Decoding the Connection between Design Patterns and Software Maintainability: An Empirical Analysis

Andrew James Milligan
Department of Computer and Mathematical Sciences
Lewis University
Romeoville, United States

***Abstract*—This study examines the influence of design patterns on software maintainability using four key metrics—Weighted Methods per Class (WMC), Lack of Cohesion in Methods (LCOM), Depth of Inheritance Tree (DIT), and Response for a Class (RFC). Analyzing 30 software programs, half employing design patterns, the research presents a nuanced view of software metrics distribution. Preliminary findings suggest that design patterns may improve maintainability, as indicated by lower WMC, DIT, RFC scores, and higher LCOM scores. However, significant variance across programs reveals the complex relationship between design patterns and maintainability. Despite potential validity threats like selection bias and confounding variables, this research offers valuable insights into the role of design patterns in software maintainability and sets the stage for further study.**

This paper embarks on an exploration of the impact of design patterns on the maintainability of software systems, driven by the growing emphasis on maintainability in the realm of software development. This is especially pertinent considering the fact that maintenance often consumes a significant portion of the software lifecycle, both in terms of time and resources. The focus of this study centers on four critical metrics, namely Weighted Methods per Class (WMC), Lack of Cohesion in Methods (LCOM), Depth of Inheritance Tree (DIT), and Response for a Class (RFC), as the yardsticks for gauging maintainability. The paper is structured to provide the readers with a holistic perspective of the research journey. Following the introduction and formulation of the Goal-Question-Metric (GQM) approach, it delves into the specifics of each metric, highlighting their significance. The paper then elucidates the selection criteria and the software programs chosen for the study, half of which use design patterns and the other half do not. After discussing the tools deployed and the data collection process, the research presents its findings in two segments, dedicated to the software programs with and without design patterns. This is followed by an analysis of the potential threats to the validity of the research, culminating in the conclusions drawn from the aggregate data.

In this study, the overarching goal is to explore the impact of design patterns on a program's maintainability empirically. To this end, the focus is on answering the following questions: 1) Do design patterns inherently decrease the complexity of classes and the diversity of possible responses from a class as represented by WMC and RFC metrics respectively? 2) How does the use of design patterns influence the extent of inheritance in the selected programs as reflected by the DIT scores? 3) Do programs implementing design patterns invariably showcase better cohesion, leading to more maintainable code? The chosen metrics—WMC, DIT, RFC, and LCOM—are essential for interpreting software complexity, depth of inheritance, class responsiveness, and cohesion, all of which contribute to software maintainability. By investigating these metrics, this research provides insights into the nuanced role of design patterns in enhancing software maintainability.

After settling on research goals, questions, and ways to measure the results of the study to be performed the next step in this research process was to define how subject programs would be selected to produce the best, most conclusive and precise data.

The selection criteria for the subject programs were strategically designed to ensure a robust and comprehensive study. First and foremost, the selected programs needed to be written in Java, ensuring a consistent language base for evaluating design patterns and reducing variation due to language differences.

Furthermore, the requirement for half of the subject programs of having implemented at least one of the GoF (Gang of Four) design patterns ensured that the study would include programs that were intentionally using design patterns, providing a critical baseline for comparison.

The size and age criteria, requiring programs to be of at least 5,000 lines of code and two years old,

respectively, were significant in making sure the selected programs were not trivial or immature. These criteria helped in providing a meaningful evaluation of the maintainability of the programs.

Programs with an active maintenance record, a minimum of 500 commits, and preferably at least 52 commits within the past year, were chosen to ensure that the study was based on living, changing codebases, which are more indicative of real-world software development scenarios.

A requirement of at least three contributors was also important, as it ensured that the program's code had been subjected to multiple coding styles and perspectives, which is typical in a development environment and could impact maintainability.

Only open-source programs were selected, providing transparency and full accessibility to the code, a crucial aspect for a thorough investigation.

Lastly, choosing programs from at least five different domains enriched the study by reflecting the broad applicability of design patterns across diverse areas of software development. This diversity added robustness to the study by reducing the likelihood that results would be skewed by domain-specific factors.

Following the defining of the selection criteria for the subjects of this study, the subjects themselves had to be carefully selected. The subject programs chosen for the study perfectly exemplify a diverse set of real-world projects, each meeting the defined selection criteria.

The "Apache Commons IO" and "Java Websocket" libraries, each more than a decade old, serve as critical infrastructure components in the software landscape, providing IO functionality and WebSocket implementations respectively. With over 4000 commits and almost 100 contributors each, they aptly reflect the selection criterion of being actively maintained.

"Apache Commons Lang," another utility library from the Apache Commons project, along with "Guava," a core library set from Google, are both over a decade old and have a robust set of contributors and numerous commits, demonstrating their long-standing value and active maintenance in the community.

