Ejemplo de arquitectura limpia con Java y Spring Boot

Juan manuel lopez Seguir
Abr 6 · 7 min de lectura ★

Crear una aplicación es fácil, simplemente siéntate para codificar y ... *habemus* . Si está atascado, simplemente vaya a Internet, busque y encontrará toneladas de ejemplos, marcos, plantillas que lo ayudarán con su aplicación. Pero crear una aplicación genial, lista para producción es una historia completamente diferente.

> *La idea de Clean Architecture es poner la entrega y la puerta de enlace en los bordes de nuestro diseño.. La lógica empresarial no debe depender de si exponemos una REST o una API GraphQL, y no debe depender de dónde obtenemos datos: una base de datos, una API de microservicio expuesta a través de gRPC o REST, o simplemente un simple archivo CSV.*

El patrón nos permite aislar la lógica central de nuestra aplicación de las preocupaciones externas. Tener nuestra lógica central aislada significa que podemos cambiar fácilmente los detalles de la fuente de datos **sin un impacto significativo o reescrituras de código importantes en la base de código** .

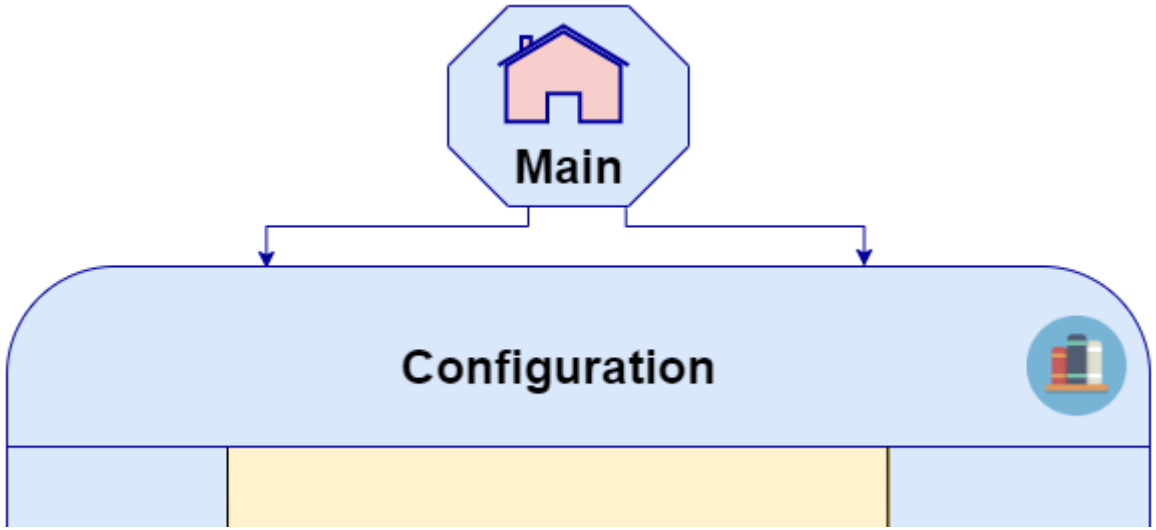
Una de las principales ventajas que también vimos al tener una aplicación con límites claros es nuestra estrategia de prueba: la mayoría de nuestras pruebas pueden verificar nuestra lógica empresarial **sin depender de**

Componentes

Quería dar un enfoque más pragmático / simplista que pueda ayudar en la primera incursión en la arquitectura limpia. Es por eso que omitiré conceptos que pueden parecer inevitables para los puristas de la arquitectura.

Mi único objetivo aquí es que entiendas lo que considero el tema principal (y más complicado) en arquitectura limpia:

Modelo de componentes



Núcleo

El " **núcleo**" se describirá en el componente Entidades modelo y Caso de uso. Aquí está el conocimiento del dominio y la "característica" que queremos implementar.

Entidades modelo

> Las entidades encapsulan las reglas comerciales de toda la empresa.

Una entidad puede ser un objeto con métodos, o puede ser un conjunto de

entidades puedan ser utilizadas por muchas aplicaciones diferentes en la empresa. Representan el dominio desnudo.

