World Journal of Advanced Engineering Technology and Sciences

eISSN: 2582-8266

Cross Ref DOI: 10.30574/wjaets

Journal homepage: https://wjaets.com/

(RESEARCH ARTICLE)

Check for updates

A research paper on software design patterns

Harshkumar Patel *

Software Engineer at Tesla, Management Information Systems, University of Houston, Clear Lake, 2700 Bay Area Blvd, Houston, TX 77058, USA.

World Journal of Advanced Engineering Technology and Sciences, 2024, 13(01), 803–813

Publication history: Received on 29 August 2024; revised on 05 October 2024; accepted on 08 October 2024

Article DOI: https://doi.org/10.30574/wjaets.2024.13.1.0499

Abstract

This article affords another exhaustive examination of software design patterns, discussing their importance, the classification system, and the influence on software architecture and design. It looks at how existing practices handle predictable problems, make things more easily serviceable, and increase developer collaboration. The article goes further and explains the use of design patterns and elaborates on real-life examples of design patterns and the difficulties and drawbacks of using them. Further, it expands on future possibilities, such as applying design patterns for cloud-new architectures, AI incorporation, serverless, and Agile/DevOps. From these trends, the article lays down the argument that design patterns are not insulated from change but are continually being adapted to fit into the software development processes of today's complex and sophisticated world and that to be relevant in the designing of today's complex, large and intelligent software-based systems, developers should be familiar with both the recognized and emergent design patterns.

Keywords: Software Design Patterns; Software Architecture; Cloud Computing; Microservices; Artificial Intelligence; Serverless Computing

1. Introduction

Taking software development as an emerging field, the big question is how to develop systems that are scalable, maintainable, and easily adaptable to changes. This evidence points to larger problems developers experience as projects become more intricate software systems: architecture, objects, behavior, and structures. To surmount these recurring problems, certain software design patterns are used. These solutions offer a frame for an efficient approach to the particular design problem. Like the previously mentioned idioms, design patterns are not ready solutions – rather guidelines or frameworks that help developers make design choices.

The concept of design patterns in software engineering was popularized in 1994 by the landmark book Design Patterns: Several Design Patterns explained in the book Elements of Reusable Object-Oriented Software by Erich Gamma, Richard Helm, Ralph Johnson, and John M. Vlissides, known collectively as the 'Gang of Four' (GoF). They developed 23 design patterns that they categorized into three main categories: Creational patterns, Structural patterns, and Behavioral patterns. These patterns have evolved into a set of reference best practices that are a great repository for application architects and developers to always solve problems of design without having to search for a solution in a new way.

While more and more software projects develop in greater size and growing complexity, applying design patterns plays a vital role in achieving such systems' flexibility and modularity. Some of the reasons include patterns that enable developers to abstract hard problems and design systems that are easier to maintain and extend. Design patterns help reduce misunderstanding by making people who work in development teams use the same terms that describe system designs.

* Corresponding author: Harshkumar Patel.

Design patterns are also effective in raising awareness of OOD concepts, including encapsulation, inheritance, polymorphism, and other principles like the single responsibility principle and separation of concerns. It makes a distinct provision that software comprises components that need to be interlinked to such an extent that it might take ages to debug a simple program. Rather, it can be subdivided into parts that can be developed, tested, and altered independently.

This article aims to present the major categories of software design patterns and then offer an aviation for them, their application in the real world, and their impact in enhancing software architecture. This article will present examples and case studies on how creational, structural, and behavioral patterns can address the most typical scenarios at the design step to increase software systems' expansibility, adaptability, and manageability. It will also help unveil the difficulties and drawbacks of applying design patterns. They provide a solution to common issues, and while using them, there is a high likelihood of creating over-engineering, whereby simple problems have very complex solutions. Also, some architectural patterns suitable for a particular programming paradigm may no longer be so when the paradigm has shifted, for example, to a functional one or where an environment is used.

2. Categories of software design patterns

Software design patterns are generally classified into three main categories: There are three main categories of design patterns known as Creational, Structural, and behavioral patterns. These categories are meaningful and solve different problems related to the software and its design, such as the creation of objects and the construction and communication of objects. They all detail solutions that can be reused when a specific design problem is met and increase code flexibility, reusability, and maintainability. Knowing these categories is crucial for using design patterns in software systems.

Creational Pattern is related to the problem of object creation, hiding the creation process in a way that makes a system neutral to the creation technique. These patterns are also valuable when it is impossible to preestablish objects' exact kinds and interrelations or when object construction presupposes intricate preparations and initializations. The recent findings support the View that creational patterns are still important in software development of the present day, especially in connection with the frameworks and libraries that may need efficient and extendable ways to create objects [6]. The best-known creational Pattern is the Singleton pattern, which aims to ensure that one class has only one instance and supplies a global point of access to it. While creating a new instance is common, Singleton can be used when a single copy of a class needs to coordinate actions across the system, such as managing the database connection or a log service. However, this Pattern is usually criticized for introducing latent relations between classes, which complicates testing and maintenance [16]. To overcome these problems, Dependency Injection methods can be used additionally to Singleton to improve testability.

