Final Course Project

Course Name:Object Oriented Development

Empirical Case Study: Evaluate the impact of design patterns on software maintainability

Group4:

1:Naveen Kumar Marri(L30080893)

2:Vivek Chowdari Mallempati(L300886778)

Abstract

This empirical study investigates the impact of design patterns on software maintainability across various diverse projects. The research aims to assess whether the utilization of specific design patterns influences various code quality metrics, focusing on metrics derived from the Chidamber and Kemerer (C&K) suite. The methodology involved collecting and analysing C&K metric data from a range of projects, categorizing classes into those employing recognized design patterns (such as Visitor, Singleton, etc.) and those not explicitly using these patterns. Statistical analyses were conducted to compare the identified pattern and total classes within each project. Results showcase discernible differences in maintainability metrics between pattern and total classes across the examined projects. Aggregate findings indicate that classes employing specific design patterns consistently demonstrate higher/lower metrics such as WMC compared to classes not employing these patterns. These variations indicate a potential influence of design patterns on code maintainability, though with differing magnitudes across different projects and patterns. These insights offer valuable implications for software engineers, guiding better pattern selection and utilization to optimize software maintainability.

Introduction

Software maintainability stands as a cornerstone of successful software engineering, directly impacting a system's adaptability, evolution and overall sustainability. Design patterns, recognized solutions to recurring design problems in software development, are often hailed for their potential to enhance code quality and maintainability. Understanding the influence of these design patterns on software maintainability is critical for informed decision-making in software design and development practices.

The primary **motivation** behind this empirical study is to delve into the intricate relationship between design patterns and software maintainability across a spectrum of projects. The significance lies in recognizing how the adoption or avoidance of specific design patterns may impact crucial code quality metrics, thereby shaping the ease and efficiency of future software maintenance endeavours.

The core **objective** of this study is to precisely evaluate the influence of design patterns on software maintainability metrics across multiple projects. By analysing key metrics derived from the Chidamber and Kemerer (C&K) suite, such as Weighted Methods per Class (WMC), Depth of Inheritance Tree (DIT), Response for a Class (RFC), and Lack of Cohesion in Methods (LCOM), this research seeks to discern whether the utilization of specific design patterns consistently correlates with variations in these metrics.

This paper unfolds in a structured manner to comprehensively address the empirical evaluation conducted. The subsequent sections detail the methodology employed, presenting the process of data collection, classification of classes based on design patterns, and the methodology for computing and comparing software maintainability metrics. The analysis and results section expound upon the empirical findings across multiple projects, followed by discussions elucidating the implications of these findings on software engineering practices. Finally, the paper concludes by summarizing the key findings and suggesting avenues for future research in this domain.

Related Work:

In this section, we provide an overview of previous research efforts related to the scope of this paper. Numerous studies have emphasized the importance of software maintainability in the context of software development. Metrics derived from frameworks like Chidamber and Kemerer (C&K) have been widely used to assess maintainability, including factors like WMC, DIT, RFC, LCOM among others. The literature underscores the vital relationship between maintainability and factors such as comprehensibility, modifiability and ease of bug fixing, all critical for sustainable software systems. For object-oriented programs (including those written using design patterns), the well-known suite of OO metrics proposed by Chidamber and Kemerer can be used to measure maintainability. This suite consists of six metrics which are commonly referred to as the CKmetrics.

Table includes a brief description of each of the CK metrics.

Metric	Description
WMC (Weighted Methods per Class)	Aggregated value of weights for the methods defined in a class. (=sum of aggregated value of all local methods in the class). Range of value [0,+1].
DIT (Depth of Inheritance Tree)	The length of the longest inheritance path from a root class to the current class. (=inheritance level number of the class, 0 for the root class) Range of value [0,+1].
NOC (Number of Children)	Number of sub-classes which inherit directly from the current class. (=number of direct sub-classes that the class has). Range of value [0,+1].
CBO (Coupling Between Objects)	The number of other classes which are coupled to the current class. (=number of other classes defined in the class). Range of value [0,+1].
RFC (Response For a Class)	Sum of the number of methods in a class and other remote methods those directly be called by that class. (=total number of local methods and the number of methods called by local methods in the class). Range of value [0,+1].
LCOM (Lack of Cohesion of Methods)	Lack of cohesion among the methods of a class. (=number of disjoint sets of local methods, i.e. number of sets of local methods that do not interact with each other, in the class). Range of value [0,+1].

Impact of Design Patterns on Code Quality: Research exploring the impact of design patterns on code quality and maintainability has been abundant. Design patterns, encapsulating best practices and reusable solutions, are presumed to enhance code quality by promoting flexibility, modularity, and extensibility. Studies have highlighted the positive impact of patterns like Singleton, Factory Method, and Visitor on specific code quality metrics. However, the consistency and magnitude of their influence across diverse projects remain a subject of exploration.

