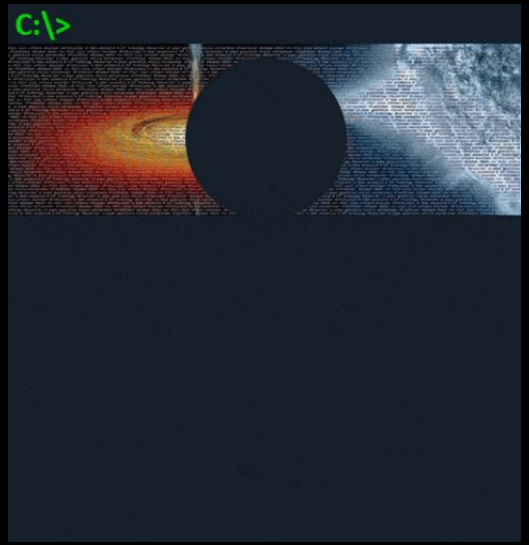
Abstracciones en Clean Architecture

"El error es tu mejor maestro, dale un abrazo"

Víctor A. Higuita C. <vickodev@gmail.com> <@vickodev>

¡Hola a tod@s!



¡Curiosidades!🤓

*Reemplazar la energía de combustible fósil que usamos requiere +1000 veces el PIB mundial.
*La contaminación por plásticos es tan alarmante que cada semana ingerimos el equivalente a una TC.
*El recalentamiento global es un problema de retroalimentación positiva.

Víctor A. Higuita C. <@vickodev>

¿Qué vamos a explorar?

- ¿Qué es la arquitectura de software?
- ¿Qué son los patrones?
- ¿Qué es abstracción? 🤯
- ¿Qué es inversión de control IoC?
- ¿Qué es inversión de dependencias DIv?
- ¿Qué es inyección de dependencias DIj?
- ¿Qué es Clean Architecture (CA)? 🚧
- La fórmula mágica para hacer **CA** y no fallar en el intento 🤔
- Revisemos el diagrama de **Application** de nuestra solución
- Revisemos diagramas de **Infrastructure** y **Adapters** layer 🤓
- Los retos para nuestra solución 🤖

¿Cuál es la dinámica?

Definición corta, ¡se me estalló la cabeza! 🤯

En el flujo de la presentación, cada diapositiva principal tendrá una **imagen alusiva al tema o concepto**, contigua a ella la definición corta como una adaptación del concepto, y aquí en esta área estará la interlocución (Explicación). (💩)

Haciendo mención al emoji una reflexión, ¿Martillo 🔨 o Alicate ✂️?

¿Qué es la arquitectura de software?

Podemos definirlo como las **decisiones de diseño** (estructura) que se toman para cumplir los **atributos de calidad** de un producto de software.

Al principio todo era oscuridad (terminales si acaso) 👀, y en la década de los 60 se empezaron a tocar conceptos como **modularidad**, **herencia**, **encapsulamiento**, pero fue en la década de los 90s donde el término **arquitectura** se usó en contraste con **diseño**, evocando nociones de **abstracción** y **estándares**.

¿Qué son los patrones?

Patrones de arquitectura

Son **arquetipos** (estructuras) que nos indican cómo se **relacionan/interactúan** a nivel macro las partes de un sistema basado en software.

Patrones de diseño

Son **plantillas** base probadas para solucionar problemas comunes. Se clasifican en tres grupos (**creacionales**, **estructurales**, de **comportamiento**). Originalmente se publicaron 23 pero ahora existen muchos más.

¿Qué es Abstracción?

Se puede definir como la separación **del qué** y **del cómo**, es decir, separar lo **que** algo hace del **cómo** lo hace.

El qué es **la acción** (Interface, Abstract class), ésta acción recibe o no parámetros de entrada y nos entrega un resultado. **El cómo** es **lo que hace** (comportamiento) con sus entradas y sus objetos internos para devolver el resultado, es decir **la Clase concreta o implementación**.

