Latex Assignment Template

Course Name, University Name

Luca Di Bello

Tuesday 19th 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.

- 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 initially selected for this assignment, but unfortunately, after inspecting the codebase it was discarded as the project presented too many platform-specific implementations

spanning multiple modules and a complete lack of documentation and tests. This would make the refactoring process too complex and time-consuming.

- 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 5'500 LOC (excluding tests and comments). After a thorough analysis, this project was selected for refactoring as it presents a clear and concise codebase divided into several packages, which will allow for a more focused refactor. Furthermore, it presents a complete JavaDoc documentation, which will be useful to understand the project's design.

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 project presents a single *Maven* module which contains the core functionality of the framework. The module contains 10 packages, each with a specific purpose. The following list provides a high-level overview of the purpose of each package:

- `database`: In-memory database with disk persistence.
- `htmlparsing`: HTML parsing utilities.
- `logging`: Logging utilities.
- `queue`: Queue of tasks to be executed asynchronously.
- `security`: Fail2Ban-like security mechanism and threat detection via log analysis.
- `state`: Singleton context to store global state.
- `templating`: Template engine to inject dynamic content into HTML pages.
- `testing`: Minimalistic testing framework for unit and integration tests.
- `utils`: General utilities.
- `web`: Web server and request handling.

From a first glance, the project seems well-structured, with clear separation of concerns between packages. The codebase is well-documented, with JavaDoc comments present in most classes and methods, providing a good starting point to understand the project's design. In section 2, the project design will be analyzed in more detail, focusing on the use of design patterns and potential code smells.

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 *minum* project uses Maven as the build system, and.

To simplify the configuration process, the creator of the library created a Maven Wrapper script (named `mvnw`) which is a self-container script that allows to automatically download the necessary Maven version to build the project, ensuring that the build process is reproducible across different environments. This script is located in the root of the project, and can be used to build the project by running the following command:

```
./mvnw clean install
```

Listing 1: Building the project using Maven Wrapper script

Note: If the current environment already has Maven installed, the project can be built using the `mvn` command instead of the wrapper script.
The build artifacts can be found in the `target` directory. The bytecode generated by the build process will be used by the *Pattern4j* tool to analyze the project's design.

2 Project health analysis and palanned refactoring

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 Static Analysis Tools

2.1.1 SonarQube analysis

After running the SonarQube analysis on the entire Minium codebase (including tests in order to get the test coverage metric), were detected a total of 588 code smells, 26 security hotspots and 34 possible bugs. In the following paragraphs the results will be briefly analyzed in order to plan the refactoring process.

Code smells Out of the 588 code smells detected in the project, 109 of them are categorized as critical, 74 as major, and 405 as minor. Table 1 provides a summary of the found code smells, categorized by severity.

Table 1: SonarQube Severity Issues Summary

Severity Type	Issues
Critical	design (90), suspicious (10), brain-overload (6) convention (1), multi-threading (1), pitfall (1)
Major	cert (40), html5 (20), obsolete (19) bad-practice (17), owasp-a3 (17), cwe (16), error-handling (15), pitfall (8), suspicious (7) accessibility (5), unused (4), wcag2-a (4) confusing (2), design (2), brain-overload (1)
Minor	convention (374), cwe (7), java8 (5), brain-overload (4), pitfall (4), performance (2), regex (2), unused (2), bad-practice (1), clumsy (1), suspicious (1)

As shown in Table 1, the most common code smells in the project are related to Java conventions, design issues, CERT secure coding standards, and the use of HTML5 in comments.

The refactoring of this project will focus on design, convention and brain-overload, as they are more related to the design and organization of the code, and can have a significant impact on the maintainability and readability of the project.

Security hotspots and bugs As cited before, the SonarQube scanner detected several bugs and security hotspots in the project. The security hotsposts are related to possible *Denial of Service* attacks leveraging Regular Expressions backtracking and log injection vulnerabilities.

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.

