Seminar 5
Higher Grade Tasks

Object-Oriented Design, IV1350

Ayda Ghalkhanbaz
aydag@kth.se

June 3, 2024

Contents

1 Introduction

The additional higher-grade tasks of the project include incorporating another design pattern, namely the Template pattern, and writing unit tests for all print statements to `System.out` in the POS program, which was developed over four previous seminars. Moreover, an additional task involves comparing the functionality of inheritance and composition as design patterns by utilizing an optional class from the Java libraries.

This report details the implementation of the mentioned design patterns and the unit tests written for this seminar. The results will be shown and discussed in later sections. The code is written in Java using Visual Studio Code, and the tests are written using the JUnit 5 library. I collaborated with Naveed Rahman and Nora Say to accomplish this task.

2 Method

2.1 Task 1

In preparation for completing this task, the relevant section in Chapter 9 of the book "A First Course in Object Oriented Development" was thoroughly examined. To implement the template design pattern for the observers in the program, a new file named `TotalRevenueTemplate` was created in the View layer. The class created in this new file implemented the `SaleObserver` interface.

Subsequently, the provided pseudo-code for the task was incorporated into the template and adjusted as necessary. Since this template class was intended to be inherited by other classes, two abstract methods, `doShowTotalIncome` and `handleErrors`, were defined within it.

In the next step, the subclasses `TotalRevenueView` and `TotalRevenueFileOutput` were modified to conform to these changes. This involved implementing the `doShowTotalIncome` and `handleErrors` methods.

Finally, the corresponding method calls in the `Sale` class were updated to integrate the new pattern.

2.2 Task 2

The first step in this task involved identifying suitable class candidates from the Java libraries to implement adapting classes. This required considerable research to determine what and how to adapt from these classes. Subsequently, the relevant section in Chapter 9.3 of the course literature was thoroughly examined to gain a solid understanding of the concept of composition and its differences from the inheritance pattern.

Once a strategy for adapting classes was established, the actual code implementation was straightforward. Beginning with the adapting class using inheritance, the corresponding methods were overridden and additional functionalities were added to this class. For the adapting class utilizing composition, an instance of the superclass was created as an attribute in the subclass. Then, the same functionalities implemented in the subclass using inheritance were coded for this class.

Finally, a Main class with a main method was created to test the functionality of the written subclasses and verify whether they behaved as intended.

2.3 Task 3

In the final task, all classes and their methods that print to `System.out`, such as `View`, `Printer`, `TotalRevenueView` and `AccountingSystem` were tested. The tests were written using the JUnit 5 library. The annotations `@BeforeEach`, `@AfterEach`, and `@Test` were used to define the functionality of each test method.

The first two annotations are executed before and after each test method, respectively, while the last annotation specifies the actual method to be run for testing. The primary methods used for validating the program were `assertAll` in conjunction with `assertTrue` to verify that the expected values matched the actual values produced by the methods.

Since the tests involved print statements, additional built-in classes such as `ByteArrayOutputStream` and `PrintStream` were employed to capture the output. The output was then converted to a string and compared to the expected output.

3 Result

The code can be found on: GitHub.

Navigate through: `Task1&3-pos-src-main-java-se-kth-iv1350-view` to find the solution to the first task, you can also find the solution to the third task through the test branch, that is `Task1&3-pos-src-test-java-se-kth-iv1350`. To find the solution to second task, navigate through: `task2` folder.

3.1 Task 1

As discussed in the previous chapter, the initial step in this task was creating the file `TotalRevenueTemplate`. This template, by implementing the `SaleObserver` interface, is responsible for updating the total revenue, important for calculating the overall income whenever a customer completes a sale. Therefore, it overrides the method `updateRevenue`.

A supporting method, `newSaleWasMade`, invokes both the `updateRevenue` method and `showTotalIncome`. The `showTotalIncome` method then invokes the implemented methods `doShowTotalIncome` and `handleErrors`, which is in the adapting classes.

Following the template pattern, classes `TotalRevenueView` and `TotalRevenueFileOutput` were adjusted. These classes inherit from `TotalRevenueTemplate` via the "extends" keyword. Despite sharing the presenting of the total revenue to observers, either through screen output or file printing, their approaches differ.

`TotalRevenueView` are connected to the abstract methods in the template, `doShowTotalIncome` and `handleErrors`. The `doShowTotalIncome` method call the private method `printTotalRevenue`, with the print statements. Also, `handleErrors` is responsible for showing the cause of any exceptions that might be thrown by `doShowTotalIncome`.

