TP – Design Patterns (Part 2)

Report

Prepared by: **Youssef GUERAIRATE**

Date: November 23, 2025

Contents

1. Exercise 1 — Strategy Pattern: Flexible Navigation

1.1 Description of the Problem

This exercise consists in building a flexible navigation system capable of computing routes using different strategies: walking, car, and optionally bike. The design pattern suitable for this problem is the **Strategy Design Pattern** to make route computation interchangeable at runtime depending on user choice.

1.2 Questions

1. Role of the Navigator Class

The `Navigator` class acts as the **Context** in the Strategy pattern. It contains a reference to a `RouteStrategy` and delegates route computation to the currently selected strategy.

2. Why Navigator Depends on RouteStrategy?

`Navigator` depends on the `RouteStrategy` interface to:

- respect the principle of programming to an interface and not to an implementation,
- allow strategies to be swapped dynamically at runtime,
- decouple the route calculation algorithm from the context.

3. SOLID Principles Applied

- **S — Single Responsibility**: each strategy computes a route in one specific way.
- **O — Open/Closed**: adding new strategies does not modify existing code.
- **D — Dependency Inversion**: `Navigator` depends on an abstraction instead of concrete implementations.

1.3 Class Diagram

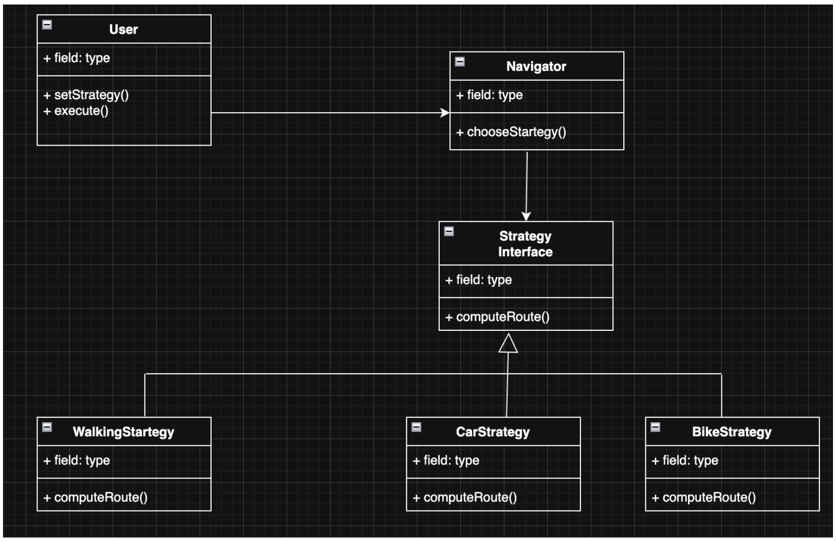

Figure 1.1: Composite Pattern for Company Maintenance Cost

2. Exercise 2 — Composite Pattern: Vehicle Maintenance

2.1 Chosen Design Pattern

The appropriate pattern is the **Composite Design Pattern**. It allows treating independent companies and parent companies uniformly by defining a common interface for computing maintenance cost which can be observed from the sentence : ” Parent Companies to which we can add independent companies ”.

2.2 Class Diagram

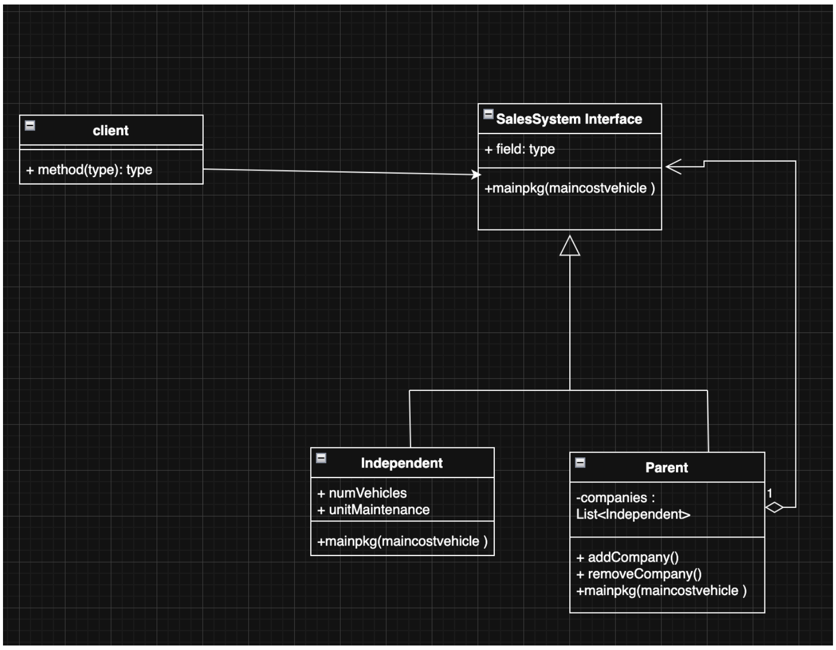

Figure 2.1: Composite Pattern for Company Maintenance Cost

3. Exercise 3 — Adapter Pattern: Payment Processor Integration

3.1 Chosen Design Pattern

The correct design pattern is the **Adapter Pattern**, which allows integrating incompatible payment services (`QuickPay`, `SafeTransfer`) with the standard `PaymentProcessor` interface. Which can be observed from the phrase " You want the user to be able to use the new PaymentProcessor interface without ..." , as the APIs are external , closed-source , we can't change their implementation. What we can do is use an Adapter Design Pattern.

