ELIJAH LEWIS

CLEAN ARCHITECTURE

ADVANCED AND EFFECTIVE STRATEGIES USING CLEAN ARCHITECTURE PRINCIPLES

CLEAN ARCHITECTURE

Advanced and Effective Strategies Using Clean Architecture Principles

Table of Contents

Chapter 1: Architectures: The Basics

This chapter focuses on:

- A brief overview of architectures
- Why is architecture important?

In simple words, architecture is a term that describes the ‘process and the product of planning and designing different structures or systems.

Anything that is built requires a proper foundation, a foundation that is strong enough to balance the entire structure**.** Owing to its strength, a strong and form architecture is expected to impart not only durability but quality as well.

Great quality architecture requires, for its final product, to be a blockbuster invention, not just great quality tools, and materials, but also a competent, diligent, and hardworking team, one that is willing to put in all efforts to yield a product that catches everyone’s eye at first glance.

A poorly built and poorly managed architecture will not only appear to be bad in everyone’s eyes. Still, it will also become a constant source of problems and worries for the developers, too, for it is the foundation, or the base, on which the entire system or building is relying upon.

An architecture is actually the science and art of drawing out a person’s abstract thoughts in real form. It does not necessarily denote a building in the literal sense, but anything that has science applied to it, and has been built to fulfill one’s clients, or business’ needs by accustoming to all their demands.

Architecture, both finalized and in the planning stages, is the need in many fields, other than the constructions and building one!

Anything that demands **organization, management, the utility of resources** as the primary requisites, will ultimately require the application of architecture in the process to proceed systemically, and yield a well-planned final result.

Why Do We Need Architecture?

Architecture makes it easier for people to take a complete look at what they plan to execute, before it happens, so that any changes that they want to be made can be made easily, instead of getting a product made, that they don’t like at all. It is through this effective communication that plans get structured, with more success ratio than the failure one. Things are made a lot simpler, and everyone’s choices get a chance to get incorporated in the process, with full secrecy being provided from telling the other people outside the project about the project which is currently being executed.

Chapter 2: Computer Architecture: An Introduction

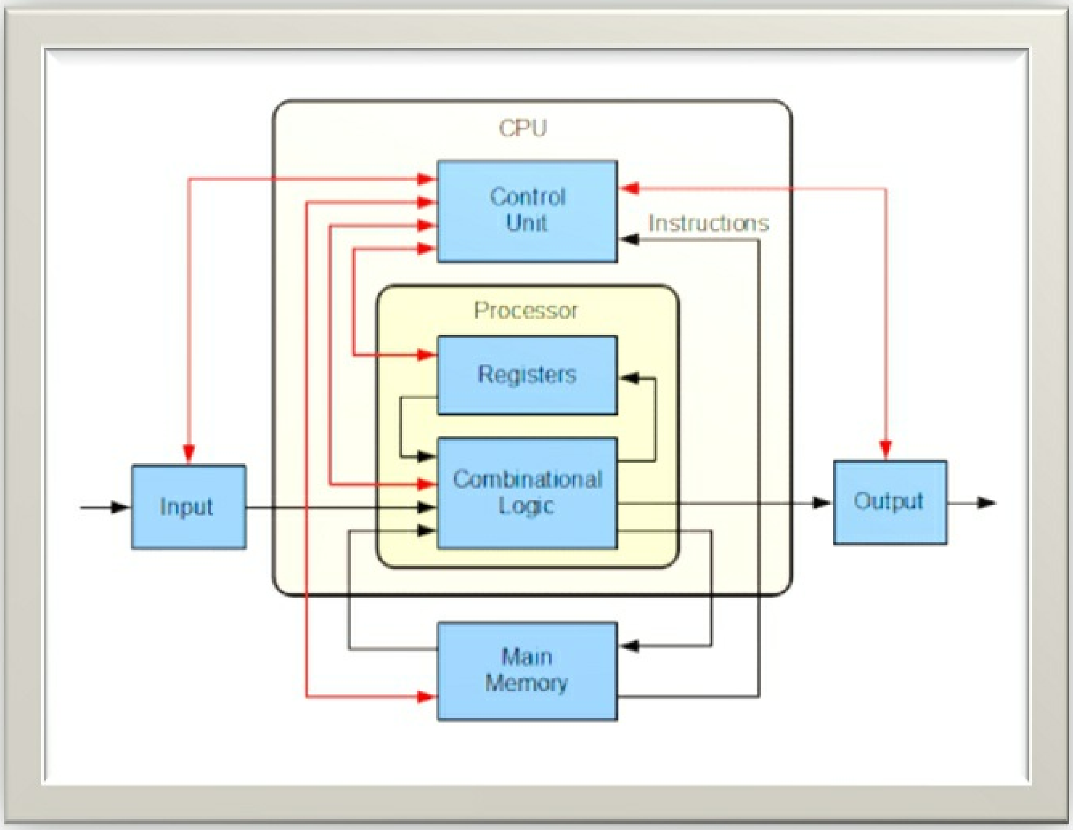
This chapter focuses on:

- What is computer architecture?
- Why is computer architecture needed?
- Different types of computer architecture
- Major components of the different types of computer architecture

Technically speaking, in terms of computer science, ‘computer system architecture’ is a broad-based term that is used to define a computer system in particular.

It explains the various functions, organizations, compatibilities, and interactions between computer hardware and software, which are required to make a system work.

Or, in other words, a computer architecture is simply the communication, or interaction between different hardware and software to make a computer system function.



Basic Computer Architecture (SOURCE: Wikipedia)

A computer system internally consists of a series of hardware and software. The 'processor' is the main 'hardware' that carries out programs. The computer system has a 'memory,' which keeps track of all the programs being run on the computer, while at the same time, storing all the data which is being used, and processed, to bring it to use for another time if required.

The 'software' is the center for controlling all the programs being carried out by the processor.

A software consists of many layers, each of which can only control and influence the one which immediately above or below it, thus ensuring a smooth flow of programs.

Need for Computer Architectures

Computer architectures are not needed but required for the proper organization and carrying out of commands as specified by the user.

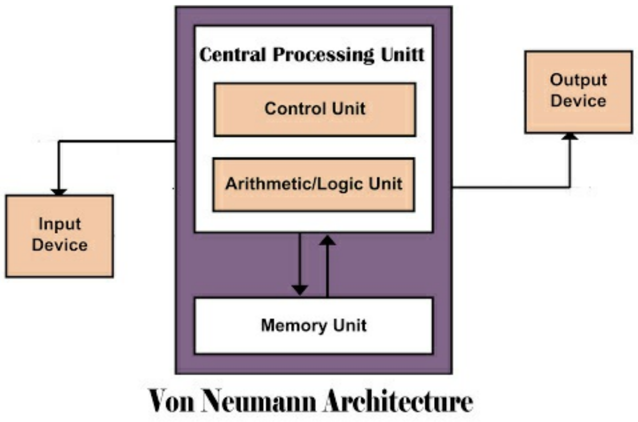
All the components involved in the set-up of a computer architecture will

work according to their specified programs to ensure the effective running of a computer system.
These components determine the requirements of a computer system and work to meet those needs. They mostly carry out read/write operations, but it is their efficiency, and properly designed working and functionality performance that makes them easy to use, and increases their demand overall.

Types of Computer Architecture and Their Major Components

There have been many types of computer architecture proposed by different people over time. A few major ones are listed below.

Von Neumann Architecture To date, many of the computers are based on the John von Neumann architecture, which was proposed by a mathematician of the same name in 1945.



Von Neumann Architecture (SOURCE: Computer Tips and Articles)

This architecture is based on the principle that all data and instructions are stored in a memory. This memory can be approached at any time, from any location, for the execution of a certain program that comes under its hold. The data storing is done within the input device, while the entire data processing is carried out in a central processing unit, or the CPU. The CPU then replays the processed data into an output device, for further workup, as per the user's needs. All of this is carried out under the same block. Maybe that is why this form of architecture is still preferred in the modern world of computers today.

Harvard Architecture

This is a complex form of architecture and has separate blocks for data and instructions. The entire set of data is contained within the Central Processing

Unit, which can process not only data but also design instructions alongside. This form of architecture relies only on the Central Processing Unit, and also contains a separate Arithmetic and Logic Unit. (ALU).

Instructions Set Architecture (ISA)

As the name suggests, the Instructions Set Architecture contains two processor-friendly sets of instructions. One is the Reduced Instructions Set Computer, or RISC, while the other is the Complex Instructions Set Computer or CIRC. The former is preferred over the latter, as it is much more efficient and increases performance on a whole new level, as compared to the latter, which is mainly composed of compilers.

Microarchitecture

Microarchitecture is a modern concept, and is also referred to as "organization system." As the name suggests, it is mostly used in microprocessors and microcontrollers. The entire architecture system consists of a built-in processor, and all the data processing work through it is carried out systemically, hence named "organized."

The data or set of instructions are read and decoded. Then, a parallel system of data is found, which is used for processing the instructions, and the output receives the final product.

System Design

The system design is more of a customized system and is mainly designed while keeping the marketing demands in mind. It is mainly used for product development, and focus ones on keeping the product demands such as architecture, interfaces, etc. as a priority, and fulfilling them subsequently.

Despite the wide range of computer architectures available today, people prefer a system that saves them and delivers perfection par excellence. This is why the Von Neumann system is preferred today, because all the components that it requires to organize, and function properly, are all contained in the same device, instead of being separately contained, which is not only a hassle but time-consuming too!

Through the advent and development of computer architecture, we can understand and reveal all the needs of a system, a user, and the technology involved. We can also have enough developed resources, that we can work on fulfilling those needs in a way that it can appear as if they were never there. The problems are all solved through logics, and designing a compact, yet complete work function that efficiently takes care of the ultimate problem.

Chapter 3: An Introduction to Software Architectures

This chapter focuses on:

- An overview of software architecture
- Why is software architecture needed
- Types of software architecture

As mentioned earlier, the architecture of a system is based on the “basic foundation,” which is laid keeping the clients’ demands in mind. An impressive and marvelous architecture consists not only of great-quality products but also of the good interaction between them.

In the world of computer science, too, there is a great role being played through architecture. A good architecture is the need, not only in the world of programming and coding but in the designing of a software system too.

Architecture is needed for the effective designing and propagation of a software system so that all the tasks that it will execute are laid out in front of the developer so that they could be carried out accordingly and as per priority.

A software architecture system is comprised of all the software elements, the interactions between these elements, and the compatibilities between these elements, which will ensure the proper organization and functionality of the software system.

Software architecture is comprised of a structured model, which is designed while keeping in mind all the organizational, management, and technical requirements to assure that the final product sparks excellence and full utilization of the provided resources.

Through a software architecture system, a programmer, or a software designer, can set limits for the software so that any additions or changes can take place within that specified space only.

A software architecture helps to reduce the complexities, and the tangles involved in the entire program to be designed, and makes working easier, with proper, and fast functionality guaranteed.

Let us take a look at all the famous types of software architecture systems that have been proposed to date, along with their pros and cons, which makes a user free to choose one based on his preferences. This should be noted that multiple software architecture systems can be used or incorporated within another system at all times, which makes it quite easier to make use, and fully utilize the benefits of all the software architecture systems alike.

So far, there have been many architecture systems proposed in the field of software. We shall discuss only the famous ones and the ones who have so far been useful for us.

Layered Architecture (n-tier Architecture)

This is the most common type of architecture, which is in use nowadays. It centers all the high yield information around the databases, which allows the users, or the businesses asking for a software to be designed to comply, and arrange their data into tables accordingly.

It is designed in such a way that data or information is fed into it through the topmost layer, and then it makes its way down all the layers while getting processed, programmed, and implemented throughout the process. The data which is fed into it reaches its destination – the center point, which is usually a database, and from there, directs further instructions as per the need of the program which is being designed.

This architecture system allows all the data to be processed individually, and independently at each level (or layer) of the architecture system, without any interference of the functions of the other levels. This gives the data a chance to be thoroughly processed, checked, and reformatted, with consistency, and prevents any errors from occurring. Even if there arises an error, it could be easily caught at the very same level where it occurred due to the diligent and efficiently checking system.

Some of the best software frameworks to exist today were designed through the implementation of layered architecture; names include Java EE, Drupal, Express, etc.

The layered architecture system works on the principle of the MVC (Model-View-Controller) pattern since it is the standard approach used by programmers in software development. It strictly follows the rules of the layered architecture system.

An “n-tiered” architecture system works best for:

- Coders and programmers who are new to the field of software development, and want to try a hand at App development through any architecture system
- Applications that need to get designed with strict maintenance, and testability features (that is, Apps that are strictly directed to be “fail-proof by the employer/business)
- Novice status applications that need to be promptly built, and introduced into the market
- High-level applications, like those associated with an enterprise, or business, and are direly needed for mirroring the traditional IT departments, and services.

Pros and Cons

Pros

- It is **easily maintained**, even by people who are new to use it!
- It becomes easy for a programmer to **assign different roles** at different levels, thanks to the “layered” design
- Provides a **wide range of testability**, which assures that there would be no errors, or failures once the App is fully launched, and starts working
- It is easier for a programmer to **enhance the quality of any of the layers** which he thinks need help with
- Can be **easily updated**, with no restraints or time boundaries, a plus point that not every architecture system offers
- It can also **add in additional layers** when needed and required by the App.

Cons

- No matter how many facilities it is providing us with, let’s admit this one real fact: it is **complex**. There needs to be a deep understanding done of the layered architecture system to start using it for your own good.

- The facility that is provides the coders with, of going wherever they want to between the layers, and making changes in them as required, can often give rise of **decoupling**, and produce a **messed up environment** of all the dependencies.
- The source code is itself of **no use, if it poorly defined**, and is not filled in with all the crucial details that are needed for its functioning. It needs to have its roles clearly defined, and all the relations clearly understood for properly carrying out its functions.
- The code can also be seen following the "**sinkhole anti-pattern**," which means that it slows down in the final stages, and diverts from its actual duties. It is just focused on passing data from layer to layer, unaware, and unbothered by the fact that it is not making sense anymore.
- Isolating each and every single layer can be too tedious of a task at times. It can sometimes also make a programmer **very difficult to understand** the entire architecture system at a single glance.
- Sometimes, **monolithic development** can become inevitable, which implies that one small change can require a change to occur in the whole system. Or in other words, a complete redeployment of the application software becomes necessary.

Event-Driven Architecture System

Sometimes, we come across such devices, such programs, and such applications that require our input to produce the desired output. Or, to put it into better words, they wait for us to give them a command, and only then do they produce the results that we desire. Such is the principle, on which the "event-driven architecture" system is based off.

The event-driven architecture system achieves this target by building up a "central unit" that receives and accepts data in all forms. It then sends it off to separate "modules" that it is sure of, can take care of the provided data, and execute it exactly in the same way that it is wanted to. This form of transferring of the data from a central unit to the different responsible modules, for the execution of the desired tasks is referred to as an "event," and is usually referring to that particular "code" that is assigned to the particular module.

As a consequence of this type of architecture, there are many different types of data present within the central unit. Still, the modules interact with only those data, which are relevant and useful for them. In this way, a single module will only interact with the type of data that is its own personal concern, unlike the layered architecture system, where every piece of data has to pass through every individual layer that exists within the architecture system.

An example of the event-driven architecture system is the practice of creating a webpage using JavaScript.

An event-driven architecture system works out best for:

- User Interfaces (UI)
- Asynchronous systems with asynchronous flow of data
- Those applications where individual data blocks interact with only a few specific modules.

Pros and Cons

Pros

- They could be **scaled** very easily
- They are flexible and can be **extended easily**, whenever an unexpected, or a new event occurs, and changes are required to be made
- They can **adapt easily** to every type of environment and carry out their functions while being unbothered in even chaotic, complex environments.

Cons

- All the modules involved are decoupled, and fully independent to the extent that to maintain a transaction based mechanism for **consistency becomes very difficult**
- Individual modules can be tested very easily. But to determine if all the modules working collectively as a system are functioning

properly or not, **testing becomes really difficult.** But this is only hypothetical, and testing is supposed to become difficult in cases where only modules are affecting each other greatly

- The central unit must always be prepared for accidental situations, like those where **the modules backfire** or stop operating altogether
- Handling and structuring strategies **to minimize errors is quite difficult** too since many of the modules are handling the same events, at the same time
- The system can overall **start slowing down**, in those cases when there is a sudden burst of messages for the central unit, and it is also busy in trying to buffer those messages to their respective centers
- The designing and development of a "system-wide" structure can be very **difficult to handle** by the architecture system, as sometimes the events have different needs too, and catering them using the same old traditional modules can be very difficult to handle

Microservices Architecture System

The microservices architecture system is labeled as a "cute" little architecture system. It arranges all the data and information into small little programs, and every time a new feature is required to be added in the software, a new little program is created. This is a very flexible and manageable form of architecture system, as with the increasing needs and demands of a software program, or an application, any architecture system can become extremely loaded and populated with data facts and figures, which can lead it to malfunction quite easily.

A very good example of the microservices architecture is Netflix, a very popular movie' and seasons premiering App, which has all its features and functions sorted out separately.

Such a system usually works out best with:

- Websites with small, and compact components

- Corporate data centers which have quite well-defined boundaries of their own
- Developmental and management teams that are spread out all over the globe (remotely, or on location, both)
- Businesses in their initial years, that plan to scale up ultimately
- Rapidly developing web applications, which have a promising and increasing traffic load in the coming years.

Pros and Cons

Pros

- **Small, compact,** and **easily manageable**!

Cons

- The services for which the microservices architecture system is being provided need to be **quite largely independent**
- **Performance can get greatly affected**, as effective communication might not be possible at all times
- Too many microservices offering different features can also prove to be **quite confusing** for the entire program as a whole, once the program starts expanding, and requiring new features every now and then
- It is **not always feasible** for all the large applications.

