JabberPoint Codebase Analysis & Refactoring Report

Improving Maintainability, Scalability, and Code Quality

**Table of Content**

[Introduction 2](#_Toc225191786)

[Chapter 1: Overall Problem Statement 2](#_Toc1147488936)

[Chapter 2: SOLID Principles, DRY, and KISS 3](#_Toc117259370)

[SOLID Principles 4](#_Toc1202019745)

[DRY (Don't Repeat Yourself) and KISS (Keep It Simple, Stupid) 5](#_Toc791704962)

[Chapter 3: Design Patterns 5](#_Toc45059130)

[Factory Pattern 6](#_Toc1349124789)

[Observer Pattern 7](#_Toc1865548748)

[Command Pattern 8](#_Toc1016303574)

[Chapter 4: Quality Measures 10](#_Toc1353787555)

## 

## Introduction

JabberPoint is a basic slide presentation tool that incorporates several object-oriented programming concepts and design patterns. While the application fulfills its core functionality, its current codebase presents several challenges in terms of **maintainability, scalability, and flexibility**. Key issues include **mixed responsibilities, tight coupling, code repetition, and incomplete design pattern implementations**, which make it difficult to extend and modify the system efficiently.

This document aims to analyze these challenges and propose **refactoring strategies** based on **SOLID principles, DRY, and KISS methodologies** to improve the overall structure of the code. Furthermore, it explores the implementation of **Factory, Observer, and Command patterns** to enhance flexibility and modularity. Lastly, the report outlines best practices for **code quality assurance**, including **unit testing, integration testing, and CI/CD pipelines**, to ensure long-term reliability and maintainability.

By addressing these issues and applying modern software design principles, JabberPoint can evolve into a **more robust, scalable, and easily extendable** presentation tool.

## Chapter 1: Overall Problem Statement

The JabberPoint codebase is a basic slide presentation tool. It uses several common object‐oriented ideas and design patterns. However, some issues affect how easy the code is to maintain and extend:

* **Mixed Responsibilities:**  
  Classes such as Presentation mix data management with view updates. This means the model and view logic are not clearly separated. Changing one part may affect the other and violates the Single Responsibility Principle.
* **Repetition and Tight Coupling:**  
  The code has repeated logic in user interaction handlers (for example, in both KeyController and MenuController). Direct use of concrete classes makes it difficult to extend or test the code. This breaks the DRY (Don't Repeat Yourself) principle and makes future changes harder.
* **Incomplete Pattern Use:**  
  Some design patterns from the Gang of Four are used, but not fully. For example, the Observer pattern is used in a way that the model is directly linked to a single view. This limits the possibility of adding more independent views.
* **Need for Abstraction and Decoupling:**  
  The code would be stronger if it used more abstract interfaces and reduced direct dependencies. This would follow the Dependency Inversion Principle and Interface Segregation Principle better. A refined MVC or event-driven design could separate the model and view more clearly.
* **Code Clarity:**  
  Better naming, error handling, and separation of concerns would make the code easier for new developers to understand and maintain.

In short, JabberPoint meets its goal as a simple presentation tool. However, a refactoring to separate concerns, remove duplicate code, and better follow SOLID principles would improve its scalability and maintainability.

## Chapter 2: SOLID Principles, DRY, and KISS

### SOLID Principles

* **Single Responsibility Principle (SRP):**
  + *Problem:*
    - The Presentation class handles slide data, view updates, and even exits the application.
    - Controllers like MenuController mix multiple UI commands into one class.
  + *Opportunities:*
    - Separate the responsibilities so that each class handles one task.
    - Refactor the controllers into smaller, dedicated handlers.
* **Open/Closed Principle (OCP):**
  + *Problem:*
    - New features or UI changes may force changes in classes that create or use objects, as they are tightly bound to specific implementations.
  + *Opportunities:*
    - Introduce abstractions that allow new behaviors to be added without modifying existing code.
* **Liskow Substitution Principle (LSP):**
  + *Problem:*
    - While the inheritance structure allows for interchangeable use of subclasses, any deviation from the expected behavior can cause problems in the system.
  + *Opportunities:*
    - Ensure that any subclass can fully replace its parent class without affecting the client code.
* **Interface Segregation Principle (ISP):**
  + *Problem:*
    - The code uses few interfaces, forcing clients to depend on methods they do not need.
  + *Opportunities:*
    - Introduce smaller and more specific interfaces that allow clients to depend only on what they use.
* **Dependency Inversion Principle (DIP):**
  + *Problem:*
    - High-level modules, such as controllers, depend directly on low-level implementations (for example, the Presentation class or specific Accessor types).
  + *Opportunities:*
    - Depend on abstractions instead of concrete classes, making testing and future changes easier.

