**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.

**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.

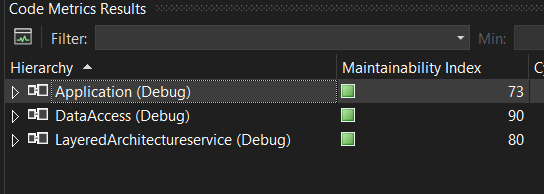
Using this tool helps to analysis results or to track even after changes are made to code.

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

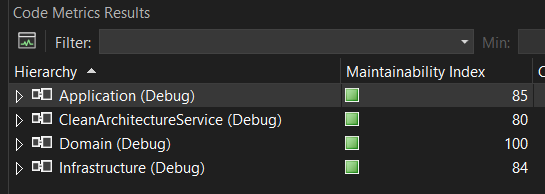
As a teaching tool, the Maintainability Index can assist developers in comprehending how their coding methods affect the software's maintainability.

**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.

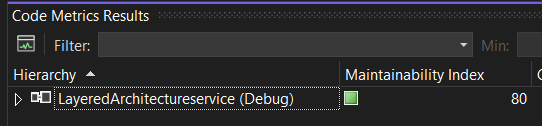
Both architectures are developed with same logical code to do same operations to get same output but using their own architecture principles. Hence all layers cannot be compared directly due to its own importance.

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

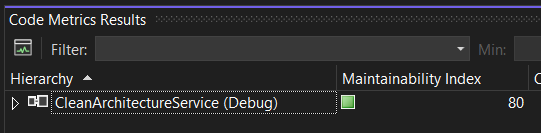
Presentation layer analysis:

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

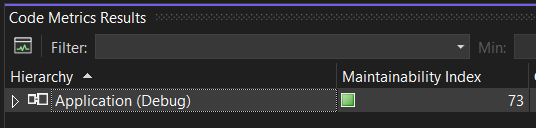


From above two presentation layer from two different architecture shows same maintainability index count that is 80.

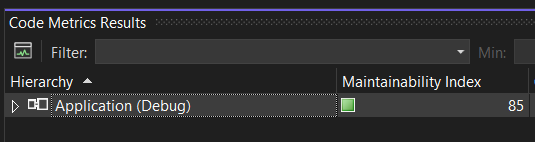
Application layer analysis:

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

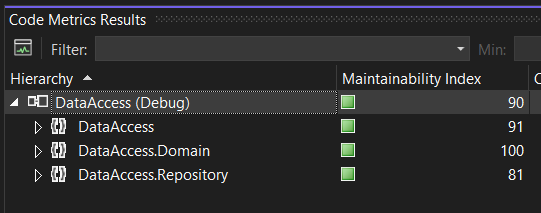


The Layered application layer maintainability index is 73 whereas the Clean architecture’s maintainability index is 85 which is a significance difference. Also as per the survey conducted with question asked that Which part of the code/module was mostly beneficial in terms of complexity and easy to change 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.

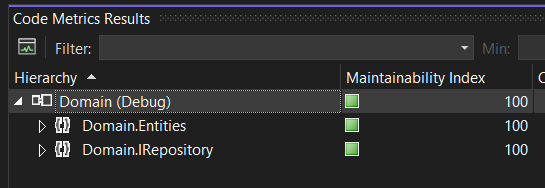
Domain Layer analysis:

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.



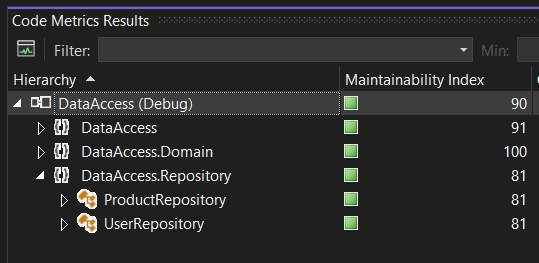
In above attachment we can see that domain is a part of data access layer in layered architecture as DataAccess.Domain but a separate layer in clean architecture as Domain as per their architecture principle guidelines. Also the maintainability index of domain area in both artefacts is 100 due to no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

