Seminar 4

Object-Oriented Design, IV1350

Mostafa Faik

2025-05-19

Project Members:
[Mostafa Faik, mfaik@kth.se]
[Derfesh Mariush, derfesh@kth.se]
[Allan Al Saleh, Allan2@kth.se]

Declaration:

By submitting this assignment, it is hereby declared that all group members listed above have contributed to the solution. It is also declared that all project members fully understand all parts of the final solution and can explain it upon request.

It is furthermore declared that no part of the solution has been copied from any other source (except for resources presented in the Canvas page for the course IV1350), and that no part of the solution has been provided by someone not listed as a project member above.

1 Introduction

This report details the implementation and design decisions made for the Process Sale system as part of the seminar tasks. The primary focus is on enhancing the robustness, maintainability, and extensibility of the system through effective use of exception handling and design patterns. Task 1 addresses the handling of exceptional conditions such as searching for non-existent items and simulating database failures using custom exceptions, ensuring proper notification to users and developers, as well as maintaining program state integrity. Task 2 implements the Observer pattern to track and display total revenue generated by all sales, along with the application of two additional (GoF) design patterns to modularize and improve discount calculation. The report covers the architectural layers involved, the design choices made, and includes testing strategies to validate the correctness of the implemented functionality.

2 Method

The steps used in this seminar is based on principles from the course literature "A First Course in Object-Oriented Development" by Leif Lindbäck, and guided by examples from lectures and exercises.

Exception Handling Implementation:

To address Task 1, we began by analyzing the alternative flow 3-4a, where a search for an item identifier might fail because the item does not exist in the inventory catalog. We decided to implement a custom checked exception, ItemNotFoundException, to explicitly signal this condition. Choosing a checked exception ensures that the calling code is forced to handle or declare the exception, promoting robust error handling at compile time.

Simultaneously, we implemented a DatabaseFailureException to simulate situations where the database cannot be accessed, such as when the server is down. This exception is thrown intentionally when a hardcoded item identifier is searched, mimicking a real-world failure scenario. By separating these two concerns into distinct exception types named clearly after their error conditions, the design achieves both clarity and appropriate abstraction.

We made sure that these exceptions inherit from java.lang.Exception and included meaningful error messages to convey the nature of the problem to both users and developers. The exception handling is integrated so that when an exception is thrown, the program state remains consistent and unchanged, following best practices to prevent side effects. The user interface layer catches these exceptions and presents informative messages, while error details are logged to a file using a dedicated logger class, facilitating developer diagnostics without disturbing the user experience.

Unit tests were written to cover all exception scenarios, verifying that exceptions are properly thrown and that program state remains intact. This testing approach ensures reliability and adherence to the specifications.

Observer Pattern Implementation:

For Task 2a, the Observer pattern was applied to implement real-time tracking of total revenue from all sales since the program started. The Sale class acts as the subject, maintaining a list of observers interested in revenue updates.

Two observer classes were created:

- TotalRevenueView, which displays the total income on the console (System.out), providing immediate feedback to the user.

- TotalRevenueFileOutput, which logs the total revenue to a file, enabling persistent storage for auditing or reporting purposes.

Both observers implement a shared observer interface, ensuring a consistent update method. This decouples the revenue display logic from the core sale processing, adhering to the principle of separation of concerns and allowing observers to be added or removed without impacting the core system.

The observers are notified whenever a sale is completed and the total revenue changes, receiving updated data through the observer interface. This design avoids direct calls between views and controllers, promoting loose coupling and enhancing maintainability.

Implementation of Additional Design Patterns:

In Task 2b, two additional (GOF) design patterns were employed to enhance system modularity and flexibility:

- Strategy Pattern: This pattern was used to encapsulate various discount calculation algorithms. Different discount strategies, such as ItemBasedDiscountStrategy, TotalBasedDiscountStrategy, and CustomerDiscountStrategy, implement a common interface for calculating discounts based on different criteria. This approach allows the system to dynamically select or combine discount policies without modifying existing code, facilitating future extensions.
- Composite Pattern: To support multiple discount strategies simultaneously, the CompositeDiscountStrategy class was designed to aggregate several discount strategies. It calculates the total discount by summing the individual discounts returned by each strategy. This structural pattern simplifies the management of complex discount rules by treating individual and composite discounts uniformly.

Both patterns contribute to the Open-Closed Principle, enabling the system to be open for extension but closed for modification. They also improve testability by allowing each strategy to be tested in isolation or as part of a composite.

Throughout the implementation, extensive unit tests were created to verify the correctness of each pattern and its integration within the sale process. Mock objects and controlled inputs ensured that scenarios, including boundary cases and error conditions, were adequately tested.

