Assignment 4: Pattern for Maintenance

M11-Discuss Patterns for Maintenance

CS4213 Design Patterns

Semester: Fall 2025
Name: Colby Frison
OUID: XXXXXXXXXX

Due: 12/5/2025

Contents

1 Task A: Select a Design to Refactor

1.1 1. System Overview: SmartWrite

SmartWrite is an electron-based application I designed and implemented with a group last semester with the purpose to digitize and organize notes. It combines a local backend (Electron Main Process) for system access with a web-based frontend (Renderer Process) for the user interface.
Purpose: To provide a seamless interface for viewing PDFs, editing Markdown notes, and interacting with an AI assistant for content analysis, all within a unified local environment.

1.2 2. Current Design Flaws

Although the application functions correctly, I identified several design choices in my original implementation that hinder maintenance and scalability:

1. **High Coupling (Flexibility Issue):**
   - I designed the `FileTree` module to handle file selection logic, but it is tightly coupled to the `PDF` module. When a file is clicked, `FileTree` directly invokes `PDF.loadDocument()`. This makes it difficult for me to add new file viewers (e.g., for images or pure text) without modifying the `FileTree` class.
   - Modules often import each other directly rather than communicating through abstractions.
2. **God Object / Orchestration Complexity (Reuse Issue):**
   - I also identified a "God Object" problem. The `FrontMain` (Frontend `main.js`) acts as a central orchestrator that manually initializes every module (`Chat`, `PDF`, `Settings`, etc.). This makes `FrontMain` a class that knows too much about the specific implementation of every feature, making it hard for me to reuse individual modules in other contexts without dragging in the entire initialization logic.
3. **Testability Issues:**
   - **Global State dependencies:** In my review of the code, I found that most modules (like `Chat` and `Theme`) directly import a singleton `State` object. This global dependency makes unit testing difficult because the state persists between tests unless manually reset, and mocking the state requires complex module interception.
   - **DOM Coupling:** Business logic (e.g., parsing PDF text) is intertwined with UI manipulation (e.g., updating the DOM), making it impossible for me to test logic without a browser environment.

1.3 3. Original UML Class Diagram

The current design relies on direct relationships and a central orchestrator.