The Factory Method pattern is another member of the pattern family of creational patterns. The factory method describes how an object can be created but does not fix the nature of the object at this step; the subclass can modify it. It is useful in frameworks where the type of object to be created depends on the particular environment. For example, in GUI toolkits, the Factory Method pattern may be applicable to make the platform-specific GUI component, so the implementations of the GUI toolkit available in the same code base should work on different platforms [7]. A pattern called Abstract Factory is used in the current term, which defines an interface for creating families of related or dependent objects without having to identify them with concrete classes. This pattern applies in cases where making several objects that should be interconnected at once is necessary. Some examples are GUI frameworks: windows, buttons, and scroll bars, which are all part of the system and must function in harmony.

The Builder pattern resolves this construction and representation issue by letting the same construction process build different representations. It is most beneficial where objects having many parameters that would commonly remain unused must be created or where the construction of an object is done in phases. Last but not least, the prototype pattern generates new objects by creating a new copy of an object; this is particularly useful when creating costly objects, especially in terms of time and space. This Pattern suits game development and graphical applications because many similar objects must be generated quickly [15].

Structural design Patterns organize classes and objects into various forms and subsystems to simplify the management process while improving the system's flexibility. Such patterns are common in applications that require building systems that relate several objects or entities to each other without establishing demand, precise linkage, and intricate control Embedded system. Structural patterns enable the creation of large systems because they comprise several smaller components with less coupling and more cohesion [12]. The technique often used in structural relationships is the Adapter method, which enables objects with incompatible interfaces to interact. This Pattern is usually used in the development of the application, where the new feature has to be inserted into the old one without changing the existing

code of the application. For instance, in integration situations where two separate systems are being integrated, an adapter can mediate between the two systems by translating the interface of one system into one suitable to the other system [4].

The next important structural Pattern is The Bridge pattern. It partitions the object into the interface and the body, where the interface only focuses on the specification of an object. As mentioned earlier, the bridge pattern may be slightly generalized and used in graphics libraries and cross-platform systems, where different implementations are necessary based on the platform or device. Thus, the Bridge pattern also allows a developer to modify one aspect of a system without changing another because the two are dissociated.

The Composite Pattern can be used when clients can work with individual objects and compositions of objects to the same extent. This Pattern is extensively used in file systems, graphic interfaces, and document systems where objects must be regarded as components of some whole. For instance, a GUI could have sub-components like buttons, text fields, and panels, but the fact that each element is rendered and managed at the same level means that the Composite Pattern groups them and presents them as one object.

The Decorator pattern is another structural pattern whose role is to add functionality to objects without changing the class of the objects. This Pattern is commonly used to add functionality to objects in runtime, and it is a better solution than subclassing. Today, in the web development process, for example, the application uses decorators to add functional characteristics, such as logging, authentication, or caching characteristics, to objects without modifying the objects' implementation.

The facade pattern makes a complex system more easily manageable and easier to interact with. It is mostly used in software libraries when internal implementation is complicated, but the consumers require a simple interface. APIs also use facades to hide the underlying details within any implementation.

Last, the Flyweight pattern minimizes the memory consumed by storing the same data for different objects. This Pattern is especially useful when many like objects are required for anything from graphical applications to data analysis systems. To reduce the memory consumption of a system, the Flyweight pattern says that the data is to be shared.

Behavioral Patterns deal with the behavior that objects exhibit when utilized and how responsibilities are shared or distributed between objects to solve the problem governing the interaction of such objects. These patterns are useful for controlling interaction between objects to execute work and apportioning liability between objects in a system [1].

The Observer pattern is one of the simplest behaviors and belongs to the most often used behavioral patterns. It establishes a one-to-many relationship between objects and sends requests to all the dependents whenever the state of an object changes. This defined Pattern is frequently used in highly synchronous applications such as event-driven architectures, likely because changes in one part of a system need to immediately translate to changes in other parts of the system, as seen in user interface frameworks [14]. For example, in the Model-View-Controller architecture, the Observer pattern helps the View to be updated each time the model alters.

The Command pattern captures a request as an object, so a client can parameterize objects with operations, execute them, or defer their execution. This Pattern is mostly implemented for undo/redo functionality and for systems with transactional characteristics, such as the operation systems of the databases [18].

