Latex Assignment Template

Course Name, University Name

Luca Di Bello

Sunday 17th November, 2024

1 Introduction

This assignment requires the use of the knowledge acquired during the course, in order to refactor an existing open-source project, aiming to improve its design. The behavior of the project should remain unchanged, as well as its input and output interfaces.

The refactoring should target at least 1000 lines of code, and the changes should be documented in a report, and pushed to a separate branch in the project's repository, allowing an easy comparison between the original and refactored versions.

Furthermore, a personal objective was set, aiming to find a recent and active project, with a high number of stars, in order to improve its design and give back to the community. To accomplish this, the project size was analyzed in order to decide whether a complete refactor was feasible.

To find valuable candidates for this assignment, the GitHub search feature was used, filtering the results by language, Java, and the number of stars, between 100 and 1000. The search results were sorted by last update date, in descending order in order to find active projects.

To conclude, each selected project size was analyzed with the web application Count LOC, in order to count the total lines of code (later referred as *LOC*) that would be affected by the refactor (considering only source code, excluding tests or other utilities).

1.1 Project selection

As cited before, in order to accomplish the personal objective set at the beginning of this assignment, the search targeted project with a high number of stars, between 100 and 1000, and written in Java language in order to leverage the knowledge acquired during the course. The search results were sorted by the last update date, in descending order, to find active projects. After an in-depth analysis of results, these are the possible candidates selected for further investigation:

- fanglet/kanzi: Kanzi is a modern, modular and efficiend lossless data compressor implemented entirely in java. It uses state-of-the art entropy coders and multi-threading in order to efficiently utilize multi-core CPUs. The design of the library is modular, allowing to select at runtime the best entropy coder for the data to compress. The project has 109 stars, 18 forks, no open issues, and approximately 20,000 LOC. Kanzi was initially selected for this assignment, but later discarded due to its size.

- byronka/minium: Minium is a minimalistic web framework written in Java, built from scratch using few dependencies. The project provides essential components for web application development, including a web server and an in-memory database with disk persistence. This project was particularly interesting as it emphasizes simplicity and minimalism, which is a good starting point for refactoring as most of the code is written from scratch without complex dependencies.

Minium has 611 stars, 38 forks, no open issues, and approximately 9'000 LOC (excluding tests). Due to its size, and lack of open issues, Minium was discarded as a candidate for this assignment.

- jinput/jinput: JInput is a Java library designed for accessing input devices such as game controllers, joysticks, and other peripherals. It provides a platform-independent API to facilitate the integration of various input devices into Java applications. The project has 150 stars, 79 forks, 29 open issues, and 10,000 lines of code (considering only core functionality, excluding tests). JInput was selected for this assignment, as it has a reasonable size, a good number of open issues and a high number of forks, indicating an active community.

Note: The data presented above was collected on the Sunday 17$^{\text{th}}$ November, 2024, and may have changed since then. Check the project's repository for the most recent information.

1.2 High-level overview of the project structure

The JInput library is organized into several key components, each serving a distinct role in providing platform-independent access to input devices in Java applications.

The **Core API** module offers the fundamental interfaces and classes that define the JInput framework. It includes abstractions for controllers, components, and events, establishing a consistent API for interacting with various input devices.

To ensure compatibility across different operating systems, JInput incorporates **platform-specific plugins** tailored to Windows, Linux, and macOS. These plugins often contain native code to interface directly with the underlying system's input APIs, enabling seamless integration with the host environment.

The **natives** component comprises the native libraries required by the platform-specific plugins. These native binaries are essential for the plugins to communicate effectively with the operating system's input subsystems.

JInput provides a collection of **example applications** demonstrating how to utilize the library's features. These examples serve as practical guides for developers to understand and implement JInput in their projects.

A suite of **unit and integration tests** is included to verify the correctness and reliability of the library's components. These tests ensure that the core API and plugins function as intended across different platforms.

This modular architecture allows developers to integrate JInput into their applications efficiently, selecting only the components relevant to their target platforms and input device requirements.

In order concentrate the refactor in a single module, the *Core API* was selected as the main target for the refactor, as it represents the core functionality of the library, thus affecting the majority of users of the library.

1.3 Additional tools and resources

In order to perform a comprehensive refactor of the project, the SonarQube static code analysis tool will be used to identify potential code smells, bugs, and vulnerabilities, while also providing insights into the overall code quality. Additionally, in order to have a more in-depth understanding of the current design of the project, the *Pattern4j* tool will be used to detect the use of design patterns in the codebase, an essential aspect of the refactoring process.

The results of both tools provide valuable information to guide the refactor, highlighting areas of improvement and potential refactoring opportunities, and, by combining both outputs, will be possible to have a more comprehensive view of the project's design and code quality.

1.4 Building the project

The *jinput* project uses Maven as the build system, and, as the analysis of the project design will target only the *Core API* module, the build process must be focused on this module only. To do so, the following command can be used to build the project:

```
mvn clean install -pl core-api
```

Listing 1: Building the project using Maven

The `-pl` flag allows to specify the module to build, in this case, the *Core API* module. Furthermore, the `clean` and `install` goals are used to ensure a clean build, and to install the artifacts in the local Maven repository, respectively.

The build artifacts can be found in the `target` directory of the *Core API* module. The bytecode generated by the build process will be used by the *Pattern4j* tool to analyze the project's design.

2 Project health analysis

The project will be analyzed using both SonarQube and *Pattern4j* tools. In the following section, the results of the analysis will be presented, highlighting potential code smells, bugs, vulnerabilities, and design patterns detected in the project.

As the configuration and usage of both tools is out of the scope of this report, the following sections will focus on the results of the analysis, providing insights into the current state of the project, and guiding the refactoring process. Refer to the respective documentation for more information on how to configure and use both tools.

2.1 SonarQube analysis

After running the SonarQube analysis on the `CoreAPI` module, were detected a total of 26 code smells and 3 security hotspots. The follwing paragraphs will provide an overview of the most relevant code smells detected in the project, and the actions to be taken to address them.

Code smells Out of the 26 code smells detected in the project, 11 of them are categorized as critical, 4 as major, and 11 as minor. The following table provides a summary of the found code smells, categorized by severity.

Severity Type	Issues (Count)
Major	Bad Practice (3), Performance (3), Pitfall (3), CERT (2), Error Handling (2), Confusing (1), CWE (1), Design (1), Suspicious (1)
Critical	Brain Overload (2), CERT (1), Suspicious (1)
Minor	Convention (9), Brain Overload (1), Clumsy (1)

Table 1: SonarQube Severity Issues Summary

As shown in Table 2.1, the most common code smells in the project are related to bad practices, performance, and error handling. As the total number of code smells is relatively small, the refactoring will take into account all the detected issues, with a special focus on the critical and major ones.

Security hotspots The SonarQube analysis also detected 4 security hotspots in the project. Three of them are categorized as *Insecure configuration*, and one as *Log Injection*. The three insecure configuration hotspots are all coming from a common problematic pattern used in the `ControllerEnvironment` class, where, in case of an exception, the stack trace is printed to the standard output. This behavior can lead to information disclosure, and should be addressed as soon as possible.

The log injection issue on the other hand is related to the use of the `Logger` class without proper sanitization of the logger name, which can lead to log injection attacks.

Both security hostspots will be addressed in the refactoring process in order to enhance the security of the project.

2.2 Pattern4j analysis

References

[1] Dafny. *Getting Started with Dafny: A Guide*. http://web.archive.org/web/20080207010024/http://www.808multimedia.com/winnt/kernel.htm. Accessed: 12.03.2024.