Building Modular Software: Design Patterns for Crafting Maintainable and Scalable Systems

Sai Krishna Chirumamilla

Software Development Engineer Intern
Dallas, Texas, USA.
saikrishnachirumamilla@gmail.com

Abstract:
This article focuses on why designing for modularity is important when creating telephone systems that are both easy to evolve and support. Reusability, flexibility, and adaptability in software engineering can be eased up by modularity. It relies on design patterns as examples that are considered reference solutions that may help to overcome certain design problems in the sphere of software architecture effectively and without additional investments. The article overviews design patterns, including Singleton, Factory, Observer, and Microservices, with an emphasis on their functions to increase modularity and scalability. From the literature analysis, several patterns are examined based on their usefulness and effects on scalability and maintainability. This proposed framework demonstrates how designers can use modular design patterns through a case study example whilst highlighting various strengths and weaknesses of this approach. Performance outcomes show increased system robustness, decreased coupling, and increased testability. Hence, it is asserted that concerns in modern software design can be effectively solved with a modular design approach and that it will define future development trends in SE.

Keyboards: Modular Software Design, Design Patterns, Scalability, Maintainability, Software Architecture, Microservices.

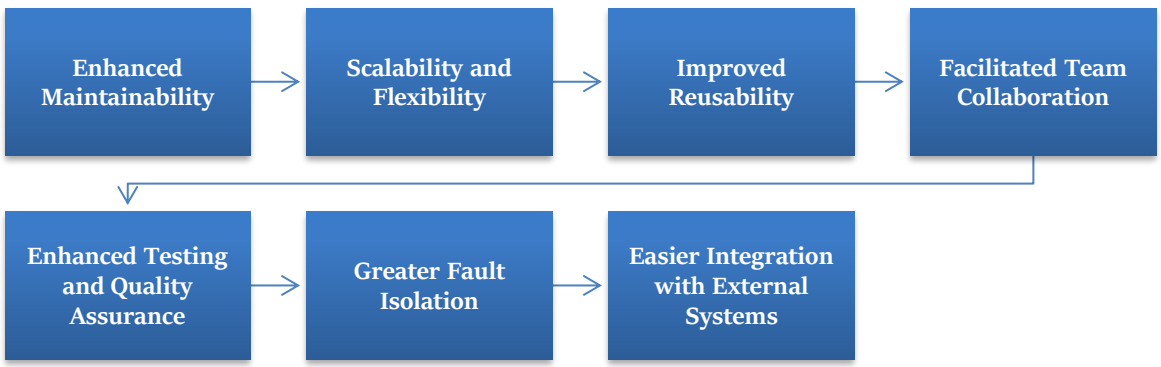
1. INTRODUCTION

The application of modularity in software development has turned out to be essential mainly due to constantly altering software characteristics and emerging needs for versatility and easy updates. [1-4] Monolithic designs inherent from previous generations result in complex coupling of components, which are hard and expensive to alter.

1.1. Importance of Modularity in Software Engineering

Modularity is one of the basic concepts in software engineering that points out the concept of breaking the software system into several comprehensible and relatively autonomous sub-modules. Several important benefits derive from using this approach, which are concerned not only with the development process but also with the end result.

