**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:**

The field of software development is one that is dynamic and always changing due to ongoing technological breakthroughs, new processes, and shifting business and user needs. In other words, maintaining a software. This environment is characterized by the quick speed of innovation, which is driven by developing technologies and altering consumer needs. Software development is not just a process; it's an ecosystem that is always changing, with new frameworks, tools, and paradigms appearing all the time. As such, companies and developers need to stay up to date on the newest developments. Because of its dynamic nature, the software creation industry is always evolving, which presents opportunities and challenges for individuals working in the field **(Aakriti Gupta and Shreta Sharma 2015)**.

About 60% of software cost goes under maintenance **(Uttamjit Kaur, Gagandeep Singh 2015)**. Software maintainability is becoming increasingly important in the modern software engineering era, and this trend is becoming more and more apparent. The industry has come to realize that while the creation phase receives significant attention, the real test of software success is the product's capacity to be successfully maintained over time. Software systems are dynamic that change with the constantly shifting environment they are designed to support. This increasing awareness is a result of the understanding that software products have long lives beyond the time they are released into the market, and that their capacity to be easily modified, improved, and fixed is essential to their continuous existence **(N. Gold; A. Mohan; C. Knight; M. Munro 2004).**

Nevertheless, there are several difficulties with this increased focus on software maintainability. Effective software maintenance is severely hampered by the complexity of contemporary software systems and the constant change in business needs. Regular upgrades, shifting legal requirements, and technology developments all contribute to a situation where software upkeep becomes a complex undertaking. The difficulties are not limited to codebase management alone; they also include system architecture, documentation, and striking a careful balance between adding new features and maintaining current functioning. In the face of these obstacles, the software development industry struggles to find the best ways to keep software systems resilient, adaptive, and sustainable. This is because the landscape of software development is always changing **(Mari, Eila 2003)**.

A software system's maintainability is a very wide term but it is also greatly dependent on its software architecture. A well-designed architecture offers a framework that makes it easier to separate concerns, modularize applications, and assign explicit roles to different components. This organization provides efficient impact analysis during changes, and it also makes it possible for developers to understand and update certain components without disrupting the system as a whole. The architecture's capacity to accommodate changing needs is improved by its flexibility and extensibility. Architecture promotes clean code practices and lowers the risk of technological debt by facilitating communication and documentation. A strong architecture also affects the processes of testing and debugging, helping to create a system that is resilient and sustainable in the face of changing software landscapes in addition to being performance-optimized. Software architecture is an essential part of software that supports a durable, flexible, and maintainable software system across the course of its lifecycle **(Uttamjit Kaur, Gagandeep Singh 2015**).

**Problem Statement**

Despite the wealth of research on software architecture in terms of maintainability, there exists a gap in understanding direct comparative effectiveness of architectures using traditional layered architectures as it very common architecture been adopted **(Juha Savolainen, Varvana Myllarniemi 2009)** with clean architecture. This research aims to address this gap by adding empirical evidence on the impact of architectural choices on software maintainability that Clean architecture provides better maintainability **(Yosep Novento Nugroho, Dana Sulistyo Kusumo Muhammad Johan Alibasa 2022)**.

**Motivation for the research**

This study is driven by an essential knowledge deficit regarding the direct influence of architectural decisions on the practical maintainability of software systems. Although Clean Architecture is theoretically supported, there isn't much concrete data that does direct comparison to show that it performs better in practice than standard layered designs. The need to close this knowledge gap and offer useful insights to software practitioners motivates this research. The goal of the study is to enable developers and architects to make well-informed decisions by exploring the true impact of Clean Architecture on software maintainability through empirical data and artefact-based comparisons. **(Yosep Novento Nugroho, Dana Sulistyo Kusumo Muhammad Johan Alibasa 2022)**.

**Research aim and objectives**

The aim of this research is to construct a practical scenario based on developing artefacts and subsequently employ analytical tools to evaluate and compare the maintainability of software systems built on Layered and Clean Architectures. This involves developing a representative scenario reflecting the complexities and dynamics of software development. Through the creation of this scenario, the study seeks to simulate a context that implements both architecture using their principles and industry practices. Following scenario creation, analytical tools were employed to assess and quantify the maintainability aspects of each architecture **(Yosep Novento Nugroho, Dana Sulistyo Kusumo Muhammad Johan Alibasa 2022)**. This approach ensures a hands-on investigation into the practical implications of architectural choices, contributing valuable insights to guide software practitioners in selecting the architecture that best aligns with the goal of achieving superior maintainability in real-world development scenarios.

**Literature Review**

**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 **(Mari, Eila 2003)**.

The subtleties of software maintainability have been thoroughly studied in a large body of study in the field of software engineering. Studies that look into how different architectural principles, design patterns, and code modularity affect, how easy it is to maintain software systems are essential to this investigation. Recurring solutions to common design problems are known as design patterns, and they have a significant impact on how maintainable a software system is. Additionally, the division of code into modular components has received a lot of attention. Studies have shown how modular design techniques improve software's readability and flexibility, two qualities that are essential to long-term maintainability **(Morteza Ghasemi, Sayed Mehran Sharafi, Ala Arman 2015).**

The literature also examines the architectural concepts that govern software systems and clarifies how they affect maintenance procedures. This body of work provides the background knowledge needed to investigate the relative effectiveness of Clean Architecture principles and conventional layered structures in the next study, as well as the fundamental knowledge basis for comprehending maintainability in software engineering.

