An Attempt to Give Swing a Breath...

Zougianos Georgios

Diploma Thesis

Supervisor: Zarras Apostolos

Ioannina, July, 2021

Τμήμα Μηχ. Η/Υ & Πληροφορικής
Πανεπιστήμιο Ιωαννίνων
Department of Computer Science & Engineering
University of Ioannina

Acknowledgments

I would like to express my gratitude to my supervisor professor Apostolos Zarras. Not only for his constant support during this thesis, but also for how he changed the way I perceive software development.

July 2021

Abstract

Swing is one of Java's graphical user interface (GUI) toolkits for desktop application development, first released back in 1997. Since Swing was first released, the software world has been changed drastically. Nowadays, Swing is considered old and something that has been left behind. In the context of this thesis, we present a small framework that was developed for Swing applications. The framework introduces modern techniques & patterns in Swing GUI development and aims at solving some of the issues the developer meets during Swing GUI development.

Keywords: Java, Swing, GUI

Περίληψη

Η Swing είναι ένα από τα εργαλεία γραφικών διεπαφής χρήστη (GUI) της Java για την ανάπτυξη εφαρμογών για επιτραπέζιους υπολογιστές, που κυκλοφόρησε για πρώτη φορά το 1997. Από την πρώτη κυκλοφορία της Swing, ο κόσμος του λογισμικού έχει αλλάξει δραστικά. Σήμερα, η Swing θεωρείται παλαιά και κάτι που έχει μείνει πίσω. Στο πλαίσιο αυτής της διπλωματικής, παρουσιάζουμε ένα μικρό framework που αναπτύχθηκε για εφαρμογές Swing. Το framework εισάγει σύγχρονες τεχνικές και προγραμματιστικά μοτίβα στην ανάπτυξη ενός Swing GUI και στοχεύει στην επίλυση ορισμένων από τα ζητήματα που αντιμετωπίζει ο προγραμματιστής κατά την ανάπτυξη ενός Swing GUI.

Λέξεις κλειδιά: Java, Swing, GUI

Table of Contents

Chapter 1. Introduction

1.1 Motivation

Day by day, companies & organizations are leaving desktop and moving to web applications. The development of a web application can have a lot of benefits in comparison to a desktop one. Especially nowadays, when there are dozens of frameworks and tools out there, capable of doing a lot of things with the fewest amount of lines of code required. There is a big controversy and numerous different opinions on whether someone or some profitable organization should spend time and effort in developing a desktop application. The most common opinion is sheer. It states that desktop applications are dead. Even if nobody can disagree that there is a decline in the number of desktop applications that are being developed during the latest years, this is not completely true. If it was, the only thing that would be installed in a personal computer would be a web browser. But it's not. Simply because both desktop and web applications are capable of solving different problems in different ways.

When it comes to desktop application development, there are plenty of tools in a variety of different programming languages anyone can choose from. Every language has its own strengths and weaknesses [Sali14]. Choosing the right tool or programming language to develop a desktop application does not differ from choosing the tool for any job. It is, or at least should be, based on the requirements.

Desktop application development with Java in specific, comes with its own advantages. One advantage that must be mentioned, comes from the phrase that used to separate Java from other languages back in the days. This phrase is, "write once, run anywhere". Java programs run in their own environment without depending on the underlying platform. Knowing that the final product is always compatible with the user's system is a big plus.

Swing, which is the core part of this thesis, has its own characteristics. It is a GUI toolkit based on Java's first GUI toolkit, named abstract window toolkit (AWT). AWT classes were originally implemented by accessing the native AWT classes (Java calls them peer classes) of the host operating system [Gris98]. That was standing against Java's motto "run anywhere" which is mentioned above. Even if the code was able to run anywhere, the result on the glass was not the same between operating systems and the native calls were expensive in terms of performance. Swing is called the successor of AWT. It introduced independence from the underlying environment and the graphical interface looks the same between all platforms. Its widgets (also called components) are considered lightweight since they do not rely on native calls. In addition, Swing supports pluggable and easy-to-install (but hard to develop) look and feel(s). All these Swing features are related to the visual representation of the GUI. However, this thesis has nothing to do with the graphics of a Swing application.

Two decades later, and after the release of JavaFX in 2008, Swing has been left behind. For the last few years, it is in maintenance and does not get any new features nor any

new releases. However, in comparison to JavaFX which is not part of the Java development kit (JDK) anymore, Swing is still there under the `javax.desktop` module.

The objective of this thesis is the development of a framework. A framework that:

- Imposes a clear separation between the GUI logic and the business logic with a variation of the Model-View-Controller (MVC) (see chapter 2). This MVC variation allows the developer to split a view object into smaller view objects in order to avoid God objects [BDV+06].
- Uses dependency injection design pattern (see chapter 2) for further decoupling between the three MVC layers.
- Makes the code simpler, more concise and easier to read with Java annotations.

1.2 Other Swing tools & libraries

As already mentioned, Swing is a framework that made its first debut more than twenty years ago. During these years, a lot of tools & libraries were built to work with Swing. In this chapter the most used tools and probably those with the most impact on the industry are described.

1.2.1 SWT

The abbreviation originated from Standard Widget Toolkit. SWT is another Java GUI toolkit that stands between AWT and Swing and comes with its own widgets. It is an open source project, and has Eclipse Foundation as its maintainer. According to Eclipse Foundation [SWTw20], "SWT and Swing are different tools that were built with different goals in mind. The purpose of SWT is to provide a common API for accessing native widgets across a spectrum of platforms. The primary design goals are high performance, native look and feel, and deep platform integration. Swing, on the other hand, is designed to allow for a highly customizable look and feel that is common across all platforms.". Interestingly, Eclipse IDE (Integrated Development Environment) which is one of the most popular Java IDEs uses SWT for its GUI.

1.2.2 SwingLabs

SwingLabs became popular with its sub-project called SwingX which introduced more capable and complex widgets that are commonly required in a desktop (and not only) GUI. Those that had the most interest were the auto-complete text fields, the collapsable panels and the date picker. SwingLabs was essentially an open source project that provided extensions to Swing where some of them were integrated into the core Swing. Swing's java.awt.Desktop and java.awt.SystemTray APIs were adopted from the SwingLabs project. Today, the SwingLabs project seems to be abandoned. Even if its source code can be found in public repositories around the web, nobody maintains it anymore.

1.2.3 JGoodies

JGoodies is a framework which consists of freeware and products. JGoodies targets both Swing and JavaFX. It does not only provide features that are related to the UI, but also features that are related with the structure of the source code such as data binding and validation.

1.2.4 Window Builders

Window builders is a family of tools that were built targeting to help the development of the user interface. Most of them are providing drag n' drop functionality of the available widgets to a static window. This window has the visual representation of the window that will be shown at runtime. Everytime the user (developer) adds (drags) a widget, the required source code is generated. However, even if they help the developer to see instantly how the GUI will be shown at runtime, they are essentially code generators. Consequently, they will always be limited and often they will generate code that is not required [ZVM+18]. World's most used Java IDEs provide such tools for Swing by default or via plugins. For instance, the Eclipse IDE has the WindowBuilder plugin and the Netbeans IDE has the Netbeans GUI designer.

Chapter 2. Background

2.1 Model - View - Whatever patterns

The idea started in 1979 from Trygve Mikkjel Heyerdahl Reenskaug when he formulated the **Model-View-Controller** pattern (Figure 1), commonly known as MVC. MVC was implemented in the 80s for the first time for the Smalltalk-80 class library. It is an architectural pattern intended to describe how concerns can be separated in a GUI application. Since MVC was introduced for the first time a lot of variations have been born on it. Some of them are the Model-View-Adapter (MVA), Model-View-Presenter (MVP) and Model-View-ViewModel (MVVM). Even if MVC was originally referred to desktop GUI applications, the idea behind it is adopted from web applications as well.

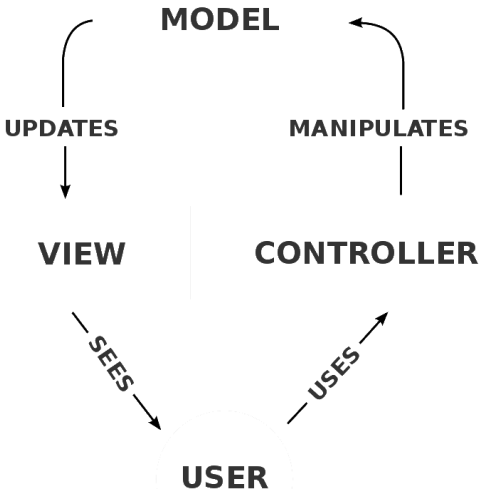

Figure 1. The three elements - layers of MVC

MVC splits concerns into three elements (this thesis uses the term "layers" instead of elements). **The model layer** stands as the central layer of the pattern (and its variations). The model layer addresses concerns related to the domain of the application such as data and logic. **The view layer** refers to anything the user sees on the screen and represents the model in a specific format. As obvious, all the windows and every kind of widget (buttons, lists, icons, etc) are part of the view. **The controller layer** orchestrates operations between the other two layers and is responsible to react upon the user's actions. When the user interacts with a view object, the controller object receives the input (usually observes the view object) and tells the model object to behave. Of course, the responsibilities of each layer can differ according to the MVC variation. Martin Fowler on GUI architectures [Fowl06-UI] states that "*different people reading about MVC in different places take different ideas from it and describe these as 'MVC'*". Nevertheless, all MVC variations preach the same idea. To keep the aforementioned concerns separated.

Analyzing MVC in detail and the differences between its variations is not addressed in this thesis. What is addressed though, is how Swing follows an alternate MVC for its widgets. Each widget contains the controller and view logic combined and some of the widgets have a separate object as their model. Models are separated in two forms. The GUI-state models and data models. For instance:

- A `JButton` widget has a `ButtonUI` delegate object. The `ButtonUI` object contains only view and painting logic. The `JButton` class itself handles both view and controller logic. A `JButton` has a `ButtonModel` object. The `ButtonModel` object is a GUI-state model. For example, the `ButtonModel` has the `isPressed()` method that returns whether the button is currently pressed or not.
- Similarly to the `JButton/ButtonUI` widget, there is the `JTable/TableUI` widget. A `JTable` object has a `TableModel`. The `TableModel` is a data model. The `TableModel` has the `getValue(int row, int column)` method that returns the value of the table cell in the given row and column. In addition, each `JTable` has a `ListSelectionModel` which is a GUI-state model that keeps the selected rows of the table.

Each widget uses listeners (observers) that are being notified when the user interacts with the system. For each different type of interaction, the corresponding type of listeners are notified. These listeners are used to install the actions and the responses of the GUI and often they delegate the call to a controller or a presenter. Snippet 1 shows how an `ActionListener` is installed to a `JButton`. When the `JButton` is pressed by the user, this listener is notified (the `actionPerformed(ActionEvent e)` method is called).

```
saveButton.addActionListener(new ActionListener() {
	@Override
	public void actionPerformed(ActionEvent e) {
		//Save document
	}
});
```

Snippet 1. Handling user's action on a button in Swing

2.2 Inversion of Control & Dependency Injection

Dependency Injection (DI) is a pattern that has been described years ago, but during the latest years has become more popular. Day by day, more and more frameworks are built upon DI. DI is a form of **inversion of control** (IoC). IoC was firstly referenced by Michael Mattsson in the conclusions of his thesis [Matt96], called "Object-Oriented Frameworks". Stefano Mazzochi promoted IoC in a defunct Apache Software Foundation project called Avalon. Afterwards, IoC became popular in 2004 by Robert C. Martin (publicly known as Uncle Bob) and Martin Fowler.

Martin Fowler on Inversion of Control containers and the Dependency Injection pattern [Fowl04] claims that "*inversion of control is a generic term and thus people find it confusing*" and kindly renames it to dependency injection. According to Mazzochi's opinion, Fowler missed the point of IoC and that IoC is about enforcing isolation. Not about injecting dependencies. Mazzochi furtherly claims that "*the need to inject*

dependencies is an effect of the need to increase isolation in order to improve reuse, it is not a primary cause of the pattern". Nevertheless, dependency injection and inversion of control are two terms related to each other.

So, what is DI? DI aims at separating the concern of usage from the concern of creation. An object - **client** that cooperates with other objects in order to behave, instead of creating them itself, it takes them as parameters in a transparent way. These parameters are called **dependencies**. The word **injection** refers to the passing of a dependency into the client. Then, there is the **injetor** object that **injects** (passes) the dependencies to the client object. There are multiple types of DI (constructor, method, field) but this thesis uses only constructor injection for its code snippets.

In snippet 2, there are two different constructors of a class. In the left constructor, the `Thesis` object creates the `Supervisor` object. Consequently, a `Thesis` object is tightly coupled to the `Supervisor` object. For instance, if the process of creation of a `Supervisor` object gets changed, the code of the `Thesis` class must be changed too. In the right constructor, the `Thesis` object does not create the `Supervisor` object. Instead, it depends on it. As a result the two objects are loosely coupled.

```
// Constructor without DI
public Thesis() {
      this.supervisor = new Supervisor();
}
```

```
// Constructor with DI
public Thesis(Supervisor supervisor) {
      this.supervisor = supervisor;
}
```

Snippet 2. An object creates its collaborators (left) - An object depends on its collaborators (right)

Dependency injection also results in more testability. At any time, the dependencies of a client object can be swapped to fake implementations.