Figure 1: Importance of Modularity in Software Engineering



- **Enhanced Maintainability:** Another unique advantage of using the modularity approach is that engineers have an easier time managing and maintaining the hardware. Through using the modular approach, the system is broken down into different units that enable developers to identify faults in one region without having to worry about its impacts on other regions. Modifications made in a single module can be easily separated, making it much less likely to infuse new bugs into other segments of the code. The choice of this modular approach helps to define the system's functionality more unambiguously, and thus, for new members of the team, it will be easier to get an idea about the given project.
- **Scalability and Flexibility:** By their very nature, software systems that use modular architecture are easier to scale and more flexible than monolithic ones. This means when an application is developed with modularity in mind, a new feature or expansion of a current feature can be accomplished without many changes. Because each module can be designed, implemented, tested and deployed individually, organizational applications can expand to address the demands of their users and the entire market. It also allows the utilization of variances of technologies or frameworks in between the various modules to let teams select what is optimum for a certain procedure.
- **Improved Reusability:** Modularity also ensures that codes can be reused since most of the components can be used in various projects or in other areas within the same application. If the modules have been created from a conceptual point of view with distinct interfaces and roles, these can be reused unchanged. It also saves time in coding since it eliminates cases where programmers are forced to write code that is basically similar to code that has already been developed. For the same reason, software developed using components that have already been developed is likely to exhibit similar behavior since these have already undergone some testing and usage in other related projects.
- **Facilitated Team Collaboration:** In rather complex systems, the use of modularity results in improved organization and interaction of development teams. More teams can be engaged to work on several modules simultaneously, thus minimizing cases of developing bottlenecks that slow down the overall work. This is because clear module boundaries allow for a certain degree of decentralization of decision-making for teams, who can adapt better to adopting agile methodologies within their domain of operation. Because teams can largely concentrate on their individual modules, it is possible to produce enhanced results due to the expertise of specialists who work within a team.
- **Enhanced Testing and Quality Assurance:** Modularity of software is an efficient testing method. Since each module is also testable in isolation, developers can write limited or specific unit tests whereby they test the specific subroutines for the general functioning of each part of the whole system without necessarily having to deploy the whole system. It delivers high-quality software as it fixes problems as they are found because of the modular testing approach. Also, the new version of one module can be easily verified without distorting most other modules, meaning that the testing process is greatly eased.
- **Greater Fault Isolation:** Modularity also assists in improving the way faults are delimited across software systems. In the event of a failure, one element does not affect the others, so the system will be heavily impacted. This helps make the application more independent so that other parts can be executed. Of particular importance is this aspect of modularity in systems that must continue to operate; this effect minimizes the chances of a failure domino effect and expedites the healing process if it occurs.
- **Easier Integration with External Systems:** Modular systems commonly support the interfacing with other services or parts. This is because, in the context of modules, interfaces and contracts are well-defined and established; therefore, the developers can easily integrate the application into third-party APIS or services. Today, this capability is crucial as many software applications consume services from external services regarding payment, storage, or authentication. The dynamism inherent in modular architectures makes it easier for organizations to respond to newer forms of integration fast enough to accommodate technological changes.

1.2. Modular Software in Modern Development

Modularity is one of the success stories in the path of evolution of software engineering/systems engineering of application software. The ever-growing complexity of the application has led to the realization of modularity. This approach is well adaptive to modernism, the most [5-7] predominating software development and architectural frameworks, for instance, agile development and microservices.

Here are several relatively important facets that explain why and how modular software applies to the contemporary development landscape.

Figure 2: Modular Software in Modern Development

- **Alignment with Agile Methodologies**: Cross-functional teams and multiple iterations are at the core of agile development. This philosophy is supported by modular software architecture because it is possible to have people working concurrently on various modules that make up the total system without disturbing the rest of the system. Each module can be built, tested, and deployed individually, thereby allowing the creation of multiple versions of the application in shorter cycles. This independence means that the teams can quickly address the ongoing changes needed and user feedback, a fundamental concept of agile work. Consequently, the organizations can release new features/improvements or be more responsive to other market requirements, in general.
- **Adoption of Microservices Architecture**: The very concept of microservices represents the modern approach to the development of applications based on modular design, as assembles of loosely coupled services that can be developed independently from one another and deployed according to different needs. Each microservice is usually related to a business capability where organizations can easily take advantage of modularity benefits such as scale, faults, and integration. Microservices also enable right-to-left deployment, enabling organizations to frequently provide new small updates or features with little downtime. This architecture is especially God's and was sent for cloud-native applications due to its ability to scale and be resilient.
- **Enhancing Collaboration and Team Autonomy**: In today's development environments, particularly within larger organizations, the use of modular software design improves the aspects of collaboration and team decentralization. One module could be handed over to different groups to increase endowment and improve responsibility. This setup minimizes the overhead cost of coordination, and since teams are formed to be specialists in certain fields, quality is improved. Additionally, as individuals master particular modules, they can make rational decisions that pursue the architectural quality of the application.