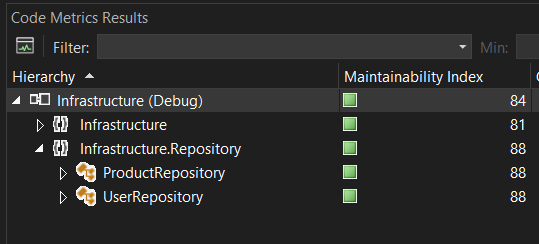
In layered architecture this is known as data access layer and in clean architecture it is implemented in infrastructure as per standard architecture principle guidelines but the core code requires to do database operation is same in the form of repositories apart from architecture structuring like how 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**

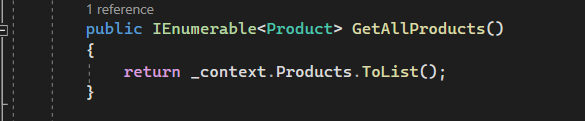
Now as we have compared all aspects of both the architecture directly or indirectly the same code operations it is performing. The maintainability index in Clean architecture has been improved in those layers here there are more dependency of other parts of the system or main parts of the solution. For 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**

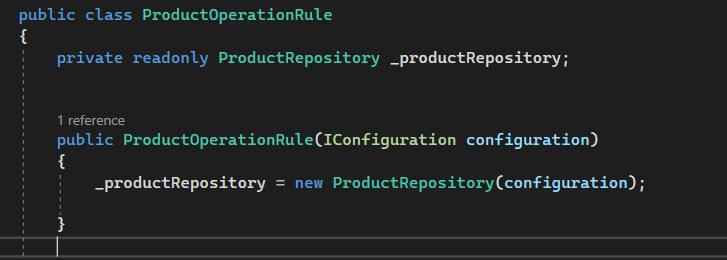
As a good software practice module should not be tightly coupled to each other. In case low module changes can affect other high level module that is dependent on low level module. For example, in our case application layer contain business logic (high level module) is dependent on data access or infrastructure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

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.

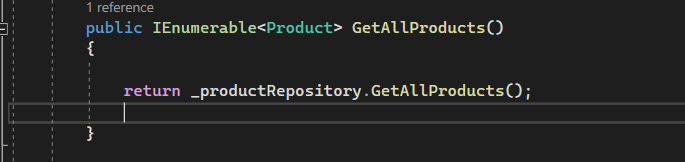
**Layered Architecture:** Product repository has **GetAllProducts()** that uses entity framework to fetch all products from product table and return it to application layer where it is been called.



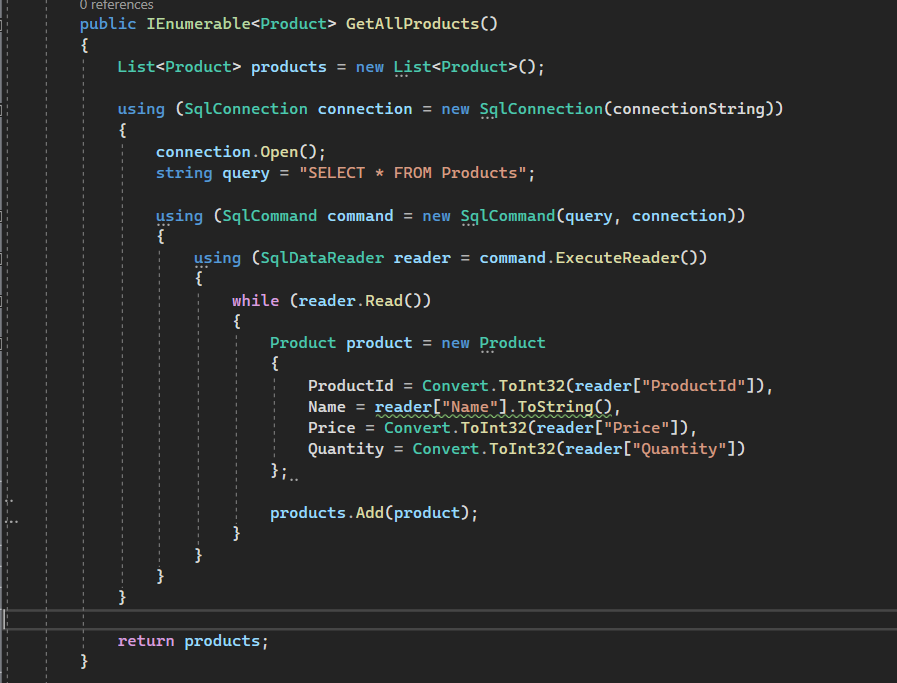
Instantiated product repository in Application layer to call **GetAllProducts()** function from repository