Space-Based Architecture System

Space-based architecture is also known as the “Cloud Architecture” system.

As programmers, we have observed many times that the majority of the websites are built over the database. Their priorities circulate around the database, and they work hard to achieve their goals, which have been assigned to them by that particular database. But as soon as the website starts gaining access to a lot of traffic, and the demand for new features keeps on increasing, the database is unable to keep up with the new demands, and as a result of this, the entire website crashes, and the system fails.

The space-based architecture or the cloud architecture system is designed, keeping this very same collapsing and crashing of the websites in mind. It comes and distributes the responsibilities of the database into multiple servers. It is mainly concerned with distributing the responsibilities at the processing, and the storage level. It manages the load being held by database, by prioritizing the one which is actually needed for use and eliminating the excess, unwanted one.

This type of a sharing architecture system works out best with:

- Websites with high volumes of data, like click streams, and user logs
- Social networking websites, like Facebook, LinkedIn, etc.
- Websites with data of a lower value, the loss of which is not that important (excluding bank transactions, and online payment services)

Pros and Cons:

Pros

- **Shares the load** of the database, and makes the website work faster
- **Simplifies many basic tasks**, which were otherwise being portrayed as extremely complex ones
- **Speeds up** the entire working, processing, and storage of the website.

Cons

- **Cannot work for transaction** and payment related websites, as it is difficult to run it on RAM databases
- It is really **difficult to design an expertise for cache** providing services for speed purposes

Microkernel Architecture System

Popularly known as a “plugin extension,” the microkernel architecture system is focused on displaying, as well as editing a file. It does so because it was specially designed for those operating systems that are exactly the same, yet used for different projects in different patterns, from time to time. So, the microkernel architecture system is responsible for designing these different sets of operations into their desired pattern, along with displaying them, editing them, and making further necessary changes to present them yet once again in another different pattern, elsewhere in the future.

A Java compiler is usually strapped on with this architecture system, only for support purposes, the rest of the main work is carried out by the microkernel architecture system itself. Not every program requires the use of the Java compiler, so it is just an accessory tool and not a necessary one. Such extra tools are often referred to as the “plug-ins,” and this is where the alternate name for this architecture system came from.

A very important practical application of the microkernel architecture system is the Eclipse – an accessory “plugin” tool that will open files, edit them, make changes in them, and annotate them, along with starting up the processors in the background. Nowadays, Eclipse is also being used to develop code for compiling with the other programming languages.

A microkernel architecture system is supposed to work out best for:

- Those tools which are freely accessible by a large audience of people and/or users
- Applications that have their personalized and fixed set of core routines to be carried out
- Applications that have a frequent set of rules that must be updated quite often at all costs
- Applications that have a clear set of division between their basic rules, and the rules of higher orders.

<u>Pros and Cons</u>

Pros

- Efficiently **manages the tedious working load** of the database
- Allows different businesses to **write plugins** for their various concerns and claims regarding a particular program

Cons

- The decision of **what to feed into the microkernel** is often quite difficult and confusing. The Microkernel itself opts for the code that is most frequently used, even if the programmer wants to select an entirely different one
- The **plugins are required to be efficient enough**, and also, to have in them quite a lot of handshaking code involved so that the microkernel is aware and updated of the fact that a plugin is currently installed, and needs to be put to work accordingly
- Although this is an established fact that choosing what to add into the microkernel is an efficient task, yet it is also established that once a code gets implemented within the microkernel architecture system, it is almost always next to **impossible to change it later**, when it is no longer needed, or no longer prioritized
- Once a fair number of plugins have been programmed to be dependent on a microkernel architecture system, **modifying them is also very difficult** as it has a lot of new responsibilities in the form of plugins on it. A smart way out if it could be to modify the plugins too, along with the architecture system

So, this was an overview of the most popular software architecture systems that are used in the field of coding and programming today. Almost each one of them had many advantages of its own, so the problem is not to select the best one out of all of them, but it is actually to select the best choice for the program that you currently wish to design. Choosing the correct kind of architecture system can either make or break your entire planning of creating a new software or web application altogether.

Some other forms of software architecture systems include:

- Client and Server architecture system
- Component-based architecture system
- Data-centric architecture system
- Monolithic architecture system
- Peer to peer architecture system
- Pipes’ and filters’ architecture system

- Rule-based architecture system
- Services oriented architecture system.

Chapter 4: An Introduction to Clean Architecture

This chapter focuses on:

- Introduction to Clean Architecture
- Principle of Clean Architecture
- What is Clean Architecture?
- Components of Clean Architecture

The main advantage, and most probably, the brightest side of living in this developing world today, is that there is always room for more developments and progress.

Every other day there is some new discovery, a new invention, a new theory being put forward, or simply an old theory getting discarded. This makes us realize how quickly the world is changing, and that how necessary it is for the people to move, and adapt according to the changing world if they really believe in progressing their journey forward.

It would be fully justifiable to say that most of the inventions and new discoveries have been made fully possible due to the increasing use of computers, and the amazing, yet marvelous developments in the software sector. Things that used to be a dream have now turned into realities, thanks to the rapid discoveries, and the implementation of new rules that have turned the old rules into obsolete entities.

Such a concept exists in the world of computer programming by the name of “Clean Architecture.”

Let’s learn more about this wonderful concept that is still making its mark in the computer programming and software industry today.

Introduction

The concept behind clean architecture was first proposed by an American software engineer, Robert C. Martin, who is also popularly known by his blogging name of “Uncle Bob.” He promoted his newly-proposed theory on his blog, from where other software engineers and programmers got to know of it and started to implement it as well.

Principle

The principle behind the clean architecture is the separation of concerns, which is attained through the separation of the software layers. The entire philosophy of clean architecture revolves around this rule.

What is Clean Architecture?

Clean architecture is a software design modality that prioritizes concerns by separating them into sections, which are more appropriately known as “layers.”

The main use of clean architecture is to make sure that all presenting concerns have been taken care of, and have been prioritized by separating them into distinctive layers, one above the other.

Components of Clean Architecture

As proposed by Uncle Bob, an effective and clean architecture must consist of the following components to function properly.

Freedom from Frameworks

Instead of relying on a single resource, or on multiple resources to run your system on, a clean architecture believes in using an independent testing system, so that one can use it freely, as per his requirements, and without the need to constrain his limitations. The programmer is free to make a change anywhere, and anytime, without the fear of having to change the whole program.

Freedom from User Interfaces (UI)

Amazingly, the User Interfaces being used here, are independent and are not relying on the entire system to change along with them. This allows a great number of interfaces to be interchanged, with full liberty, and with no restrictions to change the entire system.

Independence of Database

One’s personal rules are not in any way interfering in the way of databases. Both the business rules and database operate independently of the other.

Independence of External Agencies

This provides a ‘comfort zone’ for your business, that no matter what you do, your rules are personal, and confined within your architecture system only.

Testable

This architecture system provides a wide range of testability. It is flexible to the extent that all tests could be run onto it, without the need of any other external interface, modality, or requests. It alone, or in isolation, proves to be sufficient enough to let all tests be performed over it all alone.

Summary

In summary, clean architecture philosophy is actually an attempt at putting all the rules mentioned above into one single application and using the combined benefits of all of them as one in a customized system.

The clean architecture principle is based on the prioritizing of the concerns and separating them according to their priority levels. The clean architecture system works to ensure that all the data, details, and information are fed into the right kind of layers and that they are executed accordingly. A clean architecture system consists of as many as three to four layers. Still, the data layer, domain layer, and the presentation layer is mandatory to be present for the designing of each and every software program.

The clean architecture system has no doubt simplified so many of the complex and complicated tasks and has given programming and coding a new dimension through the concept of prioritizing the layers. Because of this principle, the designing of software and applications has gotten considerably easier, as now the programmer simply has to take a look at the information that has been fed into the system, and gain ideas on how to plan and execute this application.

This incorporation has been widely welcomed by software enthusiasts all around, and almost all of them have tried putting it to test, by integrating it either into an Android or simply into computer programming. So far, everyone seems to be happy with the introduction of this architecture system.

Chapter 5: Important Terminology Related to Clean Architecture

This chapter focuses on:

- General Terms Used While Implementing Clean Architecture
- Operations
- Use Cases
- Development
- Deployment
- Decoupling Layers
- Decoupling Use Cases
- Decoupling Modes
- Independent Develop Ability
- Independent Deploy Ability
- Duplication

Before we dive into the book and start learning the dos and don’ts of clean architecture, and how to implement it in our application development and programming, we need to get familiar with all the terminologies, and basic wordings that are routinely used while using and implementing the clean architecture system.

Maybe the majority of us will most probably be familiar with some of these words. Still, it is always better to have a guide at hand to avoid any kind of confusion or misunderstandings that may arise due to the lack of knowledge.

Here, you will find an enlisted overview of all the basic terms that you will come across every now and then while getting familiar with the clean architecture system.

OPERATION: It means the conversion, or the process of an architecture to get used, and/or adapt according to the needs and requirements of the system.

It does not matter if the architecture demands excessive work at one time and lesser than normal work at another time, it must, at all times, be intelligent enough to adapt to the ever-changing demands of the system, and act accordingly at all times, to ensure quality performance from its side.

A good architecture system is supposed to always leave this option as “open.” It should not just stick to either of both the options, i.e., carrying out larger operations only, or carrying out smaller ones only. The easy, and hassle-free transitioning from one form to another, without creating unnecessary messes is what makes an architecture an ideal architecture.

USE CASES: In simple words, we shall describe this as the “motive” or “intention” of the system under consideration, or the system that is our business. The entire coding, programming, and designing of the said software will be based on it.

Even if we forget all about architecture systems, and software engineering at this time, and think of the literal meaning of “intention,” it means the “pure feelings, or thoughts of someone, without any fakeness, or hidden meaning,” which implies that our conduct is based off our intentions. The same goes for a clean architecture system in this case. The actual outcome or the final product would be based on the “intention” that is fed into the system from the very starting. So, it is always better and always recommended that the “use cases” which are fed into the system are crystal-clear, and defining enough to elaborate on the meaning of the application in demand.

The importance of “use cases” in clean architecture can be judged from the fact that it is given the first priority, and is fed into the inner-most, or the most concentric, and important of all layers. And in accordance with that, it should be defined enough to convey the entire purpose of the program to the other outer, interfering layers.

DEVELOPMENT: This is an already-understood point. The basis and main purpose of laying an architecture is the development of a process under consideration. An architecture is fully responsible for the final development of a process or a program in our case, and a good development can only be assured by the proper vigilance and organization of all the components that will take part in building it up.

DEPLOYMENT: Deployment proceeds development. Deployment is the utilization of the essentials of the architecture. For our understanding, we shall consider it as “putting the right elements into their right places” so that they can work and function properly, as guided. It is entirely the architecture’s responsibility to effectively guide through the proper development and then deployment of all the needed resources. A good deployment will always give rise to an outstanding system software.

DECOUPLING LAYERS: Decoupling means the sorting out of the entanglement, or the removal of the unnecessary mess that has been created due to the piling up of useless information.

We use the term “decoupling of layers” in this architecture system, as it is basically the main priority on which clean architecture is based, that is, “separation of layers.”
Suppose you want to design an App that will take care of customers’ online payments. To develop it, you will need to enter in all the basic requirements or guidelines that you will need from the system to do for you. For this purpose, you will introduce all the requirements that you could think of, without thinking whether the section of the system in which you are entering your information, will be able to process it or not. This can lead to an entirely confused state in the system, and may even cause it to malfunction, leading to the production of an unhappy client, and an undesirable result. But luckily, this is where the decoupling of layers comes into play. Decoupling helps with the assortment and distribution of information according to its ranking, or priority, which makes the entire architecture system appear simple and clear, and without any messes, because all the required instructions and information have already been fed into the correct places.

DECOUPLING USE CASES: Just like the decoupling of layers is very important to avoid confusion, the decoupling of use cases is very important too. In the literal sense, use cases are instructing the entire system how, when, and where to carry out the specific set of instructions that has been fed into it, without creating any further messes.

But sometimes, these use cases can cause confusion to arise within the system, especially in such a situation where the instructions they have been given are two contradicting statements. For example, at one point, it is instructing the system to delete a particular function, but on the other point, it is instructing it to add that particular function back again. This is neither an intentional error nor a created one, as the use case is simply doing what it has been told to do. Under such circumstances, it becomes really necessary for the use cases to assure their decoupling, too, for failure to do so can result in extremely disastrous results, along with the crashing of the entire system.

Since use cases descend vertically down the layers, we must focus on keeping the use cases separate down that entire level of layers to achieve their decoupling.

A much better, and a smarter technique to achieve this decoupling of the use cases could be done by adding the User Interface (UI) and the database related to one particular use case in a single group. Then both of them could function as one, along with the use case that they were initially instructed to work for.

This way, separate use cases can contain all the interfaces and databases that were initially meant for them. This could help in achieving, not only the decoupling, but also would allow the use cases to proceed peacefully, without any effect, or influence whatsoever from the contradicting use case.

DECOUPLING MODE: This is an important and crucial part of the discussion, as many important functions are relying on it. We shall take our conversation forward from the previous point, where we worked on decoupling the use cases, for the sake of simplicity, and clearance within the architecture system.

The decoupling, which we earlier talked about achieving through “grouping together” the user interfaces, and the databases in a separate use case, can also benefit us in numerous other ways, all related to the better operation and functioning of the system. But what is more important to be noted here is that to achieve better functioning by separating these components from the rest of the use cases, we must also be sure of the fact that these components are capable of functioning to the fullest individually as well. This means that to use them somewhere else within the same architecture system, we need to make sure that only the required individual component is dragged and not the entire group. The use of such components for service somewhere else is known as providing “microservices,” and an architecture that provides and promotes such services is referred to as a “service-oriented architecture.”

However, an interesting aspect of the decoupling of modes is that these layers and use cases can be decoupled at a variety of different levels, all depending on their demand of a particular level. These levels include:

- **Source Level:** This is often referred to as the “monolithic structure.” In this type of decoupling, there is a single load that is fed into the system, to be executed further. In this type of deployment, the involved components execute within the same address space and can communicate with each other through simple function calls.
- **Deployment Level:** In this type of decoupling, the decoupled or separated components get the liberty to be gathered in independent, deployable units, like, for example, jar files, DLLs, Gem files, etc. Though in this type of deployment, many of the components can even be present within the same address space, and also communicate with each other, but still, they would be deployed enough not to impose the changing or rebuilding of one module over the other.
- **Service Level:** In this type of decoupling, we are separating the components at the levels where they are ready enough to carry out or execute their tasks. This mode of decoupling separates the components down at the level of data structures and enables them to communicate via network packets only.

Whatever the mode of the decoupling is, the aim of every single one of them is the same: to divide and separate the components so that they cannot impose their functions on other components, causing their components to be compromised, and malfunction.
Out of all these levels, there is not a single and particular level that could be designated as the “best mode” to be used or applied. This is because as the levels of a system keep on changing, so do its priorities. And a new level demands new services, so the mode that was previously being used might turn useless at some other level. To avoid this confusion, we suggest the priority and choice-based use of a particular mode.

INDEPENDENT DEVELOP ABILITY: The main purpose of decoupling the use cases and the following user interfaces and databases was to avoid any kind of confusion from arising within the system. However, the implementation of this step brought forward many advantages of its own, and an atmosphere of individual development was seen within the system. The fact that one use case is to be considered an individual entity, and is believed to act totally independently, proved to be true, as there was no interference and superimposition of any command of one use case noted over the other.

INDEPENDENT DEPLOYABILITY: And indeed, once the use cases and user interfaces, and databases are acting independently of the other, deployment has been seen to act quite flexibly as well. The proper and efficient functioning of the now independent use cases, user interfaces, and databases depends on their proper and timely deployment at places of their need.

DUPLICATION: For a software enthusiast or a programmer, duplication means death. It is that grave consequence that must be avoided at all costs. Duplication may be of two types; one is "true duplication" or the duplication that is automatically instigated in one component after it sees that a change has occurred in some other component involved in that very same layer. This is an unintentional form of duplication, in which the duplicated component has simply followed suit, and has had no personal preference of its own in doing so. Then comes "false duplication," or duplication, which occurs later in course, after a change has taken place in one component of the system. The later duplicated change is usually unaware of this change, and not necessarily does this duplication take place following suit of the original change, but can occur because it was needed to occur so. That is why it is always advised the programmers keep the coding real, and duplication true, as it would be easier for them too.

To conclude, decoupling is key. Decoupling has proved to affect the system as a whole positively, and can also help in keeping the coding clear and elaborate enough for all the functions to be carried out with ease, and pure dedication.

Let us not forget about what Sir Robert Martin's demands for a clean architecture had been. According to him, and the ideal architecture system is the one that is supportive of:

- the use cases and operations of a system
- the development of the system
- the deployment of the system

- the maintenance of the system.

Chapter 6: Why Use Clean Architecture?

This chapter focuses on:

- Background
- Major Factors Under Consideration
- Uses and Need for Clean Architecture

Not everyone welcomes changes with open hands. And changes aren't accepted whole-heartedly by everyone as well.

Whenever a new technology or a new discovery is made, there are always three categories of people who are usually observed:

- the aggressive ones – those who, right away reject your ideas and theories, label them as 'bogus,' and also warn you to 'let things be as they are right now, it will be in your own favor,'
- those who welcome your theory as if it's not just a theory, but an entirely new concept that has changed their way of living completely, they act as your true supporters and have your back no matter what.
- The third or the most interesting ones are the silent ones. They neither have any interest in seeing what you have to offer, nor is your discovery, theory, invention, or whatever, is making a difference in their life, so they just carry on with whatever they were doing before you came in, with a bang!