"JFreeChart," a free chart library and "Joda-Time," a popular replacement for Java's date and time classes, have been actively maintained for over seven years, demonstrating their relevance and widespread use. Similarly, "Jsoup," a real-world HTML parser, and "JSQLParser," a SQL parser, are more than three years old and have been actively maintained with a considerable number of commits and contributors.

"JUnit4," a popular testing framework for Java, and projects like "HDRHistogram," "Infinity for Reddit," "Priam," and "Dynmap," each with their own distinct purpose, ranging from data value recording to community platforms, all meet the criteria of active maintenance, age, and the number of contributors.

Furthermore, the study includes a range of projects with unique domains and functionalities such as "kSQL," "Minecraft Forge," "JavaParser," "Javapoet," "Mantis," "Botania," "Chronicle-Map," "Picocli," "Dependency-Track," "Keepass2Android," "AppManager," "SmartTubeNext," "NewPipe," "Aeron," "AmazeFileManager," "Mockito," and "Serve." Each of these, with their varying age, commits, and contributors, presents a unique perspective on the use of design patterns in various software domains, and together they paint a broad picture of the software landscape.

Below are tables with some quick-reference information for each project.

TABLE I. SUBJECTS USING PATTERNS

Program	Design Pattern?	Age	Commits	Contributors
Apache Commons IO	Y	16+ years old	4.2k+	99
Apache Commons Lang	Y	16+ years old	7.1k+	190

Guava	Y	13+ years old	6.1k+	289
HDRHistogram	Y	4+ years old	764	39
Java Websocket	Y	13+ years old	1.1k+	77
JFreeChart	Y	7+ years	4.2k+	25
Joda Time	Y	7+ years old	2.2k+	90
Jsoup	Y	3+ years old	1.8k+	104
JSQLParser	Y	10+ years old	1.8k+	113
Junit4	Y	19+ years old	2.5k+	154
Infinity for Reddit	Y	4+ years old	2k+	47
Priam	Y	11+ years old	1.5k+	50
Dynmap	Y	5+ years old	3.1k+	107
kSQL	Y	5+ years old	15k+	176
Minecraft Forge	Y	12+ years old	8.5k+	469

JavaParser	Y	8+ years old	8.7k+	182
Javapoet	Y	10+ years old	800+	80

TABLE II. SUBJECTS NOT USING PATTERNS

Program	**Design Pattern?**	**Age**	**Commits**	**Contributors**
Java-Websocket	N	10+ years old	1.1k+	77
jfreechart	N	9+ years old	4.2k+	25
Mantis	N	2+ years old	900+	24
Botania	N	8+ years old	9.6k+	185
Chronicle-Map	N	9+ years old	3.3k+	48
Picocli	N	6+ years old	4.4k+	119
Dependency-Track	N	5+ years old	3.8k+	88
Keepass2Android	N	7+ years old	4.2k+	44
AppManager	N	3+ years old	4.9k+	221
SmartTubeNext	N	3+ years old	7.8k+	77
NewPipe	N	7+ years old	10.7k+	793

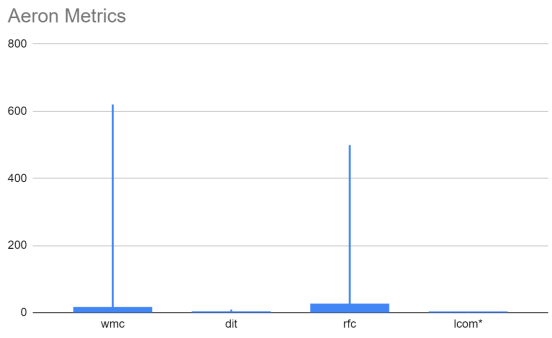
Aeron	N	6+ years old	16.6k+	103
AmazeFileManager	N	8+ years old	6k+	132
Mockito	N	14+ years old	5.9k+	276
Serve	N	3+ years old	3.4k+	134

Having defined the selection criteria and actually selecting the subjects for this study the next step was to choose the tools to collect the necessary data. The research employed two tools for data collection - the CK tool and the Design Pattern detection tool. The CK tool, developed by Maurício Aniche, is a robust software analysis tool known for its efficacy in extracting object-oriented software metrics from Java source code. Aniche's tool, available for use via GitHub, can calculate a variety of metrics such as Weighted Method Count (WMC), Lack of Cohesion in Methods (LCOM), Depth of Inheritance Tree (DIT), and Response For a Class (RFC). These metrics are instrumental in providing empirical insights into the maintainability of the codebase, making the CK tool a fitting selection to measure and quantify the impact of design patterns on the maintainability of the evaluated programs.

