that appears when the user runs the application. The login frame requires the user to enter username and password. If both of the username and the password is correct, the user can login to the |
| 2 | Welcome | After login successfully, a welcome dialog will pop up to welcome the user. |
| 3 | Warehouse | This is the main frame of this application. It shows the warehouse information and allows the manager to view all the employees working in the warehouse as well as add new employee(s) and new item(s). Also, the manager can choose to view departments. |
| 4 | EmployeeInfo | This shows the selected employee information by double-clicking on the employee in the Warehouse frame. Manager can choose whether to assign this employee to a department, or remove him/her from the |
| 5 | Assignment | This frame shows two options which represent two departments of the warehouse. Manager can choose one of the two departments which are Storage and Delivery to assign the employee. |

| | | |
|----|---------------|---|
| 6 | AddEmployee | Manager can add a new employee to the warehouse through this frame by filling in the name, description |
| 7 | AddItem | Manager can add new item(s) to each department of the warehouse through this frame by filling in the name, description, price, type, quantity and |
| 8 | DepartmenInfo | This shows all the items and employees of each department. Manager can also search items in this |
| 9 | ItemInfo | This shows the detailed information of a selected item by double-clicking on the item in the DepartmentInfo |
| 10 | SearchItem | This shows the search results. |
| 11 | UpdateItem | This allows the user to update information of an item. |
| 12 | UpdateEmploye | This allows the user to update information of an |

4.1. Login

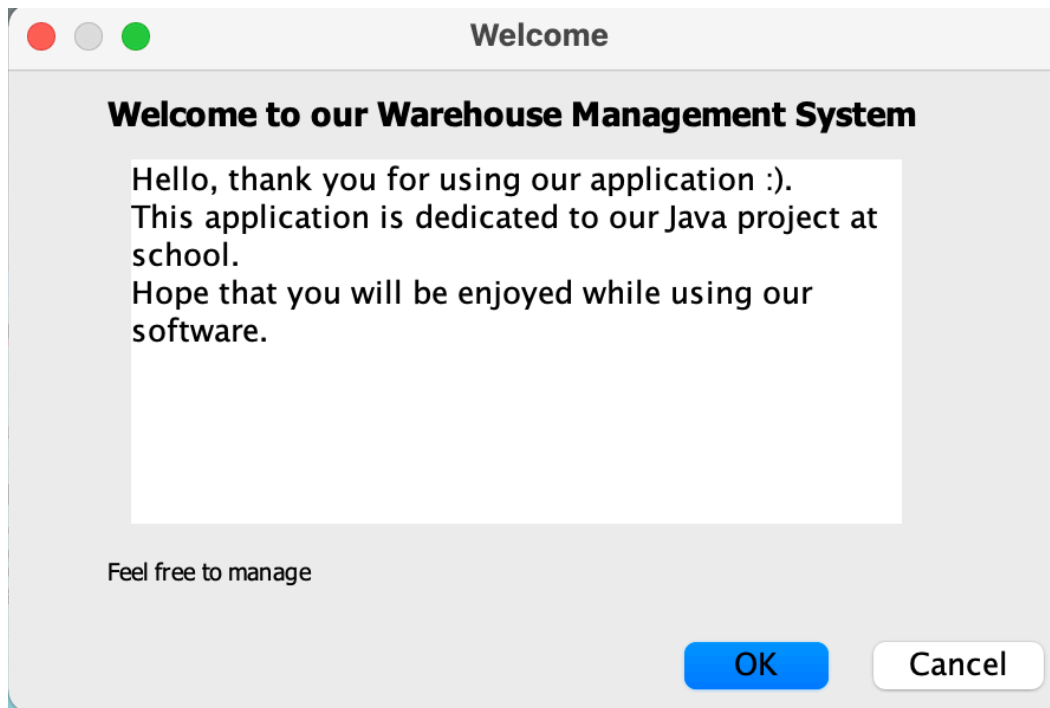


The screenshot shows a Java Swing window titled "Warehouse Management System Login". The window has a light gray background and a title bar with three colored buttons (red, yellow, green) on the left. Inside the window, the text "Warehouse Management System" is displayed in bold, followed by "Is Fake, No Account Required". Below this, there are two input fields: "Username:" followed by a text field, and "Password:" followed by a password field. At the bottom center, there is a "Login" button.

This is the login frame of the application. User has to enter the correct username and password to use this application. This JFrame has some GUI components including JLabel to display some text, JTextField to get the user's username input, JPasswordField to get the user's password input and JButton to validate the user's input username and password.

