# AHG Tasks Solution

**Object-Oriented Design, IV1350**

Mostafa Faik

2025-06-01

**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 presents the implementation and analysis of three tasks focused on object-oriented design and testing principles in the context of a point-of-sale (POS) system developed in Java. The tasks aim to deepen understanding of fundamental design patterns, the trade-offs between inheritance and composition, and the importance of automated testing for output verification.

Task 1 explores the use of the Template Method design pattern to structure revenue observers within the POS system. The goal is to promote code reuse and flexibility by defining a skeleton of the observer behavior, allowing subclasses to override specific steps without altering the overall structure.

Task 2 compares inheritance and composition as techniques for adapting existing classes from the Java standard library. In line with the discussion in Chapter 9.3 of A First Course in Object-Oriented Development, this task demonstrates how encapsulation can be preserved or broken depending on the chosen adaptation strategy. The practical implementation shows both approaches applied to the same base class, enabling a direct comparison of their implications.

Task 3 focuses on testing System.out printouts, ensuring that all user-facing messages and outputs in the POS system are verified through automated unit tests. This task emphasizes the importance of validating user communication and output correctness, especially in methods where visual feedback plays a critical role, such as in the View class and receipt generation.

# 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.

### Task 1: Implementing the Template Method Pattern

To implement the Template Method pattern in the POS system, we focused on abstracting the common behavior shared by the revenue observers. The original implementation included multiple classes for observing and displaying revenue, such as TotalRevenueFileOutput and TotalRevenueView. These classes contained similar logic for receiving updates, handling errors, and outputting results.

We introduced an abstract superclass named AbstractRevenueObserver, which defined a revenueUpdateTemplate method encapsulating the skeleton of the observer behavior. This method contained a try-catch block for error handling and invoked two abstract

methods: doShowTotalIncome() and handleErrors(Exception e). Each subclass implemented these methods based on its specific behavior writing to a file or printing to the console.

This approach improved modularity, centralized error handling, and reduced code duplication. We chose meaningful method and variable names aligned with the semantics of the system and added parameters to allow flexibility and future extensibility.

### Task 2: Adapting a Java Class with Inheritance and Composition

For this task, we selected the java.util.Random class for adaptation. This choice was motivated by its simplicity and ease of demonstration. The goal was to extend its functionality to include a logging feature that outputs each generated random value for traceability.

We created two classes:

- InheritedRandom extended Random and overrode the nextInt(int bound) method to include a print statement before returning the generated value.
- ComposedRandom held a private Random instance and provided a method nextInt(int bound) that delegated the call to the contained instance, added logging, and returned the value.

The inheritance-based solution tightly coupled the subclass to the superclass, exposing inherited methods that might be irrelevant or unintended. In contrast, the composition-based solution allowed tighter control over the public API and better encapsulation. This demonstrated the principle outlined in Chapter 9.3—prefer composition over inheritance to preserve encapsulation and minimize side effects.

### Task 3: Writing Unit Tests for Output Verification

To test all System.out outputs in the POS system, we used ByteArrayOutputStream to capture printed output and compare it with expected results. We wrote tests for the following components:

- The main method in Startup.
- The View class, ensuring all user-facing messages, including running totals and discount information, were tested.
- The receipt printout, validating the correctness of the printed receipt format and content.
- Tests for all the layers

For each test, we simulated a controlled execution flow using predefined data. Exception scenarios were also tested to ensure appropriate error messages were printed. Each assertion focused on verifying the presence of key phrases or values in the output rather than exact formatting, which allowed for robustness in the tests.

This testing strategy ensured high coverage of user-facing behavior and increased confidence in the system's correctness from a user experience perspective.

# 3 Result

This section presents the outcomes of implementing the three tasks specified in the assignment. Each task resulted in updated or newly created classes that contribute to the robustness, clarity, and testability of the Process Sale application. Alongside source code summaries, this section includes links to the public GitHub repository, representative sample outputs, and an overview of the test coverage and results.

### Git Repository:

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

### Task 1: Implementation of the Template Method Pattern

To eliminate duplicated code and enforce a consistent structure for error handling in observer classes, we applied the Template Method design pattern. This involved creating an abstract base class that defines a final template method and leaves specific steps to subclasses.