2.2.1 Guice - An Inversion of Control container

When a discussion refers to dependency injection or inversion of control in general, the word "**container**" will be likely involved. An IoC container is a framework that provides a convenient way to manage the dependencies and be responsible for their injection. It is essentially a configuration map containing bindings of interfaces and abstractions to concrete implementations. There are numerous IoC containers out there and perhaps for every object oriented programming language. In Java, some of them are PicoContainer, HiveMind, Google's Guice (pronounced 'juice') and of course the admirable Spring Framework every person in the Java world has heard of and probably used.

The framework that is being introduced in the context of this thesis uses Guice as an IoC container. Guice is an implementation of the dependency injection specification for Java (Java Specification Request (JSR) - 330). The specification defines a standard set of annotations and one interface for use on injectable classes. According to Guice, "*Guice alleviates the need for factories and the use of `new` in your Java code. Think of Guice's `@Inject` as the `new` `new`. You will still need to write factories in some cases, but your code will not depend directly on them. Your code will be easier to change, unit test and reuse in other contexts.*"

Guice takes as an input binding configuration and creates a guice.Injector object. Binding configuration (how dependencies are resolved) in Guice takes place within the so called Guice modules. A Guice module is a Java class without logic. A Guice module contains only bindings between types.

As an example, and to stay in the context of snippet 2, let's assume that a `Thesis` object depends on a `Supervisor` object. Let's also assume that `Supervisor` is an interface. In order to create a `Thesis` object with Guice, we would have the code of snippet 3. The `Supervisor` dependency of the `Thesis` object is resolved and injected by Guice behind the scenes.

```
public class GuiceSnippet {

	static class ThesisModule extends AbstractModule {
		@Override
		protected void configure() {
			bind(Thesis.class);
			bind(Supervisor.class).to(SupervisorImpl.class);
		}
	}

	static class Thesis {
		private Supervisor supervisor;

		@Inject
		Thesis(Supervisor supervisor) {
			this.supervisor = supervisor;
		}
		//...
	}

	static interface Supervisor {/*...*/}

	static class SupervisorImpl implements Supervisor {/*...*/}

	public static void main(String[] args) {
		Injector injector = Guice.createInjector(new ThesisModule());
		Thesis thesis = injector.getInstance(Thesis.class);
		//...
	}
}
```

Snippet 3. DI with Guice - Basics

Chapter 3. A Motivational Example

The discussions and issues addressed in this thesis are example-driven. This chapter describes an application developed with Swing. This application, its concepts, requirements and use cases are used as the basis for the examples that are given in the following chapters.

The application is basically a document editor that supports spell checking and correcting, and each document has its authors. Initially, the editor starts with a new and empty document that has no authors. While editing, the user can press a button and the contents of the document get saved as a `thesis.txt` file in his desktop directory. The user is also able to press a button in order to add an author to this document. An author is characterized by a first name, a last name and the date he or she was added to the document. When the user presses the "add author" button, a dialog pops up to get the user input about the first name and the last name of the author. After the dialog's confirmation, the current date is set as the author's added date. As obvious, the user is also capable of deleting one of the document's authors. Then, there is a spelling-related feature that results in misspelled words highlighting and correcting. The spelling feature can either be enabled or disabled by the user. The moment the user enables the spell checking option, the misspelled words in the editor are highlighted. If the user disables the functionality, all highlights of the text are instantly removed and the user is unable to correct all misspellings of the document.

The main area of the user interface consists of the document contents within a JTextArea widget called `contentsArea.` On top of the `contentsArea`, there are the three buttons that affect the document. The `saveButton` that when pressed, the document is saved to the user's hard disk. The `addAuthorButton` which allows the user to add an author to the document via a "input and confirm" dialog and the `correctMisspellsButton` which is enabled only if the spell checking feature is enabled. When the user presses the `correctMisspellsButon`, all misspelled words of the document are validated and the corrected text appears on the editor. On the right side of the window, the spell checking options exist. For the sake of this example's simplicity, these options are completed only by the on/off option. For changing the on/off property of the spell checking mechanism, a `JCheckBox` named `spellCheckingCheckBox` is used. All the authors, their first name, last name and date that were added to the document are shown in the bottom area of the window. Finally, next to an author's information, there is a button referred to as `deleteAuthorButton` and when pressed by the user, the author is removed from the document. Figure 1 shows how the application user interface looks initially at startup.

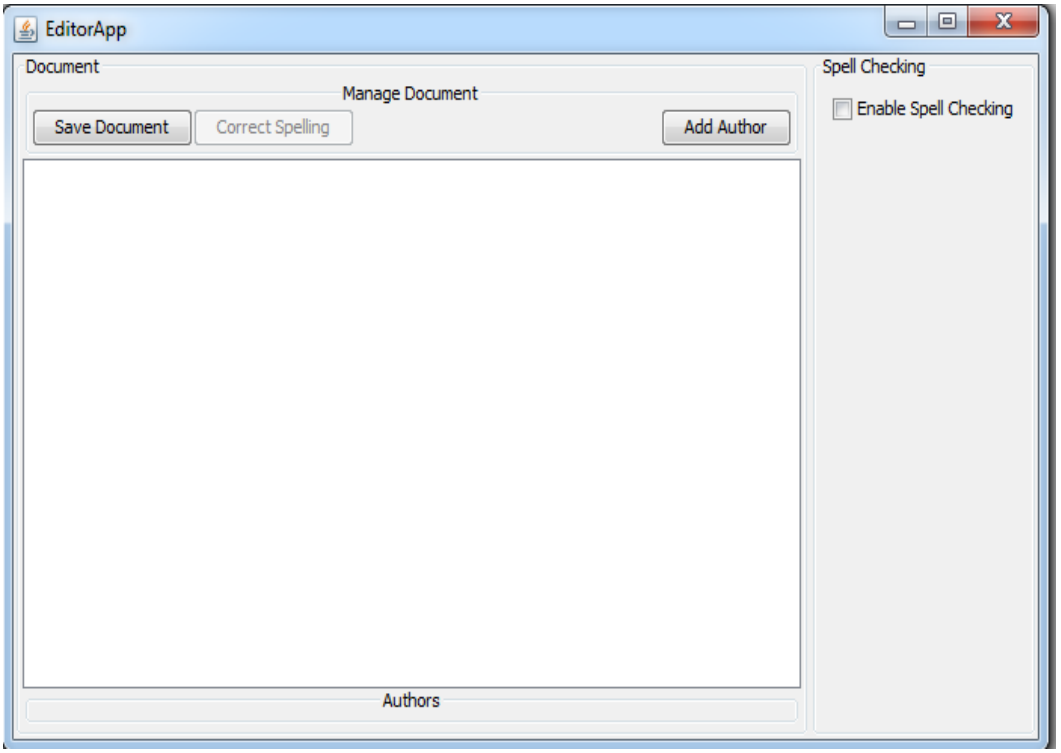

Figure 1. The user interface of the sample application at startup

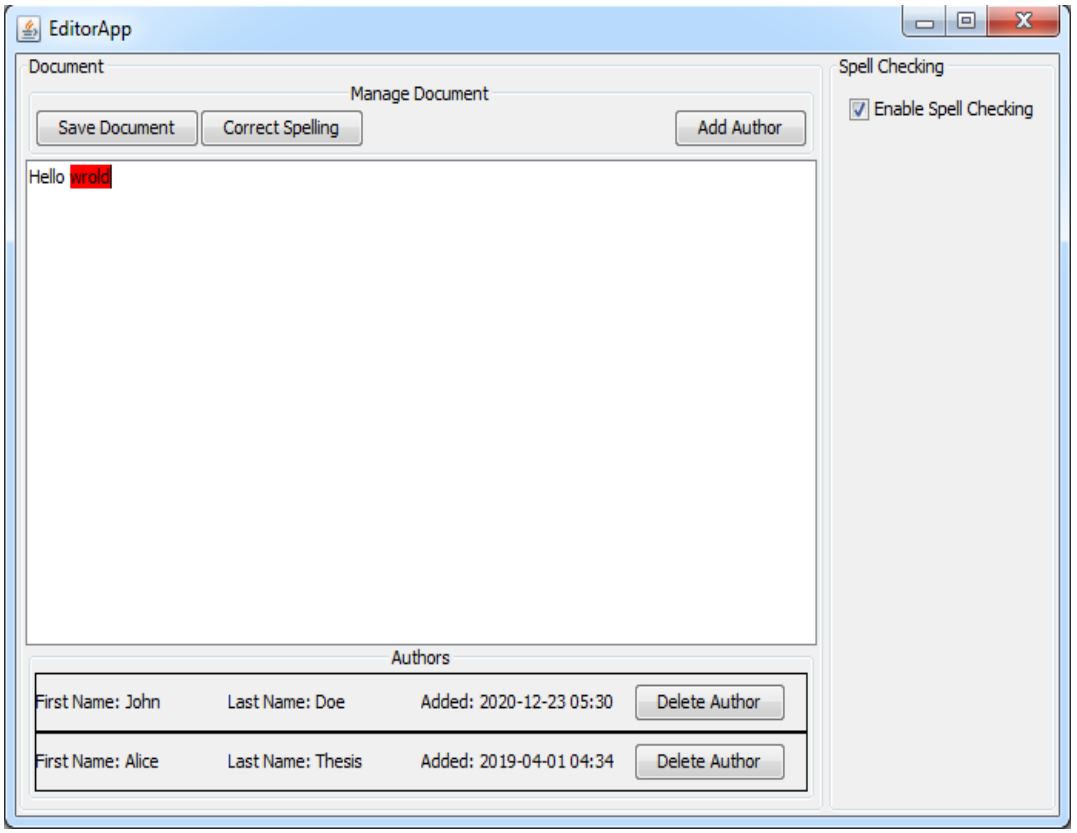

Figure 2. The user interface of the sample application with authors and spelling feature activated

Figure 2 shows the user interface after the `spellCheckingCheckBox` checkbox is ticked and after the addition of two authors. Besides the obvious difference in the visual area related to the authors, the `correctMisspellsButton` button is now enabled since the spell checking feature is activated. The word "wrold" in the text area is highlighted because it is misspelled.

Given the above features, use cases and user interface of the application, the classes `Document`, `Author`, `DocumentService` and `SpellingService` are defined in the domain model. `Document` class represents a document and it consists of a `String` property called `contents` and a collection of authors. Author class represents an author and it has two `String` properties, the first and the last name of the author and a `Date` property holding the date the author was added to the document. The `DocumentService` is a class responsible for saving the document. `DocumentService` is a separate (from `Document`) class because it handles weird file operations such as writing, file access locks, file paths etc and writing the text file in a specific format. It is assumed that the `saveDocument` operation of `DocumentService` is something heavy in terms of time it requires to complete. For example, the size of the document's contents might be large or the hard disk of the user's system is (busy) slow. Not a typical real-world example, but the point is clear. The `SpellingService` is a class responsible for detecting and correcting misspells. This class has its own (unrelated to this thesis) dependencies such as dictionaries and token analyzers. The reason for the existence of the `SpellingService` class is its `checkSpelling()` and `correctSpelling()` methods. The `checkSpelling()` method takes a string value as parameter and returns a collection of `MisspelledWord` objects. `MisspelledWord` class is a holder of two integer values. An offset and a length. Each offset and length define where exactly the misspell word exists in the given text. As shown in Figure 2, if the String "Hello wrold" passes to `checkSpelling()` method, the result collection contains a `MisspelledWord` with offset=6 ("wrold" starts there) and length=5 (word's length). It is assumed that the smaller in terms of length the given string to `checkSpelling()` method is, the faster the result with the `MisspelledWords` is calculated. The other method of `SpellingService`, the `correctSpelling()`, accepts a String value as well. The method analyzes the given text and returns it with the potential corrections. For example, if the invocation is `spellingService.correctSpelling("Hello wrold")`, the method returns the `String` "Hello world". Finally, the `SpellingService` has the `isEnabled()` method that returns whether the service is enabled and capable of doing spelling validations and corrections. A pseudo-UML class diagram of the domain classes of the application is shown in Figure 3.

