CPT304 Software Engineering II
Assignment Report

2022/2023 Semester 2

Major	Information and Computing Science
Student ID	2032801
Student Name	Jeongyeong Park

Introduction

The first challenge is the software should be able to calculate accurate commissions according to each employee's role. Three roles which are manager, supervisor, and salesperson have different commission rates respectively. Therefore, the software must calculate proper commissions based on the stated commission rate under various scenarios. Eventually, it needs to sum up the commission of every employee in the sales team and get the total sales commission for the entire sales team.

The second challenge is that the software must manage the structure of employees. The problem statements include a rule that managers and supervisors can have subordinates, and salespeople cannot have subordinates. In addition, managers can have any role as subordinates but supervisors cannot have managers as one's subordinates. This means the software should deal with a hierarchical data structure (tree structure).

The last challenge is the software must be able to handle changes. There are two possible changes that can occur according to the problem statement. The first one is a change in the sales team structure. The statement indicates that the number of subordinates for a particular employee may change occasionally. This means a new manager, supervisor, or salesperson can be added at any time. Therefore, the software should be adaptable enough to accommodate changes in a team structure while keeping the accuracy of commission calculations. The second possible change is the addition of new role or modification of commission ratio. As mentioned earlier, the commission is calculated based on employee's role and commission ratio. Thus, if the organization decides to add new role or modify the commission ratio, the software should be able to handle it without excessive efforts.

The Design

Design Pattern

The accuracy of the total commission of the sales team and the hierarchical structure of the sales team were considered at the same time and the composite design pattern is selected. The composite design pattern describes an object hierarchy and allows the software to treat both individual items and groups of objects uniformly. With this design, the software can get information such as the role, and commission of each sales employee, and also can deal with a total sales commission of the sales team formed as a tree structure. Since managers, supervisors, and salespersons are all sales employees and there are several methods that they all need to have as sales employees, the 'SalesEmployee' interface is used. As can be seen in the class diagram, figure1, 'Manager' class, 'Supervisor' class, and 'Salesperson' class implement 'SalesEmployee' interface. 'Manager' class and 'Supervisor' class are composites because they can have subordinates. Since salesperson cannot have subordinates, 'Salesperson' class is a leaf.

