**TITLE:**

A Comparative Study on Software Maintainability Strategies

**ABSTRACT:**

The primary goal of this study is to address the challenges encountered in day-to-day software maintenance and explores solutions to enhance maintainability. The research presents two distinct approaches implemented in .NET API: a traditional layered architecture and a refined solution incorporating Clean Architecture principles. The study involved the creation of two solutions, comparing a conventional layered approach with a Clean Architecture-based solution utilizing an API that uses MS SQL database. The study involves participants who are developers and software engineers with experience in real-world software maintenance scenarios. The results provide valuable insights into the effectiveness of Clean Architecture in improving software maintainability. This research contributes to the ongoing discourse on software design paradigms and offers practical implications for developers aiming to optimize the maintainability of their systems.

**INTRODUCTION AND OVERVIEW:**

**Introduction to the Area**

The software development landscape continually evolves, with an increasing emphasis on the importance of software maintainability. Software’s are often used for business or many different needs and in today’s fast changing environment the business requirement changes very often which sometime makes software’s complex which in turn makes it difficult and sometimes even impossible to maintain as per business needs. As a result, sometimes a new system needs to be launched that requires lot of time, effort and money. There are different principles & design in software that are developed for developing a better software. Using some of it based on research and requirements trying to use clean principles to develop an architecture that will improvise better software maintainability.

**Review of Main Theories and Research:**

Prior research in software engineering has extensively explored various aspects of maintainability. Studies have investigated the impact of design patterns, code modularity, and different architectural principles on the ease of maintenance.

**Evaluation of Literature:**

Evaluating the existing literature reveals a range of methodologies and practices for enhancing software maintainability. While layered architectures have been widely adopted, there is a growing interest in Clean Architecture principles as a potential solution. Evaluative comparisons between these approaches can shed light on their respective strengths and weaknesses.

**Identification of Research Gap:**

Despite the wealth of research on software maintainability, there exists a gap in understanding the comparative effectiveness of traditional layered architectures and those incorporating Clean Architecture principles. This research aims to address this gap by providing empirical evidence on the impact of architectural choices on software maintainability.

**Importance of Contribution:**

The significance of this research lies in its potential to inform developers and software engineers about the most effective architectural approaches for long-term software maintainability. Given the rapid evolution of technology and the increasing complexity of software systems, identifying best practices is crucial for sustainable development practices.

**Description of Research Design:**

The study employs a comparative research design, contrasting a traditional layered architecture with a Clean Architecture-based solution. This design aligns with the theoretical issues addressed, emphasizing the maintainability aspects inherent in Clean Architecture.

**MAIN BODY**

**Introduction to software maintainability**

Software maintainability is a leading topic in modern software engineering, addressing the constantly changing business needs and technical innovations. The dynamic field of software development necessitates flexibility and evolution because applications are essential for a wide range of corporate requirements. Software systems can become complex due to the unrelenting speed of change in business demands, which makes it difficult and often even impossible to align them with changing business needs.

Software and business requirements often dance in a complex way that results in a dilemma. Software systems need to change as companies do. Nonetheless, these changes' speed and unpredictable nature can leave codebases that are challenging to understand, maintain, and grow. This complexity significantly increases the danger to the system's general health in addition to impeding software's ability to respond to business needs.

The result of this dilemma is often seen in the choice to introduce whole new systems. But this is an expensive and time-consuming task that requires a lot of resources and frequently interferes with ongoing business activities. Therefore, it is crucial for enterprises to guarantee that their software remains relevant and agile throughout its lifecycle, in addition to building it initially with a strong and useful design.

The concepts, guidelines, and methods intended to deal with these issues are summed up in the idea of software maintainability. It entails developing software systems that are easily adjustable, scalable, and work as well as efficiently and functionally. To put it simply, maintainable software is made to adapt to requirements changes without breaking its structure or adding too much technical debt.