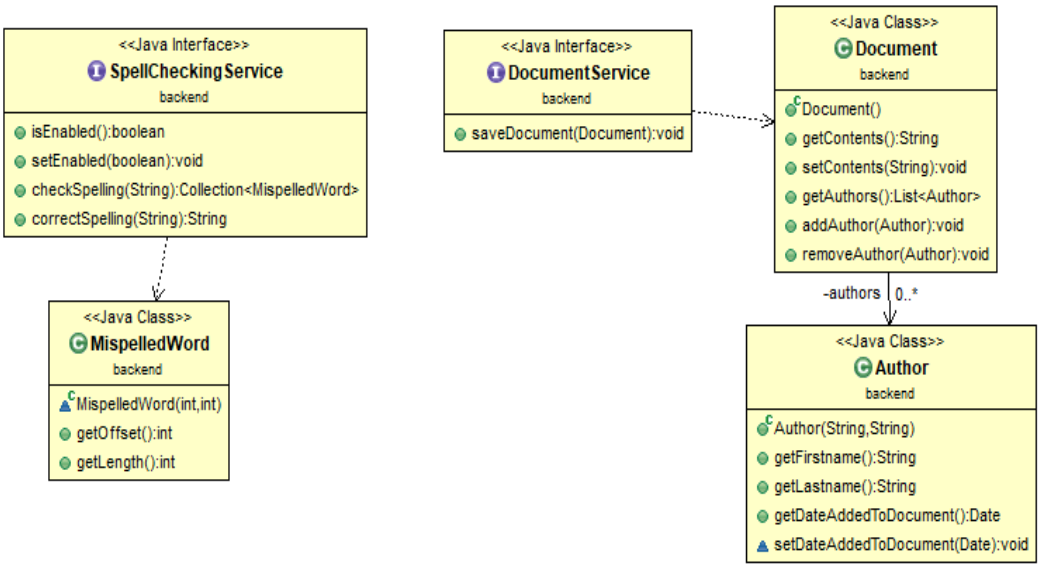

Figure 3. UML class diagram of application's backend

Chapter 4. Meanwhile in Swing

4.1 Architectural issues

Swing is a GUI toolkit for an object oriented programming language. Every aspect of Swing is represented by objects. Every widget of the toolkit is a view itself and follows MVC with its own model (state or data) and is represented by an object. A widget accepts controllers, the so-called listeners as observers that are notified when an event takes place upon that widget. This section addresses what usually happens between the controller and the view MVC layers in an application developed with Swing and the issues that possibly take place. The sample application described in the previous chapter is used for this chapter's examples.

Widgets alone are not able to describe the GUI and the meaning of each visual area that exists in the window of the application. That's why views are often described under more general terms. For example, it is not convenient to describe the document editor application by the `contentsArea` and the `saveButton` widgets because they exist in the window. A developer does not say to his coworker "when the `saveButton` is pressed, ...". The developer says "when the document gets saved, …". It is preferred that the window will be described by its own view class e.g `EditorAppView`. This `EditorAppView` takes its own meaning and possibly hides (composes) the details (widgets) of the view.

A common practice in Swing is that this kind of classes (`EditorAppView`), classes that compose multiple widgets and describe the bigger picture of the GUI, extend some widget class of the toolkit. So, the class the `EditorAppView extends JFrame` where `JFrame` is a top level container. All Swing applications have one and since it is a heavyweight container-widget (it communicates with the underlying operating system), an application should create only one. This "extending a widget" practice is used in the examples of this thesis but is kindly irrelevant to its subject. Snippet 4 demonstrates how the `EditorAppView` class could possibly stand.

```
class EditorAppView extends JFrame {

	JButton saveButton;
	JButton correctSpellingButton;
	JButton addAuthorButton;

	JTextArea contentsArea;

	JPanel[] authorsPanels;
	JButton[] deleteAuthorsButtons;
	JCheckBox spellCheckCheckBox;


	EditorApp(){
		...
		//init and add all widgets
	}
}
```

Snippet 4. All widgets are held under a class that describes the whole window. This class extends a Swing class.

But now the `EditorAppView` is defined as a (parent) view and holds all the widgets of the application, the question is how the system responds to the user's actions. Speaking of MVC, who or what is the controller of the view and what does it control?

4.1.1 (Not so) Smart UI - God views

One convenient and common approach programmers use for GUI applications development, is what Eric Evans in Domain-Driven Design: Tackling Complexity in the Heart of Software [Evan03] describes as the Smart UI "anti-pattern". In a smart UI approach, some classes (if not all of them) defined as the model such as `DocumentService` and `SpellingService` might be absent. The business logic such as the saving of a file with the contents of the document and the spell checking of a text are mixed with the GUI and everything (creation of objects, initialization of widgets, registering listeners to widgets) happen within the same class. For example, the `EditorAppView` class (as defined and shown above) is likely to register an `ActionListener` on the `saveButton` that looks like the code that exists in snippet 5.

```
saveButton.addActionListener(new ActionListener() {
	@Override
	public void actionPerformed(ActionEvent e) {
		File txtFile = new File(desktopDirectory, "thesis.txt");
		for (Author author : getDocument().getAuthors())
		{
			writeFormattedLineToFile(txtFile,
					"Author[%s,%s,%s]",
					author.getFirstName(),
					author.getLastName(),
					author.getDateAdded());
		}
		writeTextToFile(getDocument().getContents());
		JOptionPane.showMessageDialog(EditorAppView.this, "File saved successfully!");
	}
});
```

Snippet 5. Smart UI in approach in Swing

In the smart UI approach there is no MVC besides the widget-level [Kara08] MVC Swing offers and business logic is mixed with the GUI logic. The advantages and disadvantages of this approach are referred to by Evans but the result is that this approach is able to deal only with small scale applications where few use cases exist. In Swing, the window builders (see section 1.2.4) make it very tempting for a new-to-Swing programmer to use the Smart UI pattern [ZVM+18]. Window builders allow the programmer to add a listener to the button without touching the code directly. When these builders add the code for the listener, they often create a new (empty) method and the added listener delegates the callback to this new method. The programmer is happy and tempted to fill this method with a mix of business logic and GUI logic, simply because in the smart UI approach the programmer has access to every aspect of the application. In Swing, the smart UI solution is problematic on a bigger scale though. Yes, some of the advantages of a smart UI (e.g as the easy access to parts of the domain) are present, but one day if the application requires to "grow", everything will fall apart.

When a programmer uses a GUI toolkit, he has some demands. One of these demands is the support for resizable windows and windows capable of being shown in different screen dimensions. What if the programmer has a 1600x900 screen, but the user has a 1300x700 screen? Swing in order to provide this functionality offers the so-called LayoutManagers. There is a variety of layout managers and each one has its own widget placing rules. Snippet 6 is taken from Swing's official documentation as is and demonstrates the placing of a button using the "most flexible" layout manager, called `GridBagLayout`. The variable named "c" is a `GridBagConstraints` object which is basically the settings of the placing.

```
button = new JButton("Long-Named Button 4");
c.fill = GridBagConstraints.HORIZONTAL;
c.ipady = 40;      //make this component tall
c.weightx = 0.0;
c.gridwidth = 3;
c.gridx = 0;
c.gridy = 1;
pane.add(button, c);
```

Snippet 6. Placing a widget with complex UI requirements using GridBagLayout

The piece of code that exists in snippet 7 is something that often takes place in a Swing UI and it is indirectly related to the smart UI pattern. If the placing of a single widget requires (give or take) seven lines, how many lines of code exist in the `EditorAppView` class when a smart UI approach is taken? **The class ends up being a God** [BDV+06] **class. The code is hard to follow and of course the testing of the system and its units is unfeasible.**

4.1.2 Beyond widget-level MVC - God Controllers

Besides container-widgets, every other standard widget Swing offers follows the model view controller pattern and Alexandros Karagasidis in Developing GUI Applications: Architectural Patterns Revisited [Kara08] refers to it as widget-level MVC. A frequent, yet reasonable case in Swing applications is to extend the widget-level MVC and transform it to container-level MVC. So, for the `EditorAppView`, an `EditorAppViewController` is created. The `EditorAppViewController` is responsible for handling every action that takes place in the `EditorAppView`. When the user interacts with the system, Swing as the underlying GUI toolkit creates event objects based on the type of the interaction. For example, the toolkit interprets mouse movement to a `MouseEvent` and the pressure of a key to a `KeyEvent`. The moment these event objects are created, the method(s) of the corresponding listener are called and fed by these events. When the mouse event is created, all mouse listeners of the widget are triggered and have access to this event. The introduction of `EditorAppViewController` as a new class raises the following question: how is `EditorAppViewController` able to listen to these events? Since it is Swing and its widgets that recognizes them, the controller must meet the toolkit's specifications somehow.

If for example, the requirement is to listen to mouse and action events and respond upon them, the `EditorAppViewController` must be (implement) a `MouseListener` and an `ActionListener` in order to be installed into the widget(s). Snippet 7 shows a possible

definition of the `EditorAppViewController` in order to be installable to Swing widgets. The if-statement that exists in `actionPerformed()` is explained after snippet 8.

```
class EditorAppViewController implements ActionListener, MouseListener {
	EditorAppViewController(DocumentService docService,
				SpellingService spellService)
	{...}

	@Override
	public void actionPerformed(ActionEvent event) {
		if (event.getActionCommand().equals("save_document")) {
			//Save document
		}
		else if(...) {}
		else if(...) {}
	}


	public void mouseClicked(MouseEvent e) {
		//Do potential single-word corrections if clicked upon a misspelle
	}

	public void mousePressed(MouseEvent e) {		}
	public void mouseReleased(MouseEvent e) {		}

	public void mouseEntered(MouseEvent e) {		}
	@Override
	public void mouseExited(MouseEvent e) {	}
}
```

Snippet 7. EditorAppViewController class to match toolkit's specifications and be installable to widgets

Now the `EditorAppViewController` is installable to the widgets and `saveDocumentButton.addActionListener(editorAppViewController)` is a valid line. So the question, "how the controller is able to listen to Swing events" derives to: where is this controller getting installed to the widgets. Karagasidis faintly points out two approaches. Based on the first approach, the `EditorAppView` has a `registerController(EditorAppViewController controller)` method and the installation happens there. This approach is demonstrated in snippet 8.

```
class EditorAppView() {
	EditorAppView() {
		...
	}

	void registerController(EditorAppViewController controller) {
		this.saveButton.setActionCommand("save_document");
		this.saveButton.addActionListener(controller);
		this.addAuthorButton.setActionCommand("add_author");
		this.addAuthorButton.addActionListener(controller);
		...
	}
}
```

Snippet 8. EditorAppViewController is installed to widgets in a method declared in EditorAppView

The `(x)Button.setActionCommand(..)` is required in order the `EditorAppViewController` be able to recognize which button was pressed by the user. It is directly related to snippet 8 and the if-statement that exists in the `actionPerformed()` method. For every action that takes place in the view, an "if" condition in the controller has to fulfil it. Things are even worse when the controller's response is based on other types of events. Assuming another use case, the user is able to correct the spelling of a single misspelled word by pressing a right click upon it (inside the `contentsArea`) and mouse events are generated. However, the `contentsArea` widget is not the only widget of the view with interesting mouse events. MouseEvents do not have a `.getActionCommand()` method hence the detection of the publisher (the widget that recognized the event) and the "if" condition have to rely on the `getSource()` method every Swing event object has. Thus, the `EditorAppView` object must expose access to its widgets. Snippet 9 shows how the `mouseClicked()` method of the controller looks like.

```
public void mouseClicked(MouseEvent event) {
	Object source = event.getSource();
	if (source == editorAppView.getContentsArea()){
		//Do single misspelled word correction
	}
	else if (source == editorAppView.getAnotherWidgetWithInterestingClicks()) {
		//...
	}
}
```

Snippet 9. Detecting the event source requires EditorAppView expose access to widgets

These "if" statements inside the methods that are inherited from listener interfaces are hard to maintain and read, and can become extremely chaotic if the view-controller responds to numerous events.

Another approach, yet often applied in Swing applications, is to let the `EditorAppViewController` register the listeners without itself implementing Swing listener interfaces. Then, these verbose "if" statements can be avoided and for each control (response) of the view, a new, isolated listener is registered to the widgets using a Java inner class. Again, all the widgets of the view must be exposed in order to be accessed from the `EditorAppViewController`. The most common phenomena is to delegate the callback to another method of the controller. A sample of this approach is given in snippet 10.

```
class EditorAppViewController {
	EditorAppView view;
	...

	void installListeners() {
		view.getSaveButton().addActionListener(new ActionListener() {
			@Override
			public void actionPerformed(ActionEvent event) {
				saveDocument();
			}
		});
		view.getAddAuthorButton().addActionListener(new ActionListener() {

			@Override
			public void actionPerformed(ActionEvent event) {
				addAuthor();
			}
		})
		...
	}
	void saveDocument() {
		...
	}
	void addAuthor() {
		...
	}
}
```

Snippet 10. EditorAppViewController install the listeners to the widgets and delegates the call to one of its methods

Of course, the installation of the listeners with Java inner classes can also take place in the `registerController()` method of the `EditorAppView` as shown in the snippet 8. If so, the `EditorAppViewController` is the one who has to expose its methods in order to be called from the inner classes. This approach is shown in snippet 11.

```
class EditorAppView()
{

	EditorAppView() {
		...
	}

	void registerController(EditorAppViewController controller) {
		this.saveButton.addActionListener(new ActionListener() {
			@Override
			public void actionPerformed(ActionEvent event) {
				controller.saveDocunet();
			}
		});
		this.addAuthorButton.addActionListener(new ActionListener() {
			@Override
			public void actionPerformed(ActionEvent event) {
				controller.addAuthor();
			}
		});
		...
	}
}
```

Snippet 11. Swing listeners are added as inner classes in EditorAppView the callbacks are delegated to EditorAppViewController methods

4.1.3 Beyond container-level MVC - God hierarchies

Both approaches that are described in the above section are theoretically correct and valid. These approaches respect MVC and the user interface (windows and widgets) of the application is separated from the controlling. Every response of the GUI is handled by the `EditorAppViewController` and every widget is created and shown by the `EditorAppView`. But one question is still present. What if the `EditorAppView` contains numerous widgets and each one of these widgets must respond to user actions? Among an `EditorAppView` class creating and layouting all these widgets, the `EditorAppViewController` class has to handle all the actions. The result is that the controller ends up being a class that potentially consists of thousands of lines of code, making it hard to debug and maintain.

