ANALYZING METHOD DESIGN PATTERNS IN JAVA CLASSES FROM PUBLIC GITHUB REPOSITORIES

Anonymous Author(s)

Pages 17

Contents

1 Introduction

Software development in object-oriented programming (OOP) often relies on the design patterns to address common design problems in a structured and reusable manner. Design patterns provide templates for creating software systems that are efficient, maintainable, and extensible. Among the various types of design patterns, method design patterns play a vital role in defining the behavior and responsibilities of individual methods within a class.

Methods in a class can be classified into two broad categories: manipulators and builders. Manipulator methods modify the internal state of an object, while builder methods construct and return new objects. This categorization helps in organizing and understanding the functionality of methods within a class, ultimately contributing to the overall design quality and code maintainability.

Understanding and analyzing method design patterns in Java classes is important for software developers, as it helps in identifying good coding practices, potential design flaws, and opportunities for code optimization. However, studying method design patterns in a large codebase can be challenging and time-consuming.

With the advent of public code repositories, such as GitHub, it is now possible to access a vast amount of open-source Java projects. These repositories contain an abundance of Java classes that exhibit diverse method design patterns. By analyzing these classes, we can gain valuable insights into the prevalence and characteristics of method design patterns in real-world software systems.

This research paper aims to analyze method design patterns in Java classes obtained from public GitHub repositories. Specifically, we intend to collect a substantial number of Java classes, extract their methods, and classify them into manipulators and builders. By doing so, we can identify common patterns, variations, and usage scenarios of these method design patterns. Furthermore, this research will investigate the relationship between method design patterns and software quality metrics, such as code duplication, complexity, and maintainability.

This research paper is organized into sections detailing a literature review, an overview of Java method design patterns, research methodology, analysis of manipulator and builder methods in Java classes, comparative analysis, discussion of findings, implications and recommendations, conclusion, future work, and references.

2 Literature Review

This section provides an overview of the existing literature on the topic of analyzing method design patterns in Java classes, particularly focusing on the idea that methods in a class must either be manipulators or builders. The literature review is divided into three subsections: (1) Method Design Patterns, (2) Classification Approaches, and (3) Tools and Techniques.

2.1 Method Design Patterns

2.2 Classification Approaches

2.3 Tools and Techniques

3 Overview of Java Method Design Patterns

In object-oriented programming, design patterns are reusable solutions to common programming problems. Method design patterns specifically focus on the design and organization of methods within a class to enhance code readability, maintainability, and flexibility. This section provides an overview of two important Java method design patterns: manipulators and builders.

3.1 Manipulators

Manipulators, also known as mutators or modifiers, are methods in a class that modify the state of an object. They typically receive parameters and update the internal variables or fields of the object based on these inputs. Manipulators are essential for achieving encapsulation and data abstraction in object-oriented programming.

Methods that change the internal state of an object should be carefully designed to ensure data integrity and consistency. As such, manipulators should follow certain principles, such as being self-contained, performing a single logical operation, and validating input parameters. By adhering to these principles, manipulators can contribute to the cohesion and maintainability of the codebase.

3.2 Builders

Builders are methods that facilitate the construction of complex objects by providing an abstraction layer over the construction process. Instead of directly instantiating an object and setting its fields one by one, builders offer a structured and fluent interface for creating objects.

The key idea behind the builder pattern is to separate the construction logic from the object itself, allowing for more readable and flexible code. Builders typically use method chaining to sequentially set the desired parameters of the object being constructed. This approach provides a clear and concise syntax for creating objects with various configurations.

By using builders, developers can simplify the creation of objects with many optional parameters or complex initialization requirements. Builders also promote code reuse and extensibility by allowing subclassing or customization of the construction process.

3.3 Combining Manipulators and Builders

In many cases, methods within a class can be categorized as either manipulators or builders based on their responsibilities. Separating these two types of methods can improve code organization and maintainability. Manipulators handle state changes and modify the internal fields, while builders handle object construction and configuration.

The strict separation of manipulator and builder methods assists in achieving a clear and intuitive class design. This design principle can enhance code understandability, facilitate debugging, and enable code evolution. By adhering to this practice, software engineers can create more structured and maintainable Java codebases.