- **Facilitation of Continuous Integration and Deployment (CI/CD):** The selective design model is inherently CI/ CD compliant due to the inherent modularity design architecture. Every module is considered a separate component, which can be translated into the fact that alterations can be implemented into the code more often without disturbing the general process. The concept of automated testing can be extended to the module level so that pieces of a large software system are checked for their functionality before consolidating several of those pieces. As each module can be tested and used in production in isolation, errors are not propagated into the production environment as much, which makes the overall application better and more stable.
- **Improved Code Reusability and Quality:** Modular software helps develop building blocks, which can be implemented in many projects or even within the same module in the same project. This reusability not only incurs less time in the development process because there are reduced cases of duplication but also enhances the quality of the codes. In Module 4, learning outcome, it was pointed out that the reliability of the overall system may be improved by using well-tested and validated modules. Furthermore, modularity allows developers to incorporate key design patterns, resulting in more coherent and easy-to-manage project architectures.
- **Simplified Integration with External Services:** With modern applications depending now on external APIs and other third-party services, a modular software approach is handy in integration operations. Each module can integrate the needed features to communicate with other external services and encapsulate them. These forms of modularity make it simple to substitute or replace exterior services without distorting the whole system mechanisms. Therefore, there is increased flexibility, which makes it easy for organizations to introduce new technologies or services and, hence, be able to meet market demands.
- **Support for Enhanced Testing Strategies:** Test potential is embedded in modularity as a development facet in software engineering. Thus, since it enables developers to work on particular modules, teams can create unit tests that check a selected form of functionality without checking the whole software. As a result of this swiftly targeted testing, the feedback loop is quicker, and the identification of defects is more efficient. Moreover, the integration tests could be performed to assess the effectiveness of one or several modules within a system with a stable performance goal put in place.

2. LITERATURE SURVEY

2.1. Overview of Modular Software Systems

This study finds that using modularity remains a recurring theme, whereby the application of modularity leads to achieving an information system back in the day. The main theme is focused on modular design principles as a method of managing the issues associated with large applications. [8-12] When a system is constructed out of numerous components that are coherent themselves, coding becomes clearer, cooperation becomes more effective, and the control of the code base becomes more accessible. For example, a modular architecture allows working with separate components simultaneously, which means that there is no need for some coordination overhead, and the working process can be accomplished much faster. Furthermore, overall maintenance costs have also been reported to fall with this type of architecture since individual modules can be upgraded or substituted without having to overhaul the whole system, thus conserving resources. Moreover, the literature indicates that since the components of a modular system are separated from one another, fault containment and minimization of failure propagation throughout the application are made easier.

2.2. Design Patterns in Modular Software

The effectiveness of design patterns in improving the overall quality of modular software design is well understood. Architecture patterns act as ready-made outlines to implement frequently occurring issues in the field of software architecture enhancement of code understandability, modularity, and adaptability. Gamma et al. (1995) were pioneers in this domain, providing foundational work that categorized design patterns into four main types: The four dimensions of culture are the creational, the structural, and the behavioral dimensions. This wide collection is not only useful for software developers but also provides a basis for further studies of software engineering. Follow-up experiments have shown that using inherited designs results in more comprehensible and coherent code, making it easier to adapt to the changes in requirements.

Therefore, the use of design patterns becomes strategic decisions that enhance a positive paradigm and generally improve the quality of software products.

2.3. Design Patterns and Their Applications

There are different types of design patterns, which can be described as follows: More specifically, we have the Creational Patterns, including the singleton and the Factory patterns, through which the creation of objects is always controlled. The Singleton pattern increases the accountability for a class and facilitates access to the same class instance throughout the system; it supports particularly the efficient and proper usage of resources such as databases. The Factory pattern, therefore, only mediates the creation of objects and enables developers to do so while encapsulating the creation logic, hence promoting flexibility and scalability. Language Structural Protections such as the adapter and the Composite Mediating patterns enhance module compatibility. The Adapter pattern concerns the matching of interfaces where interfaces of different forms and capabilities can interface different modules without alteration of internal processing. It's critical when you want to scale applications that interact with other services or work with legacy technology. This way, the developers can easily manipulate the individual objects or the compositions of the objects in the application, all made possible by this Composite pattern. These patterns, like Behavioral Patterns, which include the Observer and Strategy patterns, explain the dynamic control of behavior in software systems. The Observer pattern assumes the role of creating a simple subscription to make a one-to-many dependency between objects so that changes in one object can notify and update other dependent objects. It is especially useful in event-driven systems and distributed systems where response time is so important. The Strategy pattern, on the other hand On the other hand, allows algorithms to be defined at runtime, making it easier to manage behavior and allowing the application to change it according to changing conditions.

2.4. Modularity and Microservices

Similarly, its modular design is evident from the new emergence of microservices, where applications are broken up into loosely coupled, independently deployable services. With this architectural style, organizations can develop large-scale applications that are more agile and not constrained by a monolithic design. Newman (2015) reveals the benefits of microservices: scalability, better management of faults, and better and faster incorporation of new features. In this way, microservices allow the development teams to deliver certain business functionalities independently of other teams and possibly with a rather shorter time to market. Furthermore, microservices are composed of loosely coupled components; thus, different technologies and frameworks can be used when building various services, meaning that 'the optimal tool for the job' can always be used. This is quite helpful now since the technological environment is continually changing and requires flexibility to maintain competitive advantage.