En código eso se ve más o menos así

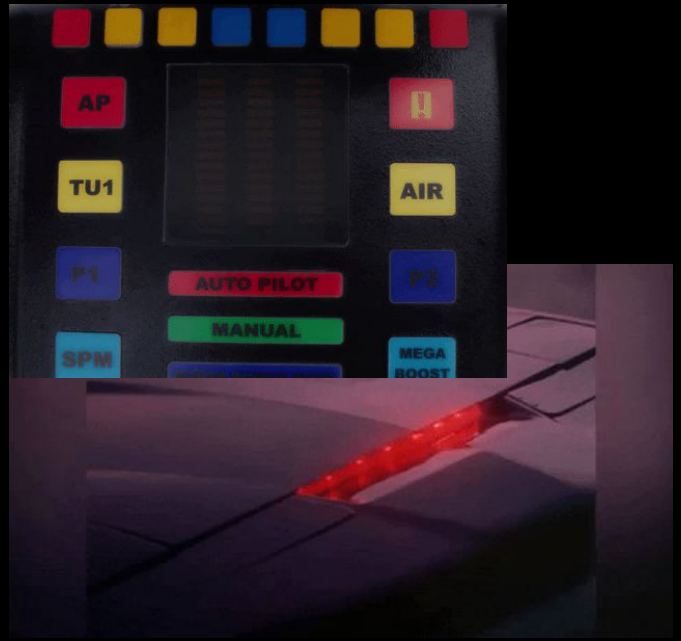
El qué hace

```
// Interface way
interface IChatProvider {
  send(recipient: string, text: string): Promise<void>;
  read(): Promise<Message[]>;
}

// Abstract class way
abstract class ChatProviderContract {
  abstract send(recipient: string, text: string): Promise<void>;
  abstract read(): Promise<Message[]>;
}
```

El cómo lo hace

```
class ChatProvider implements IChatProvider {
// class ChatProvider extends ChatProviderContract
  #address: string;
  #messages: Message[] = [];

  constructor(address: string) {
    this.#address = address;
  }

  async send(recipient: string, text: string): Promise<void> {
    // implement the how it does it
  }
  async read(): Promise<Message[]> {
    // implement the how it does it
  }
}
```

¿Qué es Inversión de Control (IoC)?



Es **delegar el control** de ciertos aspectos de una aplicación a un tercero.

Este tercero normalmente es un **framework** o **servicio**, y puede controlar desde el ciclo de vida de los objetos (DIj) hasta el de la misma aplicación, también puede controlar los **eventos** e invocar acciones de la aplicación a esos eventos (UI), también puede controlar persistencia de datos y proveedores de diferentes tipos de servicios.

¿Qué es Inversión de Dependencias (DIv)?

Básicamente es **Abstracción,** y su principal función es **desacoplar**.

Este es uno de los principios SOLI***D*** (DIvP) y aterrizando sus compendios a un lenguaje sin tecnicismos podemos decir que:

- Las **clases importantes** no deben depender de **clases menos importantes**, **ambas** deben depender de **abstracciones**.
- **El qué** no debe depender **del cómo**, **el cómo** debe depender **del qué**.

¿Qué es Inyección de Dependencias (DIj)?

Es separar la **Construcción/Creación** (instancias) del contexto de ejecución donde dicho objeto interactúa.

Es **delegar la creación** de los objetos a un tercero, en este caso a un contenedor de inyección de dependencias (DIjC) con la intención de desacoplar el núcleo de nuestra aplicación del mundo exterior, ese mundo de las cosas “triviales”.

Ahora el dilema🤔

¿Se puede usar Inyección de Dependencias
sin usar Inversión de Dependencias?

¿Qué es Clean Architecture?

Se puede definir como un marco de trabajo o conjunto de buenas prácticas cuyo objetivo es facilitar la **construcción**, **testeabilidad** y el **mantenimiento** de los productos de software.