We (programmers) often break a system or a component into smaller pieces when it starts to get bigger. Bigger in terms of complexity. Bigger in terms of lines of code. Bigger in terms of the number of the other components it depends on and collaborates with. That could be characterized as one of the oldest and most standard principles in programming (and not only), and yes, we tend to do it. From extracting methods to smaller methods, classes to smaller classes, systems to smaller subsystems. Simply because small is beautiful [Henn16]. Small provides testability and no matter the complexity of the context, small can be understanded and debugged with less effort.

And that's why someone considers MVC (or any of its variations) in its hierarchical form. Hierarchical MVC is also known as HMVC [CKP00] and it is pretty similar to presentation-abstraction-control pattern (PAC) [Cout87] but in the context of this

section their differences are not addressed. In HMVC, instead of one MVC triad, there is a hierarchy of them. In the context of the sample application, the `EditorAppView` can be separated to `DocumentView` and `SpellCheckingOptionsView` and then these two are assembled by the `EditorAppView`. Each one of these views have their own controller. For example, `SpellCheckingOptionsView` has the `SpellCheckingOptionsViewController`, etc. HMVC comes with a big advantage. Each visual area of the GUI can take its own meaning and is isolated with its own model. For instance, the model of `DocumentView` is a `Document`. When a controller is not capable of recognizing an event, the controller passes the event to its parent controller. The communication between layers and between MVC triads of the same level takes place only within the controllers. When the spell checking on/off state changes (`spellCheckingCheckBox` pressed), in order to update the document view and instantly remove all highlighted words, the `SpellCheckingOptionsViewController` will give the event to its parent, `EditorAppViewController`. Then the `EditorAppViewController` will inform the `DocumentViewController` to make the change. One drawback of HMVC is the need to construct the parent MVC triad during testing. Testing the on/off spell checking option will probably require the `EditorAppViewController` (triad) in order to handle the event, even if the `EditorAppView` itself is not affected by the change. Ideally, the testing of the unit would require to construct the `DocumentViewController`, `DocumentView`, `SpellCheckingOptionsViewController` and `SpellCheckingOptionsView`. Simply because these are the only classes involved in the result of the action. Of course, when there are two layers-levels of hierarchy this does not seem like a problem. But if there are more layers (in the hierarchy) of MVC triads, the testing of those can be really frustrating.

4.2 Concurrency issues

Every programmer that ever used Swing probably faced a “frozen” GUI in his/her life. Frozen by the terms of not responding to any kind actions. Not even those actions that are fed by the OS such as minimizing/maximizing and closing the window (`JFrame`). This is a concurrency and threading related issue. Swing is not a thread-safe toolkit. The documentation states that all Swing applications must run on their own thread and this thread is named the **Event Dispatch Thread** (EDT). Every event and all the painting (and initialization) of the widgets of the application should take place within EDT. It is known that all Java applications start from a `public static void main()` method and this method is executed in a thread called main-thread. Typically, when a Swing application starts, the only thing that happens in the main thread, is to start the initialization of the GUI in the EDT. This can be done with the `SwingUtilities.invokeLater()` method as shown in snippet 12.

```
public static void main(String[] args) {
	SwingUtilities.invokeLater(new Runnable() {
		@Override
		public void run() {
			SwingApp app = new SwingApp();
			app.getJFrame().setVisible(true);
		}
	});
}
```

Snippet 12. As soon as the Java program starts, the Swing application is created in the EDT.

But when a heavy (long running) task starts running inside the EDT, the whole application is frozen and all interaction is unable to happen. That is because the EDT is busy and events cannot be handled. Therefore all events are queued. In the sample application, it is assumed that the `documentService.saveDocument(Document d)` is a method that needs time to be completed. Ignoring the architectural related controversies, if the button is pressed with an `ActionListener` as shown in snippet 13, the GUI will freeze until the document is saved.

```
saveDocumentButton.addActionListener(new ActionListener() {

	@Override
	public void actionPerformed(ActionEvent e) {
		documentService.saveDocument(getDocument());
		JOptionPane.showMessageDialog(null, "Document saved.");
	}
});
```

Snippet 13. GUI freezes if heavy task executed within EDT

The solution is to execute the heavy task in another (background) thread. Using a background thread for heavy tasks does not block the EDT. So, the first thing that comes to a programmer's mind is to use the typical Java way, also known as the `Thread` class among a `Runnable`. So, the code of snippet 13, becomes now the code that exists in snippet 14.

```
saveDocumentButton.addActionListener(new ActionListener() {
	@Override
	public void actionPerformed(ActionEvent e) {
		Thread backgroundThread = new Thread(new Runnable() {

			@Override
			public void run() {
				documentService.saveDocument(getDocument());
				JOptionPane.showMessageDialog(null, "Document saved.");
			}
		});
		backgroundThread.start();
	}
});
```

Snippet 14. The application creates a background thread for a heavy task

Now the issue is that Swing code (`JOptionPane…`) that involves widgets and painting is executed outside the EDT. That's why the background thread must return to the EDT in order for this Swing code to run in the EDT. In other words, the `ActionListener` should look like the code of snippet 15.

```
saveDocumentButton.addActionListener(new ActionListener() {
	@Override
	public void actionPerformed(ActionEvent e) {
		Thread backgroundThread = new Thread(new Runnable() {

			@Override
			public void run() {
				documentService.saveDocument(getDocument());
				SwingUtilities.invokeLater(new Runnable() {

					@Override
					public void run() {
						JOptionPane.showMessageDialog(null, "Document saved.");
					}
				});
			}
		});
		backgroundThread.start();
	}
});
```

Snippet 15. The background thread returns to the EDT for Swing code execution

As shown in snippet 14 and snippet 15, thread and concurrency in general can be tough in a Swing environment and they become more complex if publishing the progress of the background task is a requirement. That's why Swing offers its own API, the so-called `SwingWorker` API. SwingWorkers are based on Java's `Future` class. They calculate a value in a background thread, and then Swing related code is executed inside the EDT. Since the `saveDocument` method is void, the result of the future is `Void`. An example where `SwingWorker` is used (instead of `Thread`) is shown in snippet 16.

```
saveDocumentButton.addActionListener(new ActionListener() {
	@Override
	public void actionPerformed(ActionEvent e) {
		SwingWorker<Void, Void> worker = new SwingWorker<Void, Void>(){

			@Override
			protected Void doInBackground() throws Exception {
				documentService.saveDocument(getDocument());
				return null;
			}
			@Override
			protected void done() {
				//Executed inside EDT
				JOptionPane.showMessageDialog(null, "Document saved.");
			}
		};
		worker.execute();
	}
});
```

Snippet 16. Using SwingWorker instead of Thread

SwingWorkers are able to cover all the scenarios of concurrent executions while keeping the GUI "alive". However, the programmer should always be careful. During the heavy

task execution in the background, exceptions are possible. In order to catch these exceptions, the programmer should call the `get()` method of the worker inside the `done()` method even if there is no result (the result is `Void`). The `get()` method returns the value-result of the future carrying the possible exception thrown in the background. So, the code from snippet 16 should be like the code that exists in snippet 17.

```
saveDocumentButton.addActionListener(new ActionListener() {
	@Override
	public void actionPerformed(ActionEvent e) {
		SwingWorker<Void, Void> worker = new SwingWorker<Void, Void>(){

			@Override
			protected Void doInBackground() throws Exception {
				documentService.saveDocument(getDocument());
				return null;
			}
			@Override
			protected void done() {
				try {
					Void result = get(); //nothing to do with it
					JOptionPane.showMessageDialog(null, "Document saved.");
				}
				catch(Exception backgroundException)
				{
					JOptionPane.showMessageDialog(null,
							"Document was not saved.");
				}

			}
		};
		worker.execute();
	}
});
```

Snippet 17. Catching the possible exception that occured in background

As shown in the above snippets, the execution of a heavy task in a background thread is not something trivial in a Swing environment and it comes with a lot of boilerplate code. It is even worse when publishing the progress of the background task takes place. Also, the programmer must always be aware of the calls (to components of the business domain). If a "frozen" GUI occurs in a scaled application, **it can be hard to find out where the root of the problem is and which call takes place in a wrong thread**.

4.3 Listeners fired while they shouldn't

One use case of the sample application is to let the user edit the contents of the document and if there are any misspelled words, they get highlighted. The widget that the user writes the contents of the document, is a `JTextArea`. So the task is to call the `spellCheckingService.checkSpelling(contentsArea.getText())` method every time the user makes edits to this `JTextArea`. `JTextArea` extends the `JTextComponent` widget class which uses a `javax.swing.text.Document` (not to be confused with the Document class of the sample application) as a model in order to respect MVC. Swing's documentation suggests the usage of a `javax.swing.event.DocumentListener` in order to detect the changes of the contents. Leaving architectural controversies aside, this

document listener that allows the detection of user's edits and highlighting of the misspelled words may look like the code that exists in Snippet 18.

```
contentsArea.getDocument().addDocumentListener(new DocumentListener() {

	@Override
	public void removeUpdate(DocumentEvent e) {
		userEditedTheTextArea();
	}

	@Override
	public void insertUpdate(DocumentEvent e) {
		userEditedTheTextArea ();
	}

	@Override
	public void changedUpdate(DocumentEvent e) {
		userEditedTheTextArea ();
	}

	private void userEditedTheTextArea () {
		String contents = contentsArea.getText();
		Collection<MisspelledWord> misspells = spellCheckingService.checkSpelling(contents);
		highlightMisspelledWords(misspells);
		//...
	}
});
```

Snippet 18. DocumentListener to detect user edits on the text widget and highlight misspelled words

After the code of snippet 18 is written, misspelled words highlighting is functional. However, the issue appears when the use case about the correction of the misspelled words takes place. The correction of the misspelled words happens when the user presses the `correctMisspellsButton`. So, there is an `ActionListener` somewhere that looks like the code that exists in snippet 19.

```
correctMisspellsButton.addActionListener(new ActionListener() {

	@Override
	public void actionPerformed(ActionEvent e) {
		String misspelledContents = contentsArea.getText();
		String correctContents = spellCheckingService.correctSpelling(misspelledContents);
		contentsArea.setText(correctContents);
	}
});
```

Snippet 19. ActionListener that gives the corrected text to contentsArea

When the `contentsArea.setText(correctContents)` is called, the model (`javax.swing.text.Document`) of the widget is updated. Consequently, the `DocumentListener` that exists in Snippet 18 is fired and the check spelling functionality takes place all over again. But should it be? The model of the widget is changed from the system itself, not from the user. Whether the contents contain misspelled words or not is already handled. Therefore, the triggering of this listener is unwanted here and it can possibly affect performance. In order to prevent the listener from firing, boilerplate code and logic is required. A reasonable solution is to remove the document listener, change the text of the JTextArea and then re-add the listener back as shown in snippet 20.

```
correctMisspellsButton.addActionListener(new ActionListener() {

	@Override
	public void actionPerformed(ActionEvent e) {
		String misspelledContents = contentsArea.getText();
		String correctContents = spellCheckingService.correctSpelling(misspelledContents);

		contentsArea.getDocument().removeDocumentListener(highlightingDocumentListener);
		contentsArea.setText(correctContents);
		contentsArea.getDocument().addDocumentListener(highlightingDocumentListener);
	}
});
```

Snippet 20. DocumentListener is removed, contents change and then the DocumentListener is added again

This solution requires to hold a reference to the `DocumentListener` in a variable or in a field which is more code that possibly exists in the view class. The same happens for other Swing widgets as well. **Swing widgets and their whole MVC triad are not capable of knowing whether the change is coming from the user or from the system itself** and often an application wants to make changes to its view (the `JTextArea`) while the view is passive [Fowl06-PV].

Chapter 5. Swing Boot - What

Swing Boot is the name of the framework developed in the context of this thesis. This chapter explains the main concepts of Swing Boot, what it does and how a developer can use it. The last section of this chapter analyzes the advantages and the disadvantages of (using) the framework. Note that Swing Boot is not yet another library that provides custom widgets or custom layouts. Swing Boot addresses two major concerns of GUI development in Swing. The first concern is **the architecture of the GUI**. The second concern is **concurrency and threading**. For each one of these concerns there is a separate section. In order to showcase the concepts and the features of the framework, the examples of this chapter rely on the sample application which was introduced in chapter 3.

5.1 Architecture: A different kind of MVC where each use case is a control(ler)

An application developed with Swing Boot has a three layer separation in its architecture. As in traditional MVC, the architecture consists of the model, the view and the controller. However, there are some differences in the responsibilities of the classes of each layer. The next three subsections analyze the role of each layer in a Swing Boot application.