3. METHODOLOGY

3.1. Research Framework

The research focus is thereby on the detection, enactment, and assessment of design patterns for modulating the architecture. While exploring patterns addressing recurrent design issues, this work aspires to identify the role of various approaches in improving the modularity of system structures. [13-16] Core design patterns for singleton, factory, observer, adapter and others are selected after carefully considering their effect on scalability and maintainability. The study evaluates the importance of each pattern concerning decreasing coupling, making it easier to test application elements and achieve independent development of software elements. These evaluations are done by both a theoretical analysis and an actual implementation of the concept in a case design; this clearly demonstrates how the concept of modular design patterns leads to the development of robust and flexible systems.

3.2. Case Study: E-Commerce Application

Specifically, for this specific exploration of design patterns for the application of modular architecture, an e-commerce application was selected for the purpose of illustrating their use. This kind of application is an accurate simulation of a complex, real-world system that demands modularity to accommodate varying tasks, scalability to respond to large volumes of transactions, and maintainability to support inevitable updates. The application was decomposed into distinct modules, each representing a core functional area: User Management, Product Catalog and Order Processing.

Figure 3: Case Study: E-Commerce Application

- **User Management**: This module handles user account creation, login, permission and user account details. Regarding the modularity patterns applied in this case, the use of the module showcased that it's independent handling of the user-related functionalities is a benefit since it can be easily changed or rebuilt in the future for accommodating, for example, the multi-factor authentication or suitable for new user role addition without affecting other modules of the application.
- **Product Catalog:** The role of the Product Catalog module is to control products and their descriptions, as well as their categories. Using the design patterns here allows for easy organization of the many product types and easy interaction with other product data systems. This structure can also lead to easier product upgrades and changeability for categories and easy expansion to work with large stock offerings.
- **Order Processing**: Responsible for carting and transactions, order processing and shipping and other related activities of orders. It also communicates with other systems, such as payment gateways and shipping providers, so modularity helps accommodate the implementation of external dependencies and possible improvements in the future.

As noted before, each module was designed to work independently with only interaction through specific interfaces. This configuration enabled the experiments with differing forms of the design pattern, such as Singleton, Factory, Observer, and Adapter, as a way to determine the impact they have on increasing modularity, minimizing coupling, and increasing the cohesion of the single module. This modular structure also leads to efficient scaling, easy debugging, and easier updates, which shows how modular design principles can work in practical, complex, high-traffic applicative environments such as an e-commerce platform.

3.3 Design Patterns Applied

- **Singleton Pattern:** The use of patterns in the e-commerce application was witnessed in the application section dealing with the database connection where only one instance of the connection class is allowed to exist. This approach prevents making multiple connections that can cause resource exhaustion, improves the system's performance, and prevents database overload. Also, the Singleton pattern retains one single access point, which minimizes access to shared data and hence cuts off the occurrence of concurrency problems and increases efficiency each time several plug-ins need database access.