In the next section, we propose a methodology for analyzing method design patterns in Java classes from public GitHub repositories based on the concepts discussed in this section.

4 Research Methodology

4.1 Data Collection

To analyze method design patterns in Java classes from public GitHub repositories, a large-scale dataset containing Java source code files is required. This dataset needs to cover a diverse range of projects to ensure a representative sample.

The primary source of data for this research will be the CaM dataset. Classes and Metrics (CaM) is a dataset of Java classes from public open-source GitHub repositories that updated periodically with new data.

It is the result of the analysis of Java classes in 1000 GitHub repositories against a number of metrics:

- lines of code (reported by cloc),
- lines of comments,
- blank lines,
- NCSS,
- cyclomatic complexity,
- number of attributes,
- number of static attributes,
- number of constructors,
- number of methods,
- number of static methods,
- total cognitive complexity (reported by PMD),
- maximum cognitive complexity,
- minimum cognitive complexity,
- average cognitive complexity,
- number of committers.

With this dataset, we will be able to track the use of manipulators and builders in the code. We will also calculate the percentage of the use of both principles and check whether they are used correctly.

4.2 Data Preprocessing and Cleaning

AST parsing is a technique used to extract information from code by analyzing its abstract syntax tree. An abstract syntax tree is a tree-like structure that represents the structure of the code, capturing its hierarchical relationships and syntax rules.

In this research paper, the code used in the dataset will be divided into methods by AST parsing. The main goal is to separate the code into smaller units of functionality, which are represented by individual methods. This allows for a more detailed analysis and understanding of the code's logic and behavior.

By parsing the code into methods, authors can focus on specific parts of the code and analyze them in isolation. This can be especially useful when trying to determine the pattern used in the method: manipulator or builder.

4.3 Pattern Identification and Classification

To analyze the method design patterns in the Java classes, a pattern identification and classification process will be employed. This process includes detecting and categorizing the identified design patterns into manipulators or builders.

To identify design patterns, a combination of lexical analysis, abstract syntax tree (AST) parsing, and rule-based matching will be utilized. Lexical analysis will assist in tokenizing the source code, while AST parsing will provide a hierarchical representation of the code's structure. Rule-based matching will be employed to detect specific patterns based on predefined rules and patterns from the literature.

The methods will then be classified as manipulators or builders based on their characteristics and intent. This classification will be based on a combination of static analysis of the code and LLM validation.

Thus, several variants of pattern definition will be created for preprocessing of materials:

- Rule-based code analyzer. To do this, we will go through the code line by line and check whether it returns anything, whether it changes parameters.
- Automatic LM analyzer. To determine the pattern, a request will be made to the LLM model asking what kind of pattern it is.
- Analyzer of the method name. In this case, to determine the pattern used in the method, its name will be checked.

4.4 Evaluation and Validation

The method design patterns identified and classified through the aforementioned process will be evaluated and validated to ensure their accuracy and reliability.

For evaluation, a set of randomly selected Java classes will be manually reviewed by authors to assess the correctness of the identified design patterns. They will compare the results obtained from the automated analysis with their domain knowledge to verify the accuracy of the classification.

To further validate the findings, a survey will be conducted among a group of experienced Java developers. The survey will present them with code snippets containing identified design patterns and ask them to corroborate the classification. The feedback gathered from the survey will provide insights into the agreement between automated analysis and human judgment.

4.5 Analysis and Results

The final step of the research methodology involves analyzing the results obtained from the previous stages and drawing conclusions. The analysis will involve statistical techniques, such as descriptive statistics, to summarize the characteristics of the identified design patterns and their distribution across different projects.

The results will be presented through visualizations, including graphs and charts, to aid in understanding the patterns and their prevalence. Additionally, comparisons between manipulators and builders will be performed to identify any significant differences in their usage patterns.

A discussion of the findings will consider the implications for software engineering practices and provide insights into the suitability and effectiveness of the identified design patterns in the development of Java classes.

4.6 Limitations

It is important to acknowledge the limitations of this research. Firstly, the analysis is limited to Java classes from public GitHub repositories and may not represent the entire landscape of the Java ecosystem. The results may be biased towards popular or frequently starred repositories on GitHub.