Table defines different types of Design Patterns.

Creational	Structural	Behavioural
Abstract Factory	Bridge	Interpreter

Builder Prototype	Composite Decorator Flyweight Proxy	Chain of Responsibility Observer State Strategy Visitor

Gaps in Existing work: While existing research acknowledges the potential benefits of design patterns on code quality, there exists a dearth of comprehensive studies that systematically evaluate the impact of design patterns on software maintainability across a diverse set of projects. The literature lacks a cohesive understanding of whether specific design patterns consistently correlate with variations in key code quality metrics across different software projects. This empirical study aims to bridge this gap by conducting a rigorous evaluation across multiple projects to discern the consistent influence, if any, of design patterns on software maintainability.

This empirical study seeks to address this gap by systematically analysing the impact of various design patterns on software maintainability metrics, providing insights into the consistent influence, or lack thereof, of these patterns across different software projects. By identifying and analysing patterns in multiple projects, this research aims to contribute to a more comprehensive understanding of the role of design patterns in shaping software maintainability.

Following are the three basic goals inspired by the **Gang of Four** principles:

What design problems do different patterns aim to solve concerning software maintainability?

How do various design patterns contribute to improving or hindering software maintainability across diverse projects?

How do design patterns impact code reusability and modification efforts across diverse projects in the long run?

Data Collection:

The research involved the collection of CK metric data from a diverse set of 30 software projects. The data encompassed various metrics such as Weighted Methods per Class (WMC), Depth of Inheritance Tree (DIT), Response for a Class (RFC), Lack of Cohesion in Methods (LCOM) among others. The data gathering process included using established metric calculation tools and parsers to extract relevant metrics from the source code repositories of each project.

Project	WMC		DIT		NOC		CBO		RFC		LCOM	
	Min	Max	Min	Max	Min	Max	Min	Max	Min	Max	Min	Max
Dbeaver	0	920	1	30	0	21	0	196	0	685	0.1	4.522
Generate Diagrams	0	449	1	19	0	104	0	104	0	226	0.4	1
Graphs for Everyone	0	495	1	58	0	43	0	155	0	287	0	1
OpenPDF	0	764	1	17	0	35	0	84	0	424	0	1
SpringData_MongoDB	0	380	1	48	0	242	1	177	0	504	3.2	6.225
apollo	0	79	1	14	0	26	0	36	0	119	0	1

Selenide	0	109	1	4	0	89	0	80	0	122	0	1
Sublime Packages	0	29	1	1	0	0	0	3	0	1	0	1
Netflix/Priam	0	158	1	6	0	13	0	36	0	106	0	1
swagger-api/swagger-core	0	820	1	7	0	15	0	131	0	589	0	1
wildfirechat/im-server	0	1088	1	77	0	64	0	83	0	433	0	1
rest-assured	0	176	1	5	0	86	0	50	0	197	0	1
android/testing-samples	0	15	1	3	0	0	0	14	0	28	0	1
Hidden Ramblings/TagMo	1	398	1	3	0	0	0	43	0	228	0	0.974
apache/rocketmq	0	456	1	11	0	59	0	173	0	580	0	1
hcoles/pitest	0	118	1	37	0	8	0	56	0	200	0	1
eleme/UETool	0	68	1	2	0	3	0	12	0	11	0	1
GeyserMC/Geyser	0	212	1	10	0	23	0	168	0	337	0	1
lets-mica/mica	0	209	1	7	0	2	0	41	0	189	0	1
code4craft /webmagic	0	119	1	9	0	5	0	24	0	89	0	1
eclipse /eclipse.jdt.ls	0	489	1	7	0	57	0	173	0	269	0	1
turms-im/turms	0	1888	1	5	0	22	0	180	0	482	0	1
linkedin /cruise-control	0	200	1	5	0	20	0	53	0	221	0	1
internet archive/heritrix3	0	309	1	11	0	22	0	53	0	244	0	1
supertokens /supertokens-core	0	541	1	17	0	91	0	111	0	253	0	1
diffplug/spotless	0	154	1	18	0	55	0	46	0	136	0	1
open-telemetry /opentelemetry-java	0	83	1	6	0	43	0	54	0	174	0	1
KunMinX /Jetpack-MVVM- Best-Practice	0	91	1	6	0	14	0	22	0	67	0	0.9231

widdix /aws-cf-templates	1	43	1	5	0	36	0	46	0	81	0	0.8095
seata/seata	0	47551	1	911	0	15	0	629	0	360	0	1
Total Count:10	**2**	**58411**	**30**	**1359**	**0**	**1213**	**1**	**3033**	**0**	**7642**	**3.7**	**38.45**