5.1.1 Model layer: Same as MVC

The model layer in a Swing Boot application is exactly what the model layer is in traditional MVC. Classes that belong to the model layer contain business logic and data. **Swing Boot does not address the design of the model layer** and the responsibilities of a "model class" are the same as in MVC. The only difference is that objects of the model layer have to be injectable by the IoC container. The reason behind this is because the other two layers have to be injectable as well (see next sections).

In the sample application, the model layer includes all the classes that exist in the UML class diagram of figure 3.

5.1.2 Control layer: Instead of a controller for the whole view, isolated controls for each use case

In MVC, classes that belong to the controller layer are responsible for reacting to the user's input, coordinating the flow of the GUI and standing as mediators between the view layer and the model layer. In traditional MVC, there is a one-to-one relationship between a view object and a controller. In a Swing Boot application the control layer is different. In a Swing Boot application there is one controller for each use case. These controllers are named **controls**. Keep in mind that since we are in the context of object-oriented programming, this section uses the terms "a control" and "a control object" as synonyms.

So, what is a control? As a concept, **a control is a use case handler**. When the user gives input to the system (e.g a button was clicked) a control object (instance of a control class) is responsible to handle this input. Typically, for each use case there is a control class.

From an implementation standpoint, each control class must **implement the swingboot.Control<T> interface** and its **void perform(T parameter)** method. Control objects can be considered as commands that the command pattern [GHJV94] describes. The implementation is slightly different from a typical command pattern because there is a generic parameter (command pattern does involve parameters). However, the motivation is the same. The executor of the command (view objects - widgets) does not need to know anything at all about what the command is, what context information it needs on or what it does. The `perform(T parameter)` method corresponds to the `execute()` method of the command pattern.

Going back to our motivating example, some of the controls that exist in the sample application are the following (notice the similarity between the use cases of the application and the names of the control classes):

- SaveDocumentControl
- HighlightMisspelledWordsControl
- CorrectSpellingControl
- AddAuthorControl
- DeleteAuthorControl
- etc

A control's responsibility is to stand between the model and the view. Whenever a use case is about to take place upon a user action, the control object behaves as follows:

1. Asks view objects for GUI state - What did the user change (if anything) - e.g the user edited a JTextField.
2. Requests objects that belong to the model layer to change their state and/or for results - Translates GUI state to record state [Fowl06-UI].
3. Tells view objects to update their state accordingly (probably based on values came back from "model objects")

As an example, snippet 21 shows the implementation of HighlightMisspelledWordsControl of our application.

```
class HighlightMisspelledWords implements Control<Void> {

	@Inject
	HighlightMisspelledWords(SpellCheckingService spellingService,
	                         DocumentView documentView)
	{
		//constructor args to fields
	}

	public void perform(Void parameter) {
		String contents = documentView.getContents();
		List<MisspelledWord> misspells = spellingService.checkSpelling(contents);
		for (MisspelledWord word : misspells) {
			documentView.highlightWord(word.getOffset(), word.getLength());
		}
	}
}
```

Snippet 21. HighlightMisspelledWordsControl - An overview of how a control class looks like

There are a couple of things that should be furtherly explained in snippet 21.

- First and more importantly, the `@Inject` annotation in the constructor of the class. The annotation must be present to **all control classes since all control objects are injected** (to Swing Boot internals) **by the IoC container** when the use case is about to start. Therefore, all control classes must be visible (bonded) to the IoC container in order for the injection to happen.
- Secondly, the parameter is not used at all. Thus, the generic type used is `Void`. Typically, most controls do not need to rely on the parameter. The parameter exists only to cover edge cases. Section 5.2.6 explains when a non-`Void` parameter can (or must) be passed. For the rest of the sections of this chapter the parameter is completely ignored.
- Thirdly, there are no getters and setters in the control class. A control is not meant to hold mutable state. All the mutable state of the application is held by objects of the view layer and the model layer. They ask for their dependencies in their constructor and then remain immutable [HPSS06].

5.1.2 View layer: Same responsibilities as in MVC, but in a different way

The view layer of a Swing Boot application resembles the view layer of a traditional MVC application. Just like MVC in its hierarchical form (HMVC pattern - see section 4.1.3), each view can be splitted to a number of different subviews. Objects that belong to the view layer have six responsibilities:

1. Initialize the Swing widgets. This includes the construction of the widgets and the initiation of all of their initial properties such as static text, fonts, icons and tooltips.
2. Do the layout-ing of these widgets which as shown in chapter 4 is likely a verbose task in a Swing application.
3. Be injectable (to Swing Boot internals) by the IoC Container (bonded to a Guice module).

4. Provide access methods to controls for values related to GUI state. This is mandatory since a control object can both ask and update values of a view object. There are basically two approaches (both can be combined and used at the same time):
    - The view object has getters for its widgets. In this approach, the control object has access to the widgets and directly manipulates the widgets.
    - The view object has a getter and a setter method for each dynamic value rendered in the screen. Then the control object uses these methods. In this approach each one of these methods of the view object, delegates the screen updates to its widgets.

    As an example, Figure 4 shows in UML side by side the two approaches using the class DocumentView. On the left side, there is the "getters to widgets" approach, while on the right side of the figure there is the "getters and setters for values" approach. Note that in both approaches, there are no getters for the buttons. That's because the buttons are not changed (their properties - e.g text, icon) dynamically.

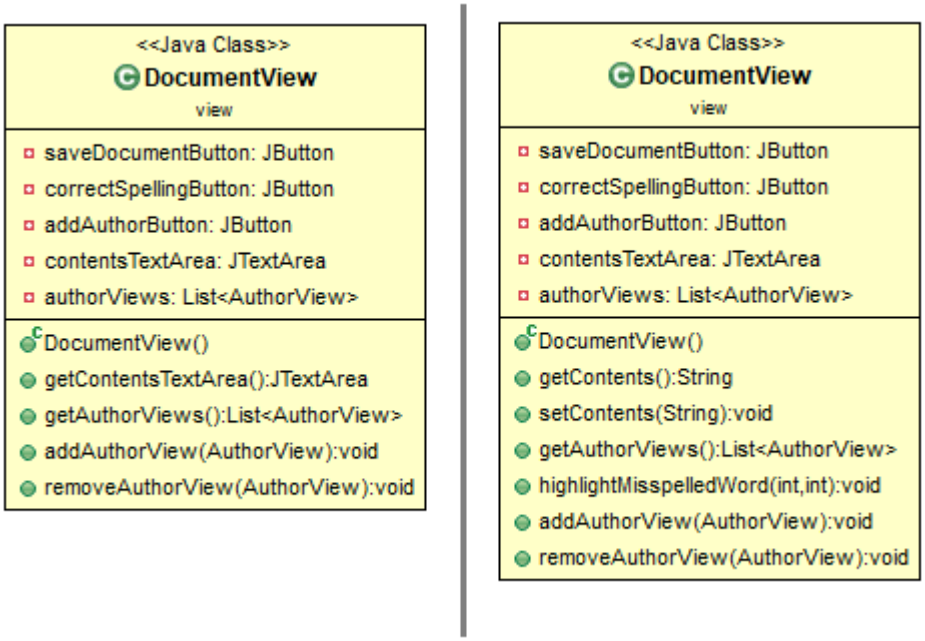

Figure 4. DocumentView accessor methods - two approaches

5. Declare controls on the widgets. A view object has to declare that when the X button is pressed, the Y control will be performed. This declaration of the controls on the widgets is a major difference from a typical Swing application in the view layer. Therefore, the next sections of this chapter are dedicated to describe this functionality.
6. Feed controls with parameters (addressed in detail in section section 5.2.6)

5.2 Installing controls and listeners to widgets with Java Annotations

Swing Boot makes a heavy usage of Java annotations. This section discusses what annotations used between the three MVC layers in a Swing Boot application.

5.2.1 @InstallControls - Indicating that a view object holds widgets that perform controls

The declaration of the controls on the widgets happen via Java annotations. All Swing widgets a view object holds that are about to interact with controls (upon user's actions) must be kept in fields of the view class. Then, annotations can be used upon these (widget) fields to declare the controls. Before proceeding to the details about how the controls are declared to widgets, let's examine the `@InstallControls` annotation.

The `@InstallControls` annotation is an annotation that can be used upon **a class or a field**. When the `@InstallControls` annotation is used in a class, it indicates that the objects of this class contain control declarations. So, when the objects of the class are injected by the IoC container, Swing Boot installs the controls to the widgets.

As an example, snippet 22 shows how the `DocumentView` class looks when Swing Boot is used. Also, notice the comments on the fields of the widgets. These comments motivate and are directly related to section 5.2.2.

```
@InstallControls
public class DocumentView {

	// When this button is pressed, perform SaveDocumentControl
	private JButton saveButton;

	// When this button is pressed, perform CorrectSpellingControl
	private JButton correctSpellingButton;

	// When this button is pressed, perform AddAuthorControl
	private JButton addAuthorButton;

	// When this text area is edited, perform HighlightMisspelledWordsControl
	private JTextArea contentsArea;

	@Inject
	public DocumentView() {
		// initialize the widgets
		// layout the widgets
	}

	// Access methods
}
```

Snippet 22.@InstallControls annotation used in a class

One question raised here is, what if an object that contains control declarations is not created by the IoC container? If the annotation is used on the class of this object, Swing

Boot will never be able to install the controls. That's why `@InstallControls` annotation can be used on fields. During the installation to the object that holds the annotated field, Swing Boot installs controls to the nested objects. In other words, the `@InstallControls` can be used on fields in order to support the nesting (composition) of view objects.

For instance, in our application this scenario can happen on the `EditorAppView`, which is a parent view. In a typical "dependency injection hierarchy", the `EditorAppView` would depend on its children, the `DocumentView` and `SpellCheckingOptionsView`. But this might not be the case. Let's assume that the `EditorAppView` creates its children. If the `EditorAppView` creates the `DocumentView` and the `SpellCheckingOptionsView`, the `@InstallControls` annotation on the classes of the child views does nothing. Because the `DocumentView` and `SpellcheckingOptionsView` objects are not created by the container. That's why the `@InstallControls` annotation can be used on the fields of the parent view. Based on the assumption we made (`EditorAppView` creates the subviews), snippet 23 shows how the installation of the controls can happen to the nested view objects. Only the parent view object needs to be created by the container.

```
@InstallControls
public class EditorAppView extends JFrame {

	@InstallControls
	private DocumentView documentView;

	@InstallControls
	private SpellCheckingOptionsView spellCheckingOptionsView;

	@Inject
	public EditorAppView() {
		super();

		this.documentView = new DocumentView();
		this.spellCheckingOptionsView = new SpellCheckingOptionsVi

		// ...
	}
}
```

Snippet 23.@InstallControls annotation used on a class and on fields of subviews

5.2.2 @OnEventHappend - Declaring controls to widgets via Java annotations

As mentioned in the previous section the declaration of the controls to the Swing widgets happens via the usage of annotations. The rest of this thesis uses the term "**OnEventHappend annotations**" to describe these annotations that declare controls.

In a typical Swing application, in order to add a listener to a widget, the developer has to create a listener class and implement the methods of this class. In a Swing Boot application the exact same process happens if an annotation is used upon the field of the widget. There are **annotations for each method of each listener**. **The name of the**

annotation is the prefix "On" followed by the method name of the Swing listener. Then, these annotations accept the class of the control that is to be performed.

For example, let's see how a control is declared for a button press. In Swing, in order to add an `ActionListener` to a `JButton`, the developer implements the `actionPerformed()` method. In a Swing Boot application, the developer uses the `@OnActionPerformed` annotation on the field of the button and gives the class of the control (as the `value()` attribute of the annotation). The same pattern is followed with most Swing listeners. Table 1 shows the matching between the **most commonly used** Swing listeners and OnEventHappend annotations.

SwingListener#method_name	Swing Boot annotation
ActionListener#actionPerformed	@OnActionPerformed
MouseListener#mouseClicked	@OnMouseClicked
MouseListener#mousePressed	@OnMousePressed
MouseListener#mouseReleased	@OnMouseReleased
MouseListener#mouseExited	@OnMouseExited
MouseListener#mouseEntered	@OnMouseEntered
DocumentListener#removeUpdate	@OnDocumentUpdated
DocumentListener#insertUpdate	@OnDocumentUpdated
DocumentListener#changedUpdate	@OnDocumentUpdated
KeyListener#keyTyped	@OnKeyTyped
KeyListener#keyPressed	@OnKeyPressed
KeyListener#keyReleased	@OnKeyReleased
FocusListener#focusGained	@OnFocusGained
FocusListener#focusLost	@OnFocusLost
ListSelectionListener#valueChanged	@OnSelectionChanged

Table 1.The matching between Swing Boot annotations and Swing listeners

That said, snippet 24 is the code of snippet 22 including the annotations that declare the controls upon the fields of the widgets.

```
@InstallControls
public class DocumentView {

	@OnActionPerformed(SaveDocumentControl.class)
	private JButton saveButton;

	@OnActionPerformed(CorrectSpellingControl.class)
	private JButton correctSpellingButton;

	@OnActionPerformed(AddAuthorControl.class)
	private JButton addAuthorButton;

	@OnDocumentUpdated(HighlightMisspelledWordsControl.class)
	private JTextArea contentsArea;

	@Inject
	public DocumentView() {
		// initialize the widgets
		// layout the widgets
	}

	// Access methods
}
```

Snippet 24.@InstallControls annotation used in a class

5.2.3 Event filtering and validation in the annotations

Each `@OnEventHappend` annotation corresponds to a Swing listener (or a part of it). When an event takes place, the Swing listener is fired and fed by an event object. This event object might or might not contain useful information. This information is typically used by the developer in order to add a condition on whether the action should take place or not.

