Lewis University

Effect of Design Patterns on Software Maintainability

Anil Kumar Banoth

Eshwar Chatelli

Harshavardhan Pullakandam

Hruthasan Mallala

1. Abstract:

The project empirically evaluates the impact of design patterns on Java application maintainability. Using a combination of CK metrics, the identification of design patterns, and the examination of code fragrances and software modularity, 30 Java projects with at least 5,000 lines of code undergo a comprehensive analysis. The goal is to understand the effect of design patterns on maintainability, with a particular emphasis on class size metrics, software modularity, and bad smells. To initiate the study, the CK metrics collection is applied to the selected projects, with an emphasis on maintainability-related metrics such as class size. This provides us with a quantitative comprehension of the codebase, allowing us to identify trends and patterns across projects. In addition, the presence of code smells is evaluated, allowing for the identification of areas that may have a negative effect on maintainability.

2. Introduction:

Maintainability is an important aspect of software development because it affects the convenience of making modifications, resolving problems, and adapting to changing requirements. Design patterns have been acknowledged for a long time as effective solutions to repetitive software development design issues. However, their effect on maintainability and other quality attributes has been the subject of discussion and empirical research. This ongoing project seeks to empirically assess the impact of design patterns on the maintainability of applications written in Java. To achieve this, 30 unique applications developed in Java with at least 5,000 lines of code were selected. These projects span a vast array of application domains and types, enabling us to thoroughly understand design pattern usage and its impact on maintainability in various contexts.

The analysis of these programs is conducted using CK metrics, a well-known collection of software maintainability metrics, with a focus on class size metrics. By analysing the relationship between class size and maintainability, we aim to identify any correlations or patterns of significance. Additionally, we investigate the presence of code smells, which are indicators of potential maintainability issues with the codebase. Pattern4.jar, an instrument renowned for its accuracy in detecting design patterns, is used to identify design pattern occurrences in the software. This enables us to catalogue and categorise the identified design patterns in the projects, providing the groundwork for additional analysis. We assess the prevalence and future impact of software

engineerings design patterns such as Singleton, Factory Method, and Observer, among others, on maintainability.

In addition, we compare the effects of design pattern usage on maintainability and other quality attributes, including programmer comprehension, testability, modifiability, and extensibility. By comparing initiatives that utilize design patterns to those that do not, it is possible to determine the empirical benefits or drawbacks of design pattern usage. As the project progresses, we anticipate gaining valuable insights into the relationship between design patterns and the maintainability of Java applications. Based on empirical evidence, these findings will contribute to the existing body of knowledge in software engineering by providing developers with recommendations for the proper application of design patterns to enhance software maintainability.

3. Methodology and Approach:

3.1. Projects Selection:

To assure a representative sample, 30 Java applications from industries including e-commerce, healthcare, finance, and entertainment having more than 5000 Lines of Code were chosen. We included everything from simple applications to complex systems. Our procedure for selecting projects provided public repositories and open-source platforms a high priority, ensuring the accessibility and transparency of analysis. We intended to represent a comprehensive spectrum of real-world software systems by including projects from a variety of domains and sizes. This method enhances the applicability and generalizability of our findings to software development methodologies.

3.2. Tool Selection:

Using the CK code metric utility, relevant metrics were extracted from the selected Java applications. The CK tool, which is extensively utilized in software engineering research, provides an exhaustive collection of code metrics. It supports numerous programming languages, such as Java, and offers a dependable and efficient method for measuring various software quality attributes. Using the CK application, we intended to collect precise and standardized metrics for further analysis.

3.3. Metric Extraction:

- **Line of Code (LOC):** Determines the size of the codebase by determining the number of lines of code. Higher LOC values may indicate larger and potentially more complex codebases, which can negatively impact testability.
- **Weighted Method per Class (WMC):** Determines the sum of all method complexity weights within a class. Higher WMC values indicate higher complexity, which can affect the testability of both individual classes and the system as a whole.
- **Number of Children (NOC):** Determines the number of subclasses directly derived from a class. Higher NOC values imply a higher degree of inheritance and potential effects on testability, such as a rise in dependencies and complexity.