**Evaluation of Literature:**

A critical analysis is conducted in the review of the literature to appraise approaches and procedures targeted at improving software maintainability. The reviewed literature recognizes that layered architecture which have long been a popular method in software design have been widely adopted. Software components are arranged into discrete layers in layered architectures, which frequently indicate various functionalities or abstraction levels. **( Juha Savolainen, Varvana Myllarniemi 2009)**

The literature notes that while layered architectures are acknowledged, there is also growing interest in the ideas of Clean Architecture. A contemporary model known as "Clean Architecture" places a strong emphasis on modularity, concern separation, and an obvious hierarchy of relationships. This new architectural paradigm is becoming more and more popular because it offers a scalable and organized base for software development, which can help address maintainability issues **(Robert C. martin 2017)**.

In the contrast of layered architectures and the emerging interest in Clean Architecture sets the stage for a contextualized exploration. By evaluating these architectural paradigms, the literature review lays the groundwork for a comparative analysis. This analysis aims to delve deeper into the practical implications of these architectures on software maintainability, thus addressing the identified research gap.

**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 **(Yosep Novento Nugroho, Dana Sulistyo Kusumo Muhammad Johan Alibasa 2022)**.

**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:**

Despite software maintainability is a broad term but and this research to maintainability for developers through coding point of view **(Uttamjit Kaur, Gagandeep Singh 2015)**. The study employs a comparative research design, contrasting a traditional layered architecture with a Clean Architecture-based solution. This design aligns with the issues addressed, emphasizing the maintainability aspects inherent in Clean Architecture **(Yosep Novento Nugroho, Dana Sulistyo Kusumo Muhammad Johan Alibasa 2022)**.

.

**Methodology**

**Research Design**

Using a comparative research design, this study compares the maintainability of software systems built with typical layered architectures **(Jun Tie; Jia Jin; Xiaorong Wang)** with those that follow the guidelines of Clean Architecture **(Robert C. martin 2017)**. 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 maintainability, a codebase analysis is done. Developers, architects were involved to answer survey questions based on their experience and analysed to develop artefact with clear idea in mind-set. Also based on two artefact implemented metric data is gathered to quantify maintainability of both architectures

**Implementation of Clean Architecture**

An API project is developed that uses the Clean Architecture principles, in accordance with Robert C. Martin's guidelines. The implementation prioritizes the dependency inversion concept, segregate components, and provide distinct boundaries between layers without dependency. **(Robert C. martin 2017)**

**Implementation of Layered Architecture Assessment**

Similarly, An API were developed that used layered architecture. Which focuses on how to keep layers separate, modularize and interact with each other and modular. **(Jun Tie; Jia Jin; Xiaorong Wang)**

**Data Analysis**

Statistical analysis was applied on quantitative data, such as maintainability metrics and code analysis results **(Yosep Novento Nugroho, Dana Sulistyo Kusumo Muhammad Johan Alibasa 2022)**. Thematic analysis of quantitative survey data give’s a comprehensive picture of developers' attitudes toward maintainability.

**Rigor and Reliability**

To calculate maintainability, the study uses quantitative measurements such as cyclomatic complexity, class coupling, depth of inheritance internally to calculate maintainability index **(Yosep Novento Nugroho, Dana Sulistyo Kusumo Muhammad Johan Alibasa 2022)**, in accordance with established coding standards to assure rigor using code metric tool. A code analysing metric tool is used with code base to calculate related metrics. A google survey form was 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.

**Summary**

This research focus from developer, architects point of view related to code base that can improvise maintainability of an application so that application lifespan can increase in terms of modification and adaptability of new technology. Therefore, it can save human efforts and cost for changing and adding new things basically comparing between clean and layered architecture and adding some evidences how clean architecture can provide better maintainability that can help developers and architects to make decision while choosing architecture to develop complex applications where dependency, adaptability and modification is a concern.

**MAIN BODY**

**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 **(J.S. O'Neal; D.L. Carver 2001)**.

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 **(F. Sultan; L. Chan 2000)**.

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 **(A. Monden; D. Nakae; T. Kamiya; S. Sato; K. Matsumoto 2002).**

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 Benjamin **(Klatt; Zoya Durdik; Heiko Koziolek; Klaus Krogmann; Johannes Stammel; Roland Weiss 2011)**.

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 **(Hongyu Zhang; Liang Gong; Steve Versteeg 2013)**.

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 **(W.M.N. Wan-Kadir, Pericles Loucopoulos 2004)**.

**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.

**(Robert C. martin 2017)**

**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 Implementation Rule according to The Common web application architectures guide learn. Microsoft 2023:

**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.

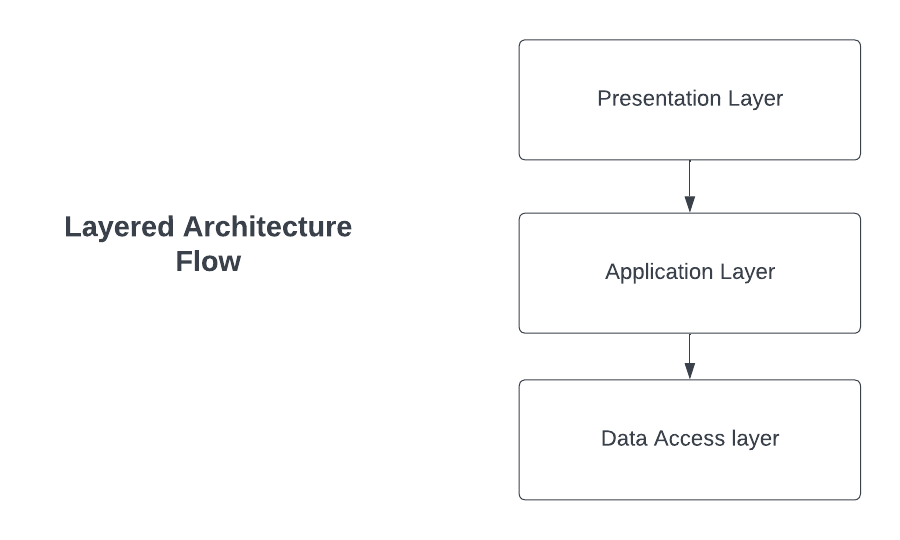
Working Flow-

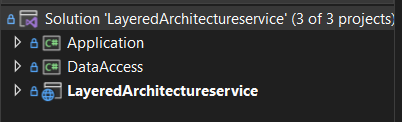
Receive Requests: The business logic layer sends requests to the data access layer for data retrieval or storage.

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

**Artifact Details**

The developed item for this applied research project is a layered architecture API project called LayeredArchitectureService. Its maintainability has been carefully considered throughout its design and implementation. The project follows the concepts of layered architecture, incorporating three separate layers: presentation, application and data access layers. Because each layer is designed to support modular development, maintainability is fostered. The artefact encourages clean code and unambiguous separation of concerns while also making CRUD operations easier for users and products by utilizing industry best practices. The final goal is to assess the maintainability index by using Visual Studio's code metrics tools to give a thorough picture of the project's sustainability and ease of further development and maintenance.

**Project Structure:**

Solution Name: LayeredArchitectureService

**API Project:** For every layer separate project is been created one as main and entry point of an API and other as application and data access.

**Description:** This architecture can be developed using UI, mobile app, web application and API but to keep things simple and focus on architecture itself decided to go with API project

**Key features:** Executing Create, Read, Update, and Delete (CRUD) operations.

RESTful Endpoints: Creating CRUD endpoints for domain such as User and product in application.

**Layers of Solution:**

**Presentation layer (Layered architecture service)**

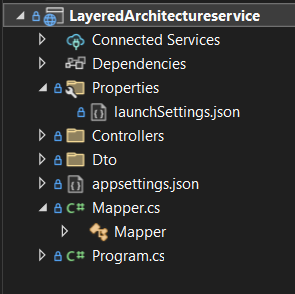
This is an entry point in the application also known as main project. Consumer makes request to API to this layer using http request passing parameters as per endpoints is been called.

Program.cs file is executed for initializing the project from this layer. When endpoint is called it calls related class function with given parameters from the Controllers folder.

DTO folder keeps all required DTO’s in the project basically to carry data within project or taking input and giving out put as response

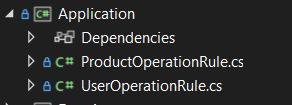
Properties folder contains launchsettings.json that has application launching related configuration which is by default present and for this artefact no changes are required

Mapper.cs is used for mapping DTO with domain objects using auto mapper package



**Application Layer:** Application Layer is a class project type. The business logic is stored in the application layer, which also coordinates communication between different components or projects.

Application layer or project is created to handle business rules and logics and isolate from rest of the system. It also uses Data transfer objects to send and receive data from other layers. It has two separate class file one is for user and other for product. This class files have all business rules implemented in the project.



**Data Access Layer:** This layer is also a class project type. Responsible communicating with data base and CRUD operation with database also this layer communications with application layer to take request and return data as return type to application layer which helps application layer to perform business logic.

Domain folder in this layer contains domain model which is User and Product in our case.

Repository Folder contains all crud logic to be performed with data base which application layer is using. ProductRepository and UserRepository are the two files in this folder for same operation. For performing CRUD operation Object relational mapper is being used which is basically a framework to perform database operation using objects from application. Context file is being used for as a starting and configuration for ORM i.e Entity frame work to be specific

**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 observed**

**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 **(Robert C. martin 2017)**

.

Characteristics of Clean Architecture-

The basic framework implementation Rule as per Robert C martin book on Clean architecture:

**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:**

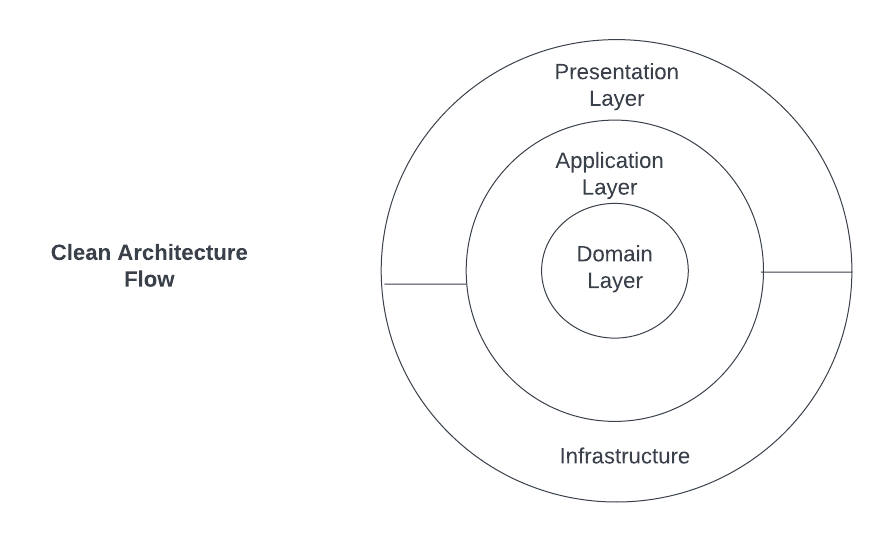
Responsibility- Contains essential business entities and rules.