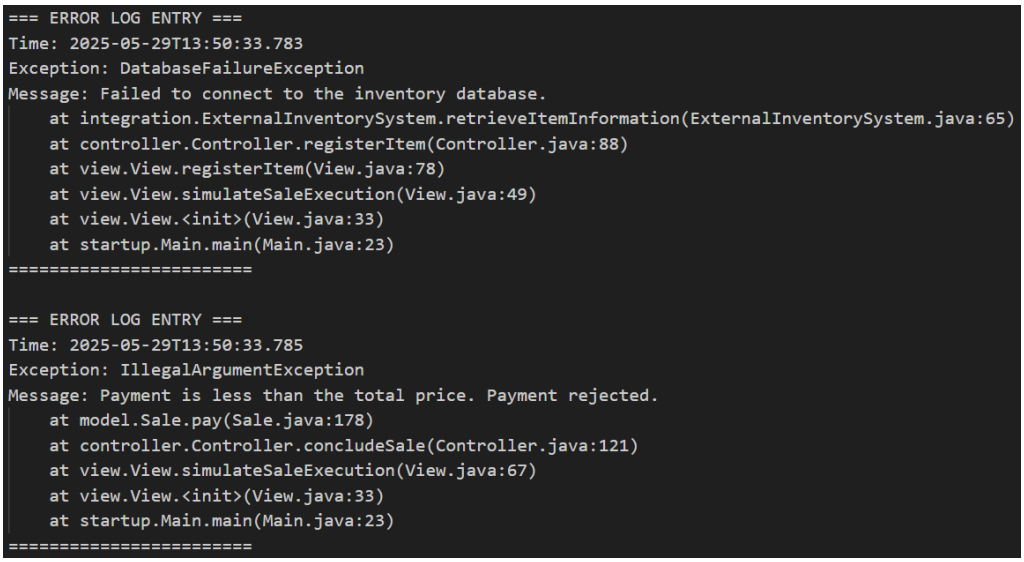
Key Classes:

- AbstractRevenueObserver:
  - Serves as the template superclass.
  - Implements newRevenue(Amount revenue) with a fixed structure. Calls doShowTotalIncome() in a try block. If an exception occurs, it is caught and passed to handleErrors(Exception e) for further processing.
  - Abstract methods:
    doShowTotalIncome(): Implemented by concrete subclasses to handle the revenue update logic.
    handleErrors(Exception e): Allows custom error handling for different observers.

- TotalRevenueFileOutput:
  - Appends revenue updates to a file named total-revenue-log.txt.
  - On failure, logs detailed error entries with timestamp, exception type, and stack trace in error_log.txt.
- TotalRevenueView:
  - Outputs revenue updates directly to System.out for user visibility.
  - Errors are handled by printing concise user-friendly messages to the console.

This structure ensures that error management and output formatting are handled consistently, while allowing each subclass to focus on its specific output mechanism.

Sample Outputs:



```
=== ERROR LOG ENTRY ===
Time: 2025-05-29T13:50:33.783
Exception: DatabaseFailureException
Message: Failed to connect to the inventory database.
    at integration.ExternalInventorySystem.retrieveItemInformation(ExternalInventorySystem.java:65)
    at controller.Controller.registerItem(Controller.java:88)
    at view.View.registerItem(View.java:78)
    at view.View.simulateSaleExecution(View.java:49)
    at view.View.<init>(View.java:33)
    at startup.Main.main(Main.java:23)
========================

=== ERROR LOG ENTRY ===
Time: 2025-05-29T13:50:33.785
Exception: IllegalArgumentException
Message: Payment is less than the total price. Payment rejected.
    at model.Sale.pay(Sale.java:178)
    at controller.Controller.concludeSale(Controller.java:121)
    at view.View.simulateSaleExecution(View.java:67)
    at view.View.<init>(View.java:33)
    at startup.Main.main(Main.java:23)
========================
```

Figure 1: Sample Output Of The Exception Errors File

This log entry is written when a simulated technical issue occurres during item registration. It demonstrates how the template method pattern ensures that exceptions are logged with detailed diagnostics, including the type of exception, message, and stack trace, enabling easier debugging.

```
Added: 316.43 SEK, total revenue so far: 316.43 SEK
Added: 316.43 SEK, total revenue so far: 632.86 SEK
Added: 316.43 SEK, total revenue so far: 949.29 SEK
Added: 316.43 SEK, total revenue so far: 1265.72 SEK
Added: 316.43 SEK, total revenue so far: 1582.15 SEK
Added: 316.43 SEK, total revenue so far: 1898.58 SEK
```