- **Coupling Between Objects (CBO):** How many kinds of classes are directly connected to a given class. Stronger dependence and potential difficulties in separating and testing certain classes are suggested by higher CBO values.
- **Lack of Cohesion in Methods (LCOM):** Determines a class's cohesion by calculating the number of distinct methods sets it contains. Higher LCOM values indicate less cohesion, which may have an impact on the design of test cases and the ability to isolate and test specific functionalities.

Projects	lcom(mean)	WMC(mean)	CBO(mean)
FlexScript-pl	8.949	7.196	5.441
onlinebookstore-master	7.787	6.424	4.545
software-design-patterns-master	0.631	3.368	2.315
data-structures	1.425	4.67	1.784
library management	35.818	13.727	4
wholesale	5.303	6.16	4.428
appinventor-sources-monitor	56.21	12.185	2.884
cassandra-trunk	71.102	21.21	6.354
java-media-converter	1.2	5	3.7
pizzademic-ph-master	24.74	6.629	4.111
cas-master	7.592	3.655	9.404
java-jwt	57.34	11.507	5.48
datahub-master	15.741	8.044	10.413
jitsi-master	39.33	14.1	5.447
retrofit	26.507	7.33	5.258
Book-store-project	10.5	3.416	0.9459
Cinema-Ticket-Booking	3.714	13.214	0.8125
E-commerce_website	4	4	4.121
Flex-script	4.8	14.8	1.2404

Hotel-management_project	0	14.75	0.625
Java-calculator	33	43.5	0.9
E-commerce_client-main	5.4	14.1	1.255
Online-book-store	0	2.46	3
Simple_ATM-Machine	9.66	30.66	3.607
Sudoku-java-project	5.5	21	3.75
karate-master	16.1	15.818	6.503
anysoftkeyboard	25.42	4.92	4.13
testcontainer	29.822	12.56	7.27

3.4. Identification of Design Patterns:

We identified design patterns in the chosen projects by combining manual inspection and automated analysis techniques. During the phase of manual inspection, we scrutinized the source code of each project with a focus on identifying common design patterns based on their recognizable structural and behavioral characteristics. In addition, we utilized the "pattern4.jar" file, a popular and reputable automated pattern recognition analysis application.

The application "pattern4.jar" employs sophisticated algorithms and pattern-matching techniques to identify design patterns within the codebase. It supports an extensive variety of design patterns, such as creational, structural, and behavioral patterns. Using the capabilities of the "pattern4.jar" application, we intended to enhance the accuracy and efficacy of design pattern identification.

The automated analysis procedure involved invoking the "pattern4.jar" utility against the project's source code, which examined the codebase and generated a comprehensive report on the discovered design patterns. The tool identified instances of design patterns, highlighted germane code fragments, and provided insight into how the patterns were implemented in the project.