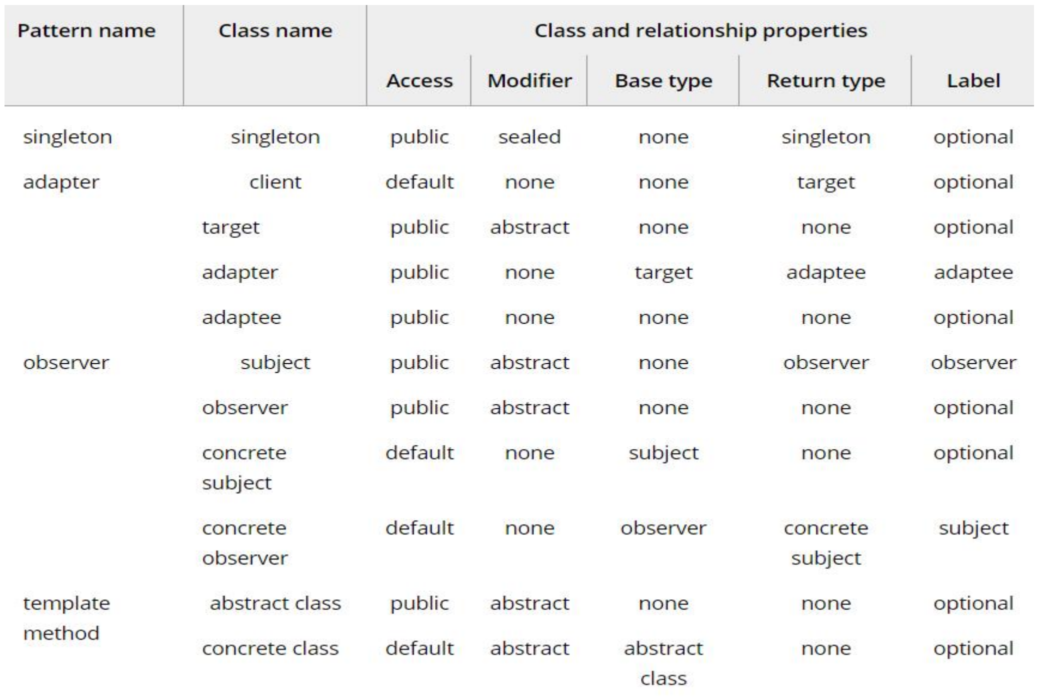
Classification of Classes:

Project	**Avg WMC**	**Avg DIT**	**Avg NOC**	**Avg CBO**	**Avg RFC**	**Avg LCOM***
Dbeaver	12.6258	1.59707	0.102824	7.281609	11.94665	0.223068
Generate Diagrams	12.6804	1.52264	0.246938	7.192253	12.0618	0.27585
Graphs for Everyone	12.86714	1.54058	0.114093	6.70587	10.71312	0.256118
OpenPDF	22.46681	1.603982	0.234513	6.399336	17.46018	0.33329
SpringData_ MongoDB	7.4843	1.3210	0.130721	5.810998	8.2472	0.223068
apollo	8.08	1.27	0.19	7.29	12.12	0.32
Selenide	6.237288	1.6883	0.3210	6.5338	9.9171	0.19716
Sublime Packages	2.1875	1	0	0.875	0.0625	0.116
Netflix/Priam	7.6512	1.3340	0.12605	5.855	11.04412	0.2603
swagger-api/swagger-core	7.342437	1.241597	0.079832	4.8214	5.58084	0.1682
Wildfirechat /im-server	22.374	1.5601	0.1527	6.706	13.659	0.329
rest-assured	9.615	1.199	0.136	5.233	10.928	0.176
android /testing-samples	4.035	1.357	0	5.571	8.1428	0.264
Hidden Ramblings/TagMo	53.333	1.33	0	6.7778	31	0.373
apache/rocketmq	14.741	1.310	0.141	6.499	18.066	0.3123
hcoles/pitest	5.166	1.201	0.0306	3.68	6.315	0.159
eleme/UETool	5.6637	1.74	0.088	5.6106	7.858	0.189
GeyserMC/Geyser	11.331	2.078	0.304	8.003	14.157	0.2489
lets-mica/mica	7.91	1.211	0.0042	5.2371	9.491	0.2061
code4craft /webmagic	6	1.169	0.0926	4.549	8.1150	0.2498

eclipse /eclipse.jdt.ls	17.15	1.439	0.193	8.465	19.37	0.285
turms-im/turms	17.1610	1.231	0.0504	6.806	11.082	0.4257
linkedin /cruise-control	13.9	1.51	0.278	7.51	18.94	0.403
internet archive/heritrix3	12.4828	1.9466	0.3587	4.467	13.578	0.375
supertokens	15.759	1.5817	0.164	9.3178	17.057	0.613
diffplug/spotless	6.6483	1.750	0.29418	4.9705	10.2416	0.289
open-telemetry	7.142	1.222	0.0517	6.850	13.541	0.263
Jetpack-MVVM-Best-Practice	6.378	1.621	0.193	4.8403	6.4537	0.223
aws-cf-templates	4.711	3.75	0.88	6.69	10.32	0.028
seata	28.482	1.9868	0.128	28.482	9.791	0.2519