The Strategy pattern identifies an algorithm group, encapsulates each group, and allows switching between them. This Pattern helps the algorithm change at a rate different from the clients who require its operation. It is commonly used in sorting algorithms, in which one of the sorting types could be implemented according to the context of the situation. The GoF calls this behavior the Chain of Responsibility pattern, through which a chain of handlers processes a request, and a handler might process the request or pass it to the next one. This Pattern can be used in applications where several request handlers, such as web servers or middleware applications, must handle requests.

Finally, the State pattern permits an object to behave differently when its state is transformed. This Pattern is common in game development to define a game character's behavior based on such states as idle, attack, or flee [20].

3. Application of design patterns in real-world scenarios

Design patterns are a project's gold mine to be accessed in software development, applying best-known solutions to emerging problems in designing, implementing, and maintaining sophisticated systems. They are used in many fields of

enterprise applications, Web applications, games, distributed systems, etc. Examining use case scenarios that incorporate design patterns assists in elucidating their potential usage in actual architectural circumstances and the purpose they serve in tackling certain architectural issues. In this part, we discuss a few of the widely known design patterns and show how they have been successfully adopted into current software projects.

The Factory Method pattern is also well used in modern web development frameworks, where different objects must be created within the appropriate context. Thus, we have a pattern that tells an algorithm for creating an object but leaves the determination of the type of object at runtime to the subclasses. In practice, a developer can integrate new forms of objects into an application without disturbing other forms of objects, enhancing the application's flexibility and scalability.

One good example is frameworks like Spring and Laravel, which adopt the factory method widely in instantiating the service objects or components [12]. For example, during the spring season, the spring framework is used in the dependency injection, where a factory method defines which beans have to be created and how they are connected. This is extremely helpful in complex applications used in large enterprises, where the concrete implementation of a service may depend on the context (e.g., development, production, or some client requirements). Delegation of object creation to the factory enables the developer to alter implementations or modify the tremendous behavior of the service without affecting the other components.

Similarly, the Factory Method manages Laravel, a well-known PHP web framework [3]. Through the application of the Factory Method in these circumstances, the application fosters the reusability of codes. Due to business logic and database structure modification, minimal adjustments will be needed.

The Singleton pattern is typically used in a system where only one object can control certain resources, such as database connection. The Singleton pattern ensures that only one class object is created, avoiding the expense of creating resource-demanding objects for high-powered applications.

Design patterns are a project's gold mine to be accessed in software development, applying best-known solutions to emerging problems in designing, implementing, and maintaining sophisticated systems. They are used in many fields of enterprise applications, Web applications, games, distributed systems, etc. Examining use case scenarios that incorporate design patterns assists in elucidating their potential usage in actual architectural circumstances and the purpose they serve in tackling certain architectural issues. In this part, we discuss a few of the widely known design patterns and show how they have been successfully adopted into current software projects.

The Factory Method pattern is also well used in modern web development frameworks, where different objects must be created within the appropriate context. Thus, we have a pattern that tells an algorithm for creating an object but leaves the determination of the type of object at runtime to the subclasses. In practice, a developer can integrate new forms of objects into an application without disturbing other forms of objects, enhancing the application's flexibility and scalability.

In the reactive model used by Angular and with the reactive programming notion, the Observer pattern is the basis for interacting components with services or a data stream. For instance, the RxJS library of Angular, a synchronous programming paradigm, extensively depends on the rightful use of the Observer pattern that enables components to subscribe to the update of data and respond to the change without having direct reference to the data source. This decoupling enhances maintainability and helps the system grow more efficiently, as different system components can respond to the same event as they occur.

The Strategy pattern is originally and mostly used in game development, where various algorithms or behaviors must be swapped at runtime. The strategy pattern shows what algorithms should be used, encapsulates these types, and allows for changes. It lets developers alter an object's behavior by substituting the algorithm or strategy that the object employs instead of modifying it. For example, the strategy pattern is applied rather actively to control characters' behavior, make decisions in AI, and calculate physics in games created with Unity3D and Unreal Engine [20]. For instance, a a game character's behavior in an action game may differ based on state – idle, attack, or escape state. In the Strategy pattern, these behaviors may be defined in a set of strategies so that characters may easily change their behavior during the game. This approach does more than just slimming down the number of files. It also makes adding or changing behaviors much easier without impacting the game mechanics.

The Decorator pattern has been extensively used in modern web applications to add behavior to objects without necessarily affecting their structure. This Pattern is especially useful when new behaviors can be reused for objects

during runtime, such as extending an existing web request and response with security, logging, and caching functionalities. In web frameworks like Node.js and Express, middleware functions can be easily seen as decorators for HTTP requests and responses. In this context, middleware functions encapsulate the fundamental request/response objects while providing additional services like verifying user credentials, validating input data, or writing logs without changing the server's core functions. It is also highly modular, making it very convenient to augment, decrease, or substitute middleware components to increase both versatility and sustainability. Also, WordPress and Drupal, which are content management systems, employ the Decorator pattern to extend the lifetime of core modules [2]. For instance, a WordPress Plugin may apply the Decorator pattern where one post or page might have additional attributes such as SEO, cache feature, or share buttons. What is more, by using decorators, these features can be applied to some types of content without impacting the rest of the system, which meets developers' demands, who seek the possibility of creating complex applications based on the given platform.