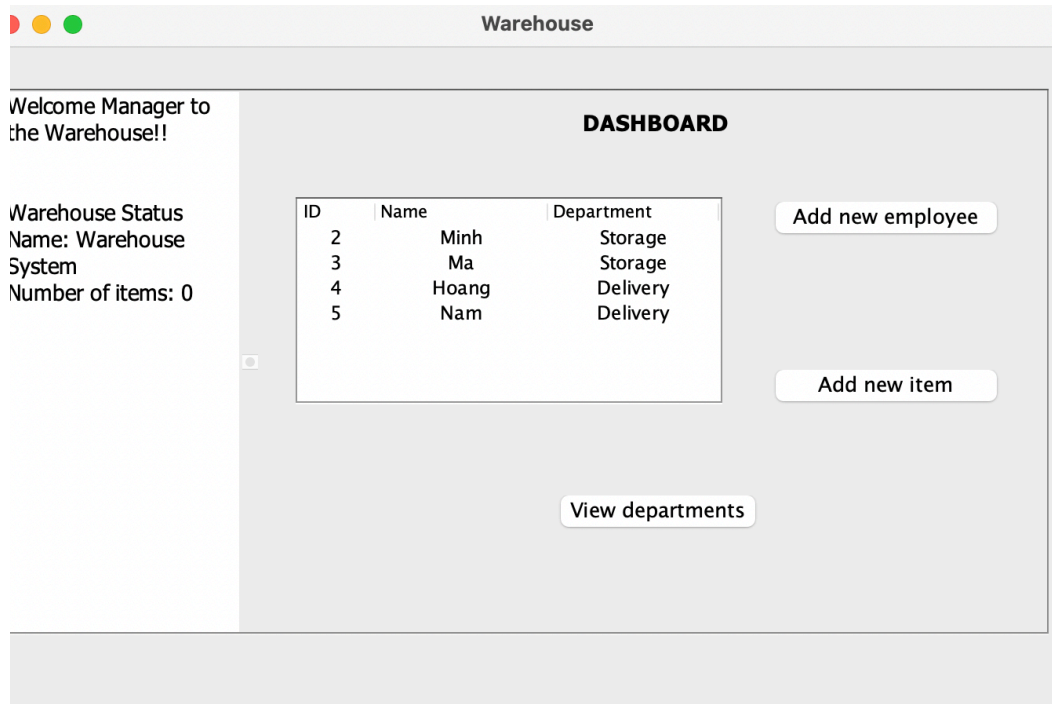
When the button is clicked, the system will get text from the text field and the password field and validate them by invoking the method `validateManager()` of the class `Manager`. If the username and password are correct, the user will get the Welcome dialog (see 4.2). Otherwise, "Wrong username or password" will be displayed.

4.2. Welcome



This is the welcome dialog displayed when the user successfully login. It displays some text with `TextArea` to show the greeting and some information about the application and two buttons “OK” and “Cancel”. When the user clicks the button “OK”, it will go to the main frame, the Warehouse (see 4.3). Otherwise, when the “Cancel” button is clicked, the application will close.

