An Introduction to Clean Software Architecture

Péter Ivanics
Department of Computer Science
University of Helsinki
Email: peter.ivanics@helsinki.fi
http://pivanics.users.cs.helsinki.fi/portfolio/

***Abstract*—The construction and the long-term maintenance of robust software architecture is a demanding task. Software architects often face the problem to ensure the extensibility of systems so that new components can be added and existing ones can be removed or replaced easily. Linearly with software growth, these aspects of software engineering are getting more and more difficult because often times the separation of concerns and the decoupling of components are not considered in the beginning of the development.**

The concept of the Clean Software Architecture was introduced recently by professional craftsmen in order to provide principles on how to achieve long-term robustness in software architecture engineering. This approach builds on top of the clean separation of concerns, finding the right abstractions on every level of the software projects, the minimal number of internal dependencies and finding the right flow of control.

The objective of this study is to review relevant professional literature and summarize the most important aspects of the Clean Software Architecture. The paper describes the benefits of this approach compared to other practices. It is explained how this approach facilitates and why is it well-suited for Object Oriented principles and software projects of the present time.

Results show that understanding the key concepts, such as The Dependency Rule and the core four layers of the Clean Architecture help developers to maintain robustness in their software. Ultimately, the concepts explained by this approach are independent from technologies and therefore easy to apply in many fields of our industry.

***Index Terms*—Software architecture, Clean Code, Software development, Refactoring, Technical dept**

I. Introduction

The construction and the maintenance of software is a difficult task from many aspects. Software development is a creative act of action where developers build intangible means from nothing [2]. Linearly with the growth of the code base of a software project, its complexity and cost of change increases [5] [3], which makes developers' job even more difficult.

Software assets can be crucial for operations in industrial, business and research fields [3] [2]. In the present time, more and more companies are established around providing software services to their customers [6], and therefore software components can become business-critical. For this reason, the process of software development requires dedicated professionals with a lot of discipline, humility, dedication and commitment to their job [3].

Despite all the previously established standards and practices, developers often face troubles of conforming to the standards and understanding code written by others or even themselves [2]. Keeping the code base maintainable, easy to understand and open for extensions is often a challenging task [2], but yet essential aspect to avoid paying the technical dept. For this reason, the concept of Clean Code [3] was established which further raised the attention towards maintainable software in the industry. Similarly, agile methods and principles became popular in order to facilitate developer productivity.

Clean Code is a concept introduced by Robert C. Martin [3] [2]. This approach to software development aims for the delivery of elegant, easy to read, fully tested code base, in order to create robust, bug-free applications [3]. Additionally, the regular need for refactoring is emphasied as increases quality of software [8], and therefore developers should not be afraid of performing this activity [2].

The Clean Code principles are getting more and more popular among wide range of software developers in the industry. On top of that, even universities consider to change their approach how to teach software development by introducing testing and maintainable code first to the students [9]. The reasons for its popularity are understandable: the principles are simple and clearly expressed, everyone who ever dealt with software development can relate to them and utilize the acquired knowledge right away.

Software architectures lay down the basis and establish the frame for quality software. During the last decades, many different approaches and design patterns were developed and proposed by professionals to facilitate robustness of programs through design patterns [5] [4] [7], which are widely used and appreciated by others worldwide. In order to ensure maintainable and robust software systems a loosely coupled, flexible and extensible frame should be provided as the basis, which is ready to accept continuously changing and new requirements. For this reason, an approach towards the Clean Architecture is emerging [1]. The Clean Architecture considers building scaleable software architecture by building on top of Clean Code principles.

This paper aims to seek answers to the following research questions by discussing relevant literature in the topic:

- RQ1: why is it important to consider Clean Code principles in relation to software architecture design?
- RQ2: what are the main aspects of the Clean Code principles in terms of software architecture?
- RQ3: what are the benefits of approaching software architecture design using Clean Code principles?

The rest of this paper is structured, as follows. The next section explains why a clean design for software architecture

is needed and who may benefit approaching software development from this scope. Section III lists and discusses the five main principles and benefits of using a Clean Architecture for any software project. Finally, the last section concludes the findings of this research and establishes the directions for further studies in the topic.

II. The Need for a Clean Design

Software development is collaborative work. More and more developers work in teams of various sizes on multiple technologies to develop their software day-by-day. Inevitably, components are developed simultaneously by multiple coders and are getting bigger by time. As time passes, developers tend to have difficulties understanding the code they have written because they do not remember what the intention behind the lines were [2]. Similarly, understanding decisions made and code written by fellow colleagues is even more difficult and challenging on every level of abstraction [2]. As the code base grows, decisions that were previously made may need to be reconsidered and code may need to be refactored to enhance its reliability and to avoid technical dept.