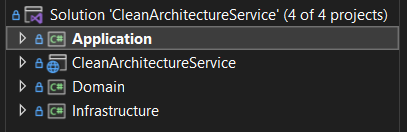
Workflow – This layer consists of business entities or also called business models. Application layer uses this models for its business logic and fill data in this model through outside world parameter or using infrastructure layer then perform some actions that may be to check some business rules across 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. None of the layers are directly dependent on each other instead they use abstraction to avoid tight coupling.

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.

**Challenges and Limitations Observed**

**Striking the Right Balance between Modularity and Simplicity:**

Finding the ideal ratio of modularity to simplicity is one of the major challenges of working with Clean Architecture. Although modularity is vital for scalability and maintainability, too much of it can add needless complexity.

Finding the ideal balance necessitates giving serious thought to the scope, intricacy, and team's experience with Clean Architecture concepts. An overemphasis on modularity can result in complex designs that are difficult to comprehend and manage. On the other hand, putting simplicity first could make the architecture less flexible. Reaching equilibrium makes that the system is both clearly and modularly designed for scalability, which facilitates development and maintenance.

**Maintaining Consistency with the Code:**

Maintaining consistency across the codebase can be difficult, particularly in bigger projects when several teams are working on different aspects of the system. Maintainability, readability, and adherence to Clean Architecture principles all depend on consistency.

Enforcing architectural requirements, coding standards, and making sure that every team member applies and understands Clean Architecture concepts consistently are all part of maintaining consistency. Integration of contributions from different team members could provide difficulties and cause the architecture to deviate from what was originally planned. Tackling this issue and maintaining architectural integrity need diligent code reviews, thorough documentation, and ongoing communication.

**Learning the Architecture Itself:**

Sometimes teams moving to Clean Architecture may find it difficult to overcome the early learning curve associated with this architectural style. It takes time and effort to comprehend the concepts and use them properly.

Understanding ideas like layering, separation of concerns, and dependency inversion are essential to learning Clean Architecture. Teams need to be able to adjust to the architectural limits and comprehend the relationships between various layers. It will need instruction, documentation, and real-world experience to overcome this obstacle. It is imperative to furnish team members with sufficient tools and assistance throughout the learning phase so they may comfortably implement Clean Architecture concepts in their day-to-day development tasks.

**Integration with Legacy Systems:**

It can be difficult to integrate Clean Architecture into projects that still contain legacy code. It may not be possible for legacy systems to integrate Clean Architecture concepts seamlessly, necessitating cautious approaches. It could be challenging to achieve a clear separation of concerns when there is legacy code present, which could reduce the amount of Clean Architecture's applicability.

**Comparison between Layered and Clean Architecture observed**

**Dependency Direction:**

Layered Architecture: Higher levels frequently rely on lower layers in a vertical fashion (e.g., presentation layer depends on business logic layer, which depends on data access layer). The specifics of lower layers' implementation are known to higher layers.

Clean Architecture: According to the Dependency Inversion Principle, dependencies are reversed. The outer layers rely on abstractions, but the central business logic (use cases) lies at its core. Details (such as databases and frameworks) rely on the independent, segregated core.

**Separation of Concerns:**

Layered Architecture: Horizontally, across layers, separation of concerns is accomplished (presentation, business logic, data access). Business rules may be distributed across multiple layers, leading to a less clear separation.

Clean Architecture: It is possible to clearly divide concerns both radially and horizontally. The core is kept apart from outside details by layers, which stand for various concerns (such as Use Cases for business logic). The concentration of business rules in the core improves maintainability and clarity.

**Testability:**

Layered Architecture: Testing can be difficult, particularly if business rules are dispersed among several layers. Setting up the whole application context is typically necessary for unit testing**.**

Clean Architecture: Because it is decoupled, core business logic is easily testable in isolation (using, for example, unit testing of use cases).

As dependencies get closer to the core, testing becomes easier and mocks and stubs can be used.

**Flexibility and Adaptability:**

Layered Architecture: A lack of flexibility may result from changes made to one layer that impact other layers. It could be difficult to replace a layer or adjust to new technologies.

Clean Architecture: Modifications to external details (frameworks, databases) barely affect the core. Permits a more flexible and adaptive design when modifications are localized.

**Technology Independence:**

Layered Architecture: Certain technologies may be tightly coupled with every layer. Modifications across layers may be necessary in order to adapt to new technology.

Clean Architecture: External technology has no bearing on core business logic.

The outer layers of technology adapt to them, allowing for easy upgrades and technological independence.

**Overall as maintainable:**

Layered Architecture: Maintainability could be difficult, particularly if changes affect several levels. Layers may grow interconnected over time, which can cause maintenance problems.

Clean Architecture: Better maintainability as a result of the distinct division of responsibilities. Modifications are restricted to particular layers, lowering the possibility of unforeseen outcomes.

**Adherence to SOLID Principles:**

Layered Architecture: It can be difficult to follow SOLID principles, particularly Dependency Inversion. High-level components frequently rely on low-level components.

Clean Architecture: Strong commitment to Dependency Inversion and other SOLID principles. The inversion of dependencies is made easier by abstractions and interfaces.

**Real-world Applicability:**

Layered Architecture: extensively utilized and comprehended, particularly in conventional business applications. Perhaps appropriate for less complex projects or scenarios.

Clean Architecture: Growing in acceptance, particularly in applications where adaptability and long-term maintainability are critical. Ideal for intricate systems with changing needs.