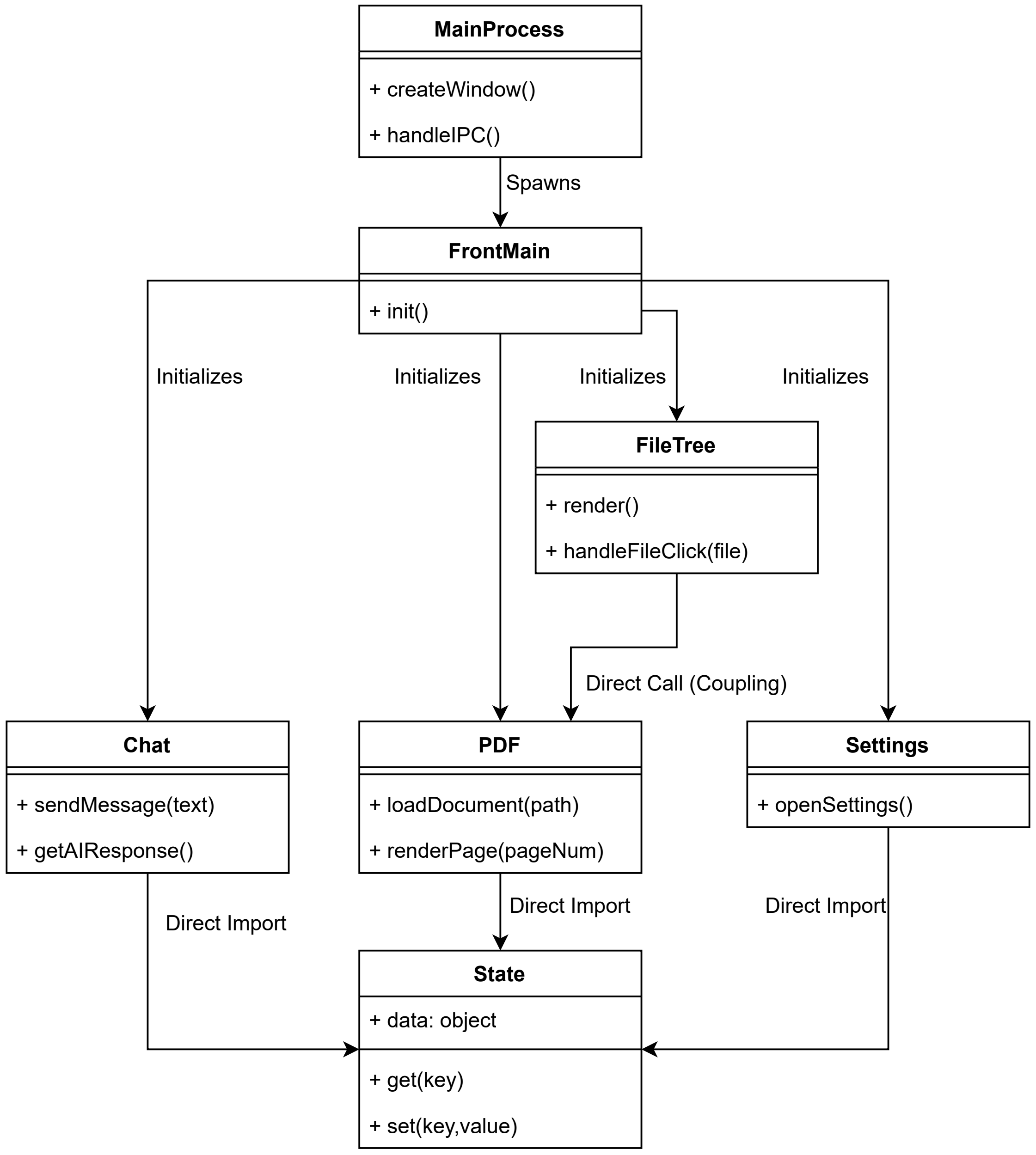

Figure 1: Original UML Class Diagram

2 Task B: Refactor for Flexibility and Reuse

2.1 Refactoring 1: Mediator Pattern (Flexibility)

Goal: Decouple the `FileTree` from specific file viewers like `PDF`.
Description:
Currently, `FileTree` explicitly calls `PDF.loadDocument()` when a user selects a file. To decouple these components, I chose to implement the **AppMediator**.

1. In this new design, the `FileTree` no longer imports `PDF`. Instead, it will notify the Mediator (e.g., `mediator.notify('fileSelected', file)`) when a user interacts with the UI.
2. The `AppMediator` holds references to `PDF`, `FileTree`, and other potential modules (like a future `MarkdownEditor`).
3. When the Mediator receives the `fileSelected` event, it checks the file extension and routes the request to the appropriate module (e.g., calling `PDF.loadDocument()` only if the file is a PDF).