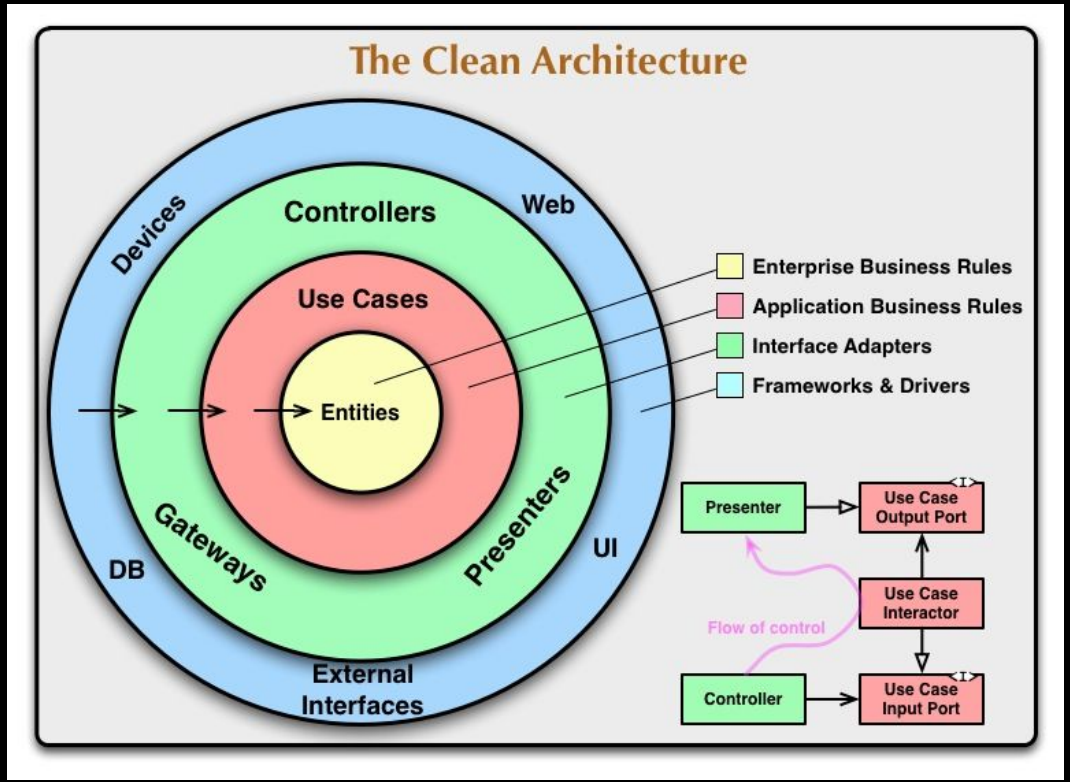
El término fue acuñado por **Robert Martin** y en este compendio tomó parte de otros conceptos arquitectónicos (**Tiers and Layers, Onion, Hexagonal**) que tienen como objetivo la **abstracción**, estructurando el código a través de capas y aislando el mundo interior (lógica