This study aims to explore the core of this important aspect of software engineering. It attempts to shed light on how maintainable traditional layered architectures and those that follow Clean Architecture's tenets are in comparison by examining the effectiveness of various architectural paradigms. The study aims to provide best techniques for software systems to stay robust, flexible, and adaptive in the face of changing demands by empirically exploring key concerns that affect developers and businesses alike.

We will explore the complexities of software maintainability and add to the continuing conversation about developing sustainable software solutions by exploring the literature, methodological approaches, and empirical findings in the parts that follow. In addition to exploring technological subtleties, this research aims to address practical issues that businesses and developers encounter when attempting to manage the intricate interactions between software and dynamic operating environments.

**Literature Review:**

**Review of Existing Literature on Software Maintainability:**

Software maintainability is highlighted as playing a crucial part in the software development lifecycle by a wealth of ideas, models, and research findings found in the literature. Maintainability is a crucial characteristic of software evolution, as defined by researchers like Lehman and Belady, who set the groundwork for later research. More recently, a variety of topics have been studied, including how design patterns, modularity, and readability of code all contribute to software systems' ease of maintenance.

**Evaluation of Traditional Layered Architectures:**

Software development has traditionally been based on traditional layered structures, which provide an organized method of arranging code. Their broad acceptance, simplicity of comprehension, and distinct division of responsibilities are their main advantages. Critiques do, however, surface when systems grow or alter frequently. Layers' rigidity can cause tight coupling, which makes it difficult to isolate changes and preserve flexibility. Furthermore, conventional architectures could find it difficult to adjust to changing business needs without causing a chain reaction of changes.

**Examination of Clean Architecture Principles:**

By placing a higher priority on modularity, testability, and maintainability, Robert C. Martin's Clean Architecture promotes a paradigm change. Its core ideas, such as the segmentation of concerns and the Dependency Rule, present a strong substitute. Resilient and flexible systems with a core that is independent of external dependencies are encouraged by clean architecture. As a result, there are fewer chances of unexpected effects when changes are made to the codebase, making it more maintainable. Development efforts are in line with the application's main value proposition because to the focus on business rules at its core.

**Identification of Gaps in the Literature:**

Some gaps remain in the literature, despite the fact that it offers useful insights into software maintainability and the comparison of architectural paradigms. There aren't many thorough empirical studies in the literature that directly compare the maintainability of traditional layered systems with ones that adhere to Clean Architecture principles. A lot of research depend on theoretical debates and rarely offer detailed assessments of real-world events. The objective of this study is to close this gap by providing a comprehensive empirical investigation that looks at the subtleties and practical implications of software maintainability.

**Methodology**

**Research Design**

Using a comparative research design, this study compares the maintainability of software systems built with typical layered architectures with those that follow the guidelines of Clean Architecture. A comprehensive examination is done on

two solutions that is developed one in Layered architecture and other in Clean architecture based performing same operations. A more detailed knowledge of how architectural decisions affect software maintainability is made possible by the comparative design.

**Participant Selection**

Participants in this study include engineers and software developers who have worked on systems created with Clean Architecture or traditional layered architectures. The selection criteria guarantee a varied representation of projects with varying dimensions, complexities, and domains.

**Data Collection**

A comprehensive approach applying quantitative methodologies is used in data collection. To evaluate modularity, readability, and adaptability, a codebase analysis will be done. Developer experiences with software maintenance will be analysed through surveys and interviews, highlighting both triumphs and obstacles.

**Implementation of Clean Architecture**

An API project will be developed that use the Clean Architecture principles, in accordance with Robert C. Martin's guidelines. The implementation will prioritize the dependency inversion concept, segregate components, and provide distinct boundaries between layers.