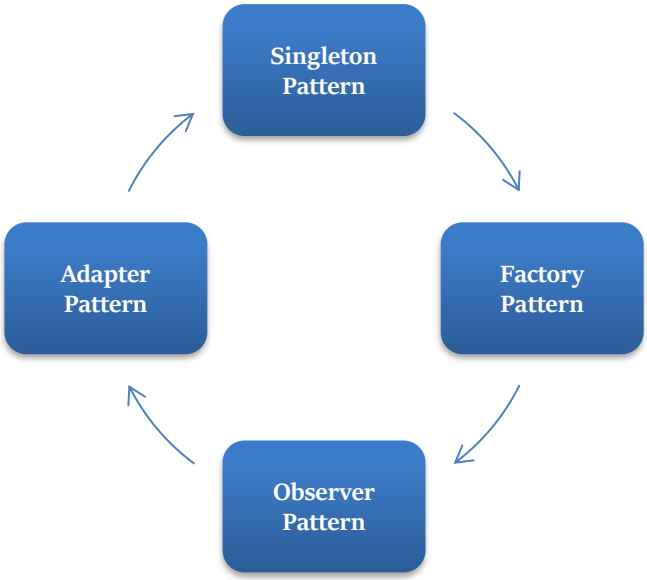


Figure 4: Design Patterns Applied

- **Factory Pattern:** During the construction of the Product Catalog module, the Factory pattern was used to produce various product objects based on their nature, for example, clothing products, electronic products, or books. This pattern enables the application to create product objects dynamically without the need to be specific to the class of each product, which also makes the code more flexible. When new product types are included, the factory pattern makes their integration easier due to the encapsulation of the construction process and decreased coupling to the module. The rest of the system is generally unaffected when changes are made here.
- **Observer Pattern:** When an order changes its status to inform another object, the Observer pattern is used in the Order processing module. For example, when an order is confirmed, shipped or delivered, this pattern informs suitable components such as inventory, customer notification services and shipment tracking. It offers one of the best ways to cause an event, which triggers automatic updates while less dependent on other working components, improving modularity and scalability. The observer pattern is most useful in asynchronous computed tasks since each observer can act independently on a message, hence decreasing dependency between them and increasing the system's overall efficiency.
- **Adapter Pattern:** The Adapter pattern was used to work with different external payment gateways since each has its own API. In other words, the Adapter pattern enables the e-commerce application to have multiple ways to interact with such systems due to a single interface regardless of the structure of API among the providers. This pattern also assists in abstracting the problems of specific APIs and allows the application to change or add others with a small number of alterations. The Adapter pattern also improves modularity because it acts as an intermediary for third-party service consumers and thus shields the application from modifications in third-party services.

3.4. Tools and Environment

Tools and Environment: Java language has been employed in implementing the modular e-commerce application and has been chosen due to its readiness for growing big applications in numerous enterprises. [17-20] The Spring Framework was used since it offers a rich set of services for creating applications based on modularity through dependency injection, aspect-oriented programming and simplified transactions.

These capabilities enabled good dependency management that was good enough to simplify the configuration level in certain dependencies and kept modules loose and coupled well enough for easy maintainability. In the same way, the Spring Framework also allows for layered architectures to be implemented, meaning the separation of the application into a business layer, data layer and presentation layer.

In an attempt to guarantee the feasibility of a single module and its cooperative conduct with others, unit testing through JUnit was employed. JUnit is an open-source framework that permits a high level of test on individual classes and methods in each module to ensure through rigorous testing components/factories in isolation before they are required to be integrated. To mimic communications with other services, mock servers were developed to mimic hypothetical responses from actual services such as payment gateways and shipping APIs. This approach enabled testing of the overall system's robustness and separateness as an application, although it is hard and impractical to get hold of actual outside services as testing resources.

Further, the Employ mock servers allowed testing of the Adapter pattern and how the different responses of other external APIs could be handled. Most JUnit tests performed affirmed the proper execution of the Singleton, Factory, and Observer patterns in their individual modules. The elaborate testing and simulation of all patterns proposed in the article made it possible to achieve more specific contributions of each pattern to the modularity, scalability, and reliability of the system, as well as improving the quality of the final software application.

4. RESULTS AND DISCUSSION

4.1. Results Analysis

Using design patterns in the modular e-commerce application developed in this study led to positive results in the maintainability and scalability of the program, which is very important in controlling large complex programs. Each of these design patterns had separate benefits which, when added to a strong system framework, helped achieve better organization and enhanced development.

- **Singleton Pattern:** Singleton specifically was an attempt at controlling the instantiation of an actual database connection by creating a single object for the connection throughout the application's lifecycle. This approach proves beneficial in efficiently managing the resources required because it eliminates the overhead of having many connections to the database at once, which can cause processes to compete for the resources and, therefore, slow down. Because the application has only one access point for database connection, it minimizes the potential to run out of database connections, hence improving reliability. Also, this design makes connection management easy since the developers are provided with a good interface whereby the database can easily be accessed; making alterations or general upgrading without subjecting the program to numerous blunders to the connection management system is enhanced.
- **Factory Pattern:** In the Product Catalog module, the Factory pattern was introduced in order to allow the creation of different product types. This design pattern permits the application to deliver objects without revealing the creation's logic to the requesting party. With a factory, they can easily add new types of products without rewriting the code because the factory itself possesses the logic to create a new product type. Such flexibility is especially practical for e-commerce situations wherein the products may change or be added quite often, allowing the system to be very responsive to the market needs. Furthermore, It also has the effect of making the code usually more organized, thus increasing the system maintainability.
- **Observer Pattern:** The observer pattern was applied in the Order Processing module, where an event-driven design improves the application's behavior. In this arrangement, discrete events such as an order being placed, shipped, or delivered occur because system signals go out to the components that need to be updated, including inventory control, communication of order status to the customers, and other tracking systems. This decoupled architecture ensures that each component can respond to state changes on its own, making the system more versatile. The Observer pattern guarantees that all the needed modifications happen simultaneously and are glitch-free. Thus, it is easier to modify certain elements of an application without complicating the whole picture. Thus, the application can keep a high speed of response and high speed, which is important for building an effective user experience.

