Group Project Phase 3 Refactoring Report

Authors:
Tristan Phay, Sovann Phay, Zach Atwood, O'Shae Berteaux

Instructor:
Timothy Carlson

Course:
ICS 372-01

Date:
December 8th, 2025

Summary

This report provides a comprehensive breakdown of our project's state at the end of Phase 2 and details the refactoring process undertaken to improve structure, reduce code duplication, and enhance the overall system architecture.

Original State

By the end of Phase 2, our Java version of the Order Tracking System had grown to 7 distinct classes. The OrderTrackingController alone contained over 600 lines of code and was overloaded with multiple responsibilities, which quickly cluttered the class.

As we moved into the next phase and began incorporating Kotlin along with new features, the controller expanded even further. Before adding Phase 3 functionality, our priority was to verify that all existing methods worked correctly and as intended. Once that baseline was stable, we introduced new implementations and confirmed they were functioning properly.

Only then did we begin the refactoring phase, where our goal was to apply design patterns and restructure the system to improve coupling and cohesion by breaking down the oversized OrderTrackingController.

Refactoring Process

1. New Classes

The following classes were added during the refactoring process:

- **OrderManager**

- **OrderTableHelper**
- **OrderFormatters**
- **OrderDialogHelper**

These classes were added because the OrderTrackingController had too much repeated code and was handling responsibilities outside its scope. By moving formatting, dialog handling, and order logic into their own classes, we spread functionality across the system instead of relying on the controller for everything. This reduced duplication, lowered coupling, and made the controller focus only on coordinating UI inputs and display.

---

2. OrderManager

The OrderManager class centralizes all order functions and state transitions into one place instead of spreading that logic across the OrderTrackingController. It acts like a façade between the controller and the persistence layer, which reduces coupling and keeps the controller focused on UI coordination. This lines up with the Single Responsibility Principle since the manager only handles how the order logic actually works.

Inside this class, we also used a strategy pattern because the single- and multi-selection functionality was repeated across different order functions. Even though delete was more complex than start or undo, we were able to simplify everything by using the executeOrderTransition() helper method. This eliminated duplicate logic and cut down lines in the controller, making the code cleaner and easier to maintain.

---

3. OrderTableHelper

The OrderTableHelper class handles everything needed to build and configure the order table in the UI. By moving this setup into its own class, the controller doesn't have to worry about both managing logic and setting up the table. It now just focuses on handling inputs and displaying data. If we ever need to add new functionality or change how the table works, we can extend the helper instead of touching the controller. This follows the Open-Closed Principle since the helper is open to extension but closed to modification.

---

4. OrderFormatters

The OrderFormatters class consolidates all formatting functions that previously existed within the controller into a single, dedicated location. Its responsibility is straightforward: handle formatting for dates, currency, and order display information.

This refactoring eliminated duplicate logic that existed in both the table helper (such as formatting dates and total costs) and reduced unnecessary code in the controller. Now, any class that requires formatting can simply call OrderFormatters instead of re-implementing the same code. This class adheres to the **Single Responsibility Principle** by exclusively handling formatting concerns and follows the **DRY (Don't Repeat Yourself)** principle by eliminating redundant methods across multiple locations.

---

5. OrderDialogHelper

The OrderDialogHelper class manages all dialog-related functionality for the UI, preventing the controller from becoming cluttered with repeated dialog code. Many functions required similar dialog setups, so instead of rewriting the same configuration repeatedly, we built this helper with reusable methods capable of handling dialogs for any function.

This keeps the controller focused on managing inputs and display while the helper handles all popup interactions. The design aligns with the **Single Responsibility Principle** since the class is solely responsible for dialog management.

Results and Conclusion

Metrics

Metric	**Before Refactoring**	**After Refactoring**	**Improvement**
Controller Lines of Code	700-800	~400	50% reduction
Number of Classes	7	12	Better separation of concerns
Code Duplication	High	Minimal	Significant reduction

Summary

After converting the project to Kotlin and adding new functionality, the OrderTrackingController grew to around 700–800 lines of code. Through our refactoring efforts and the introduction of new helper classes, we were able to reduce the controller to about 400 lines. This not only cut down the size of the class but also improved readability, reduced duplication, and made the controller more focused on its core responsibility of coordinating UI inputs and display.

This refactoring achieved the following benefits:

- **Reduced Class Size:** Cut the controller by 50%
- **Improved Readability:** Code is now more organized and easier to understand
- **Reduced Duplication:** Eliminated repeated code across the system
- **Enhanced Focus:** The controller now concentrates on its core responsibility of coordinating UI inputs and display
- **Better Maintainability:** Changes can be made to specific components without affecting the entire system
- **Applied Design Patterns:** Successfully implemented Strategy, Façade, Single Responsibility, Open-Closed, and DRY principles

The refactoring process has positioned our Order Tracking System for easier future enhancements and maintenance while maintaining all existing functionality.