Besides the `value()` attribute of each `@OnEventHappend` annotation that accepts the class of the control, `@OnEventHappend` annotations have attributes to add conditions based on the underlying Swing events.

In order to demonstrate, let's introduce a new scenario to our application. Besides the `correctSpellingButton` that performs the `CorrectSpellingControl`, the user can also correct the spelling of the contents by right clicking on the editor. In traditional Swing, a `MouseListener` would be used and the code would look like the snippet 25. The `MouseEvent.BUTTON3` corresponds to right click. The if condition is mandatory. Otherwise the code would run for any kind of click (left, middle, right).

```
contentsArea.addMouseListener(new MouseListener() {

	@Override
	public void mouseReleased(MouseEvent event) {}

	@Override
	public void mousePressed(MouseEvent event) {}

	@Override
	public void mouseExited(MouseEvent event) {}

	@Override
	public void mouseEntered(MouseEvent event) {}

	@Override
	public void mouseClicked(MouseEvent event) {
		if (event.getButton() == MouseEvent.BUTTON3) {
			doCorrectSpelling();
		}
	}
});
```

Snippet 25. Validating a right click in traditional Swing

This kind of basic validation - filtering of the event can take place on the `@OnMouseClicked` annotation. In Swing Boot the exact same functionality is achieved with the code of snippet 26.

```
@InstallControls
public class DocumentView {

	@OnDocumentUpdated(HighlightMisspelledWordsControl.class)
	@OnMouseClicked(value = CorrectSpellingControl.class, button = OnMouseClicked.BUTTON3)
	private JTextArea contentsArea;

	// other fields of widgets with annotations

	@Inject
	public DocumentView() {
		// ...
	}
}
```

Snippet 26. Validating a right click in Swing Boot

5.2.4 @InitializedBy annotation to initiate the view objects

One special annotation that Swing Boot provides is the `@InitializedBy` annotation. `@InitializedBy` annotation can be used on a class of the view layer and accepts a control class as value. **Right before Swing Boot installs the controls** (`@InstallControls` annotation) to the view object, **the control** that was declared in the `@InitializedBy` annotation **is performed for a single time**.

For instance, in our application we might have a `InitializeDocumentViewControl` that initializes the `DocumentView` with a `Document` object. In this case, we use the `@InitializedBy` annotation on the `DocumentView` class as shown in snippet 27.

```
@InstallControls
@InitializedBy(InitializeDocumentViewControl.class)
public class DocumentView {
	// ...
}

public class InitializeDocumentViewControl implements Control<Void> {
	@Inject
	public InitializeDocumentViewControl(DocumentService documentService,
					DocumentView documentView) {
		// constructor args to fields
	}

	@Override
	public void perform(Void unusedParameter) {
		Document document = documentService.createDocument();
		documentView.setDocument(document);
		documentView.setContents(document.getContents());
	}
}
```

Snippet 27. Example of @InitializedBy annotation

5.2.5 @WithoutControls annotation to temporary deactivate the controls of a view object

`@WithoutControls` solves the issue that was discussed in section 4.3. `@WithoutControls` annotation can be used **on methods** of a view class. During the execution of a `@WithoutControls` method, all the controls that were installed to the view object will be deactivated. After the execution of the method, they will be installed again. In plain words, during the execution of a `@WithoutControls` method, the view object is totally passive [Fowl06-PV]. By using the `@WithoutControls` annotation, the redundant cyclic calls to control objects do not take place.

In order to justify the existence of the annotation, we have to take in consideration a usage scenario of our application that involves the `DocumentView` object and its `contentsArea` widget. Remember that when the `contentsArea` is edited by the user, the `DocumentListener` is triggered. The event causes the `HighlightMisspelledWords` control to be performed.

While the user edits the contents of the `JTextArea`, he sees some misspelled words. Then, he presses the `correctSpellingButton`. When the `correctSpellingButton` is pressed, the `CorrectSpellingControl` is performed. Then, the `CorrectSpellingControl` executes `documentView.setContents(correctContents)`. The `setContents()` method of the `DocumentView` class calls `contents.setText(correctContents)`. This call makes `HighlightMisspelledWords` to be performed since the `DocumentListener` is fired by the change of the text. But the

performance of `HighlightMisspelledWordsControl` is redundant because the contents were changed from the system itself. In this example, the `@WithoutControls` annotation can be used upon the `setContents()` method of the `DocumentView` class. If the method has the annotation, the `HighlightMisspelledWordsControl` will not be performed. Snippet 28 shows the `DocumentView` class among the `CorrectSpellingControl` class. Also, notice the comments of the snippet.

```
public class CorrectSpellingControl implements Control<Void> {

	@Inject
	public CorrectSpellingControl(SpellCheckingService spellChecking,
	                              DocumentView documentView) {
		// Constructor args to fields
	}

	@Override
	public void perform(Void unusedParameter) {
		String misspelledContents = documentView.getContents();
		String correctContents = spellChecking.correctSpelling(misspelledContents);

		// We do not want HighlightMisspelledWordsControl to be performed
		// after this call
		documentView.setContents(correctContents);
	}
}

@InstallControls
public class DocumentView {

	@OnActionPerformed(CorrectSpellingControl.class)
	private JButton correctSpellingButton;

	@OnDocumentUpdated(HighlightMisspelledWords.class)
	private JTextArea contentsArea;

	// other widgets, other controls

	@Inject
	public DocumentView() {
		// ...
	}

	@WithoutControls
	public void setContents(String contents) {
		// HighlightMisspelledWords will not be performed because
		// the method is annotated with @WithoutControls
		contentsArea.setText(contents);
	}

	public String getContents() {
		return contentsArea.getText();
	}
	// ...
}
```

Snippet 28. Example of usage of @WithoutControls

5.2.6 Taking advantage of the parameter - @ParameterSource annotation

Something that is not addressed in the previous sections, is the parameter of the `perform(T genericParameter)` method. The parameter exists for two reasons:

- A use case (that the control handles) has a small variation and writing a new control class for the variation is not worth it.

  For example, in our application, the user might have the additional option to correct the misspelled words in the editor while ignoring words written in capital letters. Instead of having two controls (say `CorrectMisspelledWordsControl` and `CorrectMisspelledWordsIgnoreCapitalControl`), the `CorrectMisspelledWordsControl` takes a `Boolean` parameter.

- There is information not known upfront by the control object. Specifically, the Swing events.

  In the sample application, when the user edits the contents a `swing.DocumentEvent` is generated and then the `HighlightMisspelledWordsControl` is performed. This `swing.DocumentEvent` might contain valuable information (offset, length) that affects how the control is performed.

Since the declaration of the controls happen via Java annotations and annotations cannot provide such functionality, **the view object that declares the control must feed the parameter**. The view object can feed a control's parameter with methods that return a value (are not void). These **methods must be annotated with the @ParameterSource** annotation in order to be found by Swing Boot. Right before the control is performed, Swing Boot calls the method for the result and passes it to the control object.

A view object might declare multiple controls and parameter sources. Consequently, each `@ParameterSource` must be bonded with at least one control declaration. The binding between the control and the parameter source happens with the usage of an identifier that goes in both the `@ParameterSource` annotation and the `@OnEventHappend` annotation.

In order to demonstrate the usage of the parameter, snippet 29 shows how a view object can feed a `Boolean` value to `CorrectMisspelledWordsControl`. Notice that the "ignore_all_capital_words" identifier is used in both annotations.

```
public class CorrectSpellingControl implements Control<Boolean> {

	@Inject
	public CorrectSpellingControl(/* .. */) {
	}

	@Override
	public void perform(Boolean ignoreAllCapitalWords) {
		// ...
	}
}

@InstallControls
public class DocumentView {

	@OnActionPerformed(
			value = CorrectSpellingControl.class,
			parameterSource = "ignore_all_capital_words"
		)
	private JButton correctSpellingIgnoreAllCapitalWordsButton;

	@ParameterSource("ignore_all_capital_words")
	boolean shouldIgnoreAllCapitalWords() {
		return true;
	}
	//...
}
```

Snippet 29. Simple usage of @ParameterSource annotation

In snippet 29, `shouldIgnoreAllCapitalWords()` method has no parameter. Swing Boot just calls the method, takes the value and then passes it to the control object.

The question now is how Swing events or a portion of their information can be passed as parameters to the controls. When Swing Boot calls the `@ParameterSource` method, it will try to inject the Swing event as a parameter.

Going back to our motivating example, let's assume that the user can also correct the misspelled words while ignoring words written in capital letters. Let's also assume that there is a second `JButton` for this purpose. Snippet 30 shows exactly how this can be done in a Swing Boot application by using the parameter.

```
public class CorrectSpellingControl implements Control<Boolean> {

	@Inject
	public CorrectSpellingControl(/* .. */) {
	}

	@Override
	public void perform(Boolean ignoreAllCapitalWords) {
		// ...
	}
}

@InstallControls
public class DocumentView {

	@OnActionPerformed(
			value = CorrectSpellingControl.class,
			parameterSource = "ignore_all_capital_words"
		)
	private JButton correctSpellingButton;

	@OnActionPerformed(
			value = CorrectSpellingControl.class,
			parameterSource = "ignore_all_capital_words"
		)
	private JButton correctSpellingIgnoreAllCapitalWordsButton;

	@ParameterSource("ignore_all_capital_words")
	boolean shouldIgnoreAllCapitalWords(ActionEvent event) {
		return event.getSource() == correctSpellingIgnoreAllCapitalWordsButton;
	}
	//...
}
```

Snippet 30. A @ParameterSource method has the Swing event injected

As shown in snippet 30, passing the whole Swing event to the control object is not mandatory. In fact, the responsibility of a `@ParameterSource` method should be to take the swing event and extract (return) only the valuable information that the control object needs to operate.

5.2.7 Handling misplaced annotations - Swing Boot annotation processor

One disadvantage Java annotations have is that they are "misplace prone". Plain Java validates only if an annotation is placed on a correct element (field, method, type, etc). Java does not check the type of a field. So, what if `@OnActionPerformed` is used on top of a `JPanel` field? A `JPanel` object does not have an `addActionListener()` method. What if a control is parameterized but no `@ParameterSource` is declared? Swing Boot finds this kind of annotation "misusage" at runtime and exceptions are thrown. However, finding errors at runtime can decrease the developer's productivity.

Java allows and supports custom annotation processors. An annotation processor can validate if an annotation is used properly at compile time. Swing Boot comes with its own processor that validates the usage (placing) of all Swing Boot annotations. In order

to make the annotation processor work, the developer has to make some configurations in his/her development environment. Information and guides of how to enable the annotation processor in different development environments exist in Swing Boot's code repository.

Figure 5 demonstrates the usage of the annotation processor in the Eclipse IDE. The `@OnActionPerformed` is used on top of a `JPanel` (which is not allowed) and the error is spotted at compile time.

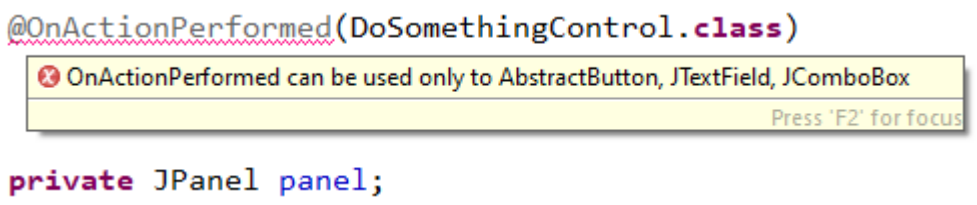

Figure 5. Swing Boot annotation processor shows the error of a misplaced annotation at compile time

5.3 Facilitating concurrency - Asserting & running in the correct thread

The section discusses how Swing Boot facilitates threading and concurrency in a Swing application. Swing Boot provides two kinds of annotations related to concurrency. The first kind addresses the assertion of the correct thread. The second kind helps the developer to handle the execution of code in the correct thread.

Keep in mind that the term "correct thread" means that the event dispatch thread is not blocked by long running tasks (e.g an HTTP request). Correct thread also means that all widget painting and changes are done within the event dispatch thread (see section 4.2).

5.3.1 @AssertUi and @AssertBackground annotations

`@AssertUi` and `@AssertBackground` annotations are simple to understand. These annotations can be used **on top of a method** to assert that the method is executed in the correct thread. On the one hand if a method is annotated with `@AssertUi` and was executed outside the event dispatch thread (EDT), a warning will be logged. On the other hand, if a method is annotated with `@AssertBackground` (the method is supposed to run in a background thread) and it was executed within the EDT, a warning will be logged. Both of these annotations have a `throwException()` attribute which is false by default. If the value of the attribute is true, an exception is thrown instead of a warning.