### DRY (Don't Repeat Yourself) and KISS (Keep It Simple, Stupid)

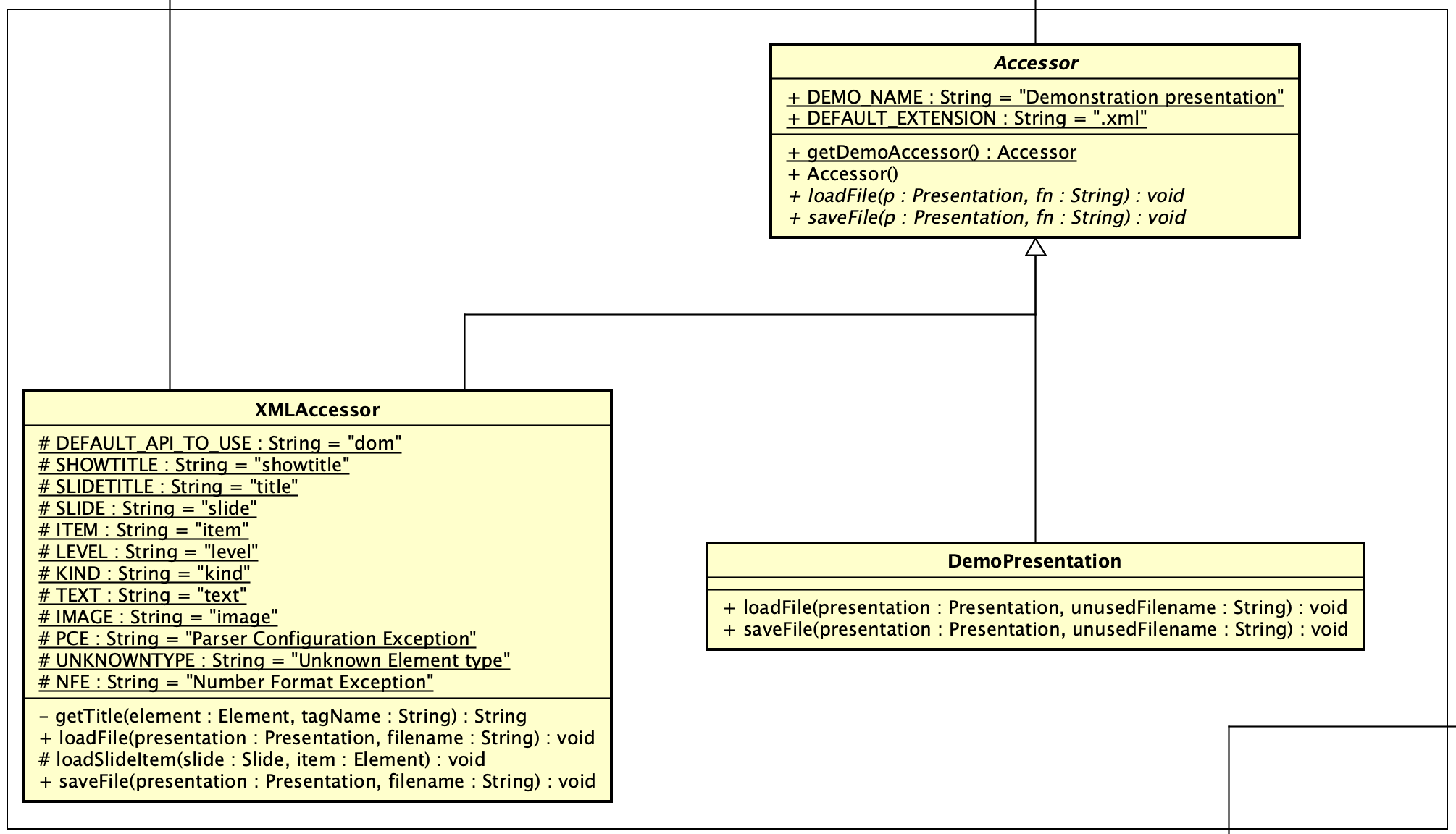
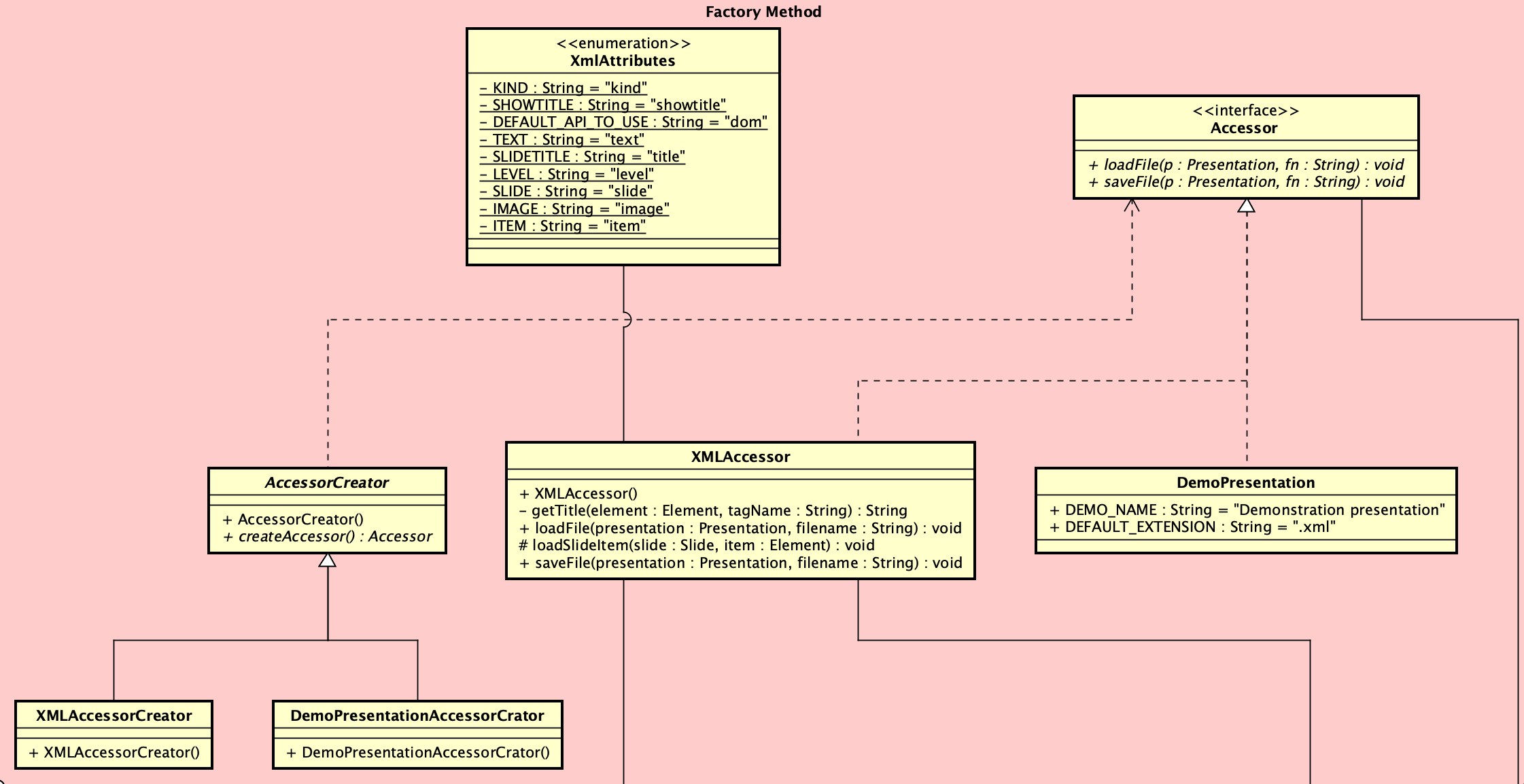
* **DRY:**
  + The same slide navigation logic appears in both KeyController and MenuController.
* **KISS:**
  + Mixing UI logic with the presentation model (e.g., in the Presentation class) makes the design more complicated than necessary.

