Arquitectura Moderna de Servicios en Java

Círculo Siete Capacitación

Clase 8 de 30
17 Octubre 2024

Estilos arquitecturales

- Arquitectura en Capas (Layered Architecture) - Década de 1970
- Arquitectura de Pizarra (Blackboard Architecture) - Década de 1980
- Arquitectura Monolítica - Década de 1980 - 1990
- Arquitectura Basada en Servicios (SOA) - Finales de 1990 y principios de 2000
- Arquitectura Microkernel - Principios de 2000
- Arquitectura Orientada a Eventos (Event-Driven Architecture) - 2000 en adelante
- Arquitectura Hexagonal (Ports and Adapters) - Años 2005-2010
- Arquitectura de Cebolla (Onion Architecture) por Jeffrey Palermo en 2008,
- Arquitectura de Microservicios - Década de 2010
- Arquitectura Limpia (Clean Architecture) - 2012 en adelante

Estilos arquitecturales

- Arquitectura Monolítica
- Arquitectura Orientada a Eventos (Event-Driven Architecture)
- Arquitectura Hexagonal (Ports and Adapters)
- Arquitectura de Cebolla (Onion Architecture)
- Arquitectura de Microservicios
- Arquitectura Limpia (Clean Architecture)

Arquitectura de Cebolla

- La **Onion Architecture** (Arquitectura de Cebolla) fue introducida por Jeffrey Palermo en 2008.
- Está estrechamente relacionada con la Arquitectura Hexagonal y la Arquitectura Limpia.
- Su propósito es estructurar aplicaciones de forma que la lógica de negocio esté protegida de los detalles de infraestructura y de implementación, creando una separación clara entre capas.

Comparación con otros estilos

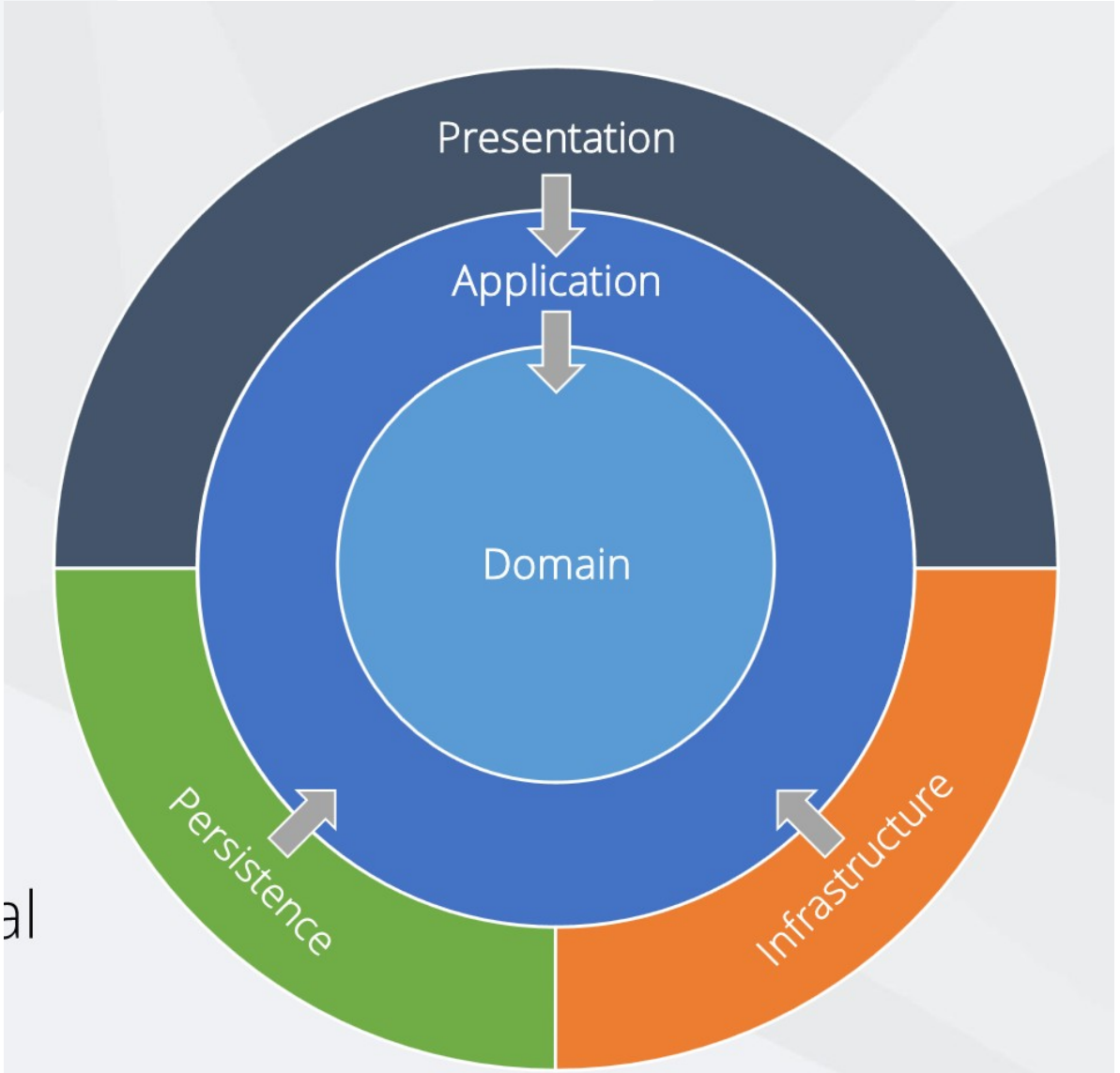
- Onion Architecture, Hexagonal Architecture y Clean Architecture comparten principios fundamentales:
  - La lógica de negocio (Domain) está en el centro.
  - Las dependencias siempre fluyen hacia el núcleo de la aplicación.
  - Los detalles de infraestructura, como bases de datos, interfaces de usuario o frameworks, están en capas externas.