Similarly, when the concept was clean architecture was put forward, it received mixed reviews. Software enthusiasts and programming geeks saw it as a great revolution and immediately went on planning on how to incorporate it into their future programming software. The other people were not that much ecstatic on this proposition, they saw it as 'nothing new,' and ultimately, the 'cons' outweighed the 'pros' for them.

However, both positive and negative acclamations have both made the concept of clean architecture earn its rightful position in the world today.

Let's take a look at what was the need for clean architecture to be introduced in the first place, what pointers were kept in mind while proposing it, and how far has this been achieved.

Background

Before clean architecture was introduced, there had previously been a lot of systems and software architectures that people had been using.

All of them were, more or less, based on the same principle of prioritizing the concern. This was achieved by simply dividing the concerns into precisely chosen sections or ‘layers’ and then taken into consideration as per needed.

However, since everything has its own advantages and disadvantages, these architecture systems, too, had some drawbacks of their own, which forced Sir Robert Martin to put forward his theory to bring forth the necessary changes where needed.

Major Factors Under Consideration

Stated below are some of the factors which Sir Robert Martin deemed necessary for the proposition of his entirely new theory of ‘Clean Architecture.’

Mixed-Up Layers

Sometimes, the layers might get mixed up, and the priorities might shift to where they weren’t initially needed. The layers might not appear to be as clear as they were initially planned to be, which might just add up to the list on presenting problems in your list.

No Independence

An efficient architecture requires the liberty, and the full independence to function properly. It does not want to rely on external sources, or other interfaces, or even other databases to fulfill its needs. Rather, it believes in the concept of fulfillment from within and emerging as an architecture that is independent of itself only.

Flexibility

Sometimes, a programmer might wish to make a change at a certain point in his program. Earlier, doing this wasn’t possible because making one single change implied making a change in the entire program, causing it to be disrupted all over, which was quite an impractical approach, that turned down the stakeholders.

But, with the introduction to clean architecture, this has been made perfectly simple, and now, a single change can be made anywhere, anytime, in the place of one’s choice in the program.

Better Communications with Stakeholders

Architecture systems as a whole, give a greater opportunity to software architects to bridge their communication gap with the stakeholders, by first designing and showing them the system, before it actually gets implemented. This helps the businesses, and the stakeholders achieve a better grip and understanding about their data and can ask for changes to be made, as per their choice.

Data Management

There often had news of business data being leaked or that the system crashed because of the UI bugs. It was all because of mismanagement, and over-burdening an already ill-defined architecture system. With the implementation of clean architecture, managing a large amount of data has been no problem, as the priorities speak for themselves, and are catered to, in the same order.

Uses and Need for Clean Architecture

With the advent of clean architecture, software architects and enthusiasts soon learned that it was the answer to all their questions. They began working on learning the basics and all tips and tricks behind the implementation of clean architecture philosophy.

It was therefore discovered that clean architecture, as a whole, was helping people in the following ways:

1. **Developing High-Quality Codes**

High-quality codes were rapidly being developed, and being put to use, thanks to the countless opportunities of trials and errors being provided through its application.

2. **Higher Rates of Dependency**

Coding has a lesser number of dependencies than it had previously. It was free to act alone, along with changes being made in it alone, with no dependencies as it used to have.

3. Organized Work

Thanks to the much-organized priorities, owing to the division into layers, the work appears to be arranged in a much more sophisticated and visually pleasing manner.

4. Separates Logic from Mechanism

Critical use of clean architecture is to ensure that the business logic and delivery mechanism have both been separated and that the business logic has entirely been separated from the other mechanisms and interfaces involved in the coding process, to protect it from threatening leakages.

5. Prioritizing the Main Concern

Since clean architecture is independent of all external applications that could otherwise be implemented on it, giving rise to a much bigger, and complex program, it is helpful to keep the main concern in the limelight, while keeping all the other concerns, like those of the databases, and frameworks secondary.

Conclusion

It is in light of these reasons, that clean architecture became the need of the software enthusiast society, and was quickly put to use to help them with their programming and coding.

It made coding and programming much simpler, easier, and flexible for all the software enthusiasts and programmers because now they were free to design a code or a program as per their wishes and without any strict restrictions on how to handle such courses.

Moreover, it offers many benefits, and its advantages outnumber its disadvantages. It is always better to learn first, and then get yourself introduced to an entirely new field, but the clean architecture system is friendly enough even to let the amateurs try their luck on it.

Clean architecture has become the need of the time because of its benefits, and no doubt, all the fame is for the right reasons.

Chapter 7: The SOLID Principles

This chapter focuses on:

- Overview of the SOLID principles
- Single Responsibility Principle
- Open-Closed Principle
- LISKOV Substitution Principle
- Interface Segregation Principle
- Dependency Inversion Principle
- Practical application examples of each principle
- Conclusion

The SOLID principles, or simply the concept of SOLID, was first introduced by Sir Robert C. Martin, in his paper, "Design Principles and Design Patterns," back in 2000.

This acronym is basically based on five important, object-oriented principles, the application of which promises to make a developing software simpler to understand, more flexible, more user-friendly, more easily and readily, acceptable to change.

Another software enthusiast, Michael Feathers, further elaborated on these five concepts.

Although initially developed for object-oriented software projects, the SOLID principles are also applicable to agile development, or adaptive software development.

The five principles hidden behind the mnemonic of SOLID are summarized as follows:

SOLID	Acronym	Extended Form
S	SRP	Single Responsibility Principle
O	OCP	Open Closed Principle
L	LSP	LISKOV Substitution Principle

I	ISP	Interface Segregation Principle
D	DIP	Dependency Inversion Principle

The foundation of a great quality software program lies in the application of these fundamental SOLID principles that a software engineer, and a programmer, should know to achieve the best quality results.

An overview of the SOLID principles, along with their practical applications, is illustrated as follows:

Single Responsibility Principle

As the name suggests, this principle believes that one class should have a single responsibility only.

That is, every module is responsible for a single component of the functionality, which is assigned to it by a particular software, and all of the processing that goes on within it must be equivocally directed towards the class or module that it would be held accountable for.

In the words of Robert C. Martin,

“There should be only one reason for a particular modality to change.”

The ‘reason’ here being any ‘responsibility’ vaguely related to the software demands.

Example: A creative example illustrating the application of the solid responsibility principle is a developing software itself. Such software has different mechanisms, or processing work being carried out by different sources, like designing, then testing. Then backend developing, each of which is carried out separately, and each through a separate modality. This implies the application of the SRP.

Another example is that of a computer processor, in which data is fed into one part, is fetched from it through another part, and is then processed as per the instructions by another processing unit, which implies that every work of different nature is being carried out by a modality that suits that nature.

Open-Closed Principle

This dual-nature principle, in terms of Robert C. Martin, states that,

“All classes, modules and functions alike, must be open at all times for

extensions, but never for modifications.”

<u>Example:</u> According to this principle, just suppose that one developer has written his part of the code that another developer does not like. He wants to make certain changes to it by adding or subtracting certain things. He can only do so if he tries extending the code, he cannot directly impose his choice of modifications over it.

A fun example to be quoted here is when we don’t like or appreciate certain opinions on different things, we don’t blatantly tell them to change whatever opinions they have for that particular thing. Still, instead, we try to come up with a convincing, non-debatable version of our opinion to convey to them what we feel about this whole ordeal.

This principle helps separate the extended code from the original one.

LISKOV Substitution Principle

The Liskov principle was proposed back in 1987 by Barbara Liskov, and according to this principle,

“Every derived, or child class must be substitutable for their respective bases or parent classes.”

This means that any child class, or any subdivision of an original class, can freely be used, and/or substituted in place of its parent class.

<u>Example:</u> Like for example, if ‘S’ is the subtype of ‘T,’ then the type of objects/programs present in type T can easily be substituted, or interchanged with the objects/programs present in type S, without causing any alterations or changes in the original program.

Interface Segregation Principle

This is the first principle in the SOLID principle series, which is applied to interfaces and not to classes, or modules.

The interface segregation principle is somewhat similar to the single responsibility principle, and states that,

“No client should be forced to use, or depend on methods that are not useful to him, and are completely irrelevant to him.”

This principle emphasizes on the need to use multiple, yet client-specific principles, instead of sticking to an extensive, generalized one.

The interface segregation principles “segregate” or separate a large number of interfaces into multiple ones, so clients can select the ones which are most relevant to them and specific to their cause. These separated or segregated interfaces are given the name of “role interfaces,” as they act specific to their roles.

An interface segregation principle is required to prevent any coupling incidents in the entire program and effectively does so, by decoupling any sort of repetition that may arise within the program, thus causing a smooth flow of events, rather than a non-sequenced, haphazard one.

Example: When you go to a supermarket, you easily go to the aisle containing your products of interest, instead of blindly roaming around here and there, on the outlook for what you want. This is because you have already seen the directions and the boards leading you to where you want to go and purchase the items of your choice.
On the other hand, if you are brought to a supermarket, which has no arrangement of any sort, have items arranged randomly. Without any directions, you would be confused and agitated, because there won’t be anything in your favor, leading to where you want to go.

This is where the use of the Interface Segregation principle comes in handy in avoiding you the hassles of going through things that you don’t need.

Dependency Inversion Principle

According to the Dependency Inversion principle,

“Classes should depend solely on abstractions, and never on concretions.”

This means that high-level modules should not be dependent upon low-level modules, but instead, both of them should be dependent upon abstractions only.
Similarly, abstractions should not be dependent upon the details, but instead, this should be the other way around, that, details are dependent upon abstractions.

The dependency inversion principle forms the core value, or one of the fundamental principles of the layered-architecture system, or clean

architecture.
The main purpose of this rule is to prevent decoupling, again, and to form a loosely coupled class.

In case this rule is not applied, and the former policies preferred, i.e., high-level modules are dependent on low-level ones, then such a program would result which would be tangled, coupled, and complexed more than asked for.

Example: For example, there are two classes, class A and class B. If there is a slight change made in any of the attributes of class A, then class B should neither be affected in any way nor will it influence from the changes made in class A.

Conclusion

The effect and the implementation of the SOLID principles to a software program results in a flexible, easily manageable, and easily testable coding program or software.
The SOLID principles are widely applied wherever object-oriented design spectrum is required because there, they rightly help in achieving the motive.
The SOLID principles help in making the code more extendable, and much simpler to be decoded, or embedded, whatever the case may be.

These principles have helped in the designing of the clean architecture system, as they form the core of this architecture system. Each and every one of the five basic principles play their own respective role in the implementation of the clean architecture system, and contribute to making this system efficient, and perform better in all its programs and applications.

And finally, the practical coding examples given at the end of the chapter were meant to display that if followed diligently. With great care, while implementing either of these five principles, the SOLID principles produce the best results, and the outcome is a flexible code, which is easy to maintain, easy to test, and easy to change!

Chapter 8: The Dependency Rule

This chapter aims to cover:

- Overview of the Dependency Rule
- Explanation of the Dependency Rule
- Importance of the Dependency Rule

In his book, “Clean Architecture: A Craftsman Guide to Software Structure and Design,” Robert C. Martin, aka Uncle Bob, has emphasized a great deal on “The Dependency Principle.”

This rule or principle seems to the major factor that moved him to propose his theory, as most of the points of his theory revolve around this once principle.

What Is The Rule?

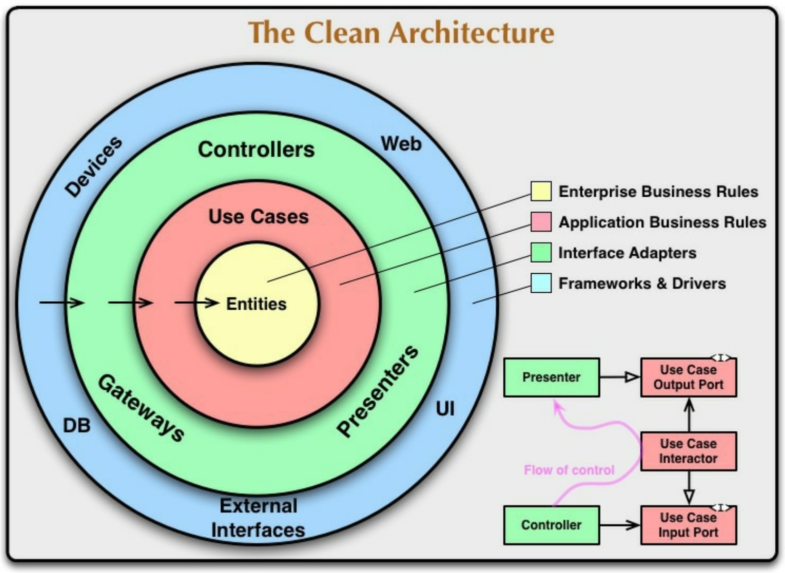
According to this rule:

“Source code dependencies can only point inwards.”

OR

“Something specified in the outer circle cannot be specified again in an inner circle.”

This principle clearly specifies the boundaries between the different layers and distinguishes them from being treated separately.



Explanation

Before diving into the details, it is very important to know that the concentric circles as theoretically put forward by Uncle Bob to support his theory, are not mere circles, but different areas of software.

These software circles upgrade their levels as they go higher.

The outer circles represent mechanisms, whereas the inner circles are used to represent policies.

According to The Dependency rule, every circle involved in the program behaves as an individual entity. It is neither supposed to have any control over the upper or lower circles nor is it supposed to impact the activity of the circles immediately above or below it.

The contents of the circle can range from variables, functions, classes, or any other entities related to software.

It is also true that a code that was in the process could only be applied from the outside towards the inside and not the other way round. This, keeping the above figure in mind, implies that the infra layer can depend on both the application layer and the domain layer, as it is the layer situated on the outermost side. In contrast, the vice versa scenario is not possible in this case, i.e., the domain layer cannot directly influence or effect the layers above it, namely the application layer and the infrastructure layer.
As a result of these implied restrictions, such a system is designed and

brought forward that is simple, to-the-point, and easy to maintain. No inner layers (UI, databases, and requests) can influence it, and alter with its functioning.

In order of priorities, the outermost circle contains the information of the lowest priority level. The values keep on increasing until it reaches the core of the circle, or the ‘domain level,’ which contains the key level, and the most vital of all information contained throughout the circles.

Entities

Critical Business Rules are enclosed in entities. These can be objects with methods; they can be data structures with functions. What they are really doesn’t matter; what is important is that the entities are usable by multiple applications across an enterprise.

If you are not part of an enterprise and are just writing a standalone application, those entities are the application’s business objectives, used for encapsulating the general rules and the high-level ones. When something happens externally, these are the least likely to change. If, for example, a change was made to security or page navigation, you would not expect this to have any effect on these objects. In fact, there should be no effect on the entity layer by operational changes to any application.

Use Cases

The use cases layer contains the ‘application-specific business rules’ for the software. This is where every system use case is enclosed and implemented. Use cases help coordinate the way data flows to and from entities. It then instructs the entities that, to achieve the use case goals, they should make use of the Critical Business Rules.

Again, changes in the use case layer should have no effect on the entities. Neither do we expect any external changes, such as changes to the UI, the database, or any common framework, to affect the use case layer. In simple terms, the use layer has been isolated, so it isn’t affected by any of these concerns.

What we do expect is that any changes in the application operation will have an effect on the use cases and, as such, the layer contains the software. If any of the use case details change, some of the code in the use case layer will need to change too.

Interface Adapters

The software contained in this layer is a set of adapters that are used for converting the data format from the format the entities and use cases find convenient to one that the web, the database, and other external agencies, find convenient. The interface adapter layer is where you will find a GUI's (Graphical User Interface) MVC architecture. The controllers, views, and presenters all belong here too. The models are more than likely to be data structures that have been passed to the use cases from the controllers and then to the views and the presenters.

None of the code that is inward from this circle should have any knowledge of the database or anything about it. If it is an SQL database, then all of the SQL must be in this layer and ONLY in this layer, specifically the sections of the layer that relate to the database.

You will also find any other necessary adapter in this layer, i.e., those used for converting data that comes from external sources, such as services, to the form required by the entities and use cases.

The Frameworks and Drivers

The outside layer of our model is typically made up of tools and frameworks, like the web framework and the database. We don't usually have a lot of code here, other than the code that talks to the next layer, known as glue code. This layer contains all the details – the web, the database, and so on. They are kept in the outer layer so they cannot do much harm.

Why Four Circles?

The diagram above is schematic, purely for illustration purposes. You may need more or less than those four circles, and there is no rule that dictates how many circles there should be. However, we cannot forget the dependency rule. Source code dependencies must always point in, and, as you move into the circle, the policy and abstraction levels increase. The outer circle contains the low-level concrete details, and each successive inner circle gets more abstract, containing ever higher-level policies. The circle in the center is the highest level and the most general.

Crossing Boundaries

I will talk more about this subject later but, for now, look at the small diagram in the bottom right of the above image. This shows you how the circle's boundaries are crossed. Here you can see the presenters and the controllers talking to the use cases from the next layer. Note the control flow – we start in the controller, go through the use case, and, when we get to the presenter, the control executes. Note that the source code dependencies all point into the use cases.

This is a contradiction, and we can resolve this with the Dependency Injection Principle. In programming languages such as Java, the interfaces and the inheritance relationships would be arranged in a way that the source code dependencies provide opposition to the flow at the exact cross-boundary points.

Let’s say, for example, that the presenter needs to be called by the use case. That cannot be a direct call because that would be in violation of the Dependency Rule. Names in outer circles cannot be mentioned in or by any inner circle. So the use case call is an interface in the inner circle, which you can see on the diagram as the “use case output port.” This interface is implemented by the presenter from the outer circle.

All the boundaries in architecture are crossed using the same technique. Source code dependencies are created, using dynamic polymorphism, to oppose the control flow, ensuring that regardless of which direction the flow control goes, we stick the Dependency Rule.

What Data Crosses Boundaries?