## 

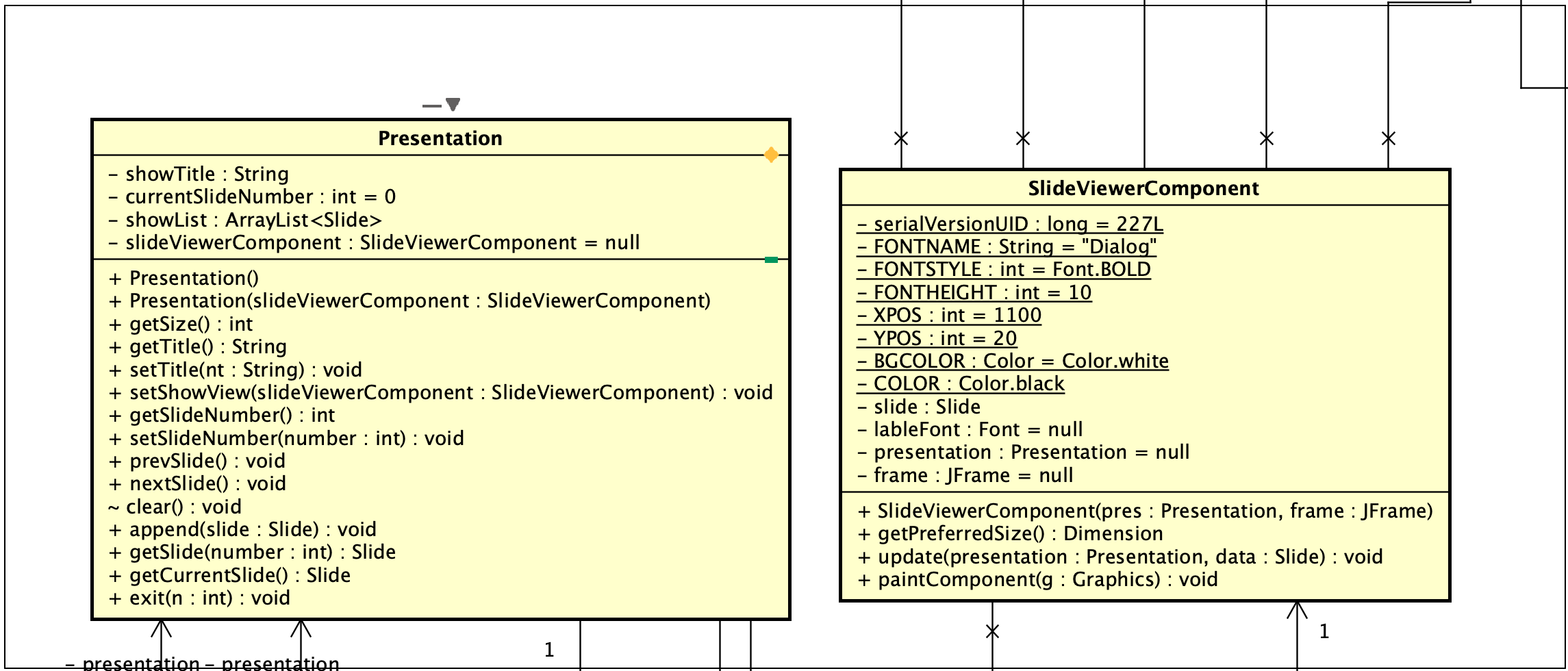