**Implementation of Layered Architecture Assessment**

Similarly, An API will be developed that used layered architecture guidelines by following Martin flowers guidelines. Which focuses on how to keep layers separate and interact with each other and modular

**Data Analysis**

Statistical analysis will be applied to quantitative data, such as maintainability metrics and code analysis results. Thematic analysis of qualitative survey data will give a comprehensive picture of developers' attitudes toward maintainability.

**Rigor and Reliability**

The study will use quantitative measurements such as cyclomatic complexity, class coupling, depth of inheritance to calculate maintainability index, in accordance with established coding standards to assure rigor. A code analysing metric tool will be used with code base to calculate related metrics. For qualitative, a google survey form will be circulated amongst senior developer, architects who have experience on architecture to get their opinion about their experience.

**Ethical Considerations**

Ethical considerations include obtaining informed consent from participants, ensuring confidentiality, and safeguarding against potential biases. The study adheres to ethical guidelines outlined by relevant institutional review boards.

**Introduction to software applications**

A software application is a computer program created to carry out particular operations, resolve issues, or offer services to consumers. These programs, sometimes referred to as apps, software, or apps, are made with programming languages and can be anything from basic utility to intricate systems. Word processors, online browsers, video games, and business management applications are a few examples.

**Importance of Software’s in Today's World:**

Software programs are essential to many facets of daily life and business in today's society. They make it possible for people to communicate, be productive, have fun, and automate many different tasks. Software applications have a vast impact and are essential to modern life, ranging from enterprise-level software that manages complicated operations to smartphone apps that promote social interactions.

**Developing or maintaining Software can be a challenging task:**

The dynamic nature of corporate environments, developing technologies, the requirement for continuous security measures, and the intrinsic complexities that build up over the course of the software's lifecycle make software maintenance difficult. A proactive and comprehensive approach is necessary to properly handle these difficulties and ensure successful maintenance.

Below are some general Challenges faced while working with software’s:

Changing Requirements: User requirements and business needs are subject to change throughout time. It is a difficult undertaking to modify the software to accommodate these changes without sacrificing current functionality.

Technological Advancements: Technology is advancing so quickly that some components or technologies may become outdated. It is difficult to update or migrate software to newer technology while maintaining compatibility.

Legacy Code and Technical Debt: Software develops technical debt and legacy code over time, which makes it more difficult to comprehend, alter, and expand. It can take some effort and careful refactoring to address these problems.

Dependencies and Integration: Software frequently uses third-party frameworks, libraries, or APIs. Modifications to these dependencies or merging with new services may cause compatibility problems that call for careful adjustment.

Bug Fixes and Issue Resolution: Finding and resolving defects or problems can be time-sensitive in order to preserve software reliability, particularly in big and sophisticated systems that require extensive testing and debugging.

Evolving Business Processes: Modifications to business procedures or organizational structures could call for changes to the software. One constant problem is getting the software to match the changing needs of the organization.

**Software architecture and why it plays an important role in developing a Software?**

The term "software architecture" describes the discipline of designing software systems and their high-level structure. It includes all of the design choices and patterns that influence how a software program or system functions, behaves, and is organized overall. Making strategic decisions in software architecture is necessary to guarantee that a system satisfies both functional and non-functional needs, including security, maintainability, scalability, and performance.

Key aspects of software architecture include:

Components and Modules: Breaking down a system into smaller parts, or modules, each in charge of particular functions. The architecture is defined by the connections and exchanges among these elements.

Patterns and Styles: Utilizing architectural styles and design patterns to solve persistent issues and accomplish certain objectives. Model-View-Controller (MVC) architecture, microservices architecture, and layered architecture are a few examples.

Data Management: specifying the internal storage, access, and management of data in the system. Decisions about databases, data models, and data flow fall under this category.

Integration and Communication: figuring out how various parts or services interact and communicate with one another. Selecting data exchange techniques, APIs, and communication protocols falls under this category.

Maintainability: Designing an architecture that makes maintenance and updates easier. This entails taking documentation, good code practices, and code modularity into account.