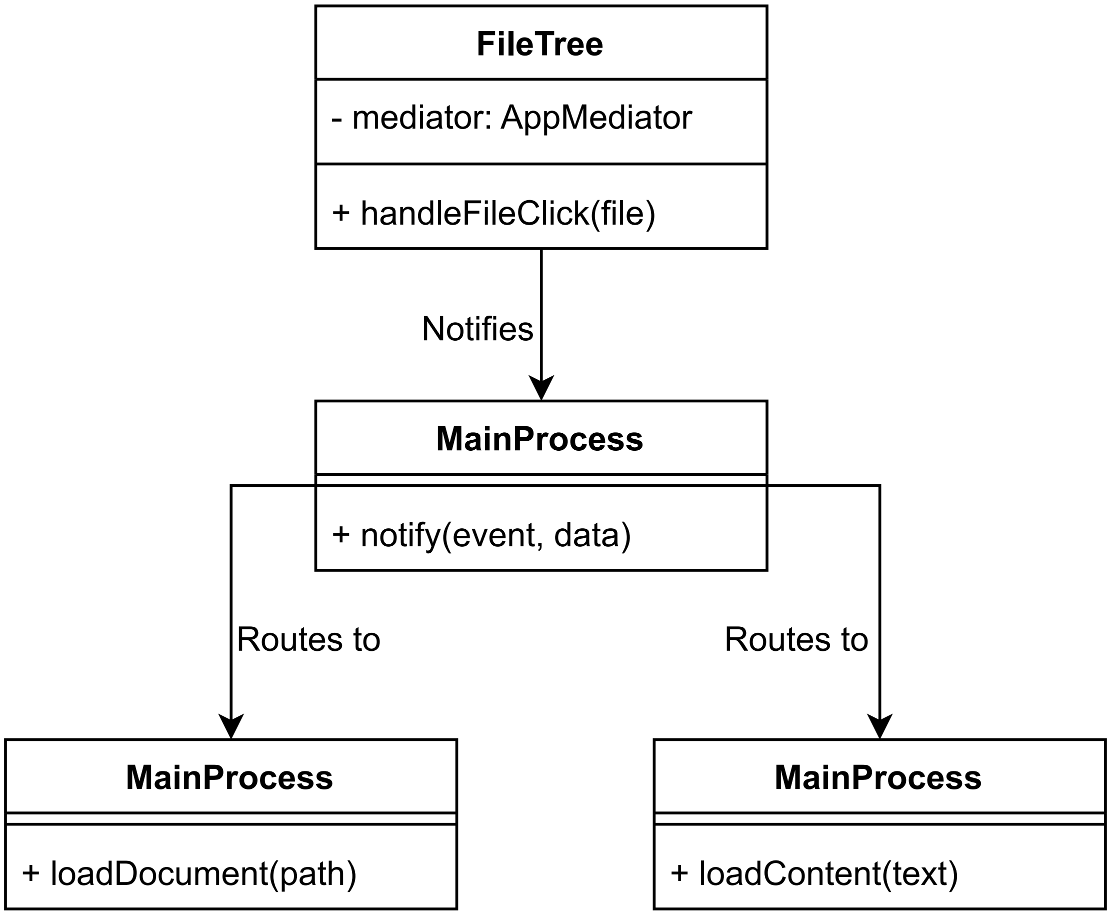

Figure 2: Updated UML: Mediator Pattern

2.2 Refactoring 2: Facade Pattern (Reuse)

Goal: Consolidate native system interactions to promote code reuse and simplify the frontend.
Description:
My application relies heavily on Electron's IPC (Inter-Process Communication) via `window.electron` exposed by the preload script. Currently, raw IPC calls like `ipcRenderer.invoke('read-file', path)` are scattered across `PDF.js`, `FileTree.js`, and `Chat.js`. To address this, I designed a **NativeBridgeFacade**.

1. This Facade provides a clean, simplified interface for all system operations: `readFile(path)`, `savePreferences(data)`, `fetchAIResponse(prompt)`.
2. The individual modules will depend on `NativeBridgeFacade` instead of the global Electron object.
3. This improves reuse: if I port the app to the web (removing Electron), I only need to replace `NativeBridgeFacade` with a `WebBridgeFacade`, without changing the business logic in `Chat` or `PDF`.