Typically, the simple data structures cross the boundaries. If we want, we can use data transfer objects or basic structs, or the data can be a function call argument. It can be in a hashmap or constructed into an object. What is important is that isolated structures cross the boundaries. Passing database rows or entity objects would be cheating; the last thing we want is data structures with dependencies in violation of the Dependency Rule.

For example, some database frameworks respond to a query by returning convenient data formats. This could be called a row structure, and we don’t want that passed across the boundary inwards. This would violate the Rule because it would be forcing an inner circle to have knowledge of an outer circle. As such, when data crosses a boundary, its format must always be in a form that suits the inner circle.

Importance

It must be made sure at all levels that no matter what happens, the layers at a high level are never in contact with the lower-level layers, and do not even communicate with them, let apart influence them. The same goes for prioritizing the need to be dependent on the abstractions only. These rules, even though small, but are very important, and need to be taken care of, so that a system works efficiently, and there is no fear of violating the rules as well.

This dependency rule holds great importance and forms the core of not only the clean architecture but other layered-architecture systems as well.
Every use and every practical application of clean architecture is centered on the dependency circle and depends on it.

Great measures are undertaken to ensure that the dependency rule does not get violated because that literally kills the purpose of applying clean architecture in the first place.

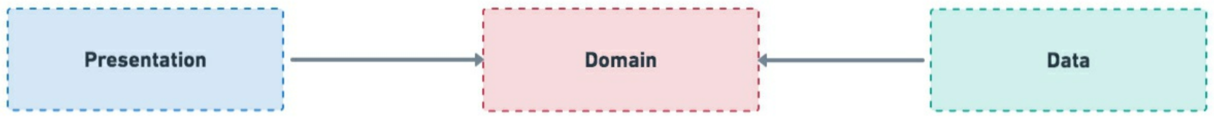
Chapter 9: The Classic Three Layered Structure

This chapter focuses on:

- Overview and relations between the layers and clean architecture
- Details about the layers
- The Domain Layer
- The Application Layer
- The Presentation Layer
- Drivers and Frameworks

Undoubtedly, when someone mentions “clean architecture” to us, the first image that pops up in our minds is that of a colorful circle, with many circular subdivisions in it. This “circle” is the magical theory, and the secret behind so many flexible, yet amazing software programs that we see today.

But this circle is not as simple as it looks like. Hold on, it has definitely made a software enthusiast, and a programmer looks forward to designing new and innovative programs. Still, the circle itself needs close attention to be given to it, to learn about it and to master in its skills of making things a lot easier for us.



Simplified Clean Architecture (SOURCE: proandroiddev.com)

The figure given above is indeed a very simplified version of the clean architecture ‘layer’ system. It shows ‘domain’ as the center of all attention, which is on the receiving end of both data and presentation. But what is the basis of all of this? What causes the domain to be centralized? And why is the domain of so much importance in this architecture? And what is the subsequent stimulus to start this cycle of layering? And most importantly, can there be more than these three layers?

We shall learn all about in the following sections, but before that, a little introduction into the basics is required.

What are these Layers?

The layered circular model is the classic representation of Uncle Bob's clean architecture. This circle consists of 4 'layers,' each of which has its own individual responsibilities and work separately, to achieve their single, main target: the separation of concerns.

Let's start learning about the circles, or layers, in order of their priorities, starting from the innermost one, which contains the most vital, most concrete form of all information.

1) The Domain Layer

The domain layer is the innermost of all circle, located in the center of the circular model. It is independent, can only receive data and information from the consecutive layer residing above it, but can itself not affect any of the layers present above it.
The domain layer contains the most vital of all information, and all the data contained here is of the highest value. The other layers, along with everything else involved in the system program, are dependent upon this layer alone.

In terms of the dependency rule, the domain layer is expected to have the highest level of dependency, as it is the layer on which almost an entire program is relying upon. It is only on the receiving end, and cannot share any of its information with the others.
One of the main reasons why this domain layer does not accept any input from the other layers is to prevent cycles, which can further add up to complicate this already simple process even more.

What Does The Domain Layer Contain?

In clear cut words, the domain layer contains a set of business 'data' and 'logics.' This layer contains a set of business-wide rules, and also some data transfer objects, both of which are collectively referred to as the "entities."

Entities are literally anything that is generally defining the business rules and regulations or simply explaining what the business object is going to be about. They could even be a simple set of data and instructions and can be used by many different applications at once.

Being independent and acting as a single entity, any change that is applied to, or any change that occurs on the layers above the domain layer, have zero to no effect on the domain layer. Its boundary remains preserved, and these rules are totally impossible to change or even let change affect them in any way.

2) The Application Layer

The application layer is the ‘logical’ layer and contains all the logical requirements that are required by the application to fulfill its functionality criteria. This logical information is used to fill in all the spaces that are required to cater to the case scenario of the given program.

What Does The Application Layer Contain?

The application layer contains application-specific information only. It operates through ‘use cases.’

The **Use Cases** function as “interactors” **and make sure that there is a smooth flow of data to and from the domain layer to the application layer**. They direct the data from the application layer into the domain layer, and vice versa too, and also make sure that the domain layer is utilizing this entity into achieving the specified goals of the application.

However, any change in this part of the layer affects the use cases, and ultimately the part of the software contained in this layer.

But all the changes taking place in the application layer will have no effect whatsoever on the entities, and their back and forth flow. Similarly, no effect will take place here due to any other external sources, such as the UI, databases, or frameworks that are imparting its effects whatsoever.

3) The Presentation Layer

The presentation layer is the topmost layer of the layered-architecture, as seen from the outside.

What Does The Presentation Layer Contain?

The presentation layer contains a set of interface ‘adapters’ that work together to convert all the data and information from a format that is easy to read by both the application layer and the domain layer into a format which is most conveniently understood by an external source, such as databases, user interfaces (UI), or common frameworks involved in the software system.

The main priority of the adapters present in the presentation layer is to

convert all the data into the easiest and most convenient form there is, for the database to understand.
All the layers present beneath it should be unaware of the now present data in the presentation layer.

It contains all the finalized classes of data that are required to send back to the client.
It also contains some other adapters which convert the external information being fed in from an outside source, into a format which shall be easily read by the use cases, and entities, at a later stage.

4) Drivers and Frameworks

This is the outermost layer that contains all the databases and frameworks. Being the outermost circle in the entire architecture system, this layer is not of much importance, and not much of the coding or programming takes place over here. It is merely used for glue coding so that the transfer of information could be encoded from outwards to inwards.

What Does This Layer Contain?

All the necessary details are fed into this layer, and leakage, if it occurs, is of no significant value, as it does not contain any of the confidential or valuable information, just the main details, like details about the web, or the database, etc.

Need For More Layers

As per the need of the program and the software, the number of circles or layers could be increased, depending on the programmer. There is no hard and fast rule for using just 3 or 4 layers every time.

It is only because of the simplicity it offers, and the easier way things get carried out, that most software designers have always preferred using the traditional 4-layered method of clean architecture.

Otherwise, the “n-layered” multi-architecture system is widely known to the software community. It allows one to put in as many layers as they want to add, without any constraints but is most often used for those applications and programs which have an increasing demand for new features to be added with time.

Chapter 10: Clean Architecture Components

A component is an entity, the smallest one that can be deployed in a system. Two examples of components are DLL and JAR files. In clean architecture, there are six component design principles – three surrounding component cohesion, or a better way of grouping classes, and three surrounding component coupling, or dealing with the relationships between components.

Component Cohesion Principles

The first three principles are:

REP – Reuse/Release Principle

REP lets us see what should be obvious, even if we see it a little later. We want to be able to reuse components, but we can't unless those components have a release number or are followed up with a process release.

This isn't as easy as you might think; when a component has no release number, we can't tell if the reused component would be compatible with other components. In more precise terms, the developer has got to know the arrival time of a release, and they must know what the release brings in terms of innovation.

From the perspective of software architecture and design, REP states that modules and classes in a component must be contained in a cohesive group. The principle also says that there cannot be any modules or classes assembled randomly in the component. Rather, they should be serving one theme or goal that all of them share. Plus, all the modules and classes in the component have got to be released at the same time. This way, each one can share a common version number and release tracking.

This principle does have one weakness – compared to the strengths of the next two principles: what this one provides is affordable. In simple terms, this principle is defined strongly by CRP and CCP but has a negative perception.

CCP – Common Closure Principle

While the SRP (Single Response Principle) is fully adapted for work with components, CCP states that no component should have more than one reason to change. More succinctly, SRP says that if there are different reasons for different functions to change, then they should be placed into different classes; CCP, on the other hand, says that if there are different reasons for different classes to change, they should be placed in separate components.

When an application code requires changing, rather than changing every component, we must solve the problem by only changing what is in one component. As such, if we keep the changes to just one component, we only need to redeploy that component, ensuring all other components remain unaffected.

At the same time, it gathers in those components conforming to OCP (Open-Closed Principle) and places them into a single component; that component is closed off and cannot be modified. As such, the principle aims to keep the number of components to be replaced as small as possible, should requirements change.

CRP – Common Reuse Principle

CRP is used for working out which component another component or class should belong to. The principle argues that if a class is reusable and it cooperates with other classes during reuse, it should be in the same component as those classes.

The principle also tells us if a component should NOT contain a particular class while telling us what classes should be placed together. When one component uses another, there is a dependency between them. If the component were a class, this would not have a weakening effect on the dependency. Because they have a connection, if the component in use is changed, that change will be reflected in the using component. If the component being used was not changed, or if the change applied had no effect on it, the component still needs to be recompiled. It must then be re-validated and redeployed.

So, if we wanted to develop something that was component-dependent, we would need to ensure that the process depends on every class contained in the component. CRP is, therefore, more like a guide than actually telling us what classes should and shouldn’t be in the component.

We can also define CRP as being a more generic version of the ISP (Independence Segregation Principle). ISP states that we should have no dependence on any class that we do not use, while CRP states that we should have no dependence on a component that we will not use.

Component Coupling Principles

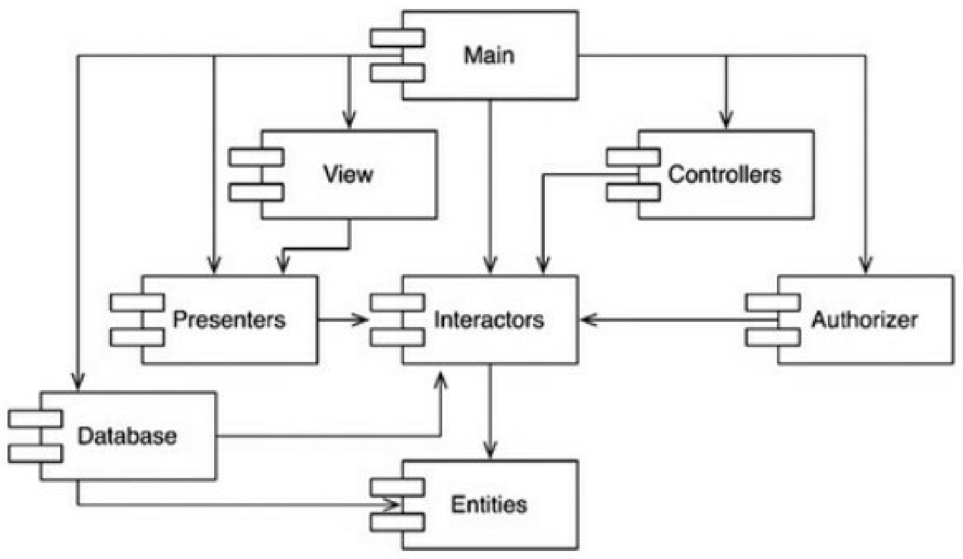
We also have three coupling principles to consider:

ADP – Acyclic Dependencies Principle

We’ve all done it; spent hours working on something, only to find it doesn’t work. When we look into it, we can see that someone else has changed something we developed as being dependent and, subsequently, everything else has stopped working. This tends to happen quite a lot in development environments where there are multiple programmers, and one or more makes changes to the source code independently of the other programmers. Where teams are small, this doesn’t cause too many problems, but when we have a large development team, it’s a different story. These are the solutions to this:

- **Weekly Build –** This tends to be used more in medium projects and involves the developers working Monday through Thursday by themselves. Each develops its code as if it were an independent code. On Friday, all the changes are integrated, along with their system. There is a problem here – the first four days are fine, but the fifth day will expose some serious problems. As the project expands, the integration cost rises, and that causes an unnecessary burden on the rest of the days. As the integration time decreases, the development team’s efficiency follows suit, and this leads to serious issues later down the line.
- **Eliminating the Dependency Cycle** – we can resolve this by splitting the development environment into separate releasable components. Each component becomes the responsibility of a developer or a development team and may be accessible by the other developers or team. When a new component release is made available, the other development teams must adapt their work to be compatible with the release. However, not all changes to one component will be critical to others, and it is down to the team to determine if their component needs to be changed.

Let’s look at an example:



The diagram above shows an application's component structure. This graph is directional and contains no cycle; as such, it is known as a 'directed acyclic' graph.

We'll assume that the development team responsible for the Presenter has gone ahead and released a new component release. Looking at the graph, we can see that the two components will be affected when the arrows are followed in the other direction – Main and View. The teams responsible for these two components must determine whether they need to make adjustments to ensure compatibility with the new release.

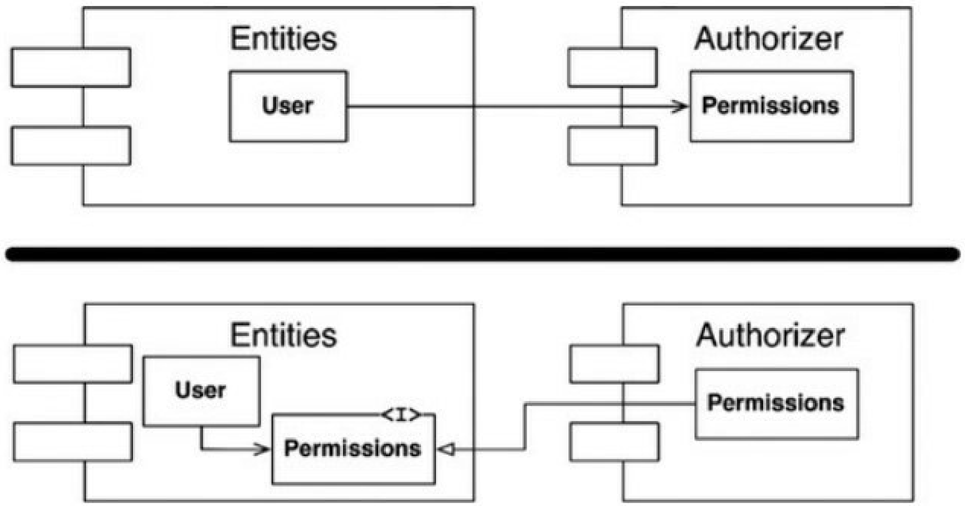
When it comes time for the entire system to be released, we follow the process from the top down. The Entities component is the first to be compiled, then tested before being released. Next comes Interactors and Database, followed by View, Presenter, and Controller. Once Main and Authorizer are complete, the process finishes.

So let's assume we have a developer working on Database. That component depends upon Entities but, because of the cycle, it also depends on both the Interactors and the Authorizer components. Database is, therefore, one of the more difficult components to release because the dependency level can, if not made compulsory, lead to serious issues.

- **Break the Cycle**

In a case like this, we could break the cycle and reconstruct our graph, and we can do this in two ways. First, we can apply the DIP – the Dependency

Inversion Principle. This will result in an interface being placed between Authorizer and Entities, thus allowing the connection to be reversed.



We can then create a component that both the Authorizer and Entities components depend on. We tend to use this when we need a temporary solution because, every time a new component is created, the dependency structure grows.

Top-Down Design

From these problems, we can conclude that we cannot design our component structure from top to bottom. When we design a system, the component structure isn’t the first consideration; simply, the structure should develop as the system grows and changes.

It is a fact that diagrams showing component dependency have little to do with the definition of the application’s functions. Rather, they are more important to how we plan the buildability and the maintainability features of the application. As such, we don’t design them from the beginning of our project. When we don’t have the software to execute those functions for building and maintaining, there is little need for any planning.

We try to keep changes localized, as far as we can, to a certain region, and we can achieve this by ensuring the component’s class distributions are placed where we make regional changes – this is done by giving CCP and SRP more importance.

As our application grows, one more concern becomes evident – the formation of reusable elements. At this stage, we are using CRP to combine the components, and the loops that result from this are broken when we introduce ADP, and the component dependency disappears. However, our graph continues to grow.

If we attempt a dependency structure for the components before we design a class, there will be consequences, some of them serious. It isn't easy to spot the reusable components and the dependencies when we don't have a design. As such, the dependency structure will expand and develop as the system is logically designed.

SDP - Stable Dependencies Principle

We cannot have a design that is wholly static. For a design to be sustainable, it must be volatile, if only a little. By using the CCP (Common Closures Principles), we can create components that are independent of other components and have a sensitivity to change. Some components are deliberately designed as volatile as they are expected to change as the project continues to grow.

However, while doing this, we also need to ensure that the temporary components do not have any dependency on components that cannot be easily changed. If you don't meet these very important criteria, you will struggle to change components that are meant to be susceptible to any change. As such, we can use the SDP – Stable Dependencies Principle – to make sure dependence such as this doesn't happen.

When a component is depended on by multiple other components, it is not easy to change it. For a component property to be changed, it must be done in accordance with every component that depends on it. As such, if a component has a great many dependents, it is called a stable component.

SAP - Stable Abstraction Principle

There will be software in your system that does not require much in the way of changes and won't need to change too many times. This includes policy decisions and high-level architecture, two architectural and business decisions that should never be temporary. As such, the software containing these system decisions must be enclosed inside components considered as stable.

However, this leads to inflexibility in the architecture. In a case like this, we can use OCP – the Open-Close Principle – together with abstract classes to ensure we create classes that are both flexible and cannot be modified.