4.3. Warehouse

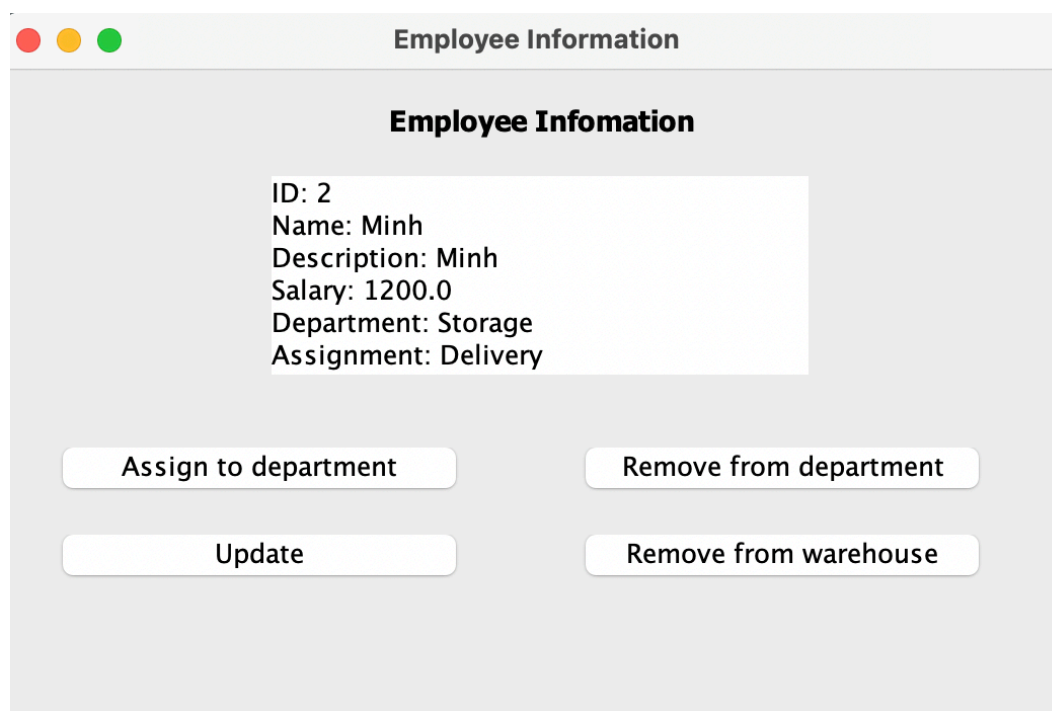


As mentioned above, this is the main frame of the application and is where the user (manager) can manipulate the warehouse by adding some new employees and items. In addition, it shows the status of the warehouse. This JFrame contains JSplitPane with the left component showing the warehouse status and the right component showing the JTable with the data of all of the employees working in that warehouse along with some buttons for the user to perform some warehouse management tasks.

The left component is a text area to show the warehouse status by invoking some getter methods in the class of Warehouse. While the right component contains a JTable inside a JScrollPane showing the employees' information and three buttons: "Add employee", "Add item" and "View departments".

The JTable gets all the employees' data through the viewAssignment() method of the class of Manager. When the user double clicks on one row of the table, the EmployeeInfo (see 4.4) of the selected employee pops up. When the user clicks on the “Add employee” or the “Add item” button, the AddEmployee (see 4.6) or AddItem (see 4.7) frame pops up. When the “View departments” is clicked, it will go to the DepartmentInfo frame (see 4.8).

4.4. EmployeeInfo

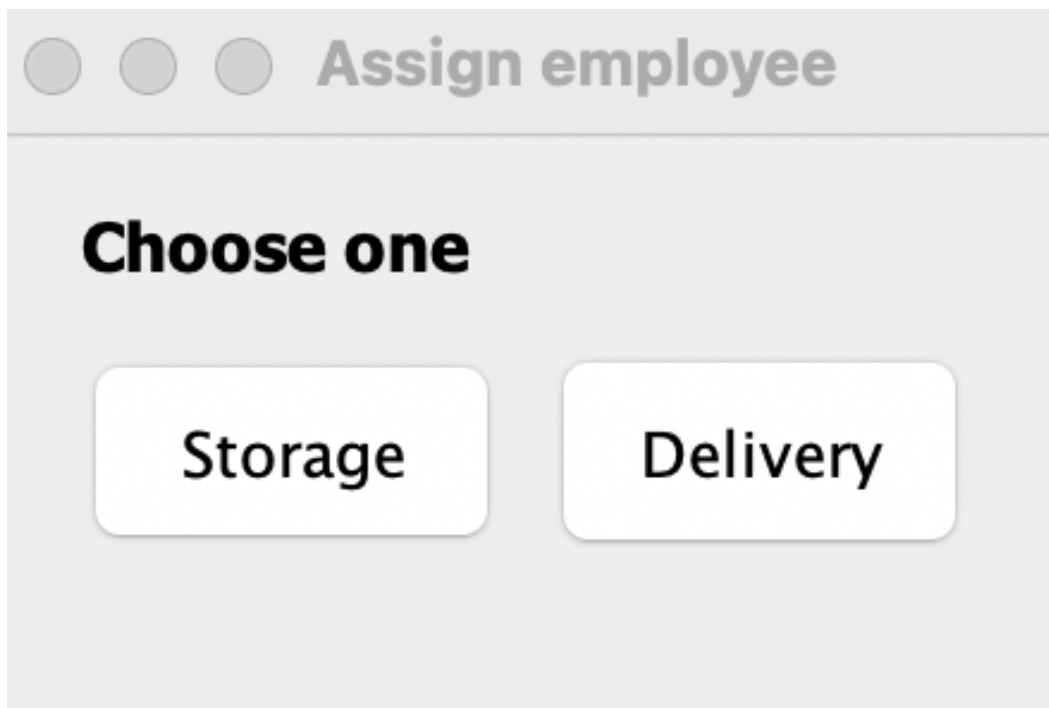


This frame shows the detailed information of an employee and allows the user to manage this employee. The user can assign this employee to a department or remove this employee from the department or the warehouse or update the employee. This JFrame

contains a JTextArea to show the employee's information and four buttons for the user to manipulate this employee.

