Technological University Dublin

Java Web Browser with Design Patterns

By Enzo Corsini

A review submitted in partial fulfilment for the Bachelor of Science in Computing Object Oriented Design Patterns Module in the School of Computing Department of Informatics and Engineering

April 2022

Declaration of Authorship. I hereby certify that this material, which I now submit for assessment on the programme of study leading to the award of Ordinary Degree in Computing in the Technological University Dublin - Blanchardstown, is entirely my own work except where otherwise stated.

Author: Enzo Corsini

Dated: April 2022

The following work studies the application of programming design patterns in the creation of a web browser. Design patterns are divided into three categories, the ones applied to this project are the following:

- Creational pattern: Singleton
- Structural pattern: Decorator,
- Behavioural patterns: Template, and Memento

The codebase used was extracted from the second chapter of the book Advanced Java 2 Platform: How to Program by Deitel and Deitel where they explain the use JComponents by creating several web browsers with different functionalities. I decided to merge two of the examples into one browser. A browser with a split screen to view your favourite websites and a browser with Localization using three languages.

The code snippets in this report will try to omit code that is not directly related to design patterns in order to focus on the patterns instead.

Creational Patterns: Singleton

Contrary to how many creational patterns work, singleton is used to restrict the creation of objects.

I chose this pattern because the program uses a small JFrame with a combo box to select the language of the browser. This frame, which is described as the launcher, was ideal to implement as a Singleton because only we do not require multiple instances of this class.

Other creational patterns weren't used because the instances that the application constructs are too simple to apply other patterns like Builder, the values passed are always of the same type so it's impractical to use Factory or Abstract Factory.

```
public class Launcher extends JFrame {
      private static Launcher instance = new Launcher();
      private JComboBox<Locale> localeComboBox;
      private Launcher() {
            super("Browser Launcher");
            …
      }
      public static Launcher getInstance(){
            return instance;


}
```

In this code snippet we see that the class Launcher has a private constructor and a static method is used to get the instance of the class.

Structural pattern: Decorator

Decorator was used in the front-end instead of the back-end to "decorate" buttons.

The Decorator pattern was placed in a separate folder where the abstract decorator class and the concrete decorator(HighlightDecorator) were coded.

```
abstract class Decorator extends JComponent {
	public Decorator(JComponent c) {
		setLayout(new GridLayout());
		add(c);
	}
}public class HighlightDecorator extends Decorator {
	boolean mouse_over;
	JComponent comp;
	public HighlightDecorator(JComponent c) {
		super(c);
		comp = this;
		mouse_over = false;
		c.addMouseListener(new MouseAdapter() {

		@Override
			public void mouseEntered(MouseEvent e) {…}
		@Override
			public void mouseExited(MouseEvent e) {…}
	});
		setVisible(true);

	}
	@Override
	public void paint(Graphics g) {
		…
	}
```

The mouseEntered method inside the highlighter decorator draws a yellow rectangle to whichever JComponent it is applied to. The mouseExited repaints it back to the default.

Behavioural Pattern: Template

While researching for a behavioural pattern to implement, Template method pattern was highlighted because it seemed that I had already applied it to the program without even noticing it.

Template describes that the implementation of a method is declared in a superclass and it is later defined in the subclasses.

```
public abstract class TemplateAction extends AbstractAction {

	public TemplateAction() {}
	…
	public abstract void actionPerformed(ActionEvent e);

}
```

The template action class is an abstract class that declares an abstract method to be implemented by another subclass

```
public WebToolBar(WebBrowserPane browser, Locale locale) throws
MalformedURLException {
		…
		TemplateAction backAction = new TemplateAction() {

			@Override
			public void actionPerformed(ActionEvent e) {
				…
			}
		};
		…
		TemplateAction forwardAction = new TemplateAction() {
			@Override
			public void actionPerformed(ActionEvent e) {
				…
		};
		…
		TemplateAction historyAction = new TemplateAction() {

			@Override
			public void actionPerformed(ActionEvent e) {
				…
			}
		…
```

Later on, the template action is instantiated in the WebToolBar class and each object that is created is required to implement the actionPerformed method.

Behavioural Pattern: Memento

The memento pattern was chosen to build a history log by saving the url of the pages visited into a list of states. Even though the main application -web browser- already uses a list to store the pages it has visited in order to make the forward and back buttons, I have chosen to implement a Memento pattern to create a history instead of using the same list that the browser uses.

The memento pattern consists of three classes, Memento(EditorState), Originator(Editor), CareTaker(History).

```
public class Editor {
	private URL content;
	public EditorState createEditorState(){
		return new EditorState(content);
		}
	public URL getContent() {
		return content;
	}
	public void restore(EditorState state) {
		content = state.getContent();
	}
	public void setContent(URL content) {
		this.content = content;
	}
}
```

The Editor class' main job is to create EditorState objects that later will be stored in an instance of the History class.

```
public class EditorState {
	private final URL content;

	public EditorState(URL content) {
		this.content = content;
	}
	public URL getContent() {
		return content;
	}
}
```

The History class works sort of like a database by storing EditorState objects and also providing methods to access and view the states.

```
public class History {
      …
      private List<EditorState> states = new
ArrayList<EditorState>();

      public void push(EditorState state) {
       states.add(state);
      }
      public EditorState pop() {
            var lastIndex = states.size()-1;
            var laststate = states.get(lastIndex);
            states.remove(laststate);
            return laststate;
 }
      public List getList(){
            return states;
      }
      @Override
      public String toString() {
            …
      }
}
```