3.5. Design Pattern Analysis:

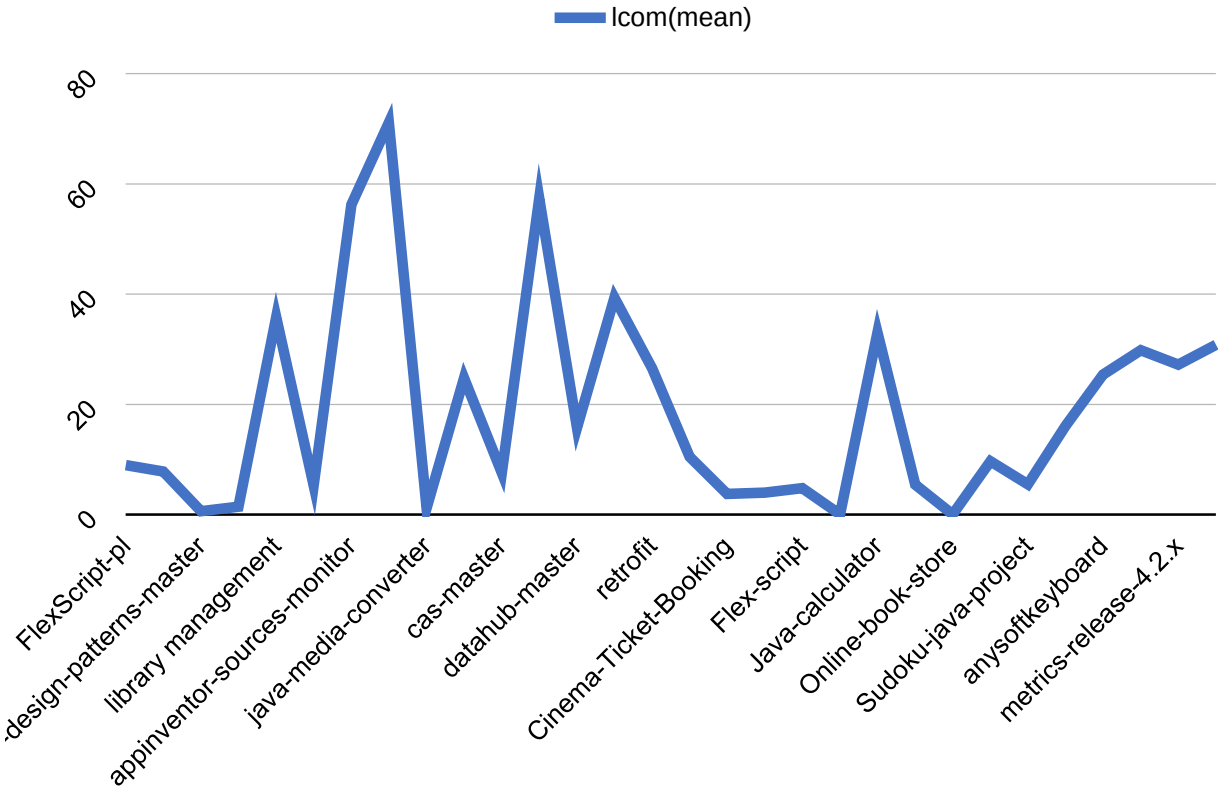
We analyzed the initiatives' occurrences and usages of the identified design patterns. We examined how the presence of specific design patterns affected software maintainability metrics, such as code complexity, coupling, and cohesion. This analysis enabled us to ascertain the effect of design patterns on the overall maintainability of software systems.