The text area will get the information of the employee by invoking some getter methods in the class of Employee. When the user clicks the "Assign" button, an Assignment frame (see 4.5) pops up. Similarly, when the user clicks the "Update" button, the UpdateEmployee frame pops up. When the user clicks the "Remove" button, this employee is removed from the department that he/she was assigned to by invoking the removeEmployeeFromDept(Employee employee, Integer deptCode) method of the class of Manager. In the same manner, when the user clicks the "Remove from warehouse" button, the employee is removed from the warehouse by invoking the removeEmployee(Employee employee) method of the class of Manager.

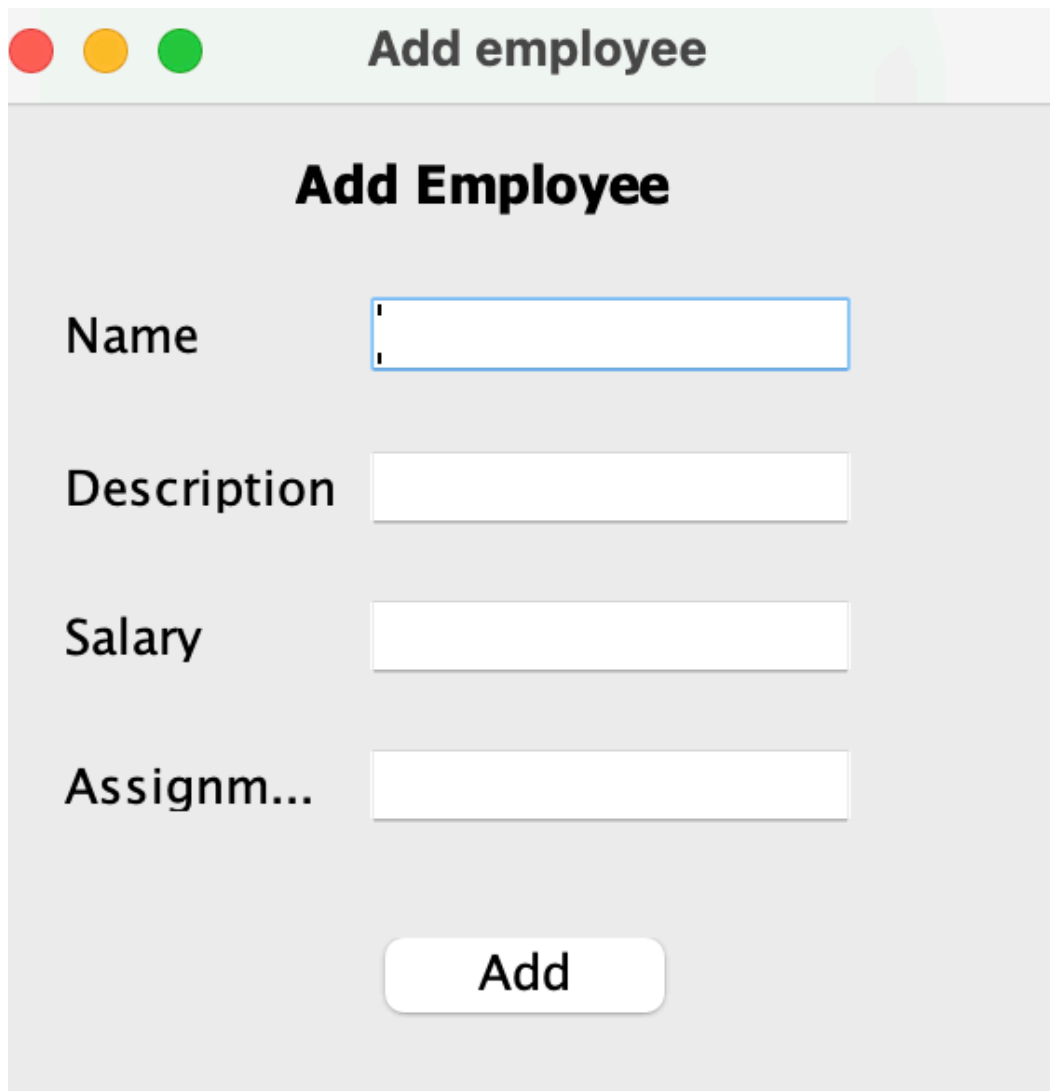
4.5. Assignment



Caption

This frame shows two options that the manager can choose to assign the employee. Two JButton are the left button “Storage” and the right button “Delivery”. After the user chooses which department the employee is assigned to, the assignEmployeeToDepartment(Employee employee, Integer deptCode) method of the class Manager is invoked to add this employee to the chosen department.