The command pattern is employed in several cases within the enterprise application domain to carry out requests more flexibly, such as objects. This Pattern frees the sender of a request from the object that performs it, introducing the possibilities of undo/redo, a queue, and transactions. The command pattern is used widely in enterprise resource planning (ERP) and customer relationship management (CRM) frameworks where users actively demand action invocation like record modification, orders, or reports. Since each operation is converted into a command object, it is easier to track, log, or even undo user actions, which adds more control to a system. For example, applying the Command pattern in a CRM system like Salesforce makes it possible to implement the audit trail. This command keeps track of the client's actions and can be undone or replayed in the future. In addition, the Command pattern is applied in distributed systems and microservice architecture, as the operations must be handled concurrently. These systems may use the ProvidedInvoker approach, where invocation is done at a later time with the help of the command system. Thus, the system retains responsiveness when working on long operations, which can take time [21]. This Pattern particularly applies to cloud-based applications where manageability and the ability to grow and recover from failure are significant.

4. Impact of design patterns on software architecture and development

Software design patterns have been instrumental in defining the current architecture and practices in software development. Design patterns also enhance code maintainability and flexibility since they offer developers usable solutions to compressing design issues and aid them in constructing scalable applications. Given the organization and sequence of steps, making the process more controlled and achievable is easier, thus laying a solid foundation for the software architecture. This section will show how different design patterns affect software's architectural design, maintainability, scalability, and the development phase.

The most significant effect of design patterns on software architecture] is constraint, or ensuring that architectural principles that improve system quality and modularity are followed. Said patterns help to build compliance with SOLID principles, which are essential for creating highly extensible and maintainable systems [7]. For example, Factory Method and Strategy are examples of the patterns supporting the Open/Closed Principle: software entities should be open for extension while closed for modifications. These patterns enable the developers to bring new functionality into a system without modifying existing code, which minimizes the chances of creating new errors while improving the software.

Responsibilities like Facade and Adapter help in practicing the basic principle of SoC, where a program is divided into different parts or each part solves a different problem. This modularity enhances the software design and flexibility by expanding push systems since new features or modules can be added without much affectation on the other parts [16]. For instance, where there are numerous subsystems in large enterprise systems, the subsystems may interact with one another through the Facade pattern, which can simplify complex subsystems and prevent numerous interdependent modules from arising. Originally, the Facade concealed the complex design of other systems that make up the structure, improving architectural simplicity and adaptability.

Furthermore, the Bridge pattern has the crucial function of uncoupling the abstraction from its implementation; one can modify the interface or switch the implementation at will. This Pattern is specifically useful in applications that work under several platforms where implementation is a different platform. This Bridge pattern guarantees that modifications to the implementation within the higher level of abstraction mostly benefit system reliability and malleability [4].

A major benefit of design patterns in software development projects is the improved maintainability of the developed code. The most important benefit of maintainable code is that it is easily extensible, which is very useful in large software development and multi-developer projects. Singleton, Decorator, Observer, etc., are often used to make the program

structure more modular and least invasive by inverting the concerns of the system and minimizing the coupling with the other components.

For example, there is a Singleton pattern, which many developers consider questionable because it produces hidden dependencies; however, when applied to certain application Alters, such as database connections or other hard-wired configuration variables, it can be tremendously useful. This way, Singleton de facto unifies the control, making it considerably easier to manage and modify some of the most critical system components as only one exists [10]. But in recent years, developers have not been liberal with Singleton as its usage puts the application in a tightly coupled zone, and unit testing has become more difficult. Therefore, with the lack of modifiability, modern software architecture tends to use Singleton with the help of dependency injection to achieve testability.

The Decorator pattern has been effective in dynamically adding dynamically adding behavior to the objects and retaining code flexibility instead of relying on subclasses. This Pattern is extensively used in user interface libraries. For example, appending scroll bars to a list control bu, idling borders around a button, or using shadows or bevels to press a button doesn't change the button's behavior. Decorator relieves the possibility of refactoring and extension by not making the inherited hierarchies go very deep.

The observer pattern, common in event-driven systems, increases the availability in some other way by providing the components to interact in a non-tightly coupled manner. Whenever a change is made to an element, all other components, depending on it or observers, get to know it immediately. These partitioning features make the system more composite and easier to control, for changes in one component do not entail changes in the other. Observer is most useful in modern systems with the GUI where the views' updates depend on users' actions [14].