For this reason, considering the intent behind a decision is key even on the deepest level of components, for instance classes, objects, functions or even variable names. The usage of intention-revealing names on all parts of software projects greatly enhance the readability of their code [3]. This enables developers to read and understand others' code more efficiently and work on code written by others with more confidence.

Demonstrating the intent behind design decisions is a key characteristic of design based on Clean Code principles. Intent-driven architecture helps developers to understand what the program code does and how it is structured. This is particularly important while debugging or refactoring detailed parts of the code. However, it is not limited to low-level components of the software: applications should express the purpose of their components even on the highest level [1].

Naturally, a complex system is composed of multiple components on different levels. In order to understand the "big picture" of such system, one may want to look at it on the top level from different angles. If an "outsider" (i.e. a new developer or system architect to the software) looks at a complex system, they may find it inconvenient, hard to understand or confusing at first, if the components, connection points or relationships are not named properly.

For example, many frameworks and developers adopt the Model-View-Controller (MVC) design pattern. The MVC guidelines are particularly easy to understand first, however they tell how the pieces are made rather than telling what they do. In other words, the high level components do not express their role and purpose, but their internal structure. As a result, large-scale systems that adopt MVC may be hard to understand on a high level.

As soon as components hold the property of telling their role at once in the ecosystem, they become easier to understand [3]. Intent-revealing components typically easy to adapt to the other key principles of clean design that are explained in the following section.

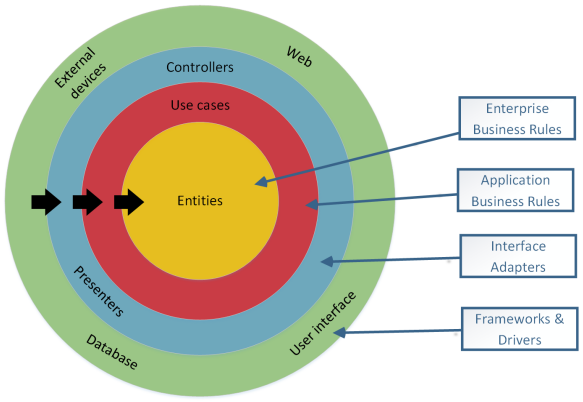

Fig. 1. The layers of the Clean Architecture and their inward-pointing dependencies (adapted from [1]).

III. Principles

Communicating the intent behind the components is a good start for a Clean Software Architecture. Nevertheless, there are some other principles which help developers to avoid tight coupling and complex connections between the entities. The aim should be an approach, where each component is standalone and is handled as plugin to other parts of the application [3] [1]. In other words, components should be independent, which leads to the first principle of Clean Architecture, namely, the onion-like separation of architectural layers [1].

A. Architectural Layers

Many of the commonly used system-level design patterns may be easy to understand and implement, but are also hard to scale up to bigger systems. For example, the traditional Model-View-Controller (MVC) approach defines three main layers, namely user interface, business logic and database for the software. MVC is widely used in the industry due to its simplicity, however it carries the risk of tight coupling between the layers of the architecture as well as the difficult separation of concerns [4].

Tight coupling of the layers immediately leads to a hardly-scalable architecture unless the software is relatively small [4]. As time passes, components grow bigger, at the same time technology develops and requirements change. Tightly coupled systems have hard time to follow-up on such changes, which is inevitable in case of business-critical systems [4]. Therefore, a flexible and scalable approach to architectural layers is needed, which is flexible enough to support rapid system growth.

The solution to achieve a flexible, largely scalable architecture lies within the definition of the correct layers and the simple dependencies between them [1] [4]. For instance, the Onion Architecture [4] suggests to define the layers from inside-out, from core entities to to infrastructure as shown on Fig. 1.

The centermost layer, the Enterprise Business Rules define the domain specific business-entities and the domain-specific business logic. The responsibility of this layer is strictly limited to define entities and specific logic that are specific for the application's domain. More general functionality, that can be reused in other applications is delegated to the layers further away from the center. The Application Business Rules layer around the models is responsible to implement the business logic upon the business objects. As R. C. Martin points out [1]: *"The outer circles are mechanisms. The inner circles are policies"*.

Interface Adapters act as facilitators between the business logic and the events happening in the user interface, external input sources and the database [4] [1]. This means that the business logic is defined closely on top of the entities and the external components, such as user interface or database control the entities through the rules indirectly. This greatly helps developers to distinguish the level of abstraction for every layer.