Similarly, `TotalRevenueFileOutput` implements the two abstract methods from the template, and has the same the functionality as `TotalRevenueView`. However, it differs in its output by directing notifications to a file instead.

Consequent to these adjustments, the method `newSaleWasMade` is now called within `Sale.notifyObservers` instead of `updateRevenue`.

The result of sample run for `TotalRevenueFileOutput` can be found below.

.....The Revenue log using FileLogger .....

```
Current total revenue at 2024-05-29 12:56:56 is: 83,75 kr.

Current total revenue at 2024-05-29 12:56:56 is: 626,27 kr.
```

3.2 Task 2

The class selected for this task to inherit from is `HashSet` from `java.util`, which belongs to the `Collection` hierarchy alongside other classes such as `List`, `Stack`, `LinkedList`, and `PriorityQueue`.

The first subclass, named `HashSetInheritance`, inherits from the `HashSet` class. This class manages only integers and has a `boolean` attribute, `negativeNumbers`, which can be adjusted in its `constructor`. When set to true, negative numbers are permitted; otherwise, the attribute defaults to false. The `add` and `contains` methods have been overridden to follow the requirements of the adapting class.

The add method inspects the element to be added based on the `negativeNumber` setting, determining whether negative numbers are allowable. If the number is less than 101, it is added using the superclass's add method. However, if the number exceeds 101, it is decremented by 100 before being added.

Similarly, the contains method has conditional checks based on the `negativeNumber` setting and utilises the superclass's contains-method to execute.

Furthermore, the `HashSetInheritance` class includes three additional methods: `intersection`, `union`, and `difference`, which are the most common operations on sets. Their implementations are straightforward, employing for-loops, if-statements, and the overridden `add` and `contains` methods to execute their respective tasks.

The `HashSetComposition` class serves a similar purpose to `HashSetInheritance`, featuring its own definitions of `intersection`, `union`, and `difference` methods. However, unlike `HashSetInheritance`, this class does not inherit from `HashSet`. Instead, it maintains a private attribute of type `HashSet`, which it utilizes to perform operations. Additionally, this class provides a range of getter, `add`, `remove`, and `contains` methods.

Finally, in the `Main` class, these two classes undergo testing, with the results shown in Figure 3.1.

3.3 Task 3

The implemented tests for the `View`, `TotalRevenueView`, `Printer`, and `AccountingSystem` classes confirm that all print statements give the expected output.

Since the tests are designed to verify outputs as strings, we utilized `ByteArrayOutputStream` and `PrintStream` to capture the content logged to `System.out`. The characters captured in the `ByteArrayOutputStream` instance are then converted to a string. This output is tested by using the `contains` method to check whether the expected output is present in the actual output. Each of these tests is validated by an `assertTrue` method. For simplicity, all these `assertTrue` validations are executed within a single `assertAll` operation.

All the tests written for this task follow a similar structure.

```
Set1:
[1, 2, 3, 4, 5, 6]
Set2:
[4, 5, 6, 7, 8, 9]

---------Composition---------
set 1 intersect set2: [4, 5, 6]
set 1 union set2: [1, 2, 3, 4, 5, 6, 7, 8, 9]
set 1 intersect set2: [1, 2, 3]

---------Inheritance---------
set 1 intersect set2: [4, 5, 6]
set 1 union set2: [1, 2, 3, 4, 5, 6, 7, 8, 9]
set 1 intersect set2: [1, 2, 3]
```

Figure 3.1: The test outputs for composition vs inheritance implementations

4 Discussion

4.1 Task 1

The implemented Template design pattern for the `SaleObserver` originates from the provided pseudocode within the task. It is an abstract class that implements the `SaleObserver` interface and overrides the method `updateRevenue` to calculate the updated total revenue. Additionally, it includes a private method, `showTotalIncome`, which calls the printing functions from its adapting classes. Essentially, the `TotalRevenueTemplate` class encapsulates the common algorithm structure, which its subclasses then implement. This class also has two abstract methods, `doShowTotalIncome` and `handleErrors`, implemented by its subclasses.

The subclasses, `TotalRevenueView` and `TotalRevenueFileOutput`, inherit the updating of revenue from the template class and adjusts the print methods to their specific purposes (i.e., printing notifications on the screen or writing them to a file). As mentioned earlier, these subclasses provide their own implementations of the abstract methods from their superclass, adapted to their structure and purposes. In conclusion, the output from the sample runs, shown in Figure 3.1, validates the functionality of the template class and its subclasses, showing the structure of the implemented design pattern.

Utilising the Template pattern design has added consistency and reusability to the program. Since the common algorithm is defined only in the abstract class, the code is no longer duplicated across several classes. In other words, updating the total revenue is now centered in the template method, while the subclasses focus on their specific tasks.