One of the major concerns in software systems is its scalability, as we are targeting the building of large software applications for the cloud, enterprises, and complex distributed systems. Emerging design patterns have a social impact in helping scale systems with new paradigms for the scalable use of resources to accommodate growing loads. Out of these, Flyweight, proxy, and composite patterns are particularly significant in improving the system's efficiency and utilization of the system's resources.

The Flyweight pattern reduces the number of objects that can be generated, thus enabling new resources to be shared efficiently. This Pattern is most efficient in applications requiring many similar objects, as it is for game engines and large data visualization systems. By using the concept of shareable and immutable objects, Flyweight minimizes the number of objects generated in a system at any instance and, as such, optimizes the performance of a system: it makes it scalable.

Likewise, the proxy structural Pattern is used to handle system resources better by regulating the usage of objects. It is similar to lazy initialization in that the proxy can create the object or forward the interaction only when needed; this concept holds special importance in distributed systems where network latency and bandwidth are critical factors. For instance, Proxy design patterns in remote service architectures create mediation interfaces that handle the communication between the local and remote object to offset the cost and difficulty of this kind of call [8]. This results in the making of efficient systems that are capable of handling increasing workloads appropriately.

Scalability is another important area where the Composite Pattern contributes to system improvement. By making individual objects and groups of such objects indistinguishable, the Pattern allows for addressing the scalability issues more efficiently. This Pattern is widely used in organizations based on a hierarchical structure like file systems and interfaces, where separate objects and complex structures should be managed simultaneously. Due to the ability of Composite to break down objects in a system and group them in a tree fashion, new objects can easily be added or deleted as a system increases in complexity.

Design patterns, when used, can greatly enhance the software development process by presenting well-solved, agreed-upon design problems. Factory Method, Builder, and Command are complex to integrate with the design phase, but they provide highly flexible frameworks that can be adapted to various scenarios. It not only releases design and debug time but also ensures the implementation of future changes with little impact on the rest of the code [13].

Among the GoF patterns, it is noteworthy to discuss the practical application, for instance, the Command pattern, which helps to uncouple the sender and the receiver of the request and can parameterize and extend the actions. This Pattern is used to implement the undo and redo operation, operation queuing, or management of transactions in flexible systems to control the command operation. Thus, turning each request into an object minimizes the complexity of the operations and increases the system's modularity. Besides, the possibility of recording/Playback helps in the

automation of commands, thus resulting in improved predictability compared to the normal process of development/testing.

The Builder pattern also enhances the development cycle by severing the construction of intricate entities from its depiction. This Pattern is greatly useful in cases where objects have some settings or abilities in more than one form or where some parameters are just optional. Hence, the Builder makes the construction process efficient and cleaner, and it helps to minimize the creation of errors during the construction of an object [15]. Also, it increases the modularity of systems by providing an obvious and comprehensive form for creating test data or mock objects.

5. Challenges and limitations of using design patterns

Though design patterns are very effective in software development, bringing advantages such as maintainability, scalability, flexibility, and ease of implementation, they also possess several drawbacks and limitations. All these problems stem from operational issues that result from the misuse or application of patterns and other factors arising from the inherent characteristics of some of the patterns. Knowledge of such problems becomes necessary for developers to apply patterns and prevent mistakes properly.

The first issue with design patterns is that developers can overuse them – and those who are just approaching the use of patterns tend to do that most of the time. This is a common practice that people, especially developers, call the "pattern obsession" or "patterns," in which developers work extra hard to use patterns in cases that do not need them or where the patterns do not seem to suit the problem [8]. This means that if patterns are overused, they can compound the problem with design by making the design even bigger and more complex, a situation that will defeat the very aim of using patterns to improve design by making design and its management easier. For example, applying the Singleton pattern to every globally accessible object leads to tight coupling and decreased system flexibility; testing or extending the system becomes almost impossible. Similarly, the improper use of the Factory Method in simple object instantiation contexts where the Pattern is not necessary results in a more complex design with greater code opacity.

Some of these patterns bring extra layers of abstraction into the system, making it complex, especially when many patterns are used within the same design. This can complicate your code, making it difficult, especially for new developers or a new team that was not involved in the initial designing of the patterns before a project [14]. For instance, patterns such as decorator or chain of responsibility can cause the creation of complex object-oriented hierarchies in which it is difficult either to trace how exactly the execution is going on or to define how a specific element should behave. Within large systems, a developer can easily spend more time trying to understand the design than solving the problem.

The learning curve linked to design patterns forms another limitation to its adoption. Even when applying the patterns, some level of control is provided to the designer to understand that these patterns represent a proper succession of steps toward solving a frequent design problem and to know which cost is involved in applying each Pattern [15]. New developers with less experience will be in a position where they will not be able to determine the correct patterns for certain environments and may produce incorrect patterns.

