Using Design Principles to Develop More Maintainable and Flexible Object-Oriented Software

Mihir Patel
Computer Science
DePaul University
Chicago, IL, USA
mihir.patel130403@gmail.com

ABSTRACT

Software flexibility and maintainability are a crucial factor in software design and development as they reflect the ability to adapt to changes. It is a topic that has been discussed for a long time, which suggests its importance to software development. However, creating a flexible software design is not easy. To design scalable software, design principles provide fundamental concepts. On the other hand, design patterns provide proven solutions to recurring problems. When used correctly, design principles and patterns can be used to improve software flexibility and maintainability [1]. To improve maintainability and flexibility, this research proposes the use of object-oriented design principles such as SOLID and design patterns to address complexity, technical debt, and reusability challenges. This approach serves as a valuable concept for developing scalable and coherent software architectures by linking these principles to sustainable development in evolving systems.

KEYWORDS

Software Flexibility, Software Maintainability, Design Pattern, Design Principles, SOLID Principles, Complexity, Technical Debt, Reusability

1. Introduction

In modern software development, it is not adequate for code to simply work; Maintaining a well-structured codebase is crucial for long-term sustainability and collaboration. Code that lacks structure often results in duplicates, poor readability, and difficult debugging, making maintenance difficult for other developers. Both industry and academia recognize the need for software applications with robust object-oriented design structures that ensure code is more understandable, reusable, flexible, and maintainable. Traditionally, developers have used personal or organizational preferred styles when designing, which can lead to clarity and maintainability issues over time. In recent decades, object-oriented design approaches that favor low coupling and high cohesion to achieve design quality have become widespread [2]. By adopting proven design principles - such as the Single Responsibility Principle (SRP), the Open-Closed Principle, DRY (Don't Repeat Yourself) and YAGNI (You Aren't Gonna Need It) [4]. Design patterns Reliable and Reusable results produce solutions that enhance code readability and maintainability [5]. These principles and patterns offer practical guidelines for developers to avoid poor designs [4] and facilitate a shared understanding of the application's architecture, with a focus on future scalability and maintainability [3]. However, selecting the appropriate principles and patterns remains a challenge. Improper or forced use can create unnecessary complexity. This paper empirically examines the impact of implementing SOLID principles on software quality using a comprehensive case study. The case study included two different designs: one with SOLID-based improvements and one without applying SOLID principles. We saw quantifiable benefits by first identifying fundamental violations in the original design and then implementing corrective improvements. Comparing the two designs using Chidamber and Kemerer Java Metrics (CKJM) confirms the effectiveness of these design principles and highlights the increase in flexibility and maintainability.

*Mihir Patel, Department of Computer Science, DePaul University, Chicago, IL, USA, mpate252@depaul.edu

1.1 Overview of Design Principles

Design principles are fundamental guidelines that help developers and designers develop and maintain a software framework. Design principles help newcomers to software architecture avoid errors and pitfalls in object-oriented design.

1.1.1 S-Single Responsibility Principle (SRP)
According to the Single Responsibility Principle (SRP), each class can only change for one reason. Each class should have a single responsibility; A class with multiple responsibilities or the potential to add more responsibilities to the class is called high coupling. Unstable designs caused by high coupling can unpredictably destroy all improvement conditions.

1.1.2 O-Open Close Principle (OSP)
"Software Entities like classes, modules and functions should be open for extension, but closed for modification [7]." When a single change triggers a chain reaction of changes across related modules, it creates undesirable characteristics associated with poor design: the software becomes vulnerable, inflexible, unpredictable, and difficult to reuse. The open-closed principle

addresses this problem by promoting modules that remain stable. As requirements evolve, new functionality is added by introducing new code rather than modifying existing, working code.

1.1.3 L-Liskov Substitution Principle (LSP)
Liskov substitution principle related to the substitution property. Functions or modules that use references or pointers to super classes can be replaced by their objects of subclasses [6]. This should not break the application. This is related to the substitution property.

1.1.4 I-Interface Segregation Principle
"Clients Should Not Be Forced to Depend Upon Interfaces That They Do Not Use [7]." Changes to an unrelated interface can unintentionally impact client code and result in unintended coupling between all clients. The Interface Segregation Principle (ISP) recommends that clients should not interact with these interfaces as a single class. Instead, clients should rely on abstract base classes with coherent, focused interfaces.

1.1.5 D-Dependency Inversion Principle
"High Level Modules should not depend upon Low Level Modules. Both should depend upon Abstractions. Abstractions should not depend upon Details. However, details should depend upon abstractions [2]." To decouple high-level modules from low-level modules, an abstraction layer should be introduced between them. According to the dependency inversion principle, this abstraction layer must remain independent of the specific details of the problem.