4.6. AddEmployee



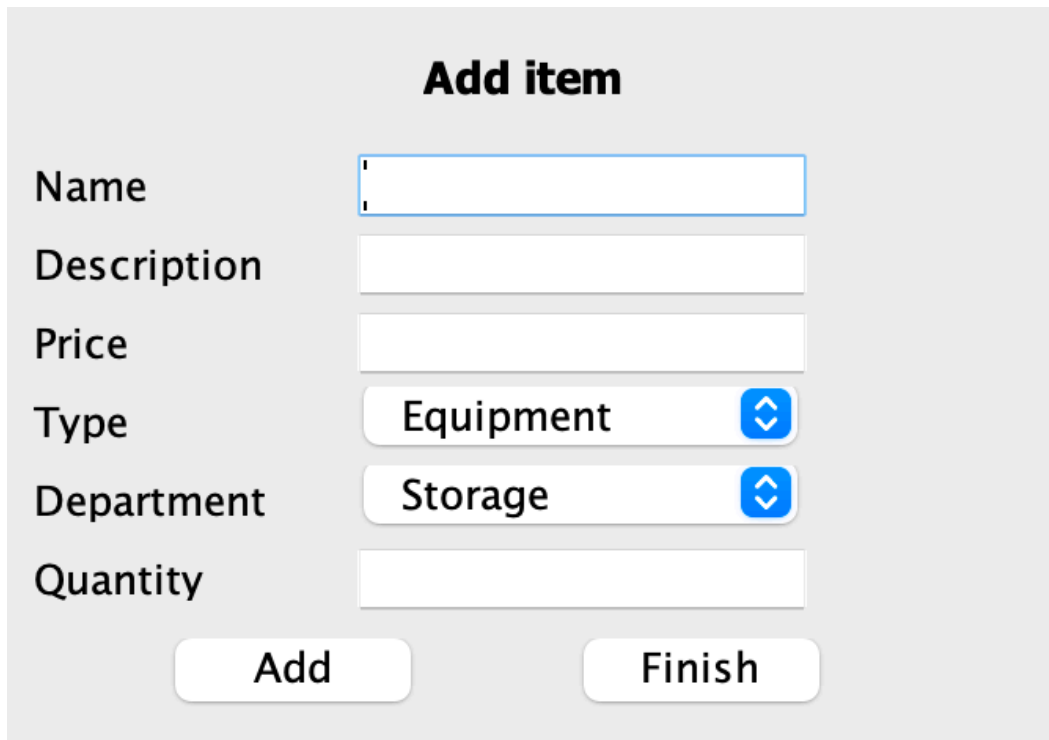
The image shows a Java Swing dialog box titled "Add employee". The dialog has a standard macOS-style title bar with red, yellow, and green window control buttons. The main content area has a light gray background and is titled "Add Employee" in bold black text. It contains four text input fields with labels to their left: "Name", "Description", "Salary", and "Assignm...". The "Name" field is currently selected with a blue border. At the bottom center of the dialog is a white button with rounded corners and a shadow, labeled "Add".

Caption



This frame allows the user to add a new employee by inputting necessary information about the new employee. This frame contains many text fields to get the user's input on the new employee data and a button "Add" to commit the act of adding the new employee.

When filling in all the needed information, the user clicks on the "Add" button, the system will get text from all the text fields and then convert it to corresponding data type in the database and then add this new employee to the warehouse through the `addEmployee(String name, String description, Float salary, String assignment)` method of the class of Manager.

4.7. AddItem



The image shows a user interface for adding a new item. It has a title "Add item" at the top. Below the title are six input fields: "Name", "Description", "Price", "Type", "Department", and "Quantity". The "Type" field is a dropdown menu with "Equipment" selected, and the "Department" field is a dropdown menu with "Storage" selected. At the bottom of the form are two buttons: "Add" and "Finish".

| Add item | |
|--|---|
| Name | <input type="text"/> |
| Description | <input type="text"/> |
| Price | <input type="text"/> |
| Type | Equipment  |
| Department | Storage  |
| Quantity | <input type="text"/> |
| <div><div>Add</div><div>Finish</div></div> | |

Caption

Similar to the AddEmployee frame, this frame allows the manager to add more items to the warehouse department. Likewise, this frame also contains many text fields letting the user enter necessary information about the new items. For the type and department, it contains JComboBox letting the user choose among the concrete values. The user may need to add many items to one department, this frame also supports that by setting two lists storing the data, one for the Storage department and one for the Delivery department.