Adaptability to Change: Constructing an architecture that is flexible enough to adjust to new needs and developing technology. This covers methods for making the system future-proof.

Since it establishes the framework for the entire system, software architecture is a crucial stage in the software development lifecycle. A scalable, maintainable, and business-aligned system can result from wise design choices.

**Layered architecture**

**Characteristics of Conventional Layered Architecture:**

The core method of software design known as "Layered Architecture" groups components into distinct layers with distinct roles. A design architecture that divides a software system into discrete layers or tiers, each in charge of particular functional areas. This strategy encourages modularity, which facilitates the management, upkeep, and scalability of complicated applications.

The basic framework frequently consists of:

**Presentation Layer:**

oversees the interactions and user interface. consists of parts for displaying data, processing user input, and producing views.

Responsibility –

Manages the user interface (UI) and user interactions.

Presents data to users and captures user inputs.

Working Flow -

User Interaction: Requests or inputs from the user cause presentation layer operations to happen.

Interface Rendering: Depending on what the user does, the presentation layer renders the relevant user interface.

Communication with Business Logic Layer: The presentation layer talks to the business logic layer when it needs to process or retrieve data.

**Application layer:**

Includes the fundamental Business logic and rules. In charge of handling and modifying data in accordance with established guidelines.

Responsibility -

Includes the fundamental business logic and rules.

coordinates application behaviour, carries out business rules, and processes data.

Working Flow –

Receive Requests: The Presentation layer sends data or requests to the business logic layer.

Data processing: It modifies and processes data in accordance with established business guidelines.

Execution of Business Rules: carries out the fundamental business logic by applying rules and making judgments.

Interaction with Data Access Layer: The business logic layer interacts with the data access layer to store and retrieve data.

**Data Access Layer:**

Manages communication with the data storage system underneath. Includes data retrieval, transactions, and database queries.

Responsibility -

Manages interactions with the underlying data storage.

Executes database queries, transactions, and handles data retrieval and storage.

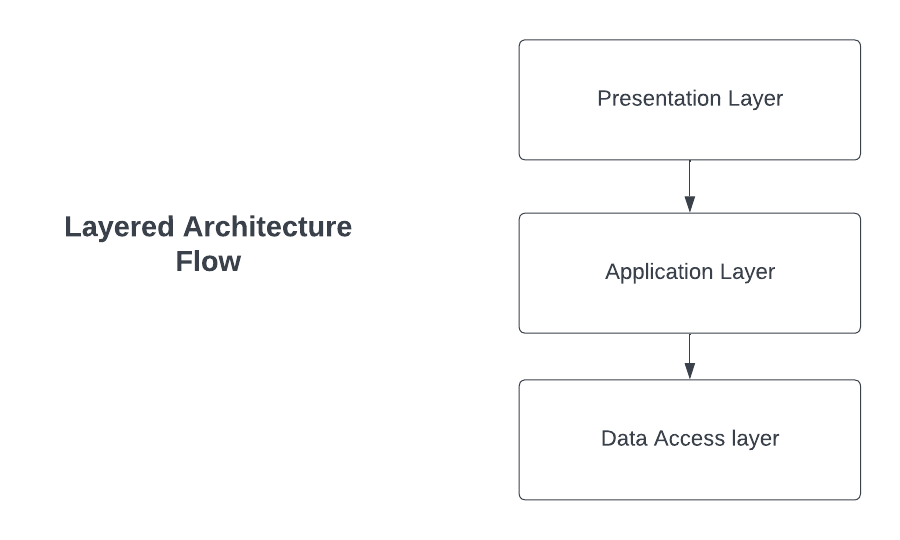
Working Flow-

Receive Requests: The business logic layer sends requests to the data access layer for data retrieval or storage.

Database Interaction: It communicates with the database by carrying out transactions or queries as required.

Data Storage/Retrieval: This process involves storing or retrieving data from a database.

