The Evolution of Software Design Patterns: An In-Depth Review

Mrs. Elavarasi Kesavan[1]

[1]*Full Stack QA Architect, Cognizant*

OPEN ACCESS

Volume: 4

Issue: 1

Month: February

Year: 2025

ISSN: 2583-7117

Published: 20.02.2025

Citation:

Mrs. Elavarasi Kesavan. The Evolution of Software Design Patterns: An In-Depth Review. International Journal of Innovations In Science Engineering And Management, vol. 4, no. 1, 2025, pp. 163–167.

DOI:

10.69968/ijisem.2025v4i1163-167

Abstract

Design patterns are repeatable fixes for common issues in software design. Even if it's helpful for software analysis, finding design patterns may be difficult, particularly in big and intricate software systems. A number of tools have been put out in this area to automate this procedure. Review the many studies on software design patterns in the literature in this topic. This review highlights the varied impact of design patterns on software quality and maintainability. While some studies suggest that design patterns enhance software quality, others argue they can be detrimental, with results varying based on factors like failure rates, performance, and maintainability. Design patterns such as Data Management UI Page and Dependent Dropdown Filters improve consistency, code reuse, and development efficiency. Among 42 design pattern detection (DPD) tools, only ten are available online, with low detection accuracy and weak agreement among tools. GEML, a novel approach using evolutionary machine learning, improves detection accuracy but may generate false positives due to limited training data.

Keyword: Software design patterns or Design pattern, Design pattern detection (DPD) tools, Software quality and maintainability, Creational Pattern

1. INTRODUCTION

The fundamental challenge in the developing area of software development is how to create systems that are quickly adjustable to changes, scalable, and maintainable. This data suggests that developers encounter increasingly significant issues with design, objects, behaviour, and structures as projects become more complex software systems [1]. In order to overcome these persistent issues, certain software design patterns are used. A framework for an effective solution to the specific design challenge is provided by these solutions. Design patterns, like the idioms stated before, are frameworks or principles that assist developers in making design decisions rather than prefabricated solutions [2]. The groundbreaking book Design Patterns: Several Design Patterns, written by the "Gang of Four" (Erich Gamma, Richard Helm, Ralph Johnson, and John M. Vlissides) and described in the book Elements of Reusable Object-Oriented Software, popularised the idea of design patterns in software engineering in 1994. They created twenty-three design patterns, which they divided into three primary groups: behavioural, structural, and creational [3]. These trends have developed into a collection of reference best practices that are an excellent resource for application architects and developers to always solve design issues without needing to look for a fresh approach [4].

Since software projects are becoming larger and more complex, using design patterns is essential to attaining the flexibility and modularity of these systems. One of the reasons is that patterns allow developers to abstract complex issues and create systems that are simpler to expand and maintain [5]. By ensuring that developers in development teams use the same terminology to describe system designs, design patterns assist to minimise misunderstandings. Design patterns are also useful for promoting awareness of OOD concepts, such as inheritance, polymorphism, encapsulation, and other principles like separation of concerns and the single responsibility principle [6].

It explicitly states that software is made up of parts that must be connected and interconnected to the point where debugging a basic program might take a very long time. Instead, it may be broken up into separate components that can be independently produced, tested, and changed [7].

Software design pattern

Software design patterns, also known as design patterns, are broad, repeatable solutions to problems that arise often in various software design settings. The structure of a design pattern is not inflexible enough to be incorporated straight into source code [8]. Instead, it is a description or a template for resolving a certain kind of issue that may be used in a wide range of circumstances. Design patterns may be thought of as codified best practices that programmers can use to address typical issues while creating a system or software application [9]. The links and interactions between classes or objects are usually shown using object-oriented design patterns, which do not define which final application classes or objects are involved. It is possible that functional programming languages are not the best fit for patterns that suggest mutable state [10]. Object-oriented patterns are not always appropriate for non-object-oriented languages, and certain patterns may be made superfluous in languages that have built-in support for resolving the issue they seek to tackle [11], [12].

Categories of design patterns

Design patterns may be divided into three primary categories: structural, behavioural, and creational. These categories are significant and address several issues pertaining to the program and its design, including build and communicate things as well as create items. In addition to improving code flexibility, reusability, and maintainability, they all provide solutions that may be used again when a particular design issue is resolved. To use design patterns in software systems, it is essential to understand these categories [4].

