Advice

Software Quality Improvements

Group 5

Cristian Trifan,

Mihail Josan

NHL Stenden University of Applied Sciences

Table of Contents

1. Introduction

This document provides recommendations for improving the quality, maintainability, and extensibility of the JabberPoint system based on the system analysis conducted. The improvements target software design, architecture, and best practices to enhance code readability, usability, and future scalability.

2. Architectural and Design Improvements

2.1 Modernizing the UI Framework

JabberPoint currently relies on Java AWT and Swing, which are outdated technologies for building graphical applications. Migrating the user interface to JavaFX or a web-based UI framework (e.g., React, Vue.js) will improve interactivity and responsiveness.

2.2 Enhancing MVC Implementation

Refactor the system into a more robust MVC architecture with clearer separation between Model, View, and Controller. This will improve modularity and maintainability.

3. Applying Design Patterns to Improve JabberPoint

1. Factory Method Pattern (for SlideItem Creation)

Why?

The SlideItem hierarchy consists of SlideItem, TextItem, and BitmapItem.

Problem: Object creation is currently tightly coupled with XMLAccessor and DemoPresentation, making it hard to extend with new types.

Solution: Implement a Factory Method Pattern to centralize object creation.

How?

Create a SlideItemFactory class with a createSlideItem(type, data) method.

XMLAccessor and DemoPresentation will call this factory instead of directly instantiating objects.

Why not another pattern?

Abstract Factory is unnecessary because there is no need for multiple families of objects.

Builder Pattern is overkill as objects are not complex.

2. Observer Pattern (for MVC Update Mechanism)

Why?

SlideViewerComponent must be notified whenever the Presentation changes.

Problem: Currently, Presentation directly updates SlideViewerComponent, leading to tight coupling.

Solution: Implement the Observer Pattern, allowing multiple views (e.g., GUI, console output) to observe Presentation.

How?

Presentation becomes a Subject, and SlideViewerComponent is an Observer.

When Presentation changes, all observers are notified.

Why not another pattern?

Mediator Pattern is unnecessary since we only need a simple update mechanism.

Publisher-Subscriber is similar, but Observer is better suited for UI updates.

3. Strategy Pattern (for XML Parsing Optimization)

Why?

The current XML parsing uses DOM, which is inefficient for large files.

Problem: The parsing logic is hardcoded inside XMLAccessor.

Solution: Implement Strategy Pattern so different parsing strategies (DOM, SAX, StAX) can be used.

How?

Create a ParsingStrategy interface with a parse(String filePath) method.

Implement DOMParserStrategy, SAXParserStrategy, and StAXParserStrategy.

XMLAccessor chooses the best strategy at runtime based on file size.

Why not another pattern?

Adapter Pattern is not suitable because we need to replace the parsing logic, not just wrap an existing API.

Decorator Pattern is unnecessary since we are not adding behaviors.

Final Thoughts

These patterns were chosen because:

- Factory Method reduces object creation complexity for SlideItem.
- Observer enhances the MVC architecture, ensuring decoupled updates.
- Strategy improves performance and flexibility in XML parsing.â

4. Quality Improvements

4.1 Standardizing Comments and Documentation

Implement Javadoc for all classes and methods to ensure consistent and structured documentation.

4.2 Implementing Proper Access Modifiers

Ensure proper encapsulation by applying correct access modifiers (private, protected, public).

4.3 Improving Code Readability

Use meaningful parameter names, consistent indentation, and remove redundant code.

4.4 Handling Styles Dynamically

Move hardcoded styles to external configuration files (XML, JSON) for flexibility.

5. Performance and Scalability Improvements

5.1 Optimizing XML Parsing

Replace inefficient DOM parsing with SAX or StAX for better performance and memory efficiency.

5.2 Refactoring File I/O Operations

Separate data persistence from business logic using a dedicated File I/O Manager.

6. Error Handling and Testing Improvements

6.1 Strengthening Error Handling Mechanisms

Implement structured logging using `java.util.logging` or Log4j and provide meaningful error messages.

6.2 Implementing Unit Tests

Use JUnit for testing core functionalities such as file I/O, slide rendering, and user interactions.

7. Summary of Key Recommendations

Area	Current Issue	Recommended Improvement	Benefit
UI Framework	Uses outdated Swing & AWT	Migrate to JavaFX or web-based UI	Modern look, better usability
MVC Architecture	Mixed responsibilities	Improve separation of concerns	Better maintainability
Code Documentation	Inconsistent comments	Standardize Javadoc usage	Easier understanding
Access Modifiers	Missing in some places	Apply correct modifiers	Better encapsulation
Readability	Vague names, inconsistent formatting	Improve naming, apply formatting rules	Easier maintenance
Hardcoded Styles	Styles defined in code	Move to external config	More flexibility
XML Parsing	Uses inefficient DOM	Switch to SAX/StAX	Faster performance
File I/O Handling	Mixed with logic	Create dedicated File I/O Manager	Better structure
Error Handling	Minimal exception handling	Implement structured logging	Easier debugging
Unit Testing	No automated tests	Implement JUnit tests	Higher reliability

8. Conclusion

By implementing the above improvements, JabberPoint can enhance its software quality, usability, and maintainability. These enhancements will make the system more adaptable for future changes and improve the overall development experience.