Return Data to Business Logic Layer: The business logic layer receives the requested data or an operation's confirmation from the data access layer.



Because of the separation of concerns made possible by this structured flow, it is simpler to grow, maintain, and change different parts of the program separately. But problems like tight coupling and maintainability constraints could arise, which is why other architectures like Clean Architecture should be taken into consideration to solve

**Benefits:**

**Modularity and Separation of Concerns:**

Conventional layered structures encourage modularity, which makes it easier to divide up the concerns. Because each layer is responsible for a different purpose, certain features are easier to understand and manage.

**Clear Structure:**

A well-defined and structured framework for development is offered by the layered structure. This clarity makes it easier for development teams to collaborate and for new engineers to be on boarded

**Widespread Adoption:**

Within the software development community, traditional layered structures have gained widespread adoption and are commonly understood. This familiarity makes system maintenance and development easier.

**Reusability of Components:**

Every layer's components are frequently made to be reused. This lowers redundancy and boosts efficiency by enabling developers to take advantage of already-existing functionalities.

**Challenges and Limitations**

**Tight Coupling**

Tight coupling between levels can occur in traditional layered designs. Modifications to one layer may have repercussions that cascade down to dependent levels, reducing the system's ability to adapt.

Observations: When updating or modifying a system, tight connection may present difficulties. Changing the business logic could require modifications to the data access and presentation layers, which would require more work in the development process.

**Limited Flexibility**

Literature Perspective: Conventional layered architectures may be less flexible due to their predetermined structure, which makes it difficult to adjust to changing business needs without requiring significant changes to several layers.

Observations: Agile development cycles highlight a lack of flexibility. Quick iterations may be impeded by rapid changes in user requirements that require updates in numerous layers.

**Maintainability Concerns:**

Although maintainability is the goal of layered systems, the degree of modularization may not always lead to simple maintenance. Large-scale projects built with conventional layered architectures can become difficult to maintain as codebases grow.

Observations: The simplicity of maintenance may be impacted by the need for significant navigation among interconnected levels in order to locate and understand certain features.

**Technology Stack Dependency:**

Conventional layered architectures frequently show reliance on particular technological stacks. When integrating or moving with new technology, this may provide difficulties.

Observations: When attempting to introduce modern technologies, dependence on a specific technological stack becomes apparent. There may be compatibility problems and a need for significant reworking.

**Clean Architecture**

A new paradigm in software design, Clean Architecture emphasizes basic ideas over particular technology and implementation specifics. Creating software systems that are reliable, flexible, and easy to maintain is the fundamental goal of the Clean Architecture philosophy. It highlights how important business principles should be independent from the complexities of external frameworks, databases, and user interfaces, and it promotes a clear division of responsibilities.

Clean Architecture is based on the fundamental principle of long-term sustainability. Clean Architecture prioritizes readability and clarity by organizing code around important business principles and use cases. It acknowledges that software development will inevitably change and offers an organized method that lets systems change without sacrificing their essential features.

Developers are encouraged by Clean Architecture to go beyond the short-term needs of a project and take into account the timeless rules that guide efficient software architecture. It removes superfluous dependencies, guaranteeing that an application's essential logic is isolated and unaffected by the rapidly evolving tool and technology landscape.

Clean Architecture provides a conceptual framework that can be adjusted to different settings, but it does not mandate a particular set of layers or implementation specifics. This flexibility is a key component of Clean Architecture since it recognizes the variety of software projects and offers a set of standards that are universally applicable.

Essentially, Clean Architecture is an approach to software design that is not limited to any one particular technology. It motivates programmers to design systems that are not just effective and functional but also resistant to change. Through its embrace of the ideals of independence, clarity, and long-term maintainability, Clean Architecture is a paradigm shift toward a more deliberate and long-lasting approach to software development.