While Clean Architecture offers a more flexible, adaptable, and maintainable framework, traditional Layered Architecture overcomes some of its disadvantages by providing a clear and commonly understood approach. For projects where long-term sustainability and ease of maintenance are top goals, Clean Architecture is especially appealing due to its obvious separation of interests and adherence to concepts like Dependency Inversion.

**Analysis**

**Maintainability index**

A Maintainability Index is a composite statistic that measures how simple it is to maintain a software—a module, class, or project as a whole. The overall maintainability of the code is shown by a numerical score that is produced by combining multiple code metrics. A comprehensive summary of the variables commonly taken into account when determining the Maintainability Index is provided below:

Lines of Code (LOC): The software artefact’s complete line count of code. Although a greater LOC count may indicate more complexity, this is not always a bad thing. Code that is clear and legible is encouraged by the LOC component.

Cyclomatic Complexity: The number of independent paths through the code serves as a proxy for the complexity of the code. Elevated cyclomatic complexity may suggest heightened challenges in comprehending and sustaining the code.

Halstead Volume: Program length, vocabulary, and volume are examples of Halstead metrics that help determine how much work goes into maintaining code. The program's size is best indicated by the Halstead Volume.

Estimated Effort: The approximate amount of work needed to comprehend and update the code. It takes into account variables like quantity, cyclomatic complexity, and the quantity of delivered bugs.

Formula for Maintainability Index: These elements are used to create a formula that is used to calculate the Maintainability Index. Even though the precise formula varies, this is a typical representation:

Maintainability Index = 171 - 5.2 \* ln(Halstead Volume) - 0.23 \* (Cyclomatic Complexity) - 16.2 \* ln(Lines of Code)

The more the maintainability index count the more maintainable it is. **(Ilja Heitlager, Tobias Kuipers, Joost Visser 2007)**

**Practical implementation**

**Tool Integration:**

Code metric tools that are incorporated into development environments are used to construct the Maintainability Index. For instance, Visual Studio has tools like Code Metrics that offer this feature.

So for calculating maintainability index Code metric tool is used for calculation and display maintainability index in Visual studio IDE.

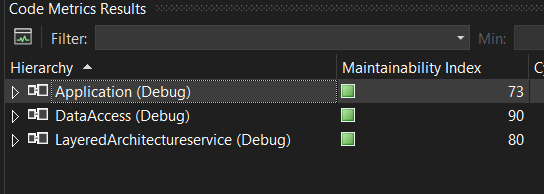
Using this tool helps to analysis results or to track even after changes are made to code.

Developers benefit from rules that are established for the Maintainability Index thresholds. Better maintainability, for instance, might be indicated by a higher score.

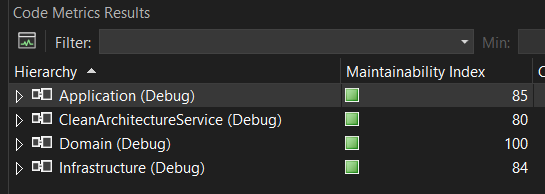
As a teaching tool, the Maintainability Index can assist developers in comprehending how their coding methods affect the software's maintainability. **(Code Metrics in Visual Studio," Microsoft.Learn 2023)**

**Applying Code metric tool on Artefact’s**

**Results of code metric on Layered architecture**



**Results of code metric on Clean architecture**



Comparing maintainability index of all layers between clean and layered architecture cannot be conducted directly as both have layers with some differences and importance’s based on their own architecture principles.

Basically first step is to analyse average maintainability index of entire solution which gives overall comparison.

**Finding the overall average of maintainability index of entire solution.**

**Average Maintainability Index** = Sum of Maintainability Index Values of Each Layer / Total Number of Projects

**Average Layered architecture maintainability index**

**=73**(Application Layer) + **90**(Data access layer) + **80**(Presentation layer) **/ 3**(Project count in solution)

**=243 / 3**

**= 81**

**Average Clean architecture maintainability index**

**=85**(Application Layer) + **84**(Infrastructure Layer) + **80**(Presentation layer) + **100**(Domain Layer)**/ 3**(Project count in solution) / 4 (Project count in solution)

**=349 / 4**

**= 87.25**

So from above overall comparison it can be seen clearly that clean architecture improved maintainability index.

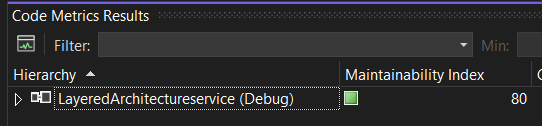
Both architectures are developed with same logical code to do same operations to get same output but using their own architecture principles. Hence all layers cannot be compared directly due to its own importance.

Below are the details of each layer or area wise analysis comparison

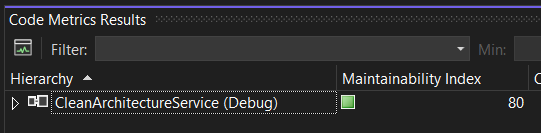
Presentation layer analysis:

Common Entry Point: The presentation layer functions as the common entry point for managing external requests in both Clean Architecture and Layered Architecture, particularly when it comes to HTTP APIs. Receiving incoming HTTP requests and returning response are its main duties. As there is no major difference in both architectures hence maintainability index of this layer in both architectures are same.

Layered Presentation Layer



Clean Presentation Layer

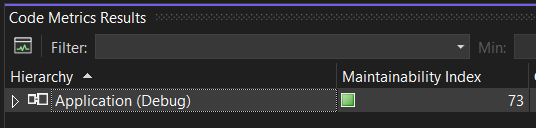


From above two presentation layer from two different architecture shows same maintainability index count that is 80.