Decoupling of different roles and the separation of responsibilities are key aspects in this design. Every layer has one task to cover and each holds the property of being possible to remove and replace by another component that has similar behavior but different internal logic. This kind of isolation embraces layers to be plugins to eachother and therefore easy to modify, extend or even replace during future development.

The four layers shown on Fig. 1 create the core of a Clean Architecture. Nevertheless, as systems grow bigger, they may divide layers and separate them even further. For example, one may divide entity model objects from their business logic to have an even more clear distinction of roles among the components.

B. Separation of concerns

A well-designed system-wide architecture also aims to separate concerns. By separating responsibilities in the application well, the code not only gets easier to understand but also more flexible, loosely coupled and easier to test [2] [1] [4]. This helps to tell the responsibility and place the components by their names to the correct layers described in the previous section.

Consequently, to allow understanding the software architecture on its highest level, clear decisions should be made on the responsibility of the components. Once the components are chosen, their name should tell right away the intent they were made for. Applying the same approach on every level of abstraction throughout the application is crucial as it keeps components minimal, and lightweight.

C. The Dependency Rule

The approach to a layered architecture also requires the definition of carefully designed dependencies. Identifying the correct dependencies and keeping them minimal is key to a loosely coupled software [3]. The Clean Architecture explained in the previous sections is designed in a way, that such dependencies are considered at the first place.

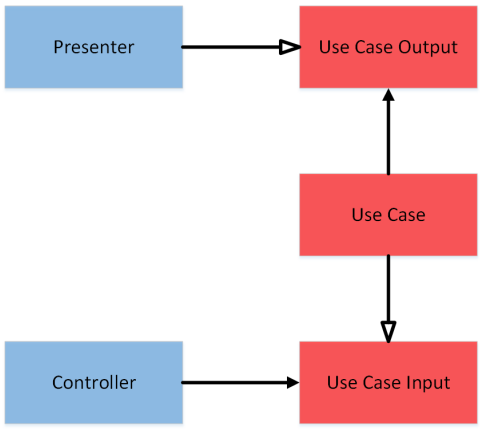

Fig. 2. The flow of control between neighbouring layers in the Clean Architecture [1]. The filled arrows represent data submission to, the hollow arrows data retrieval from the pointed component.

The Dependency Rules dictates that the inner layers of the architecture do not know anything about the layers that are outer than them in the architecture [1] [4]. In other words, all dependencies point towards the middle of the layered structure, towards the business logic and the entities.

Consequently, individual layers do not know about in which context they are being used, but they know the insides of the encapsulated entities underneath their level. The deeper we go into the architecture, the less dependencies exist and less complications are experienced.

This approach allows the layers to be plugins to eachother: in case one of the mechanisms (for instance the user interface or database) change, it is sufficient to replace its layer and conform it to the interfaces [1] [4]. As soon as correct interfaces are defined the edges of the layers, such mechanisms become immediately external. This kind of decoupling greatly lowers the maintenance cost for the lifetime of an application [4]. On top of that, the software becomes more flexible and adapts to new or changing requirements easier.

D. Flow of control

Crossing the boundaries from inside out may be required in certain situations. For example, if an action initiated by a Controller object has an impact on the business logic and thus on the state of the entities, the user interface or the database may need to be updated once the changes are committed. For this reason, the layers should be designed in a manner that they take inputs on one interface and produce output on another [1]. The mechanism which is to handle the output can respond to the received data and perform the necessary actions accordingly.

As shown on Fig. 2, the control flow goes through interfaces when the boundaries are crossed. The layer which is closer towards the center of the architecture receives data through its input interface, processes it and passes it forward on its output interface. More specifically, a Controller object may trigger an event to process some data in a Use Case, which data is then

forwarded through the output interface and presented in the necessary component.

This way each layer encapsulates its own logic. The responsibilities remain clearly separated and the Dependecy Rule is not violated, because layers do not know anything of the insides of the others. The inner layer passes data back to an outer layer through an interface, that is implemented in the receiver class. This is called the Dependency Inversion Principle [1] [4].

For simplicity, it is suggested that the interfaces deal with simple data types [1]. For the sake of not violating the Dependency Rule, the inner circles should expose their objects and data structures when passing data to an interface. Therefore, it is suggested to define the interfaces such that only simple data structures cross the boundaries [1].

E. Testability

The layered structure and inward-pointing dependencies have other benefits than flexibility and easy maintenance. Due to the fact that every layer has interfaces (some may call them input and output ports), they become easy to test [1].