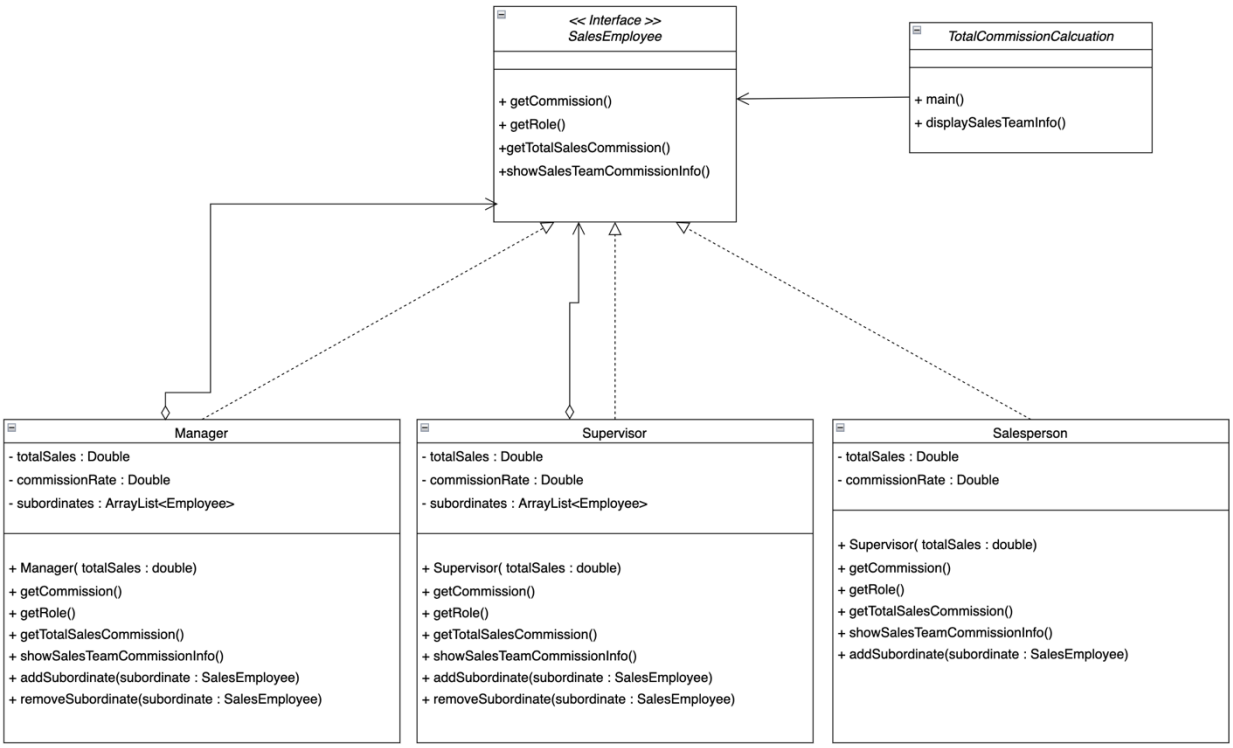

Figure 1. Class Diagram of the Software

'Manager' class and *'Supervisor' class* has basically the same structure except for some parts because supervisor cannot have manager as subordinates. The total sales commission is calculated with *getTotalSalesCommission()* method (Figure2). First, it adds employee one self's sales commission and then it adds subordinate's sales commission to get the total sales commission.

```java
@Override
public double getTotalSalesCommission() {
    double totalSalesCommission=0;

    // Adding one's commission first
    totalSalesCommission += getCommission();

    // Adding subordinates' commissions
    for (SalesEmployee subordinate : subordinates) {
        totalSalesCommission += subordinate.getTotalSalesCommission();
    }

    return totalSalesCommission;
}
```

Figure 2. getTotalSalesCommission() method in 'Manager' and 'Supervisor' classes

'Salesperson' class also has *getTotalSalesCommission()* method but since it cannot

have subordinate, it only adds up salesperson self's sales commission.

```
@Override
public double getTotalSalesCommission() {
    double totalSalesCommission = 0;
    return totalSalesCommission+getCommission();
}
```

Figure 3. getTotalSalesCommission() method in 'Salesperson' Class

Regarding the subordinates, *'Manager', 'Supervisor*', and *'Salesperson'* classes all have *addSubordinate(subordinate: SalesEmployee)* method. In *'Manager' class*, this method will *add* subordinate (SalesEmployee) to attribute, SalesEmployee Arraylist.

```
public void addSubordinate(SalesEmployee subordinate) {
    subordinates.add(subordinate);
}
```

Figure 4. addSubordinate(subordinate : SalesEmployee) method in 'Manager' class

In *'Supervisor'* class, this method will add subordinate (SalesEmployee) to attribute, SalesEmployee Arraylist. However, if manager is tried to be added, it will show the error message as can be seen in the figure 5.

```
public void addSubordinate(SalesEmployee subordinate) {
    if (subordinate.getRole().equals(anObject:"Manager")){
        System.out.println(x:"----------------------------------------");
        System.out.println(x:"Supervisor can't have manager as subordinate");
        System.out.println(x:"----------------------------------------");
    }
    else {
        subordinates.add(subordinate);
    }
}
```

Figure 5. addSubordinate(subordinate : SalesEmployee) method in 'Supervisor' class

'Salesperson' class is a leaf but still has this class. It is because when a user tries to add a subordinate to a salesperson, the software will inform the user it is impossible.

```
public void addSubordinate(SalesEmployee subordinate) {
    System.out.println(x:"----------------------------------------");
    System.out.println(x:"Sorry, Salesperson can't have subordinate");
    System.out.println(x:"----------------------------------------");
}
```

Figure 6. addSubordinate(subordinate : SalesEmployee) method in 'Salesperson' class

This design can also address the third problem mentioned earlier which is dealing with change. Even though the structure of the sales team has been changed, the structure can be modified and the total sales commission can be successfully calculated in the client class which is *'TotalCommissionCalculation' class*. Moreover, if the organization wants to add a new role to the sales team, it can be done by adding a new class regarding the role without modifying of existing code. It means that the design satisfies the open closed principal.

Coding Style and Naming Convention

The software was implemented in a way that other people can also easily understand. Every name in the source code is intuitive so everyone can easily know what is the purpose of it. Classes start with a capital letter and are nouns such as 'Manager' class and 'TotalCommissionCalculation' class. For methods, the names of them are verb and start in lowercase such as 'getRole()' and 'getTotalSalesCommission()'. Variables are named following the camel case naming convention, for example, totalSales and commissionRate. Commenting is done in order to explain parts that might be confusing. For example, showSalesTeamCommissionInfo() method that shows information about sales employees who are in the sales team has comment lines. With commenting, codes can be understood easily and know which information is printed.

```
@Override
public void showSalesTeamCommissionInfo(){
    // Displaying infomration about Sales employee oneself
    System.out.println(x:"-----------------------------------------");
    System.out.println("The role of the employee : "+getRole());
    System.out.println("Sales commission : $ "+getCommission());

    // Displaying information about subordinates
    for (SalesEmployee subordinate : subordinates){
        subordinate.showSalesTeamCommissionInfo();
    }
}
```

Figure 7. showSalesTeamCommissionInfo() method in 'Manager' class

Testing and Results

The software test was conducted with hardcoding. 5 cases were tested in the client class (*'TotalCommissionCalculation' class*). For better understanding, the data structure is showed in the figure. Example test code is only showed for the first case.