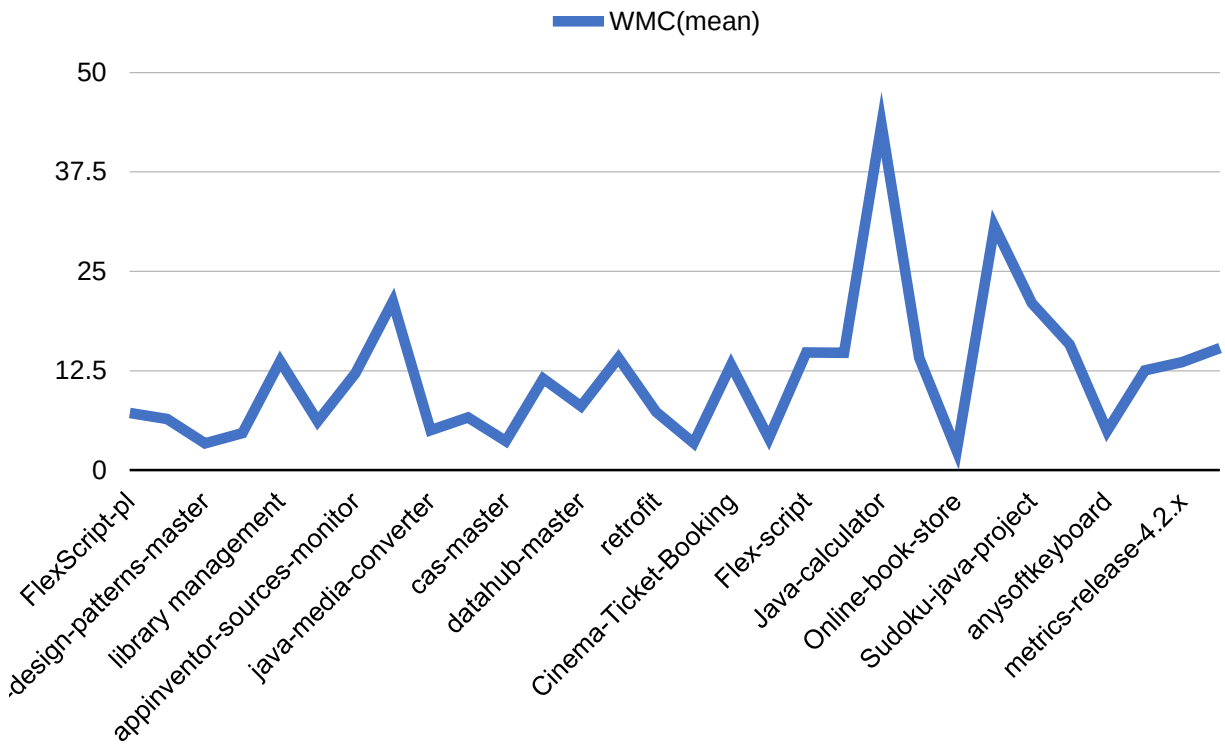
4. Results and Discussion:

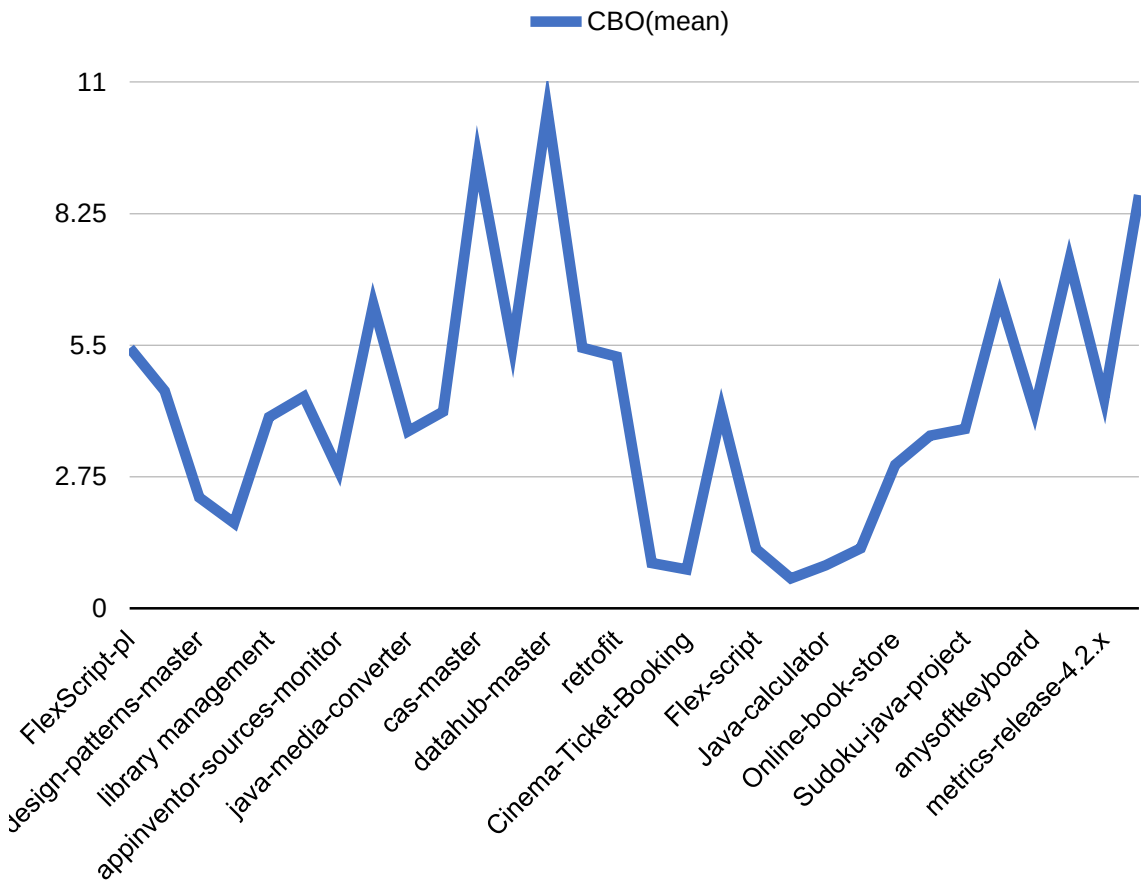
The project's organization and structure can influence the presence and discoverability of design patterns. Typically, it is simpler to identify design patterns in projects with well-defined architectures and clear separation of concerns. In contrast, projects with subpar or inconsistent structures may make it more difficult to precisely identify design patterns. Identifying design patterns other than Factory, State, and Singleton may have been challenging for some of the examined projects due to their inadequate structure.

