Clean architecture y Programación Reactiva.

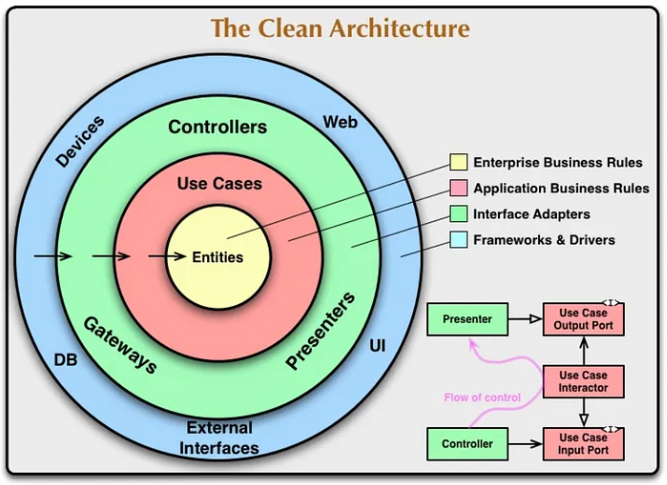
**CLEAN ARCHITECTURE**

**Why do we need to architect?**

“The goal of software architecture is to minimize the human resources required to build and maintain the required system.” *― Robert C. Martin, Clean Architecture.*

Advantages of Proper Architecture:

* Testable
* Maintainable
* Changeable
* Easy to Develop
* Easy to Deploy
* Independent



Each circle represents different areas of the software. The outermost layer is the lowest level of the software and as we move in deeper, the level will be higher. In general, as we move in deeper, the layer is less prone to change.

**The Dependency Rule**

The Dependency Rule states that the source code dependencies can only point inwards.

This means nothing in an inner circle can know anything at all about something in an outer circle. i.e. the inner circle shouldn’t depend on anything in the outer circle. The Black arrows represented in the diagram show the dependency rule.

Diagrama

Descripción generada automáticamente

Remember, the arrow should be read as “depend on”. Frameworks and Drivers should depend on Interface Adapters, which depend on Application Business Rules which depend on Enterprise Business Rules.

Nothing in the bottom layer should depend on the top layer.

**Frameworks and Drivers**

Software areas that reside inside this layer are

* User Interface
* Database
* External Interfaces (eg: Native platform API)
* Web (eg: Network Request)
* Devices (eg: Printers and Scanners)

**Interface Adapters**

This layer holds

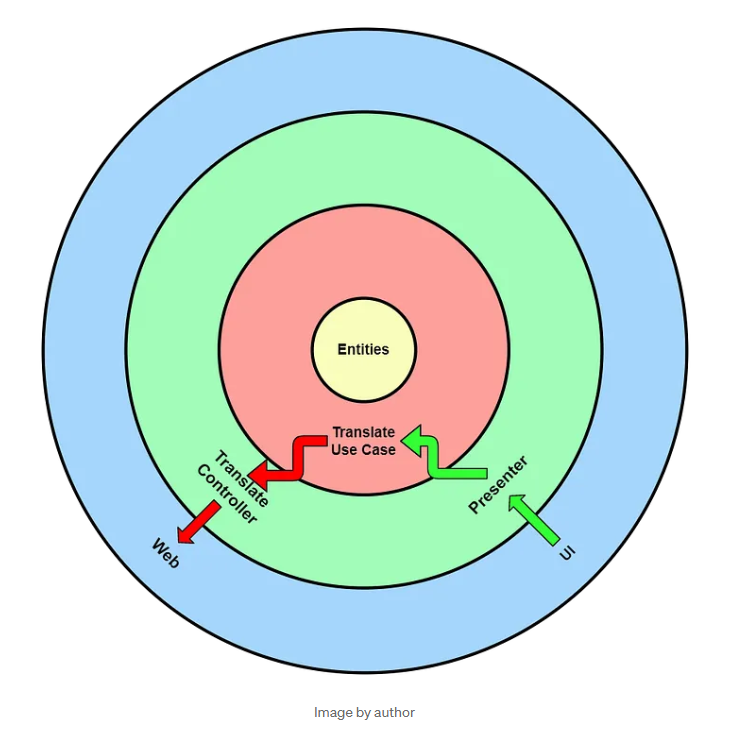
* Presenters (UI Logic, States)
* Controllers (UI Logic, States)(Interface that holds methods needed by the application which is implemented by Web, Devices or External Interfaces)
* Gateways (Interface that holds every CRUD operation performed by the application, implemented by DB)

**Application Business Rules**

Rules which are not Core-business-rules but essential for this particular application come under this. This layer holds Use Cases**.**As the name suggests, it should provide every use case of the application. i.e. it holds each and every functionality provided by the application.

Also, this is the layer that determines which Controller / Gateway to be called for the particular use case. Sometimes we need controllers from different modules.

