Clean Architecture Implementation Impacts on Maintainability Aspect for Backend System Code Base

1st Yosep Novento Nugroho
School of Computing
Telkom University
Bandung, Indonesia
yosepnoventon@student.telkomuniversity.ac.id

2nd Dana Sulistyo Kusumo
School of Computing
Telkom University
Bandung, Indonesia
danakusumo@telkomuniversity.ac.id

3rd Muhammad Johan Alibasa
School of Computing
Telkom University
Bandung, Indonesia
alibasa@telkomuniversity.ac.id

2022 10th International Conference on Information and Communication Technology (ICoICT) | 978-1-6654-8165-6/22/$31.00 ©2022 IEEE | DOI: 10.1109/ICOICT55009.2022.9914890

***Abstract*—The rapid growth of backend technology correlates positively with the number of problems that arise. One of the issues is the negligence of following an ideal backend system architecture. It causes various problems such as huge numbers of code duplication in many locations of the backend system codebase leading to a low maintainability level. This research aims to investigate the impacts of applying clean architecture on the backend codebase by creating several layers in the codebase according to the business logic hierarchy. To analyze the impacts, this research utilizes several metrics such as Cyclomatic Complexity, Weighted Method Count, Kan's Defects, Halstead's score, and Maintainability Index. The results showed that all maintainability metric scores improved after refactoring a selected backend codebase by applying the clean architecture principle. The improvements range from 21% to 61% for various maintainability metrics. This study validates that the implementation of a clean architecture in the backend codebase could increase the maintainability, reduce its complexity, and reduce developer effort to modify the codebase.**

Keywords—clean architecture, maintainability, backend

I. Research Background

The rapid growth of backend technology has been inseparable from various challenges in the last ten years [1]. The increasing demand and complexity of features in backend applications cause development to cut back quality, especially maintainability aspects during the development process [2]. It is indicated by a huge number of code duplications in the codebase [3]. This issue is because developers must read the entire code and must consider the changes that do not impact the existing flow which will result in prolonging the feature development process.

The previous problem can be solved by applying a concept called clean architecture [4] that helps increase the maintainability of the application because a system can have lower complexity compared to a system that does not adopt this architecture. In addition, it can also reduce the responsibilities of each function, decrease maintenance costs, and minimize the possibility of errors in the system. The problem of low maintainability and huge numbers of code duplication can be controlled by adopting the principle of clean architecture proposed by Uncle Bob because each layer is not responsible for the layer outside it [5]. The code duplication issue occurs often, and it leads to expensive costs for the maintainability of the codebase if the workflow needs to be modified [6].

This research proposes procedures on how to implement clean architecture principle on the backend application to improve the maintainability of the application, especially when additional features are added. The scope of this research is to apply clean architecture to improve any inefficient code writing such as code duplication or unnecessary code problems, by following our proposed systematic refactoring methods [7]. To validate our methods, we use several codes metrics related to maintainability, such as cyclomatic complexity, weighted methods score, Halstead's score, Kan's defect, and maintainability index. We measured these metric results from the codebase that has been refactored using our proposed refactoring methods that follow clean architecture principle and from the original codebase without refactoring.

II. Related Work

A. Clean Architecture

Clean architecture is a software design philosophy by separating design elements into each service level [8]. An important goal of clean architecture is to give developers a way to organize code in such a way that it encapsulates the business logic but remains separate from the delivery mechanism.

The main rule of clean architecture is that code dependencies can only move from the outer layer to the inner layer [4], [5]. The inner layer codes will have no information about the outer layer functions. Variables, functions, and classes (any entity) present in the outer layer cannot be enumerated at a deeper level. It is also recommended to keep the data format separated between service layers.

Clean architecture, created by Robert C. Martin or more familiarly called Uncle Bob, is promoted on his blog and in his book Clean Architecture: A Craftsman's Guide to Software Structure [5]. Like other software design philosophies, clean architecture seeks to provide a cost-effective methodology that facilitates the development of quality code that will perform better, be easier to modify, and have fewer dependencies [5]. Visually, the level of clean architecture is defined at an unspecified number of service layers (see Figure 1). The outer service layer is the lower layer mechanism, and the inner service layer contains policies and entities.