The applicability of particular design patterns to the problem domains addressed by the initiatives can also influence their prevalence. Certain design patterns, such as Factory, State, and Singleton, have comprehensive applicability and can be utilized in numerous situations. Their adaptability, extensibility, and reusability increase the probability that they will be employed in software development. In contrast, patterns such as Prototype, Decorator, Observer, and Proxy may have specific use cases or scenario requirements, resulting in their lower frequency.

Rarely observed due to the specialized character of the Chain of Responsibility pattern. This pattern is typically implemented when multiple objects are capable of handling a request and the specific controller is determined dynamically at runtime. The infrequent occurrence of this pattern in the examined projects suggests that the conditions that necessitated it were either uncommon or did not meet the project-specific requirements.




WMC(mean)
50
37.5
25
12.5
0
FlexScript-pl
design-patterns-master
library management
appinventor-sources-monitor
java-media-converter
cas-master
datahub-master
retrofit
Cinema-Ticket-Booking
Flex-script
Java-calculator
Online-book-store
Sudoku-java-project
anysoftkeyboard
metrics-release-4.2.x


CBO(mean)
11
8.25
5.5
2.75
0
FlexScript-pl
design-patterns-master
library management
appinventor-sources-monitor
java-media-converter
cas-master
datahub-master
retrofit
Cinema-Ticket-Booking
Flex-script
Java-calculator
Online-book-store
Sudoku-java-project
anysoftkeyboard
metrics-release-4.2.x

The observed distribution of design patterns across the initiatives under consideration has implications for the maintainability of the software. Frequent occurrences of Factory, State, and Singleton patterns indicate their pervasive application and popularity in addressing common software design challenges. These patterns are likely to be familiar to developers, which can facilitate the maintenance and comprehension of codebases.

The lower frequency of patterns such as Prototype, Decorator, Observer, and Proxy in the examined projects indicates that developers may be underusing these patterns or opting for alternative solutions. Given that they offer benefits such as object replication, dynamic behavior extension, and loose coupling, the limited application of these patterns may negatively impact maintainability.