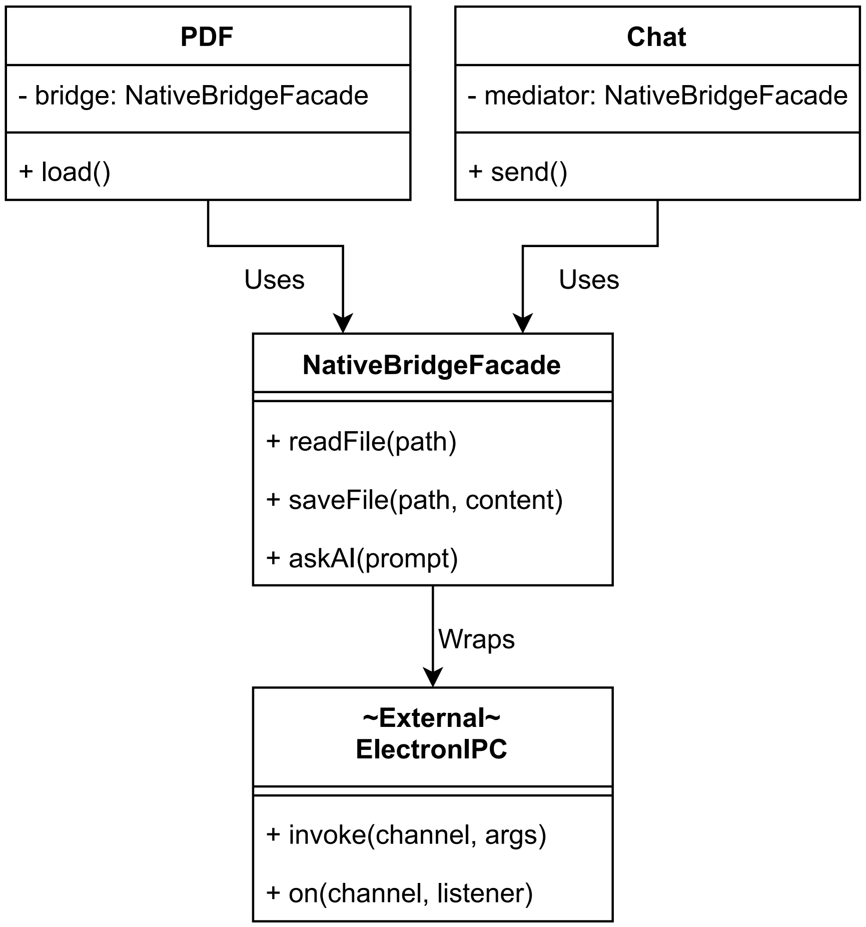

Figure 3: Updated UML: Facade Pattern

2.3 Refactoring 3: Strategy Pattern (Design Needs)

Goal: Allow runtime switching between different AI models (e.g., OpenAI vs. Local LLM).
Description:
Currently, I hardcoded the interaction with a specific AI provider in the `Chat` module. I decided to extract this logic into an **AIModelStrategy** interface.

1. I define an interface `AIModelStrategy` with a method `generateResponse(prompt)`.
2. I create concrete implementations: `OpenAIStrategy` (calls external API) and `LocalLLMStrategy` (calls local Python process or Ollama).
3. The `Chat` module will hold a reference to the current strategy and delegate the generation call. This allows the user to switch providers in Settings without rewriting `Chat.js`.