It must be clear that design patterns are only sometimes useful across the board, and some may not be easily implementable within a given PARADIGM or project. For example, design patterns emerged from the OOP paradigm, but although they can be used in function- and procedural-oriented programming, this type of application is constrained or less efficient [16].

For example, the Observer or Command pattern is suitable for event-driven or GUI-based applications, but they offer less value in other, less intensive applications for event-driven interactions. Like the Factory pattern, the Abstract Factory pattern is also useful in a large system where the limits of an object's creation are unknown and must be flexible.

Also, some patterns provide specific solutions that may not be suitable for adaptation as the need changes. Onceging or restructuring the code becomes a rather complex task once this Pattern becomes inscribed into the architectural image. For instance, as soon as it is applied in an application, the Singleton pattern may only be easily replaced with a considerable restructuring of the application, and more so if it integrates closely with other system parts [10].

Certain design patterns lead to performance penalties and are associated with such design issues as extra layers of abstraction or object generation. For example, even though the Decorator pattern is quite valuable in adding behavior to objects, it often means extra function calls, which has disadvantages, especially for games and real-time systems [20].

Similarly, when used to regulate access to remote objects, the Proxy pattern could pose latency and network overhead that can compromise the high performance of a system.

6. Future trends in software design patterns

Thus, changes in software development over time, subject to new technologies and paradigm changes, change the roles of design patterns. Some of the existing patterns stay as valid as before. Still, emerging trends and issues connected with cloud computing, artificial intelligence, and microservices architecture define the tendencies in the further evolution of software design patterns. This section will discuss some emergent trends associated with the next generation of design patterns, pointing at the likely affective changes within the software development realm.

As the software capabilities shift to cloud and microservices, both design patterns evolve and adopt new ones to fit the distributed system landscape. When working in an environment where services are spread over multiple instances on several servers or continents, patterns focus on scalability, recoverability, and fault tolerance.

A pattern is the proliferation of patterns particular to microservices, including circuit breaker, Bulkhead, and service mesh [11]. These patterns assist in addressing issues of distributed communication, network failure, and the dependency that comes with service provision in cloud environments. For example, the Circuit Breaker pattern limits requests' redirection to a service that either fails or receives too many requests, thus protecting the entire system. The Bulkhead pattern helps avoid situations where one service lets through too many requests that overload other services by limiting resources and making service specific. These patterns are thus increasingly used for constructing strong, reliable, and scalable cloud-native applications.

Furthermore, the Service Mesh pattern emerging as a basis for topological underlay that insulates all communication delivered to/from microservices in the distributed context has cemented its role in improving observability, security, and overall traffic control architecture [17]. These patterns have been realized as cloud-native architectures keep expanding; they will be adjusted to cover more sophisticated scenarios like the multiple cloud or the half cloud.

With the growing use of artificial intelligence (AI) and machine learning (ML) in applications, adopting new design patterns that complement integrating such technologies into application systems is crucial. Basic design patterns are being applied and further enhanced to fit the requirements of AI/ML processes, including data preprocessing, model development, and real-time prediction.

An example is the MVC (Model-View-Controller) architecture development for embracing AI components in applications [21]. The simpler Model component now can contain machine learning models, which in turn must be trained and deployed in some cases, and the View must work with dynamic changes according to the real-time inference results. Furthermore, the Observer pattern is being implemented in Artificial Intelligence systems where a change in the value of some real-world data leads to re-training/ re-estimation of models to make their predictions more accurate concerning current data updates.

Another well-developed pattern is the pipeline pattern, which is widely used in data processing applications and AI workflows with greater frequency at the current phase. This Pattern divides the AI/ML process into fundamental parts that can be achieved in distinct stages, including data acquisition, data preprocessing, model choice, and model implementation [9]. This modular approach does benefit the AI system's maintainability and enables one to experiment with a different algorithm and a model. It does not affect the structural framework of the system.

Another significant trend that defines the development of software design patterns in the future is serverless & edge computing. Traditional solutions no longer fit for purpose in serverless, where a third party manages resources and infrastructure, so new architectures appear to satisfy new requirements for speed, cost, and scale.

Thus, in edge computing, where the computation is close to the data source, such patterns as Edge-Caching and Data Locality are becoming more prevalent [20]. These patterns prioritize the availability, speed, and efficiency as well as the processing and extraction of information near the network periphery for efficient and timely delivery of time-sensitive data such as IoT systems and big data real-time analytics.

With Agile and, more specifically, DevOps practices influencing software development paradigms, patterns for designing software structures are also being influenced by practices such as CI/CD. Developers are adopting features such as Feature Toggle and Blue-Green Deployment to release the features quickly and securely.[17] While the Feature Toggle

pattern lets developers toggle features on or off at runtime, used in AB testing and phased releases, Blue-Green Deployment ensures zero downtime through two live instances.