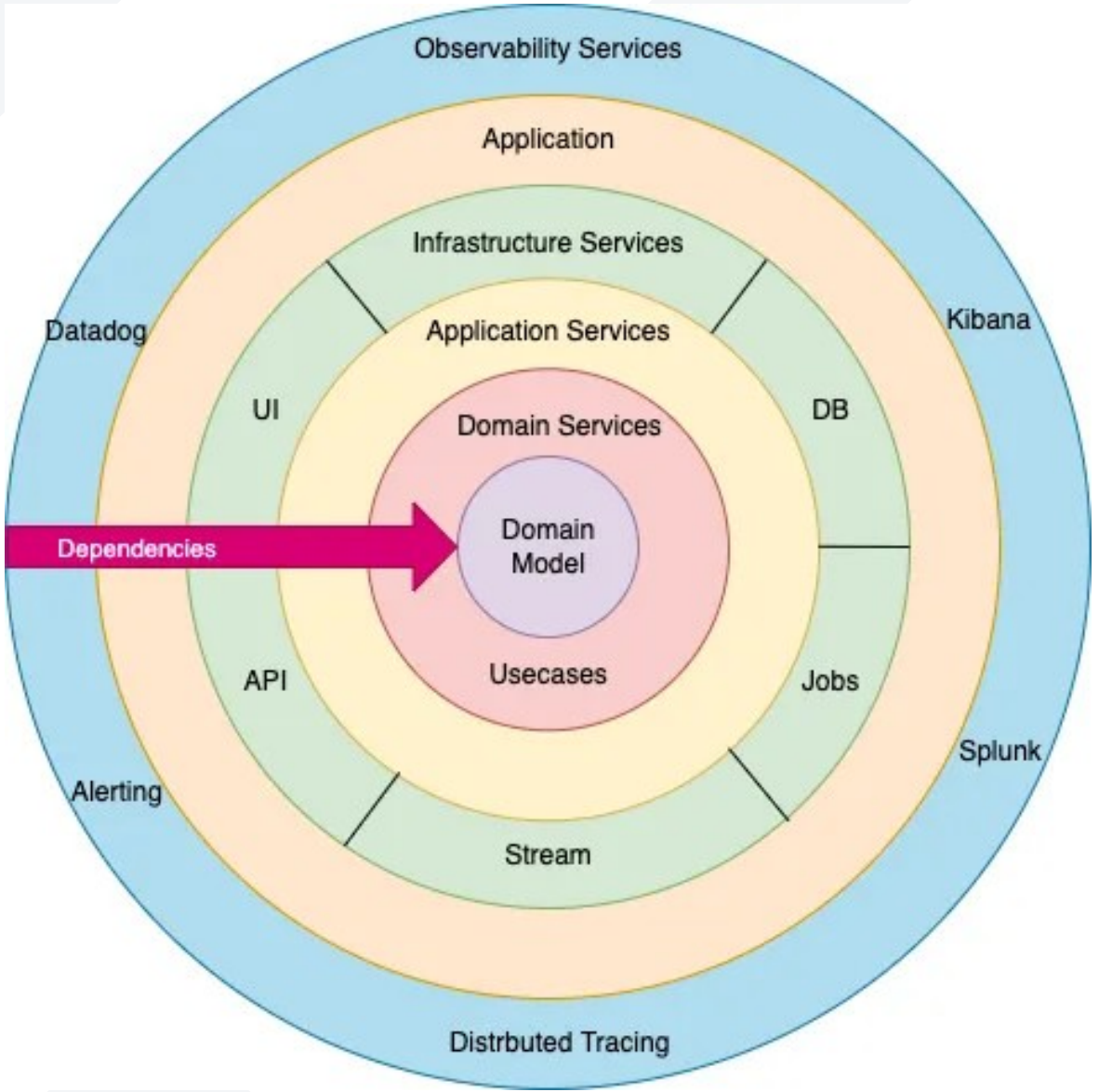
Onion <> Hexagonal

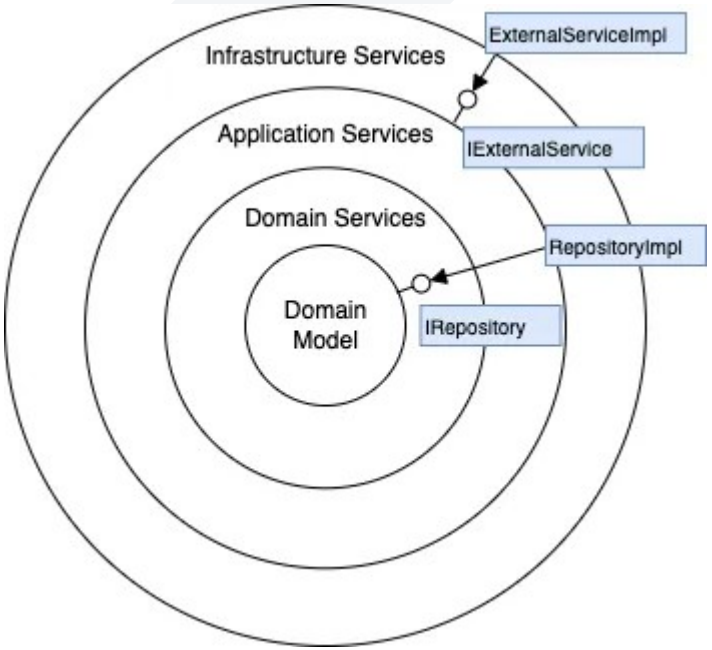
- Onion Architecture es muy similar en concepto a la Arquitectura Hexagonal.
- Aunque su enfoque visual y algunas particularidades en cuanto a términos y capas pueden diferir.
- Mientras que la Hexagonal habla de "puertos y adaptadores", la Onion simplemente organiza las capas en círculos concéntricos, con el dominio en el centro.

Características de Arquitectura de Cebolla