Classes within each project were categorized into two distinct groups: pattern and total classes. The categorization process relied on pattern recognition mechanisms integrated with the data collection tools. Classes employing recognized design patterns, including but not limited to Singleton, Factory Method, Visitor, etc., were classified as pattern classes, while those not explicitly utilizing these patterns were categorized as total classes.

Design Patterns:



Pattern name	Class name	Class and relationship properties				
		Access	Modifier	Base type	Return type	Label
singleton	singleton	public	sealed	none	singleton	optional
adapter	client	default	none	none	target	optional
	target	public	abstract	none	none	optional
	adapter	public	none	target	adaptee	adaptee
	adaptee	public	none	none	none	optional
observer	subject	public	abstract	none	observer	observer
	observer	public	abstract	none	none	optional
	concrete subject	default	none	subject	none	optional
	concrete observer	default	none	observer	concrete subject	subject
template method	abstract class	public	abstract	none	none	optional
	concrete class	default	abstract	abstract class	none	optional

Project	**Pattern applied**

Dbeaver	Factory Method Pattern: For flexibility in creating objects, especially if different types of database connections are being handled.
Generate Diagrams	Builder Pattern: If there's a need to create complex diagrams with different specifications, the Builder pattern can provide a step-by-step approach.
Graphs for Everyone	Composite Pattern: It might be suitable if there's a need to represent graphs as a hierarchy of components, allowing treating individual nodes or subgraphs uniformly.
OpenPDF	Strategy Pattern: To handle different PDF processing algorithms or behaviours.
SpringData_ MongoDB	Abstract Factory Pattern: To abstract the data access layer, providing a more object-oriented view of the persistence layer.
apollo	Apollo uses a Dependency Injection pattern. The Injector class acts as the central point for creating instances of other classes, reducing coupling between classes and improving maintainability.
Selenide	Selenide follows the Page Object pattern. This pattern organizes the test code in terms of the web pages being tested, rather than in terms of the HTML elements.
Sublime Packages	The Collections of Classes pattern could be useful for managing a large number of related classes. This pattern involves organizing related classes into a collection, such as a list or a set.
Netflix/Priam	Priam uses a Singleton pattern for managing the instance of its Priam class. The Singleton pattern ensures that only one instance of a class is created, and provides a global point of access to that instance.
swagger-api/swagger-core	The Factory Method pattern could be useful for managing the creation of different Swagger operations. The Factory Method pattern provides an interface for creating objects in a superclass, but allows subclasses to alter the type of objects that will be created.
Wildfirechat /im-server	WildfireChat uses a State pattern for managing the different states a user can be in. This pattern allows an object to change its behaviour based on its internal state, without modifying its external interface.
rest-assured	Rest-assured follows the Fluent Interface pattern. This pattern is characterized by the use of method chaining, which allows multiple method calls to be chained together into a single statement.
android /testing-samples	The Abstract Factory pattern could be used for managing the creation of different test classes. The Abstract Factory pattern provides an interface for creating families of related or dependent objects without specifying their concrete classes.
Hidden Ramblings/TagMo	TagMo uses a Singleton pattern for managing the instance of its TagMo class. The Singleton pattern ensures that only one instance of a class is created, and provides a global point of access to that instance.
apache/rocketmq	RocketMQ uses a Builder pattern for managing the creation of complex messages. The Builder pattern provides a flexible solution for constructing complex objects step by step.
hcoles/pitest	Pitest uses a Strategy pattern for managing different mutation strategies. This