1. **Creational Pattern**

Because it conceals the creation process in a manner that renders a system neutral to the creation method, the creational pattern is connected to the issue of object creation. Additionally, these patterns are useful when creating objects requires complex initialisations and preparations, or when it is difficult to preestablish the precise sorts and relationships between objects. The latest results provide credence to the idea that creational patterns are still crucial in today's software development, particularly when it comes to frameworks and libraries that can need effective and expandable methods for producing objects. With the goal of guaranteeing that a single class has a single instance and providing a global point of access to it, the Singleton pattern is the most well-known creational design [13]. Although it is normal practice to create a new instance, Singleton might be used when a single copy of a class requires coordinating operations throughout the system, such administering a log service or database connection. Latent relations between the classes are introduced using this pattern, which makes testing and maintenance more difficult. These issues may be resolved by using Dependency Injection techniques in addition to Singletons to enhance testability.

2. **Structural design Patterns**

By grouping classes and objects into different forms and subsystems, structural design patterns increase the system's flexibility and streamline administration. These patterns are typical in applications that need to construct systems that connect several things or objects to one another without requiring complex control, exact connection, or demand. Because structural patterns are made up of several smaller components with more cohesiveness and less coupling, they make it possible to create enormous systems. In structural relationships, the adapter method is often used to allow objects with incompatible interfaces to communicate with one another [14]. This pattern is often used when developing an application when a new feature has to be added to an existing one without altering the application's code. For instance, an adapter may act as a mediator between two distinct systems in integration scenarios by converting one system's interface into one that is appropriate for the other system.

3. **Behavioral Patterns**

Behavioural patterns address how things behave when they are used and how roles are divided or shared across objects to address the issue dictating how these items interact. These patterns are helpful for allocating responsibility among objects in a system and managing how objects interact to carry out tasks.

LITERATURE REVIEW

(Alhunait & Khan, 2023) [15] It is difficult and complicated to design software, and it is much more difficult to maintain high standards and quality throughout the process. In order to address difficult software development and design challenges, several software development projects have used design patterns, which are repeatable solutions to certain frequent issues. The research study that follows looks at and investigates how design patterns affect the quality and maintainability of software. The research investigates if there are more correctly alternatives to design patterns or whether they are the most effective way to address typical software design issues. The study looks at what has been written about Design Patterns and how they affect the software development process.

(Patel, 2024) [4] provides yet another thorough analysis of software design patterns, going into their significance, the organisation of the categorisation, and their impact on software architecture and design. In addition to explaining how to utilise design patterns, the article goes into detail on real-world instances of design patterns as well as the challenges and disadvantages of doing so. The options for the future are further expanded, including the use of design principles for serverless, AI integration, cloud-new architectures, and Agile/DevOps. The article makes the case that design patterns are not immune to change and are constantly modified to integrate into the software development processes of the sophisticated and complex world of today. It also argues that in order for developers to be relevant when designing today's complex, large, and intelligent software-based systems, they must be familiar with both established and emerging design patterns.

(Khwaja & Alshayeb, 2016) [16] As a result, instead of capturing algorithms and data, others began working on design pattern languages to methodically record the abstraction described in the design pattern. While some design-pattern specification languages aim to find design patterns in code or design diagrams, others have other goals. Some have attempted to describe the design pattern in a textual or graphical environment. A comparison of these design-pattern specification languages and an analysis of their advantages and disadvantages have not yet been attempted, nevertheless. Using an assessment approach for design-pattern specification languages, this paper surveys and compares the current design-pattern specification languages. Design-pattern specification languages are categorised in order to do analysis. The tools available for the design-pattern specification languages are also briefly described. Lastly, we list a few unresolved open research questions.

(Moreira et al., 2022) [3] summary, the current detection technologies have limitations, such as the inability to compare the tools' outputs in terms of accuracy and agreement. By comparing design pattern recognition technologies and reviewing the literature, we fill up some of these gaps. Despite the large number of tools that have been released, the majority of design pattern identification technologies are unusable and useless. In particular, practitioners may find it difficult to locate a tool that meets their needs. It is necessary to find ways to either improve or combine the current techniques since they provide complimentary but erroneous detection findings.