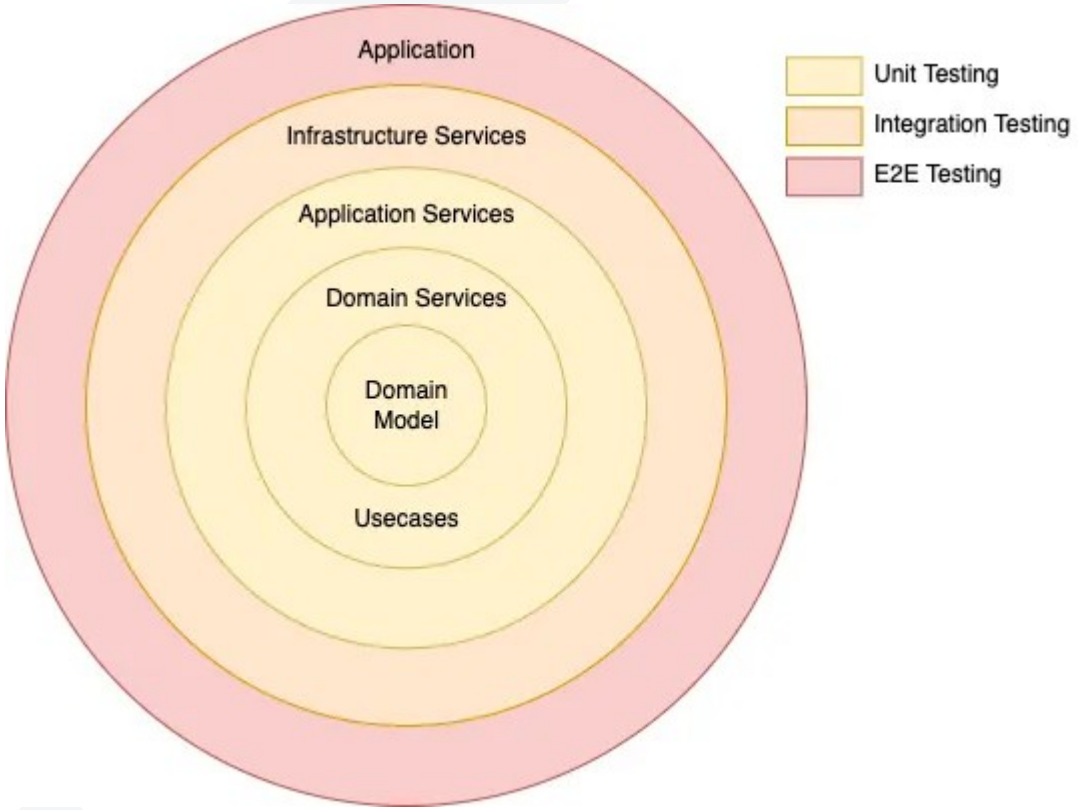
- Se basa en capas concéntricas, con el dominio (entidades y lógica de negocio) en el centro.
- El flujo de dependencias es hacia adentro, es decir, las capas externas dependen de las internas, pero no al revés.
- Las interfaces y los casos de uso se colocan en capas intermedias, mientras que las capas externas manejan los detalles de la infraestructura como bases de datos, frameworks, y APIs externas.