The Stable Abstraction Principle gives us a much-needed relationship between abstractness and stability. It states that stable components should be made abstract to ensure they can be extended, but it also states that unstable components must be unstable and concrete and that we should be able to modify concrete code easily.

Chapter 11: Other Forms of Layered Software Architectures

This chapter focuses on:

- Brief Overview
- The Onion Layered Architecture
- The Hexagonal Architecture
- Other forms of Layered Architecture Systems.

To stay relevant to the context, this book mainly focuses on clean architecture, and different principles, and rules circulating around this principle only.
However, different forms of architecture have been introduced in the field of software design for years and years.

Every principle has its own pros and cons, and none of it is perfect. It requires years and years of coding and programming practice to make a programmer skilled enough to apply those architectures while planning and designing his software.

Discussing each and every single architecture would definitely be beyond the scope of this book, and would make things more complex. So, for the purpose of simplicity and to understand things from a better perspective, we are discussing the two main types of architecture, which, along with clean architecture, are still in use today.

Almost both of these architecture systems are based on the same laws and rules but only have minor differences, which are elaborated below.

The Onion-Layered Architecture System

Jeffry Palermo proposed this form of architecture in 2008. He wanted to eliminate all the problems that were arising due to the application of layered architecture as a whole.

Just like the horizontal layers that are unraveled when an onion is cut, this architecture system is named so because of its layers being arranged in the same form.

This principle is based on the Dependency Inversion Principle, and in accordance with the onion architecture, this rule states that,

“The outer layers present in the architectural model are free to depend on the inner layers, but none of the data or coding present in the inner layers can freely communicate with, or influence any of the outer layers.”

As seen in the figure above, this form of architecture system consists of 4 layers, each of which is summarized below, in the order from inwards to outwards:

Domain Model Layer

The domain model layer consists of the business-specific components, i.e., entities, value objects, etc. All these components help in identifying the main purpose of the application, which is under progress. It elaborates on what the final product is going to look like.

Domain Services Layer

The domain services layer consists of domain-specific entities. This layer provides all the ‘raw materials’ for the final product that is yet to be produced.

Application Services Layer

The application services layer stores all the ‘use cases’ within it. These elements are all specific to the application.

Infrastructure Layer

This is the topmost and the outermost layer. It is a medium through which entities contained throughout the layer communicate with those present outside. This can range from user interfaces to databases and different tests.

Application Core

The inner three layers, namely the domain model layer, the domain services layer, and the application services layer, are collectively known as the “**Application Core.**” All three of them contain almost the same kind of information which is required for the prompt testing and functionality of the application software.

Main Features of the Onion Architecture

Just like the clean architecture system, the onion architecture system, too, provides a wide range of testability to be conducted over it.

It provides the designer the flexibility to change the coding as many times as he wants, without having to make major changes in the entire drafting of the software system.

The dependency rule is strictly followed amongst all the four layers of the onion architecture system, which somewhat makes it look like a restricted version of the clean architecture, and the hexagonal architecture system.

“The application core” strictly contains all the application-related data in it. It is not allowed to communicate back and forth with the outer layer, with anything related to the information contained in it.

Within the application core, free communication regarding the data, or set of rules can take place, but nothing can go outside from here.

But there are some cons related to the application of onion architecture as well. For example, one tends **to mix up the responsibilities** or the duties in between the layers, which leads to the coupling and clustering of data. Also, the **interfaces might be more** than what seems to be normal, thus adding up to the complexities of the system.

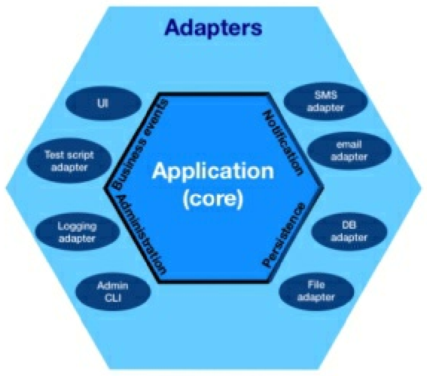
However, this architecture is still used in programming, especially in C+ programming tasks.

The Hexagonal Architecture System (Ports and Adapters)

Alistair Cockburn proposed the Hexagonal Architecture system in 2005.
He felt the need to propose this principle because of the discrepancies that he had been observing in the previously introduced architecture versions of the object-oriented software designs.

These errors and discrepancies included the leakage, and subsequent involvement of both the business data and logic (inner layers), with the user interfaces, and databases (outer layers), and also the unwanted reliabilities and dependencies which often aroused between the layers.

The principle is named so because Alistair wanted to illustrate such a principle where there is ample amount of space to represent all the components and elements that are needed for the effective designing and then the performance of a software system.



Hexagonal Architecture System (SOURCE: Wikipedia)

As illustrated in the figure above, the hexagonal architecture system is somewhat different than the other traditional, customized layered architecture systems.
Instead of being divided into concentric layers, here, a hexagon can be seen, which consists of loosely-arranged components or elements of a future software system that can be interchanged.

The idea behind choosing a hexagon to illustrate his principle was not to showcase six elements, but rather an arbitrary one – to imply that there could be as many sides as the programmer wanted (read: layers in terms of the other architecture systems) in the program involved.

Each component or element present in the hexagon is connected through "**ports**." Communication between all the respective components takes place through these ports. Whereas, for communication with external sources, or those components which are not present in the hexagon, "**adapters**" are used.

The number of ports present may vary, there may be a single port, two ports, and even multiple ports present within the hexagon, depending upon the complexity of the project under consideration.

A brief summary of the different elements or layers contained within the hexagon is given below.

The Domain Layer

As always, the domain layer occupies the innermost layer of the hexagon. It consists of all the business logic, that is, general details about what the future software is going to be about, and ideas regarding how it will be designed, etc.

The size of the domain layer depends on the size of the information and the logic contained within it.

The Application Layer

The application layer lies just outside of the domain layer. It allows the back and forth flow of information and logics to and from the domain layer. Also, it receives input messages from the outermost infrastructure layer and easily transfers them to the domain layer.

The Frameworks Layer

The frameworks layer is the outermost layer of the hexagonal architecture system. It contains the application-specific code and consists of instructions as put into it by the application software. Not only this, but the frameworks layer also accepts information that is fed into it from the outside or external sources.

Main Features of the Hexagonal System

Just like the onion-layered architecture system, and clean architecture, the hexagonal architecture system, too, provides the facility of testing it freely in isolation. This testability helps in improving its worth, as a whole.

The hexagonal architecture system makes things a lot easier to be managed and handled. The name “Hexagonal architecture system” has now been widely replaced by the name of the “**Ports and Adapters**” system, although many people still prefer to call it by its former name.

It is, in fact, a very revolutionizing invention and has made programming a lot simpler and easier.

Like both its counterparts, this architecture system, too, is based on the dependency inversion principle.

Other Forms of Layered Architecture Systems

Apart from the famous architecture systems described above, there are a few more systems, all with one same motive: **separate the concerns** and prioritize them accordingly!

A few of them are named as follows:

1) **Screaming Architecture** – proposed by Robert C. Martin
2) **DCI (Data, Context, and Interaction) Architecture System** – proposed by James Coplien, and Trygve Reenskaug
3) **BCE Architecture (Boundary-Control-Entity) Architecture System** – proposed by Ivar Jacobson

Each of these architecture systems believes in diving the different elements of a software into their constituent subdivisions, or “layers,” to clarify what the actual motive is, and work together towards achieving that one common goal.

Chapter 12 Clean Architecture: When We Cross Boundaries

This chapter focuses on:

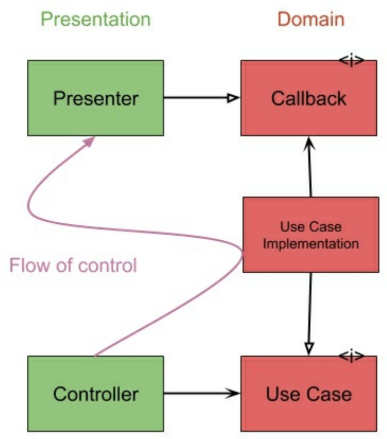
- Crossing Boundaries
- What data can pass through these layers?
- Discussion on the ‘boundary-crossing’ figure

Since we have read in detail about the individual layers of a clean architecture system and how they function separately to make a system software effective, we now move on to explore the boundaries of each layer, what happens, and what changes take place when those boundaries are crossed.

The literal meaning of the word “boundary” is limit. Boundaries define the limitations and elaborate the restrictions of a particular thing, that is under consideration. Working under boundaries, we know better what our defined limit is, and within what specified area we are supposed to work within.

Talking in particular about the field of software and programming, boundaries play an important here in this field too. It helps in keeping the desired elements and components in a restricted limit, and prevents them from interfering, and peeking into the other side of the system.

Crossing Boundaries



Crossing Boundaries (SOURCE: Talk Desk Engineering)

The figure illustrated above show how boundaries are being crossed, while in the application of clean architecture.

What Data Can Pass Through These Layers?

First and foremost, to remember is that the data that crosses through the boundaries is none other than the simple data structure that has already been fed into the individual layers at their subsequent levels. An important thing to be kept in mind is that only simple and isolated data can cross the boundaries. You cannot simply try to combine data from let's say, the domain layer, and the application layer, and then send the collective sum of both of them outside the boundaries. This is beyond the scope of the boundaries, and since it violates the Dependency Inversion principle, it does not take place that easily too.

Figure Discussion

The figure above shows the exchange of communication taking place between two layers; that is, both the controllers and presenters are seen exchanging communication with the other layer, i.e., the 'Use Cases' layer.

In the very same figure, if we visualize the flow of data, we see that the data starts flowing from the controller, moves through the 'use case' layer, and finally ends up settling in the 'presenter' layer. While at the same time, source code dependencies can be seen pointing inwards only.

If such a situation arises, that the ‘use case’ layer needs to contact the ‘presenter’ layer, it cannot be made possible under normal conditions, as it directly violates the ‘dependency rule,’ by contacting an outer circle layer. So, this issue is resolved by using the ‘use case call’ as an interface, and have the ‘presenter layer’ contact the ‘use case’ using this medium as the output port.

This is how boundaries are crossed in different architecture systems. This not only makes things easier, but also provides a free leverage for important data to cross through, which cannot be otherwise possible without violating, and killing principles that hold the entire architecture system together.

Chapter 13: The Architecture of Application Design

Up to now, we’ve looked at the principles that go into clean architecture, along with the components that pull it all together. Now, perhaps it is the time to ask ourselves what architecture actually is in terms of application design.

We need to know what it is and what it can do and cannot do, so we’ll start with an obvious question – what is an architect?

In terms of software and applications, an architect is nothing fancier than a computer programmer. There are those who are mistakenly under the impression that a software architect only deals with the issues and problems that arise with high-level application development – this is not true, at least most of the time. Yes, it is fair to say that software architects are some of the very best computer programmers in existence. It is fair to say that, for the most part, they handle complex tasks and issues in the programming arena. They lead their development teams to be as productive as they can be, producing application designs that offer maximum productivity. What they don’t do is write reams and reams codes like most programmers do. They are, however, fully and limitlessly involved in every task related to development and programming.

So, we can say that the architecture of any application is, in fact, the combined shape of every person involved in the development project. That shape takes the form of dividing the unit components, at the system level, including the way those components have been arranged and how they interact with one another.

This shape occurs for one primary reason – the fact that all the system requirements must be enabled – that includes the development process, component deployment, application and component maintenance, and so on. The approach surrounds having as many options as possible open for as long as they are required.

Did that surprise you?

Most people are of the opinion that software architecture is only about making sure systems are optimal and that they work exactly as intended. That is a major objective, of course, and that must be supported by the system architecture as a priority. But there is so much more to it, and system architecture actually doesn’t have a lot to do with whether or not a system works.

There are plenty of applications about right now that are very poor in architecture, but they still work extremely well. The issues aren’t really about the way an application operates; they are more about how they are deployed, in the maintenance and in continuous deployment.

However, this isn’t to say that architecture plays no supporting role in the behavior of the system. I can tell you that it most certainly does play a role and a critical one at that. But, critical though it is, it is still a passive, cosmetic role.

There is one primary objective to software and application architecture – it provides support to the life cycle of a system’s development. If the application architecture is as near to perfect as it can get, we can understand the system that much easier. That leads to development becoming much easier, along with maintenance and deployment. Make one system easy, and the rest will fall into place. We can say that architecture has a direct objective – to reduce the lifetime costs of an application significantly while at the same time increasing the productivity of the development teams and the programmers.

Application Development

If an application is difficult to develop, it is much more likely to have a longer life cycle and is more likely to be productive. That means the architecture of a system is designed to ensure the development team has an easier job when they develop the system.

The structure of the development team will have a direct impact on the architectural decisions made. On the one side, a small team that consists of just a few developers can effectively collaborate on the development of larger systems without having to rely on those components and hierarchies that must be perfect in their design. A team such as this is far more likely to look at the criticisms of an architecture and see those criticisms as a barrier, especially in the early stages of development. This is the reason why so many applications have a poor architecture – they began with little to nothing, and because a small development team was all they had, they opted out of creating megastructures so they could avoid those barriers.

On the other side of things, where we have multiple development teams, with each team consisting of multiple developers, a system is less likely to push forward in progress unless the application is divided into components that have reliable, stable interfaces. At the end of the day, the architecture is more likely to be developed on multiple components, with one team working on each component.

The result of this is likely to be a less than good architecture where deployment, development, and maintenance of the system is concerned. However, it is still the most likely way that development teams will work, especially when they are working on a tight development schedule.

In another way, if multiple teams are involved in the development of a system, it is unlikely that the system will progress satisfactorily unless they split the application into several components or elements, each stable, reliable, and well-defined. If you have five development teams, the application would be divided into five primary components.

Application Deployment

In terms of application efficiency, every application must be simple to deploy. If the cost of that deployment is too high, the system simply won’t be that useful. So, from this, we can say that application software has a primary objective – to design systems that are easy to deploy.

Throughout the development phase, right at the start, we rarely give any thought to deployment. Because of this, we can develop an architecture that ensures systems can be developed much easier but are much harder to deploy.

As an example, when they are in the early phases of developing a system, the developers can choose to use micro-service architecture. This ensures that systems can be developed easily because micro-service architecture provides component boundaries that are much firmer and interfaces that are far more stable. However, when it comes to system deployment, the developers would find that they had rather a lot of micro-services. Those services have configuration links between them and, together with the initiation timing for the micro-services, these were a huge area where errors could occur.

If the architects thought about the deployment earlier in the development phases, they could have decided that they didn’t need so many services, or they could have made the decision to combine some of the services that had ‘in-process’ components, as well as to develop better methods for link management.

Application Operation

There hasn’t been so much study on how architecture affects system operations as there has been on how it affects development, deployment, and maintenance. When we keep adding hardware to a system, it should be able to handle the increase in operational challenges while not having an adverse effect on the application architecture.

This is not uncommon. Where an application system has a bad architecture, we can make it better just by adding some new hardware, such as storage devices and servers. Hardware is much cheaper now than it ever has been; what is expensive is the cost of people, and the result is that architecture that puts up barriers to operations is much cheaper than the architecture that puts up barriers to development, deployment, and sometimes even maintenance.

I am not trying to imply that we shouldn’t find architecture that is in full alignment with operations as alluring. It certainly is, but when we figure out the cost equations, we can see that they tend to point more towards development, deployment, and maintenance.

All that said, application architecture has one more role in operations – if it is well-defined, it can easily communicate what the operational needs of the system are.

Perhaps a better way of putting this would that the architecture of a system should ensure that all the developers involved should be aware of what the system operation is. The architecture should not only show that operation, but it should also ensure that the features, system behavior, and use case are raised to the high-level entities seen as milestones in the project by the development team. This would make sure that we have a much easier time understanding the system, and that, in turn, ensures that the system development is easier to do, as is system maintenance.

Application Maintenance

Take a look at an application or software; see what the features are. You should be able to deduce that maintenance features are among the most expensive. We continuously see a stream of new structures and added to the unavoidable errors that arise, and the solutions require to fix them, which causes a significant rise in the wastages from human resources. The major cost of maintenance lies in locating the errors and working out what and where the risks are; every application must be combed through thoroughly, and any errors must be fixed as soon as they are found.

However, as we fix one error, there is a high likelihood that we will cause another, albeit unintentionally. This adds to the risk cost, but we can mitigate this in one way – careful investigation of the application architecture. We can also try to minimize the risks by dividing an application into components or elements and isolate those with stable interfaces.

Every application has two values associate with it – structure and behavior. The structural value offers the most importance because it is the structure that ensures software is soft. Applications were devised because we needed to find a much easier way of changing the way a machine behaved. Flexibility relies almost entirely on the shape of the system, the planning that goes into the elements, and how the elements are interlinked.

This is how we keep software soft while ensuring the most amount of options possible are left open. We can break or disintegrate pretty much any application into two critical parts – policy and detail. Policy defines the business rules and the business procedures, and this is where the real value of an application lies. The details contain everything that is needed for humans, other systems, and the shape of the system to consider the policy as critical while, at the same time, making sure the details are of no consequence to the policy. By doing this, we ensure that we can delay or, if necessary, defer any decisions to be made on the policy.

An example of this can be broken down as follows:

- It is not really considered critical to choose the database system in the early stages of development. Why? Because, at that time, the high-level policies don't really care about the database and don't have any interest in a specific one. If our architecture were cautious, it would be safe to say that high-level policy isn't really relevant to relational, distributed, or hierarchical databases.
- At the same time, we also don't need to decide on a web server in the early stages of development; the high-level policy should not

know that it is being sent to the internet. If the high-level policy has no knowledge of HTML, AJAX, JSP, JSF, etc., then there is no requirement to decide on a specific web server until you are further into the development cycle. More often than not, you won't need to make the decision on whether the system requires the web for delivery.