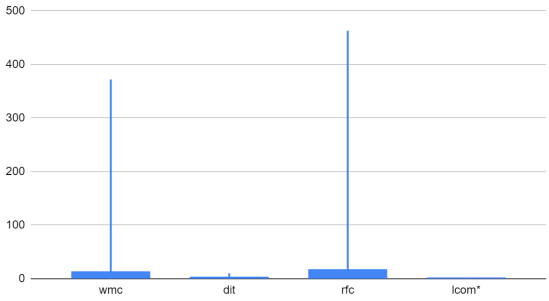
The Design Pattern detection tool, on the other hand, is a product of thorough research from the team at Concordia University. It is currently maintained in its version 4.13 - build 25/02/2020, and can be accessed through the university's domain. This tool has the critical functionality to identify the use of specific design patterns within the codebase. It systematically recognizes and catalogs the use of different Gang of Four (GoF) patterns, thus, offering detailed data about the implementation of design patterns. Its precise tracking allows for an extensive examination of design pattern usage across diverse domains and project scales.

The combination of these tools, each with its unique strengths, has enabled the researchers to assemble a comprehensive and rigorous dataset. This selection of tools was instrumental in conducting an empirical evaluation, shedding light on the impact of design patterns on software maintainability.

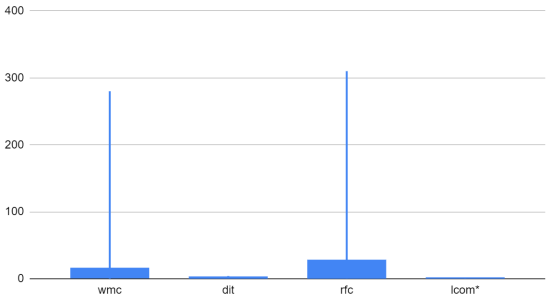
Upon the completion of the other critical steps in this study, defining the goals and questions, defining selection criteria for subjects and choosing them, and selecting appropriate tools for data collection from these subjects, the next step was to actually collect the data. The data collection process consisted of the following steps: first each selected project was downloaded onto a personal computer. Next, all the projects were compiled and or built as necessary to be able to be targeted by the Design Pattern detection tool and each subject using and not using design patterns was validated and placed into the appropriate category. Following this step, each program had the “CK” tool run on them to determine the appropriate metrics for the program’s code. This data was then cleaned and placed into visual representations of the data found below this paragraph. Candlestick charts were selected for the visual representation of this data due to their capacity to represent multidimensional data in a digestible format. Their graphical nature facilitates clear visual distinctions that can enhance the understanding of range and directionality of data values, contributing to a thorough yet comprehensible depiction of the findings. First among the provided visualizations are the subject programs that are not using any design patterns.