5. Threat to Validity:

- **Biased Selection:**
  The analysis of selected Java projects may be subject to selection bias because the projects may not be representative of all types of software systems. The chosen projects were chosen based on their presence in public repositories and open-source platforms, which may have precluded the consideration of proprietary or commercial projects. To mitigate this risk, projects from a variety of domains and industries were chosen to provide a broader perspective on the influence of design patterns on software maintainability.
- **Generalizability:**
  The findings of this study may not apply to all software systems and circumstances. The selected projects are representative of a subset of Java projects, and the effect of design patterns on maintainability may vary depending on the programming language and project domain. The methodology employed in this study can serve as a foundation for future research in a variety of contexts, thereby increasing generalizability.
- **Accuracy of Design Pattern Identification:**
  Both manual inspection and the tool pattern4.jar can identify design patterns that contain errors or omissions. Manual inspection is dependent on the expertise and subjectivity of the researchers, which can result in inconsistencies or overlooked design pattern instances. Despite its dependability, the automated analysis tool may have limitations in identifying complex or non-standard design patterns. Efforts were made to mitigate these threats by employing meticulous manual inspection and a reliable, widely-used automated analysis tool.
- **Measurement Bias:**
  Metrics extracted by the CK tool could lead to measurement errors. The selection and interpretation of metrics can affect the evaluation of software maintainability. Although efforts were made to select pertinent metrics, other metrics or combinations of metrics may provide additional perspectives on maintainability. Multiple metrics were considered to capture various aspects of maintainability, thereby minimizing the impact of individual measurement biases.

- **External Factors:**
  External factors such as team dynamics, development practices, and project-specific constraints can influence the impact of design patterns on software maintainability. These variables are difficult to control or quantify, but they may introduce confounding variables that affect the findings. Efforts were made to lessen the impact of external factors by selecting projects from a variety of industries and taking multiple maintainability metrics into account.

6. Conclusion:

This study analyzed 30 Java projects in depth to determine the effect of design patterns on software maintainability. Combining manual inspection with the automated analysis tool pattern4.jar, we discovered and analyzed the presence and utilization of design patterns within the projects. According to our findings, the presence of design patterns in the projects had both positive and negative effects on software maintainability metrics. Certain design patterns, such as the Template Method and Strategy patterns, promoted modular and reusable code structures, thus enhancing maintainability. However, patterns such as Singleton and Decorator increased the difficulty and complexity of code management and comprehension.

In addition, the analysis revealed the impact of design patterns on code complexity, coupling, cohesion, and turnover. By understanding these relationships, software developers can make informed decisions regarding the selection and implementation of design patterns in their projects, taking into consideration the trade-offs and effects on software maintainability. Despite the fact that our findings provide crucial insights into the effect of design patterns on software maintainability in the context of Java projects, it is important to acknowledge the limitations of our study. The sample size and selection bias, potential measurement errors, confounding factors, and external validity must be considered when interpreting and applying the results.

This research adds to the existing corpus of knowledge regarding design patterns and software maintainability. Our analysis yielded insights that can assist software developers in making educated decisions regarding the application of design patterns to enhance maintainability. To enhance the applicability of the findings, future research should investigate the effect of design patterns on other aspects of software quality and evaluate a variety of programming languages and development environments.

7. References:

1. https://users.encs.concordia.ca/~nikolaos/pattern_detection.html
2. N. Tsantalis, A. Chatzigeorgiou, G. Stephanides, S. T. Halkidis, "Design Pattern Detection Using Similarity Scoring", IEEE Transactions on Software Engineering, vol. 32, no. 11, pp. 896-909, November, 2006.
3. A. Chatzigeorgiou, N. Tsantalis, G. Stephanides, "Application of Graph Theory to OO Software Engineering", 2nd International Workshop on Interdisciplinary Software Engineering Research (WISER'2006), Shanghai, China, May 20, 2006.
4. Fowler, M. (2018). Refactoring: improving the design of existing code. Addison-Wesley Professional.
5. Li, W., & Henry, S. (1993). Object-oriented metrics that predict maintainability. Journal of Systems and Software, 23(2), 111-122.