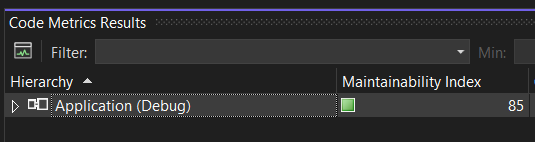
Application layer analysis:

This layer is the core of the application that contains actual business rules. So in both architectures it takes arguments from presentation layer, process and return response to the request. But the major difference is level of direct dependency it has on other parts of the system to get request, process data is the major thing.

Layered Application Layer



Clean Application Layer

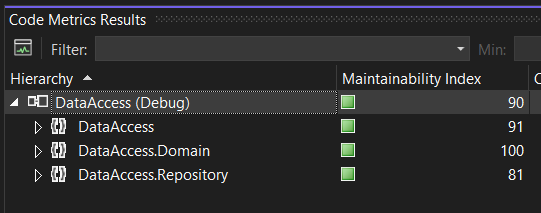


The Layered application layer maintainability index is 73 whereas the Clean architecture’s maintainability index is 85 which is a significance difference. Also as per the survey conducted with question asked that Which part of the code/module was mostly beneficial in terms of complexity and easy to change after working with clean architecture as per your experience from day to day work? And about 72% of the people choose business logic. Reason can be all other part of the system is meant for this layer that to perform business logic which connects other layers as well to persist data, get request etc. Hence this layer is the major challenge in terms of maintainability in long run specially for complex application. This is one of the major evidences that shows how clean architecture improves maintainability or in other word how it helps to manages tight dependency, adaptability issue etc.

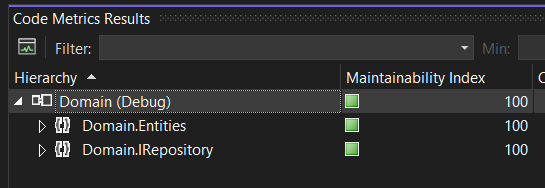
Domain Layer analysis:

This area in application basically consist of business domains in form of model object. That has data that be used to perform some business logic in application layer. It basically doesn’t have its own logical code hence no scope for improvisation.

Layered Domain.



Clean Domain.



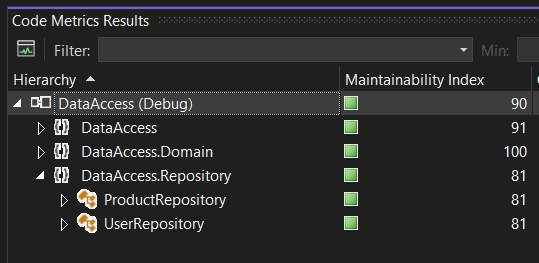
In above attachment we can see that domain is a part of data access layer in layered architecture as DataAccess.Domain but a separate layer in clean architecture as Domain as per their architecture principle guidelines. Also the maintainability index of domain area in both artefacts is 100 due to no logical code present except business entities.

Data or Infrastructure analysis

This part of the system in both the architecture involved related to database related operations using Object relational mapper basically a framework that performs CRUD with database. Usually we use this area by business logic based on requirement to perform database operations. So this part of architecture helps to keep database access logics separate from business logic. Most of the time after business logic this layers contains code but only related to database related operations.

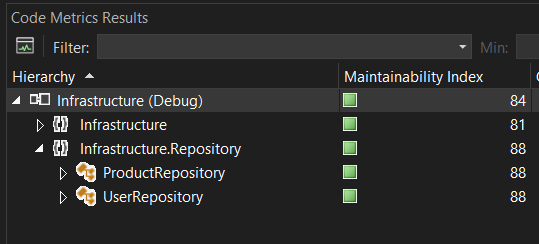
In layered architecture this is known as data access layer and in clean architecture it is implemented in infrastructure as per standard architecture principle guidelines but the core code requires to do database operation is same in the form of repositories apart from architecture structuring like how this layers are used by other layers and their dependency is the key difference between them.

Layered data access analysis



This layer in layered architecture contains domain as well repository for data access. So focusing on Data access part for this section of analysis that is Repository which in depth contains ProductRepository and UserRepository. Overall and individually we can see from above screenshot from artefact the Maintainability index count is **81** of repositories.

Clean data access analysis



In Clean layer same comes under Infrastructure repository and as seen from above clean infrastructure screen shot the repository section maintainability index is **88** as it is improvised from layered architecture which was **81**

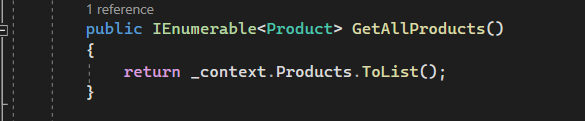
Now as we have compared all aspects of both the architecture directly or indirectly the same code operations it is performing. The maintainability index in Clean architecture has been improved in those layers here there are more dependency of other parts of the system or main parts of the solution. For demonstration purpose only two simple domain model and their CRUD operation is considered but in real world as system keeps on growing this maintainability index difference count will grow. So in small system it may not be make a huge difference in terms of maintainability but in complex application of real world it clearly leaves a remarkable impact.