Applying clean architecture has several benefits, such as high testability and independence on several software aspects including framework, UI, database, and any external agency. These independences are obtained as each layer can only access its own layer and does not have control over the outside layers.

B. The Dependency Rule in Clean Architecture

The concentric circles represent different software areas. In general, the further we go, the higher the level of the software. The outer service layer is the mechanism of the

lower layer. The inner service layer is the thing that contains policies and entities [5][8].

The main rule that makes this architecture work is The Dependency Rules. This rule says that source code dependencies can only point inward [9]. No one in the inner circle can know anything about something in the outer circle. In particular, the name of something declared in the outer circle should not be mentioned with the code in the inner circle. It includes functions, classes, variables, or other software entities.

In the same way, the data format used in the outer circle should not be used by the inner circle, especially the format generated by the framework in the outer circle. Anything in the outer circle is not allowed to affect the inner circle.

- **Entities**: The entity encapsulates business rules throughout the company. An entity can be an object with methods, or it can be a collection of data structures and functions. It does not matter if the entity can be used by many different applications [10]. This is the layer closest to the database where this layer will not make objects affected by changes in the outer circle. There are no operational changes for a specific application that will affect the entity layer.

- **Use Cases**: The software at this layer contains application-specific business rules. It encapsulates and implements all system use cases. These use cases govern the flow of data to and from entities and direct those entities to use business rules to achieve the intended use [11]. This layer will not affect the entity layer. We also don't expect this layer to be affected by changes to externalities such as the database, UI, or other frameworks. This layer must be isolated from it. However, we can expect changes to the operation of the application to affect the use cases and therefore the software at this layer. If the use case details change, then some code in this layer will be affected.

- **Interface Adapters**: Software at this layer is a set of adapters that convert data from a format that is more suitable for use cases and entities, into a format that is more suitable for a particular external agency such as a database or the web [8]. It is this layer, for example, that completely contains the MVC architecture of the model, view, and controller all here. The model may be just a data structure passed from the controller to the use case, and then from the use case to the renderer and view.

Likewise, data is converted, at this level, from the form best suited for the entity and use case, to the form most suitable for the persistence framework being used, namely the database. None of the codes in this loop should know anything about the database. If the database is a SQL database, then all SQL should be restricted to this layer and to the part of this layer that deals with databases. At this layer as well, other adapters are required to convert data from external forms, such as external services, to internal forms that are used by use cases and entities.

- **Frameworks and Drivers**: The outermost layer usually consists of frameworks and tools such as databases, web frameworks, and so on. Usually, it is not necessary to write a lot of code at this layer other than pasting the code that communicates to the next layer into [5]. The web is all about details. Databases are the details. It is in this layer that all the details are to keep things that do the least amount of damage.

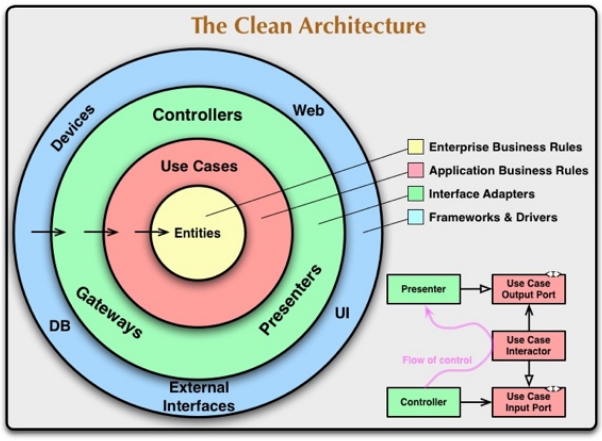


Fig. 1. Clean Architecture Concept

C. *Another Approach besides Clean Architecture*

Another architectural pattern approach could use a three-layered architecture. Broadly speaking, these two patterns are almost similar. For example, they both have business logic and entities, but in this pattern the entities are in the data access layer. However, the difference is in the treatment of the entity only. If in a clean architecture, the entity stands alone as a model that stores business logic and related data storage in the repository class. However, if the three-layered architecture entity interaction with the database is quite close, and the entity extends directly to the object relation mapping.

The three-layered architecture is divided into three parts [12]. The presentation layer is how the user interacts with the application. The business logic layer is responsible for all logic apps, where this layer interacts directly with the database through entities that extend the object relation mapping. The data access layer is where entities can directly interact with the database [12].

