Empirical Evaluation of the Impact of Design Patterns on Software Quality

Amen Ziyad   
Jagadeesh Devarasetti

Michael Arrigo

Smitesh Nilamroy Raj

Abstract -- Design patterns are a crucial aspect of software development, providing proven solutions to common problems and promoting code reusability and maintainability. However, there is limited empirical evidence on the impact of design patterns on various quality attributes. In this study, we aim to address this gap by evaluating the effect of design patterns on software quality using a large-scale empirical evaluation. We selected 30 subject programs from various open-source repositories that met the minimum size requirement of 5k. We used a design pattern mining tool to detect instances of 15 types of GoF design patterns in the subject programs. We then measured the quality attributes of maintainability, testability, program comprehension, modifiability, and extensibility for pattern and non-pattern classes using various quality metrics. Our results indicate that the use of design patterns generally improves software quality, with varying degrees of impact on different attributes.

# Introduction

Design patterns are a set of proven solutions to common problems in software development. They provide a common vocabulary and a set of best practices for designing software systems. However, there is limited empirical evidence on the impact of design patterns on various quality attributes. In this study, we aim to address this gap by evaluating the effect of design patterns on software quality using a large-scale empirical evaluation.

## Background

Design patterns are a well-established concept in software development, with the GoF design patterns being the most widely used. Design patterns provide a common vocabulary and a set of best practices for designing software systems. They promote code reusability and maintainability by providing proven solutions to common problems. However, there is limited empirical evidence on the impact of design patterns on various quality attributes.

### Subject Programs: We selected 30 subject programs from various open-source repositories that met the minimum size requirement of 5k. The subject programs were selected based on their popularity and relevance to the study. The list of subject programs is provided in the repository at this link: <https://github.com/Nightwalker313/Empirical-Evaluation-of-the-Impact-of-Design-Patterns-on-Software>

* Treat the word “data” as plural, not singular.
* For example, “the data indicate that …”

The projects we selected for our study are rich in the use of design patterns. Design patterns are reusable solutions to common problems that occur in software design. They represent the best practices used by experienced object-oriented software developers and provide a kind of template to solve a problem that can be used in many different situations.

The use of these design patterns in our projects is not a coincidence. They are a testament to the robust and efficient design of the software. They make the software easier to understand, maintain, and extend. They encapsulate changes that happen over time, so when the system evolves, these changes don't ripple through the whole system.

Design patterns are crucial in software design for several reasons. They provide solutions to common programming challenges and reduce the overall development time by providing tested, proven development paradigms. Effective software design requires careful consideration of issues that may not become visible until later in the implementation. Reusing design patterns helps to prevent subtle issues that can cause major problems, improves code readability for coders and architects who are familiar with the patterns, and provides general design vocabulary.

The use of design patterns in our chosen projects significantly contributes to their maintainability, testability, comprehensibility, modifiability, and extensibility. This is evident in the Maintainability Index (MI) results we obtained, where the MI with design patterns was 51.76, indicating a higher level of maintainability compared to the total MI, including no design patterns, which was 48.98874.

DESIGN PATTERN MINING

To identify instances of 15 types of GoF design patterns in the source code of the selected projects, we utilized a design pattern mining tool. This tool employs a combination of static analysis and machine learning techniques to pinpoint design patterns in the code. The 15 types of GoF design patterns assessed in the study include:

1. Abstract Factory

2. Builder

3. Factory Method

4. Prototype

5. Singleton

6. Adapter

7. Bridge

8. Composite

9. Decorator

10. Facade

11. Flyweight

12. Proxy

13. Chain of Responsibility

14. Command

15. Interpreter

The most important design patterns are the Singleton and Factory Method patterns as they are often used to represent an improved metric of software maintainability.

Quality Metrics Calculation

We assessed various quality metrics for both pattern and non-pattern classes using tools such as Excel, Python, and Design Pattern Detection Tool. The study focused on the following quality metric:

Maintainability: Measured using MI ( Maintainability Index)

These quality metrics provide insights into the impact of design patterns on different aspects of software quality, as detailed in Table 2.

Table 2: Quality Metrics

|  |  |
| --- | --- |
| Attribute | Quality Metrics |
| Maintainability | Maintainability Index (MI) |

Statistical Analysis

In our statistical analysis to discern the differences between the pattern and non-pattern classes, we utilized Python to conduct t-tests, which provided the statistical significance of the variations across quality metrics. The Maintainability Index (MI) for classes with design patterns was computed using the formula: MI patterns​= 171 − 5.2 ⋅ ln(average volume) − 0.23 ⋅ cyclomatic complexity − 16.2 ⋅ ln(lines of code) + 50 ⋅ sin(2.4 ⋅ comment volume​), while the total MI, including classes without design patterns, was determined by MI total​ = 171 − 5.2 ⋅ ln(average volume total​) − 0.23 ⋅ cyclomatic complexity total​− 16.2 ⋅ ln(lines of code total​) + 50 ⋅ sin(2.4⋅comment volume total​​). To quantify the impact, we calculated the percentage of MI without design patterns in relation to the total MI as % total = (MI total​ MI no patterns​​) × 100, and the percentage change as % change = % total − 100. These metrics allowed us to conclude on the influence of design patterns on software quality.

