Refactoring Jabberpoint Report

ENHACING MAINTAINABILITY AND SCALABILITY

Rares Husarescu
UNIVERSITY OF APPLIED SCIENCES NHL STENDEN

Sommario

Chapter 1: Executive Summary

Jabberpoint, a Java-based tool originally crafted for straightforward presentation needs, stands as a testament to early software design in the multimedia domain. Capable of delivering basic functionalities and reading static presentations from XML files, the application carved its niche among straightforward presentation tools. However, as technology advanced, the limitations of Jabberpoint's architecture became increasingly evident. Its inability to gracefully evolve with user demands and modern development practices exposed a myriad of questionable design choices that were made when software architecture was a nascent art.

Its primitive nature, while once a hallmark of simplicity, now underlines its obsolescence. The lack of modern features, a user interface that has not kept pace with user experience trends, and a codebase fraught with rigid structures highlight the urgent need for renovation. The application's overreliance on XML files, without supporting more dynamic or interactive content, further cements its status as a relic of a bygone era.

In my journey to rejuvenate Jabberpoint, I embarked on a thorough refactoring process. This process was not merely a facelift but a foundational overhaul, aimed at rectifying the deep-seated architectural flaws while preserving the simplicity at the core of Jabberpoint's initial success. Through the introduction of the MVC pattern, the embrace of abstraction, and the rigorous application of modern error handling and design principles, we sought to transform Jabberpoint into an application befitting the demands of today's users and developers alike.

Chapter 2: Initial Architectural Assesement

2.1 – Tight Coupling and Inflexibility

The initial class structure of Jabberpoint posed significant challenges. The blending of UI logic, data management, and application control logic within singular classes led to a high degree of tight coupling. This rigidity made simple tasks, like updating UI components or modifying data structures, disproportionately complex and error-prone.

2.1.1 – Code Example

The **Slide** class is responsible for both managing slide items (**Vector<SlideItem> items**) and rendering (**public void draw(Graphics g, Rectangle area, ImageObserver view)**). This tight coupling between data management and UI rendering makes the class inflexible to changes in either functionality.

```
protected Vector<SlideItem> items; //The SlideItems are kept in a vector
//Draws the slide
public void draw(Graphics g, Rectangle area, ImageObserver view) {
    float scale = getScale(area);
    int y = area.y;
//The title is treated separately
    SlideItem slideItem = new TextItem(0, getTitle());
    Style style = Style.getStyle(slideItem.getLevel());
    slideItem.draw(area.x, y, scale, g, style, view);
    y += slideItem.getBoundingBox(g, view, scale, style).height;
    for (int number=0; number<getSize(); number++) {
      slideItem = (SlideItem)getSlideItems().elementAt(number);
      style = Style.getStyle(slideItem.getLevel());
      slideItem.draw(area.x, y, scale, g, style, view);
      y += slideItem.getBoundingBox(g, view, scale, style).height;
    }
  }
```

2.2 – Single Responsibility Principle Violation

Classes in the original design frequently violated the Single Responsibility Principle, taking on multiple roles within the application. This not only made code comprehension and maintenance difficult but also stifled unit testing efforts due to the intertwined nature of functionalities.

2.2.1 – Code Example

The **Presentation** class merges the management of presentation slides (such as maintaining a list of slides and the current slide index) with direct interaction with the UI component (**SlideViewerComponent**). This amalgamation of presentation logic and view management duties, particularly in methods like **setSlideNumber(int number)**, which updates both the model's state and the view, exemplifies a breach of the Single Responsibility Principle. The tight coupling between model operations and UI updates restricts the class's adaptability and complicates modifications to either the data model or how it's displayed.

```
private ArrayList<Slide> showList = null; //An ArrayList with slides
private SlideViewerComponent slideViewComponent = null; //The view component of
the slides
//Change the current slide number and report it the the window
public void setSlideNumber(int number) {
    currentSlideNumber = number;
    if (slideViewComponent != null) {
       slideViewComponent.update(this, getCurrentSlide());
    }
}
```

2.3 – Lack of Abstraction and Direct class Manipulation

Direct instantiation of classes without interfaces led to a codebase that was resistant to change. This absence of abstraction layers hindered the potential for future integrations and the adaptability required for an evolving product landscape.