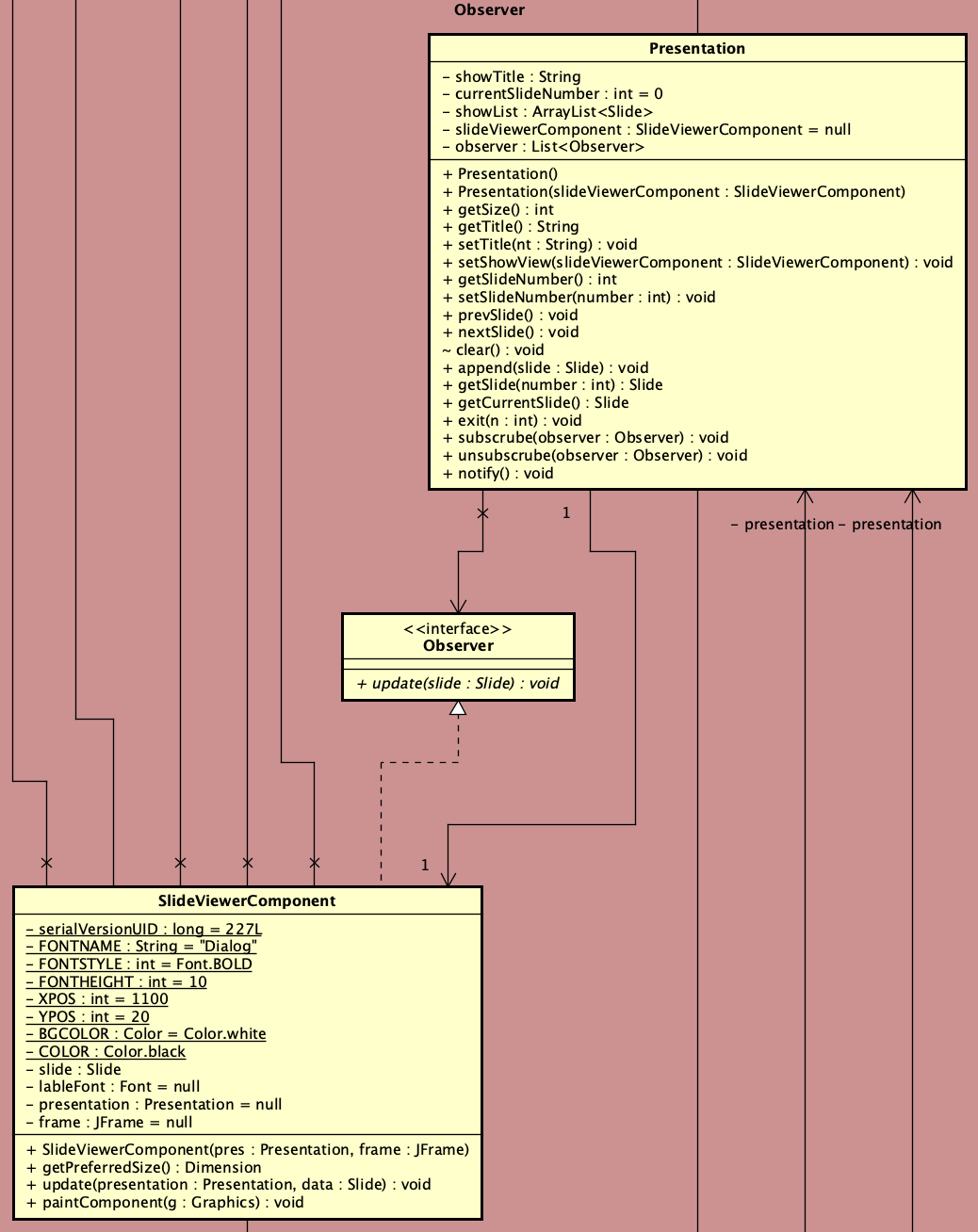
## **Chapter 3: Design Patterns**

