Object Oriented Design
Carla Redmond
1586331
1 February 2024
Arvid Fens

Jabberpoint Refactoring

The purpose of this report is to identify issues in the current Jabberpoint application and its source code, and to suggest refactoring solutions. Jabberpoint is a rudimentary presentation tool, primarily for showing XML files in the form of slideshows.

From the standpoint of user experience, the application lacks many essential features. Moreover, its underlying source code exhibits architectural shortcomings. Addressing these issues would enhance the application's maintainability, sustainability, and scalability, and optimize its performance and efficiency.

In this report, each class within the application will be examined in alphabetical order. The focus will be on identifying specific problems or architectural peculiarities that warrant refactoring.

Problem	***Solution***	***Justification***
About		
An abstract class has a public constructor	Add a private constructor	An abstract class should not have public constructors to prevent instantiation.
Message in AboutBox uses unnecessary string concatenation	Replace this String concatenation with a Text block.	Replacing string concatenation with a text block improves readability.
A link inside the AboutBox message uses HTTP instead of HTTPS	Change to HTTPS	Using HTTPS instead of HTTP enhances security.
Accessor		
Unused field 'DEFAULT_EXTENSION'	Remove field	Removing dead, unused code declutters the codebase
Accessor is an abstract class with no state maintenance or code sharing so it would better serve its purpose as an interface	Convert to interface	No code sharing or state maintenance necessary therefore this class can become an interface
An abstract class has a public constructor	Change the visibility of this constructor to "protected".	Abstract classes should not have public constructors as it could lead to mistakenly initializing an abstract class instance

Method modifiers not in conventional order	Reorder the modifiers to comply with the Java Language Specification.	Method modifiers follow a specific order to improve code readability and consistency
Insufficient error handling	Creating a custom Accessor Exception class	Creating custom exceptions enhances code readability and makes debugging easier.
Violates Single Responsibility Principle, this class has a lot of responsibilities which makes the code difficult to comprehend and difficult to debug	Split the class into single responsibility classes	Adhering to SRP makes the code more maintainable and easier to debug and make changes to in future
BitmapItem		
Violates Single Responsibility Principle, the loading of the image and handling an exception are both done in the constructor, at the least error handling should not be in the constructor	Separate loading of image and handling exception in constructor	Adhering to SRP makes the code more maintainable and easier to debug and make changes to in future
Inefficient error handling	Replace this use of System.err by a logger.	Logs are more readable, recordable, accessible and secure than using standard outputs
Field 'imageName' is not final	Field 'imageName' may be 'final'	Marking a field as final ensures its immutability after initialization.
Constructor 'BitmapItem()' is never used	Remove empty constructor	Removing dead, unused code declutters the codebase
DemoPresentation		
Violates Single Responsibility Principle, this class handles both reading and writing of the demo presentation	Split into two classes	Adhering to SRP makes the code more maintainable and easier to debug and make changes to in future
Field 'unusedFileName' is never used	Remove field	Removing dead, unused code declutters the codebase
JabberPoint		
Violates Single Responsibility Principle, styles are created here	Move creation of styles to its own class	Adhering to SRP makes the code more maintainable and easier to debug and make changes to in future

Fields that are not going to change and are used in multiple places	Move to an enum	Moving fields to a utility class centralizes common functionality, reducing data clumps, duplication and enhancing code reusability
KeyController		
Field 'presentation' may be 'final'	Make final	Marking a field as final ensures its immutability after initialization.
keyPressed(keyEvent) overrides KeyAdapter method	Add the "@Override" annotation above this method signature	The @Override annotation ensures methods are correctly overriding superclass methods
The used switch case is too long	Merge the previous cases into this one using comma-separated labels.	Merging switch cases into comma-separated labels simplifies the code, reduces duplication, and enhances readability
Duplicate exit method	Use centralized method	Centralizing a frequently used method enhances maintainability and reduces duplication
MenuController		
"parent" is the name of a field in "MenuComponent".	Rename	Having fields with the same name in different classes can lead to confusion.
Field 'parent' may be 'final'	Make final	Marking a field as final ensures its immutability after initialization
Field 'presentation' may be 'final'	Make final	Marking a field as final ensures its immutability after initialization
serialVersionUID field is not clearly marked	serialVersionUID' can be annotated with '@Serial' annotation	Using the `@Serial` annotation clarifies which elements are intended for use with serialization.
Unclear assignment of menuItem	Extract the assignment out of this expression	Extracting assignments out of expressions simplifies debugging.
Anonymous new ActionListener() can be replaced with lambda	Replace with lambda	Lambda expressions offer more concise and readable code compared to anonymous inner classes.

Redundant casting of PAGENR to Object	Remove this unnecessary cast to "Object".	Removing redundant code simplifies the codebase, making it easier to maintain and understand
Violates Single Responsibility Principle, would be good to have a method for each menu for clarity	divide class into more manageable methods for each function	Adhering to SRP makes the code more maintainable and easier to debug and make changes to in future
Presentation		
Variable 'slideViewComponent' initializer 'null' is redundant	Remove	Removing redundant code simplifies the codebase, making it easier to maintain and understand
Explicit type argument is used in ArrayList<Slide>	Replace with <>	It allows the compiler to infer the appropriate type, making code cleaner.
Redundant casting of showList.get(number) to Slide	Remove cast	Removing redundant code simplifies the codebase, making it easier to maintain and understand
Slide		
Using Vector type	Replace the synchronised class "Vector" by an unsynchronized one such as "ArrayList" or "LinkedList".	Improved performance, reduced overhead, better scalability, simplicity.
Explicit type argument is used in Vector<SlideItem>	Replace with <>	It allows the compiler to infer the appropriate type, making code cleaner.
Method 'getSlideItem(int)' is never used		Removing dead, unused code declutters the codebase
Redundant casting of items.elementAt(number) to SlideItem	Already slide items	Removing redundant code simplifies the codebase, making it easier to maintain and understand
SlideItem		
Variable 'level' initializer '0' is redundant	Remove	Removing redundant code simplifies the codebase, making it easier to maintain and understand
Abstract class has a public constructor	Change the visibility of this constructor to "protected".	Abstract classes should not have public constructors