- We do not need to adopt REST early in the development because, once again, the high-level policy does not need to be worried about SOA implementation or even the frameworks for the micro-services. Plus, high-level policy has no need to concern itself with SOA or micro-services either.
- Finally, we do not need to implement a framework for dependency injection in the early development stages, either. It comes back to high-level policy, again, not needing to know about any dependency resolution.

If we can develop the high-level policy without having to make a commitment to the system specifics, we can delay the decisions regarding the details or even defer them for a much longer period. In fact, the longer we can delay them, the more time we have to gather critical information to make sure they are developed as they should be.

This brings another benefit – the ability and the time to try different tests. Where some of the policy works well but isn’t sure of the database, we can test it by creating links to several different databases, thus determining the database that performs the best. We can also apply this testing to web frameworks, systems, servers, even the web itself. The important thing is that we leave the maximum amount of systems open, at least as far as we can, allowing us time to experiment further. The more experiments we can do, the quicker we can get to where we no longer need to defer the decisions.

On occasion, it will be apparent that many of those decisions will already have been made by someone else. Or it may come to light that the organization already has a commitment in place to use a specific database, server, or framework. Any good architect will work under the assumption that those decisions have NOT been made, and the system will be developed in such a way that any decisions already made may be changed for as long as possible. And a good architect will also make sure that the biggest possible number of decisions are available at any one time.

Device Independence

What is device independence? The easiest way to explain this is to pop back in time. In the 1960s, programmers were mathematicians or engineers, all from different fields, and the idea of a system was relatively new. In those days, many mistakes were made, but nobody realized that these mistakes were errors. The commonest one was to bind a program code to an IO device, i.e., if something needed printing, the code had to be written, so it used the correct IO instructions to control a specific printer device. These codes were called ‘device-dependent.’

Let’s see an example; the following codes would be used to print something to a teleprinter:

```
PRTCHR, 0

TSF

JMP .-1

TLS

JMP I PRTCHR
```

PRTCHR is a subroutine used to print specified characters on specified printers. In line 1, the 0 was used for storing a return address, and TSF, in line 2, was used to skip the instruction that came before it, so long as there was a character ready for printing. Should the printer already be busy, TSF would drop through to JMP-1 in line 3. In turn, that would go back to TFS. Should the printer not be busy, TSF would jump straight to TLS, where the stored character would be sent directly to the printer. Lastly, JMP-1 PRTCHR is returned.

This seemed to work well at the beginning. If we wanted a card reader to read a card, we needed a specific code to communicate with the reader directly. However, if we wanted the card punched, we need a code created for direct deployment to the punch. Things did work well, and it was hard to understand that the biggest mistake was the approach. It wasn’t easy to handle huge amounts of punched cards; they can be lost, defaced, dropped, shuffled, and so on. Losing a card is not difficult; neither is inserting additional cards, and that meant it wasn’t going to be too much longer before one of the biggest concerns was the integrity of the data.

Magnetic tapes were the next thing; the cards could be moved to those tapes, and if we dropped a tape, it wouldn’t get damaged, the cards couldn’t be shuffled, or damaged. And we can keep accidental card loss to an absolute minimum with the taps, making the system that much more secure.

Tape was not only a more secure method, but it also proved faster for reading and writing to, as well as being easier to back up. Unfortunately, at the time, most applications were developed around the premise of card readers and card punchers, and that meant a massive amount of work to rewrite them to use magnetic tapes.

As the decade drew to an end, many lessons were learned, and programmers began to make use of device-independence. At the time, the operating systems abstracted the IO devices into functions that could handle an individual application that closely resembled a card. These systems would invoke the OS services that dealt with abstracted devices with single records.

In this way, it was easier for the operators to determine if the OS should link their services to a card reader, a magnetic tape, or another unit-recording device and tell the OS what to do. This meant programs could now read and write cards and, at the same time, read and write to tape, without having to change a single thing. That is where OCP came into the picture.

Chapter 14: Architectural Independence

We already know that good architecture needs to support all of the following:

- A developing system’s use case
- A developing system’s operations
- System development
- System deployment
- System maintenance

So now, we need to delve a little deeper into all of these to discuss them further; the more understanding you have, the more likely you are to use clean architecture in your programming and application development.

System Use Cases

The use case is the primary critical element; the system architecture should always support the system’s purpose. Let’s say, for example, that we had developed a shopping catalog. The application architecture would need to fully support the use cases for a shopping cart. This should be the real significance of the application and the main concern of any architect.

However, we now know that the architecture and the system performance are not really connected. The result of that is that the architecture doesn’t leave many behavior-related options open. What you do need to keep in mind is that influence isn’t everything. The perfect architecture relies heavily on behavior clarification and exposure, thus allowing us to know the intentions of the system from the architectural level.

Going back to the example of a shopping catalog, what would the best architecture be for a shopping cart? Quite simply, it would be a standard application for any shopping cart. We would easily see a use case like this in the structure, meaning developers don’t have to hunt the behavior down – it is there, fully visible, in the top-level of the system. The components are modules, functions, or classes, and they will be located in the architecture where they can be seen. Plus, their names will be descriptive, clear, and easy to understand.

System Operation

Architecture plays a critical role when it comes to support for system operations. Let’s say that we required our system to handle at least 10,000 customers/clients per second; it is vital that this operation has the full support of the architecture, as should each use case’s response time. Should our system be designed in a way that huge amounts of data could be queried in milliseconds, our architecture has to be designed around that.

For some systems, this will require that we use arrays of services for the processing elements, and these must be capable of running simultaneously on multiple servers. Other systems will several small threads, all sharing the address space for one process and just a single processor. At the same time, we will only need a few processes for these systems, running in isolated address spaces, and we may also have other huge programs that are relatively simple and require just a single process.

So, does this seem a little odd to you? That is a decision that a good architect is likely to leave open. Huge systems with a dependence on massive structures can’t be upgraded to multiple servers, threads, or processes very easily, should they need to be. On the other hand, if an architecture is designed to properly isolate unit components, with no assumption that they will have a means of communication, then, as the requirements of the operation change, upgrading and transferring new processes, services or threads in will be a good deal easier.

System Development

Somewhere else that architecture plays a vital role is in providing support for the application development. According to Conway’s Law, “any organization designing a system will produce designs with structures that replicate the communication structure of the organization.”

If a system is created by several teams and multiple concerns, the architecture should be designed in a way that each team can manage its own independent action. This means the teams will not interfere with one another throughout the development phase. Accomplishing this is best done by partitioning a system into components, each isolated, and this must be done so that the components can be deployed independently. Those elements are then allocated to one or more teams that are independent from the other teams.

System Deployment

The last important role for architecture is to ensure that the deployment of the system is smooth and trouble-free. Great architectures will not rely on a requirement for a small script and tweak deployment. Neither will it have any requirements for the manual creation of directories or files should they need organization. Truly great architecture ensures that systems can be easily deployed once they have been built.

This is achievable by making sure that the unit components are partitioned properly and isolated. These are the master component that glue the system and hold it together, and every one of them must be properly initialized, integrated, and monitored.

Good architects will find it easy to balance the elemental structure with every concern we have discussed in a way that everything is happy and works together seamlessly. This won’t be easy to achieve because that balance is a tough one to find. Most of the time, developers won't know what the use cases are, what the team structure is, the constraints in terms of the system operation, nor will they know what is needed for deployment to go smoothly. And the biggest downside is that, even if these things were known up front, there is a good chance that, as the system life cycle progresses, new developers will come on board.

Most of the principles surrounding architecture are not expensive to adopt, and this helps to keep the concerns balanced, even when there is little knowledge of the targets. These principles help developers to correctly partition a system, creating the elements that allow programmers to leave the maximum number of open options for as long as possible. Simply put, great architects can build systems that are open to change in several ways and with lots of open options.

De-Linking the Layers

To understand what de-linking is, we need to examine the use case. One of the main priorities for an architect is that the structure they build takes care of every use case the system requires. The problem is the architect has no idea what those use cases are when he begins development. He does, however, know what the system is being developed for. Let’s use our shopping cart example again. The architect knows that he can use SRP (Single Responsibility Principle), and he can use CCP (Common-Closure Principle) to make sure that, if anything has the potential to change, no matter what the reason is, it is kept separate. He can also make sure that all those things that change for similar or the same reasons are grouped together. To do this, he needs to know the full context of the system's intention, i.e., what it is for.

With our shopping cart, the reasons why the user interfaces might need to change are not included in the business rules. We also know that the use case elements will need to change. Architects should know that, in a use case, all the user interface parts need to be separated from the business rules, and this has to be done so that they can change independently while, at the same time, making sure the use cases are fully visible.

More often than not, there is a close relationship between the business rules and the software. However, there are some cases where rules may only be stated generally. An example would be that the validation of input fields is regarded as a business rule closely related to the software, whereas accounting interest estimation or counting inventory are both general rules closely related to the domain. These rules change, not always for the same reason, and not always at the same time. That means they must be kept separate so we the changes can happen independently of one another.

Another example is a schema and a database. Both of these use a query language, and that language is a technical detail. It is not related, in any way, to the business rules or the user interface. The language in each of them changes at different rates, independent of the other and independent of the rest of the system. A good architect will always make sure that they are kept separate from the rest of the system so that the changes can happen independently.

With our shopping cart example, the layers the system has been divided into are called de-linked layers. These contain the application-specific business rules, the application-independent business rules, the database, the user case, and other parts of the system.

De-Linking the Use Case

Use cases may also need to change, as well. Back to our shopping cart, the use case required for adding orders is liable to change for many different reasons and different rates too. Plus, the use case can also remove orders as well as adding and changing them.

The use case could be considered as a series of narrow vertical slices, each one cutting through the horizontal system layers. Every use case has a user interface that they use, along with application-specific rules application-independent business rules and a certain amount of database functionality. That means, when we divide the system into horizontal layers, we also get a vertical use case that slices through the layers.

This de-linking is achieved by separating the user interface for the addOrder use case from the user interface related to the delete-item use case. The same is applies to the database, while the use cases and the business rules are stored down the vertical height of the system.

This creates a pattern. If we de-link the elements that are liable to change, regardless of the reason, we can continue adding new use cases with no effect on the existing ones. If we provide support to those use cases by grouping the user interface and the database, existing use cases will not have any effect on how new ones are added.

The De-Linking Mode

Let's think about the way system operations and de-linking are related. If we separated all the parts of the use case, in one way or another, the use cases with the highest output would have been separated from those with the lowest output. If we separate the business rules from the user interface and the database, each will use a different server to run on. If more bandwidth is required for some, they are replicated using multiple servers.

In simple terms, whatever de-linking you do for a use case will also be helpful for the system operations. But, to make sure you take full advantage of the benefits in operational terms, you must use the right mode for the de-linking. If the use cases are running in isolated servers, none of the components should have been dependent on address-space sharing within the processor. The services need to be independent, communicating through the network.

These elements may be termed as 'services' or 'micro-service' by the architect. Architecture tends to be based on services and is often called 'service-oriented.' On occasion, the most critical thing to make sure of is that the components have been properly separated, down to service-level, and good architects ensure that options remain open, and one of those options is decoupling or de-linking mode.

Cases and layers can be decoupled in multiple ways:

- **Source-Code or Binary Level –** in this level, we must be able to control the dependencies that exist between each module in the source code, in such a way that changes to one do not force others to change or recompile. Take the Gem files in Ruby, for example. Every component executes in one address space, and function calls are used for communication between one another. There is one executable in memory, and these structures are called ‘monolithic.’
- **Deployment Level –** in this level, the deployable units have dependencies between them. Included in this are shared libraries, DLL files, and JAR files, and the way they are controlled ensures that when the source code in one module changes, none of the other modules need to be rebuilt or redeployed. The address space is shared by the elements, and function calls are used for communication. Some components share a processor in different processes, interacting by way of linkages, interprocess sockets, or shared memory. The important this is that the components that have been decoupled get partition so they are in units that can be independently deployed, like JAR, DLL or Gem files.
- **Service Level –** in this level, we can minimize the dependencies to data structure level, using network packets for exclusive interaction. Each unit that needs to be executed is binary-independent, and the source moves to components like the services or micro-services.

When the project starts, it won't always be easy to work out which model will work the best. As the project develops, it will all change. Let’s take a system working on a single server; it could end up being developed to a stage where some of its elements need their own servers to run on. Our shopping cart system uses a single server, so, for the time being, it is enough to delink at source-level. Later down the line, we may need to do more de-linking, so things are broken down into small units or services that are easily deployable.

Service-level de-linking is one of the more common, but there is a downside to it. For one, it is cost-intensive, and, for two, it comes with the potential for rough decoupling. It makes no difference if the micro-services are too, shall we say, micro, either; the decoupling will not go smoothly. Service-level decoupling also tends to be expensive at both the developmental level and in the required system resources. If boundaries such as this are not required, it is a complete waste of effort time, and memory, not to mention processing cycles, in having to deal with them.

It is best to use a system whereby we move decoupling to where a service is only created when it is required. In this case, we can leave the components in the address space for as long as they can absolutely stay there, which will leave the options open for service.

Using this kind of approach, we can separate the components at the source-code level, which works well for the project life cycle. However, an approach like this brings its own challenges at the development phase or in deployment, and the result will be that some de-linking is forced to deployment level – this will suffice for the short-term.

The developer will face increasing issues in development, deployment, and operation, but they must remain particular in regard to the units that should be deployed to other services, and they should continue to work at moving the system the right way. As time passes, there will be fewer operational requirements on the system, and the small amount of de-linking at the service-level will move to deployment or source-level delinking.

The best architecture will make sure that we can establish and deploy large systems as on file. Then the system can be evolved into a series of uniquely deployable units. As the system continues to transform, we can reverse this process, reverting our system back to what it was originally.

And great architecture will also protect the source code from changes such as this and leave the de-linking mode open, allowing larger deployments to use single modes, while small deployments will use another mode.

Independent Developability

Development plays a huge role in the life cycle of application development. When we strongly de-link elements, the interface that exists between the teams receives some mitigation. If the business case knows nothing about the user interface, the team responsible for the latter won’t affect the team working on the former. If there is decoupling between the use cases, the team responsible for the addOrder use case won't affect the team responsible for the delete-order use case.

As long as there is proper de-linking between the layers and the use cases, the system architecture provides support for the organization of the team, regardless of whether the team is an element, a feature, or some other variation.

Independent Deployability

When we de-link use cases and layers, we get a much higher flexibility in the development. Provided it is done correctly, we can swap the de-linked use cases and layers in running systems by simply adding another use case by way of a JAR file or service, leaving the components that already exist as they are.

Duplication

If there is one thing that all architects fear, it is developing systems with some kind of duplication. This is never a good thing because, when we have duplicated code, the programmer needs to either reduce the code or remove it.

There are two primary types of duplication:

- **True –** whenever we make a change to an instance, we must make an identical change each copy of that instance.
- **False –** this is also known as 'accidental' duplication. Where duplicated codes develop down paths, at rates, and for reasons that are different, they are not true du0licates. Checking will show that each code has its own components, different from the other duplicate.

Let's take a pair of use cases as an example. Both have similar screen structures, and a developer might be tempted to share the code of the structure. However, this would be classed as false or accidental duplications. In time, the screens would likely diverge, and thigs would eventually look very different. To avoid unwanted unification, careful handling is needed; otherwise, the developers would have a challenge on their hands, attempting to separate the two later on.

When vertical separation is used on use cases, there is a high likelihood of challenges arising. It would be tempting to join the use cases where their databases, screen structure, or algorithms are similar, and, because of this, developers need to exercise care, not removing duplications until they are 100% certain of whether the duplication is true or false.

In a similar way, when horizontal separation is used on layers, there is the chance a developer will see the data structure of a database and another screen as being similar. It will be tempting to pass the user interface the record, but what we should do is create a new view model that looks the same and copy the elements over. This is because we are looking at false duplication; developers should be able to create another model easily and ensure the layers are kept properly de-linked.

Chapter 15: An Overview of the Implementation of Clean Architecture For Programmers

In this chapter, we shall focus on:

- Technologies Used and Required for the Implementation of Clean Architecture in Android Apps
- Components Used and Required for the Implementation of Clean Architecture in Android Apps
- An Overview on the Approach of Android Applications Building through the implementation of Clean Architecture.

Moving towards the end, and concluding part, we thought of making a guideline for all the enthusiastic software engineers, and programmers, so that they can benefit from it, whenever they want to start programming.

This guide shall serve as a helpful resource for all the software enthusiasts, for it provides all the know-how that is required for the implementation, and designing of clean architecture.

Technologies Used and Required for the Implementation of Clean Architecture in Android Apps

This list is composed of all the technologies and the software that is required for the implementation of clean architecture in the development of Android Apps.

1. Kotlin

According to Google statistics, Kotlin has been regarded as the “first-class language” for writing and designing Android apps. It is a cross-program platform, which has been statically typed and has been used for general-purpose programming language, along with type inference. It has been designed, keeping in mind that it can interoperate to the fullest with Java.

Ever since the advent of Android Studio version 3.0 in 2007, Kotlin has been included in the lists of standard Java compiler as a better alternative.

2. Dagger 2

The Dagger 2 is a dependency injection framework**,** which is based on the Java Specification Request, or JSR 330. It is based on annotations and uses the code applications and generations. The resultant code that it generates is relatively easier to debug and be read.

It is a fully static App, which makes use of the compile-time dependency injection, and is used on both JAVA and Android.

The annotations allow the users to define and configure dependencies in the development of Android applications.

3. RX Java 2

RX Java, or Reactive Extensions Java, is a programming library that follows the principles of Reactive Programming for the composition of asynchronous and event-based programs by making use of observable sources.

