ADVICE

JabberPoint

Timofei Arefev
Sviatoslav Zubrytskyi

Contents

Introduction

JabberPoint is an application that allows to present and create presentations. This application can open presentation and it supports different shortcuts like: opening presentation, closing presentation, saving presentation, and making new presentations.

Due to the unproperly structured code the whole application requires heavy modifications to be functional. The code is hard to ready, it uses “magic numbers”, it has a lot of bugs and does not follow any coding conventions. This document will propose a solution on how to fix the code base and how to improve some of its aspects.

Problems

Modifying document

- The application implies that the document can be modified, but the functionality for this is not presented and accessible throughout the application.

Shortcuts

- Creating new presentation and going to the next slide both have the same shortcut (ctrl + n)
- “Go to” shortcut does not have an upper or lower boundary

General functionality

- Open file does not work
- Save file does not work
- Slides that contain images can’t be opened

Code

- Most classes don’t follow code conventions
- Some classes don’t follow general code structure of Java (Style)
- Code contains magic numbers (Style)

Code conventions

To fix the coding conventions in the code base we decided to follow the NHL Stenden code conventions due to our previous experience with using them.

Code conventions

Testing

Testing is required by scoring rubrics and should contain at least 70% of the code covered. That is lower than the ideal bar, but more than enough for testing all critical functionalities of the software.

Unit testing

As we will write code on plain Java, and it is a standard software for the pc consisting of classes, the main thing that must be tested is classes, particularly methods inside them. What is the best way to do this? Unit tests.

We will use JUnit 5 (https://github.com/junit-team/junit5/) to test our application on

Line coverage

To determine the exact line coverage of the code we may use Jacoco (https://github.com/jacoco/jacoco). It is a library that will automatically analyze all our code and return a percentage of line coverage.

End to end tests

No automated end-to-end tests will be made.

Other tests

As far as we are concerned, we do not need any kind of network testing, additional integrity testing, database testing ect.

Automated testing (CI/CD)

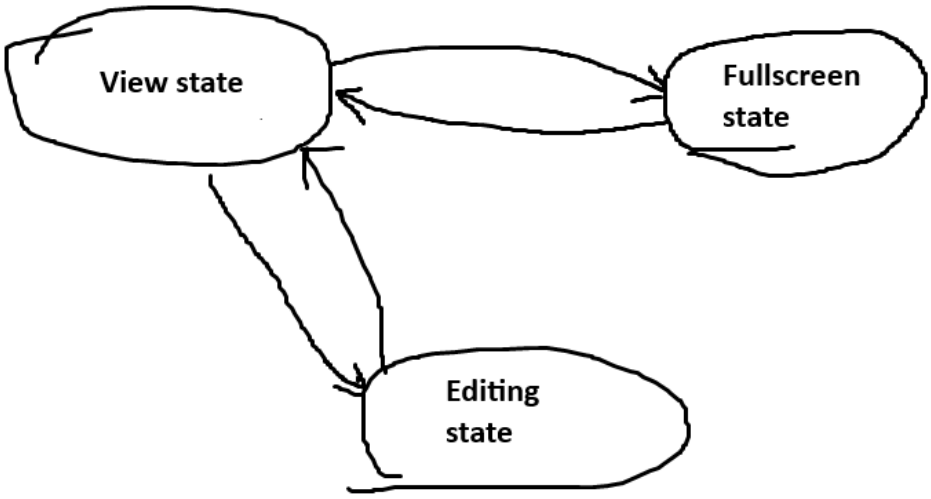
To ensure that no untested code gets into production, our main branch will be locked from direct pushing. Instead, we will be using Pull Requests and Github Actions to automatically test code on Pull Request and DENY it if any of the checkmarks fails. Those checkmarks will be based off our Unit Tests written on Junit 5.

Desing Patterns

State Pattern

The project require usage of different states (editing mode, viewing mode, fullscreen mode). The state pattern will allow us to introduce this pattern which will ensure that the

program does not have any wrong transitions which will reduce number of bugs in the code, while removing significant amount of code. By isolating mode-specific logic within separate state classes, we can eliminate unnecessary conditionals (if-else or switch statements), leading to cleaner and more maintainable code. This approach will also make it easier to introduce new modes in the future without modifying existing logic."



Why?

By encapsulating behavior related to each mode within separate state classes, we ensure that each mode follows a well-defined transition path without side effects. This will help to prevent errors where UI elements or application logic act incorrectly due to being in an invalid state. Additionally, this separation allows for better code reusability and scalability, as new states can be introduced without affecting the existing ones.

Mediator Pattern

The Mediator Pattern will allow the code to ensure absence of chaotic dependencies by centralizing communication between UI components and the Presentation class. Instead of communication with each other the system will communicate with PresentationMediator which will remove dependency of different components on each other.

Why?

By using this pattern, we can eliminate direct dependencies between KeyController, MenuController, SlideViewerComponent, and Presentation, making it easier to modify or extend functionality without affecting multiple components. This approach will help to improve modularity by allowing UI elements to be reused and not being tied to specific implementations.

Factory Method

The Factory Method Pattern will allow the system to handle the creation of SlideItem objects (such as TextItem and BitmapItem) in a more flexible and scalable way. Instead of creating objects directly within the XMLAccessor class or other parts of the code, the Factory Method will give this responsibility to specialized factory classes. This ensures that object creation is centralized and controlled, reducing code duplication and making it easier to introduce new slide item types in the future.

Why?

Using a factory makes it easier to add new slide items without changing existing code. Instead of manually creating objects, we let the factory handle it, ensuring a consistent and organized way to generate different slide items. This keeps the code clean, flexible, and easy to maintain, while also reducing duplication and dependencies between classes.