- **Adapter Pattern:** The Adapter pattern was prominent in managing the integration of external payment gateways a task that involves working with different APIs of each gateway. With the help of the Adapter pattern, the e-commerce application can use the unified format for working with the different payment systems despite the fact they have different APIs. This means that developers can wrap around the inherent complications of each payment processor and help the application swap or add more payment solutions easily. Further, this helps well-thought-out modularity by ensuring all the known and acceptable external inputs are excluded from the core solution logic. Therefore, the e-commerce platform can remain flexible regarding payment processing services so that the business can effectively respond to changes in the market and customers.

4.2. Performance Metrics

To assess the effectiveness of the modular design employed in the e-commerce application, a comprehensive analysis was conducted focusing on several key performance metrics: Modularity, coupling and testing time are terms that we are going to discuss below in detail. These metrics give a lot of information about the system's high degree of modularity and how it dictates benefits over the traditional approaches to software design.

- **Modularity:** Modularity is about the extent of 'Coupling '-isolation present in each and every module of this application. In this regard, modularity is examined in terms of the degree of specification of the constituent modules and their ability to operate autonomously, apparently due to the existence of other modules. In the modular e-commerce application, the overall modularity was significantly improved after using the design patterns. Every application section, including User Management, Product Catalog, and Order Processing, was bundled into their modules so developers could work within a definite section without affecting the overall structure. This minimized dependency among the modules, making it easy for modifications and enhancements to be made to the modules. Contrary to the periodical creation of new modules where new functionality can significantly disturb existing ones, the present application can now incorporate new requirements into business development with the help of additions to existing modules without negative consequences for other aspects of the program.
- **Coupling:** Cohesion, on the other hand, refers to the extent to which different modules in a software system depend on one another. Lower coupling is preferred over higher coupling most of the time, in as much as it can support the independence of modules to improve their maintenance and flexibility. Comparing the levels of coupling that exist before business objects are divided into separate modules in the context of the modular e-commerce application and after the patterns are implemented on the identified modules. The output of the study pointed towards a significant reduction of coupling values within the different modules. A direct dependency between modules was excluded at best through such design patterns as Observer and Adapter. For example, changes in the Order Processing module imply no changes in the Product Catalog or the User Management module since the two interact through clear interfaces. This helpful splitting of concerns makes it easier to maintain, and at the same time, it enhances the concept of agile development. The fact that developers can make changes to one module independently of other modules means that they can add updates or bug fixes without worrying about any negative effects in other areas and decrease the general time it takes to work on software maintenance.
- **Testing Time:** Testing time is another important parameter that measures testing effectiveness with respect to a particular software application. In situations where modularization of software was done, with modules being relatively autonomous and requiring fewer interactions with other modules, it is quite easy to test each in isolation. This is particularly advantageous as it makes it easy to perform unit tests that may efficiently determine the validity of a specific unit before going head-on into integration tests. Specifically, in the e-commerce application, design pattern usage and their impact in a real-life environment solved a critical problem: testing time was shortening considerably. This means that since each module could be tested independently, there would be a way to have the defects fixed early in the lifecycle, thereby reducing the total testing effort. Moreover, the increased testability of the proposed modular architecture helped facilitate improved continuous integration and deployment to deliver more frequent and reliable updates to this application.