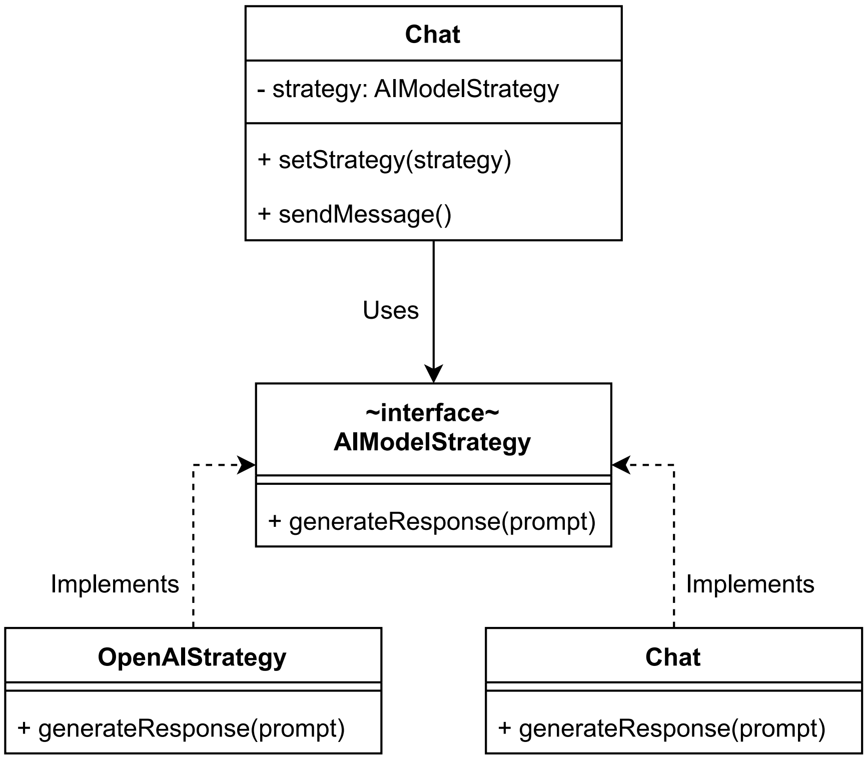

Figure 4: Updated UML: Strategy Pattern

3 Task C: Refactor for Testability + Tools

3.1 1. Refactoring for Testability

Refactoring A: Dependency Injection (DI)
Currently, modules like `Chat` import the global `State` singleton directly. This makes testing hard because state persists across tests.

- **Change:** I modified `Chat` and `PDF` constructors to accept dependencies (`state`, `bridge`) as arguments.
- **Benefit:** In unit tests, I can now inject a `MockState` or `MockBridge`. This ensures that a test for `Chat` doesn't accidentally fail because of unrelated data left over in the real `State` object.

Refactoring B: Logic Isolation (Humble Object Pattern)
The `PDF` module mixes complex logic (text extraction, search algorithms) with DOM manipulation (rendering to Canvas).

- **Change:** I extracted the pure logic into a `PDFProcessor` class (handles parsing, text analysis) and kept the UI code in `PDFViewer`.
- **Benefit:** This allows me to write fast, headless unit tests for `PDFProcessor` using Jest/Node.js without needing a browser environment or complex DOM mocks.

3.2 2. Tool Research: VS Code Built-in Refactoring Tools

Key Features:

- **Rename Symbol (F2):** Automatically renames a variable, function, or class across the entire workspace. It uses the language server's Abstract Syntax Tree (AST) understanding to ensure it only updates the correct references, avoiding the dangers of a global "find and replace."
- **Extract Function/Method:** Allows me to select a block of code and automatically extract it into a new function. The tool handles argument passing and return values, ensuring the logic remains identical while reducing method size and complexity.
- **Move to File:** Helps in breaking down "God Classes" by safely moving classes or functions to new files and automatically updating all import statements across the project to point to the new location.

Safe Refactoring with VS Code:
These tools ensure safety by performing **Automated AST Manipulation** rather than manual text editing.

1. When decoupling the `FileTree`, I can use "Extract Method" to pull the file routing logic out of the click handler before moving it to the `AppMediator`.
2. Using "Rename Symbol" ensures that when I change internal class property names during the Facade implementation, I don't accidentally break obscure references in other files.
3. The "Find All References" feature allows me to confidently verify that no legacy code is still calling the old direct dependencies after I've introduced the Dependency Injection pattern.

4 Task D: Reflection on Refactoring Impact

Summary:
All of the refactorings on the code base had a significant impact on the system's functionality and maintainability. However, the **Mediator Pattern** had the **greatest impact** on the system's flexibility and reusability.

Prior to this change, the `FileTree` had to "know" about every possible file viewer in the system. If I wanted to add an Image Viewer, I would have to modify the `FileTree` code to say `if (file.ext == 'png') ImageViewer.load()`. This violation of the Open/Closed Principle meant that adding features required modifying existing, stable code, risking bugs in the file selection logic.

By introducing the `AppMediator`, the `FileTree` became purely a "notifier". It says "someone clicked a file" and stops caring. The Mediator handles the complexity. This shifted the system from a brittle mesh of point-to-point connections to a clean, centralized event handling architecture.

This experience changed my view of design improvement from "making code cleaner" to "making code **changeable**". The value wasn't just in the lines of code saved, but in the hours of future debugging saved by decoupling components.