After filling in the needed information and choosing the items in the combo boxes, the user clicks the button “Add”, this action will create a new item with the given data and then store to a predefined list and clear all the text fields. When the user clicks the button “Finish”, all of the items in the list of one department will be added to the warehouse through the method `receipt(List<Item> items)` of the class `WareHouse`.

4.8. DepartmentInfo

The screenshot shows a Java Swing window titled "Departments Information". It has two tabs: "Storage" (selected) and "Delivery".

Under the "Storage" tab, there are two panels:

- Items:** A table with 3 columns: ID, Name, and Quantity. It contains 8 rows of data.
- Employees:** A table with 2 columns: ID and Name. It contains 1 row of data.

At the bottom of the "Storage" tab, there is a button labeled "Move to Delivery".

| ID | Name | Quantity |
|----|----------|----------|
| 4 | Tra Sa | 12 |
| 5 | Tra Cam | 24 |
| 6 | Tra Mach | 24 |
| 7 | Cam | 12 |
| 8 | Quyt | 12 |
| 9 | Xoai | 12 |
| 11 | Bun Cha | 12 |
| 13 | Keo Tre | 12 |

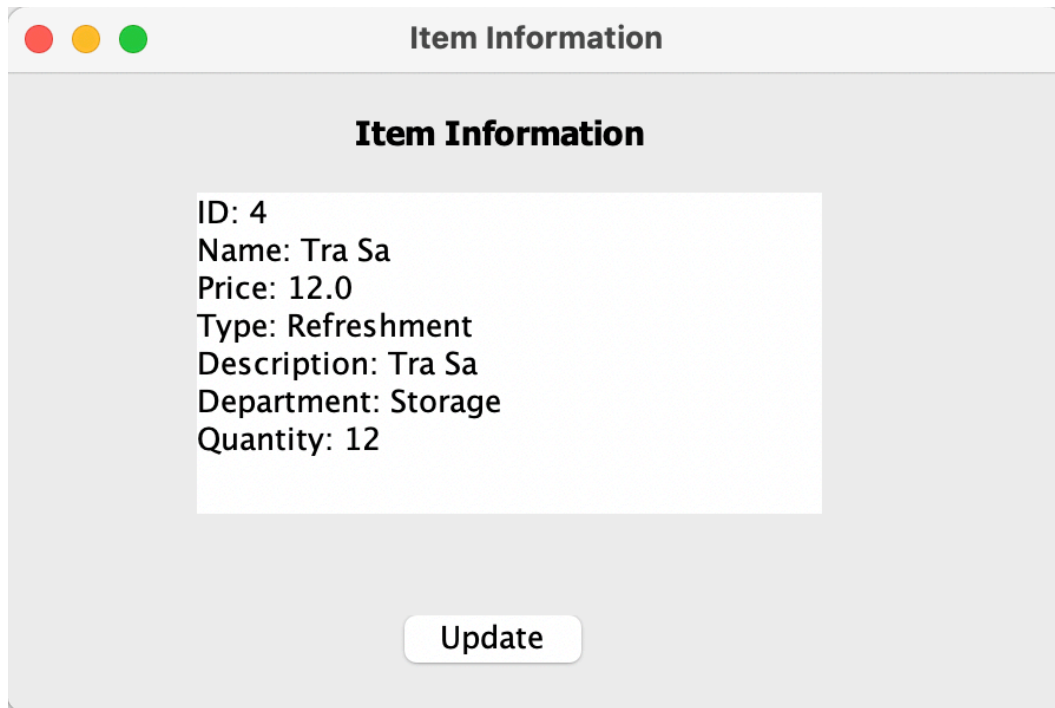
| ID | Name |
|----|------|
| 1 | Minh |

This frame shows all the employees as well as all the items in each department. This frame contains JTabbedPane to separate the two departments. Each panel in the JTabbedPane shows the employees and items of each department with JTable inside JScrollPane and a button (“Move to Delivery” in case of Storage tab, “Export” in case of Delivery tab). In addition, the user can search for items by entering the ID and name of the items in the JTextField that is next to the JLabel “Search”.