Three layered architectures can be used if the project that is made is not too large and complex, because the layer breakdown is not too much like a clean architecture. However, the function is difficult to test because all logic apps are included in the business logic layer.

D. *Maintainability*

Maintainability comes from the word maintain which means maintenance and ability which means ability. Maintainability is defined as the ability to maintain a system that has the possibility to be damaged to be returned to full working condition for a predetermined period [3].

Maintainability exists to overcome problems that arise when the system has been used by the user, where there is a system error that has not been handled in the latest system release [7]. So there needs to be further feature development from the development team, so that the system will fix errors in the previous system.

E. *The purpose of maintainability*

Ensuring maintainability serves various purposes, as follows:

- Enables production quality and satisfaction to be achieved through proper regulation, support and operation of equipment.

- Take advantage of the useful life of the system itself.

- Keeping the system secure prevents the development of security vulnerabilities.
- Minimizes total production costs that can be related to maintenance and repairs.
- Maximize production from existing system resources.
- Prepare the development team, facilities, and methods to be able to carry out maintenance activities.

F. Maintainability Criteria

The maintainability of software is divided into several criteria that can be measured [3]. Consistency in the code is essential because it can determine whether the source code versions between before and after refactoring accomplish the same output [13]. Conciseness, code development in a succinct manner. Simplicity and low complexity will be followed by simplicity in code. Modularity is the degree to which a computer system or program consists of separate components in such a way that changes in one component have minimal impact on other components. Self-documentation, code that documents itself as if it were written using a human-readable name, usually consists of phrases in human language that reflect the meaning of the symbol.

The measured maintainability criteria are described as follows:

1) Cyclomatic Complexity

Cyclomatic complexity is a measure of the complexity of the control structure of a function or procedure. This method is used to measure the simplicity and conciseness of the maintainability aspect.

2) Weighted Method Count

Weighted method count is the number of methods parameterized by an algorithm to calculate the weight of a method or function. This method is used to measure modularity from the maintainability aspect.

3) Kan's Defects

Kan's defects are the sum of the overall calculations of the while, switch, and if algorithms in a codebase. This method is used to measure simplicity and modularity in the maintainability aspect.

4) Maintainability Index

The maintainability index is a software metric that measures how maintainable (easy to make changes) the codebase is. The maintainability index itself is calculated as a factor formula consisting of the number of lines of code, cyclomatic complexity, and Halstead volume. This method is used to measure simplicity, modularity, conciseness, and self-documentation in maintainability aspects.

III. Methodology

A. Research Methods

The research method used is the creation of a backend codebase system using the concept of a clean architecture. Clean architecture itself is an architectural concept to separate each service element into their respective responsibilities. This will increase the maintainability of the backend codebase. The research method will look like in the following flow chart:

In terms of maintainability, a comparison will also be made between before and after the implementation of the clean architecture concept to the codebase. Code metrics calculated in the codebase include cyclomatic complexity, weighted method count, Kan's defects, and maintainability index. So that we can take the results of the effectiveness and validity of this clean architecture concept on its maintainability, compared to without using this concept.

a) Study of Literature

A review of the research that has been done previously is carried out and summarizes the facts and theories needed for this research. This is done by reading related articles and journals. At this stage, we also analyze the problem and provide reasons why the problems need to be solved.

b) Looking for a Codebase

A review of the codebase that is searched through the GitHub platform is carried out. Where the rule for choosing this codebase is to look at the tech stack of the program, which is focused on the Laravel framework and is a backend project codebase.

c) Check Codebase Maintainability with PHP-Metrics

The codebase maintainability needs to be checked. This check itself is done using a tool called PHP-Metrics to find out values such as cyclomatic complexity, maintainability index, and so on. This is done to retrieve the data and become a comparison after refactoring the codebase. The results of checking this step are reported in section IV (Table I).

d) Creating Unit Testing for Each Route API

Unit testing is added to each route URL in the codebase. Unit testing is a test of individual units or groups of related units [14]. The purpose of unit testing itself is to validate the program from input and output to produce predetermined results. This stage is important because developers can be more confident to make changes to a function.

e) Create a Refactor Strategy for Each Function Using a Clean Architecture

A strategy is developed to refactor using the concept of clean architecture, such as mapping each responsibility function to be made from the function to be refactored. A strategy will be drawn up to break down a function that already has a big responsibility. By carrying out the preparation plan at the beginning, it will be able to increase the speed and efficiency in doing the refactor at a later stage, because the developer already has a clear direction on what will be refactored using the rules of the clean architecture. One example of developing a refactor strategy in this codebase is the OrderController@store function with the initial structure as shown in Figure 2.