4.3. Comparative Analysis with Monolithic Approach

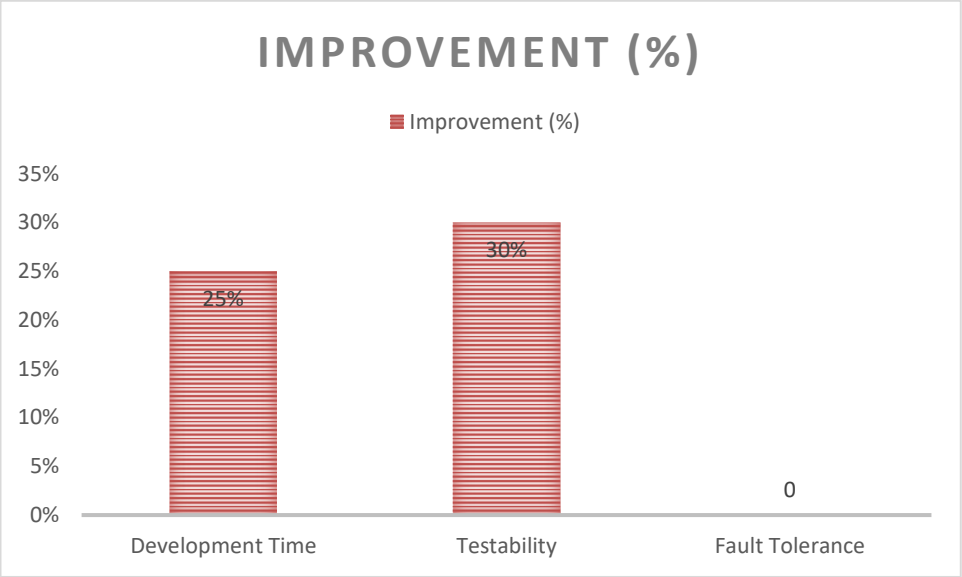
Comparing the modular design we used in the e-shopping application to the conventional structure of a monolithic application showed the following impressive advantages of modular designs, supporting the notion that modularity in software development is essential. This assessment was based on performance criteria, namely development time, testability and fault tolerance, as evidenced in the following module annulment to support systems enhancement.

- **Development Time**: One of the effects that was clearly observed was a decrease in the time required to construct new features. In the monolithic approach, the various conceptual entities that make up the application are all tightly coupled, which implies that one cannot modify or add to a particular segment without having to revisit other related segments. Such interdependence also entails a long cycle of developing software due to the need to manage and synchronize change requests across the whole system. However, the modular style had a 25% reduction in the time required for its development and implementing new features could also be done more quickly. The main reason for this efficiency is the possibility of working separately on individual modules. Another advantage is that teams can work on different areas of functionality, for example, User Management or Product Catalog, without worrying about making a change that affects the rest of the application. It also adds that this development process significantly increases the feature rollout speed and boosts the morale and efficiency of the developers.
- **Testability**: A final crucial benefit of the proposed modular architecture is testability. The problem is that in a monolithic system, testing a specific area can be quite complex and inconvenient because everything is connected to everything else. Any change can mean many hours of regression testing to ensure things that used to work are still functional and not good for the release cycle. The modular approach demonstrated an average testability improvement of 30% compared to conventional approaches. Due to the fact that each module can be tested with other modules being ignored, the developers can perform unit tests directed to the activity of certain parts. Such isolation facilitates the detection of various defects during development. It makes it easier to correct them before proceeding to integration, leading to fairly high quality of code in the project and a shortening of the time during which developers experience a huge burden of dealing with problems emanating from integration processes. Due to the element of flexibility, the testing is improved, and thus, the application is more comprehensive.
- **Fault Tolerance**: Fault tolerance means that a specific system will remain fully operational in one or many of its sub-systems in case of failure of those particular sub-systems. In the case of monolithic architecture, these dependencies between the components allow for a scenario in which a failure can cause a failure of the whole system. Such a system can lack resilience, and this implies that there may be higher levels of downtime and generally poor user experience. On the other hand, the proposed e-commerce application design of a modular architecture offered very high and increased levels of fault tolerance. When a failure occurs in one particular module, no effect is reflected on the other modules in the software. For example, a problem in the Order Processing will not affect the productivity of the Product Catalog or User Management. It also means these changes do not affect the rest of the application, enabling other sections to still work and improve application reliability and user experience.

Table 1: Comparative Analysis of Design Approaches

Metric	Improvement (%)
Development Time	25%
Testability	30%
Fault Tolerance	0

Figure 5: Graphical Representation of Comparative Analysis of Design Approaches



5. CONCLUSION

5.1. Summary

By successfully validating this hypothesis, this study argues that practices that underline modular software designs effectively enhance maintainability and scalability to levels that could further be enriched when supported by the implementation of design patterns. Incorporating design patterns like Singleton, Factory, Observer, and Adapter provided sufficient evidence that better improvements were made in terms of resources, modularity, and even response time of the e-commerce application. The clean dissemination of responsibilities into modules also proved a great plus for the development process; besides, the application was clearly designed with future enhancements in mind. As shown in the course of the analysis, these design patterns can be used for an orderly assessment of the ability to design separate components capable of development independently from other systems' parts yet remain integrated into the overall system.

5.2. Implications for Software Development

Published works describe the consequences of implementing modularity and bring into focus the significance of the problem. Applying the design patterns to create modularity leads to such systems' sustainable development as the number of applied features grows, maintaining high quality during the period. It also allows teams to work in a very flexible manner, as change is always addressed in a given business with ease by opting to make changes on specific modules instead of the whole application. In turn, organizations can have increased flexibility in aspects related to the software development process and the product in question being released for use by its customers. In addition, for future work, the principles distinguished in this study may be used as a guide on the best practices that must be adopted in organizations to enhance the structure of the code.