Furthermore, the practice known as Infrastructure as Code (IaC), which implies that infrastructure can also be described as a code, is becoming more popular in the DevOps working environments for the automation of creation and management of cloud infrastructures. In conjunction with the containerization patterns exemplified by the Immutable Infrastructure or the Sidecar, specific for the container orchestration platforms such as Kubernetes, the designated Pattern helps to cement the fact that scaling of software systems is relatively effective and that the action of maintaining those systems when placed in the highly dynamic environments, is readily easily [11].

7. Conclusion

The present paper's key emphasis was software design patterns and their relation to the future of software design based on present patterns. In this session, you heard the main problems that design patterns can help solve and how they facilitate communication between developers and make the code more maintainable and understandable in the structure of programs. One can find project-specific solutions when patterns are divided into creational, structural, and behavioral.

Design patterns must address their challenges with modern software development solutions transitioning to cloud, AI, and serverless models. The Microservice-specific patterns, AI-based workflow, and DevOps show how the design pattern changes and is part of more recent generations of design patterns. However, design patterns should be watched to prevent misuse and make systems hard to construct.

The overall architecture of new pattern designs will remain modular, scalable, and fault-tolerant as applications become more complex. These trends can be addressed and used to build complex and effective software systems that address the user's needs and emergent technological conditions. Lastly, the patterns will be relied on to expand in ostentation, capability, and flexibility to a stage where software architects are in a position to design sound, credible systems that can be scalable and robust and, therefore, one of the most precious resources that an engineer may have in the software engineering toolkit.

References

[1] Ahmed, R., Sultana, N., & Kumar, R. (2022). A survey on behavioral design patterns in event-driven systems. Journal of Software Engineering, 45(2), 123-134.

[2] Almeida, L., & Torres, V. (2021). Improving web server performance using the Chain of Responsibility pattern. International Journal of Web Engineering, 33(4), 67-79.

[3] Almeida, L., et al. (2021). Patterns for serverless architectures: Function as a service and beyond. Cloud Computing Journal, 15(3), 112-124.

[4] Alqahtani, A., & Khalid, M. (2021). Bridge pattern applications in cross-platform systems. Software Practice and Experience, 51(9), 1175-1190.

[5] Alqahtani, A., Abdul-Rahman, R., & Khalid, M. (2021). Adapting legacy systems: The role of the adapter pattern. Journal of Software Maintenance and Evolution, 33(1), 12-22.

[6] Bastarrica, M., & Azanza, M. (2020). The evolution of creational patterns in modern software frameworks. ACM Transactions on Software Engineering, 29(3), 234-252.

[7] Brown, S., Wang, L., & Lee, C. (2021). Factory Method pattern applications in GUI toolkits: A case study. Software Practice and Experience, 51(9), 1083-1101.

[8] Brown, S., Wang, L., & Lee, C. (2021). The risks of pattern obsession: Avoiding overengineering in software design. Journal of Software Architecture, 33(3), 65-83.

[9] Cheng, H., Wang, X., & Lee, Y. (2023). Modular AI pipelines: Patterns for scalable machine learning integration. Journal of Applied AI Research, 48(2), 98-115.

[10] Cheng, T., & Yang, L. (2020). Reducing coupling with Singleton and dependency injection: Best practices. Journal of Object-Oriented Programming, 39(4), 123-140.

[11] Fowler, M., & Lewis, J. (2022). Microservices patterns for cloud-native architectures. Software Engineering Today, 37(4), 45-62.

[12] Ganguly, A., Hussain, T., & Ray, A. (2021). Command pattern and its applications in workflow management systems. Journal of Software Development, 36(5), 143-162.

[13] Ganguly, A., Hussain, T., & Ray, A. (2021). Structuring scalable systems using structural design patterns. Software Architecture Journal, 28(2), 97-113.

[14] García, S., et al. (2021). Observer pattern in event-driven user interfaces: A review. ACM Transactions on Software Engineering, 44(2), 187-203.

[15] Hu, X., Wang, P., & Zhao, Z. (2022). Improving object creation with Builder patterns in large-scale systems. IEEE Software Engineering Journal, 48(1), 54-62.

[16] Iqbal, N., Rahman, S., & Khalid, M. (2020). Adapting object-oriented design patterns in functional programming: A case study. Software Practice and Experience, 50(9), 1020-1040.

[17] Rahman, F., Kumar, N., & Khan, I. (2022). Singleton design pattern: Use and misuse in large-scale applications. International Journal of Software Engineering, 47(6), 89-105.

[18] Rahman, S., Kumar, V., & Iqbal, T. (2022). DevOps design patterns: Supporting continuous delivery with feature toggles and blue-green deployments. Agile Software Practices Journal, 21(1), 87-103.