Figure 2 shows that the function has a request that is sent to the store function in the order controller, then the function has logic for many entities directly. The entities are order entity, address entity, product entity, and order item entity. With the background functions as above, the code can be broken down into several functions that can be reusable for use by other functions using the clean architecture rules, and the results are as shown in Figure 3.

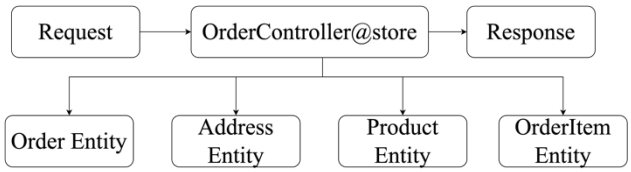


Fig. 2. Function Flowchart Before Implementation of Clean Architecture

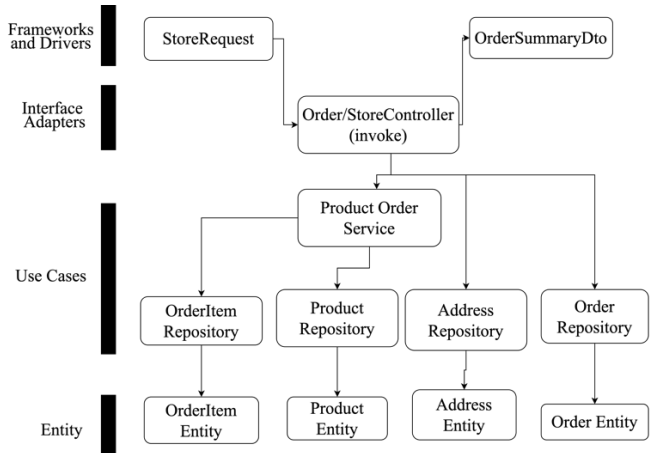


Fig. 3. Function Flowchart After Implementation of Clean Architecture

f) Refactor Codebase

The author refactors the code based on the preparation of the refactor strategy in the previous stage. The principle of code refactoring itself can be likened to tidying up a previously messy room, but still maintaining the same functionality. The author uses the principle of clean code in refactoring the codebase and using a clean architecture for layer breakdown.

g) Running Existing Unit Testing

The refactors carried out on each function have been completed, to give confidence to the developer, it is necessary to run unit testing with existing unit testing that the developer has done in several previous stages.

The purpose of this stage is to ensure that the refactor that the author has done is in accordance with the initial requirements that have been determined. If any function is still not passed when the unit test is run, it is necessary to re-check the refactored function. If the unit test has been passed completely on all routes, then you can move on to the next step.

h) *Check Maintainability After Refactor*

The codebase maintainability checks that have been refactored using the clean architecture rules. This check is done using the same tools as before, namely PHP-Metrics. And the data from these tools need to be stored for analysis in the next section. The results of checking this step are reported in section IV, in Table I.

i) *Analysis of Test Results*

The analysis is carried out based on the test results from the stage before the codebase refactors and after codebase was implemented a clean architecture. A comparison will be made between the two codebase maintainability measurements.

j) *Conclusion*

The author will provide conclusions from the results of experiments on the implementation of the concept of clean architecture in the codebase on aspects of codebase maintainability.