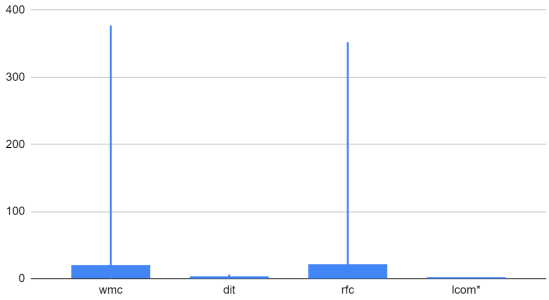
Amaze Metrics



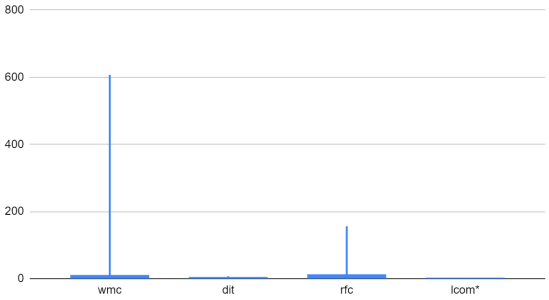
Dependency Track Metrics



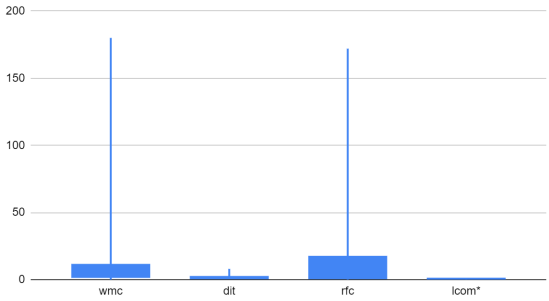
Appmanager Metrics



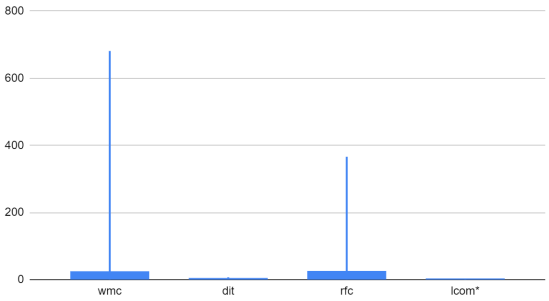
Java Websocket Metrics



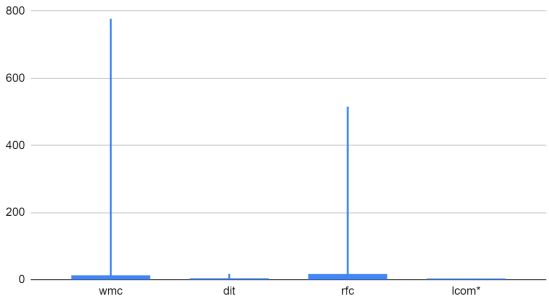
Botania-metrics



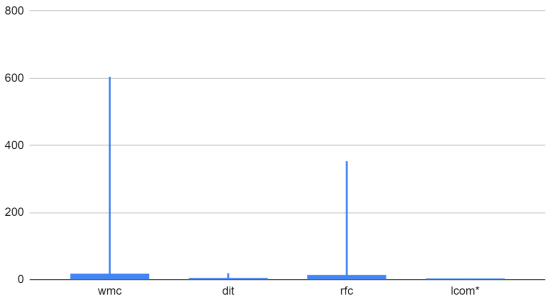
Jfreechart Metrics

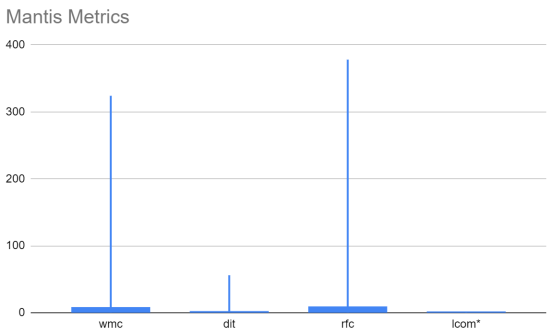


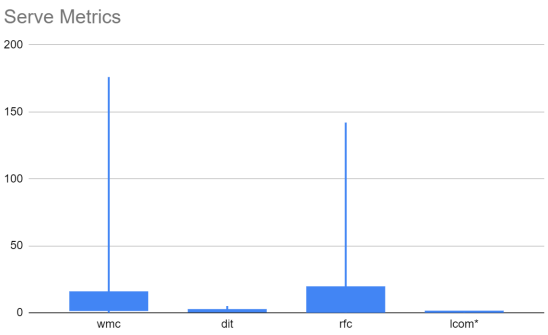
Chronicle Map Metrics

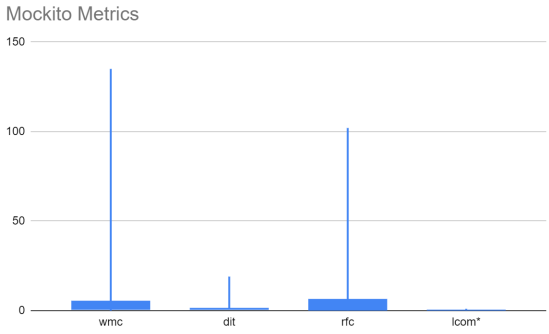


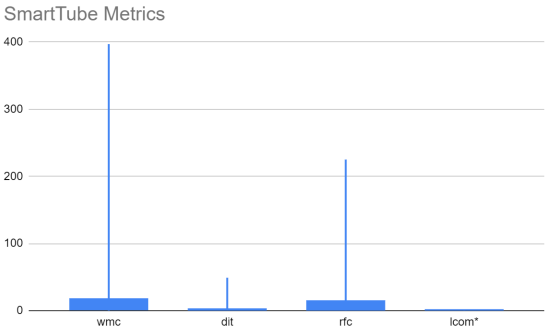
Keepass2android Metrics

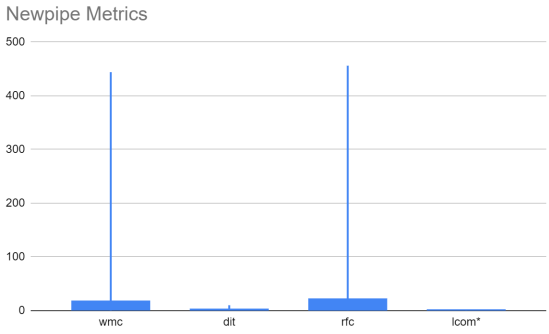


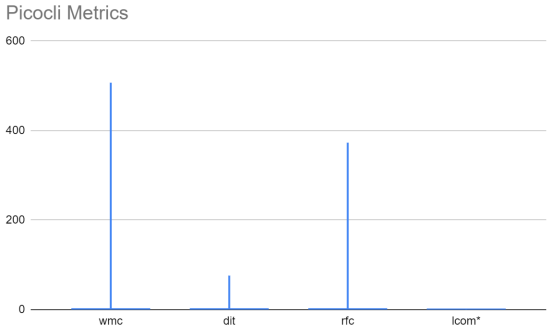












Analyzing the data from programs not utilizing design patterns offers interesting findings relevant to software maintainability. The Weighted Method per Class (WMC) metrics widely vary across the projects, indicating a diversity in class complexity. This might suggest that without the structure provided by design patterns, methods could be unevenly distributed across classes, potentially hindering maintainability.

The Depth of Inheritance Tree (DIT) metrics is consistently low for most programs, typically having a median of 1. This implies that the software has a flat class hierarchy, which generally aids in comprehensibility and maintainability. However, the absence of design patterns may be limiting the use of inheritance, which when appropriately used, can enhance code reusability.