3 Result

Overview of the Program:

The developed program simulates a point-of-sale system that handles sales transactions and applies various discount strategies. It robustly manages exceptions related to inventory lookups and database connectivity, providing clear user feedback and error logging.

Additionally, the program tracks the total revenue from all sales using the Observer pattern, while employing Strategy and Composite design patterns to flexibly calculate discounts.

Source Code Changes for Observer Pattern:

To implement the Observer pattern for tracking total revenue, the following key classes were created or modified:

- Controller: The controller was extended to catch exceptions thrown during item lookups, such as ItemNotFoundException and DatabaseFailureException. It propagates user-friendly messages to the view and triggers logging of errors when necessary.
- View: The view was updated to display informative messages when exceptions occur, such as notifying the user if an item ID does not exist or if the database is unreachable. It also displays sale details, including items added, total costs, VAT, and final receipt information. Additionally, the view subscribes to revenue updates from the observer pattern, displaying the total revenue after each completed sale.
- Sale: The Sale class was updated to maintain a list of observers subscribing to revenue updates. After each sale, the total revenue is updated and all registered observers are notified via a common interface method.
- TotalRevenueView (Observer): This class implements the observer interface and outputs the cumulative revenue to the console (System.out), providing immediate visual feedback to the user.
- TotalRevenueFileOutput (Observer): Also implementing the observer interface, this class writes the total revenue updates to a log file using file I/O operations, enabling persistent revenue tracking.

These classes interact solely through the observer interface, ensuring loose coupling and modularity.

Source Code Changes for Strategy and Composite Patterns:

For flexible and extensible discount calculation, the following changes were made:

- DiscountStrategy Interface: Defined a common interface for all discount calculation strategies.
- ItemBasedDiscountStrategy, TotalBasedDiscountStrategy, CustomerDiscountStrategy: These classes implement different discount calculation rules, encapsulating the logic to calculate discounts based on items, total price, and customer information, respectively.

- CompositeDiscountStrategy: This class implements the composite pattern by aggregating multiple discount strategies and calculating the total discount as the sum of the individual discounts. This enables combining various discount rules seamlessly.
- DiscountDatabase and DiscountDatabaseImpl: These classes simulate a discount database accessed by the strategy classes for retrieving discount rates or amounts.

Through these changes, the program supports adding or modifying discount policies without impacting the core sale process, improving maintainability and scalability.

Git Repository:

The full source code and test cases for this project are publicly available in the following GitHub repository:
https://gits-15.sys.kth.se/derfesh/IV1350-VT25-Objektorienterad-design

Sample Run Output:

Below is a sample output from running the program that demonstrates exception handling and observer notifications:

```
Sale initiated.

Item added:
Item ID: 1
Item name: BigWheel Oatmeal
Item cost: 29.90 SEK
VAT: 6.0%
Item description: BigWheel Oatmeal 500 ml

Total cost (incl VAT): 31.69 SEK
Total VAT: 1.79 SEK

Item added:
Item ID: 1
Item name: BigWheel Oatmeal
Item cost: 29.90 SEK
VAT: 6.0%
Item description: BigWheel Oatmeal 500 ml

Total cost (incl VAT): 63.39 SEK
Total VAT: 3.59 SEK

Item added:
Item ID: 2
Item name: YouGoGo Blueberry
Item cost: 14.90 SEK
VAT: 6.0%
Item description: YouGoGo Blueberry 240 g

Total cost (incl VAT): 79.18 SEK
Total VAT: 4.48 SEK

Item added:
Item ID: 3
Item name: Luxury Bread
Item cost: 49.90 SEK
VAT: 12.0%
Item description: Just a normal bread
```

Figure 1: Sample output of the program

```
Item added:
Item ID: 4
Item name: Golden Apple
Item cost: 149.90 SEK
VAT: 25.0%
Item description: It's a golden one

Total cost (incl VAT): 322.45 SEK
Total VAT: 47.95 SEK

Item added:
Item ID: 1
Item name: BigWheel Oatmeal
Item cost: 29.90 SEK
VAT: 6.0%
Item description: BigWheel Oatmeal 500 ml

Total cost (incl VAT): 354.14 SEK
Total VAT: 49.74 SEK
[USER MESSAGE] The item with ID 'fail' was not found. Please check the ID and try again.

Total cost (incl VAT): 354.14 SEK
Total VAT: 49.74 SEK
[USER MESSAGE] The item with ID '999' was not found. Please check the ID and try again.

Total cost (incl VAT): 354.14 SEK
Total VAT: 49.74 SEK
[USER MESSAGE] There is a technical issue accessing the database. Please contact support.

Total cost (incl VAT): 354.14 SEK
Total VAT: 49.74 SEK

End sale:
Discount applied: 37.71 SEK
Total cost (incl VAT): 316.43
Accounting system updated with payment: 1000.00
Inventory system updated
```