Characteristics of Clean Architecture-

The basic framework frequently consists of:

**Presentation Layer:**

Responsibility-

Charged with managing user interfaces and user engagement.

transforms application-layer data to make it suitable for UI rendering or Response of a request in API’s.

Workflow - Gathers events and user input from the user interface. Calls the relevant Application layer methods. It pulls information from the application layer and modifies it before presenting it. Renders the user interface with the modified data.

**Application Layer:**

Responsibility- Executes use cases and business rules particular to the application.

coordinates the data transfer between the levels of the presentation, domain, and infrastructure.

Workflow - Gathers requests from the layer of Presentation. Applies business rules while validating and processing the requests. Uses Domain layer methods to access and alter key business items. Interacts with the infrastructure layer through communication in order to store data or use external services. Provides changes or results back to the Presentation layer.

**Domain Layer:**

Responsibility- Contains essential business entities and rules.

Workflow – This layer consists of business entities or also called business models. Application layer uses this models for its business logic and fill data in this model through outside world parameter or using infrastructure layer then perform some actions that may be to check some business rules across execution of code or for some crud operations.

**Infrastructure Layer:**

Responsibility- Handles information and technology from outside sources. Includes interfaces that have been implemented and declared at the Application and Interface Adapters. It can be any third party tool, service or database.

Workflow – Carries out data access methods to store and retrieve data. Integrates with tools, frameworks, and other services. Provide the interfaces specified in the Application and Interface Adapters layer’s tangible implementations. Returns data or outcomes via communication to the Application layer.

Benefits-

Framework Independence:

Characteristic: The primary focus of clean architecture is on the entities and business rules' total independence from other frameworks, libraries, and tools.

Significance: Preserves the integrity of the fundamental business logic while guaranteeing flexibility and adaptability to technological advancements.

encourages long-term maintainability and lowers the danger of technological lock-in.

Separation of Concerns:

Characteristic: The architecture is arranged into discrete layers, each of which is in charge of a particular task, encouraging a definite division of responsibilities.

Significance: By separating various components of the system, including as business rules, application logic, and external dependencies, it becomes easier to read and manage.

makes it simpler to understand and adjust each component separately.

Focus on Business Logic:

Characteristic: The primary function of the innermost layer, or Domain, is to define and encapsulate the major business entities and regulations.

Significance: keeps the major emphasis on the important components of the application, avoiding the dispersion of business logic across the system.

improves readability and clarity by consolidating the essential business ideas.

Strict Dependency Rule:

Characteristic: Dependencies follow a rigorous directionality, moving from outer levels (Infrastructure) to inner layers (Application, Domain, and Presentation).

Significance: Reduces the reliance of inner layers on outside details, thereby improving maintainability.

simplifies system evolution and modification without compromising the essential business logic.

Adaptability:

Characteristic: A feature of clean architecture is that it can adapt to modifications in databases, frameworks, or other technologies without impacting the main business logic.

Significance: Reduces the dangers of technology obsolescence and makes sure the design can change to meet new needs.

facilitates the smooth addition of new technologies as required.

Clear Boundaries with Layers:

Characteristic: The architecture is structured into layers – Presentation, Application, Domain, and Infrastructure – each with its distinct purpose.

Significance: Provides a conceptual framework for organizing code, making it easier to reason about and maintain.

Clarifies the responsibilities and interactions of different parts of the system.

Focused Interface Adaptors:

Characteristic: By converting data formats, interface adapters (Presentation layer) serve as a link between the application and external interfaces.

Significance: guarantees a clear division of responsibilities between the external delivery mechanisms—like databases or UI frameworks—and the main application logic.

permits the development of several adapters for various interfaces without changing the fundamental business regulations.

Testability:

Characteristic: A feature of clean architecture is that testability is given priority by isolating business rules and the main application logic from outside information.

Significance: Provides the ability to build thorough test suites for the most important parts independently of external dependencies.

encourages the use of test-driven development techniques to produce reliable software.