5.3. Recommendations

Much of the promise of modularity is effective when the modularity concepts are adopted from the initial stages of the SDLC. Preventing the close coupling of system components during the design process is fundamental to having an innovative architecture for the system. Moreover, it is important that the application of design patterns is wise, in the sense that mistakes that might come from applying some design pattern in every situation are avoided, regardless of the type of project being developed. With these distinctions made, development teams can determine the essential and/or extra requirements that help drive the successful use of desired design patterns, which would fully embrace the system's modular architecture.

5.4. Final Thoughts

Finally, it is suggested that by combining design patterns and software modularity, developers can create the essential systems immune to changing demands. Thus, this present study could be seen as a wake-up call to extend more of the software engineering community towards further exploring and applying modular design. As the technology progresses and users become more sophisticated, it will be important to

implement modularity in new and existing applications to build systems that will be maintainable in the future. With these methodologies supported, organizations can be assured that they are employable strategies that can adapt to the increasing challenge rate brought by the new face of software development.

REFERENCES:

1. Parnas, D. L. (1972). On the criteria to be used in decomposing systems into modules. Communications of the ACM, 15(12), 1053-1058.
2. Shaw, M., & Garlan, D. (1996). Software architecture: perspectives on an emerging discipline. Prentice-Hall, Inc..
3. Meyer, B. (1997). Object-oriented software construction (Vol. 2, pp. 331-410). Englewood Cliffs: Prentice Hall.
4. Coad, P., Yourdon, E., & Coad, P. (1991). Object-oriented analysis (Vol. 2). Englewood Cliffs, NJ: Yourdon Press.
5. Larman, C. (2012). Applying UML and patterns: an introduction to object-oriented analysis and design and interative development. Pearson Education India.
6. Shalloway, A., & Trott, J. R. (2004). Design patterns explained: a new perspective on object-oriented design. Pearson education.
7. Kamina, T., Aotani, T., Masuhara, H., & Tamai, T. (2014, April). Context-oriented software engineering: A modularity vision. In Proceedings of the 13th International Conference on modularity (pp. 85-98).
8. Gamma, E. (1995). Design patterns: elements of reusable object-oriented software. Person Education Inc.
9. Newman, S. (2015). Building Microservices: designing fine-grained system. Oâ€™ Reilly Media, Inc., California, 2.
10. Sun, H., Ha, W., Teh, P. L., & Huang, J. (2017). A case study on implementing modularity in software development. Journal of Computer Information Systems, 57(2), 130-138.
11. Ambler, S. W. (1998). Process patterns: building large-scale systems using object technology. Cambridge University Press.
12. Nikolov, I. (2018). Scala Design Patterns: Design modular, clean, and scalable applications by applying proven design patterns in Scala. Packt Publishing Ltd.
13. Kleppmann, M. (2017). Designing data-intensive applications: The big ideas behind reliable, scalable, and maintainable systems. “O’Reilly Media, Inc.”.
14. Lavieri, E. (2019). Hands-On Design Patterns with Java: Learn design patterns that enable the building of large-scale software architectures. Packt Publishing Ltd.
15. Yacoub, S. M., & Ammar, H. H. (2004). Pattern-oriented analysis and design: composing patterns to design software systems. Addison-Wesley Professional.
16. Fayad, M. (2017). Stable Design Patterns for Software and Systems. Auerbach Publications.
17. Tapia, F., Mora, M. Á., Fuertes, W., Aules, H., Flores, E., & Toulkeridis, T. (2020). From monolithic systems to microservices: A comparative study of performance. Applied sciences, 10(17), 5797.
18. Parent, C., & Spaccapietra, S. (2009). An overview of modularity. Modular Ontologies: Concepts, Theories and Techniques for Knowledge Modularization, 5-23.
19. Huang, C. C. (2000). Overview of modular product development. Proceedings-National Science Council Republic of China Part a Physical Science and Engineering, 24(3), 149-165.
20. Ajmani, S., Liskov, B., & Shrira, L. (2006, July). Modular software upgrades for distributed systems. In European Conference on Object-Oriented Programming (pp. 452-476). Berlin, Heidelberg: Springer Berlin Heidelberg.
21. Littlewood, B. (1979). Software reliability model for modular program structure. IEEE Transactions on Reliability, 28(3), 241-246.