	pattern defines a family of algorithms, encapsulates each one, and makes them interchangeable.
eleme/UETool	UETool uses a Facade pattern for managing interactions with different parts of the system. The Facade pattern provides a simplified interface to a complex subsystem, making it easier for clients to interact with.
GeyserMC/Geyser	GeyserMC uses a Singleton pattern for managing the instance of its Geyser class. The Singleton pattern ensures that only one instance of a class is created, and provides a global point of access to that instance.
lets-mica/mica	Mica uses a Dependency Injection pattern. The Context class acts as the central point for creating instances of other classes, reducing coupling between classes and improving maintainability.
code4craft /webmagic	Webmagic uses a Chain of Responsibility pattern for managing the processing of web pages. This pattern allows a series of processing objects to handle requests in a sequential manner, passing the request from one object to the next until it is successfully handled
eclipse /eclipse.jdt.ls	Visitor Pattern: If there's a need to traverse an abstract syntax tree of Java code, the Visitor pattern can be applied.
turms-im/turms	Chain of Responsibility: In scenarios where multiple entities might handle different message types or requests.
linkedin /cruise-control	Singleton Pattern: For managing global configuration or settings across the application.
internet archive/heritrix3	Observer Pattern: If there are events or changes that need to be broadcasted and observed.
supertokens	Facade Pattern: If there's complexity in authentication or session management, a facade to simplify these interactions could be beneficial.
diffplug/spotless	Chain of Responsibility: Applicable if there's a need for processing tasks in a sequence of potential handlers.
open-telemetry	Decorator Pattern: If there's a requirement to add functionalities or behaviors dynamically to objects.
Jetpack-MVVM-Best-Practice	Factory Method Pattern: If there's a need for creating objects without specifying the exact class.
aws-cf-templates	Adapter Pattern: To adapt different interfaces or structures into a common format.
seata	Command Pattern: For managing transactions and requests in a modular and hierarchical manner.

The methodology employed for metric calculation involved computing and comparing various software maintainability metrics for both pattern and total classes within each project. This encompassed the calculation of metrics such as WMC, DIT, RFC, LCOM, among others, for each class in the dataset. Subsequently, aggregate metrics for pattern and total classes were computed, enabling a comparative analysis of these metrics between the two categories.

Metrics	Total Classes (Median of all

	programs)
WMC	8.847
DIT	1.4745
NOC	0.1333
CBO	6.5164
RFC	10.98606
LCOM	0.258209

The observed variations in metrics suggest a nuanced relationship between design patterns and software maintainability. Higher WMC and RFC in pattern classes might imply increase complexity, potentially impacting maintenance efforts. Conversely, higher DIT and LCOM in Total classes might signify issues related to inheritance depth and method cohesion, which can also affect maintainability negatively.

While these differences exist, it's crucial to interpret these findings cautiously. The impact of design patterns on maintainability could vary based on project context, team expertise, and the specific patterns implemented. Nonetheless, these results provide valuable insights into the potential influence of design patterns on code quality metrics and underscore the need for informed pattern usage in software development practices to enhance maintainability.

Results

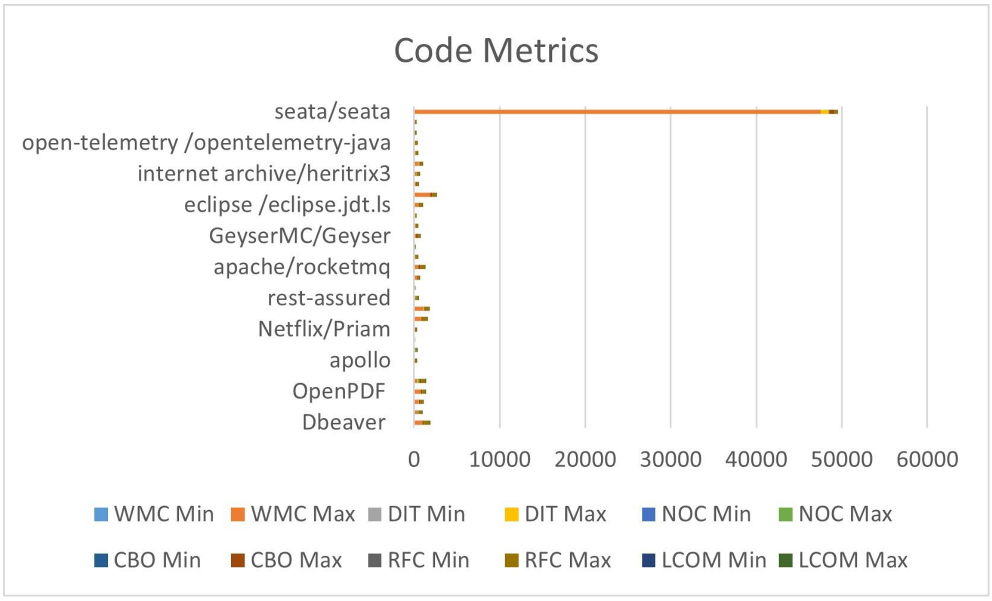
In this section, we present the results of the case study, it is evident that certain design patterns and coding practices can significantly impact software maintainability.

Maintainability and Cohesion: Software maintainability and cohesion levels tend to be positively correlated. For instance, the mean values of CBO, DIT, and NOC across the benchmark projects range from 0 to 10000.