de aplicación y de negocio) de las cosas "triviales" como el mundo exterior.

Revisemos las características de CA

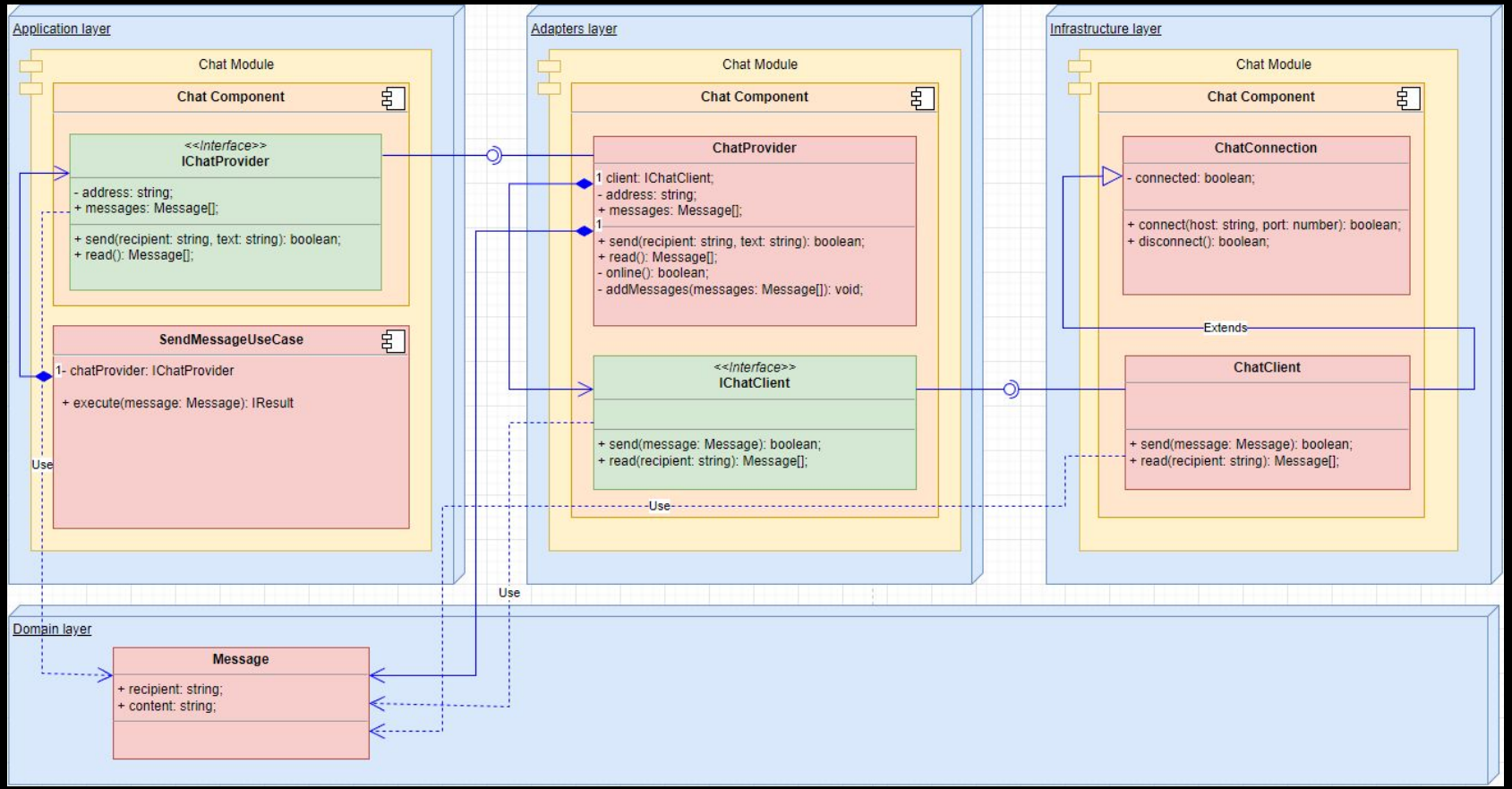
- El acoplamiento sólo es permitido entre una capa y su capa vecina interna. *
- Una capa interna no debe depender (estar acoplada) de una capa externa.