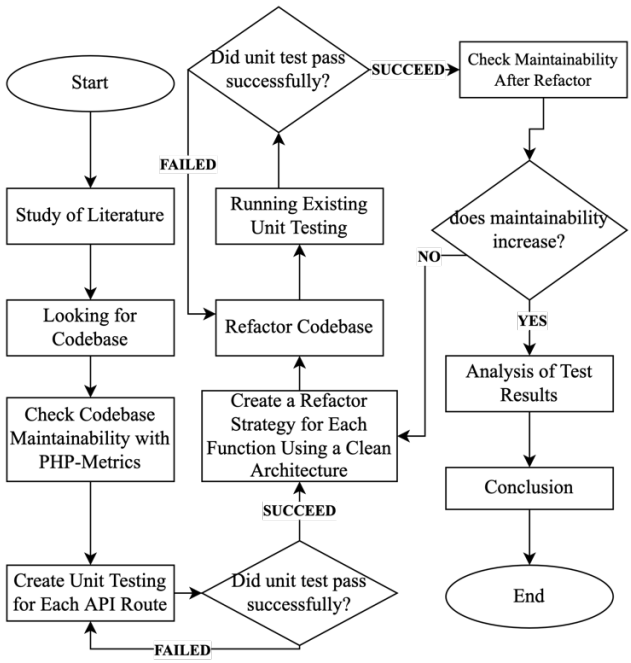


Fig. 4. Research Methodology Flowchart

B. Data collection

The data from this research is a codebase of a backend application and taken from the GitHub platform under a project named ApiEcommerceLaravel[1], which was created by the developer with the username MelarDev. The dataset is selected randomly by looking at provisions such as a repository that focuses on backend implementation and is developed with the Laravel framework. This repository is a form of e-commerce API system application. This application is made using the Laravel framework. This repository itself was created on February 17, 2019. The application is obtained from the GitHub platform, where GitHub itself is a special platform for application developers to build brands, a place for project management, and social media for the developers themselves.

The criteria for collecting this data are based on which are explained as follows:

- **Framework**: In accordance with the limitations of the problem contained in section one, it is mentioned using Laravel which is a framework from the PHP programming language.
- **Open source**: An open-source project is a codebase that is open to anyone and allowed to be modified by other developers, without having to ask permission first. This project was taken randomly from two codebases in different repositories and different developers, so that the maintainability measurement is not only based on one project.

The data taken next is the data from the maintainability measurement in the initial codebase using the php-metric

[1] https://github.com/melardev/ApiEcommerceLaravel

package tools. After knowing the results of the initial measurements, the author will apply the concept of clean architecture to the codebase. And followed by a re-measurement of its maintainability.

IV. Result and Discussion

A. Analysis of Test Results

The results of the refactor are at our GitHub repository[2]. This section will explain the results of generating Laravel e-commerce API codebase metrics using the php-metrics tool package before and after refactoring. The folder that generates maintainability is the app folder, as in that folder, the model-view-controller of the codebase is located.

Based on Table 1, the average maintainability of the codebase before refactoring is also quite high. The average weighted method counts per class at 3.3. The average cyclomatic complexity per class is 2.01. Average relative system complexity at 44.99. Average Halstead at 0.04. Average Kan's Defect at 0.24. Average Maintainability Index at 90.71.

The average maintainability of the codebase after refactoring has good maintainability. The average weighted method count per class is 2.5. The average cyclomatic complexity per class is 1.57. Relative system complexity at 17.18. Halstead at 0.02. Kan's Defect is 0.21. Average Maintainability Index at 107.94.

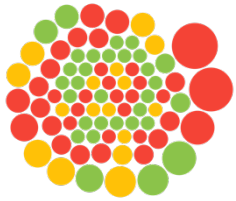
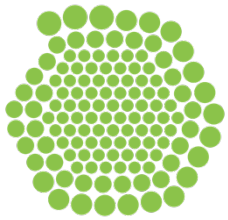
TABLE I. Result of Maintainability Index on Percentage of Difference

Complexity	*Before*	*After*	*Percentage of Difference*
Average Cyclomatic complexity by class	2.01	1.57	-21.8905 %
Average Weighted method count by class	3.3	2.5	-24.2424 %
Average Relative system complexity	44.99	17.18	-61.8137 %
Average Difficulty	2.54	2.17	-14.5669 %
Average Halstead by class	0.04	0.02	-50 %
Average defects by class (Kan's Defect)	0.24	0.21	-12.5 %
Average Maintainability Index by class	90.71	107.94	18 %

Table II shows a representation that describes the results of the maintainability index and cyclomatic complexity of each function in the project folder. The color of the circle is a representation of the maintainability index, if green indicates that the maintainability index is high, and if the color of the circle is red it indicates that the maintainability is low. The color in the circle indicates the level of cyclomatic complexity, the larger the circle, the higher the cyclomatic complexity, and the smaller the circle, the lower the cyclomatic complexity.