Also, this is the layer that determines which Controller / Gateway to be called for the particular use case. Sometimes we need controllers from different modules.

Example wrong:  


**Enterprise Business Rules**

This is the layer that holds core-business rules or domain-specific business rules. Also, this layer is the least prone to change.

Change in any outer layer doesn’t affect this layer. Since Business Rules won’t change often, the change in this layer is very rare. This layer holds Entities.

Diagrama

Descripción generada automáticamente

*After all, how can we expect the*web*to throw some data to**the*Controller*without the*Controller*being dependent on it? Also, how can we expect the*Use Case*to get the proper data from the*Controller*without depending on it?*

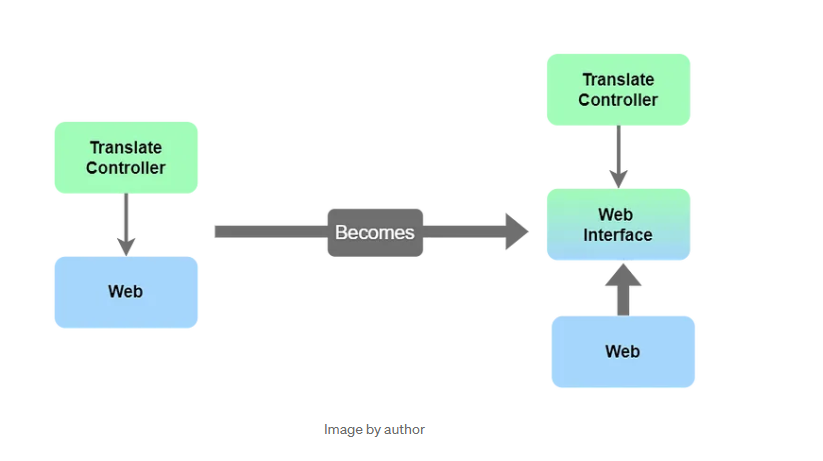
*But the Dependency Rule strictly says dependencies can only point inwards. It adds up by saying this is the rule that makes the architecture work.*

In order to pass this rule, we need to invert the arrow to the opposite direction. Is that possible? Here comes Polymorphism**.**When we include some Polymorphism here, something magic happens.

Simply by having an Interfacebetween these 2 layers, we could invert the dependency. This is known as The Dependency Inversion Principle.

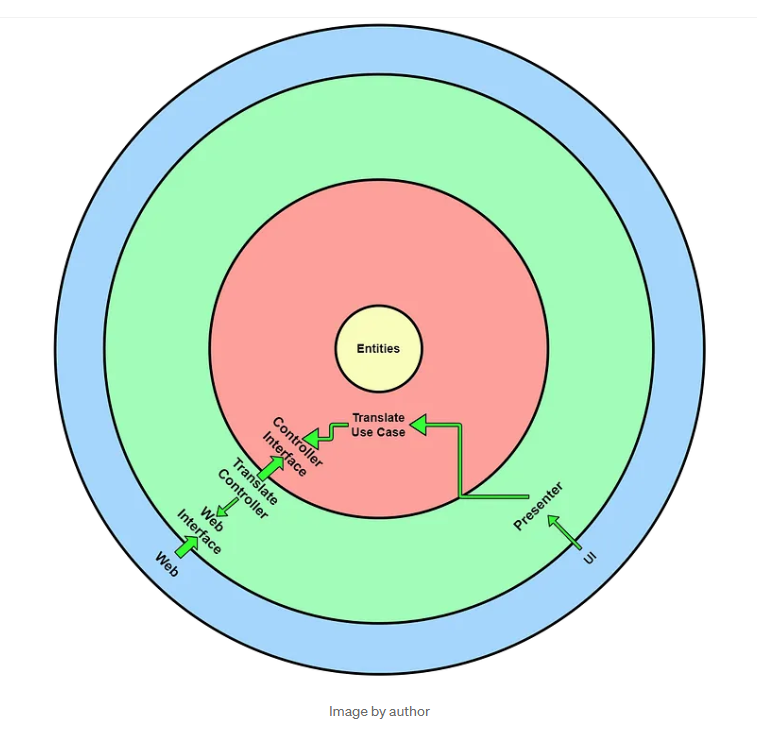
Let’s implement the Dependency Inversion Principle in the cases where the Dependency Rule is violated.

RIGTH WAY:



Diagrama

Descripción generada automáticamente



Imagine you’re in a hotel. We want the hotel to serve us what we want, but not what they offer right?. The same thing is happening here, we want the DB to give the data the application needs but not the data it has.

Also, the single-way dependency rule saves the application from the deadlock state. i.e. imagine in a 2 layer architecture, the first layer depends on the second layer, and the second layer depends on the first layer. In such a case, If we need to change anything in the first layer, it breaks the second layer. If we need to change anything in the second layer, it breaks the first layer. This can be rejected by following the deadlock state.