Entities = ***Dominio*** de negocio
Use Cases = Dominio de ***Aplicación*** y orquesta a el dominio de negocio
Layer green = capa de ***Adaptadores***
Layer blue = capa de ***Infraestructura***

* Parte de esto se negocia con el equipo técnico



Fórmula mágica para hacer CA y no fallar 🤓



Víctor A. Higuita C. <@vickodev>

En código eso se ve más o menos así

```ts
// Application layer
interface IChatProvider {
  send(recipient: string, text: string): Promise<void>;
  read(): Promise<Message[]>;
}
```

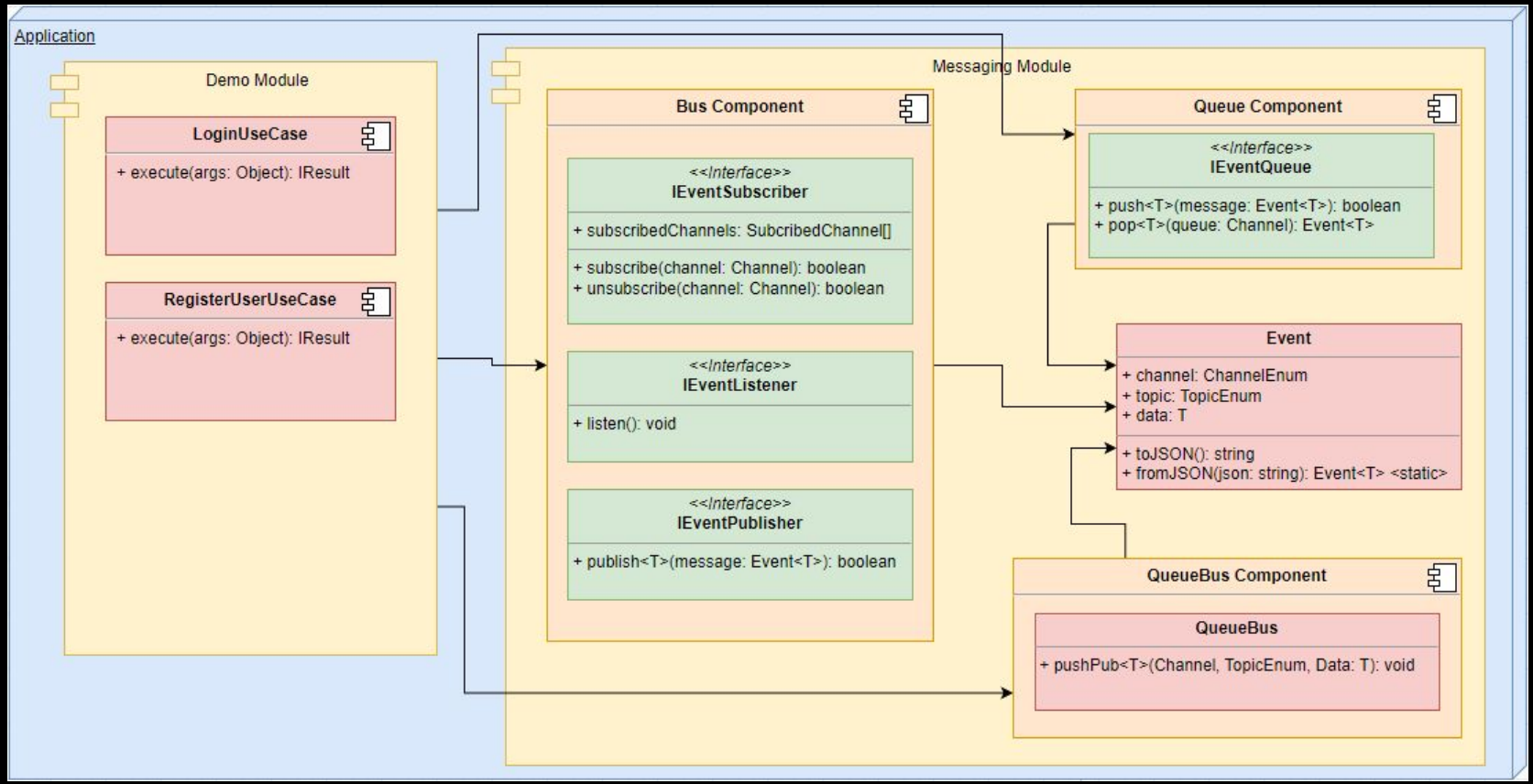
```ts
// Adapter layer
interface IChatClient {
  send(message: Message): Promise<boolean>;
  read(recipient: string): Promise<Message[]>;
}

class ChatProvider implements IChatProvider {
  #address: string;
  #chatClient: IChatClient;
  #messages: Message[] = [];

  async send(recipient: string, text: string): Promise<boolean>
{
    // implementation
  }
  async read(): Promise<Message[]> {
    // implementation
  }
}
```

```ts
// Infrastructure layer
class ChatConnection {
  #connected: boolean = false;
  #chatIO: ChatIO;

  connect(host: string, port: number): boolean {
    // implementation
  }
  disconnect(): boolean {
    // implementation
  }
}

class ChatClient implements IChatClient {
  #connection: ChatConnection;

  async send(message: Message): Promise<boolean> {
    // implementation
  }
  async read(recipient: string): Promise<Message[]> {
    // implementation
  }
}
```