Presentation
Application
Domain
Persistence
Infrastructure


Observability Services
Application
Infrastructure Services
Application Services
Domain Services
Domain Model
Usecases
Datadog
Kibana
UI
DB
Dependencies
API
Jobs
Splunk
Alerting
Stream
Distrbuted Tracing


Infrastructure Services
Application Services
Domain Services
Domain
Model
ExternalServiceImpl
IExternalService
RepositoryImpl
IRepository

Estrategias de pruebas



Enfoque visual

- Las capas están organizadas como anillos concéntricos alrededor de la lógica central.
- La analogía con una cebolla ayuda a visualizar cómo la lógica de negocio está “protegida” por las capas externas.

Capa Dominio Central

- En el centro está el dominio (el "core" del sistema)
- Que incluye las entidades y reglas de negocio.
- Es la parte más importante del sistema y no debe depender de nada externo, como bases de datos o interfaces de usuario.

Capa Dominio Central

- **Entidades**: Representan los objetos fundamentales con identidad y comportamiento.
- **Agregados**: Agrupaciones de entidades con reglas de consistencia.
- **Servicios de dominio**: Lógica de negocio que no encaja dentro de una entidad o un agregado.

Capa de aplicación

- Rodeando al dominio, esta capa contiene la lógica de la aplicación (casos de uso).
- Se encarga de coordinar las interacciones entre el dominio y las capas externas, como la interfaz de usuario o el acceso a datos.
- No se implementa lógica de negocio, sino lógica para orquestar los flujos de trabajo.
  - **Servicios de aplicación**: Implementan casos de uso específicos y coordinan el flujo entre el dominio y las infraestructuras externas.

Capa de infraestructura

- Esta capa contiene detalles técnicos como la persistencia de datos (bases de datos), integración con servicios externos (APIs, microservicios), envío de correos electrónicos, etc.
- La infraestructura debe ser fácilmente reemplazable sin afectar el núcleo del dominio ni la lógica de la aplicación.
  - **Adaptadores**: Pueden ser entradas (que inyectan información al sistema) o salidas (como el acceso a bases de datos o mensajería).

Interfaz de usuario

- Es la capa más externa y contiene la lógica relacionada con la interacción del usuario, como controladores HTTP, vistas y manejo de la entrada/salida.
  - WebAPIs
  - gRPC
  - CLI (Command Line Interface)

Ventajas

- Independencia del dominio: La arquitectura de cebolla promueve que el dominio no dependa de detalles externos (infraestructura o interfaces de usuario), lo que permite una evolución más simple de la lógica del negocio sin impacto en otras áreas.
- Testabilidad: Al desacoplar el dominio de otros componentes, es más fácil realizar pruebas unitarias, ya que el dominio puede ser testeado de manera independiente.
- Mantenibilidad: La separación de responsabilidades y el bajo acoplamiento facilitan el mantenimiento y la evolución del sistema.

Comparación con otros patrones

- **Arquitectura limpia**: La arquitectura de cebolla y la arquitectura limpia son muy similares. Ambas se centran en mantener el dominio central independiente de los detalles externos, aunque la arquitectura limpia a veces incluye una capa adicional de "entidades" puras, fuera del dominio de la aplicación.
- **Arquitectura hexagonal**: También conocida como arquitectura de puertos y adaptadores, es un concepto similar en cuanto al desacoplamiento de dependencias externas. La diferencia principal es que la arquitectura hexagonal se enfoca más explícitamente en la comunicación a través de puertos (interfaces) con adaptadores para conectarse a las tecnologías externas.

Revisión código

- onion-app.zip

Arquitectura Limpia

- Estilo propuesto por Robert C. Martin ("Uncle Bob")
- Tiene como objetivo crear sistemas que sean independientes de frameworks, fáciles de probar, independientes de la UI y altamente mantenibles.
- La idea principal es separar las preocupaciones del sistema en capas de manera que los detalles externos (como frameworks, bases de datos, UI) estén desacoplados del núcleo de la aplicación, lo que permite que el código sea más modular, flexible y resistente al cambio.

Principios clave de Clean Architecture

- **Independencia de frameworks**: El sistema no debe depender de frameworks. Estos son herramientas, no la base de la arquitectura.
- **Testabilidad**: Al separar las responsabilidades del sistema, cada parte del código debe ser fácilmente testeable de manera independiente.
- **Independencia de la UI**: Los detalles de la interfaz de usuario deben estar desacoplados de las reglas de negocio, lo que permite cambiar la interfaz sin afectar la lógica de la aplicación.
- **Independencia de la base de datos**: El sistema no debe depender de ningún tipo específico de base de datos o de acceso a los datos.
- **Independencia de agentes externos**: Cualquier sistema externo, como APIs o bases de datos, debe estar en la capa más externa del sistema.

