Seminar 5

Object-Oriented Design, IV1350

Dana Ghafour, danagf@kth.se

2023-06-04

Contents

1 Introduction

Seminar 1

This seminar task involves improving the score of one or more previous seminars by submitting an updated report and solution. The seminar that I chose to improve is seminar 1, where I created a domain model and a system sequence diagram for a retail store's sales process. I worked with Ali Madlool on seminar 1, task 1 and 2.

Seminar 4

This seminar task was to improve the code of a point-of-sale system using design patterns. The code was previously written for seminar 4. The goal is to make the code more flexible and robust by applying suitable design patterns. For the improvement, I decided to use the following design patterns:

1. Strategy: To implement different discount algorithms for the sale class.
2. Observer: To notify different observers when a payment is made.
3. Singleton: To ensure that only one instance of some classes exists.

2 Method

Seminar 1

To improve my solution for seminar 1, I followed these steps:

1. I reviewed the feedback that I received from the teacher and the peer assessment group on my original report and solution.

2. I identified the main defects and areas of improvement in my domain model, such as missing or redundant classes, attributes, and associations, incorrect or unclear naming conventions, and lack of explanation or justification.

3. I revised my domain model based on the feedback and the established guidelines for object-oriented analysis and design. I also consulted the course literature and some online resources.

Seminar 4

I used the following steps to choose and implement the design patterns in the code:

1. I analyzed the feedback from the teacher and identified some areas where the code could be improved using design patterns. For example, I noticed that the discount calculation was hard-coded in the sale class, which made it inflexible and difficult to change. I also realized that I could use observers to update different systems when a payment is made, instead of calling them explicitly in the controller class. I also wanted to make sure that some classes, such as the inventory system and the accounting system, were only instantiated once and shared by all users.

2. I searched for design patterns that could solve my problems and found the strategy, observer, and singleton patterns. I studied their structure, participants, etc. from the course book. I also looked at some examples of these patterns in Java.

3. I applied the strategy pattern to the sale class by creating an interface called Discounter that defined a common method for all discount algorithms. I then created concrete classes that implemented this interface, such as QuantityDiscounter and CustomerDiscounter. I also added a field and a method to the sale class to set and use a Discounter object as a strategy for applying discounts.

4. I applied the observer pattern to the payment class by creating an interface called PaymentObserver that defined a method for updating observers when a payment is made. I then created concrete classes that implemented this interface, such

as TotalRevenueObserver and TotalRevenueFileOutput. I also added a list of observers and a method to notify them to the payment class. I also modified the controller class to add observers to the payment class when initializing a new sale.

5. I applied the singleton pattern to thee inventory system and accounting system classes by making their constructors private and adding a static field and a static method to access their single instances. I also modified the controller class to use these methods instead of creating new instances of these classes.

6. I tested the code using JUnit and verified that it worked as expected. I also checked that the code was readable, documented, and followed naming conventions.

3 Result

Seminar 1

This chapter presents my improved solution for seminar 1, task 1, along with an explanation of the changes that I made.

Task 1. Domain Model

Figure 3.1 shows my revised domain model for a retail store's sales process. Compared to my original domain model, I made the following changes:

1. I renamed the class DiscountDatabase to DiscountCatalog, to better reflect its role as a collection of discounts rather than a database system.

2. I removed the class ItemIdentifier, since it was redundant and did not represent a real-world entity. Instead, I added an attribute itemId to the Item class.

3. I have an association between Payment and Item, since it was necessary to reflect the reality of how payments and items are related. The asssoociation added was "givenAfter". I also added an association between Item and Sale, "soldIn", to show that a sale consists of one or more items.

4. I added an association between Discount and Sale, to show that a sale may have a discount applied to it. I also added an attribute discount to the Sale class, to store the amount of discount for the sale.

I think these changes make my domain model more accurate, consistent, and understandable.

Seminar 4

This subsection presents my improved code using design patterns. I changed the following classes when implementing the chosen patterns:

1. Sale: I applied the strategy pattern to this class by creating an interface called Discounter that defined a common method for all discount algorithms. I then created concrete classes that implemented this interface, such as QuantityDiscounter and CustomerDiscounter. I also added a field and a method to the sale class to set and use a Discounter object as a strategy for applying discounts.