3.2 Participants

- **Target**: `PaymentProcessor` interface
- **Adaptees**: `QuickPay`, `SafeTransfer`
- **Adapters**: classes that adapt each third-party API
- **Client**: e-commerce system

3.3 Class Diagram

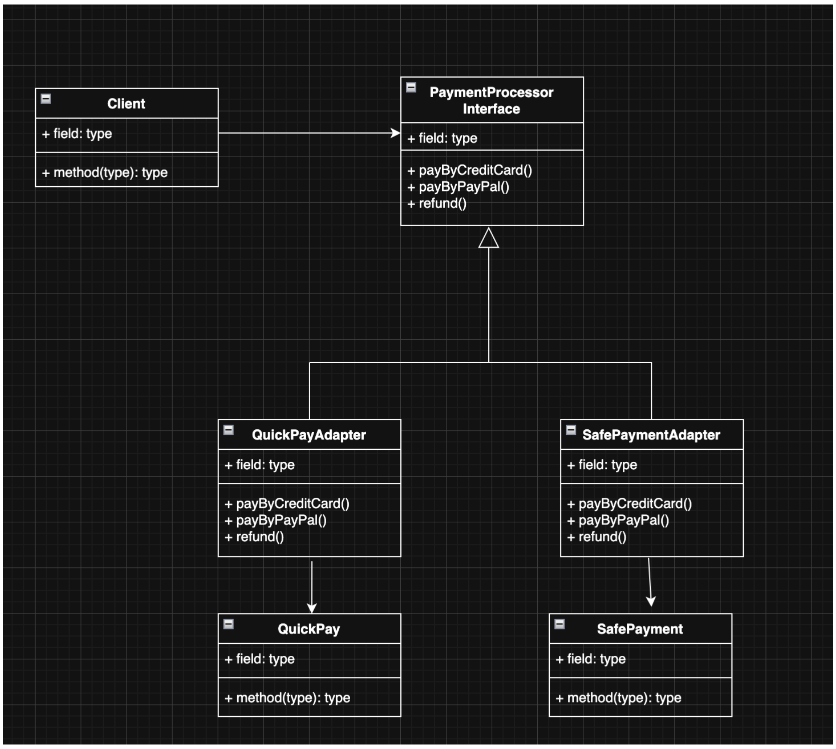

Figure 3.1: Adapter Pattern for Payment Services Integration

4. Exercise 4 — Observer Pattern: GUI Dashboard Notifications

4.1 Chosen Design Pattern

The correct design pattern is the **Observer Pattern**. It ensures that multiple observers (Logger, LabelUpdater, NotificationSender) are notified whenever a GUI element (Button, Slider) changes state.

4.2 Class Diagram

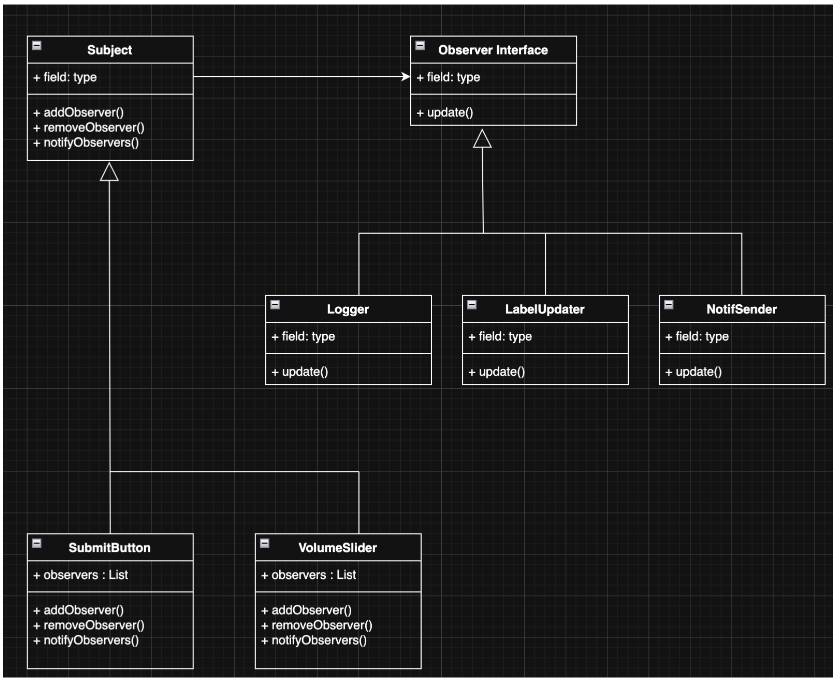

Figure 4.1: Observer Pattern for GUI Element Notifications