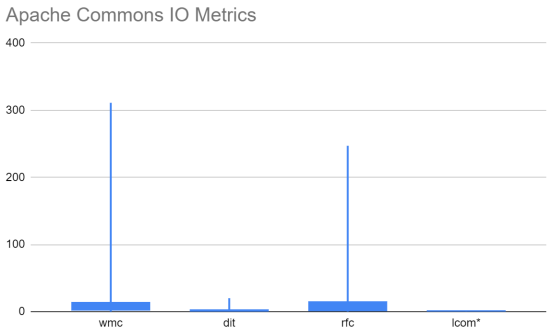
The Response for a Class (RFC) shows a wide range of values among the programs. Some programs display a high RFC, pointing to a large set of distinct methods that can be invoked. This might make the program harder to maintain due to increased complexity and testing requirements.

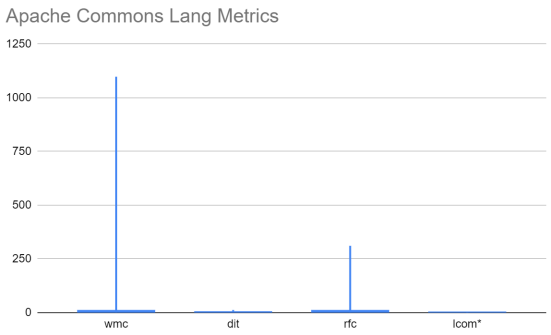
Regarding the Lack of Cohesion in Methods (LCOM), a majority of programs have lower median values. Lower LCOM values indicate better cohesion, which is desirable for maintainability. However, a few programs display higher LCOM scores,

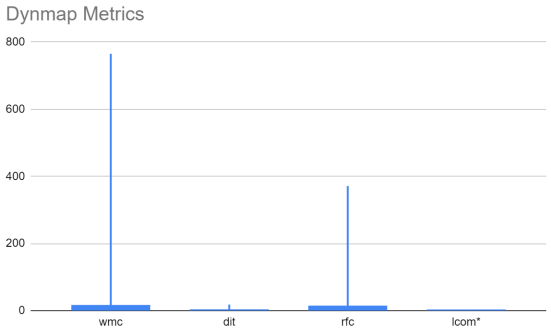
indicating potential design issues and less maintainability.

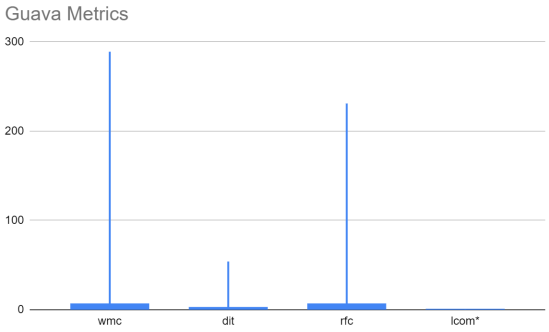
These findings suggest that the absence of design patterns in a program can lead to a wide range of metrics results, indicating varied levels of maintainability. Without the use of design patterns, the organization of methods among classes and the use of inheritance may not be as optimal, leading to potential maintainability issues.

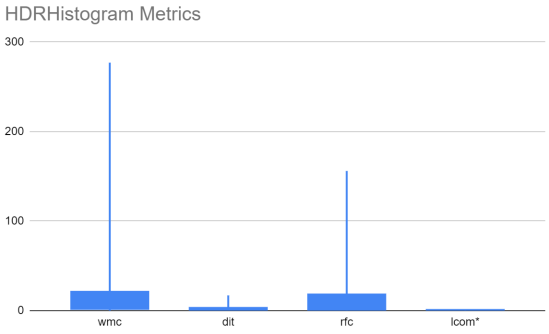
Next are the visualizations detailing the data collected for each metric from the subjects found to be using design patterns.