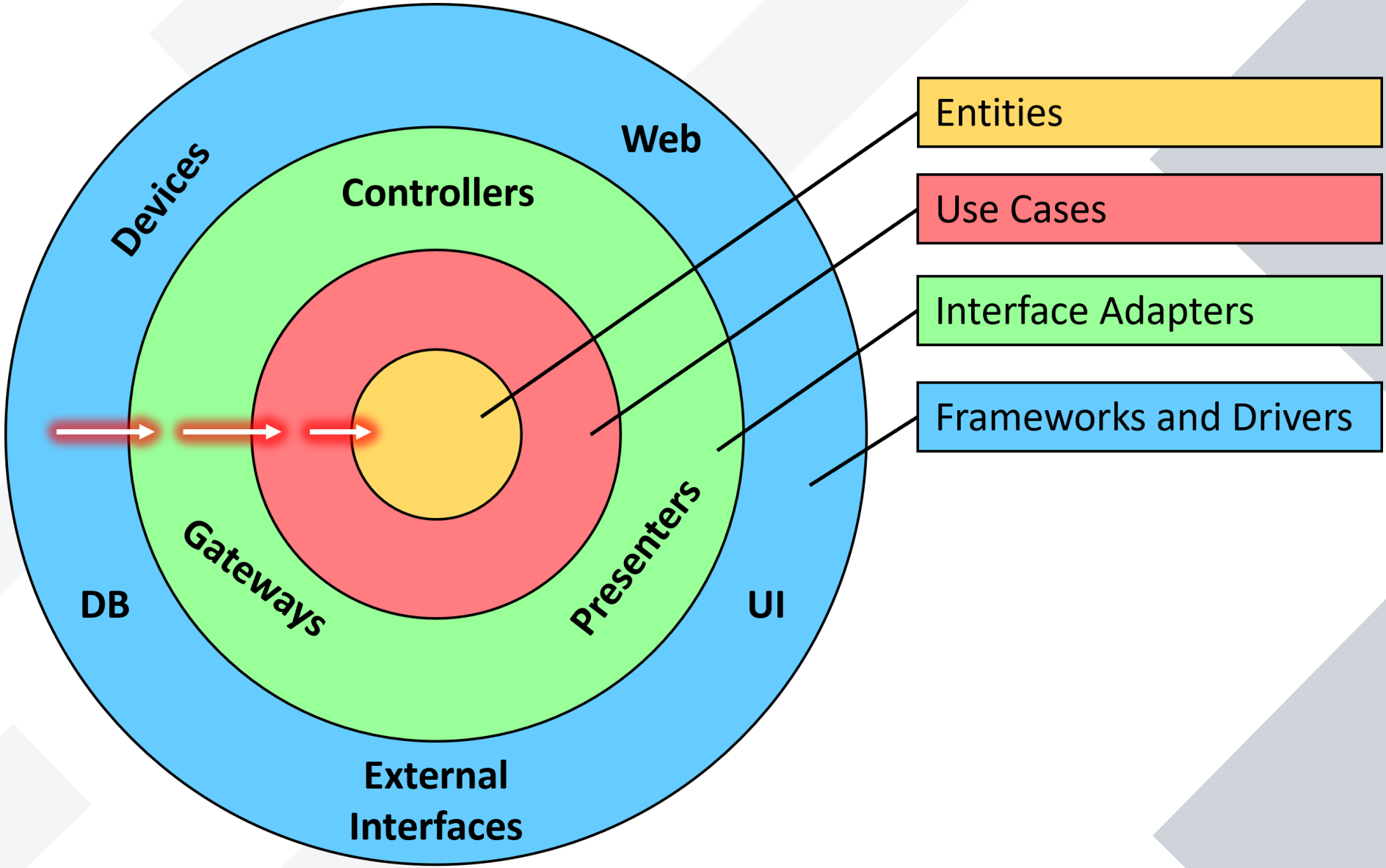
Componentes principales de la Clean Architecture