Figure 2: Sample Output Of The Total Revenue File

Figure 2 shows the output that was generated by the TotalRevenueFileOutput class. Each entry shows how much revenue was added during a sale and the cumulative total. It verifies that the observer receives correct updates and persistently logs revenue growth after each transaction.

```
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 3: Sample Output Of The Sale

```
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
[TOTAL INCOME - VIEW] Total revenue: 316.43 SEK
Accounting system updated with payment: 1000.00
Inventory system updated
```

Figure 4: Sample Output Of The Sale

This is the standard output to the console, printed by the View class, showing the details of each item scanned in the sale. It confirms that the program correctly fetches and displays item information and updates the running total and VAT.

The output also shows user-facing feedback for invalid inputs or system failures, such as entering an incorrect item ID or simulating a database failure. The messages are printed by the View class to inform the user of the issue in a clear and friendly way.

```
----------------------- Begin receipt -----------------------
Time of Sale: 2025-05-29 14:40

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
```

Figure 5: Sample Output Of The Receipt

Figure 5 shows the receipt, generated by the Sale class and printed through View, shows the transaction summary including item names, quantities, prices, total cost, VAT, discount, cash paid, and change. It validates the full functionality of the sales process.

### Task 2: Inheritance vs Composition

To demonstrate the differences between inheritance and composition, we adapted the java.util.Random class in two different ways. The source code for this task can be found under src, in the folder inheritanceAndComposition.

Inheritance Approach – InheritedRandom:

- Subclassed Random and overrode methods like nextInt(int bound) and nextDouble() to add logging before returning the values.
- This demonstrates a direct reuse of behavior via extension, but also exposes the internal workings and inherited methods of Random, reducing encapsulation.

Composition Approach – ComposedRandom:

- Used a private instance of Random and provided wrapped methods for number generation.
- This approach maintains tighter encapsulation, allowing the class to expose only a specific interface and behavior.

Sample Output:

```
=== Using InheritedRandom ===
[InheritedRandom] Generated int: 6
[InheritedRandom] Generated double: 0.8880918060664447

