## **Design Patterns Advice for JabberPoint**

## **Observer Pattern:**

JabberPoint is a Java-based presentation software that currently lacks a proper mechanism to automatically update the user interface when the presentation state changes. This leads to tightly coupled code and an inefficient update process. To address this issue, we propose integrating the Observer Pattern, which will decouple the presentation logic from the user interface, ensuring that UI elements automatically reflect changes in the presentation.

### **Justification for Using the Observer Pattern**

The Observer Pattern is a behavioral design pattern that establishes a one-to-many dependency between objects, ensuring that when one object (the subject) changes state, all its dependents (observers) are notified and updated accordingly. This pattern is highly applicable to Jabber Point for the following reasons:

1. **Separation of Concerns:** The current implementation tightly couples the Presentation class with the UI components. Implementing the Observer Pattern will ensure a clear distinction between the presentation model and the view, following the Model-View-Controller (MVC) paradigm.
2. **Automatic UI Updates:** Currently, whenever the presentation changes, UI elements must be manually updated. Using the Observer Pattern will eliminate the need for direct updates, reducing errors and improving maintainability.
3. **Scalability and Flexibility:** Implementing this pattern will make it easier to introduce new UI components, such as different slide views (e.g., slide sorter, next slide preview), without modifying the Presentation class.

### **Implementation Plan**

To integrate the Observer Pattern into JabberPoint, we will define a subject (Presentation) and its observers (UI components that rely on presentation state changes). The following steps outline the implementation:

1. **Modify the Presentation Class**
   1. The Presentation class will act as the **subject** that maintains a list of observers.
   2. It will provide methods to **register, remove, and notify** observers.
2. **Define an Observer Interface**
   1. This interface will be implemented by UI components that need to be updated when the presentation changes.
3. **Modify the SlideViewerComponent to Observe Changes**
   1. The SlideViewerComponent (responsible for rendering slides) will implement the observer interface and update itself when notified by Presentation.
4. **Updating Other UI Components**
   1. The MenuController and KeyController can also be modified to act as observers if needed.

**Observer Pattern - Step-by-Step Process**

1. **Observers Register with the Presentation**  
    At the start of the program, UI components such as SlideViewerComponent, MenuController, and KeyController register as observers of the Presentation class. This establishes a dependency where these components will receive updates whenever the presentation state changes.
2. **A Slide Change Occurs**  
    When the user navigates to a different slide, the Presentation class updates the current slide. Instead of directly modifying UI components, it invokes notifyObservers(), ensuring that all registered observers are informed of the change.
3. **Observers Respond to the Update**  
    Each observer automatically receives the update and adjusts accordingly:
   1. SlideViewerComponent updates the displayed slide.
   2. MenuController updates the displayed slide number in the menu.
   3. KeyController adjusts keyboard navigation if necessary.

This structured approach ensures loose coupling, automatic UI updates, and improved maintainability within JabberPoint.

### **Expected Benefits**

By implementing the Observer Pattern, the following improvements will be achieved:

* **Decoupling UI from Presentation Logic:** UI components will no longer directly depend on Presentation, making the code more maintainable.
* **Automatic Synchronization:** When a slide changes, all registered observers will be updated immediately without additional method calls.
* **Improved Extendability:** Future enhancements, such as additional slide views, can be integrated with minimal changes.

### **Conclusion**

The integration of the Observer Pattern into JabberPoint will significantly enhance the software’s maintainability, scalability, and responsiveness. This pattern will establish a robust event-driven system where UI components dynamically respond to presentation changes without direct dependencies. Implementing this solution aligns with best practices in software design and ensures that JabberPoint remains adaptable to future improvements.

## **Factory Method Pattern:**

JabberPoint currently creates SlideItem objects, such as TextItem and BitmapItem, through direct instantiation. This approach results in a tightly coupled codebase where changes to the creation process require modifying multiple parts of the system. Additionally, adding new types of slide items, such as a VideoItem or CodeSnippetItem, would require significant modifications. To address this issue, the Factory Method Pattern is introduced to centralize and streamline object creation.

### **Justification for Using the Factory Method Pattern**

The Factory Method Pattern is a creational design pattern that provides a standard way to create objects while allowing flexibility in determining their specific types. Instead of directly instantiating objects within the main program logic, the creation process is delegated to a dedicated factory.

Using this pattern in JabberPoint provides several key advantages:

1. **Encapsulation of Object Creation**
   1. The creation of SlideItem objects is currently spread across multiple classes. With the Factory Method, all instantiations are centralized in one place, reducing redundancy and improving organization.
2. **Improved Scalability and Extensibility**
   1. If additional slide item types need to be introduced, such as VideoItem or GraphItem, the factory can be extended without modifying the core program logic.
3. **Reduction of Code Duplication**
   1. By standardizing the way objects are created, the Factory Method prevents repeated instantiation logic in different parts of the code.
4. **Improved Maintainability**
   1. Centralized object creation simplifies debugging and modifications, making the codebase easier to manage and adapt to future requirements.