2. Research Methods Used

2.1 Literature Review

The first step of this study was a review of the literature on software maintainability and object-oriented design principles, with an emphasis on important works such as Robert C. Refactoring by Martin Fowler, which highlights the importance of code restructuring for readability and adaptability. and Martin, who values robust, SOLID-focused architectures. Additionally, Arthur J. Riel's object-oriented design heuristics provided valuable insights to achieve high cohesion and low coupling necessary for flexibility. These resources support the case study's focus on adaptive, object-oriented architecture by providing a foundation for understanding how design principles and patterns improve the flexibility and maintainability of software.

2.2 Case Study Analysis

This case study examines how SOLID principles were applied to an e-commerce platform to solve problems maintaining and scaling a growing code base. Tight coupling and frequent errors were issues the team had to deal with as the system grew with features like multi-currency support and personalization. The team increased modularity, reduced errors, and improved flexibility by refactoring the code to conform to SOLID principles, which include separation of responsibilities (SRP), enabling extensibility (OCP), ensuring stable subclass substitution (LSP) and the development of focused interfaces (ISP). and Introducing Abstractions (DIP). As an example of how design principles can significantly improve the maintainability and scalability of software, this strategy made it easier to scale and adapt the platform to new requirements.

2.2 Comparative Analysis

To evaluate the effect of SOLID principles on maintainability and flexibility in object-oriented design, a comparative study between their application and non-application was conducted. The main topics of this analysis were the Liskov Substitution Principle (LSP), which improves component interchangeability, Dependency Inversion (DIP) and Interface Separation (ISP), which reduces module coupling, and the Single Responsibility Principle (SRP), which functions isolated to avoid bloated classes. When examining ERP modules, the study evaluated how adherence to SOLID principles enables scalable changes without creating interdependencies, while ignoring them often results in monolithic structures with tightly coupled modules, increasing the complexity and error-proneness of refactoring and updates increased significantly.

3. Case Study Analysis

3.1 Case A: Design Without Solid Principles

This study investigates how the software quality of an e-commerce system is affected when the SOLID principles are not followed. Using CKJM (Chidamber and Kemerer Java Metrics), the goal is to evaluate software quality and see how a non-SOLID approach affects maintainability, modularity and flexibility.

Project Description:
The e-commerce system in this study has standard functions such as product search, order processing, user registration and payment processing. The non-SOLID design methodology creates monolithic classes with tightly coupled dependencies that manage multiple tasks.

In this design:
Single Responsibility Principle (SRP):
According to the Single Responsibility Principle (SRP), each class should have a single responsibility to maintain manageable and understandable code. Nevertheless, numerous classes handle multiple tasks in this non-SOLID design. For example, order creation, payment processing, and inventory management are handled by the OrderProcessor class. This makes classes more complicated and difficult to maintain because changes to one function can unintentionally affect other functions in the same class.

Open/Closed Principle (OCP):
The Open/Closed Principle (OCP) states that classes should be closed to changes but open to extensions. This allows new features to be added without changing the code that already exists. Since

new functionality is often added by directly modifying existing classes, this principle is not adhered to in non-SOLID design. For example, changing the OrderProcessor class to add a new payment type risks making previously implemented functions unstable and increases the likelihood of errors in the system.

Liskov Substitution Principle (LSP):
According to the Liskov Substitution Principle (LSP), subclasses should be interchangeable with their base classes without affecting the correctness of the program. LSP is not considered in this non-SOLID design because subclasses introduce behaviors that are different from those of their base classes. Subclasses related to payments, for example, override base methods inconsistently, leading to errors or unexpected results when used in place of the parent class.

Interface Segregation Principle (ISP):
Smaller, more targeted interfaces are encouraged by the Interface Segregation Principle (ISP), which states that customers should only implement interfaces that are relevant to them. Because interfaces are large and contain methods that some classes do not need, this principle is ignored in this case. For example, classes are forced to implement unused methods because the payment interface contains methods that are irrelevant to certain payment types. This leads to redundant code and makes the code less readable.

Dependency Inversion Principle (DIP):
The Dependency Inversion Principle (DIP) suggests that high-level modules should be based on abstractions instead of low-level modules to increase the flexibility and adaptability of the system. Since high-level classes are based directly on concrete classes, DIP is not used in this design. For example, since OrderProcessor depends on specific implementations such as PayPalPayment and CreditCardPayment, adding or changing payment providers requires a change to the OrderProcessor code, resulting in a less flexible and adaptable design.

Methodology:

- **Non-SOLID implementation:** The e-commerce system was put into operation without adhering to SOLID guidelines, resulting in a design with tightly coupled components and monolithic classes.