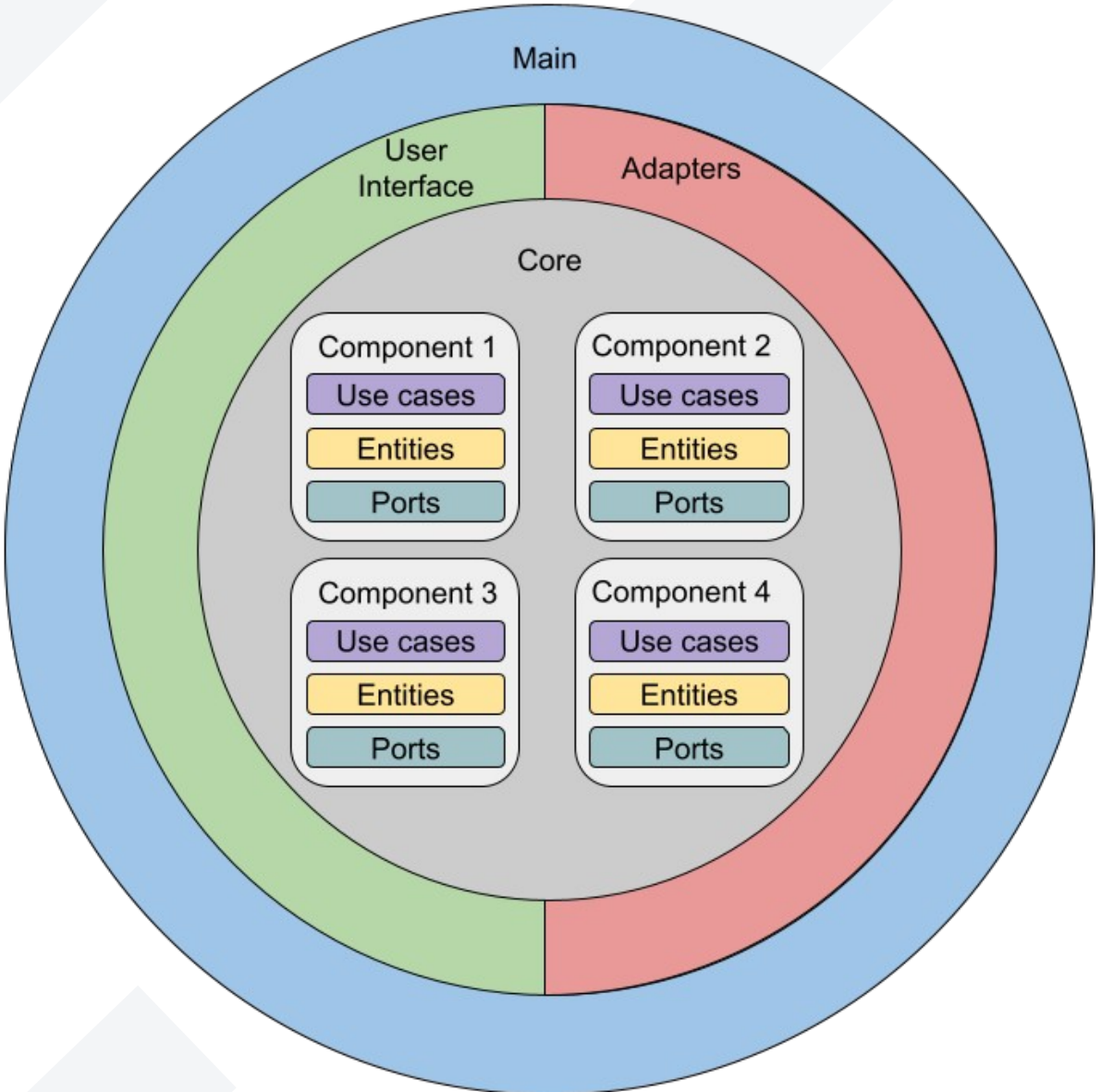
- **Entidades**: Representan las reglas de negocio de más alto nivel. Son las más estables y no dependen de nada más.
- **Casos de uso (Use Cases)**: Contienen la lógica de la aplicación. Describen las interacciones específicas entre las entidades y definen cómo se deben ejecutar las reglas de negocio en función de los requisitos.
- **Interfaces de Entrada/Salida**: Proveen la conexión entre las capas internas y externas. En Clean Architecture, a menudo se utilizan adaptadores y puertos para que las capas externas interactúen con las capas internas sin acoplamiento directo.
- **Capa externa (Frameworks y Drivers)**: Aquí es donde se encuentran los detalles técnicos, como los controladores web, las bases de datos, y cualquier otro servicio externo. Estas partes deben ser lo más fácilmente reemplazables posible.

Organización en capas

("Círculo de la Clean Architecture")

- **Capa de Entidades (Core Entities)**: Las reglas de negocio fundamentales.
- **Capa de Casos de Uso (Use Cases)**: Define el comportamiento específico de la aplicación.
- **Capa de Interfaces**: Provee adaptadores para las capas externas (controladores, servicios externos).
- **Capa de Infraestructura**: Contiene los detalles de implementación, como frameworks y bases de datos.


Devices
Web
Controllers
Gateways
Presenters
DB
UI
External
Interfaces
Entities
Use Cases
Interface Adapters
Frameworks and Drivers


Main
User Interface
Adapters
Core
Component 1
Use cases
Entities
Ports
Component 2
Use cases
Entities
Ports
Component 3
Use cases
Entities
Ports
Component 4
Use cases
Entities
Ports

Principios SOLID y Clean Architecture

- Clean Architecture sigue de cerca los principios SOLID, promoviendo una arquitectura que es altamente mantenible y flexible.
- Clean Architecture busca crear un sistema que sea modular y fácil de mantener, donde los detalles técnicos puedan ser reemplazados o cambiados sin impactar en el núcleo del sistema.
- Es especialmente útil para proyectos a largo plazo, donde se espera que los requisitos cambien con el tiempo.