Secondly, the reliance on automated analysis and classification techniques introduces the possibility of false positives

or false negatives in identifying design patterns. Expert validation and feedback from experienced developers will mitigate this limitation to some extent.

Despite these limitations, this research aims to provide valuable insights into method design patterns in Java classes and contribute to the existing body of knowledge in software engineering.

5 Analysis of Manipulator Methods in Java Classes

Methods play a significant role in the design and functionality of Java classes. In this section, we analyze the concept of manipulator methods in Java classes and their relevance in software development. We propose that methods in a class must either be manipulators or builders, and explore the implications of this design pattern in practice.

A manipulator method, also known as a mutator method, is a method that modifies the internal state of an object. It typically receives parameters and makes changes to the object's attributes or invokes other methods to perform specific tasks. Manipulator methods are commonly used to implement operations such as setting values, increasing or decreasing attributes, and updating internal data structures.

6 Analysis of Builder Methods in Java Classes

In Java classes, methods can be categorised into various design patterns based on their purpose and functionality. One common design pattern is the builders pattern, which is used to construct complex objects step by step. Builders provide a flexible and readable way to create objects by breaking down the construction process into multiple method calls.

In this section, we focus on analyzing the usage of builder methods in Java classes obtained from public GitHub repositories. The idea behind this analysis is to determine whether the methods in a class exhibit either manipulator or builder behavior.

7 Comparative Analysis of Manipulator and Builder Methods

In this section, we compare the properties and characteristics of manipulator and builder methods in Java classes. As discussed earlier, manipulator methods are responsible for modifying the internal state of an object, while builder methods are used to construct and configure object instances. We focus on understanding the differences and similarities between these two types of methods and identify the benefits and drawbacks of each approach.

7.1 Properties of Manipulator Methods

7.2 Properties of Builder Methods

7.3 Comparison of Manipulator and Builder Methods

8 Discussion of Findings

9 Implications and Recommendations

9.1 Implications

9.2 Recommendations

10 Conclusion

In conclusion, this research paper aimed to analyze method design patterns in Java classes from public GitHub repositories, specifically focusing on the idea that methods in a class must either be manipulators or builders.

11 Future Work

Although this research paper analyzed and identified method design patterns in Java classes from public GitHub repositories, there are several potential directions for future research and improvements. The following subsections outline some possible areas for future work.

11.1 Evaluation of Other Design Patterns

While this work focused on identifying manipulator and builder patterns in Java classes, there still exist various other design patterns that can be analyzed. Future research could explore the detection and categorization of other commonly used design patterns, such as observer, singleton, factory, and adapter patterns. This would provide a more comprehensive understanding of the design patterns employed in publicly available Java code.

11.2 Evaluation of Method Interactions

In this study, the analysis solely considered the patterns based on individual methods within a class. However, method interactions play a crucial role in understanding the overall behavior and function of a software system. Future research could investigate the detection and analysis of patterns that emerge from the interaction between multiple methods, potentially providing deeper insights into the design and architecture of software systems.

11.3 Validation of Patterns

While the pattern identification algorithm used in this study provides promising results, additional validation is necessary to ensure the accuracy and reliability of the identified patterns. Future work could involve employing rigorous validation techniques, such as manual inspection by experts or comparisons with established design pattern repositories, to verify the patterns identified through automated analysis.

11.4 Extension to Other Programming Languages

Although this study focused on Java classes, the approach used to identify design patterns can potentially be extended to other programming languages as well. Exploring the applicability and effectiveness of the proposed methodology in languages like Python, C++, or C# could help broaden the understanding of design patterns in different software development ecosystems.

11.5 Integration with Development Tools

Integrating the pattern identification algorithm into popular Integrated Development Environments (IDEs) and software development tools would greatly benefit software developers. The automated detection and visualization of design patterns directly within the development environment could aid in the understanding, maintenance, and refactoring of software systems.

Overall, this research paper provides a strong foundation for future research on the analysis and understanding of method design patterns in Java classes. The suggested future work directions aim to extend the current research and address important aspects that were not covered comprehensively in this study.

12 References

List of Sources Used