Automated testing is another great tool which can contribute to the robustness of the software, because it allows developers to have confidence in their software [2]. Running automated tests does not take too much effort or time as soon as they are written. It does not matter if existing components of the project have to be refactored, replaced or new functions are to be added, automation can always increase productivity and quality assurance in this context [3].

Making continuous changes to the software is inevitable, because software components extend, hence change and are refactored every now and then. Once automated tests are set up, developers will have confidence to change confusion, inefficient or tightly coupled parts of the application [2]. Seeing the test coverage data and the green stage of automated tests gives the confidence of quality in their code base. As R.C. Martin points out, software developers *"are afraid they will break it (their code) (...) Because they do not have tests."* [2]

Consequently, testing the layered architecture is possible and strongly suggested on every level of detail. This property greatly contributes to continuous development, developer productivity and quality assurance. On top of that, technical dept becomes easier to manage and avoid.

IV. Conclusion

The construction and the maintenance of software is a demanding and difficult task. As time passes, many components grow too big or get outdated, which asks for the refactoring of the code base. In many cases, the underlying software architecture is coupled too tightly, which makes developers' job difficult to extend their software. Therefore, software architecture forms the frame of every software application.

Accordingly, there is need for a good, robust and flexible design to ensure the avoidance of technical dept and keep the code base robust as well as extensible. Many design patterns on multiple level of abstractions were suggested by researchers in the past to help developers achieve such goals. Recently, the principles of Clean Code were introduced in the industry, based on which the Clean Software Architecture is emerging.

The Clean Code suggests an intent-driven approach to software development. This means all code that is written, should clearly communicate the reason why it was written and explicitly tell what it is doing. The Clean Software Architecture adapts this rule and puts it in relationship with software architecture, where the components are expected to be placed, connected and named with intention. Intent-driven design of components greatly help any developer to understand for what purpose the system was made for.

Additionally, this approach to software architecture embraces principles, such as the separation of concerns, the dependency rule and the inward-pointing flow of control. As soon as these criterion is matched, faster deployment, more robust and flexible software architecture is guaranteed. The software becomes platform-independent, because the components are designed with intent for one single purpose. Therefore the architectural framework will not dictate or violate the high-level architecture of the software in any ways. As a result loose coupling is guaranteed, pieces can be easily plugged into and out from the system.

This paper introduced the above concepts in relation to software development based review of relevant literature. It was concluded that the Clean principles can be adapted to software architecture design, which help developers to collaborate, ensure quality software and to avoid technical dept in the long run. Another finding concerning this approach to development is the decoupled and testable components it provides, which are theoretically independent from platform and easy to replace.

It was shown what the most important approaches towards a Clean Software Architecture are and how it can be achieved. Some of the most important advantages were pointed out and summarized in this paper. Further research in this topic may involve a case study or practical analysis on the topic.

References

[1] R. C. Martin. (2012, Aug. 13) *The Clean Architecture* [Online]. Available: https://8thlight.com/blog/uncle-bob/2012/08/13/the-clean-architecture.html

[2] R. C. Martin. *The Clean Coder: A Code of Conduct for Professional Programmers*, 1st ed. Boston, MA, Pearson Education, Inc. 2011.

[3] R. C. Martin. *Clean Code: A Handbook of Agile Software Craftsmanship*, 1st ed. Boston, MA, Pearson Education, Inc. 2008.

[4] J. Palermo. *The Onion Architecture* [Online]. Available: http://jeffreypalermo.com/blog/the-onion-architecture-part-1/

[5] S. Steve McConnell. *Code Complete: A Practical Handbook of Software Construction*, 2nd ed. Microsoft Press. 2004.

[6] M. Cusumano. *The Changing Software Business: From Products to Services and Other New Business Models Paper 236*. MIT Center for Digital Business. 2007.

[7] E. Gamma, R. Helm, R. Johnson, J. Vlissides *Design Patterns: Elements of Reusable Object-Oriented Software*, 1st ed. Addison Wesley. 1994.

[8] M. Wahler, U. Drofenik, W. Snipes. *Improving Code Maintainability: A Case Study on the Impact of Refactoring*. 2016 IEEE International Conference on Software Maintenance and Evolution. 2016.

[9] M. Doyle, B. Buckley, W. Hao, J. Walden. *Work in Progress - Does Maintenance First Improve Student's Understanding and Appreciation of Clean Code and Documentation*. Frontiers in Education Conference (FIE). 2011.