### **Implementation Plan**

To integrate the Factory Method Pattern into JabberPoint, a dedicated factory class will be introduced to handle the creation of SlideItem objects. The implementation process consists of the following steps:

1. **Creation of a Factory Interface**
   1. A factory interface will be defined, specifying a method for creating slide items. This ensures consistency in object creation across different item types.
2. **Implementation of Concrete Factories**
   1. Separate factory classes will be created for TextItem and BitmapItem. These classes will implement the factory interface and define how each item type is instantiated.
3. **Development of a General Factory Provider**
   1. A general factory class will be implemented to determine which specific factory to use based on the slide item type. This factory will receive parameters such as the item type, level, and content, and return the appropriate object.
4. **Modification of the XML Loading Process**
   1. The XMLAccessor class, responsible for loading presentations from XML files, will be updated to utilize the factory instead of manually instantiating SlideItem objects.
5. **Refactoring the Slide Class**
   1. The Slide class will be modified to use the factory when adding new slide items, ensuring that object creation remains consistent throughout the application.

### **Factory Method Pattern - Step-by-Step Process**

1. **The XML File is Loaded**
   1. The XMLAccessor class reads slide item data, including the type (text, image, etc.), content, and level.
2. **The Factory Method is Invoked**
   1. Instead of directly instantiating TextItem or BitmapItem, the factory method is used to generate the appropriate object.
3. **The Appropriate Factory Creates the Object**
   1. The general factory determines the correct factory to use and delegates object creation to it.
4. **The Slide Item is Created and Added to the Slide**
   1. The Slide class appends the newly created slide item to its internal list.

### **Expected Benefits**

By implementing the Factory Method Pattern, several improvements will be achieved:

* **Decoupling of Object Creation from Core Logic**
  + The main application logic no longer needs to be concerned with how slide items are created. This reduces dependencies and simplifies maintenance.
* **Facilitation of Future Enhancements**
  + The introduction of new slide item types will not require modifying existing code, as the factory can be extended independently.
* **Reduction of Redundant Code**
  + The elimination of repeated instantiation logic ensures greater code efficiency and consistency.
* **Improved Code Organization**
  + Having a centralized location for object creation improves the clarity and structure of the codebase.

### **Conclusion**

Integrating the Factory Method Pattern into JabberPoint will significantly enhance the software’s scalability, maintainability, and flexibility. The pattern provides a structured way to manage slide item creation, allowing future expansions without major code modifications. With this approach, JabberPoint will be better positioned for future enhancements, including support for new slide item types and improved modularity in object creation.

## **Decorator Pattern:**

JabberPoint currently lacks a flexible way to enhance slide items (such as text and images) with additional formatting or features. If a developer wanted to add support for bold text, underlined text, animated images, or captions for images, they would need to modify the TextItem and BitmapItem classes directly. This approach leads to tightly coupled code, making future extensions difficult.

To solve this issue, we propose implementing the Decorator Pattern, which allows additional functionalities to be dynamically added to slide items without modifying their base classes.

### **Justification for Using the Decorator Pattern**

The **Decorator Pattern** is a **structural design pattern** that allows behavior to be dynamically added to individual objects without altering the structure of existing classes. This pattern is highly suitable for JabberPoint because:

1. **Enhancing Slide Items Without Modifying Their Code**
   1. Instead of modifying TextItem or BitmapItem, decorators can be used to add formatting features (e.g., bold, italic, underlined) or functional extensions (e.g., clickable links, captions) without changing their original implementation.
2. **Improved Flexibility and Scalability**
   1. New features can be introduced without altering existing code. For example, if a future version of JabberPoint needs highlighted text or shadowed images, a new decorator can be created instead of modifying the base classes.
3. **Encapsulation of Additional Behaviors**
   1. Formatting logic is kept separate from core slide item logic, improving maintainability.
4. **Stackable Enhancements**
   1. Multiple decorators can be applied to the same slide item. For example, text can be both bold and underlined, or an image can have a caption and a border at the same time.

### **Implementation Plan**

To integrate the Decorator Pattern into JabberPoint, we will introduce a set of decorators that enhance SlideItem objects dynamically. The following steps outline the implementation:

1. **Define a Base Decorator Class**
   1. This class will extend SlideItem and wrap an existing SlideItem, allowing additional behavior to be added.
2. **Implement Specific Decorators**
   1. Example decorators include:
      1. BoldTextDecorator → Makes text bold.
      2. UnderlineTextDecorator → Underlines text.
      3. ImageCaptionDecorator → Adds a caption to images.
3. **Modify the Slide Class to Support Decorators**
   1. Instead of adding plain SlideItem objects, slides will now accept decorated versions of these objects.
4. **Ensure Compatibility with Existing Functionality**
   1. The decorators should respect the existing methods of SlideItem, ensuring that all enhanced items still render correctly on the slide.