1. Testing Correct Data Structure

Data is hard-coded as can be seen below figure 8. Sales employees are created first and subordinates are added according to sales team structure. Finally, the head of the sales team is passed to displaySalesTeamInfo() method in client class to get the final result.

```
// -------------Test 1---------------
// Creating Sales Employee
Manager Haley = new Manager(totalSales:2000);
Supervisor Neil = new Supervisor(totalSales:5000);
Salesperson Jasmine = new Salesperson(totalSales:1000);
Salesperson Douglas = new Salesperson(totalSales:4000);

// Adding subordinates
Neil.addSubordinate(Douglas);
Haley.addSubordinate(Neil);
Haley.addSubordinate(Jasmine);

// Calculate total sales commission and display
displaySalesTeamInfo(Haley);
```

Figure 8. Data hard-coding in client class ('TotalCommissionCalculation' class)

The data structure looks like left figure below and the result got from the software can be seen in the right figure. The expected result and the real result are the same.

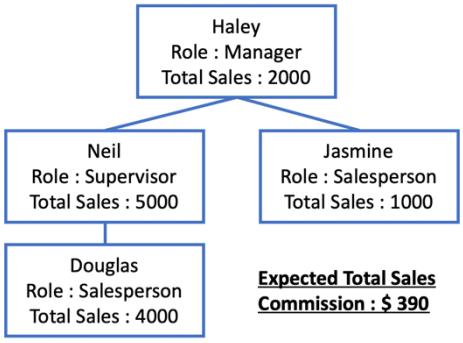

Figure 9. Test Data Structure

```
-----------------------------------------
The role of the employee : Manager
Sales commission : $ 40.0
-----------------------------------------
The role of the employee : Supervisor
Sales commission : $ 150.0
-----------------------------------------
The role of the employee : Salesperson
Sales commission : $ 160.0
-----------------------------------------
The role of the employee : Salesperson
Sales commission : $ 40.0
-----------------------------------------
Total sales commission is $ 390.0
-----------------------------------------
```

Figure 10. Test Result

2. Testing Incorrect Data Structure (Adding Manager as subordinate of Supervisor)

As stated in the problem statement, supervisor cannot have manager as subordinate. Therefore, if manager is added to subordinate of supervisor error message will be shown as can be seen from right figure. Other correct data will be added to data structure and total sales commission is calculated correctly.

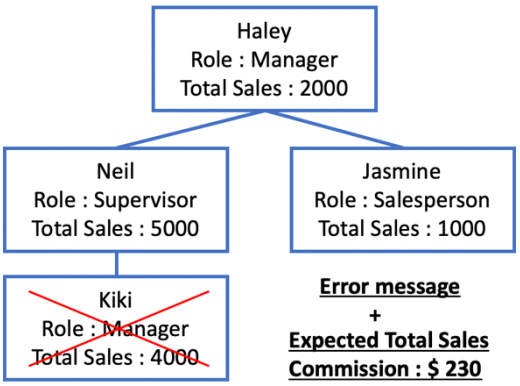

Figure 11. Test Data Structure

```
-----------------------------------------
Supervisor can't have manager as subordinate
-----------------------------------------
-----------------------------------------
The role of the employee : Manager
Sales commission : $ 40.0
-----------------------------------------
The role of the employee : Supervisor
Sales commission : $ 150.0
-----------------------------------------
The role of the employee : Salesperson
Sales commission : $ 40.0
-----------------------------------------
Total sales commission is $ 230.0
-----------------------------------------
```

Figure 12. Test Result

3. Testing Incorrect Data Structure (Adding other role as subordinate of Salesperson)

As stated in the problem statement, salesperson cannot have subordinate. Therefore, if any role is added to subordinate of salesperson, error message will be shown as can be seen from right figure. Other correct data will be added to data structure and total sales commission is calculated correctly.