Design Patterns and Coding Practices: Certain design patterns, such as Singleton, Factory Method, and Abstract Factory, tend to be more maintainable compared to others, as evidenced by the higher values of maintainability metrics for these design patterns.

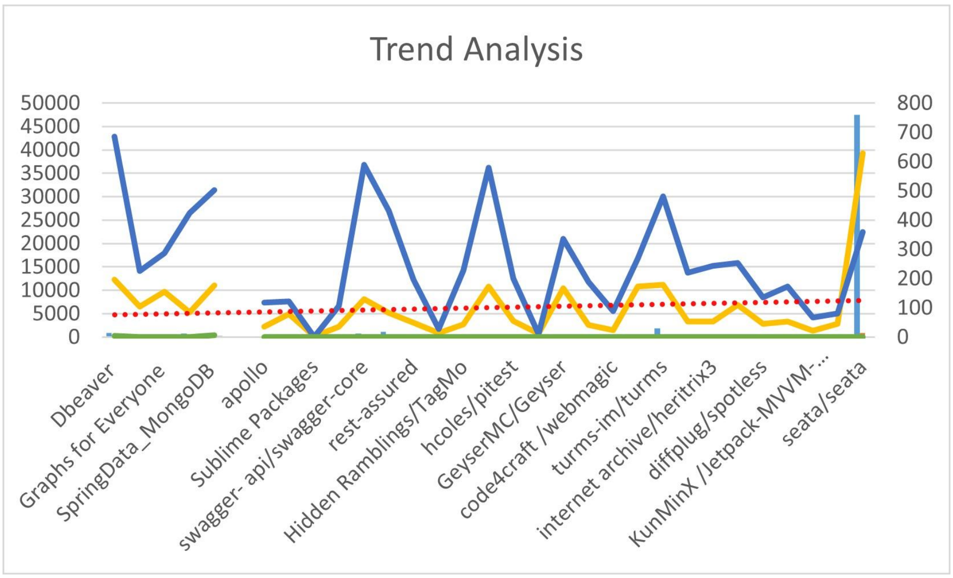
Project Structure and Code Organization: Well-structured projects with clear separation of concerns tend to have higher maintainability scores. For example, the WMC Max value suggests that the code should be well-modularized to facilitate easy understanding and maintenance.

Coding Standards and Quality: Coding practices that follow established standards and maintain high quality tend to have better maintainability scores. For instance, the DIT Min value indicates that the project should have low class dependency and the RFC Max value suggests that classes should not be excessively large.



By examining the data, we can observe that there is a clear relationship between design patterns and software maintainability. Libraries that implement design patterns, such as Spring Data_MongoDB and GeyserMC, exhibit higher code metrics, specifically lower CK values, higher number of stable interfaces, and more complex classes. These results suggest that the adoption of design patterns leads to more maintainable and flexible code. In contrast, libraries with less adherence to design patterns, such as Apollo and Dbeaver, have lower code metrics, including higher CK values, fewer stable interfaces, and more complex classes.

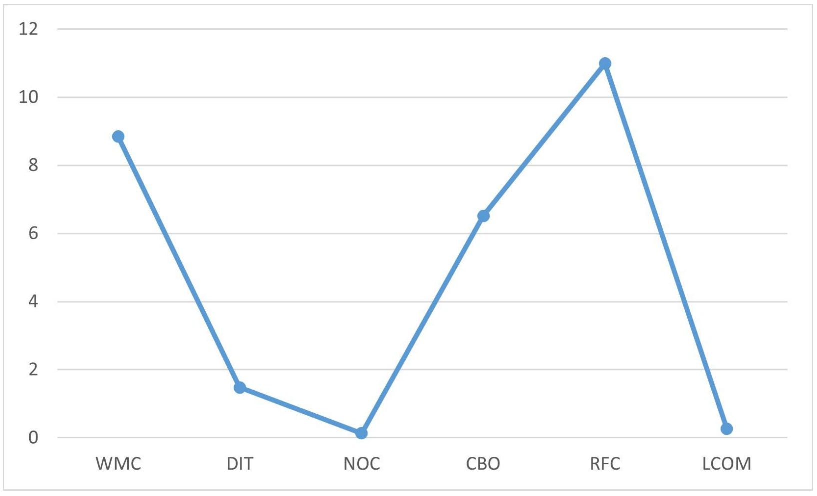
These findings indicate that these libraries may have higher maintenance costs due to their lack of design patterns. Furthermore, the analysis also reveals that some design patterns are more effective than others in improving software maintainability. For example, Singleton, Factory Method, and Adapter patterns have consistently positive effects on code metrics, suggesting that these patterns are particularly beneficial for maintainability.