Reflection on the Problems in the code that can be solved by the implementation of this design patterns.

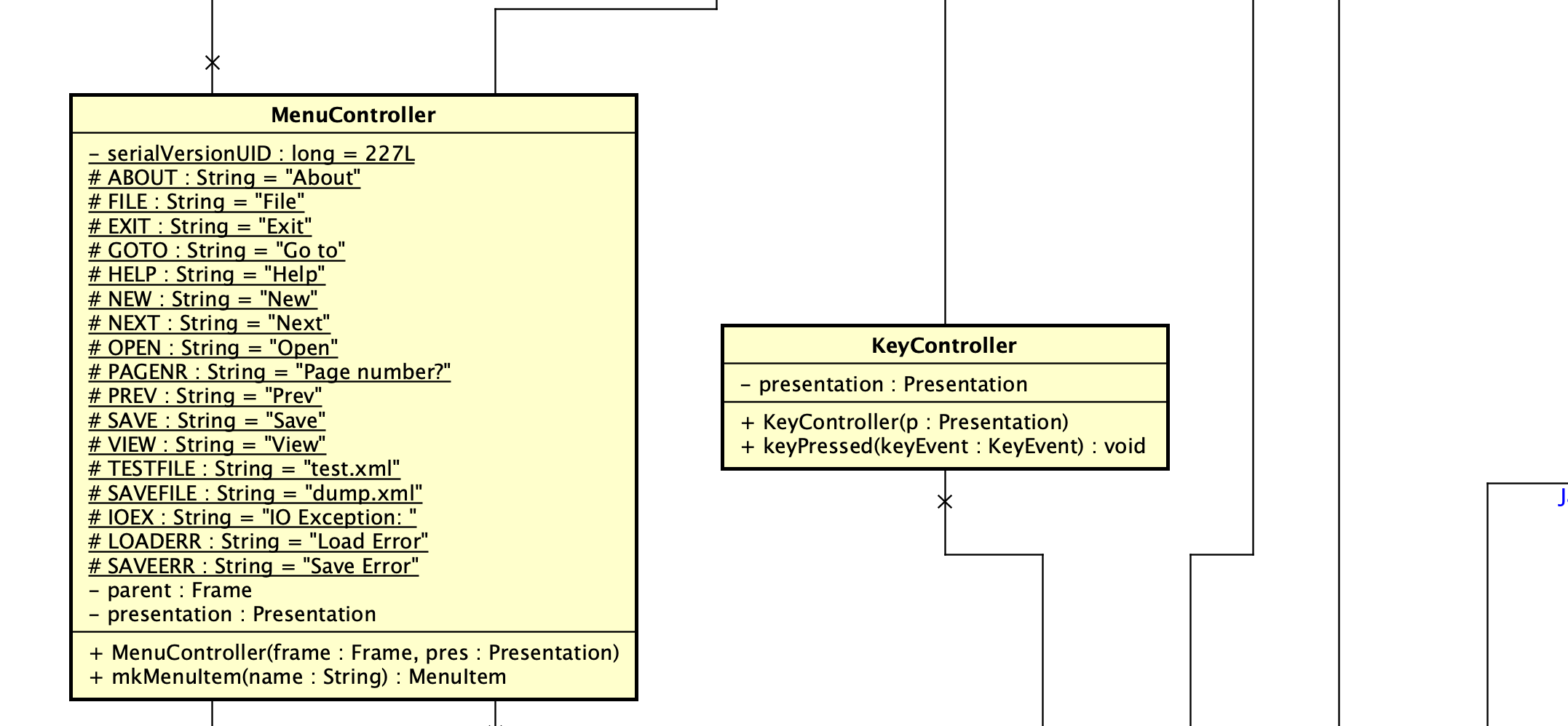
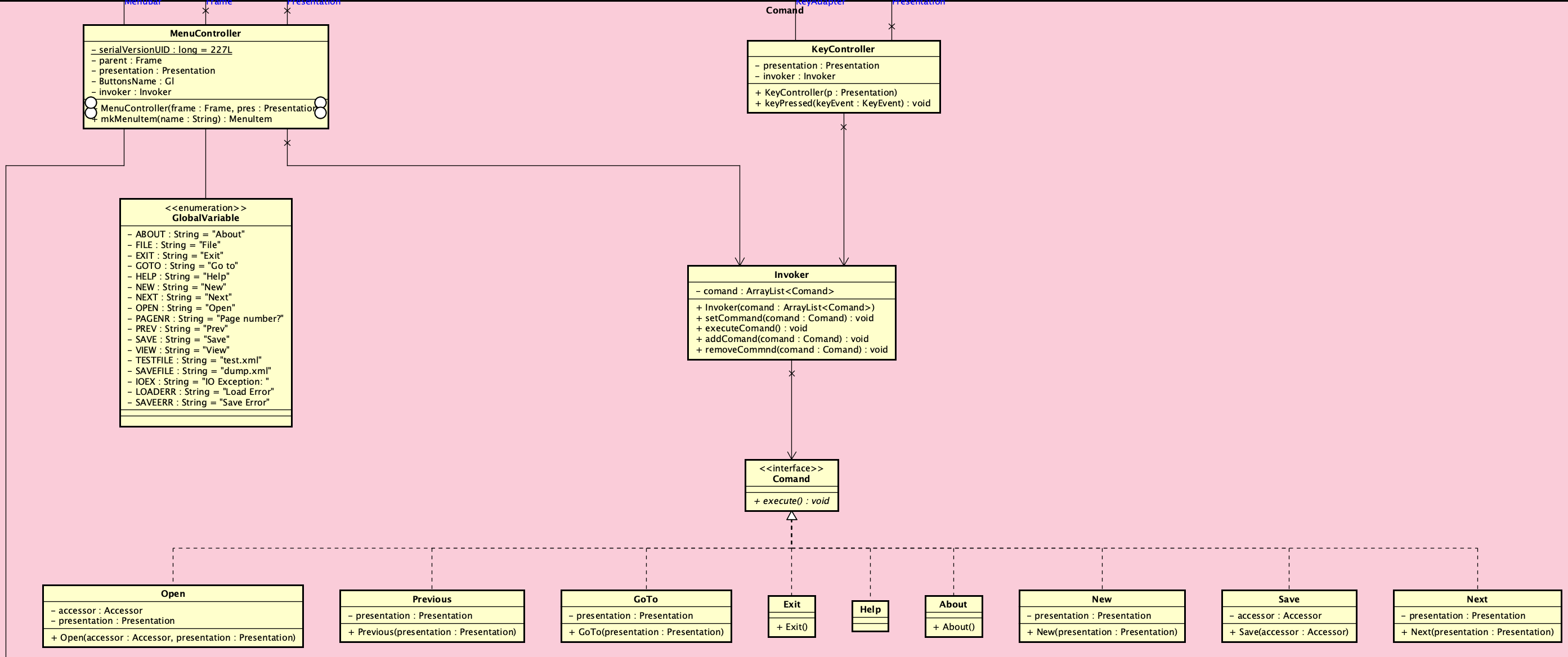
### Factory Pattern