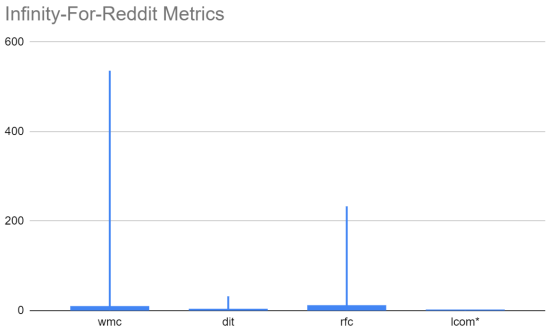


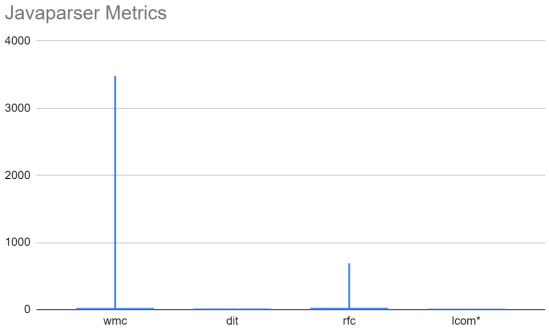


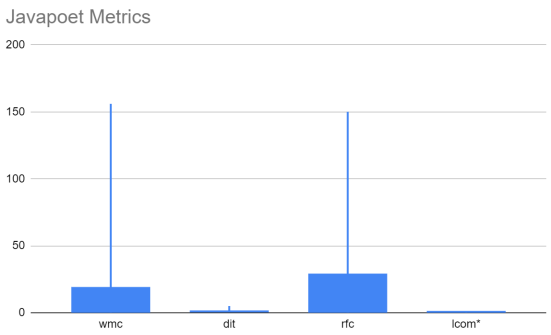


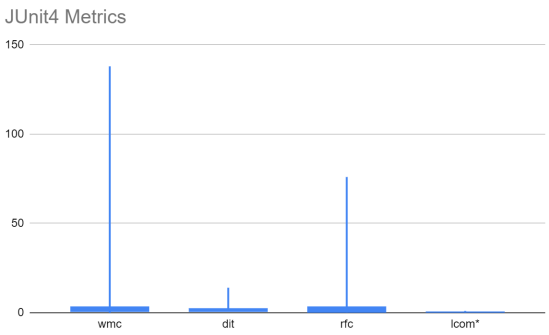


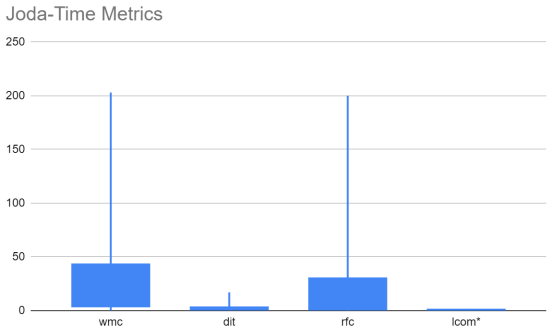


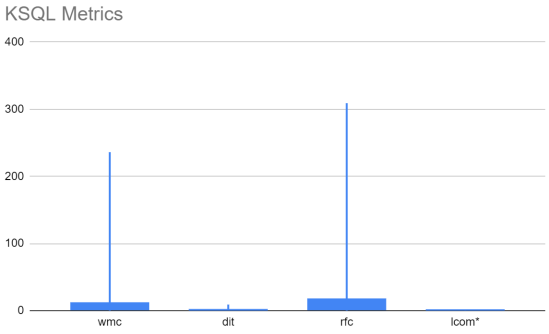


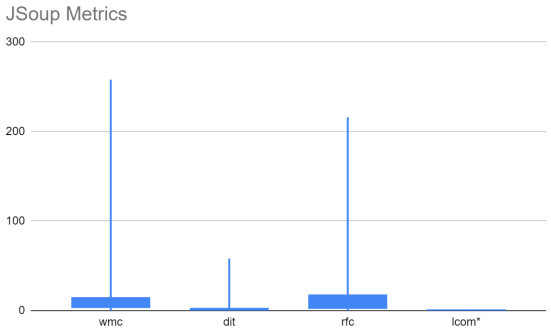


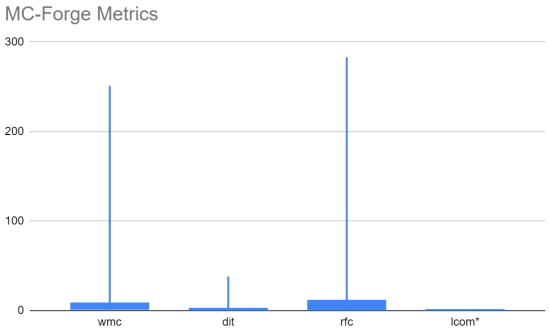


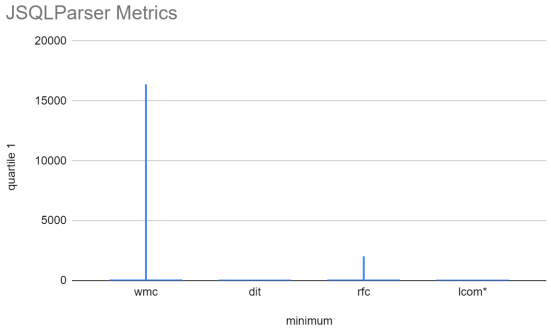


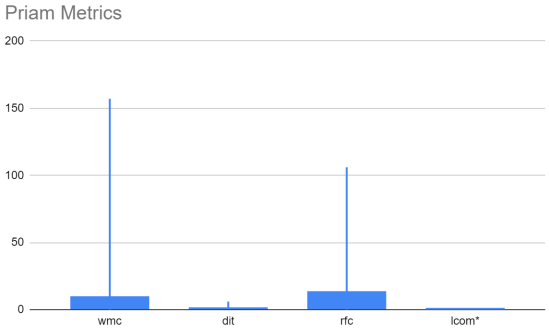












Examining the data for the programs implementing design patterns, there are several notable patterns. For the Weighted Method per Class (WMC), there is considerable variation across different programs, similarly to the non-design patterns data set. However, some programs like "Joda Time" and "jsqlparser" show significantly high WMC

values, suggesting a possible influence of design patterns on class complexity.

Looking at the Depth of Inheritance Tree (DIT), the majority of programs maintain a shallow class hierarchy, indicating that even with the use of design patterns, inheritance is used sparingly. This could still contribute to improved comprehensibility and maintainability, albeit potentially underutilizing the advantages of inheritance for code reusability.

When it comes to Response for a Class (RFC), the range remains extensive across different programs. This suggests that the use of design patterns does not necessarily limit the number of distinct methods that can be invoked, and maintainability may still be impacted by increased complexity and testing requirements.

As for the Lack of Cohesion in Methods (LCOM), a majority of programs have relatively low LCOM values, pointing towards better cohesion and potentially higher maintainability. Notably, some programs, like "Joda Time", display high LCOM values, suggesting that the use of design patterns may not always guarantee better cohesion.

The use of design patterns may be linked to better software structure and potential maintainability, but the actual impact can vary significantly depending on the specific implementation within each program. The wide range of results from the data collected for this study suggest that design patterns, while providing a structured framework, may not necessarily ensure enhanced maintainability across all aspects.

After addressing the findings of the study and before any conclusions are drawn from the data, there are several potential threats to the validity of this study, given the methodology and selection criteria for the study subjects. First, there's a selection bias risk, as the programs chosen for the study may not represent a broad enough spectrum of software projects. The representativeness of the selected subjects could significantly impact the external validity of the study—how generalizable the findings are to other contexts, software projects, and perhaps other programming languages.