2.3.1 – Code Example

The class directly uses the DOM API for XML parsing, with no abstraction layer that would allow for changing the parsing mechanism without modifying the **XMLAccessor** class.

```
DocumentBuilder builder =
DocumentBuilderFactory.newInstance().newDocumentBuilder();
Document document = builder.parse(new File(filename)); //Create a JDOM document
```

It directly interacts with the file system to save presentations to XML files, using **PrintWriter** and **FileWriter** without an intermediate abstraction that would facilitate different storage mechanisms.

```
PrintWriter out = new PrintWriter(new FileWriter(filename));
```

Chapter 3: Refactored Design Overview

My refactored design introduces a clear **MVC** architecture, segregating responsibilities into distinct layers, and promoting code that is robust, scalable, and easier to maintain.

3.1 – Model-View-Controller (MVC) Architecture

3.1.1 - Model Layer

The Model is now purely focused on data and business logic, encapsulating the state of the application and its fundamental behaviors. The introduction of **PresentationModel** and **SlideModel**, each handling specific domains of the data model, ensures a single source of truth for the state of the application.

3.1.2 - View Layer

The View has been liberated from any data manipulation responsibilities. Classes such as **MainView** and **SlideView** now solely manage the presentation of data, reacting to model updates and providing a dynamic user experience.

3.1.3 - Controller Layer

Controllers now act as the intermediaries for user input and application output. **MainController**, **MenuController**, and **FileController** serve to translate user actions into operations on the Model and updates to the View, maintaining a clear boundary between user interface concerns and application logic.

3.2 – Implementation of Design Patterns

3.2.1 – Observer Pattern

Introducing the Observer pattern, **PresentationModel** and **SlideModel** notify **MainView** and **SlideView** of changes, promoting a reactive design where updates to the model are automatically reflected in the user interface.

Chapter 4: Justification of the Refactored Design

4.1 – Addressing Tight Coupling and Inflexibility

Old Design Issue: The original **Slide** class combined slide item management with rendering functionality, causing tight coupling between data management and UI rendering.

New Design Solution: In the refactored design, slide management is handled by **SlideModel**, focusing solely on data representation, while **SlideView** takes charge of rendering slides. This separation follows the MVC architecture, where **SlideModel** belongs to the Model layer and **SlideView** to the View layer.

4.2 – Resolving Single Responsibility Principle Violation

Old Design Issue: The **Presentation** class was overloaded with responsibilities, managing presentation slides and directly interacting with the view component.

New Design Solution: The responsibility for managing the presentation logic now rests with **PresentationModel**, focusing exclusively on the sequencing and storage of slides. Meanwhile, **PresentationView** observes changes in **PresentationModel** and updates the UI accordingly, facilitated by implementing the Observer pattern.

4.3 – Enhacing Abstraction and Reducing Direct Class Manipulation

Old Design Issue: Direct manipulation of XML files for saving and loading presentations in the **XMLAccessor** class lacked abstraction.

New Design Solution: Abstraction is introduced through **DataAccessInterface**, which **XMLDataAccess** implements. This interface abstracts away the specifics of data storage and retrieval, allowing for flexibility in changing data sources or formats without altering the rest of the application.

4.4 – Overcoming Challenges in Extensibility and Testing

Old Design Issue: The monolithic structure made it difficult to extend functionality or isolate components for testing.

New Design Solution: By adhering to the MVC architecture and using design patterns like Observer, the new design is modular, with clear boundaries between components. This modularity supports easier extension of functionality and isolation for unit testing.