Figure 2: Sample output of the program

```
----------------------- Begin receipt -----------------------
Time of Sale: 2025-05-19 22:13

BigWheel Oatmeal 3 x 29.90 89.70 SEK
YouGoGo Blueberry 1 x 14.90 14.90 SEK
Luxury Bread 1 x 49.90 49.90 SEK
Golden Apple 1 x 149.90 149.90 SEK

Total: 354.14
Discount: -37.71
Total after discount: 316.43
VAT: 49.74

Cash: 1000.00
Change: 683.57
----------------------- End receipt -----------------------

Change to give the customer: 683.57
[TOTAL INCOME - VIEW] Total revenue: 316.43 SEK
```

Figure 3: The receipt output of the program

The sale process starts and items are added sequentially, displaying their details including item ID, name, cost, VAT, and description. The total cost including VAT and total VAT are updated and displayed after each item is added.

The program handles exceptions gracefully, such as when the user inputs an invalid item ID (fail or 999). The user is notified with clear messages, while the sale continues without adding invalid items.

At sale completion, the program applies discounts calculated by the implemented discount strategies and displays the discount amount. The program updates the accounting and inventory systems accordingly. A detailed receipt is generated, showing the timestamp, all items with quantities and prices, totals, VAT, payment, and change.

Finally, the total revenue observer displays the updated total revenue, demonstrating the successful integration of the Observer pattern. This output shows how the program handles missing items and database failures gracefully, updates total revenue using observers, and applies discounts through the strategy patterns.

4 Discussion

Exception Handling Evaluation:

The program's exception handling closely follows the nine best practice guidelines outlined in chapter eight of the textbook:

- Checked vs. Unchecked Exceptions: We chose checked exceptions (ItemNotFoundException, DatabaseFailureException) for recoverable conditions that the program is expected to handle, ensuring that error handling is enforced by the compiler.
- Correct Abstraction Level: Exceptions are named precisely after the error conditions they represent, such as ItemNotFoundException for missing items and DatabaseFailureException for simulated database errors.
- Informative Error Information: Each exception includes detailed messages that specify the cause and context of the failure, aiding debugging and user notification.
- Use of java.lang.Exception: Custom exceptions extend Exception, inheriting standard functionality and allowing rich exception handling features.
- State Consistency: The program guarantees that no object state changes occur if an exception is thrown, preserving the integrity of the sale process and inventory data.
- User Notification: When exceptions are caught, the user interface displays clear, informative messages guiding users to correct their input.
- Developer Notification: Errors indicating serious issues (e.g., database failures) are logged to a file for developers to review, enabling postmortem debugging.
- Unit Tests: Dedicated unit tests verify that exceptions are properly thrown and handled, ensuring robustness.

Design Pattern Implementation Review:

- Observer Pattern: The Observer pattern was implemented correctly. The Sale class acts as the observed object, maintaining a list of observers implementing a common interface. Observers are notified of revenue updates by calling a defined update method whenever a sale is completed. This decouples the observers from the controller and allows multiple observers to react independently. Both TotalRevenueView and TotalRevenueFileOutput implement the observer interface consistently, supporting multiple forms of output.
- Chosen GoF Patterns (Strategy and Composite): The Strategy pattern was used to encapsulate discount calculation algorithms, allowing flexible discount strategies to be plugged in without modifying the sale logic. The Composite pattern was used to compose multiple discounts into a single discount entity, supporting combinations of discount strategies transparently. These implementations follow the classical definitions of these patterns, supporting extensibility and code reuse.

Use and Motivation of Design Patterns:

The choice of the Observer pattern for revenue tracking is well motivated: it cleanly separates revenue display logic from core sale logic and supports multiple, independent views of revenue (console and file). This follows the principle of separation of concerns.

The Strategy and Composite patterns were chosen to address the complexity of discount rules, allowing new discount types to be added without modifying existing code, which aligns with the Open/Closed Principle.

Observer and Observed Object Choices:

The Sale class was selected as the observed object because it is the logical entity where revenue changes occur. Observers track the sum of all completed sales. This choice makes sense because it aligns with the domain concept of a sale generating revenue.

Observer Reference and Coupling:

The reference to observers is managed via a registration method on the observed object (Sale), which maintains a list of observers. This design maintains low coupling because the Sale class only depends on the observer interface, not concrete implementations. Observers are decoupled from both the Sale and the controller/view layers.

Data Passed from Observed to Observer and Encapsulation:

Observers receive only the necessary data — the updated total revenue — as a parameter in the update method. This avoids exposing internal details of the Sale class or its internal state, thus preserving encapsulation. Observers do not manipulate or depend on Sale internals beyond the revenue data.