Casos de uso

> *que contienen reglas comerciales específicas de la aplicación. Estos serían equivalentes a los* Servicios de aplicaciones *con la advertencia de que cada clase debería centrarse en un* Caso de uso *particular* .

Encapsula e implementa todos los casos de uso del sistema. Estos casos de uso organizan el flujo de datos hacia y desde las entidades, y ordenan a esas entidades que usen sus reglas de negocio para toda la empresa para lograr los objetivos del caso de uso.

Infraestructura

Todo esto es solo porque la programación de la computadora no es perfecta, necesitamos mucho apoyo para permitir que el ***núcleo*** funcione en un entorno real. Estos componentes se conocen como la " ***Infraestructura*** " está compuesta por:

> En la entrega encontramos los adaptadores de interfaz que reciben solicitudes desde el exterior del microservicio y les responde.

Es común implementarlo como un servidor HTTP REST , o consumir mensajes de algún intermediario de mensajes o Job Schudeler

Pasarelas

> En la entrega, encontramos los adaptadores de interfaz que reciben solicitudes desde el exterior del microservicio y entregan (de ahí su nombre) una respuesta a ellos.

Es común implementarlos como Clientes HTTP REST, Productor de Message Broker o cualquier otro Cliente API.

Repositorios

> Estos son adaptadores de interfaz para sistemas destinados a almacenar y recuperar objetos de aplicación (serializados) (generalmente entidad).

El repositorio son las interfaces para obtener entidades, así como para

comunicarse con las fuentes de datos y devuelven una sola entidad o una lista de entidades. (por ejemplo, UserRepository)

Configuración

> La configuración es la parte del sistema que compone los diferentes componentes en un sistema en ejecución.

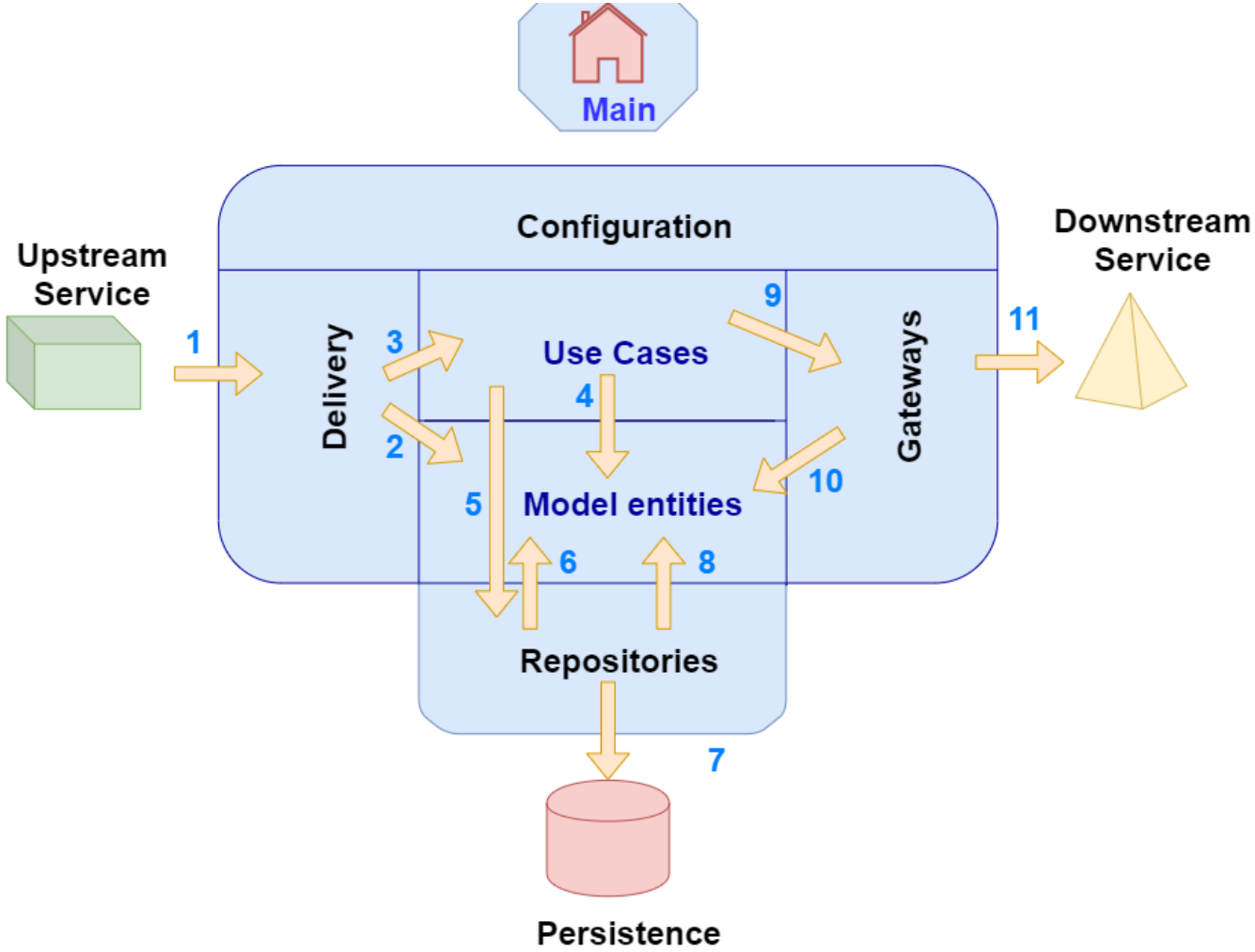
En este módulo creamos la aplicación de acuerdo con las implementaciones que estamos desarrollando y el comportamiento que queremos darle.