Constructor 'SlideItem()' is never used	Remove	Removing dead, unused code declutters the codebase
SlideViewerComponent		
Slide and Presentation are not transient even though this class uses Serialisation	Make "slide" transient or serializable.	could cause crashes
Variable 'labelFont' initializer 'null' is redundant	Remove	Removing redundant code simplifies the codebase, making it easier to maintain and understand
Variable 'presentation' initializer 'null' is redundant	Remove	Removing redundant code simplifies the codebase, making it easier to maintain and understand
Variable 'frame' initializer 'null' is redundant	Remove	Removing redundant code simplifies the codebase, making it easier to maintain and understand
serialVersionUID field is not clearly marked	serialVersionUID' can be annotated with '@Serial' annotation	Using the `@Serial` annotation clarifies which elements are intended for use with serialization.
Unclear override method	Add the "@Override" annotation above this method signature	The @Override annotation ensures methods are correctly overriding superclass methods
SlideViewerFrame		
serialVersionUID field is not clearly marked	‘serialVersionUID' can be annotated with '@Serial' annotation	Using the `@Serial` annotation clarifies which elements are intended for use with serialization.
Unclear override method	Add the "@Override" annotation above this method signature	The @Override annotation ensures methods are correctly overriding superclass methods
Method modifiers not in conventional order	Reorder the modifiers to comply with the Java Language Specification.	Method modifiers follow a specific order to improve code readability and consistency
Style		
Unclear assignment of fontSize	Extract the assignment out of this expression.	Extracting assignments out of expressions simplifies debugging.
TextItem		

Field 'text' is not final	Field 'text' may be 'final'	Marking a field as final ensures its immutability after initialization
Constructor 'TextItem()' is never used	Remove	Removing dead, unused code declutters the codebase
text.length()===0 is unnecessary	text.length() == 0' can be replaced with 'text.isEmpty()'	Using the built in isEmpty method makes the code more readable
Could use a for loop here	while' loop can be replaced with enhanced 'for'	Using an enhanced for loop here iImproves the maintainability and readability of the function
Implicit cast from 'float' to 'int' and from 'double' to 'int' in compound assignment can be lossy	Cast whole expression to int	Implicitly casting from 'float' to 'int' and from 'double' to 'int' in compound assignment can be lossy
Explicit type argument TextLayout	Replace with <>	It allows the compiler to infer the appropriate type, making code cleaner.
XMLAccessor		
Field 'DEFAULT_API_TO_USE' is never used	Remove	Removing dead, unused code declutters the codebase
Variable 'max' initializer '0' is redundant	Remove	Removing redundant code simplifies the codebase, making it easier to maintain and understand
Variable 'maxItems' initializer '0' is redundant	Remove	Removing redundant code simplifies the codebase, making it easier to maintain and understand
No security for xml parsing	Disable access to external entities in XML parsing.	Disabling access to external entities in XML parsing is essential to prevent potential XXE injections
Inefficient error handling	Replace this use of System.err or System.out by a logger.	Logs are more readable, recordable, accessible and secure than using standard outputs
Vector is outdated	Replace the synchronized class "Vector" by an unsynchronized one such as "ArrayList" or "LinkedList".	Improved performance, reduced overhead, better scalability, simplicity.
Redundant casting of SlideItem	Remove this unnecessary cast to "SlideItem".	Removing redundant code simplifies the codebase, making it easier to maintain and understand

Cast and check can be a lot shorter	Replace this instanceof check and cast with 'instanceof TextItem textitem'	Using pattern matching makes code less verbose and improves readability

In the process of refactoring JabberPoint, practical implementation revealed several areas for improvement that were not initially anticipated in the theoretical phase of the report. Throughout these changes, the focus was on enhancing the application's scalability and future capabilities, while maintaining its current functionality.

Key updates included the addition of JavaDocs to all classes and methods, enhancing future maintainability. Classes were reorganized into distinct packages to maintain a tidy codebase, especially in anticipation of future class additions. Redundant text variables were consolidated into an enum class with getters, simplifying their management.

The Accessor class was converted to an interface instead of an abstract class. To adhere to the Single Responsibility Principle, the Accessor class was divided into two interfaces to individually handle the read and write functions.

The subclass structure under the Accessor class, initially comprising XMLAccessor and DemoPresentation, was similarly divided. In addition, DemoPresentation was renamed to DemoAccessor for naming consistency.

In the XMLReadAccessor class, the loadFile method, previously lengthy and complex, was extracted into smaller, more manageable methods such as initialiseXmlParser, parseXmlFile, processPresentation, and processSlide. This restructuring aimed at improving code readability and adherence to the Single Responsibility Principle.

The MenuController was also reorganized, with each menu being encapsulated in its own method, thereby improving file readability.

Lastly, the Style class, which initially created an instance of itself, was split into two classes: Style and StyleCollection. The StyleCollection now utilizes a hashmap for style storage, selected for its efficiency in data retrieval and storage.