Pattern in action Prototype Report

Class: CS 4213 - Software Design Patterns
Semester: Fall 2025
Applied Pattern: Strategy Pattern

Name: Colby Frison, Emily Locklear, Grant Parkman, James Totah, Antonio Natusch Zarco

Group: H

1 Introduction & Problem Analysis

The goal of this assignment was to refactor the legacy `ModelManager` class in the SmartWrite application. The original implementation suffered from significant architectural flaws typical of monolithic design.

1.1 The "Before" State: Monolithic Design

In the original codebase, the `ModelManager` class was a "God Object". It handled:

- State management (which model is active).
- Error tracking (counting failures).
- Fallback logic (deciding which model to try next).
- **Specific implementation details** for every supported AI model (GPT-4, Claude, Gemini, etc.).

This design relied heavily on large `switch` statements to dispatch requests. Adding a new model required modifying the core `ModelManager` class, directly violating the **Open/Closed Principle**.

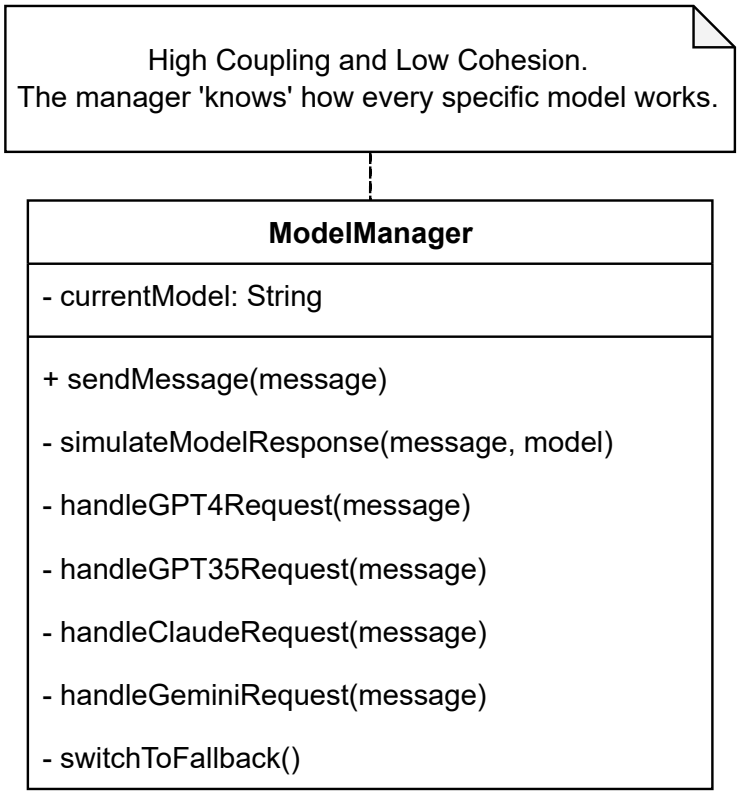

Figure 1: Legacy ModelManager: Monolithic architecture with tightly coupled model logic and decision flow.

2 The Solution: Strategy Pattern

To resolve these issues, we applied the **Strategy Pattern**. This pattern allows us to define a family of algorithms (AI model implementations), encapsulate each one, and make them interchangeable.

2.1 The "After" State: Decoupled Architecture

The refactored architecture separates the *management* of models from the *execution* of models.

- **Context (`ModelManager`)**: Now purely responsible for configuration, error tracking, and selecting the active strategy. It treats all models uniformly via the strategy interface.
- **Strategy Interface (`ModelStrategy`)**: Defines the contract (`sendMessage`) that all models must adhere to.
- **Concrete Strategies**: Individual classes (`GPT4Strategy`, `GPT35Strategy`, etc.) that contain the specific logic for each API.

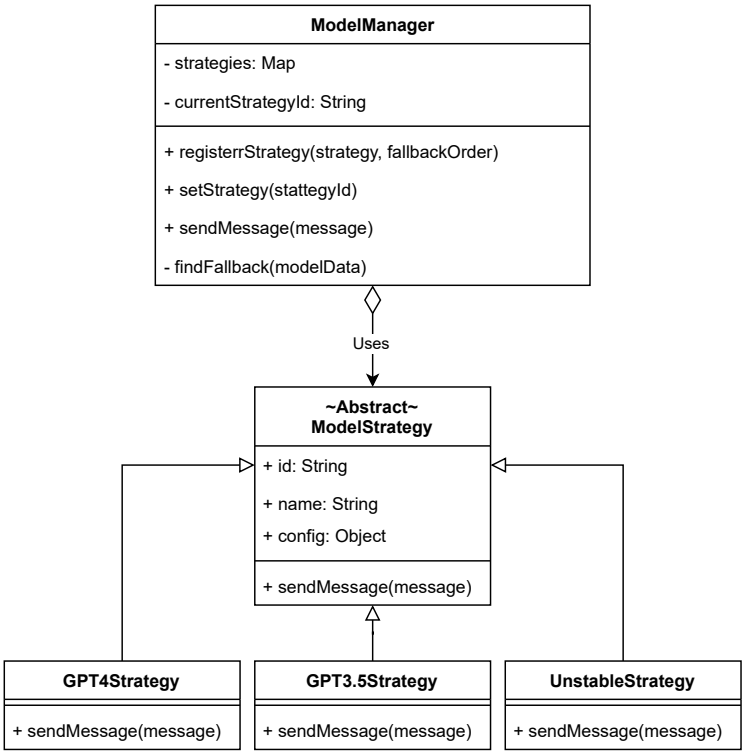

Figure 2: Refactored ModelManager: Decoupled architecture with strategy interface and concrete strategies.

3 Key Resolves & Benefits

The refactor achieved the following improvements:

1. **Adherence to Open/Closed Principle**: We can now add new AI models (e.g., "Claude 3.5") by creating a new file `ClaudeStrategy.js` and registering it. We do **not** need to touch the `ModelManager` logic.
2. **Enhanced Testability**: We demonstrated this by creating an `UnstableStrategy` specifically for testing. This strategy is designed to fail, allowing us to verify the fallback logic without relying on unstable external APIs or modifying production code.
3. **Separation of Concerns**: The `ModelManager` focuses on *reliability* (retries, fallbacks), while the Strategies focus on *connectivity* (API calls).