Second, there's the risk of confounding variables. In the collected data, other factors, apart from the usage or non-usage of design patterns, might influence the WMC, DIT, RFC, and LCOM scores. For example, the skill level of the programmers, the size of the software, and other practices in the development process could have influenced these metrics. These confounding factors, if not controlled, pose a threat to the study's internal validity.

Third, the method of data collection can also pose a threat to validity. The data for this study was cleaned manually, so there's a risk of human error, introducing bias into the results. The accuracy of the tools used for data extraction and analysis could also be a concern, as these tools may not have been able to determine every metric with complete accuracy.

Lastly, the use of candlestick charts for data representation might present an oversimplification of the complex data relationships. Candlestick charts may not adequately capture outliers or the distribution shape, both of which could provide vital insights. Furthermore, interpreting these charts requires a certain level of expertise, and misinterpretation could lead to erroneous conclusions.

While the study provides valuable insights into the potential impact of design patterns on software maintainability, these threats to validity should be carefully considered when interpreting the results and making broader conclusions. Future research could aim to mitigate these threats by using a larger, more representative sample, controlling for confounding variables, ensuring accurate data collection, and employing a variety of data visualization tools.

When considering both data sets for programs using design patterns and those not using them, several conclusions can be drawn about the impact of design patterns on software maintainability. First, the use of design patterns does not necessarily translate into a uniform impact on the four metrics—WMC, DIT, RFC, and LCOM—that determine maintainability. For instance, both groups exhibit a wide range in WMC and RFC values, suggesting that design patterns do not inherently decrease the complexity of classes or the diversity of possible responses from a class. It can then be reasoned that while design patterns provide a systematic approach to problem-solving, they do not guarantee reduced complexity.