Independencia de Frameworks

- Este principio sostiene que ningún framework externo debe dictar la estructura del sistema.
- Los frameworks suelen ser herramientas poderosas que proporcionan características como inyección de dependencias, controladores HTTP, ORM (Object-Relational Mapping), etc., pero no deberían ser la base de la arquitectura.
  - **Problema que resuelve**: Si la arquitectura depende de un framework, es difícil cambiarlo en el futuro. Además, el código se vuelve dependiente de las características específicas del framework, lo que lo hace menos flexible.
  - **Aplicación práctica**: Utilizar el framework sólo como una herramienta en la capa más externa del sistema. Esto se logra con el uso de interfaces y adaptadores que aíslen las dependencias del framework de la lógica de negocio. Por ejemplo, en lugar de utilizar directamente un ORM en el código de negocio, se puede usar un repositorio o una abstracción que permita cambiar la implementación de acceso a datos en el futuro sin afectar el resto del sistema.

Testabilidad

- Uno de los objetivos principales de Clean Architecture es que el código sea fácil de probar.
- Esto se logra a través de la separación de responsabilidades y el desacoplamiento de las diferentes capas del sistema.
  - **Problema que resuelve**: Un sistema que depende directamente de tecnologías externas (bases de datos, APIs, frameworks) es difícil de probar. Las pruebas tienden a ser más complejas y lentas, ya que necesitan interactuar con estas dependencias.
  - **Aplicación práctica**: Al desacoplar el código de la lógica de negocio de los detalles técnicos, se pueden hacer pruebas unitarias con facilidad. Por ejemplo, al probar casos de uso o entidades, las dependencias externas pueden ser reemplazadas por objetos simulados (mocks), permitiendo probar la lógica sin necesidad de una base de datos o API real.

Independencia de la Interfaz de Usuario (UI)

- La UI es un detalle técnico que no debe influir en las reglas de negocio o la lógica del sistema. El sistema debe funcionar independientemente de cómo se presente la información al usuario.
  - **Problema que resuelve**: Las interfaces de usuario suelen cambiar más frecuentemente que la lógica de negocio, y si están acopladas, cualquier cambio en la UI puede obligar a modificar la lógica interna del sistema.
  - **Aplicación práctica**: Clean Architecture coloca la UI en una de las capas más externas, de modo que las capas de negocio no dependan de ella. Así, puedes cambiar de un sistema web a una aplicación móvil sin necesidad de modificar el núcleo del sistema. Los controladores de UI simplemente actúan como "orquestadores" que traducen las interacciones del usuario a llamados a casos de uso, sin contener lógica de negocio.

Independencia de la Base de Datos

- El acceso a la base de datos es un detalle de implementación, no debe influir en la lógica central de la aplicación. La base de datos es solo un mecanismo para almacenar información y no debe afectar las reglas de negocio o los casos de uso.
  - **Problema que resuelve**: Si el sistema está fuertemente acoplado a una base de datos específica (por ejemplo, SQL vs NoSQL), puede ser muy difícil o costoso cambiar a otra tecnología de base de datos en el futuro. Además, dificulta la reutilización de la lógica de negocio en otros contextos.
  - **Aplicación práctica**: Los repositorios y adaptadores para acceso a datos deben ser implementaciones externas, desacopladas del núcleo de la aplicación. Se utilizan interfaces para abstraer la forma en que se accede o manipulan los datos, permitiendo cambiar la base de datos sin impactar las reglas de negocio.

Independencia de Agentes Externos

- Cualquier agente externo, como servicios de terceros (APIs, servicios en la nube, etc.), debe estar en una capa externa y no afectar las reglas de negocio ni los casos de uso.
  - **Problema que resuelve**: Si la lógica de negocio depende directamente de un servicio externo, cualquier cambio en el servicio externo puede romper la funcionalidad de la aplicación. Además, es difícil probar la lógica sin conectarse al servicio real.
  - **Aplicación práctica**: Se puede utilizar un patrón como el de "puertos y adaptadores" (también conocido como arquitectura hexagonal) para interactuar con agentes externos. La aplicación debería definir puertos (interfaces) que representen las interacciones necesarias con estos servicios, y los adaptadores implementan estos puertos para interactuar con el servicio real o con una implementación simulada en los tests.

Otras consideraciones clave

1. 1. Reversión de las Dependencias (Dependency Rule)
   - Las dependencias solo pueden ir de capas exteriores a capas interiores. Las capas internas nunca deberían conocer detalles sobre las capas externas. Esto significa que los detalles de implementación, como frameworks, bases de datos o sistemas de entrega, dependen del núcleo de la aplicación, no al revés.
2. 2. Capas Circulares
   - La Clean Architecture se suele visualizar en capas concéntricas, donde cada círculo representa un nivel de abstracción o responsabilidad:
   - Entidades: El círculo más interno, contiene las reglas de negocio generales.
   - Casos de uso: Representa las reglas específicas de la aplicación.
   - Interfaces: Se ocupan de la presentación de datos y la interacción con el mundo externo.
   - Infraestructura: El círculo más externo que contiene frameworks, bases de datos, APIs, etc.