RX Java 2 is gaining extreme levels of popularity in the Android world because of its capability to outmatch the current solutions for writing a composable code, as per the provided guidelines.

The RX Java are extensions which are designed especially for the Java Virtual machine (JVM).

4. Retrofit 2

Retrofit 2 is a client that is a safe HTTP type, which works as a bridge between Java and Android.

It is a REST client which makes retrieval and uploading of JSON, and other types of structured data via a webservice which is REST-based.

Components Used and Required for the Implementation of Clean Architecture in Android Apps

Now comes an overview of the basic terminologies which are used in the implementation of the clean architecture for the designing and development of Android applications.

Here are some of the commonly used components of clean architecture:

1) **Room**

Room is an abstraction layer, which is totally dependent on SQ Lite.
It utilizes the whole power of SQ Lite and uses it to direct more robust database access.

2) **View Model**

View Model is a user-friendly and widely used component which is designed for storing and managing User Interfaces, and its related data in a conscious, life-style type of way. (We too, have seen and incorporated the use of View Model in some of our tutorials in the previous chapters).

3) **Live Data**

Just like the View Model, the Live Data is a life-cycle conscious cycle. This is more life-cycle conscious than View Model in the sense that it is aware of, and also, respects the life cycle of the other involved components in the entire Application software system such as the fragments, activities, and services.

4) **Paging Library**

The Paging Library is a useful component of the clean architecture system, which makes the gradual uploading of data easier, and allows it to be uploaded with grace within the application's Recycler View, taking care of its boundaries and limitations too.

An Overview on the Approach of Android Applications Building through the implementation of Clean Architecture

In this section, we shall see a brief overview of how Clean Architecture is implemented on the building up of Android Apps, and how a programmer is supposed to take care of all the data that goes in each of the individual layers, following its specified boundaries and limitations.

As we all know by now, the focus of our programming is going to be directed towards three main layers. These are represented in the form of circles in the clean architecture system and represents that only these three layers are supposed to be prioritized so that implementation and the coding of programs could be done on this basis only.

These layers (in descending order) are the Presentation Layer, the Data layer, and the Domain layer.

Overview of the Domain Layer

As seen earlier, the domain layer is the innermost or the center of the clean architecture system. It is the most dominant of all layers, as it is fed with the most important and crucial of all the data. The data components that are fed into the domain layer include:

- **Data Transfer Objects (DTO)**: DTOs are the basic, and usual Kotlin data classes.

```
data class Project(

        val id: Long,

        val name: String,

        val date: Date,

        val url: String,

        val categoryList: List<String>,

        val imageUrl: String?,

        val thumbnailUrl: String?,

        val isMatureContent: Boolean,

        val description: String,

        val imageList: List<Image>,

        val ownerList: List<User>,

        val stats: Stats,

        val colorList: List<Int>

    )
```

- **Repository:** a repository is kind of a “storage” interface, which is used for describing the incoming parameters, and give information on the type of object that has been returned. An example is shown below:

```
interface ProjectRepository {


        fun getProjectList(
        page: Int,
        perPage: Int,
        query: String? = null,
        category: String? = null,
        time: String? = null,
        colorHex: String? = null
    ): Single<List<Project>>

    fun getProject(projectId: Long): Flowable<Project>
}
```

- **View Model:** The View Model interacts with Use Cases. It follows the orders of the "Use Case" and ends up producing an object with all the necessary parameters, as described in the Use Case. It also sends a request, which can either end up returning with a desired result or an unexpected error.
- **Use Cases:** The "Use Cases" are supposed to act as an intermediate component or a bridge between the View Model and Repository.

They also can request data from the various repositories. With this data, they can either combine all of it to take the form of another use case or can also use them for further additional calculations.

Use Cases can be built independently, from a great variety of classes, there is no restriction of using the same kind of use cases within a program.
A detailed example of a Use Case implementation is shown below:

1-

```
abstract class UseCase {

        protected var lastDisposable: Disposable? = null
        protected val compositeDisposable = CompositeDisposable()
        fun disposeLast() {
            lastDisposable?.let {
                if (!it.isDisposed) {
                    it.dispose()
                }
            }
        }

        fun dispose() {
            compositeDisposable.clear()
        }
    }
```
2-

```
abstract class SingleUseCase<T, in Params> : UseCase() {

        internal abstract fun buildUseCaseSingle(params: Params): Single<T>

    fun execute(
        onSuccess: ((t: T) -> Unit),
        onError: ((t: Throwable) -> Unit),
        onFinished: () -> Unit = {},
        params: Params
    ) {
        disposeLast()
        lastDisposable = buildUseCaseSingle(params)
            .subscribeOn(Schedulers.io())
            .observeOn(AndroidSchedulers.mainThread())
```

```
                .doAfterTerminate(onFinished)
                .subscribe(onSuccess, onError)
            lastDisposable?.let { compositeDisposable.add(it) }
        }
    }
```

3-

```
class GetProjectListUseCase @Inject constructor(private val projectRepository: ProjectRepository) :
        SingleUseCase<List<Project>, GetProjectListUseCase.Params>() {

        override fun buildUseCaseSingle(params: Params): Single<List<Project>> {
            return with(params) {
                projectRepository.getProjectList(page, perPage, query, category, time, colorHex)
            }
        }

        class Params(
            val page: Int,
            val perPage: Int,
            val query: String? = null,
            val category: String? = null,
            val time: String? = null,
            val colorHex: String? = null
        )
    }
```

Overview of the Data Layer

The Data Layer lies just outside to the Domain layer and is concerned with the handling of Rest Clients, Databases, Adapters, and much more.

- **Implementing Repositories:**

The implementation of repositories is important because doing so provides the programmer with access to the data, along with allowing him to select a data source as per his need, and according to the conditions of the software in progress.

```
class ProjectRepositoryImpl @Inject constructor(
        private val database: AppDatabase,
        restClient: RestClient
     ) :
        ProjectRepository {
        private val projectDao = database.projectDao()
        private val userDao = database.userDao()
        private val projectUserJoinDao = database.projectUserJoinDao()
                    private val projectService =
restClient.retrofit.create(ProjectService::class.java)
        override fun getProjectList(
           page: Int,
           perPage: Int,
           query: String?,
           category: String?,
           time: String?,
           colorHex: String?
        ): Single<List<Project>> {
              return projectService.getProjectList(page, perPage, query,
category, time, colorHex)
                .map { response ->
                    val data = response.mapToEntityList()
                    val projectEntityList = mutableListOf<ProjectEntity>()
                    val userEntityList = mutableListOf<UserEntity>()
                    val projectUserJoinEntityList =
mutableListOf<ProjectUserJoinEntity>()
                    data.forEach { project ->
                        projectEntityList.add(ProjectAdapter.toStorage(project))
                        val userEntitySubList = project.ownerList.map {
UserAdapter.toStorage(it) }
                        userEntityList.addAll(userEntitySubList)
                        val projectUserJoinEntitySubList =
                                                    userEntitySubList.map   {
ProjectUserJoinEntity(project.id, it.id) }
                        projectUserJoinEntityList.addAll(projectUserJoinEntitySub
                    }
                    database.runInTransaction {
                        projectDao.saveProjectList(projectEntityList)
                        userDao.saveUserList(userEntityList)
                        projectUserJoinDao.saveProjectUserJoinList(projectUserJo
                    }
                    data
                }
            }

            override fun getProject(projectId: Long): Flowable<Project> {
```

```
            val local = Flowable.combineLatest(
                projectDao.getProject(projectId),
                projectUserJoinDao.getUserListForProject(projectId),
                BiFunction<ProjectEntity, List<UserEntity>, Project> {
projectEntity, userEntityList ->
                    ProjectAdapter.fromStorage(projectEntity, userEntityList)
                }
            )
                .distinctUntilChanged { oldData, newData -> oldData.id ==
newData.id }
            val remote = projectService.getProjectById(projectId)
                .map {
                    val project = it.mapToEntity()
                    projectDao.saveProject(ProjectAdapter.toStorage(project))
                    project
                }
                .toFlowable()
            return Flowable.merge(local, remote)
                .distinctUntilChanged()
        }
    }
```

Adapters: Adapters are mainly present to convert the entities which were used by the databases and network clients, back to their older version that existed in the Domain layer, and vice versa. While doing so, there are a whole lot of pros and cons that the adapters face, such as:

Data changes in one layer are limited to one layer only; they don't affect any other layer

There is a possibility that multiple duplications can take place, and end up making a system even more cluttered up that in originally was

While making changes in the data, the mapper needs to be changed as well

Annotations, and third party dependencies, both of which are required for a library, do not get implemented into the other layers, and have no influence on them whatsoever.

```
object ProjectAdapter {

        fun toStorage(adaptee: Project) =
            ProjectEntity(
                adaptee.id,
                adaptee.name,
                adaptee.date,
                adaptee.url,
                adaptee.categoryList,
                adaptee.imageUrl,
                adaptee.thumbnailUrl,
                adaptee.isMatureContent,
                adaptee.description,
                adaptee.stats,
                adaptee.colorList
            )

        fun fromStorage(adaptee: ProjectEntity, userEntityList:
List<UserEntity> = listOf()) = Project(
            adaptee.id,
            adaptee.name,
            adaptee.date,
            adaptee.url,
            adaptee.categoryList,
            adaptee.imageUrl,
            adaptee.thumbnailUrl,
            adaptee.isMatureContent,
            adaptee.description,
            listOf(),
            userEntityList.map { UserAdapter.fromStorage(it) },
            adaptee.stats,
            adaptee.colorList
        )
    }
```

Overview of the Presentation Layer

The presentation is the outermost, or final layer of our clean architecture system. This final layer consists of View Models and the User Interfaces. It is in this layer that we connect all the dependencies from the other layers, to present a final shape, or "Architecture" of our proposed software, which has been built through prioritizing all of its concerns at their individual levels.

A series of implementations being done at the level of the presentation layer is shown below:

1-

```
class MainApplication : Application(), HasActivityInjector {

        @Inject
        lateinit var activityInjector: DispatchingAndroidInjector<Activity>

        override fun onCreate() {
            super.onCreate()

            DaggerApplicationComponent
```

```
                .builder()
                .application(this)
                .build()
                .inject(this)
        }

        override fun activityInjector() = activityInjector
    }
```
2-
```
@Singleton
    @Component(
        modules = [
            AndroidInjectionModule::class,
            ApplicationModule::class,
            ActivityModule::class,
            RepositoryModule::class
        ]
    )
    interface ApplicationComponent {

        @Component.Builder
        interface Builder {

            @BindsInstance
            fun application(application: Application): Builder

            fun build(): ApplicationComponent
        }
```

```
            fun inject(application: MainApplication)
        }

3-
@Module(includes = [AndroidSupportInjectionModule::class])
        interface ActivityModule {

            @ActivityScope
            @ContributesAndroidInjector()
            fun splashActivityInjector(): SplashActivity
            @ActivityScope
            @ContributesAndroidInjector()
            fun projectListActivityInjector(): ProjectListActivity

            @ActivityScope
            @ContributesAndroidInjector()
            fun projectDetailsActivityInjector(): ProjectDetailsActivity
        }
4-
@Module
        class ApplicationModule {

            @Singleton
            @Provides
            fun provideContext(application: Application): Context =
application.applicationContext

            @Singleton
            @Provides
            fun provideSharedPreferences(context: Context): SharedPreferences =
                PreferenceManager.getDefaultSharedPreferences(context)

            @Singleton
            @Provides
            fun provideRestClient() = RestClient()

            @Singleton
            @Provides
            fun provideDatabase(context: Context) = AppDatabase.getInstance(context)
        }
```

5-

```
@Module
interface RepositoryModule {


    @Binds
    @Singleton
    fun bindCategoryRepository(categoryRepository: CategoryRepositoryImpl): CategoryRepository


    @Binds
    @Singleton
    fun bindProjectRepository(projectRepository: ProjectRepositoryImpl): ProjectRepository
}
```

You all must be aware that, while using Use Cases, the probability of memory leaking into the other layers, and also, to the external sources is very high. This could have disastrous effects on the entire software, and prove to be a very worst-case scenario for the programmer or coder.

To avoid this big problem, experts suggest unsubscribing from the Use Cases, when using them.

And unsubscribing from them is our main motive for which we use the "View Model" layer because it offers us simplified processing of the life cycle, and itself has a better understanding for the protection of privacy of the individual layers, and the confidential data contained within them.

For an even more convenient and simpler way out, we shall prefer creating a "Basic View Model Class," for making things easier for us. Creating this class will also enable us to create another subclass out of it, which we can use and will use to direct all the Use Cases towards the base constructor. And after doing all of this, we can proceed to unsubscribe from all the Use Cases, which are present in the View Model layer.

A graphical, and coding illustration will be more appropriate to insert here, for better understanding of the entire scenario taking place in the above paragraph:

1-

```
abstract class BaseViewModel(vararg useCases: UseCase) : ViewModel() {

        protected var useCaseList: MutableList<UseCase> = mutableListOf()

    init {
        useCaseList.addAll(useCases)
    }


    override fun onCleared() {
        super.onCleared()
        useCaseList.forEach { it.dispose() }
```

```
            }
        }
```

2-

```
class SplashViewModel(
            private val getCategoryListUseCase: GetCategoryListUseCase
        ) : BaseViewModel(getCategoryListUseCase) {
                        private val _categoryListReceivedLiveData = MutableLiveData<Unit>()
            val categoryListReceivedLiveData: LiveData<Unit>
               get() = _categoryListReceivedLiveData

            fun requestCategoryList() {
               getCategoryListUseCase.execute(
                  onSuccess = { _categoryListReceivedLiveData.value = Unit },
                  onError = {},
                  params = Unit
               )
            }
        }
```

Now, comes another scenario into play, and that being, the incorporation, or the introduction of the dependencies into the View Model layer.

For doing this, the programmer, or the developer needs to make a “View Model Factory,” which will help us in getting all the required dependencies in the constructor, using the help and services of the “Inject” annotation. After this is done, then this newly created model will be able to create the required “View Model,” using these dependencies in the “create ()” function, as shown below:

```
class SplashViewModelFactory @Inject constructor(
    private val getCategoryListUseCase: GetCategoryListUseCase
) :
    ViewModelProvider.Factory {

    @Suppress("UNCHECKED_CAST")
    override fun <T : ViewModel> create(modelClass: Class<T>): T {
        if (modelClass.isAssignableFrom(SplashViewModel::class.java))
{
            return SplashViewModel(getCategoryListUseCase) as T
        }
        throw IllegalArgumentException("Unknown ViewModel class")
    }
}
```

And this way, taking the help of the already existing View Model Factory pattern, which was made in Activity with DI, we finally create the “View Model.” Once it is successfully created, we can request and observe all the data that has been coming into the “View Model.”

```
class SplashActivity : DaggerAppCompatActivity() {
    @Inject
    lateinit var splashViewModelFactory: SplashViewModelFactory
    private lateinit var viewModel: SplashViewModel
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_splash)
        setupViewModel()
    }

    private fun setupViewModel() {
        viewModel =
                                    ViewModelProviders.of(this,
splashViewModelFactory).get(SplashViewModel::class.java)
        viewModel.categoryListReceivedLiveData.observe(this, Observer
{
                showNextActivity()
        })
        viewModel.requestCategoryList()
    }
    private fun showNextActivity() {
        startActivity(ProjectListActivity.getStartIntent(this))
    }
}
```

Paging Library Using Clean Architecture

If you are a programmer or a developer who likes to try out new and innovative ideas every now and then, then you must have also thought of trying out the "Paging Library" feature, along with clean architecture, together in developing an Android Application.

If you have thought of doing so, then you must also be aware of the fact that you will face problems ahead once you start doing so, and that is because either one of them uses their own individual data sources.

And since it is next to impossible to make a request to the network, or the existing database in the Presentation Layer, we will just resort to using the data sources as specified in the Data Layer, taking the help of the "View Model."

And for doing so to achieve the desired results, we shall simply delegate all the requests from the Paging Library in to the View Model, like this:

1-

```
class ProjectListAdapter(diffCallback: ProjectDiffCallback) :
}

            PagedListAdapter<Project, ProjectListAdapter.ProjectViewHolder>
(diffCallback) {


        var onProjectItemClickListener:
ProjectItem.OnProjectItemClickListener? = null

        override fun onCreateViewHolder(parent: ViewGroup, position: Int)
=
                ProjectViewHolder(ProjectItem(parent.context),
onProjectItemClickListener)

        override fun onBindViewHolder(viewHolder: ProjectViewHolder,
position: Int) {
            val data = getItem(position)
            val itemView = viewHolder.itemView as? ProjectItem
            if (data != null && itemView != null) {
                itemView.bind(data)
            }
        }

        class ProjectViewHolder(
            itemView: ProjectItem,
                onProjectItemClickListener:
ProjectItem.OnProjectItemClickListener?
        ) : RecyclerView.ViewHolder(itemView) {

            init {
                itemView.onProjectItemClickListener =
onProjectItemClickListener
            }
            }
        }
```

2-

```
class ProjectDiffCallback : DiffUtil.ItemCallback<Project>() {
            override fun areItemsTheSame(oldItem: Project, newItem: Project) =
oldItem.id == newItem.id
                 override fun areContentsTheSame(oldItem: Project, newItem:
Project) = oldItem == newItem
```

3-

```
class ProjectDataSource(
            private val dataSourceDelegate: DataSourceDelegate<Project>
        ) : PositionalDataSource<Project>() {
            @MainThread
                 override fun loadInitial(params: LoadInitialParams, callback:
LoadInitialCallback<Project>) {
                dataSourceDelegate.requestPageData(
                    params.requestedStartPosition,
                    params.requestedLoadSize
                ) {
                    callback.onResult(it, 0)
                }
            }
            @MainThread
                 override fun loadRange(params: LoadRangeParams, callback:
LoadRangeCallback<Project>) {
                        dataSourceDelegate.requestPageData(params.startPosition,
params.loadSize) {
```

```
                callback.onResult(it)
              }
            }
          }
        }
```

4-

```
interface DataSourceDelegate<T> {

    fun requestPageData(startPosition: Int, loadSize: Int, onResult: (List<T>) -> Unit)
  }
```

But we have been using the implementations of code in such a way that we do not need the Paging Library to request for data in an entirely new thread. Because we have the use Cases to take care of this matter, we create yet another new thread. That is the “Main Thread Executor” which is supposed to be used for the “set Background Thread Executor” function, and also, for the “set Main Thread Executor” function, in the following way:

1-

```
class MainThreadExecutor : Executor {
    private val mHandler = Handler(Looper.getMainLooper())
    override fun execute(command: Runnable) {
      mHandler.post(command)
    }
  }
```

2-

```
class ProjectListViewModel(private val getProjectListUseCase: GetProjectListUseCase) :
    BaseViewModel(getProjectListUseCase),
    DataSourceDelegate<Project> {

    override fun requestPageData(
        startPosition: Int,
        loadSize: Int,
        onResult: (List<Project>) -> Unit
    ) {
        val page = (if (startPosition == 0) 20 else startPosition) / loadSize
        getProjectListUseCase.execute(
            onSuccess = {
                onResult(it)
            },
            onError = { it.printStackTrace() },
            params = GetProjectListUseCase.Params(page, loadSize)
        )
    }
}
```

One thing that is to be kept in your mind while programming this way is that the Use Cases which have been mentioned above can be summoned countless times to be executed. Like for instance, while a user is entering a phrase for searching purposes on the screen, a simultaneous request would be sent to the network for every character that is entered in and short debounce. Also, since requests are made asynchronously, the result of the first request may come in last, and vice versa. This can eventually cause discrepancies, and ultimate inconsistencies to arise, because there would apparently be a glitch present between the data found, and the text that was actually entered.