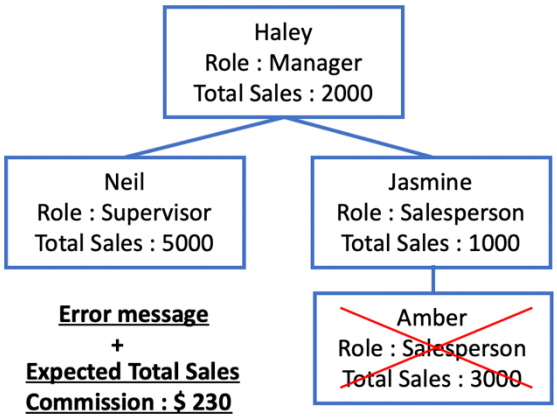

Figure 13. Test Data Structure

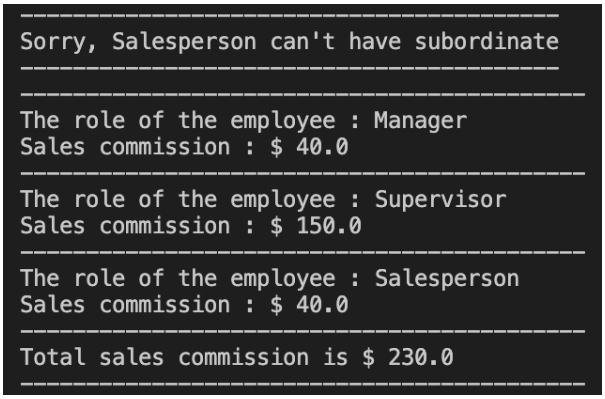

Figure 14. Test Result

4. Testing Extreme Data Structure

The sales team can be only formed with managers or supervisors. This test case will test a sales team consisting of only managers. The result is displayed as expected.

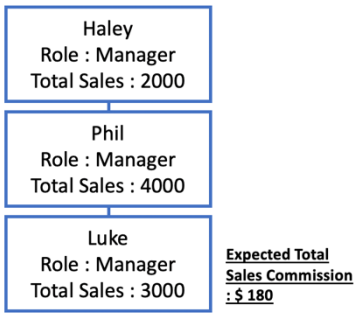

Figure 15. Test Data Structure

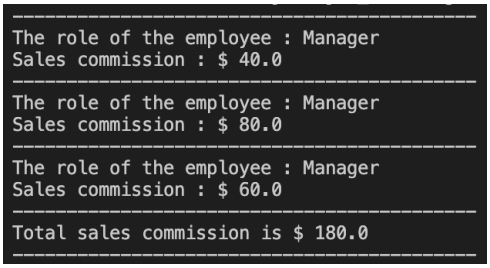

Figure 16. Test Result

5. Change of Data Structure

In order to check if the software can deal with change of data structure, salesperson 'Cam' is added after getting total commission of the sales team. As can be seen from the right figure, after adding 'Cam', total sales commission and information about Cam is correctly displayed with other sales employees.

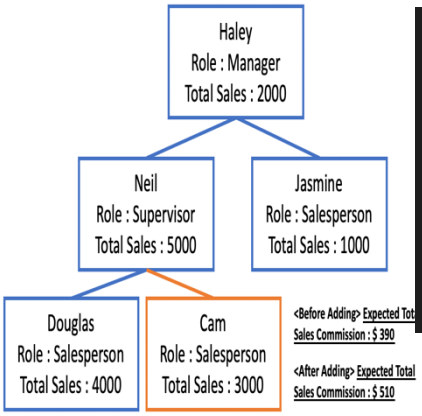

Figure 17. Test Data Structure

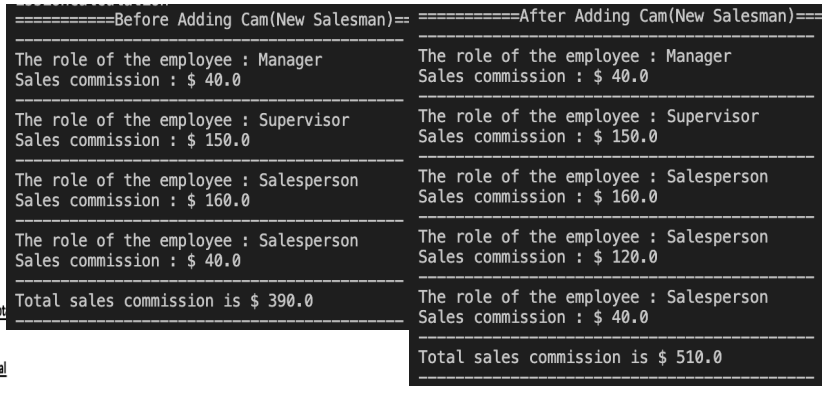

Figure 18. Test Result divided into two parts