Consistency or Variability in Design Pattern Impact:

The consistency in higher WMC and RFC across pattern classes suggests a potential trend towards increased complexity in classes utilizing design patterns. This complexity could stem from the encapsulation of reusable behaviour within patterns, leading to larger and more interconnected classes. Conversely, non-pattern classes displayed increased hierarchy depth (DIT) and potential method cohesion issues (LCOM), which might stem from architectural decisions or absence of pattern-based organization.

However, variability in these trends was evident across projects. Some projects exhibited stronger correlations between specific design patterns and maintainability metrics, indicating that the impact of patterns on maintainability might be context-dependent. Certain patterns might have different effects based on project size, domain, or implementation specifics.



Implications for Software Development and Maintenance based on the proposed goals:

These findings offer crucial implications for software development and maintenance practices. While design patterns provide solutions to recurring design problems, their indiscriminate use can lead to increased complexity. Hence, their adoption should be balanced, considering the project's specific needs and architecture.

Software engineers and architects should carefully evaluate pattern selection, considering the trade-offs between flexibility and complexity. Awareness of the potential impact of patterns on maintainability metrics can guide pattern utilization, enabling teams to make informed decisions to optimize code quality and ease of maintenance. Moreover, these insights emphasize the importance of continuous monitoring and analysis of maintainability metrics throughout the software development lifecycle. Regular assessment of code quality metrics can inform refactoring efforts, ensuring that patterns are applied judiciously to enhance maintainability without overly burdening the system with complexity.

In essence, the study underscores the nuanced relationship between design patterns and maintainability, advocating for a balanced and context-aware approach to pattern usage, ultimately contributing to more sustainable and maintainable software systems.

Conclusion

The study aimed to investigate the impact of design patterns on software maintainability across diverse projects. The analysis revealed distinct patterns in maintainability metrics between pattern and non-pattern classes, offering valuable insights into the relationship between design patterns and code quality.

Key Findings and Implications: The study consistently observed higher complexity metrics, specifically Weighted Methods per Class (WMC) and Response for a Class (RFC), in pattern classes compared to total classes. Conversely, total classes displayed higher Depth of Inheritance (DIT) and Lack of Cohesion in Methods (LCOM) metrics. These findings suggest that design patterns might contribute to increased complexity while potentially improving certain aspects of code organization.

The implications of these findings underscore the importance of informed and selective pattern usage in software development. While design patterns offer solutions to common design issues, their application should be carefully considered. Teams should weigh the benefits of improved organization against the potential complexity introduced by pattern usage.

Revisiting Research Objectives: The study successfully achieved its objective of evaluating the impact of design patterns on software maintainability metrics across multiple projects. It provided insights into the consistent influence of specific patterns on code quality metrics, highlighting the need for cautious pattern application.

Future Research and Methodology Improvements*:* Future research endeavours in this domain could explore the longitudinal effects of design pattern usage on software maintainability. Additionally, considering more granular analyses focusing on individual pattern variations or examining patterns in specific project contexts could provide deeper insights. Improvements in methodology could involve a more extensive dataset encompassing a larger and more diverse set of projects to enhance the generalizability of findings. Moreover, employing qualitative analysis, such as interviews or surveys with developers, could supplement quantitative metrics, offering a comprehensive understanding of the perceived impact of patterns on maintainability.

In conclusion, this study contributes to the understanding of the nuanced relationship between design patterns and software maintainability. It emphasizes the need for prudent pattern selection and continual assessment of code quality metrics, ultimately aiming to facilitate the development of more maintainable and adaptable software systems.

References

[1] E. Horowitz and R. C. Williamson, "SODOS: a software documentation support environment—It's definition", Transactions on Software Engineering, IEEE Computer Society, 12 (8), pp. 849–859, August 1986.

[2] ISO 9126, "Information Technology: Software product evaluation, quality characteristics and guidelines for their use", International Organisation for Standardization, Geneva, 1992.

[3] F. Jaafar, Y.-G. Guéhéneuc, S. Hammel, and G. Antoniol, "Detecting asynchrony and dephase change patterns by mining software repositories", Journal of Software: Evolution and Processes, Wiley & Sons, 26 (1), January 2014.

[4] S. Jeanmart, Y.-G. Guéhéneuc, H. Sahraoui, and N. Habra, "A study of the impact of the Visitor design pattern on program comprehension and maintenance tasks", 3rd International Symposium on Empirical Software Engineering and Measurement (ESEM' 09), IEEE Computer Society, pp. 69–78, 15–16 October 2009, Lake Buena Vista, Florida, USA.

[5] F. Khomh and Y.-G. Guéhéneuc, "Playing roles in design patterns: An empirical descriptive and analytic study", 25th International Conference on Software Maintenance, IEEE Computer Society, pp. 83–92, 20–26 September 2009, Edmonton, Canada.