The user can see the table of items and employees of one department by clicking on the corresponding tab with the name of the department. The table shows the information by getting data from the class of its department. Instantly, the table shows all the items in the Storage by getting data through the method `getItems()` of the class Storage. Similarly, the employee table gets information through the method `getEmployee()`.

The item table also supports the act of managing items. By double clicking on the row in the item table, a `ItemInfo` (see 4.9) pops up. By holding the Ctrl and clicking on many rows in the item table, the user can select many items to move to Delivery (in the case of Storage tab) or export (in the case of Delivery tab) and press the button to perform the action. When the user clicks on the button, all of the selected items will be stored in a list and for each item in the list, the method `issue(Integer itemID)` of the class of corresponding department is invoked. The user can search for items by entering the ID or name in the text fields and then clicking on the small button beside it, a `SearchItem` (see 4.10) frame pops up.

4.9. ItemInfo

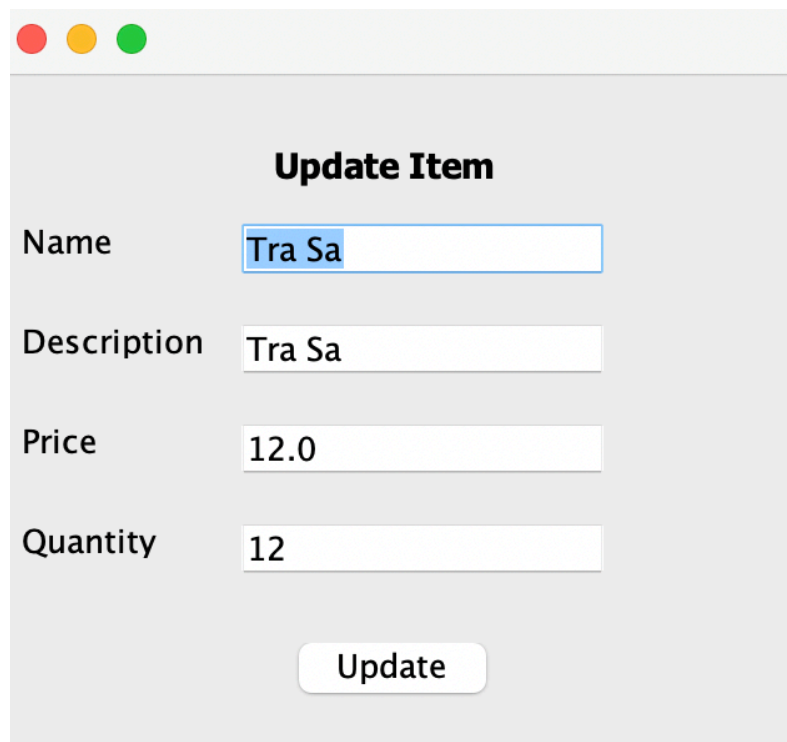


Caption

Similar to the EmployeeInfo, this frame shows the detailed information on a specific item. This frame contains a JTextArea to show the data getting through some getter methods in the class of Item.

When the user clicks on the button “Update”, a UpdateItem frame pops up.