(Wedyan & Abufakher, 2020) [7] The goal is to provide an explanation for these findings by taking into account quality-affecting implementation challenges, measurements, practices, and confounding variables. According to their findings, quality is clearly impacted by pattern documentation, pattern class size, and pattern dispersion degree. Researchers used several measures to distinct modules in case studies. The designs of controlled experiments varied significantly. In order to reach consensus on the impact of patterns, it is necessary to take into account influencing elements, apply uniform metrics, and agree on which modules to assess. It is advised that future study examine ways to enhance the modularity of patterns.

(Barbudo et al., 2021) [11] present GEML, a cutting-edge evolutionary machine learning-based detection technique that makes use of a variety of software attributes. To begin with, GEML uses an evolutionary algorithm to extract the features that best define the DP. These features are expressed in terms of rules that are accessible by humans and whose syntax complies with a context-free grammar. Second, to determine whether new code has a concealed DP implementation, a rule-based classifier is constructed. Five DPs from a publicly available repository that is often used in machine learning research have been used to verify GEML. In order to demonstrate its efficacy and resilience in terms of detecting capabilities, we then raise this number to 15 distinct DPs. A preliminary parameter analysis was used to fine-tune a parameter configuration whose effectiveness ensures the approach's broad applicability without requiring the adjustment of intricate parameters to a particular pattern. Lastly, a demonstration tool is also included.

(Naghdipour et al., 2021) [17] Based on software engineers' expertise, design patterns are a tried-and-true method for resolving persistent issues and are used to produce high-quality software designs. It is challenging to choose the best design pattern for a given design challenge, nevertheless, due to the abundance of them. To address this challenge, a number of strategies using various techniques have been put forward to automate the process of choosing design patterns. This study aims to provide a framework known as "DPSA" that comprises the categorisation of current methods, a comparison of methods according to specified standards, and an analysis of each method according to these standards. Future research benefits from DPSA in two ways: a) by using the present methodologies while considering their specifications, and b) by comparing the current and future studies.

(Yarahmadi & Hasheminejad, 2020) [18] The majority of systems lack a document that would aid engineers in identifying DPs from the codes. Consequently, many methods for identifying design patterns have been proposed. This study examines the various design pattern detection literature that is currently accessible and reports on a variety of topics, including data representation, design pattern type, benefits and drawbacks for various methodologies, quantitative findings, etc. In addition to providing a foundation for the selection of the best design patterns, the current inquiry report aims to direct future research by bringing attention to the possible flaws in earlier studies.

CONCLUSION

This review examines the impact of design patterns on software quality and maintainability, revealing mixed findings. While some studies suggest that design patterns improve software quality, others argue they may be detrimental. The effect varies depending on factors such as failure rates, performance, and maintainability. Certain design patterns, such as Data Management UI Page and Dependent Dropdown Filters, enhance consistency, code reuse, and development efficiency. However, the effectiveness of design patterns often depends on their combination with other approaches. Regarding design pattern detection (DPD) tools, 42 were identified, but only ten are accessible online. Among the Gang of Four design patterns, Composite and Observer are the most frequently detected. Current DPD tools exhibit low accuracy and weak detection agreement, with a high rate of false positives. GEML, a novel approach based on evolutionary machine learning, offers a more flexible and adaptable detection process through an extendable context-free grammar. By incorporating pruning methods and classification strategies, GEML demonstrates improved accuracy and robustness. However, limited training samples may impact its false positive rate. Future research should focus on refining detection tools and optimizing design pattern applications for enhanced software quality and maintainability.

REFERENCES

[1]. M. Z. Asghar, K. A. Alam, and S. Javed, “Software design patterns recommendation: A systematic literature review,” Proc. - 2019 Int. Conf. Front. Inf. Technol. FIT 2019, pp. 167–172, 2019, doi: 10.1109/FIT47737.2019.00040.

[2]. F. Al-Hawari, “Software design patterns for data management features in web-based information systems,” J. King Saud Univ. - Comput. Inf. Sci., vol. 34, no. 10, pp. 10028–10043, 2022, doi: 10.1016/j.jksuci.2022.10.003.