Principal

> *El principal es solo el punto de entrada a la aplicación.*

Flujo del proceso

Para proporcionar el comportamiento deseado, estos componentes interactúan entre ellos. El flujo de acciones, generalmente sigue este patrón:



Flujo de proceso de solicitud

1. Un sistema externo realiza una solicitud (una solicitud HTTP, un

2. La entrega crea varias entidades modelo a partir de los datos de la solicitud.
3. La entrega llama a un caso de uso ejecutar
4. El caso de uso opera en entidades modelo
5. El caso de uso hace una solicitud para leer / escribir en el repositorio
6. El repositorio consume alguna entidad modelo de la solicitud de caso de uso
7. El repositorio interactúa con la persistencia externa (DB, caché, etc.)
8. El repositorio crea entidades modelo a partir de los datos persistentes.
9. El caso de uso solicita la colaboración de una puerta de enlace
10. La puerta de enlace consume las entidades modelo proporcionadas por la solicitud de caso de uso
11. La puerta de enlace interactúa con servicios externos (otra API, microservicios, poner mensajes en colas, etc.)

Implementación de Java

Entidades modelo

Los objetos de dominio (p. Ej., Una película o una ubicación de filmación) no tienen conocimiento de dónde están almacenados.

```
@Datos
@AllArgsConstructor
public class Category extiende SelfValidating < Category > implementa Serializable {

        @Min ( 0 )
         Identificación larga privada ;

        @No vacío
         nombre de cadena privada ;

         Booleano privado disponible;

}
```

category.java alojado con ♡ por GitHub ver en bruto

Entidad modelo

Caso de uso

son clases que organizan y realizan acciones de dominio: piense en objetos de servicio u objetos de casos de uso. Implementan reglas comerciales

complejas y lógica de validación específica para una acción de dominio (por ejemplo, incorporación de una producción)

Ejemplo del caso de uso:

```
@AllArgsConstructor
La  clase  pública GetAllCategoriesUseCaseImpl  implementa  GetAllCategoriesUseCase {

    Private  final  CategoryRepositoryService categoryRepositoryService;

    @Anular
     Colección  pública < Categoría > execute () {
        return categoryRepositoryService . getAllCategories ();
    }

}
```

GetAllCategoriesUseCaseImpl.java alojado con ♡ por GitHub ver en bruto

Al tener la lógica de negocio extraída en el caso de uso, no estamos acoplados a una capa particular de transporte o implementación de controlador. El caso de uso puede ser desencadenado no solo por un controlador, sino también por un evento, un trabajo cron o desde la línea de comandos.