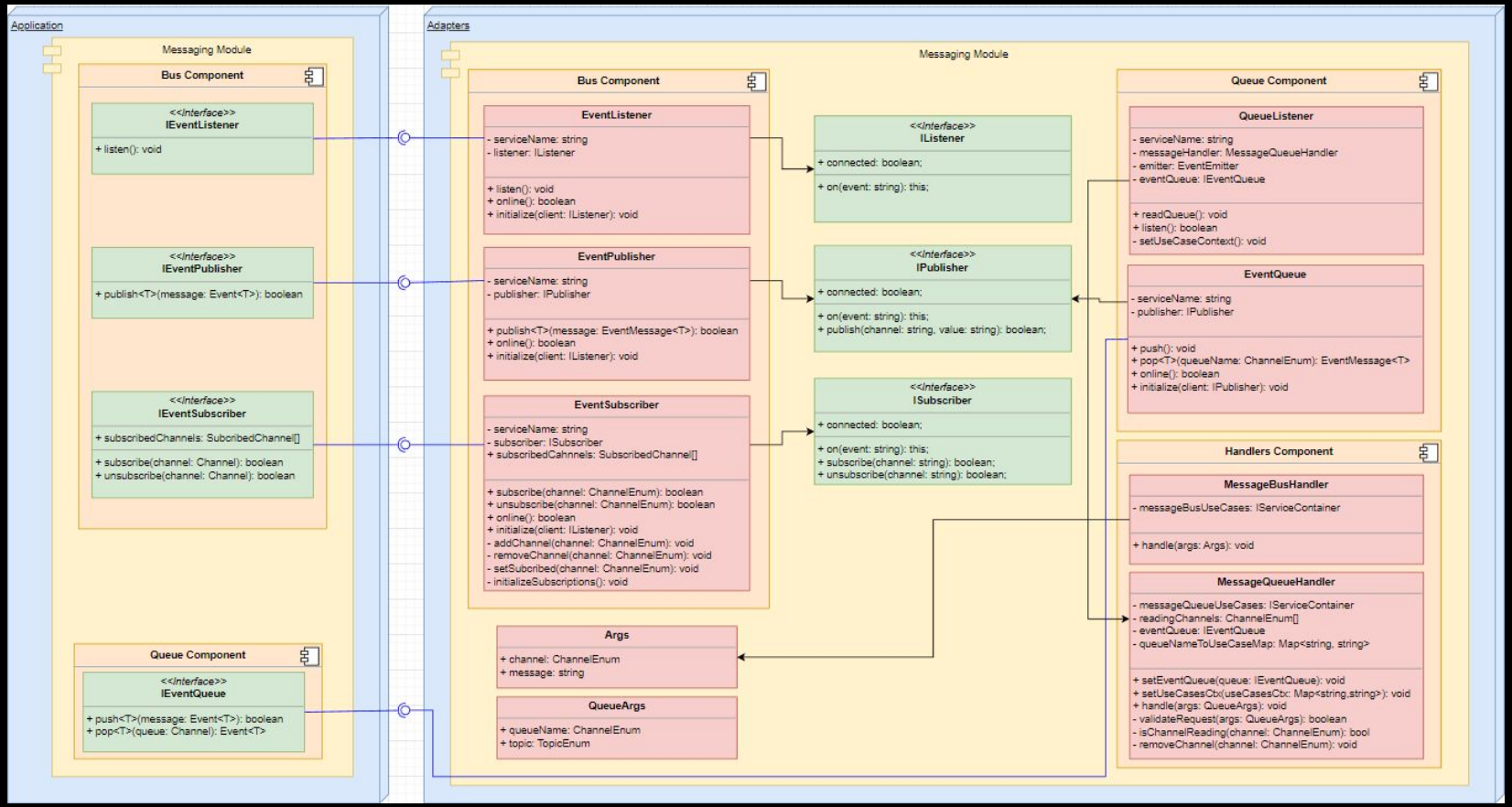
Ahora sí, vamos a ver una implementación 🤓

Nos enfocaremos en las capas de **Application**, **Infrastructure** y **Adapters** de una solución basada en Clean Architecture 🤩