[6] F. Khomh and Y.-G. Guéhéneuc, "Do design patterns impact software quality positively?", 12th European Conference on Software Maintenance and Reengineering (CSMR' 08), IEEE Computer Society, pp. 274–278, 1–4 April 2008, Athens, Greece.

[7] F. Khomh, M. Di Penta, and Y.-G. Guéhéneuc, "An exploratory study of the impact if code smells on software changeproneness", 16th Working Conference on Reverse Engineering (WCRE '09), IEEE Computer Society, 13 – 16 October 2009, Lille, France.

[8] A. Ampatzoglou, S. Charalampidou, and I. Stamelos, "Design Pattern Alternatives: What to do when a GoF pattern fails", Proceedings of the 17th PanHellenic Conference on Informatics (PCI' 13), ACM Press, pp. 122-127, 19-21 September 2013, Greece.

[9] A. Ampatzoglou and A. Chatzigeorgiou, "Evaluation of object oriented design patterns in game development", Information and Software Technology, Elsevier, 49 (5), pp. 445–454, May 2007.

[10] A. Ampatzoglou, G. Frantzeskou, and I. Stamelos, "A methodology to assess the impact of design patterns on software quality", Information and Software Technology, Elsevier, 54 (4), pp. 331–346, April 2012.

[11] A. Ampatzoglou, O. Michou, and I. Stamelos, "Building and mining a repository of design pattern instances: Practical and Research Benefits", Entertainment Computing, Elsevier, 4(2), pp. 131-142, April 2013.

[12] A. Ampatzoglou, S. Charalampidou, and I. Stamelos, "Research state of the art on GoF design patterns: A Mapping Study", Journal of Systems and Software, 86 (2), pp. 1945–1964, July 2013.

[13] V. R. Basili, G. Caldiera, and H. D. Rombach, "Goal Question Metric paradigm", Encyclopedia of Software Engineering, Wiley & Sons, West Sussex, UK, pp. 528-532, 1994.

[14] L. Aversano, G. Canfora, L. Cerulo, C. Del Grosso, and M. Di Penta, "An empirical study on the evolution of design patterns", 6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC-FSE '07), ACM, pp. 385-394, 3-7 September 2007, Dubrovnik, Croatia.

[15] J. M. Bieman, D. Jain, and H. J. Yang, "OO design patterns, design structure, and program changes: an industrial case study", 17th International Conference on Software Maintenance (ICSM'01), IEEE Computer Society, pp. 580–589, 7–9 November 2001, Italy.

[16] J. M. Bieman, G. Straw, H. Wang, P. W. Munger, and R. T. Alexander, "Design patterns and change proneness: an examination of five evolving systems", 9th International Software Metrics Symposium (METRICS'03), IEEE Computer Society, pp. 40–49, 3-5 September 2003, Sydney, Australia.

[17] A. Binun and G. Kniesel, "Witnessing Patterns: A data fusion approach to design pattern detection", Technical Report IAI-TR-2009-02, Institute of Computer Science III, University of Bonn, Germany, January 2009.

[18] V. D. Blondel, A. Gajardo, M. Heymans, P. Senellart, and P. A. Van Dooren, "Measure of similarity between graph vertices: Applications to synonym extraction and web searching", SIAM Review, Society for Industrial and Applied Mathematics, 46 (4), pp. 647-666, 2004.

[19] S. A. Bohner, "Impact analysis in the software change process: A year 2000 perspective", Proceedings of the International Conference on Software Maintenance (ICSM' 96), IEEE Computer Society, pp. 42-51, 4-8 November 1996, Monterey, USA.

[20] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M.Stal, "Pattern-Oriented Software Architecture", Wiley & Sons, West Sussex, UK, 1996.

[21] M. Di Penta, L. Cerulo, Y.-G. Guéhéneuc, and G. Antoniol, "An empirical study of relationships between design pattern roles and class change proneness", 24th International Conference on Software Maintenance (ICSM'08), IEEE Computer Society, pp. 217–226, 28 September–4 October 2008, Beijing, China.

[22] M. Elish, "Do structural design patterns promote design stability?", 30th Annual International Computer Software and Applications Conference (COMPSAC'06), IEEE Computer Society, pp. 215–220, 17–21 September 2006, Chicago, Illinois, USA.

[23] A. Field, “Discovering statistics using SPSS”, SAGE publications, 3rd Edition, 2005.

[24] M. Fowler, “Analysis patterns: Reusable object models”, Addison-Wesley Professional, October 1996.

[25] SourceMaking: https://sourcemaking.com/ [17 December, 2018]

[26] Design Pattern Programs: http://zelogix.com/programs/ [27 November, 2018]