* **Description of the Pattern:**  
  The Factory pattern provides an interface for creating objects. It allows the creation of objects without specifying the exact class. In JabberPoint, the abstract class Accessor and its subclasses (XMLAccessor and DemoPresentation) work in this way.
* **Problem Occurred:**  
  When loading or saving presentation data, the code needed different methods. Without a factory, the creation of these objects was tightly bound to specific implementations.
* **Screenshot from the Class Diagram (Problem):**  
  
* **Problem Solved:**  
  By using the Factory pattern, the code now creates Accessor objects without knowing their concrete type. This makes it easier to extend and modify the file loading/saving process.
* **Screenshot of the Implemented Pattern:**  
  

### Observer Pattern

* **Description of the Pattern:**  
  The Observer pattern allows an object (the subject) to notify other objects (observers) when its state changes. In JabberPoint, the Presentation class updates the SlideViewerComponent when the slide changes.
* **Problem Occurred:**  
  The model (Presentation) was directly linked to a single view, reducing flexibility. It did not support multiple independent views.
* **Screenshot from the Class Diagram (Problem):**  
  
* **Problem Solved:**  
  A more decoupled observer design would allow several views to subscribe to changes. This would reduce the direct dependency between the model and the view.
* **Screenshot of the Implemented Pattern:**  
  

### Command Pattern

* **Description of the Pattern:**  
  The Command pattern encapsulates a request as an object. This allows the parameterization of clients with different requests and supports features like undo/redo. In JabberPoint, the KeyController and MenuController handle user commands to navigate slides and open files.
* **Problem Occurred:**  
  The user actions (like key presses and menu selections) were hard coded in the controllers. This made it difficult to change the behavior or map new commands without modifying the controller code directly.
* **Screenshot from the Class Diagram (Problem):**  
  
* **Problem Solved (How):**  
  By abstracting commands into separate objects, the code can remap keys and menu actions more easily. This leads to a more flexible and maintainable design.
* **Screenshot of the Implemented Pattern:**   
  

## Chapter 4: Quality Measures

This chapter outlines quality measures to ensure the code remains reliable, maintainable, and easy to extend.

* **Unit Testing:**  
  Each class should have its own set of unit tests. Testing small, isolated units helps catch issues early and ensures changes do not break existing functionality. For instance, tests for Slide, TextItem, and BitmapItem can verify drawing and layout behavior.
* **Integration Testing:**  
  Integration tests should verify that controllers, models, and views work together as expected. This testing confirms that patterns like Observer and Command are functioning correctly during user interactions.
* **Automated Testing and CI/CD:**  
  Establish an automated testing framework (for example, JUnit for Java) and integrate these tests into a CI/CD pipeline. Continuous integration and continuous deployment ensure that every code change is automatically built, tested, and deployed. This approach helps catch issues early and maintains high code quality.
* **Documentation and Maintenance:**  
  Clear inline comments, external documentation, and up-to-date class diagrams help new developers understand the system quickly. Good documentation also supports easier maintenance and reduces the chance of introducing errors during changes.