Application orders what data it wants and it doesn’t care how DB or API prepares the data. This way, the application doesn’t depend on DB or API. If we need/want to change the DB or API Schema in the future, we can simply change it. As far as it gives what the application asks for, the application doesn’t even know the change in DB or API.

Also, the single-way dependency rule saves the application from the deadlock state. i.e. imagine in a 2 layer architecture, the first layer depends on the second layer, and the second layer depends on the first layer. In such a case, If we need to change anything in the first layer, it breaks the second layer. If we need to change anything in the second layer, it breaks the first layer. This can be rejected by following the deadlock state.

Source: [The Clean Architecture — Beginner’s Guide | by Bharath | Better Programming](https://betterprogramming.pub/the-clean-architecture-beginners-guide-e4b7058c1165)

**Reactive Programming:**

Patrón **de diseño Observer**: necesidad de notificar cambios.

Analogía del canal de Youtube: un publicador (canal) y los observadores (suscriptores). El canal necesita que todos los suscriptores noten cuando él sube un nuevo vídeo a su canal de Youtube. Se necesita un método para registrar sus observadores.

Diagrama

Descripción generada automáticamente

**What’s reactive programming?**

Reactive programming is an approach to handling asynchronous and event-based programming that has gained popularity in recent years due to the rise of real-time data-intensive applications. The reactive programming model allows developers to build more efficient, scalable, and resilient systems.

1. Reactive programming is a design approach that uses asynchronous programming logic to handle real-time adjustments. So the core of reactive programming is a data stream that we can observe and react to, even apply back pressure as well. In plain terms, reactive programming is about non-blocking applications that are asynchronous and event-driven and require a small number of threads to scale.
2. A key aspect is to understand the concept of backpressure, which is a mechanism just to ensure producers don’t overburden consumers

**Why is a need for Reactive Programming?**

Several modern applications today require the ability to handle multiple concurrent or simultaneous requests. Therefore, traditional methods are inadequate to handle these operations.

Reactive programming on the server side allows web applications or server-side applications to perform and scale more efficiently. By utilizing this structure, server-side applications can handle multiple requests asynchronously, improving performance, increasing scalability, and handling high-user traffic.

Diagrama

Descripción generada automáticamente

The Publisher creates an event or message for every result obtained, and it can send the Subscriber (n) numbers of values by calling onNext, but the important thing to keep in mind here is that it can also signal an error by calling onError, which can terminate the sequence, while if onComplete is called, the sequence is terminated by default.

In Reactor, Flux, and Mono, there are two primary primitive publishers.

**Flux:** A Flux Publisher in Reactor publishes data or event and can produce from 0 to N values asynchronously.

**Mono:** There is only one value that a Mono publisher can produce, or perhaps none at all.

**Reactive framework support in Spring with Spring Web Flux**

1. Spring Web Flux is a reactive programming model introduced by Pivotal in Spring 5.
2. It provides an asynchronous, non-blocking, and event-driven architecture for building web applications that are resilient and responsive.
3. It enables developers to build applications that can handle high loads of traffic without compromising on performance.
4. The framework is based on the Reactive Streams specification and provides support for both client-side and server-side development.
5. With the help of Spring Web Flux, developers can create applications that are more efficient and scalable than traditional ones.

**Understanding Backpressure in Reactive Programming**

In Reactive programming, backpressure is a mechanism that allows the consumer of data to control the rate at which the producer produces data. This is important because, in many scenarios, the producer may be generating data faster than the consumer can process it, leading to a buildup of unprocessed data, which can cause memory and performance issues.

Backpressure enables the consumer to signal the producer to slow down or stop producing data until the consumer is ready to accept more. This can be implemented using various techniques such as buffering, dropping data, or requesting the producer to slow down.