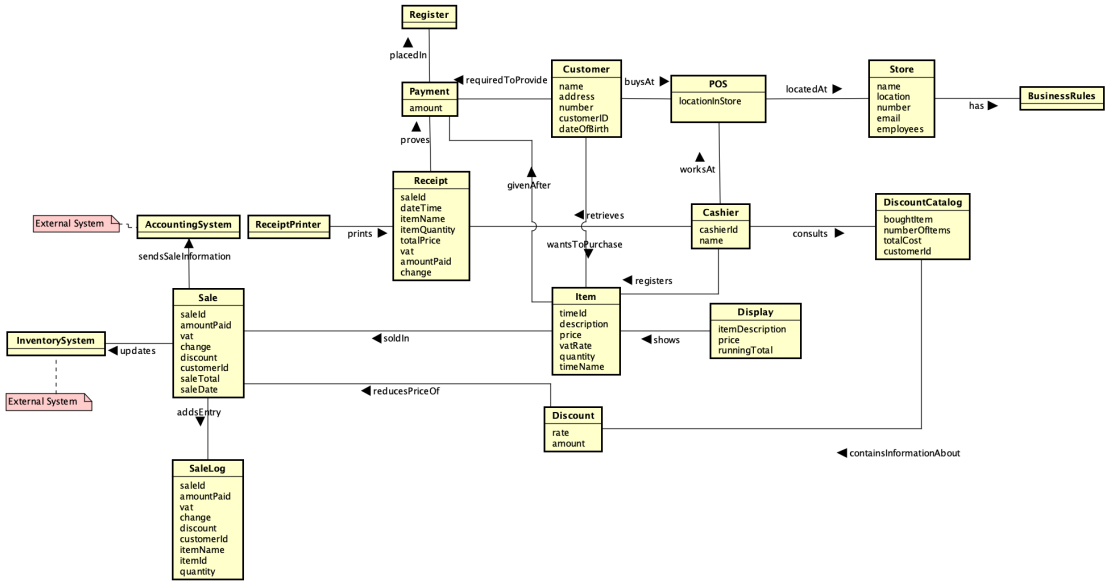

Figure 3.1: The revised domain model for a retail store's sales process.

2. Payment: I applied the observer pattern to this class by creating an interface called PaymentObserver that defined a method for updating observers when a payment is made. I then created concrete classes that implemented this interface, such as TotalRevenueObserver and TotalRevenueFileOutput. I also added a list of observers and a method to notify them to the payment class.

3. InventorySystem and AccountingSystem: I applied the singleton pattern to these classes by making their constructors private and adding a static field and a static method to access their single instances.

You can find the source code for all these classes in the GitHub repository: `https://github.com/danaghafour/iv1350`

Figure 3.3 shows the first part of the printout of the sample run of my system.

Figure 3.3 shows the second part of the printout of the sample run of my system.

```
Product ID was not found in the inventory system.
Developer log: The inventory system server is currently unavailable.

Inventory system server is down.
Product ID was not found in the inventory system.
Products in sale:
Price of products in sale:
Quantity of products in sale:
Price of products in sale: 0.0
The discounted price is: 0.000

--------------------
Total revenue: 0.0
--------------------

Product ID was not found in the inventory system.
Product ID was not found in the inventory system.
Products in sale:
Eggs
Price of products in sale:
25.0
Quantity of products in sale:
1
Price of products in sale: 25.0
```

Figure 3.2: The first part of the printout of the sample run of the system.

```
Price of products in sale: 0.0
The discounted price is: 0.000

--------------------
Total revenue: 0.0
--------------------

Product ID was not found in the inventory system.
Product ID was not found in the inventory system.
Products in sale:
Eggs
Price of products in sale:
25.0
Quantity of products in sale:
1
Price of products in sale: 25.0
The discounted price is: 0.000

--------------------
Total revenue: 25.0
--------------------
```

Figure 3.3: The second part of the printout of the sample run of the system.

4 Discussion

In this part, I highlight my achievements according to the assessment criteria and provide some concrete examples from my solution.

Seminar 1

I don't think my DM is a programmatic DM or a naive DM. A programmatic DM is one that models the program rather than the reality, and a naive DM is one that models only what is explicitly stated in the requirements specification. I tried to avoid these mistakes by identifying and organizing classes that represent real-world entities, such as Customer, Item, Sale, Payment, Discount, etc. I also included classes that are outside the requirements specification but are relevant to the problem domain, such as Store, BusinessRules, ReceiptPrinter, Display, etc.