=== Using ComposedRandom ===
[ComposedRandom] Generated int: 3
[ComposedRandom] Generated double: 0.289323910824244
```

Figure 6: Sample Output of Inheritance and Composition

This output demonstrates how both the subclass (InheritedRandom) and the composed class (ComposedRandom) generate random numbers while adding logging. It clearly illustrates the difference in adaptation styles: inheritance exposes the full parent interface, while composition wraps and controls it.

### Task 3: Testing Output

In Task 3, the focus was on ensuring that all output to System.out in the Process Sale system is correctly printed and verified through unit testing. While the task specifically required testing output from the main() method, the View class, and receipt generation, this implementation went further by covering all layers involved in producing visible console output, thereby increasing the reliability and test coverage of the application.

The following layers and classes were tested:

- View Layer:
  View class: Verified all user-facing messages such as item registration, totals, and error handling.
- Utils Layer:
  TotalRevenueView: Verified revenue updates are printed to System.out with correct formatting.
- Model Layer:
  Sale: Tested the receipt text generated from the sale and printed to console.
- Controller Layer:
  Controller: Verified that outputs triggered by high-level sale operations are correctly integrated and indirectly reflected in the console.
- Observer Utilities:
  TotalRevenueView, TotalRevenueFileOutput, and LogHandler: Confirmed that printed messages (and file outputs) from observers were triggered as expected during revenue updates and exception logging.

All test cases used ByteArrayOutputStream to redirect System.out, allowing assertions on printed messages. Only printouts containing informative and meaningful content (e.g., totals, revenue updates, item info, error messages) were asserted to avoid noise from structural output.

These scenarios confirmed that user-facing error messages and log entries are properly routed to both the console and file outputs, demonstrating robust output handling under fault conditions.

The thorough output testing revealed that:

- All expected messages were printed correctly and in the intended format.
- Exception scenarios were gracefully handled with both logging and user-friendly console messages.
- Output from multiple layers remained consistent and traceable throughout the program's lifecycle.

As a result, the application's output logic is now highly robust, transparent, and easily verifiable, significantly improving maintainability and user trust in the system's behavior.

# 4 Discussion

### Task 1: Template Method Pattern

The implementation of the Template Method pattern in Task 1 correctly follows the design principle of defining the program skeleton in a superclass while deferring specific steps to subclasses. The abstract class AbstractRevenueObserver encapsulates the invariant parts of revenue observation and logging, such as error handling. Concrete subclasses (TotalRevenueView and TotalRevenueFileOutput) then implement the specific behavior for outputting revenue to the console and writing it to a file, respectively.

This implementation is a textbook example of the Template Method pattern because:

- The abstract superclass defines a final method that cannot be overridden.
- The method includes a try-catch block structure, ensuring consistent error handling across subclasses.
- The doShowTotalIncome() method is declared abstract, enforcing that each subclass must provide its own specific behavior.

Using the Template Method pattern in this application brings several benefits:

- Code reuse: The common logic for updating revenue (including error handling and method structure) is centralized, reducing duplication across observers.
- Consistent behavior: All observers follow the same invocation pattern and exception management, making the system more predictable and easier to debug.
- Separation of concerns: Core algorithm logic is decoupled from output formatting or logging details.

The Template Method pattern not only fits well within this system architecture, but also provides strong architectural advantages by promoting maintainability and reducing the likelihood of subtle bugs caused by inconsistent behavior among observers.

### Task 2: Inheritance vs. Composition

In this task, two different adaptation strategies were applied to the java.util.Random class: one using inheritance (InheritedRandom) and one using composition (ComposedRandom). Both classes added new printout functionality while preserving the core behavior of random number generation.

Table 1: Comparison of Inheritance and Composition

| **Criteria** | **Inheritance** | **Composition** |
|---|---|---|
| Encapsulation | Breaks encapsulation by exposing internal structure of the superclass. | Preserves encapsulation by hiding the internal implementation. |
| Flexibility | Less flexible, restricted to a single superclass. | More flexible; allows combining multiple objects. |
| Reusability | Tightly coupled, limiting reuse in different contexts. | More reusable due to loose coupling. |
| Maintenance | Fragile to changes in the superclass. | More robust and easier to maintain. |
| Testability | Harder to mock or isolate behavior. | Easier to mock and test the composed object. |
| Extensibility | Adding new behavior often requires modifying superclass. | Behavior can be extended without modifying original classes. |

While inheritance offers a quick and convenient way to extend functionality, it sacrifices long-term maintainability and flexibility. Composition is generally the preferred approach, especially in large or evolving codebases. It leads to more modular, testable, and robust systems by promoting encapsulation and loose coupling.

### Task 3: Evaluation of Unit Tests

The unit tests written for this project were designed following industry-standard testing practices, particularly with respect to testing outputs and system behavior across layers.

All unit tests in the project were written using the JUnit 5 testing framework, ensuring a standardized and structured approach to testing. Each test is self-evaluating, meaning that it either runs silently if it passes or outputs a detailed message upon failure. This approach improves clarity and simplifies debugging.

For each class tested, a corresponding test class was created, ensuring a one-to-one mapping between test classes and SUT classes. All important control flows, including conditional branches, were thoroughly tested using various inputs and scenarios to verify behavior across both normal and edge cases. Parameter values passed to methods during testing were chosen carefully to include representative, boundary, and invalid values to ensure comprehensive coverage.

Throughout the test design and implementation, programming best practices were followed, such as meaningful method names, isolation of test cases, use of mock or stub classes where appropriate, and assertive validation using assertEquals, assertThrows, and other assertion methods. This comprehensive testing process not only increased

confidence in the correctness of the system but also improved its maintainability and robustness.

Tests are not limited to the View and receipt printout only. The controller, model, observers, startup, and utility layers are also tested to ensure complete end-to-end validation of System.out output behavior.

All printouts with informative value to the user were asserted, improving the confidence that the application communicates correctly and clearly.

The unit tests provide high confidence in the correctness of the output functionality and exhibit strong adherence to test design best practices. They are complete, precise, and modular, which supports continuous integration and future maintenance.