- **Measuring quality with CKJM metrics:** The software quality of the non-SOLID design was evaluated using the following CKJM metrics: Coupling between objects (CBO): Evaluates dependencies between classes. Lack of Method Cohesion (LCOM): Indicates how closely related a class's methods are to each other. The degree of inheritance of the class structure is reflected in the Depth of Inheritance Tree (DIT). Based on the quantity and complexity of methods, Weighted Methods per Class (WMC) calculates the complexity of classes.

Results:
Tight coupling between classes resulted in high coupling between objects (CBO) for the non-SOLID design. Rigidity resulted from the difficulty of changing or replacing individual parts without affecting others due to high interdependence. Insufficient Method Cohesion (LCOM): LCOM was low because many classes had to multitask, resulting in lower coherence. For example, the OrderProcessor class had multiple responsibilities, which made maintenance difficult and complicated. The lack of structure of the inheritance hierarchy resulted in an inconsistent DIT (Depth of Inheritance Tree) metric. Unexpected behavior was often the result of subclasses having different responsibilities than their parent classes. The non-SOLID design had a high weighting of methods per class (WMC) because classes contained many methods to accomplish different tasks, making the design more complex and less readable.

Challenges Observed:

- **Difficulties in extending functionality:** Requiring changes to existing classes to add new functionality resulted in bugs and complicated testing.
- **Increase the likelihood of errors:** Due to tight coupling and lack of modularity, changes to a section of code often resulted in unexpected results.
- **Low maintainability:** The lack of single responsibility classes and the low coherence of the system made maintenance difficult. Understanding and modifying the code required extensive knowledge of the interdependent components.
- **Difficulty in scaling:** Scaling was difficult due to the rigidity of the codebase and lack of adherence to SOLID principles. Significant revisions were required to add new features such as user roles or more payment options.

3.2 Case B: Design With Solid Principles

This study investigates how the software quality of an e-commerce system is affected when the SOLID principles are followed. Using CKJM (Chidamber and Kemerer Java Metrics), the goal is to evaluate software quality and see how SOLID approach affects maintainability, modularity and flexibility.

Project Description: Functions such as product search, order processing, user registration and payment processing are provided by the e-commerce system. With oversized interfaces, tightly coupled dependencies, and large classes that handle numerous responsibilities, the non-SOLID design was monolithic. Each class has been improved in SOLID-based design to conform to SOLID principles.

In this design:

Single Responsibility Principle (SRP):
In this principle to improve maintainability and clarity, each class has been redesigned to cover only one responsibility. For example, different classes were used for order and payment processing, which simplified debugging and reduced class complexity.

Open/Closed Principle (OCP):
This principle allows new features to be added without changing existing code because classes should be closed to modification but open to extension. For example, developers could add new classes that extend the payment processing module to add new payment methods, reducing the likelihood of breaking pre-existing functionality.

Liskov Substitution Principle (LSP):
The principle conveys that classes and subclasses should be designed so that they can be used in place of their parent classes without causing unexpected behavior. When managing different payment processor types, this was particularly helpful in ensuring that each subclass (e.g. PayPalPayment or CreditCardPayment) can be used interchangeably without changing the basic payment logic.

Interface Segregation Principle (ISP):
Classes only implemented methods relevant to their interfaces, which were divided into smaller, more focused interfaces. This made it easier to extend the system and eliminated the need for redundant or empty methods by reducing unnecessary dependencies and ensuring that classes were not forced to implement unnecessary methods.

Dependency Inversion Principle (DIP):
High-level modules were more flexible because they relied on abstractions rather than specific implementations. For example, instead of relying on specific payment classes, the payment module used an abstraction of a payment interface. This made it easier to add or remove payment methods without affecting the basic logic of the payment processing module.

Methodology:

- **Non-SOLID Implementation:** Since SOLID principles were not applied when the system was first designed, it had monolithic classes with numerous responsibilities and a high level of dependencies.

- **SOLID refactoring:** The code was then rewritten to apply SOLID concepts, resulting in a modular and flexible architecture.

- **Assessing Quality Using CKJM Metrics:** The quality of both designs was assessed using the following CKJM metrics: Coupling Between Objects (CBO): Evaluates dependencies between classes. Method coherence within a class is assessed using the LCOM (Lack of Cohesion of Methods) metric. The inheritance structure is specified by the Depth of Inheritance Tree (DIT). Class complexity is reflected in the weighted methods per class (WMC), which is based on the quantity and complexity of the methods.

Results:
Strong interdependence was indicated by the inter-object coupling (CBO) of the non-SOLID design. Due to improved modularity and fewer dependencies, the CBO was converted to the SOLID-based design. The lack of method cohesion (LCOM) of the non-SOLID design was caused by classes that manage multiple tasks. LCOM improved after using SRP and ISP, demonstrating increased class cohesion and focus. In the SOLID-based design, the depth of the inheritance tree (DIT) remained moderate because LSP ensured that the inheritance structure was efficient and predictable. The SOLID-based design had a lower weighting of methods per class (WMC) because each class focused on a specific task, reducing complexity.