By analyzing the maintainability index before the refactoring, there are still many functions that have low maintainability, where there are red and yellow colors. In addition, the size of the circle with one another has a wide gap which indicates that there are still functions that have high cyclomatic complexity. However, in the data related to the maintainability index after the refactor, all functions have a green color, which indicates that the maintainability is high. In addition, the sizes of the circles have adjacent gaps, which indicates that the cyclomatic complexity is low.

TABLE II. Result of Maintainability Index on Figure

Representation of Maintainability Index & Cyclomatic Complexity	
Before	
After	

B. Disadvantage of Clean Architecture

Clean architecture has several disadvantages, leading to other approaches. Clean Architecture would be redundant as a codebase for CRUD-related functions, but it is great for long-term, microservices, and complex projects. There are many interfaces in the codebase, so it is possible to make the application heavy on memory usage if we use it on a monolithic codebase.

V. Conclusion

Based on the results and discussions, we obtain conclusions that can be drawn from this research. The results of this study prove that the implementation of a clean architecture in the codebase can be used as a reference for backend application development because it can increase maintainability, reduce complexity, and reduce developer efforts to improvise on the codebase because the code is clean, tidy, and eliminates code duplication. The average decrease in complexity metrics in the codebase is 30% and the maintainability index increases by 18%.

Architecture is important for any application, where the team creates several programs, many implementations will change based on the new features that need to be developed. If the developer does not set the architecture that is applied to a project, the developer will have different unique ways of implementation [5]. This will affect the scalability and maintainability of the project. Where we will have a lot of side effects from simple changes or new features. Therefore, uniting concepts in a project will make it easier for developers to have a common vision. The concept of clean architecture has a good consistency with respect to the scalability of the application. In addition, the important thing that will be

[2] https://github.com/ventodeco/clean-architecture-codebase

affected by clean architecture is maintainability. Maintainability is an important attribute of clean architecture. It is assumed that the requirements change due to the environment such as changes in some controls or bug fixes or other changes.

REFERENCES

[1] Barbosa, F. and Aguiar, A., 2013. Removing Code Duplication with Roles. 2013 IEEE 12th International Conference on Intelligent Software Methodologies, Tools and Techniques (SoMeT),.

[2] Koller, H., 2016. Effects of Clean Code on Understandability: An Experiment and Analysis. Oslo, Norway: Department of Informatics University of Oslo.

[3] Fanta, R. and Rajlich, V., 1999. Removing clones from the code. Journal of Software Maintenance: Research and Practice, 11(4), pp.223-243.

[4] Martin, R., 2017. Clean Architecture: A Craftsman's Guide To Software Structure And Design. [S.l.]: Prentice Hall.

[5] Aguilar, P. and Figueira, L., 2020. Clean Architecture is not only about business logic. In: I Workshop de Tecnologia da Fatec Ribeirão Preto. [online] São Paulo.

[6] Fowler, M., 2018. Refactoring: Improving the Design of Existing Code. Addison-Wesley Professional.

[7] Gill, G. and Kemerer, C., 1991. Cyclomatic complexity density and software maintenance productivity. IEEE Transactions on Software Engineering, 17(12), pp.1284-1288.

[8] Ivanics, P., 2017. An Introduction to Clean Software Architecture. University of Helsinki.

[9] Feathers, M., 2004. Working Effectively with Legacy Code. 1st ed. Pearson.

[10] McConnell, S., 1993. Code Complete: A Practical Handbook of Software Construction. 1st ed. Microsoft Press.

[11] Gamma, E., Helm, R., Johnson, R. and Vlissides, J., 1995. Design Patterns: Elements of Reusable Object-Oriented Software. Pearson Deutschland GmbH.

[12] Casalino, G., Turetta, A. and Simetti, E., 2009. A three-layered architecture for real time path planning and obstacle avoidance for surveillance USVs operating in harbour fields. OCEANS 2009-EUROPE,.

[13] Moser, R., Abrahamsson, P., Pedrycz, W., Sillitti, A. and Succi, G., 2008. A Case Study on the Impact of Refactoring on Quality and Productivity in an Agile Team. Balancing Agility and Formalism in Software Engineering, pp.252-266.

[14] Runeson, P., 2006. A survey of unit testing practices. IEEE Software, 23(4), pp.22-29.