As an example, in our application we assumed that the `saveDocument()` method of `DocumentService` does an expensive, long running IO operation. The method should always run in a background thread, otherwise the whole GUI will "freeze" (see section 4.2). This method is the perfect candidate for the `@AssertBackground`. Snippet 31 shows the usage of the `@AssertBackground` annotation. If the method is called within the EDT, an exception will be thrown.

```
public class DocumentServiceImpl implements DocumentService {
	//...

	@Override
	@AssertBackground(throwException = true)
	public void saveDocument(Document doc) throws IOException {
		//...
	}
}
```

Snippet 31. A method with @AssertBackground annotation

5.3.2 @InUi and @InBackground annotations

`@InUi` and `@InBackground` annotations decide on which thread **a method** will be executed. If a method is annotated with `@InUi`, it will run within the event dispatch thread (EDT). If a method is annotated with `@InBackground`, it will run in a background thread. These annotations provide a concise way to "jump" from the EDT to a background thread and then back to EDT for UI updates.

In our application, based on the assumption we made (`saveDocument(Document d)`) method must run in background), snippet 32 shows the complete implementation of `SaveDocumentControl`. Notice the print statements along with the comments in the snippet.

```
public class SaveDocumentControl implements Control<Void> {

	@Inject
	public SaveDocumentControl(DocumentView view, DocumentService service) {
		// constructor args to fields
	}

	@Override
	public void perform(Void parameter) {
		System.out.println(Thread.currentThread()); // prints EDT

		Document document = documentView.getDocument();
		saveDocument(document);

		System.out.println(Thread.currentThread()); // prints EDT
	}

	@InBackground
	void saveDocument(Document d) {
		System.out.println(Thread.currentThread()); // prints background
		try {
			service.saveDocument(d);
			showDocumentWasSaved();
		} catch (IOException exc) {
			exc.printStackTrace();
			showDocumentWasNotSaved();
		}

	}

	@InUi
	void showDocumentWasNotSaved() {
		System.out.println(Thread.currentThread()); // prints EDT
		JOptionPane.showMessageDialog(null, "Document was not saved!");
	}

	@InUi
	void showDocumentWasSaved() {
		System.out.println(Thread.currentThread()); // prints EDT
		JOptionPane.showMessageDialog(null, "Document was saved!");
	}

}
```

Snippet 32. Usage of @InUi and @InBackground - jumping from one thread to another

In snippet 32, it is worth mentioning that the `@InUi` annotation is mandatory in both `showDocumentWasNotSaved()` and `showDocumentWasSaved()` methods. If the annotation is missing from the methods, these methods will run in the background thread because they are called from an `@InBackground` method.

`@InBackground` has some limitations in comparison to the traditional way of executing code in a background thread (SwingWorker - see section 4.2). These limitations are:

- The `@InBackground` method must be void. If the method returns a value, this value will always be `null`. As snippet 32 shows, after calling `saveDocument(document)` the EDT keeps running. A value cannot be returned from the method.

- An `@InBackground` calculation cannot be canceled. If canceling the background task is a requirement, a SwingWorker must be used.

5.4 Advantages of using the framework

In this section, we discuss the advantages of a Swing Boot application compared to an application developed with plain Swing.

- **Separation of concern.** Separation of concern exists to a Swing Boot application because it **respects a slightly different variation of MVC** (see section 5.1). Just like traditional MVC, the concerns are separated to each one of the layers of the architecture. In the model layer of a Swing Boot application, there is the business logic. In the control(s) layer, the user's input is handled. Finally, UI elements like widgets, layouts, static text and icons exist in the view layer.
- **Low coupling - There are no complex hierarchies & God objects.** In section 4.1, we discussed some of the possible issues of traditional MVC and how a standard MVC approach can lead to God objects and complex hierarchies (e.g hierarchical MVC). Specifically in the control(ler) and the view layer when breaking a view class to smaller view classes is attempted. In a Swing Boot application, only the composition (as hierarchy) of the view classes exist. There are no parent-child associations in the control(ler) layer. As a consequence, **the control layer and the view layer are loosely coupled.** Figure 6 shows a brief UML class diagram of our application using Swing Boot. Notice that each control depends only on the view object it needs.

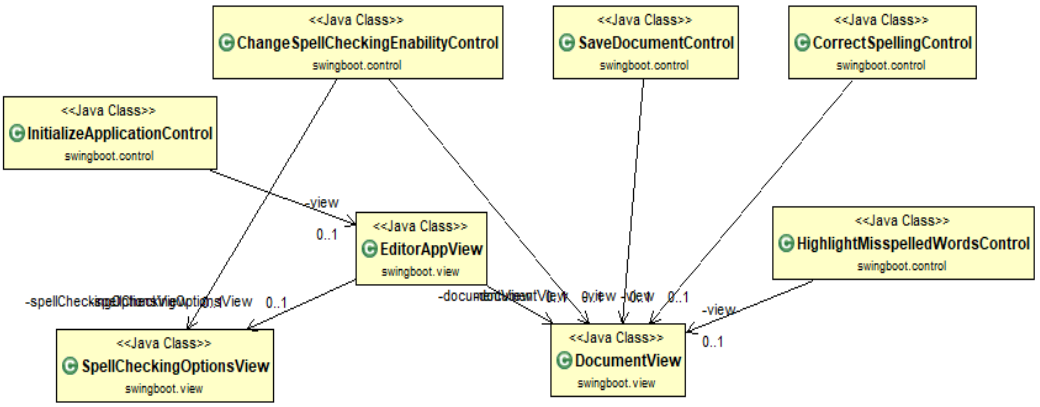

Figure 6. Brief UML class diagram of the motivating example - Swing Boot

- **Reduced code noise**. Kevlin Henney used the term code noise [Henn10] to describe comments in the code. However, this thesis uses the term code noise to describe the number of lines required in Swing for a simple delegation. Specifically, the no-op methods of an anonymous class - listener.

  Let's assume that we want to know whether the user **pressed** right click upon a widget. In traditional Swing, we have to write the code of snippet 33. We are forced by the compiler to implement five methods. The one method that we are interested in (`mousePressed()`) and then leave the other four methods as no-op.

```
widget.addMouseListener(new MouseListener() {

      @Override
      public void mouseReleased(MouseEvent e) { }

      @Override
      public void mousePressed(MouseEvent e) {
            boolean isRightClick = e.getButton() == MouseEvent.BUTTON1;
            if (isRightClick)
                  doSomething();
      }

      @Override
      public void mouseExited(MouseEvent e) {     }

      @Override
      public void mouseEntered(MouseEvent e) { }

      @Override
      public void mouseClicked(MouseEvent e) { }
});
```

Snippet 33. Adding a listener in Swing - boilerplate code

The boilerplate code in snippet 33 is obvious. If there is a lot of interaction between the user and the view object, things are even worse. In comparison to the code of snippet 33, the **exact same functionality is achieved** concisely when Swing Boot is used **within two lines**. Snippet 34 shows these lines.

```
@OnMousePressed(value = DoSomethingControl.class, button = MouseEvent.BUTTON1)
private JComponent widget;
```

Snippet 34. Declaring a control in Swing Boot - 2 lines

- **Convenient UI prototyping**. In a Swing Boot application the view layer is (can also be totally) decoupled from the model layer. **A view object depends only on other view objects. As a result, prototyping the user interface does not require much effort.** The developer can write a simple Java `main()` method, create a view object and prototype the user interface without any controls installed. Creating a small demo to see if all margins, paddings, borders, static text and icons are in place does not require the construction of "heavy" objects of the model layer.

  Assume that in our application, we want to prototype and see how the `DocumentView` looks on the screen. All it takes to achieve the prototyping is to have the code of snippet 35. (for `invokeLater()` method see section 4.2).

```
public static void main(String[] args) {
	SwingUtilities.invokeLater(new Runnable() {

		@Override
		public void run() {
			JFrame demoFrame = new JFrame("DocumentView UI prototype");
			demoFrame.add(new DocumentView());
			demoFrame.setVisible(true);
		}
	});
}
```

Snippet 35. Convenient UI prototyping

5.5 Disadvantages of using the framework

Using Swing Boot for an application does include only advantages. Unfortunately, there are some disadvantages as well. This section discusses these disadvantages.

- **The project is tied to Swing Boot.** Perhaps the most important disadvantage of using Swing Boot for a Swing application, is that the application - project becomes tied to the framework. All annotations and the swingboot.Control interface that all controls must implement are part of Swing Boot. I**f one day the developer decides to remove it from the project's classpath, the project will end up with multiple compilation errors**. In addition, the loose coupling between the project and Guice (see section 2.2.1) is lost.
- **Controls - A lot of classes to coordinate.** Controls follow the same structure of the command pattern [GHJV94] (see section 5.1.2). The user's input is handled by isolated control objects and each control object is an instance of a different class. As a result, these **control classes will be numerous**, depending on the size of the application and the number of the use cases.
- **The developer has to learn new things for an old GUI toolkit.** As mentioned in the introduction, Swing is a toolkit that was developed more than twenty years ago. Therefore, there is a lot of existing content (books, tutorials, guides, etc) available for Swing. A lot of Java developers develop Swing applications based on this content. **Learning how to use Swing Boot requires extra effort from the developer's perspective** since Swing Boot introduces new ways to achieve functionalities.

Chapter 6. Swing Boot - Internals

This chapter explains how Swing Boot works internally. The framework consists of two subsystems. **The control subsystem and the concurrency subsystem.** For each one of these subsystems, there is a different section that explains how the subsystem works. Some (implementation) details are not included since these details are out of the scope of this thesis. Also, keep in mind that since we are in the context of object oriented programming, this chapter uses the term "*the <class-name> does the Y*" as "*the <class-name>* ***object*** *does the Y*". For example, "*the ControlInstaller installs the controls*" means "*the ControlInstaller object installs the controls*".

6.1 Control subsystem behind the scenes

The control subsystem is the one that introduces the controls (see section 5.1) and all related annotations (`@OnEventHappend`, `@InstallControls`, etc) to a Swing application. In order to explain how the control subsystem works, figure 7 shows a (partial) UML class diagram. The classes of the diagram contain the core functionalities of the control subsystem.

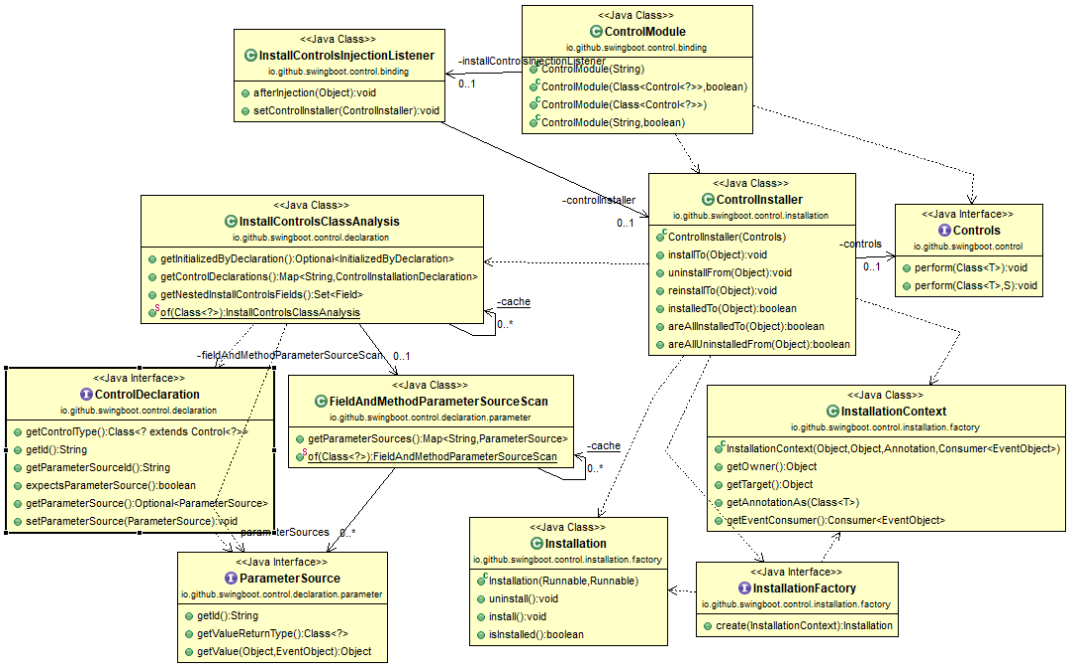

Figure 7. UML class diagram of the control subsystem

Figure 9 gives an example that we employ to introduce some terminology that we assume in the internals of the control subsystem. The code is in the context of our application, but it does not play any important role.

```
@InstallControls
public class DocumentView {

        @OnActionPerformed(value = SaveDocumentControl.class, id = "saveDocument")
        private JButton saveDocumentButton;

        @ParameterSource("saveDocument")
        Void dummyParameterSource() {
                return null;
        }
}
```

DocumentView: A DocumentView **object** is an **Owner**, meaning it "owns" control declaration(s).