On the other hand, the DIT metric, which measures the depth of inheritance, is generally low across both groups, indicating that irrespective of the use of design patterns, inheritance is not heavily used. A lower DIT score can suggest more manageable code since a deep inheritance tree can increase complexity, but it also implies that the programs

might not fully leverage the benefits of inheritance for reusability and modularity.

Regarding LCOM, it appears that programs using design patterns tend to have better cohesion, potentially leading to more maintainable code. However, some exceptions exist with high LCOM values, signifying that the use of design patterns alone does not guarantee enhanced cohesion.

Design patterns can offer a structured approach that has the potential to improve software maintainability, their effectiveness relies heavily on the specific context and implementation within each program. The usage of design patterns should be accompanied by good programming practices and judicious decisions to balance complexity and maintainability. The empirical evidence suggests that the application of design patterns should be a thoughtful choice rather than a universally applied rule for software maintainability.

REFERENCES

[1] Arun Agrawal, “Priam”, 2012, Available in https://github.com/Netflix/Priam
[2] Chris Povirk, “Guava”, 2023, Available in https://github.com/google/guava
[3] Christian Schabesberger, “NewPipe”, 2016, Available in https://github.com/TeamNewPipe/NewPipe
[4] David Gilbert, “jfreechart”, 2023, Available in https://github.com/jfree/jfreechart/releases/tag/v1.5.4
[5] Docile Alligator, “Infinity-For-Reddit”, 2019, Available in https://github.com/Docile-Alligator/Infinity-For-Reddit
[6] Federico Tomassetti, “Javaparser”, 2015, Available in https://github.com/javaparser/javaparser
[7] Gary Gregory, “Apache Commons IO”, 2023, Available in https://github.com/apache/commons-io
[8] Gary Gregory, “Apache Commons Lang”, 2021, Available in https://github.com/apache/commons-lang
[9] Gil Tene, “HDRHistogram”, 2019, Available in https://github.com/HdrHistogram/HdrHistogram
[10] Harsh Bafna, “Serve”, 2020, Available in https://github.com/pytorch/serve
[11] Jake Wharton, “Javapoet”, 2013, Available in https://github.com/square/javapoet
[12] Jeff Chao, “Mantis”, 2021, Available in https://github.com/Netflix/mantis
[13] Jim Galasyn, “KSQL”, 2018, Available in https://github.com/confluentinc/ksql
[14] Jonathan Hedley, “jsoup”, 2023, Available in https://github.com/jhy/jsoup
[15] LexManos, “MinecraftForge”, 2011, Available in https://github.com/MinecraftForge/MinecraftForge
[16] Marcel Prestel, “Java Websocket”, 2022, Available in https://github.com/TooTallNate/Java-WebSocket
[17] Marc Philipp, “junit4”, 2021, Available in https://github.com/junit-team/junit4
[18] Martin Thompson, “Aeron”, 2017, Available in https://github.com/real-logic/aeron
[19] Maurício Aniche, “Java code metrics calculator (CK)”, 2015, Available in https://github.com/mauricioaniche/ck
[20] Mike Primm, “Dynmap”, 2018, Available in https://github.com/webbukkit/dynmap
[21] Muntashir Al-Islam, “AppManager”, 2020, Available in https://github.com/MuntashirAkon/AppManager
[22] Nikolaos Tsantalis, “Design Pattern detection Tool ”, 2020, Available in https://users.encs.concordia.ca/~nikolaos/pattern_detection.html
[23] Philipp C., “Keepass2android”, 2016, Available in https://github.com/PhilippC/keepass2android
[24] Remko Popma, “Picocli”, 2017, Available in https://github.com/remkop/picocli
[25] Roman Leventov, “Chronicle-Map”, 2014, Available in https://github.com/OpenHFT/Chronicle-Map
[26] Stephen Colebourne, “joda-time” 2023, Available in https://github.com/JodaOrg/joda-time
[27] Steve Springett, “Dependency-Track”, 2018, Available in https://github.com/DependencyTrack/dependency-track
[28] Szczepan Faber, “Mockito”, 2009, Available in https://github.com/mockito/mockito
[29] Tobias, “JSQLParser”, 2023, Available in https://github.com/JSQLParser/JSqlParser
[30] Vasco Lavos, “Botania”, 2015, Available in https://github.com/VazkiiMods/Botania
[31] Vishal Nehra, “AmazeFileManager”, 2015, Available in https://github.com/TeamAmaze/AmazeFileManager
[32] Yuriy L, “SmartTubeNext”, 2020, Available in https://github.com/yuliskov/SmartTubeNext