However, despite its benefits, the Template design pattern has also introduced complexity and sophistication to the program. The `updateRevenue` method has a straightforward structure, and dedicating a template for this simple algorithm adds unnecessary complexity. In the previous version of the program (before implementing the template), each adapting class implemented the `SaleObserver` interface and overrode the `updateRevenue` method, resulting in a simpler structure.

In my opinion, the Template design pattern would be more beneficial if the common algorithm structure were more complex. In such a scenario, the subclasses could focus on their specific tasks while letting the superclass handle the more challenging aspects and keep the program more consistent by avoiding the duplicated code across the program.

Therefore, although the Template design pattern brings consistency, reusability of common code, and flexibility, it can also complicate the program by adding an extra level of abstraction, especially in this case where the code in the template is relatively simple and might not call for a dedicated abstract class.

4.2 Task 2

Composition and inheritance are two fundamental principles of OOP (object-oriented programming) that offer different ways to code reusability. This task provided good understanding of which patterns is preferred and how to choose the appropriate one for a program.

Starting with inheritance, it is good for less code redundancy. For instance, in our case study, when the class `HashSetInheritance` inherits from `HashSet`, it reuses all the code written in the superclass, i.e., it inherits all of the code and extends the base class with its implementation of the `intersection`, `union`, and `difference` methods. However, in this particular case, we have overridden the add and contains methods to suit our needs. Inheritance also ensures that subclasses follow a standard interface, meaning that all subclasses have access to all public and protected methods and attributes of the superclass. In this case, the `HashSetInheritance` class has all methods implemented in the `HashSet` class, such as `add`, `remove`, `contains`, etc., giving consistency across all implemented methods in the subclasses.

However, inheritance comes with some drawbacks that might influence our choice of pattern. The first drawback is that inheritance makes the code tightly coupled. This means that even a small change in the superclass can affect its subclasses, as inheritance creates a strong relationship between super and subclasses. These changes might cause serious issues, especially if the program is complex, and due to this tight coupling updating the subclasses might be challenging. Although `HashSetInheritance` is not a very large and complex program, making changes in the code should not be too difficult if the superclass is modified, but it can still be time-consuming.

Another significant drawback of inheritance is that it breaks encapsulation. As mentioned earlier, all public and protected methods and attributes in the superclass are exposed to its derived classes, leading to tight coupling between subclasses and the base class. In the case of `HashSetInheritance`, this class is highly dependent on its parent, `HashSet`. All defined methods in this class would not function if the superclass changes.

Due to these potential problems, composition is often preferred. Composition offers some benefits, including low coupling. It utilises existing classes to create more complex classes without breaking encapsulation and promotes code reuse. It allows users to create an instance of an object and implement their own methods, minimising dependency on the class. In our case, `HashSetComposition` has an attribute of the type `HashSet` and creates its own version of `add`, `contains`, and `isEmpty` methods. In other words, composition provides more flexibility as objects can be dynamically composed at runtime and easily adjusted without affecting the entire program.

A good way to determine which pattern is suitable for a program is to assess the relationship between the program and the (super)class. Inheritance is ideal if the class has an "is-a" relationship with the superclass, whereas composition is preferred in a "has-a" relationship.

The implemented code for this task is not large or complex enough to be significantly affected by the drawbacks of inheritance or the benefits of composition. Although both patterns suit this program well, composition is preferred here since this class has a

"has-a" relationship with the HashSet class.

4.3 Task 3

The tests are written using the `JUnit` framework. All the tests are self-evaluating, meaning they print an informative message only in the event of a failure, while they remain silent if the test passes. These tests are located in the test directory within the same package as the System Under Test (SUT) alongside the previous tests.

Each class under test—namely `View`, `TotalRevenueView`, `AccountingSystem`, and `Printer`—has its own dedicated test class. The parameters tested are those of significant interest, such as total price, item price, and cash back to the customer.

To verify the print statements directed to System.out, we captured the output using a `ByteArrayOutputStream`. This approach allows us to convert the captured output into a string and verify its correctness. The validation process involves checking if the expected output is present within the captured string using the contains method.

Each individual validation is performed using the `assertTrue` method checking whether the expected conditions hold true. To make sure of comprehensive evaluation and maintain clarity, all these `assertTrue` validations are executed within a single `assertAll` operation. This method makes sure that if any assertion fails, all the failures are reported together making it easier to find and address issues.

Overall, the correctness and reliability of the program is approved by a structured validation through assertion operations, output capture and interesting parameter checks.