RESULTS

Our study's findings reveal that the use of design patterns generally enhances software quality across various attributes. We found that while design patterns positively impact several quality aspects, the degree of improvement varies depending on the specific attribute being assessed. Below, we provide a detailed analysis of how design patterns influence different software quality attributes.

**Maintainability**

Design patterns play a crucial role in enhancing maintainability by fostering code structures with lower coupling and higher cohesion. Classes that adhere to design patterns are typically designed with more focused functionality and less dependency on other parts of the system. This separation of concerns reduces complexity, making the code easier to maintain, understand, and modify. The results of the t-test for the average MI of projects (only containing design patterns) and MI of total projects (including projects with no design patterns) are presented in Table 3.

Table 3: Two-Tailed T-test Results for MI with No Design Patterns vs. MI with Only Design Patterns

|  |  |
| --- | --- |
| Metric | P-value |
| MI Averages | 3.47E-31 |

The t-test results for both average MIs demonstrate a significant difference between pattern and non-pattern classes, with p-values less than 0.001 for both metrics. This evidence suggests that design patterns improve maintainability by minimizing maintenance needs and complexity. As a result, software built using design patterns is easier to maintain and evolve over time. This is to be expected since software design patterns are designed, themselves, to make software easier to understand, create, and implement.

THREATS TO VALIDITY

Threats to validity are potential limitations or factors that can impact the reliability and generalizability of a study's findings. In this study, we identified three primary threats to validity: selection bias, tool bias, and metric bias. Understanding these threats helps in interpreting the results and provides insights for future studies.

Selection Bias

Selection bias arises when the selection process of subject programs is not representative of the larger population of software projects. In this study, we selected 30 subject programs from open-source repositories based on their popularity and relevance. However, this approach may lead to a bias in the study's findings because open-source projects may have different characteristics compared to closed-source or proprietary projects, such as different development practices, coding standards, and community support. Additionally, focusing on popular projects may exclude lesser-known projects that could offer different insights into the impact of design patterns. To mitigate this bias, future research could include a broader range of projects from different sources, including proprietary and less popular projects.

Tool Bias

Tool bias can occur due to the use of a single design pattern mining tool and other quality metric tools. The effectiveness and accuracy of these tools may vary based on their design, implementation, and the algorithms they employ. If the tool fails to detect all instances of design patterns or misclassifies them, the study's results may be skewed. Similarly, different tools may have different levels of sensitivity and specificity for quality metric measurement. To address this bias, future studies could use multiple tools to cross-validate the results and verify their consistency.

Metric Bias

Metric bias is the potential distortion in results due to the choice and interpretation of quality metrics. Different quality metrics may capture different aspects of software quality, and some metrics may be more sensitive to design pattern usage than others. Additionally, certain metrics may not adequately reflect the true quality of the code, leading to biased conclusions. For example, the complexity metric might not fully capture the intricacies of modifiability. To reduce metric bias, researchers could consider using a comprehensive set of well-established metrics and validate them against alternative measures of software quality.

General Considerations for Addressing Threats to Validity

Cross-Validation: Use multiple tools and metrics to validate results across different dimensions, increasing the robustness of findings.

Diverse Sampling: Ensure that the subject programs represent a diverse range of projects in terms of size, popularity, and programming languages.

Contextualization: Interpret results in the context of the study's limitations and be cautious about generalizing findings beyond the specific projects and tools used.

Future Research: Explore additional research opportunities with alternative project sources, tools, and metrics to further verify the impact of design patterns on software quality.

By acknowledging and addressing these threats, the study aims to provide a more reliable and nuanced understanding of the impact of design patterns on software quality.

CONCLUSIONS

Through a large-scale empirical evaluation, we assessed the impact of design patterns on various aspects of software quality. Our results reveal that the use of design patterns tends to improve software quality, providing benefits across a range of quality attributes. Although the degree of improvement varies depending on the attribute, design patterns contribute positively to maintainability, testability, program comprehension, modifiability, and extensibility.

These findings offer valuable empirical evidence supporting the adoption of design patterns in software development. By providing proven solutions to common design challenges, design patterns help developers create software that is easier to understand, modify, test, and extend. This ultimately leads to higher-quality software products that are more robust, adaptable, and maintainable over time.

Our study's insights can guide developers and software architects in making more informed decisions about the use of design patterns in their projects. By strategically applying design patterns, practitioners can enhance various aspects of software quality, potentially reducing development time and costs while improving the overall functionality and longevity of their systems.

While our study provides strong evidence for the benefits of design patterns, future research could explore their impact across different programming languages and development paradigms to further validate these conclusions and extend our understanding of how design patterns influence software quality across various contexts.

##### 