### **Decorator Pattern - Step-by-Step Process**

1. **The Slide Loads Its Items**
   1. When a slide is created, it loads basic slide items like TextItem and BitmapItem.
2. **A Decorator is Applied to an Item**
   1. Instead of using a plain TextItem, a decorator (such as BoldTextDecorator) wraps it, adding additional formatting behavior.
3. **The Decorated Item is Added to the Slide**
   1. The slide now stores and processes this enhanced item, treating it just like a regular SlideItem.
4. **The Slide Renders All Items**
   1. When the slide is drawn, each decorated item displays its enhancements (e.g., bold text, underlined text, an image caption).

### **Expected Benefits**

By implementing the Decorator Pattern, JabberPoint will gain several key improvements:

* **Dynamic Feature Enhancement**
  + New features can be added to slide items without modifying their core implementation.
* **Better Code Maintainability**
  + Formatting logic and rendering logic are separated, making the code easier to understand and extend.
* **Flexible Feature Composition**
  + Multiple enhancements (such as bold and underline) can be applied to the same text without duplicating code.
* **Scalability for Future Features**
  + If JabberPoint needs more advanced text styles (e.g., strikethrough, highlight) or image effects (e.g., borders, drop shadows), new decorators can be created without modifying existing code.

### **Conclusion**

The integration of the Decorator Pattern into JabberPoint will provide a powerful and flexible way to enhance slide items without modifying their original implementations. This approach improves scalability, maintainability, and flexibility, allowing new formatting options to be added dynamically. Implementing this pattern will future-proof JabberPoint by making it adaptable to future styling and feature enhancements.

## **Testing and CI/CD Pipeline:**

To ensure that JabberPoint remains stable, scalable, and maintainable, we will implement a structured testing strategy integrated into a CI/CD pipeline using GitHub Actions. This approach will help automate testing, catch issues early, and maintain code quality.

## **Testing Strategy**

### **Unit Testing (JUnit)**

* Unit tests will be written for core functionalities such as:
  + Slide navigation (Presentation.nextSlide(), prevSlide())
  + Slide item creation (SlideItemFactory)
  + Observer updates (notifyObservers())
  + File loading (XMLAccessor.loadFile())
* JUnit 5 will be used for testing, ensuring each class functions correctly in isolation.

### **Integration Testing**

* The interaction between various components will be tested, including:
  + Presentation and SlideViewerComponent (ensuring UI updates occur when slides change)
  + Factory Method implementation (verifying correct SlideItem instantiation)
  + Decorator Pattern (ensuring multiple decorators can be applied correctly)
* These tests will ensure seamless integration between different modules of the application.

### **UI Testing (Selenium or AssertJ Swing)**

* Automated GUI tests will validate that:
  + Menu buttons function as expected
  + Keyboard navigation works correctly
  + The Observer Pattern ensures that UI updates automatically when slides change

### **Linting and Code Quality Checks**

* A static code analysis tool will be integrated to enforce coding standards and detect potential issues early.
* Tools such as Checkstyle, SpotBugs, or PMD will be configured to run automatically in the CI/CD pipeline.
* This will help identify code smells, unused imports, inconsistent naming, and other maintainability issues.

#### **CI/CD Pipeline in GitHub Actions**

A GitHub Actions workflow will be used to automate testing and deployment.

### **Branching Strategy**

The project will follow a DTAP (Development, Testing, Acceptance, Production) branching model:

* **Development (dev)** → Active development where new code is committed. No tests are run at this stage.
* **Testing (test)** → Merges from dev undergo automated testing, including unit tests, integration tests, and UI tests.
* **Acceptance (acc)** → After passing tests, code is merged into acc for final validation and manual review.
* **Production (main)** → Only fully tested and accepted code is merged into main, ensuring that the production version remains stable.

### **Automated Testing in GitHub Actions**

The CI pipeline will be triggered when code is merged into the test branch. The following steps will be executed:

1. Checkout code from the repository.
2. Set up the Java environment.
3. Run linting checks to enforce coding standards.
4. Execute unit tests using JUnit 5.
5. Run integration tests to validate component interactions.
6. Perform UI tests (if applicable).
7. Generate and store test reports.

If all tests pass in the test branch, the code is merged into acc for acceptance testing. Once manually approved, the code is merged into main for production.

### **Deployment and Code Review Process**

* Code reviews will be required for pull requests before merging.
* The test branch must pass all automated tests before moving forward.
* The acc branch serves as the final validation step before merging into main.
* The main branch contains only stable, production-ready code.

## **Expected Benefits**

* **Automated Testing:** Ensures all code is tested before merging, reducing the risk of introducing bugs.
* **Early Bug Detection:** Catches errors before they reach production, improving software reliability.
* **Improved Code Quality:** Linting and static analysis enforce coding standards and maintain clean, readable code.
* **Faster Development:** CI/CD eliminates manual testing, allowing developers to focus on feature implementation.
* **Code Stability:** Only tested and reviewed code reaches the main branch, ensuring software consistency.

## **Conclusion**

By integrating unit testing, integration testing, UI testing, linting, and a CI/CD pipeline, the JabberPoint project will achieve higher reliability and scalability. This structured approach will improve development efficiency, enforce code quality, and support future enhancements without introducing regressions.