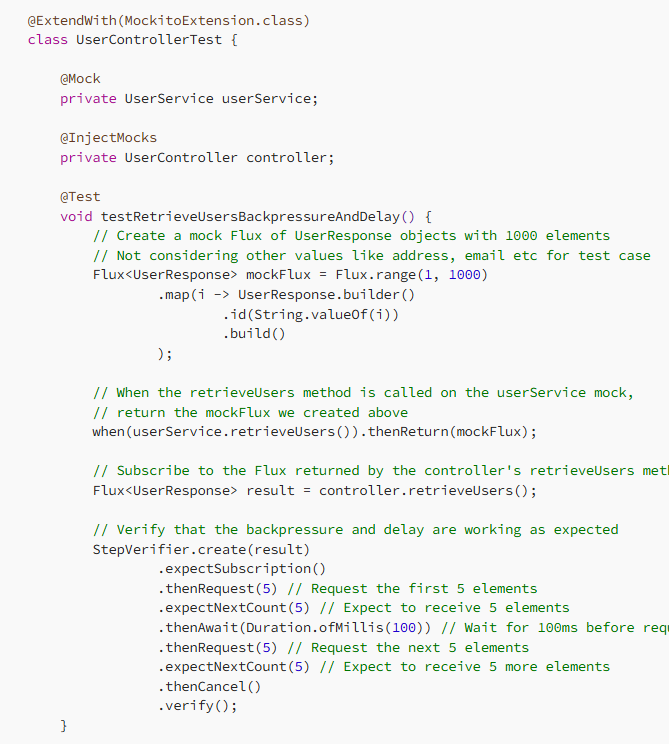
Texto

Descripción generada automáticamente

The **onBackpressureBuffer(10, BufferOverflowStrategy.DROP\_OLDEST)** operator is then applied to the Flux, which limits the buffer size to 10 elements and uses a buffer overflow strategy of dropping the oldest elements when the buffer becomes full. This means that if the downstream subscriber cannot keep up with the rate of emissions, the buffer will store up to 10 elements before it starts dropping older elements to make room for newer ones.

Finally, the **delayElements(Duration.ofMillis(100))** operator is applied to the Flux, which adds a delay of 100 milliseconds before emitting each element. This means that the UserResponse objects emitted by the Flux will be spaced out by at least 100 milliseconds, which can help control the rate of emissions and prevent backpressure issues.

To mock this situation and test the backpressure and delay of elements, you can create a test and use a mocking framework like Mockito to create a mock implementation of ***userService.retrieveUsers()***. You can then use the ***StepVerifier*** class from Reactor to subscribe to the ***Flux*** returned by ***retrieveUsers()*** and verify that the backpressure and delay are working as expected.



**Test case 1: testRetrieveUsersBackpressureAndDelay()**

* Mocks a Flux of UserResponse objects with 1000 elements
* Sets up the UserService mock to return the above-created Flux when **retrieveUsers**() method is called
* Calls the controller’s **retrieveUsers**() method
* Uses ***StepVerifier***to verify that backpressure and delay are working as expected by requesting the first 5 elements waiting for 100ms, then requesting the next 5 elements and canceling after that.

**Conclusion:**

Reactive programming is a powerful programming paradigm that can help you build responsive, resilient, and scalable applications. By using Reactive programming with Spring Boot, you can take advantage of the many benefits it offers while still leveraging the power and flexibility of Spring Boot.

Marble diagram (diagrama de mármol):

Imagen que contiene Diagrama

Descripción generada automáticamente

**¿Qué es un diagrama de mármol?**

Un diagrama de mármol es una representación visual de cómo fluyen los datos a través del tiempo en un sistema reactivo. Los círculos verdes representan elementos de datos (en este caso, los valores emitidos por un Mono), y la línea horizontal representa el eje del tiempo. Las flechas indican la dirección del flujo de datos.

**Entendiendo el diagrama en el contexto de Mono.defer()**

**Mono.defer()** es un operador en Reactor que retrasa la creación de un Mono hasta que alguien se suscribe a él. Esto es útil cuando necesitas crear un Mono diferente cada vez que alguien se suscribe, asegurando que siempre se obtenga el último valor o estado.

**Desglosando el diagrama:**

1. **Dos suscriptores:** El diagrama muestra dos suscriptores (los círculos azules en la parte superior), cada uno esperando un valor de un Mono.
2. **Deferimiento:** Cuando el primer suscriptor se suscribe, se invoca a Mono.defer(). Esto crea un nuevo Mono (el círculo verde en el medio) y se suscribe a él.
3. **Emisión de valor:** El Mono creado dentro de defer() emite un valor (el círculo verde en el medio). Este valor es entregado al primer suscriptor.
4. **Segundo suscriptor:** Cuando el segundo suscriptor se suscribe, se crea un nuevo Mono (el círculo verde en la parte inferior) dentro de defer() nuevamente. Este nuevo Mono emite un valor que es entregado al segundo suscriptor.

**¿Por qué usar Mono.defer()?**

* **Valores frescos:** Garantiza que cada suscriptor reciba un valor calculado o obtenido en el momento de la suscripción, evitando compartir estados entre suscriptores.
* **Retraso de cálculos costosos:** Si la creación de un Mono implica cálculos costosos, defer() puede ayudar a posponer esos cálculos hasta que sean realmente necesarios.
* **Manejo de condiciones variables:** Si el valor que se debe emitir depende de condiciones que pueden cambiar con el tiempo, defer() permite recalcular el valor en cada suscripción.

Source: [Deep Dive into Reactive Programming with Spring Boot | by Kushagrasehgal | Simform Engineering | Medium](https://medium.com/simform-engineering/deep-dive-into-reactive-programming-with-spring-boot-d62cae63bb03)