DocumentView.class: The **owner's class**.

saveDocumentButton: The **object** (the value of the field) saveDocumentButton is a **Target**, meaning it "accepts" the control installation.

JButton saveDocumentButton: This is a **Target Element**. A target element is the **type** (class) of the field and the **name** of the field. It is not the value of the field.

dummyParameterSource(): The **method** is a **ParameterSource**.

ControlDeclaration: A control declaration includes the **annotation**, the **parameter source** and the **target element**.

Figure 9. Terminology inside the control subsystem

For each type (class, interface) of the class diagram (figure 7) there is a CRC card [Papa07] below. If the card refers to an interface, the card contains the implementations of this interface.

Class: (an implementation of) `ControlDeclaration`	
Responsibilities	**Collaborators**
• Holds (meta)data for a control declaration • Checks if a control declaration is valid (e.g `@OnActionPerformed` cannot be declared to a `JPanel`)	• `ParameterSource` • `java.lang.reflect` package
Implementations: • `InitialzedByDeclaration` • `ControlInstallationDeclaration`	

Class: (an implementation of) `ParameterSource`	
Responsibilities	**Collaborators**
• Holds (meta)data for a parameter	• `java.lang.reflect` package

source ● Checks if a parameter source is valid (e.g method should not be `void`)	
Implementations: ● `MethodParameterSource` ● `FieldParameterSource`	

Class: `FieldAndMethodParameterSourceScan`	
Responsibilities	**Collaborators**
● Locates the parameter sources in the owner's class ● Creates `ParameterSource` objects	● `java.lang.reflect` package ● `ParameterSource`

Class: `InstallControlsClassAnalysis`	
Responsibilities	**Collaborators**
● Locates the control declarations in the owner's class ● Creates `ControlDeclaration` objects ● Binds `ControlDeclaration` objects to their parameter source (if any)	● `java.lang.reflect` package ● `ParameterSource` ● `FieldAndMethodParameterSourceScan` ● `ControlDeclaration`

Class: `InstallationContext`	
Responsibilities	**Collaborators**
● Holds information of a control installation (an owner, a target, an annotation)	● -

Class: `Installation`	
Responsibilities	**Collaborators**
● Holds a pair of `Runnable` objects - the runnable that installs a listener to a Swing widget and the runnable that uninstalls the listener from the widget	● -

Class: (an implementation of) `InstallationFactory`	
Responsibilities	**Collaborators**
• Given an `InstallationContext` object, creates an `Installation` object	• `InstallationContext` • `Installation`
Implementations: • `OnActionPerformedInstallationFactory` • `OnKeyPressedInstallationFactory` • `OnMouseClickedInstallationFactory` • …	

Class: (an implementation of) `Controls`	
Responsibilities	**Collaborators**
• Given an a `Class<Control>`, creates a control object of this class and call the `perform()` method	• (Ioc Container)
Implementations: • `InjectedByGuiceControls`	

Class: `ControlInstaller`	
Responsibilities	**Collaborators**
• Orchestrates the whole installation process for an owner • For each `ControlDeclaration`, creates `InstallationContext` objects and passes the `InstallationContext` objects to a corresponding `InstallationFactory`	• `InstallControlsClassAnalysis` • `Controls` • `InstallationFactory` • `InstallationContext` • `Installation`

Class: `InstallControlsInjectionListener`	
Responsibilities	**Collaborators**
• Listens for injections • If an injected object is an owner (`@InstallControls`), calls `ControlInstaller`	• `ControlInstaller`

Class: `ControlModule` (implements `guice.Module`)	
Responsibilities	**Collaborators**
• Registers an `InstallControlsInjectionListener` • Binds a `Controls` implementation • Binds to the IoC Container all controls of a package (optional)	• `Guice` • `InstallControlsInjectionListener`

Figure 10 shows the package diagram of the control subsystem.

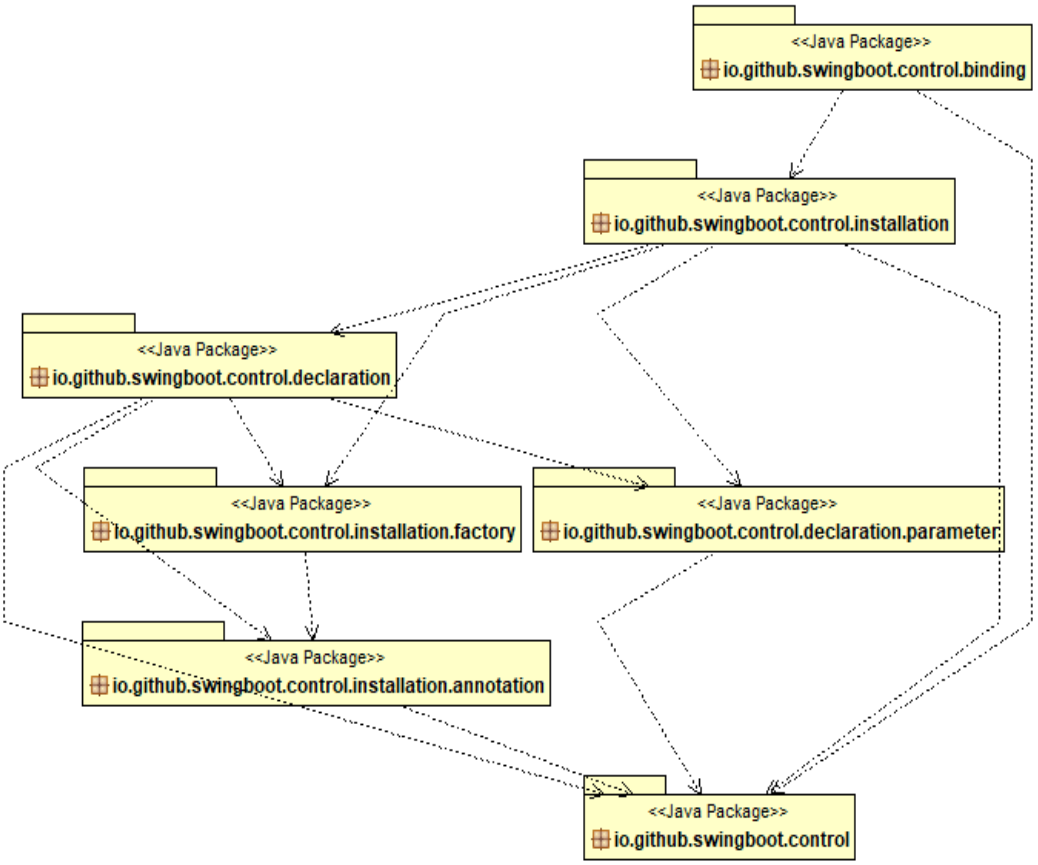

Figure 10. Package diagram of the control subsystem

6.2 Concurrency subsystem behind the scenes

Concurrency subsystem works by **utilizing Aspect Oriented Programming** (AOP) [Hank03][Papa04]. Guice supports AOP by letting the developer write method interceptors. For each one of concurrency related annotations, `@AssertUi`, `@AssertBackground`, `@InUi` and `@InBackground`, there is a different method interceptor.

As in the control subsystem and the `ControlModule` class, there is the `ConcurrencyModule` class. The `ConcurrencyModule` class is a `guice.Module` which binds the four method interceptors.

The additional behavior (also called advice) of the `@AssertBackground` and `@AssertUi` corresponding interceptors is a check of the current thread **before the execution of the method**. If the thread of the execution matches the expected thread, the interceptor proceeds the invocation of the method. Otherwise, depending on the `throwException()` attribute of the annotation, an exception is thrown or a warning is logged (see section 5.3.1).

The advice of the `@InUi` and `@InBackground` corresponding interceptors is to swap the current thread of execution to the expected one. As obvious, these two method interceptors swap the thread of execution before the actual method is executed. More details about the `@InUi` and `@InBackground` method interceptors are beyond the scope of this thesis.

Chapter 7. Summary & future work

7.1 Summary

In this thesis, we discussed some of the issues a developer meets in Swing GUI development. More importantly, issues related to the architecture of the GUI and concurrency. We proposed a framework which intends to modernize Swing and solve these issues. In order to solve these issues the framework introduces modern patterns and techniques such as dependency injection and aspect oriented programming (AOP).

Now, is the framework worth using it? It depends. On the one hand, if the application that is going to be developed is small with only few use cases, traditional approaches might be a safer choice. One the other hand, if the application is medium-sized with enough complexity, the benefits of using the framework are going to be highlighted.

7.2 Future framework extensions

Some features that could be provided by the framework in a later version could be:

- **Support other inversion of control (IoC) containers than Guice.** If a project already uses a dependency injection (DI) framework other than Guice, using Swing Boot (which uses Guice) might give hard time to the developer to configure both DI frameworks. Therefore, it would be convenient if Swing Boot would support other DI frameworks, such as Spring.
- **Add support for cancel-able @InBackground tasks.** Section 5.3.2 that refers to `@InBackground` annotation, mentions that an `@InBackground` task cannot be canceled. It would be handy if the annotation provided a `cancelWhenCalledAgain()` boolean attribute. Based on the value of this attribute, if the `@InBackground` method is called twice the previous task would be cancelled (if it is not finished).
- **Replace reflection with code generation.** Section 5.2.7 mentions that Swing Boot comes with its own annotation processor. This annotation processor validates only if the annotations are used properly and have a correct placing. Annotation processors are also capable of generating code. In order to increase performance, a more advanced annotation processor can replace all internal parts of Swing Boot that use reflection with generated code.

8. References

[Sali14] Shadman Salih. Selection of Computer Programming Languages for Developing Distributed Systems, pp 1, Leicester, United Kingdom, 2014

[Gris98] Scott J. Griswold. A Java Implementation of a Portable DesktopManager, pp 29-35, University of North Florida, 1998

[Henn10] Kevlin Henney. 97 Things every programmer should know, pp 33-35, 2010

[SWTw20] "Standard Width Toolkit", Wikipedia. Last modified: 15 October 2020, https://en.wikipedia.org/wiki/Standard_Widget_Toolkit

[Fowl06-UI] Martin Fowler. GUI Architectures, July 2006, https://martinfowler.com/eaaDev/uiArchs.html

[Matt96] Michael Mattson. Object-Oriented Frameworks A survey of methodological issues, University College of Karlskrona/Ronneby, 1996

[Fowl04] Martin Fowler. Inversion of Control Containers and the Dependency Injection pattern, section #InversionOfControl, January 2004, https://martinfowler.com/articles/injection.html

[Fowl06-PV] Martin Fowler. Passive View. 18 July 2006 [Online], Available: https://martinfowler.com/eaaDev/PassiveScreen.html [Accessed: 23-Nov-2020]

[Henn16] Kevlin Henney. Small is beautiful, GOTO conference, Copenhagen 2016, Available: https://gotocon.com/cph-2016/presentations/show_talk.jsp?oid=7705 [Accessed: 01-Dec-2020]

[BDV+06] Bart Du Bois, Serge Demeyer, Jan Verelst, Tom Mens and Marijn Temmerman. Does God class decomposition affect comprehensibility? Universiteit Antwerpen, pp 1-6, January 2006

[Kara08] Alexandros Karagkasidis. Developing GUI Applications: Architectural Patterns Revisited, A Survey on MVC, HMVC, and PAC Patterns. pp 2-32

[Evan03] Eric Evans. Domain-Driven Design: Tackling Complexity in the Heart of Software. August 20, 2003, pp 79-83. ISBN: 0-321-12521-5

[ZVM+18] Apostolos V. Zarras, Panos Vassiliadis, Georgios Mamalis, Aggelos Papamichail, Panagiotis Kollias. And the Tool Created a GUI That was Impure and Without Form:Anti-Patterns in Automatically Generated GUIs. EuroPLoP '18, July 4–8, 2018, Irsee, Germany

[Papa07] Panagiotis Papadakos. Very rapid software-design prototyping with interactive 3D CRC cards. Computer Science Department University of Crete, Heraklion, November 2007, pp 1-17

[Papa04] George A. Papadopoulos. Aspect Oriented Programming for a component-based real life application: a case study. January 2004

[Hank03] Mario B. Hankerson. Towards a Taxonomy of Aspect-Oriented Programming. Department of Computer and Information Sciences East Tennessee State University, December 2003, pp 11-21

[CKP00] J. Cai, R. Kapila, G. Pal. HMVC: The layered pattern for developing strong client tiers. https://www.infoworld.com/article/2076128/hmvc--the-layered-pattern-for-developing-strong-client-tiers.html [Accessed: 03-May-2020]

[Cout87] Joëlle Coutaz. PAC, an Object Oriented Model for Dialog Design. Laboratoire d'informatique de Grenoble (University of Grenoble), 1987.

[GHJV94] Gamma Erich, Helm Richard, Johnson Ralph, Vlissides John. Design Patterns: Elements of Reusable Object-Oriented Software, October 1994, pp 233-242. ISBN: 978-0201633610

[HPSS06] Christian Haack, Erik Poll, Jan Schafer, Aleksy Schubert. Immutable Objects in Java. April 2006, pp 01- 08