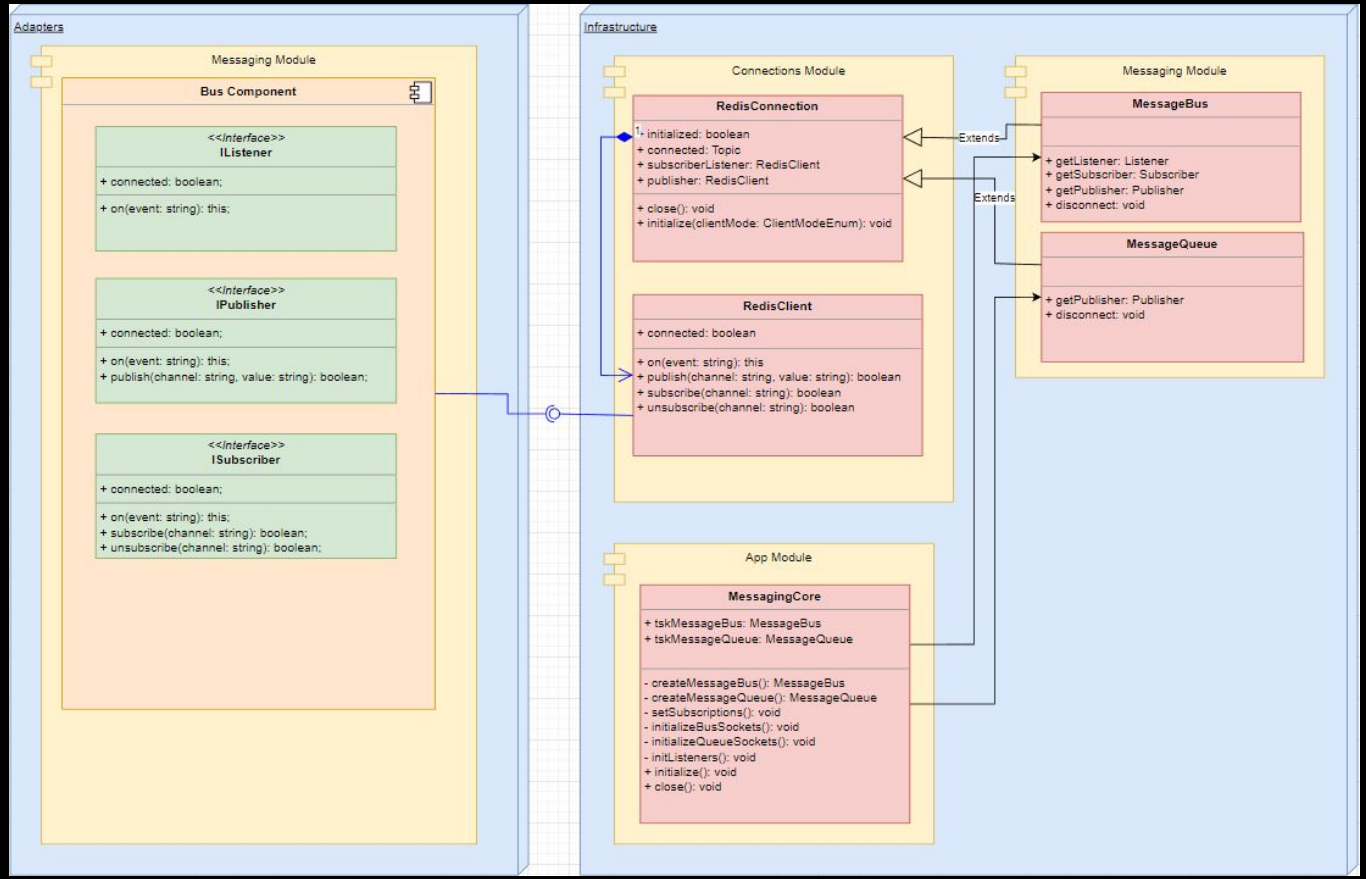
Revisemos nuestra capa de Application



Revisemos nuestra capa de Adapters



Revisemos nuestra capa de Infrastructure



Preparación para los Retos

Repository

¡Aprendemos haciendo, entonces manos a la obra!

Clonar el repositorio y seguir las instrucciones del archivo README para ponerlo a correr.
Tener en cuenta que la **branch** que tiene la implementación de **redis** es feature/queue-bus-events

Una vez el repositorio esté corriendo, prueba los request entregados en los ejemplos del README, tanto el de registrar usuario como el de autenticación, prueba con el usuario por defecto y los nuevos que desees crear (**Ojo, base de datos en memoria***).

Los Retos

Juega con el template, explora sus componentes y trata de entender cómo funcionan, luego desbarata eso, cambia y mueve cosas, y analiza con detenimiento los efectos que te vas encontrando. Level1

Crear un único Handler de messaging Messaging.handler.ts en un directorio messaging en el path 'src\adapters\messaging\handlers' y remover los demás directorios de handlers (directorios bus y queue) y manejar los casos de uso desde ese único handler SIN USAR el EventEmitter de la Queue, el cual debes eliminar de la solución. Level2

De forma abstracta montar un proveedor de caché usando la infraestructura de Redis actual que conviva con el Message Bus y el Message Queue, y esa caché usarla en el caso de uso de autenticación para cachear la sesión. Revisa a través del terminal que la sesión quede en la caché. Puede usar el **maskedUserId** como key.
Los contratos deben quedar en el path 'src\application\shared\cache' y todo debe seguir funcionando tal cual, además de la nueva funcionalidad del proveedor de caché. Level3

SCAN ME

A los que logren los retos **Level2** o **Level3** se les dará ***GitHub Swag,*** y abriremos una discusión para dudas o inquietudes de los retos.

Desacoplar la capa de **Adapters** de la de **Infrastructure** en cuanto a las dependencias acopladas desde **Redis**. Level3+

¡Éxitos y que aprendas mucho!

Víctor A. Higuita C. <@vickodev>

Agradecimientos, recursos y créditos

Article

Architecture

Profile SDP

IoC

Clean Architecture

Animaciones personalizadas https://ezgif.com y GIFs de https://giphy.com/

¡Eso es todo, muchas gracias!

@vickodev