Capas en Clean Architecture

- Cada capa tiene responsabilidades claras y está diseñada para estar lo más desacoplada posible de las otras capas.
- Esto promueve una estructura modular y facilita el mantenimiento y la escalabilidad del software.

Entidades (Entities)

- Las entidades representan el núcleo del dominio de negocio de la aplicación. Son objetos que contienen las reglas más fundamentales del negocio, las cuales son universales y aplican en cualquier contexto. Estas entidades no dependen de nada externo, ni de frameworks, bases de datos, UI, o cualquier detalle de implementación.
- Función:
  - Reglas de negocio de alto nivel: Las entidades encapsulan las invariantes del negocio. Por ejemplo, una entidad "Usuario" en una aplicación de gestión de usuarios puede tener reglas como: "la edad mínima del usuario debe ser 18 años".
  - Independencia: Las entidades deben poder existir sin depender de la infraestructura o la tecnología. Son completamente independientes.

```java
public class User {
    private String name;
    private String email;
    private int age;

    public User(String name, String email, int age) {
        if (age < 18) {
            throw new IllegalArgumentException("Age must be 18 or older");
        }
        this.name = name;
        this.email = email;
        this.age = age;
    }

    // Getters, Setters, reglas de negocio adicionales...
}
```

Casos de Uso (Use Cases)

- Representan la lógica de la aplicación y describen cómo las entidades interactúan entre sí para cumplir con los requisitos de la aplicación. Un caso de uso es una acción que la aplicación puede realizar, como "crear un usuario", "procesar una orden", o "validar un pago".
- Función:
  - Reglas de negocio específicas de la aplicación: Los casos de uso coordinan las interacciones entre las entidades para implementar funcionalidades de la aplicación.
  - Orquestación: Actúan como "orquestadores" de las entidades, coordinando las acciones sin que las entidades se conozcan entre sí. Esto promueve la separación de responsabilidades.
  - Ciclo de vida del caso de uso: Un caso de uso puede involucrar acciones como validar datos, interactuar con un repositorio para guardar o recuperar entidades, y manejar excepciones.

```java
public class CreateUserUseCase {

    private final UserRepository userRepository;
    private final EmailService emailService;

    public CreateUserUseCase(UserRepository userRepository, EmailService emailService) {
        this.userRepository = userRepository;
        this.emailService = emailService;
    }

    public void execute(User user) {
        if (userRepository.existsByEmail(user.getEmail())) {
            throw new IllegalArgumentException("Email already exists");
        }
        userRepository.save(user);
        emailService.sendWelcomeEmail(user.getEmail());
    }
}
```

Interfaces de Entrada/Salida (Controllers, Presenters, Gateways)

- Permiten que las capas internas interactúen con el mundo externo (interfaz de usuario, base de datos, servicios externos, etc.). Las interfaces de entrada/salida se utilizan para desacoplar la lógica de negocio de la infraestructura.
- Función:
  - **Interfaces de Entrada (Controllers)**: Estas son las clases que actúan como controladores de entrada a la aplicación. Los controladores reciben las solicitudes externas (de la UI o de una API REST, por ejemplo) y las traducen a comandos que los casos de uso pueden procesar.
  - **Interfaces de Salida (Gateways)**: Representan los puertos para interactuar con recursos externos como bases de datos, servicios web, etc. Los repositorios son un tipo de interfaz de salida.

Capa Externa (Frameworks y Drivers)

- La capa externa es donde viven todos los detalles técnicos de la aplicación, como los frameworks web, bases de datos, sistemas de mensajería, o servicios externos. Esta capa es lo más desacoplada posible de la lógica de negocio, para que los detalles técnicos puedan ser cambiados sin afectar el núcleo de la aplicación.
- Función:
  - **Implementación de interfaces**: Los adaptadores que implementan las interfaces de los repositorios, servicios de email, controladores, etc., están aquí. Por ejemplo, si usas Spring para controlar el acceso a datos o HTTP, esa implementación estará en esta capa.
  - **Cambio fácil de tecnologías**: Esta capa puede cambiar fácilmente sin afectar el núcleo del sistema. Puedes cambiar de una base de datos SQL a NoSQL, o de un sistema de correo electrónico a otro, sin modificar la lógica de negocio.

Ejemplo General de Flujo

1. Un controlador de API recibe una solicitud HTTP y la convierte en un objeto de dominio.
2. El caso de uso es invocado por el controlador y ejecuta la lógica de negocio, interactuando con entidades y repositorios.
3. El caso de uso llama a un repositorio (un adaptador de salida) para guardar o recuperar datos.
4. El repositorio está implementado en la capa de infraestructura (usando una tecnología como JPA, JDBC, etc.).
5. El caso de uso puede invocar otros adaptadores, como un servicio de correo o un servicio externo.

Revisión Casos de uso

- Abrir archivo casos_de_uso.pdf

Tarea

- Estudiar/repasar los principios SOLID
  - https://bit.ly/48b3NSM
- Intentar hacer funcionar onion-app.zip
  - mvn spring-boot:run