**Abstraction**

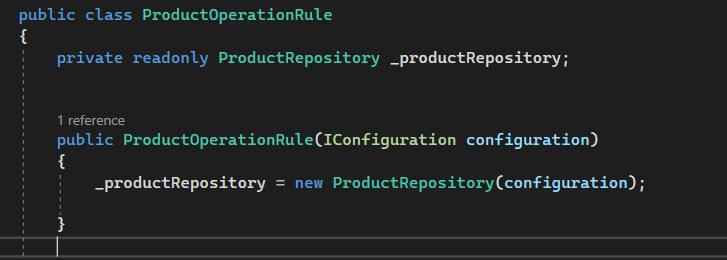
As a good software practice module should not be tightly coupled to each other. In case low module changes can affect other high level module that is dependent on low level module. For example, in our case application layer contain business logic (high level module) is dependent on data access or infrastructure layer (low level module) to perform its operation. **(Richard Mordinyi, Eva Kühn, Alexander Schatten 2010)**

Clean Architecture follows DIP (dependency inversion principle) that helps software easier to maintain and has multiple benefit’s. To demonstrate this low level code like data access and infrastructure layer is modified and instead repositories using ORM like entity framework now changed it into use ADO.net for database related operations. This repository is used in high level module in application layers to perform business logic. Now analysing it how it affects in layered and clean architecture.

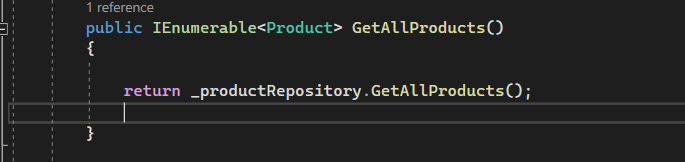
**Layered Architecture:** Product repository has **GetAllProducts()** that uses entity framework to fetch all products from product table and return it to application layer where it is been called.



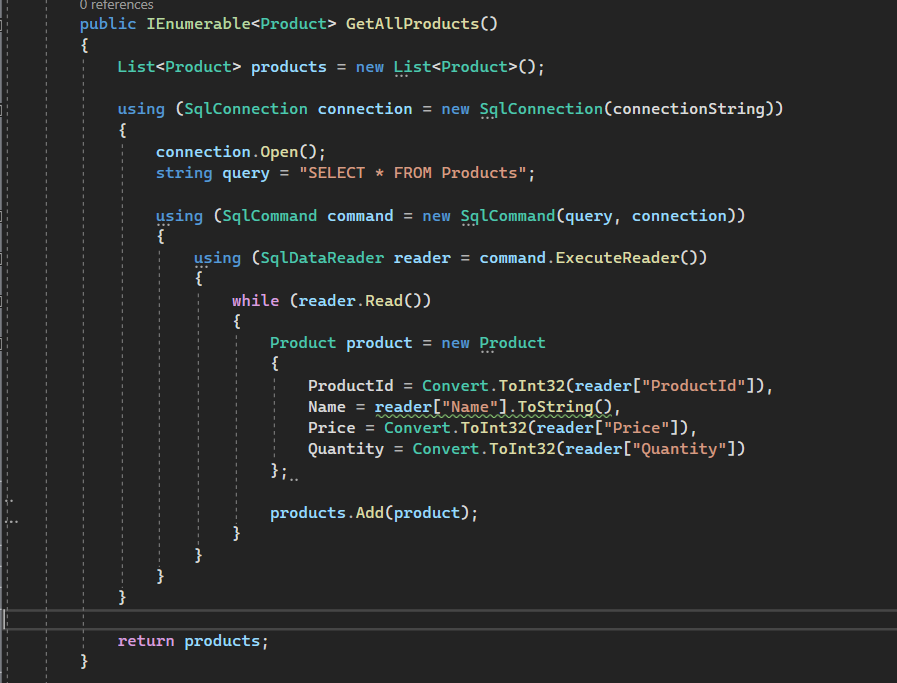
Instantiated product repository in Application layer to call **GetAllProducts()** function from repository



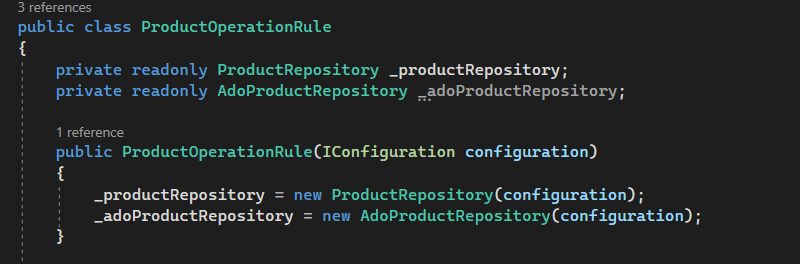
Calling Repository function in business logic function.



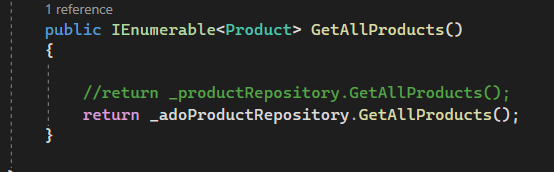
Now as per business requirement repository changes and need to use ADO.net instead of entity framework. A separate class repository is created and added same operation with Ado.net



Changing technology and its implementation needs to be used in application layer with below changes as per conventional layered architecture. Instantiated ado.net repository in application layer



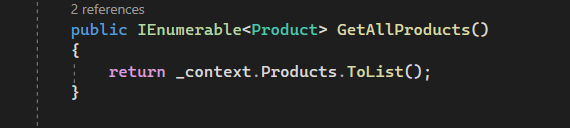
Changing in GetAllProducts() application layer function from entity frame work repository to ADO.net repository as below