2.1.2 Pattern4j analysis

Unfortunately, the pattern4j tool was not able to analyze the entire project codebase due to the use of Java 21 features in the project. By running the tool using the custom `run-pattern4j-headless.sh` script, the following error was raised:

```
Exception in thread "main" java.lang.IllegalArgumentException: Unsupported class file major ve
```

For this reason, the design pattern usage anaylsis will be skipped. Fortunately, this step is not crucial for the refactoring of the project as it would have only provided additional insights into the design of the project rather than pinpointing specific issues.

2.1.3 Large class detection

In order to detect large classes, a custom bash script was developed to count the number of lines of code of each class in the project. The script uses the `cloc` tool to count the number of lines of code of each file in the project, and then orders the results by the number of lines of code in order to detect the largest classes. The following are the top 4 largest classes in the project (excluding test classes and comments):

1. `com.renomad.minum.web.WebFramework` - 415 LOC
2. `com.renomad.minum.htmlparsing.HtmlParser` - 372 LOC
3. `com.renomad.minum.database.` - 219 LOC
4. `com.renomad.minum.web.Response` - 214 LOC

The average number of lines of code is 55 LOC (5), which is considered acceptable. However, the top 4 largest classes should be refactored as they are too large.

2.2 Main Issues and Refactoring Plan

In the following sections, the most important code smells and design issues will be presented, along with a refactoring plan to address them. This allows the reader to understand the current state of the project and the steps that will be taken to improve it.

2.2.1 Codebase Structure

As cited in subsection 1.2, the project codebase is divided into 10 packages, each with a specific purpose. The overall structure of the codebase is showcase a very thorough organization, with each package containing a set of classes and interfaces related to a specific aspect of the library. However, by inspecting in more detail the structure of single packages four main issues were identified:

1. The framework defines many custom exceptions in order to handle specific errors. These exceptions are scattered across the codebase and are not properly organized. This makes it difficult to understand the error handling mechanism of the library.
2. Especially the `util` package, contains many self-contained classes that perform specific tasks. These classes are properly documented but are not properly organized hierarchically. Also, most of these utility classes have multiple responsabilities (i.e, function to convert a string to bytes, and also methods related to authentication).
3. Model classes and interfaces are not properly organized in the `model` package. Each package contains a set of classes and interfaces related to a specific aspect of the library. This makes packages unnecessarily large and difficult to navigate. A better approach would be to use the a layered-architecture (model, service, repository) to organize the codebase.
4. There are test classes inside the `main` package, which should be moved to the `test` package. Some of them (i.e., `FullSystemTests`) have empty methods, without comments or assertions, which should be removed.

2.2.2 Java Conventions, Duplicated Code and General Design Issues

Especially inside test classes, the code is not properly organized, presenting multiple problems such as duplicated code, misuse of Java naming conventions and name shadowing. As test classes are not part of the final library, these issues will not be addressed in the refactoring process.

On the other hand, the main codebase presents problems that must be addressed. The following list summarizes the main issues found in the project:

- Class/interface naming issues: there are classes which do not follow neither the Java naming conventions nor the project naming conventions. For example: `TheBrig`/`ITheBrig`, `SetrfSws`, `MyThread`.
- Reuse of fixed values: there are fixed values used in multiple classes, which should be extracted into constants. For example, inside the `TheRegister` class,
- Even if there is a `Logger` class, there are many `System.err.println` statements in the codebase, which should be replaced by proper logging.

2.2.3 Class-specific issues and Possible Design Patterns usage

`Response` class Inside the *com.renomad.minum.web.Response* class there are different methods to build different kinds of response. This could be a good candiate in order to use the *Factory Method* along with the *Chain of Responsability* pattern, in order to streamline the response building process and make it more modular.

`WebFramework` class The `com.renomad.minum.web.WebFramework` class is the main entry point of the library, and is responsible for handling the HTTP requests and responses after the socket connection. This class is too large (around 845 LOC), and contains many methods that are not directly related to the main responsability of the class.

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.