[3]. R. Moreira, E. FERNANDES, and E. FIGUEIREDO, “Review-based Comparison of Design Pattern Detection Tools,” SugarLoafPlop 2022 Lat. Am. Conf. Pattern Lang. Programs, Oct. 18, 2022, Online, vol. 1, no. 1, pp. 1–16, 2022.

[4]. H. Patel, “A research paper on software design patterns,” vol. 13, no. 01, pp. 803–813, 2024.

[5]. H. Zhang and J. Liu, “Research Review of Design Pattern Mining,” Proc. IEEE Int. Conf. Softw. Eng. Serv. Sci. ICSESS, vol. 2020-October, pp. 339–342, 2020, doi: 10.1109/ICSESS49938.2020.9237742.

[6]. L. Wang, T. Song, H. N. Song, and S. Zhang, “Research on Design Pattern Detection Method Based on UML Model with Extended Image Information and Deep Learning,” Appl. Sci., vol. 12, no. 17, 2022, doi: 10.3390/app12178718.

[7]. F. Wedyan and S. Abufakher, “Impact of design patterns on software quality: A systematic literature review,” IET Softw., vol. 14, no. 1, pp. 1–17, 2020, doi: 10.1049/iet-sen.2018.5446.

[8]. G. Luitel, M. Stephan, and D. Inclezan, “Model level design pattern instance detection using answer set programming,” Proc. - 8th Int. Work. Model. Softw. Eng. MiSE 2016, pp. 13–19, 2016, doi: 10.1145/2896982.2896991.

[9]. F. M. Alghamdi and M. R. J. Qureshi, “Impact of Design Patterns on Software Maintainability,” Int. J. Intell. Syst. Appl., vol. 6, no. 10, pp. 41–46, 2014, doi: 10.5815/ijisa.2014.10.06.

[10]. M. O. Onarcan and Y. Fu, “A Case Study on Design Patterns and Software Defects in Open Source Software,” J. Softw. Eng. Appl., vol. 11, no. 05, pp. 249–273, 2018, doi: 10.4236/jsea.2018.115016.

[11]. R. Barbudo, A. Ramírez, F. Servant, and J. R. Romero, “GEML: A grammar-based evolutionary machine learning approach for design-pattern detection,” J. Syst. Softw., vol. 175, 2021, doi: 10.1016/j.jss.2021.110919. Chordia et al., “Deceptive Design Patterns in Safety Technologies: A Case Study of the Citizen App,”

Conf. Hum. Factors Comput. Syst. - Proc., 2023, doi: 10.1145/3544548.3581258.

[12]. M. Kumar and M. Kumar, "Pattern Design and its Applicability in Software Design Mechanism," Res. Rev. Int. J. Multidiscip., vol. 3, no. 11, pp. 1153–1154, 2018, doi: 10.31305/rrijm.2018.v03.i11.244.

[13]. M. A. Jalil, N. A. A. Rahman, N. H. Ali, S. A. M. Noah, N. M. M. Noor, and F. Mohd, "Development of A Learning Model on Software Design Pattern Selection for Novice Developers," ACM Int. Conf. Proceeding Ser., pp. 108–113, 2020, doi: 10.1145/3383923.3383966.

[14]. S. A. B. A. Alhunait and M. S. Khan, "The Impact of Design Patterns On Software Quality and Maintainability," pp. 1–5, 2023, [Online]. Available: www.JSR.org

[15]. S. Khwaja and M. Alshayeb, "Survey on software design-pattern specification languages," ACM Comput. Surv., vol. 49, no. 1, 2016, doi: 10.1145/2926966. Naghdipour, S. M. Hossien Hasheminejad, and M. Reza Keyvanpour, "DPSA: A Brief Review for Design Pattern Selection Approaches," 26th Int. Comput. Conf. Comput. Soc. Iran, CSICC 2021, no. December, 2021, doi: 10.1109/CSICC52343.2021.9420629.

[16]. H. Yarahmadi and S. M. H. Hasheminejad, Design pattern detection approaches: a systematic review of the literature, vol. 53, no. 8. Springer Netherlands, 2020. doi: 10.1007/s10462-020-09834-5.