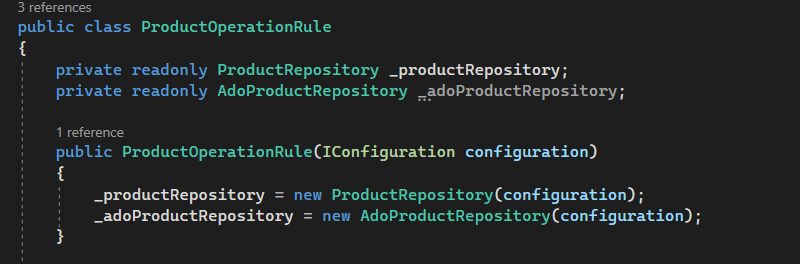
Calling Repository function in business logic function.



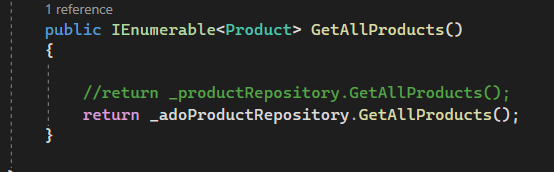
Now as per business requirement repository changes and need to use ADO.net instead of entity framework. A separate class repository is created and added same operation with Ado.net



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



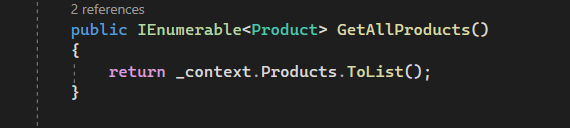
Changing in GetAllProducts() application layer function from entity frame work repository to ADO.net repository as below



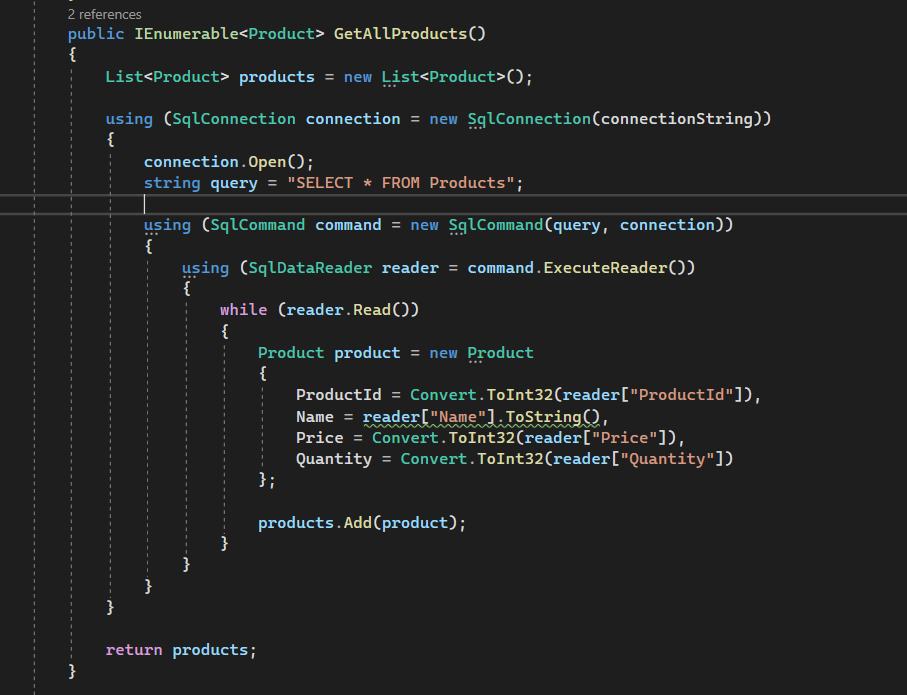
As observed from above screen shot’s changes in low level module that is data access layer affects application layer. A tight dependency was observed and multiple changes needs to be made in application layer to achieve this operation. Which basically decreases maintainability index. This is the simplest 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.

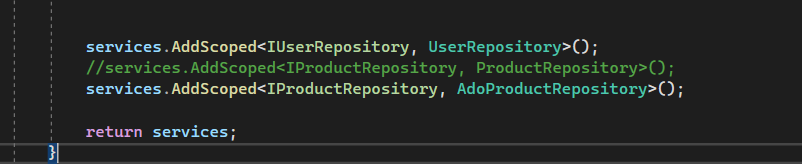
Entity framework repository



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 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