[19] Stevenson, J., Ali, M., & Harris, R. (2021). Performance implications of design patterns in real-time game development. Journal of Game Development, 29(1), 47-65.

[20] Stevenson, J., et al. (2022). Edge computing design patterns for IoT applications. Future Networks Journal, 28(1), 33-51.

[21] Wang, P., & Li, Z. (2023). AI-driven evolution of MVC architecture: Patterns and practices for integrating machine learning in web applications. Journal of Software Design and AI, 50(3), 66-78.

[22] Krishna, K. (2022). Optimizing query performance in distributed NoSQL databases through adaptive indexing and data partitioning techniques. International Journal of Creative Research Thoughts (IJCRT). https://ijcrt. org/viewfulltext. php.

[23] Krishna, K., & Thakur, D. (2021). Automated Machine Learning (AutoML) for Real-Time Data Streams: Challenges and Innovations in Online Learning Algorithms. Journal of Emerging Technologies and Innovative Research (JETIR), 8(12).

[24] Murthy, P., & Thakur, D. (2022). Cross-Layer Optimization Techniques for Enhancing Consistency and Performance in Distributed NoSQL Database. International Journal of Enhanced Research in Management & Computer Applications, 35.

[25] Murthy, P., & Mehra, A. (2021). Exploring Neuromorphic Computing for Ultra-Low Latency Transaction Processing in Edge Database Architectures. Journal of Emerging Technologies and Innovative Research, 8(1), 25-26.

[26] Mehra, A. (2024). HYBRID AI MODELS: INTEGRATING SYMBOLIC REASONING WITH DEEP LEARNING FOR COMPLEX DECISION-MAKING.

[27] Thakur, D. (2021). Federated Learning and Privacy-Preserving AI: Challenges and Solutions in Distributed Machine Learning. International Journal of All Research Education and Scientific Methods (IJARESM), 9(6), 3763-3764.

[28] Goe, M. S., & Martey, P. (2020). The influence of leadership on employee commitment to small and medium enterprises.

[29] Rahman, M.A., Uddin, M.M. and Kabir, L. 2024. Experimental Investigation of Void Coalescence in XTral-728 Plate Containing Three-Void Cluster. European Journal of Engineering and Technology Research. 9, 1 (Feb. 2024), 60–65. https://doi.org/10.24018/ejeng.2024.9.1.3116

[30] Rahman, M.A. Enhancing Reliability in Shell and Tube Heat Exchangers: Establishing Plugging Criteria for Tube Wall Loss and Estimating Remaining Useful Life. Journal of Failure Analysis and Prevention, 24, 1083–1095 (2024). https://doi.org/10.1007/s11668-024-01934-6

[31] Rahman, Mohammad Atiqur. 2024. “Optimization of Design Parameters for Improved Buoy Reliability in Wave Energy Converter Systems”. Journal of Engineering Research and Reports 26 (7):334-46. https://doi.org/10.9734/jerr/2024/v26i71213

[32] Julian, Anitha, Gerardine Immaculate Mary, S. Selvi, Mayur Rele, and Muthukumaran Vaithianathan. "Blockchain based solutions for privacy-preserving authentication and authorization in networks." Journal of Discrete Mathematical Sciences and Cryptography 27, no. 2-B (2024): 797-808.

[33] Zhu, Yue. "Beyond Labels: A Comprehensive Review of Self-Supervised Learning and Intrinsic Data Properties." Journal of Science & Technology 4, no. 4 (2023): 65-84.

[34] Elemam, S. M., & Saide, A. (2023). A Critical Perspective on Education Across Cultural Differences. Research in Education and Rehabilitation, 6(2), 166-174.

[35] Rahman, M.A., Butcher, C. & Chen, Z. Void evolution and coalescence in porous ductile materials in simple shear. Int J Fracture, 177, 129–139 (2012). https://doi.org/10.1007/s10704-012-9759-2

[36] Rahman, M. A. (2012). Influence of simple shear and void clustering on void coalescence. University of New Brunswick, NB, Canada. https://unbscholar.lib.unb.ca/items/659cc6b8-bee6-4c20-a801-1d854e67ec48

[37] Y. Pei, Y. Liu, N. Ling, Y. Ren and L. Liu, "An End-to-End Deep Generative Network for Low Bitrate Image Coding," 2023 IEEE International Symposium on Circuits and Systems (ISCAS), Monterey, CA, USA, 2023, pp. 1-5, doi: 10.1109/ISCAS46773.2023.10182028.

[38] Y. Pei, Y. Liu and N. Ling, "MobileViT-GAN: A Generative Model for Low Bitrate Image Coding," 2023 IEEE International Conference on Visual Communications and Image Processing (VCIP), Jeju, Korea, Republic of, 2023, pp. 1-5, doi: 10.1109/VCIP59821.2023.10402793.