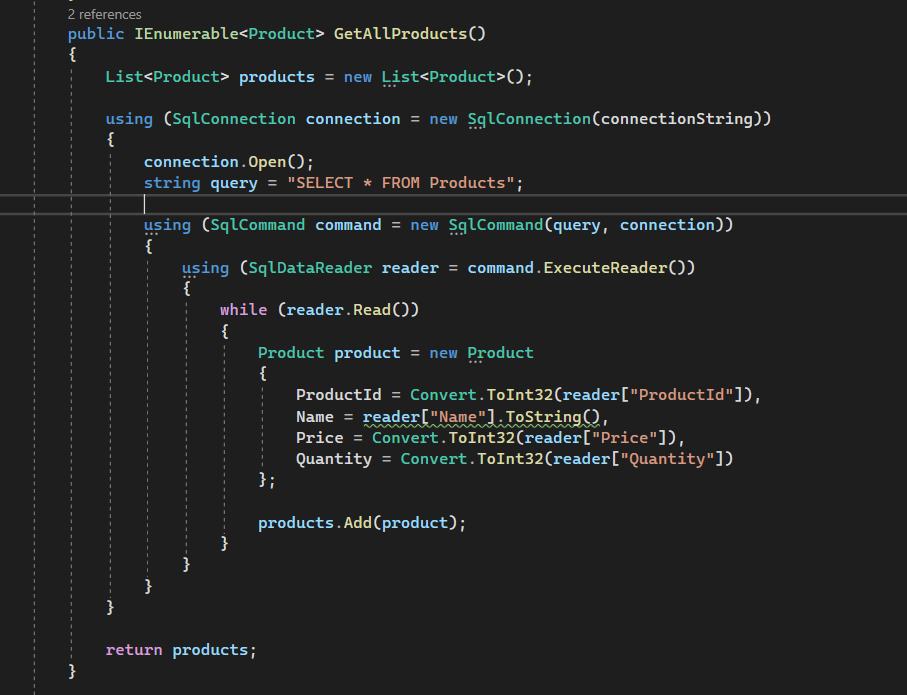
As observed from above screen shot’s changes in low level module that is data access layer affects application layer. A tight dependency was observed and multiple changes needs to be made in application layer to achieve this operation. Which basically decreases maintainability index. This is the simplest demonstration for understanding but in real world application it can be very difficult and time consuming.

**Clean Architecture:** Similarly, with same requirement change ADO.net repository change is required here as well instead of entity framework.

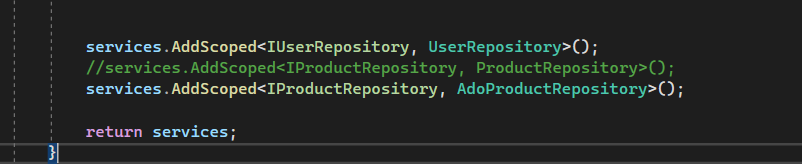
Entity framework repository



Now instead of this entity frame work repository ADO.net repository is implemented.



Now unlike instantiating ado.net repository in application business layer to use it we just need to inject it in using dependency injection principle. So there will be no code change in application layer.



In above screen shot it is basically used to configure dependency between modules using interface abstraction. So the commented line is using entity framework which was been called in application layer. Which then after comment was replaced by ADO.net repository on just below line. That’s the only place where object needs to be injected and no code changes required in application layer as it is dependent on abstraction instead of concrete class, no tight dependency. This makes code cleaner and more easy to maintain hence increase in maintainability index. This is simple demonstration but as application grows and becomes complex this approach helps make life of software developer and architect easy.

As clean architecture strictly follows dependency inversion principle **(Robert C. martin 2017)** and as per our experiment following conclusion was observed.

Decoupling High-Level and Low-Level Modules:

Without DIP: High-level modules (like business logic) in a system without DIP frequently rely directly on low-level modules (like infrastructure or data access). Because of the tight connection that results, it is difficult to modify or expand the system.

With DIP: DIP promotes the definition of contracts through the use of abstractions, such as abstract classes or interfaces. These abstractions are implemented by low-level modules, and high-level modules rely on them. As a result, the high-level and low-level components are separated, increasing the system's modularity.

Flexibility and Maintainability:

Without DIP: In a rigid design, minor adjustments to low-level components (such as databases or external APIs) may have a ripple effect on higher-level modules, resulting in significant changes.

With DIP: Low-level information can be isolated within interface implementations by depending on abstractions. High-level modules don't change as long as the contracts that the abstractions express don't change. This improves the flexibility and maintainability of the system.

Dependency Injection (DI)

Without DIP: Dependencies are frequently hard-coded in non-DIP compliant systems, making it challenging to switch out implementations without changing the dependant code.

With DIP: Dependency Injection, in which dependencies are injected into components, is encouraged by DIP. This increases the system's flexibility by enabling the simple replacement of implementations without altering the dependant classes.

Adherence to Clean Architecture Principles:

Without DIP: Systems that do not follow DIP typically have less clean architecture because of the interconnection between high-level and low-level components.

With DIP: Clean architecture places an emphasis on breaking down issues into discrete layers. By encouraging the separation of concerns, DIP adheres to the principles of clean architecture by simplifying the enforcement of layer boundaries.

Ease of Extension:

Without DIP: It might be difficult to add new features or expand functionality without DIP, particularly if the components that are already in place are closely connected. As seen changing from entity framework to ADO.net for database related operations in layered architecture.

With DIP: DIP's open/closed principle enables capability expansion without requiring changes to already-written code. Following pre-existing abstractions allows for the introduction of new implementations. As seen in Clean architecture no code change was required in application layer when repository was changes from entity framework to ADO.net