I think my DM is understandable and correct. It shows all the relevant classes for the sales process, their attributes and associations, and how they relate to each other. It covers both the basic flow and the alternative flows of the sales process. It also follows the business rules and clarifications given in the requirements specification, such as the different VAT rates for items, the discount calculation based on various factors, and the information to be included in the receipt.

I think my DM has a reasonable number of classes. I don't think any important classes are missing or any irrelevant classes are present. I tried to include all the classes that are necessary to model the sales process and its actors and external systems. I also tried to remove any redundant or unnecessary classes that do not represent real-world entities or do not add any value to the model. I don't think my DM has any spider-in-the-web classes. A spider-in-the-web class is one that has too many associations with other classes, making it overly complex and hard to understand. I tried to avoid this mistake by limiting the number of associations for each class to four or five at most, as recommended by the course literature. I also tried to make sure that each association is meaningful and reflects a real-world relationship between classes. I agree with the choices between class and attribute. A class is a representation of a real-world entity that has identity and behavior, while an attribute is a property of a class that describes its state or characteristics. I tried to follow this distinction by making classes for entities that have identity and behavior, such as Customer, Item, Sale, Payment, Discount, etc., and making attributes for properties that describe their state or characteristics, such as name, address, number, dateOfBirth for Customer; price, vatRate, quantity, itemId for Item; amountPaid, change, discount, saleTotal for Sale; etc.

I think there is a reasonable number of associations in my DM. I understand them and I don't think any important associations are missing. I also don't think there are any

classes without associations. An association is a link between instances of classes that indicates some kind of relationship or interaction between them. I tried to include all the associations that are relevant to the sales process and its actors and external systems. For example, the association "buysAt" between Customer and POS, "wantsToPurchase" between Customer and Item, "requiredToProvide" between Customer and Payment, and so on. I also tried to give descriptive names to each association that indicate the nature of the relationship or interaction between classes, such as shows, retrieves, givenAfter, etc.

I followed naming conventions in my DM. For example, I used pascal case notation for class names (e.g., Customer, Item, Sale, etc.) and camel case notation for attribute names (e.g., name, address, number, dateOfBirth, etc.). I used singular nouns for class names (e.g., Customer, Item, Sale, etc.) and plural nouns for collection classes (e.g., DiscountCatalog, SaleLog, etc.). I also used descriptive and meaningful names that indicate the purpose or role of each class and attribute.

Seminar 4

In this part, I highlight my achievements according to the assessment criteria and provide some concrete examples from my solution.

1. Exception handling: my solution follows all the best practices for exception handling from chapter eight of the lab tutorial. I use checked exceptions for recoverable errors, such as database failure or file output failure. I use descriptive messages and stack traces to provide useful information about the errors. I avoid using empty catch blocks or swallowing exceptions. I handle exceptions at the appropriate level and propagate them when necessary. I use finally blocks to ensure that resources are released in case of an exception. I use exception chaining to preserve the original cause of the error. I use custom exceptions to indicate specific errors in my system. I use exception specifications to declare the exceptions that a method can throw.

2. Pattern implementation: my solution implements the strategy, observer, and singleton patterns correctly and effectively according to their structure, participants, and collaborations as described in the course book. I follow the naming conventions and coding standards for these patterns. I use design principles, such as favoring composition over inheritance or programming to interfaces rather than implementations, to make my code more flexible and type-safe.

3. Use of design patterns: my solution explains and motivates the use of design patterns in my code. I state the problems they solve and the benefits they provide. I compare my improved code with my previous code and highlight the differences and advantages of using design patterns. I use design patterns appropriately and consistently throughout my system.

4. Choice of observer and observed object: I choose the observer and observed object in my solution logically. I choose the payment class as the observed object because

it represents an event that occurs in my system and affects other objects. I choose the total revenue observer and the total revenue file output as observers because they represent different systems that need to be updated when a payment is made.

5. Reference to observer passed to observed object: I pass the reference to observer to observed object by using a method called addPaymentObserver in the payment class. This method takes a PaymentObserver object as a parameter and adds it to a list of observers in the payment class. This method follows the principle of dependency inversion, which reduces the coupling between the payment class and its observers and makes them more loosely coupled. It also increases the cohesion of each class because they have a clear responsibility and role in the system.

6. Data passed from observed object to observer: I pass data from observed object to observer by using a method called update in the PaymentObserver interface. This method takes a Payment object as a parameter and updates the observer according to its logic. This method follows the principle of information hiding, which preserves the encapsulation of the payment class and makes it more secure and reliable. It also provides relevant information to its observers that they need to perform their tasks.