4. Conclusion

The significant influence of the implementation of the SOLID principles on software quality is shown by the comparative analysis of the e-commerce system. The result of the non-SOLID design was a tightly coupled, complex system that was difficult to scale and maintain. CKJM metrics showed low modularity and high complexity. On the other hand, the SOLID-based design showed quantifiable advantages in terms of flexibility, maintainability and modularity. The risk of errors in changes was reduced and scaling was made easier by the SOLID-based approach, which reduced class coupling and increased cohesion. The conclusion that adherence to the SOLID principles improves software quality - particularly in terms of maintainability and scalability during system expansions - is supported by this case study and highlights the importance of these principles for developing resilient and flexible software architectures.

Increasing complexity and the need for rapid adaptation have made design principles like SOLID more important than ever in the software industry. In order for modern software systems to adapt to changing user needs, business requirements and technologies, they must be scalable, resilient and agile. SOLID principles can be used to build a solid architecture that can change without interruption. These principles promote long-term maintainability and help teams manage complexity and reduce technical debt in the age of microservices, cloud-based apps, and continuous integration and delivery.

In addition, SOLID and other design principles improve system structure and make it easier for teams to collaborate. Code can be better understood, modified and extended by developers when there are clear modular boundaries and well-defined roles, ultimately increasing productivity and reducing errors. In a rapidly changing field, the use of design principles is critical to developing systems that can both withstand the demands of the future and produce reliable software today. This highlights the importance of the SOLID principles as a foundation for sustainable software engineering practices.

5. Future Scope

Future research could build on the results of this study and explore the use of SOLID principles in larger and more complex software environments such as distributed systems, cloud-native architectures, and microservices. In industries where scalability and maintainability are critical, such as healthcare, finance, and the Internet of Things, more empirical research could evaluate the impact of SOLID principles on software quality. SOLID principles can also provide a more thorough method of handling system complexity when combined with other cutting-edge design frameworks such as event-driven architectures or domain-driven design. To help developers put these principles into practice more effectively, machine learning techniques could also be explored to automatically identify and recommend SOLID-compliant code structures. To further improve code quality, reduce technical debt, and support resilient, adaptive software architectures that meet changing industry needs, future research could explore how SOLID principles can be integrated into cutting-edge technologies such as AI-driven code analysis tools.

REFERENCES

[1] **Rana, M. E., Murad, M., Atan, R. B., Khonica, E., Rahman, W. N. W. A., & Azmi, M. A. (2020).** Impact of design principles and patterns on software flexibility: An experimental evaluation using Flexible Point (FXP). *Journal of Software Engineering, 17*(7), 1-15. Retrieved from https://thescipub.com/pdf/jcssp.2021.624.638.pdf.

[2] **Prajna, R., & Kavitha, S. N. (Year).** Adoption of design principles and design patterns for developing software application. *RV College of Engineering, Bengaluru, 8*(5), India, 1-5. https://www.jetir.org/papers/JETIR2105707.pdf.

[3] **Haoyu, W., & Haili, Z. (2012).** Basic design principles in software engineering. In *2012 Fourth International Conference on Computational and Information Sciences* (pp. 1251-1254). IEEE. https://doi.org/10.1109/ICCIS.2012.91.

[4] **Braeuer, J. (2015).** *Measuring object-oriented design principles*. In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)* (pp. 882-885). IEEE. https://doi.org/10.1109/ASE.2015.17

[5] **Mu, H., & Jiang, S. (2011).** *Design patterns in software development*. In *2011 IEEE 2nd International Conference on Software Engineering and Service Science* (pp. 322-325). IEEE. https://doi.org/10.1109/ICSESS.2011.5982228

[6] **Singh, H., & Hassan, S. I. (2015).** Effect of SOLID design principles on quality of software: An empirical assessment. *International Journal of Scientific & Engineering Research, 6*(4), 1-4. https://www.ijser.org

[7] **Martin, R. C. (2000).** *Design principles and design patterns*. Object Mentor. Retrieved from http://www.objectmentor.com

[8] **Fowler, M. (1999).** *Refactoring: Improving the design of existing code*. Addison-Wesley Professional. Retrieved from https://martinfowler.com/books/refactoring.html

[9] **Martin, R. C. (2003).** *Agile software development: Principles, patterns, and practices*. Prentice Hall. Retrieved from https://www.pearson.com/store/p/agile-software-development/

[10] **Riel, A. J. (1996).** *Object-oriented design heuristics*. Addison-Wesley. Retrieved from https://www.pearson.com/store/p/object-oriented-design-heuristics/