To solve this problem, we can always try to debounce the request time but must be aware of the fact that this, too, can create problems for the user and debunk his experience.

Luckily, we are already aware of the solution that will work best in this case scenario. As we have discussed above, the best way out to avoid these errors could be found by making the un-subscription from the previous request necessary, when the user is making a new request. This will provide the new request to be met with a clean and new use case and will definitely avoid the mixing up of the requests, and results would be simultaneous too, in the end.

Also, you must have noticed that when we are using RX Java, and Single Repository patterns together, a problem with error processing is presented to us almost always. In case such an error appears in one of the requests being made, or the ones which have already been made, and are awaiting results, we might see the data no arriving within the specified time, and the Use Case may present with a subsequent error as well. To avoid this error, a smart programmer can make use of the RX Java “Merge Delay Error Method” instead of using the traditional merge methods. By applying this technique, we shall observe that even though changes are being made and observed in the database, yet we still are not getting any error at all. This is all because the “on Complete” function is not called at all.

To solve this problem once, and for all, separate a given Use Case into two. The first one should be assigned the task to command the repository for making a new request and save the subsequent data that it receives in return, in the database. The second part of the split Use Case should be assigned the task to summon the required and specified data from the first request that was made, all the while closely monitoring the changes being made in the database. In this way, you shall be able to notice the activity taking place within the database at every step and process the errors, if any of them arise, at any step of the processing in the database.

Chapter 16: Clean Architecture: The Other Side of the Picture: AKA: Drawbacks!

This chapter focuses on:

- The drawbacks of the application of clean architecture.

We have seen enough examples and practical applications of the clean architecture system. So far, we have been able to see how easy and simple the application of the clean architecture system has been making our programs, but still, with so many pros, come so many cons as well.

In this chapter, we will dive deep into the clean architecture basics, and try to come up with logical arguments as to why the application of clean architecture principles could be a problematic cause for all of us.

1. **Sacrificing Simplicity**

With the concept that it has fed everyone, that it will require simply input to give simple results, this might itself give rise to problems. As at times, we do not always need simple programs.

In such situations where we need to design a complex software system, we will eventually have to let go of the clean architecture system, because implementing clean architecture to a complex program means using countless "use cases" which isn't possible at all times and can further complicate things for a programmer too.

2. **Repetitions**

A programmer or a software enthusiast is always on the lookout for new opportunities to do better at what they love doing: coding and programming. Through clean architecture, many programmers have complained, saying that all their time is consumed in setting up and designing layers, and layers, and layers. This repetition is indeed a boring task for them, but they can't skip it as well, as no work could be done without implementing the layers in a software system. So, this repetitive work is often boring and tedious for the programmers.

3. **Restrictions**

Undoubtedly, every work comes along with its share of boundaries and limitations. But with clean architecture, these boundaries could be more than actually required, because the kind of restrictions that clean architecture has to offer are not always logical ones. That, and the fact that these restrictions need to be overcome by actually working on them, is what might turn the application of clean architecture into a forced work, something that nobody looks forward to.

4. Over Working, Over Engineering, and Over Use!

Everyone is aware of how the clean architecture was invented keeping large scale, big sized applications in mind. But with the need to create a great number of files for every step, no matter how big or small the entire software system is going to be, this seems to be an unwanted task that drags and delays the work for no reason.

5. Increasing Complexity

Clean architecture, overall, makes things a lot more difficult and complex for people to implement it. In fact, it is its attempts at making things easier for the development of a program that each step by step guide must be followed through, but it always ends up making things a lot more differently carried out, which is not always welcomed and wanted.

6. Difficult to Learn and Implement

Many of the senior and experienced programmers have also been reported to have said that this clean architecture is not for everyone, as it requires a lot of practice and experience for one to master its art. So, as much as it seems to be a piece of cake and a really simple option, it can prove to be quite overwhelming and tricky for the amateur programmers and coding freaks.

Bonus Chapter: Clean Architecture Interview Questions

Architects are software experts, responsible for making some pretty high-level choices in terms of design. They also dictate some of the technical standards, which includes the tools used, the platforms, and the coding standards for the software. Architecture is referring to an application or system’s high-level structures, how those structures are created, and what documentation is required.

If you are thinking about turning to clean architecture as a career choice, these are some of the questions you may be asked at interview:

Question One - Explain what ‘program to interfaces, not implementations’ means.

The definition of ‘coding against interface’ says that there will always be a factory-supplied interface object in the client code. Instances that the factory returns are of the interface type, and these are implemented by candidate classes. In this way, the client program doesn’t have to concern itself with implementation, and the operations to be done are determined by the interface signature.

We can use this approach to change how a program behaves at run-time. And there is the added benefit that it helps programmers write programs that are much easier to maintain.

Question Two - Explain the difference between continuous integration, continuous delivery, and continuous deployment.

When a developer does continuous integration, they are continuously merging any changes they make back into the primary branch. By doing this, they can avoid the issues caused when they wait until release day to get their changes merged back.

Continuous delivery extends from continuous integration and is used for ensuring that new changes can quickly and easily be released to customers in a more sustainable way. What this means is that, as well as your testing being automated, your release process is also automated, and your application may be deployed simply by clicking a button when required.

With continuous deployment, we move on a step from continuous delivery. Each change that successfully goes through the production pipeline can then be released to the customer. There is no need for any human intervention, and the only thing that will stop changes from going to production would be a test that fails.

Question Three - Explain what SOLID stands for and what the principles are.

SOLID stands for the first five OOD (object-oriented design) principles that Robert C Martin devised. They are:

- **S –** SRP – the Single Responsibility Principle. This states that classes should only change for one reason and that means that every class has just one job
- **O –** OCP – the Open-Closed Principle. This states that an entity or an object should be left open to being extended but should be closed to any modifications
- **L –** LSP – the Liskov Substitution Principle. This states that every child or derived class should be substitutable for their parent or base class
- **I –** ISP – the Interface Segregation Principle. This states that clients shouldn't be forced into implementing any interface they don't need to use, not should they be forced into a dependency on methods they have no need of.
- **D –** DIP – the Dependency Inversion Principle. This states that there should be no dependency on concretions; the dependency should be on abstractions instead. It also states that high-level modules shouldn't have a dependency on low-level modules, but they should have a dependency on abstractions.

Question Four – Explain what a system's BASE property is.

A BASE property is a common property of a NoSQL database that has evolved. The CAP theorem states that BASE systems are no guarantee of consistency. BASE is an acronym for the following system property as per the CAP theorem:

- **Basically Available –** this is an indication that a system is guaranteed as being available

- **Soft-state –** this is an indication that the system state could change, even if there is no input; the primary reason for this is the ‘eventually consistent model.’
- **Eventual consistency –** this indicates that a system will develop consistency over the course of time, given that there is no input to the system throughout that time

Question Five – Explain the CAP Theorem.

Eric Brewer was responsible for publishing the CAP Theorem for Distributed Computing, and the theorem states that distributed systems cannot provide these three guarantees at the same time:

- **Consistency –** when every node sees the same data simultaneously with concurrent updates
- **Availability –** this guarantee states that all requests will get a success or failure response
- **Partition tolerance –** a guarantee that, in spite of part of the system failing or arbitrary message loss, the system will carry on operating

CAP is correspondent to those guarantees, and this is the theorem that today's approach to distributed computing is based on. The theorem is used by some of the largest companies in the world, receiving the largest volumes of traffic, such as Facebook, Google, and Amazon, as the basis for how their architecture is determined. The most important thing to understand is that a system can only guarantee to meet two out of the three conditions.

Question Six: What are the Twelve-Factor App Principles?

Twelve-factor app is a methodology used to build SaaS applications – Software as a Service. The methodology is a set of best practices aimed at enabling an application to be built with resilience and portability when it is deployed to the internet:

1. **Codebase –** every deployed service should have a single codebase, and this can be used for multiple deployments
2. **Dependencies –** every dependency must be declared, and there must be no implicit reliance on any libraries or system tools
3. **Config –** if a configuration is different between each deployment,

it must be stored within the environment

4. **Backing Services –** every backing service must be treated as if it were an attached resource. The execution environment must attach it and detach it
5. **Build, Release, Run –** these should be the only three things in a delivery pipeline, nothing else
6. **Processes –** an application must be deployed as at least one stateless process, with a backing service used for storing persisted data
7. **Port Binding –** all services that are self-contained should be available, using specified ports, to other services
8. **Concurrency –** when individual processes are scaled, concurrency is advocated
9. **Disposability –** where you have a resilient, robust system, fast startup, and fast shutdown are advocated
10. **Dev/Prod Parity –** every environment should be as similar as they can be
11. **Logs –** every application should produce an event stream in the form of a log, leaving the execution environment to aggregate
12. **Admin Processes –** all required admin tasks must be retained in source control, and they should be packaged together with the application

Question Seven – Explain Heuristic Exceptions.

Heuristic expressions reference the decision taken by a transaction participant to take action unilaterally without requiring consent from the transaction manager. This will normally be with the result of a catastrophic failure occurring between the transaction manager and the participant.

Communication failures can occur in distributed environments. If it isn't possible for communication to take place between recoverable resources and transaction managers for a long period of time, the recoverable resource may take the decision to commit or rollback the changes in a transactional context unilaterally. A decision like this may be called a heuristic decision, and it is one of the very worst errors. It can result in some parts of a transaction becoming committed and other parts rolling back; as such, the atomic property of the transaction is violated, and this could lead to a corruption in the data integrity.

Because heuristic exceptions are dangerous, where a recoverable resource makes one, it must keep all the relevant information about the decision in a stable storage solution until it is told, by the transaction manager, that it can forget about it. What data is stored about the decision is dependent on what type the recoverable resource is, and it is not classed as standardized. The idea here is that the data can be examined by the system manager and, where required, edited to ensure data integrity issues are corrected.

Question Eight – Explain Shared Nothing Architecture and Discuss How it Scales.

SN, or Shared Nothing architecture, is described as an approach to distributed computing where every node is both self-sufficient and independent. No point of contention is required across the whole system.

- The nodes do not share any resources – No Shared memory, and No Shared file storage
- Each node can work in its own, independent of the other nodes where work is concerned
- If one node fails, it will only affect those that use the node; the remaining nodes will continue working with no disruption.

This is a scalable approach because it gets around the existence of a system bottleneck. SN has become quite popular in recent times, especially in terms of web development, because it is linearly scalable. The theory is that an SN system may be infinitely (almost) scaled just by adding inexpensive machines as nodes.

Question Nine – Explain What Eventually Consistent Means.

Relational databases have a property of Strict Consistency, and, unlike this, the Eventual Consistency property of any system makes sure that transactions won't take a database to one valid state from another, at least not straight away. What this means is that it may have inconsistent intermediate states between several nodes.

A system that eventually consistent is useful when it isn't critical to use absolute consistency. Take a status update on Twitter; for example, it won't exactly be devastating to a system if some users fail to see a recent update for a user.

A system that is eventually consistent may not be used for any use case that requires strict or absolute consistency. An example is a transactional system for a bank; this cannot use eventual consistency because the system must be consistent in the state of any transaction at any given point. An account balance should never show different balances from different ATMs.

Conclusion: The Book in a Nutshell

Now heading towards the end, we shall see how the book has helped us in instilling some really important knowledge of a phenomenal, and theoretical invention of our very own **Uncle Bob**. It was his keen knowledge and expertise that helped him shape up a revolutionary concept of the “clean architecture” system.

We are not going into all the details now, as we all are very much well aware of the facts and figures, and the ins and outs of clean architecture.

The clean architecture system was simply proposed because Uncle Bob had seen how the other architecture systems were failing at prioritizing what was important for designing a software, or an application, and what was not. He was very much concerned that maybe doing so would lead to drastic results in the future of software and programming, so he immediately came up with bigger and better plans of his own.

And so, within a subdivision of the layered (or n-tiered) architecture system, and inspired from the onion layered architecture system, and the hexagonal architecture system, came yet another new system, by the name of the **“clean architecture system.”** This system was based on the concept that it will **separate all the concerns** involved in the designing of a new software program and shall **prioritize all those concerns** in an ascending manner.

The advent of clean architecture proved to be really helpful in the field of software development and programming, and all the coders and programming enthusiasts were really happy to finally get their hands on something that was making programming more fun than it already is!

The benefits that clean architecture provides are great enough to encompass all the cons that it comes along with as well.

The clean architecture system has this amazing ability to let the programmer **test it numerous times**, to assure its proper functioning, and never does it limit the times that it could be tested. This is an advantage, greatly beneficial for all the amateur coders and programmers, who simply want to tests their skills, or want to come up with a trial and error version of their own application. This makes the system **flexible** and allows it to be changed as many times as the programmer wants it to change without creating a mess of errors, which is quite frustrating for both the application and the programmer.

Also, because of the separation of individual concerns in individual layers, we also observe that there is almost zero to **none issues of decoupling**, which is actually an entanglement of all the work present in the architecture system. Instead of queuing or piling up, the data is all over the places, and that creates an atmosphere of disorganization and decoupling within an architecture system. This brings up to yet another advantage of the clean architecture system, and that being, the presentation of all the data in an **organized, hassle-free form**, which makes any programmer or coder easily understand whatever the program is going to be about, only by looking at the details presented superficially. These advantages are exactly why everyone wanted to implement and develop Apps using the application of a clean architecture system.

Although not in majority, but clean architecture also has some cons along with it. But all these cons are mostly faced due to lack of experience, or improper practices, and can be overcome easily by practicing even harder and becoming a pro at it. However, some cons are unavoidable and have to be faced one way or the other. The best way out to avoid such problems from arising is to come up with even better and innovative solutions, that every problem, whether spontaneous or a long overdue one, should be dealt with in a very smart way, that it gets resolved as soon as it aroused.

One of the main downsides of using clean architecture is its **complexity**. The clean architecture system was invented, keeping large scale, and the extensive business demands in mind. It will obviously be a problem for those businesses and applications which require low complexity, and apps to be built within a short period of time. The complexity exists only because there cannot be a chance of “trial and error” at least when creating Apps for a bigger scale setup. Also, there have been many complaints from the amateur programmers that they have always wanted to try this architecture system out, but due to it **requiring skilled hands** to run it, they often are faced with failures. So, it does actually mean that the clean architecture system is not that easy to implement at all, but it **requires expertise** and professionalism; it runs it in a better and sophisticated way.

Out of the many assumptions created about the advent of this clean architecture system, the authentic one is that the clean architecture system is actually based on the **SOLID principles**, with the “D” of the SOLID holding phenomenal value. These five principles, namely, the **Single Responsibility**

Principle (SRP), the Open and Closed Principle, the Liskov Substitution Principle, Interface Segregation Principle, and the Dependency Inversion Principle, form the core value of the clean architecture system. The SOLID principles help in building up an easy to understand, and clearly designed program that is tailor-made to be flexible and easily maintainable by anyone who uses it. The **"D" of the SOLID holds great importance** in clean architecture because it is actually the same principle that laid down the foundation for the introduction of layers to this system. It states that **high-level modules are in no way dependent on the modules of a lower level, and every single thing involved in the software must depend on abstractions, which also have the details dependent upon them**. This made the rest of the rules clear enough.

The clean architecture, under certain limitations, and restriction also **allows its boundaries to be crossed**, but only in those situations where doing so is the need of the time. It, at any cost, prevents excuses and reasons that could interfere with its proper functioning.

We also learned the there are **three main layers** in a clean architecture system, **the data layer, the domain layer, and the presentation and frameworks layer**. We learned what kind of information or details go into each of these layers and how they are further taken care of for their execution at a later time. We also learned to implement coding through different tutorials on Android applications, and it is expected that by the time readers reach the end of the book, they are already ready and geared up with ideas of their own to start programming and building up Apps. Of their own!

Moreover, with the great number of examples, and practical applications given in the book, we expect our readers to come up with coding and programming implementation codes of themselves and create programs that are revolutionary and groundbreaking in the field of software programming.

Resources

Chapter 15
rubygarage.org