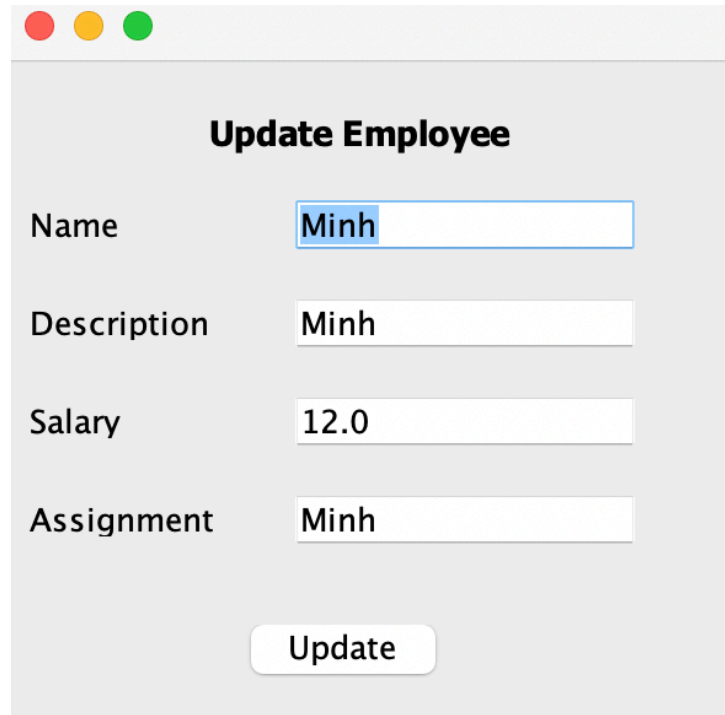
4.11. UpdateItem



The image shows a Java Swing window titled "Update Item". The window has a standard macOS-style title bar with red, yellow, and green buttons. Inside the window, there are four labels on the left: "Name", "Description", "Price", and "Quantity". Each label is followed by a text input field. The "Name" field contains the text "Tra Sa" and has a blue selection highlight. The "Description" field also contains "Tra Sa". The "Price" field contains "12.0". The "Quantity" field contains "12". At the bottom center of the window is a button labeled "Update".

This frame allows the user to update some information of an item. All of the `TextField` is filled with that item's information. If the user want to change something, he/she can edit the `TextField` and press the "Update" to update the new information. This action invokes the function `updateItem` of the class `WareHouse`.

4.12. UpdateEmployee



A screenshot of a Java Swing window titled "Update Employee". The window has a light gray background and a title bar with three colored buttons (red, yellow, green). It contains four text input fields with labels to their left: "Name" (containing "Minh"), "Description" (containing "Minh"), "Salary" (containing "12.0"), and "Assignment" (containing "Minh"). At the bottom center is a rounded rectangular button labeled "Update".

Similar to the `UpdateItem`, this frame allows the user to modify an employee's information. If the user wants to change something, he/she can edit the text in the `TextField` and press the "Update" button. This will invoke the function `updateEmployee` of the class `Manager`.

IX. Conclusion

The project for our team is an exciting journey to learn more about Java and its Object-Oriented programming. During the time working on Warehouse System, one thing that really inspired us is the unlimited directions that this application can grow. Thus, it carries tremendous **advantages**:

1. Component Comprehensiveness: our approach to building this application is to mimic as close as possible to a warehouse operation. The results are those above classes and functions: Department (to mimic different types of warehouse storage), Employee, Manager (to state different roles in warehouse), Item (what types of items would come to warehouse).
2. Scalability: when accessing our code, you can see that we designed the class so that they can be scale horizontally. In detail, adding more departments, item types in the future are effortlessly convenient. This is partly due to the OOP design in general, especially from the help of **Interface** and **Abstract Class**.
3. Maintainability: by using the MVC architecture, our group sees the prominent advantages: clarity and readability. Our code when following the architectural design is better organized, we know exactly where functions or classes should be placed, based on its responsibilities. In the future, even if the code is growing more, we believe that they are still manageable. Furthermore, we even drew the details of the UML class diagram and Entity Relationship, and online documentation to help better maintenance.
4. User-friendly Design: Our team put effort into the GUI design so that everybody can understand and use it easily.

However, despite the advantages above, we still encountered **disadvantages**:

1. Lacking Usability: although we have enough components to depict a warehouse operation, we are missing functionalities to help this application be used in reality. For example, we do not add the “incoming order” or “export order” because of

time and competency limitations.

2. Lack of best practice: a lot of confusion occurred as we did not know the best practice for implementing functionalities, and it took us time and effort to find out. However, it is still not clear to us if we have done it right. For instance, Database Connectivity should be implemented using Singleton Principle (S in SOLID), or it is better to use Abstract Class rather than its derived classes when dealing with encapsulation.
3. GUI Design: the guide to build a user-friendly interface is very difficult to follow. Moreover, Swing is quite-outdated in comparison to modern technology. Thus, making it hard for us to find support on forums.

We do not feel that this project is in its best quality, although we have tried our best to make it work. There are obviously rooms for development. However, as time does not allow, we would complete this application here. For the score, we suppose that approximately **1.3** would be best appropriate.

The reason is that we believe the score should reflect the attitude and the growth of students when participating in the course. During the period, we have focused on lectures and exercises, which provided us with lots of useful information about OOP. Together with building this Warehouse System as a practical project, we feel that we have learned a lot through this course, especially in engineering skill. Thus, a reasonable score such as **1.3** will best represent our result.

Some functionalities that we would like to add into the application:

1. Ordering Functionality: people, organizations outside of the warehouse for example factories, customers should also be able to order an issue/export. Thus, this functionality is important.
2. Resources Management Functionality: people inside the warehouse need to use resources of all kinds, such as Forklift, Clothes, etc.. Therefore, functionalities to manage these resources are also needed.
3. Customers Functionality: every business needs to store their customer information to further enhance the business. Function to enter, retrieve, analyze customer data are useful and should be implemented.