La capa de transporte puede activar un interactor para realizar la lógica empresarial. Lo tratamos como una entrada para nuestro sistema. La capa de transporte más común para microservicios es la capa HTTP API y un conjunto de controladores que manejan las solicitudes.

```
@AllArgsConstructor
La clase pública CategoryPersistenceImpl implementa CategoryRepositoryService {

    privado CategoryRepository categoryRepository;
    privado CategoryRepositoryConverter categoryRepositoryConverter;

    @Anular
     Colección pública < Categoría > getAllCategories () {
        return categoryRepository . findAll () . corriente () . map (category - > categoryReposi
                .collect ( Collectors . toList ());
    }

    @Anular
    public void saveCategory ( categoría de categoría ) lanza NetflixException {
        if (doesCategoryNameExists (category . getName ()))
            lanzar una nueva BadRequestException ( ExceptionConstants . BAD_REQUEST_EXISTS_CAT
        categoryRepository . save (categoryRepositoryConverter . mapToTable (category));
    }

    private Boolean doesCategoryNameExists ( String name ) {
        volver ! categoryRepository . findByName (nombre) . esta vacio();
    }
```

Cada controlador maneja una solicitud específica, convierte la carga útil de la solicitud a DTO o entidades modelo, y llama a la ejecución apropiada. Finalmente, convierte la respuesta de ejecución en la carga útil de respuesta y la devuelve a la persona que llama.

Pasarelas

Aquí el Gateway aprovecha un RestClient provisto para conectarse a un servicio externo.

```
1  @FeignClient ( name = " filmClient " , url =  " http: // storage-image / film-info / film / " )
2   interfaz  pública StorageImageRestClient {
3
4      @RequestMapping ( method  =  RequestMethod . GET , value  =  " {id} " , produce  =  " applica
5      StoraImage  findById ( @PathVariable ( " id " ) final  Long  id );
6
7  }
```

FeignClient.java alojado con ♡ por GitHub ver en bruto

El Gateway no tiene lógica de negocios, sino conversiones y mapeos.

Repositorios

Una fuente de datos implementa métodos definidos en el repositorio y almacena la implementación de obtención y envío de datos.

```
@AllArgsConstructor
La clase pública CategoryPersistenceImpl implementa CategoryRepositoryService {

    privado CategoryRepository categoryRepository;
    privado CategoryRepositoryConverter categoryRepositoryConverter;

    @Anular
     Colección pública < Categoría > getAllCategories () {
        return categoryRepository . findAll () . corriente () . map (category - > categoryReposi
                .collect ( Collectors . toList ());
    }

    @Anular
    public void saveCategory ( categoría de categoría ) lanza NetflixException {
        if (doesCategoryNameExists (category . getName ()))
            lanzar una nueva BadRequestException ( ExceptionConstants . BAD_REQUEST_EXISTS_CAT
        categoryRepository . save (categoryRepositoryConverter . mapToTable (category));
    }

    private Boolean doesCategoryNameExists ( String name ) {
        volver ! categoryRepository . findByName (nombre) . esta vacio();
    }

}
```

Configuración

En este ejemplo de código, existe la configuración del comportamiento de todas nuestras capas. Entonces enviamos un bean a todos los beans que dependen de él.

```
@Configuración
 clase  pública CategoryConfiguration {


    @Frijol
    public  CategoryRepositoryConverter  createCategoryRepositoryConverter () {
        return  new  CategoryRepositoryConverter ();
    }

    @Frijol
    public  CategoryRestConverter  createCategoryRestConverter () {
        return  new  CategoryRestConverter ();
    }

    @Frijol
    public  CategoryServiceImpl  createCategoriServiceImpl () {
        return  new  CategoryServiceImpl (categoryRepository, createCategoryRepositoryConverter
    }

    @Frijol
```

```
23      }
24
25      @Frijol
26      public  CreateCategoryUseCaseImpl  createCreateCategoryUseCase () {
27          devolver  nuevo  CreateCategoryUseCaseImpl (createCategoriServiceImpl ());
28      }
29
30  }
```

Principal

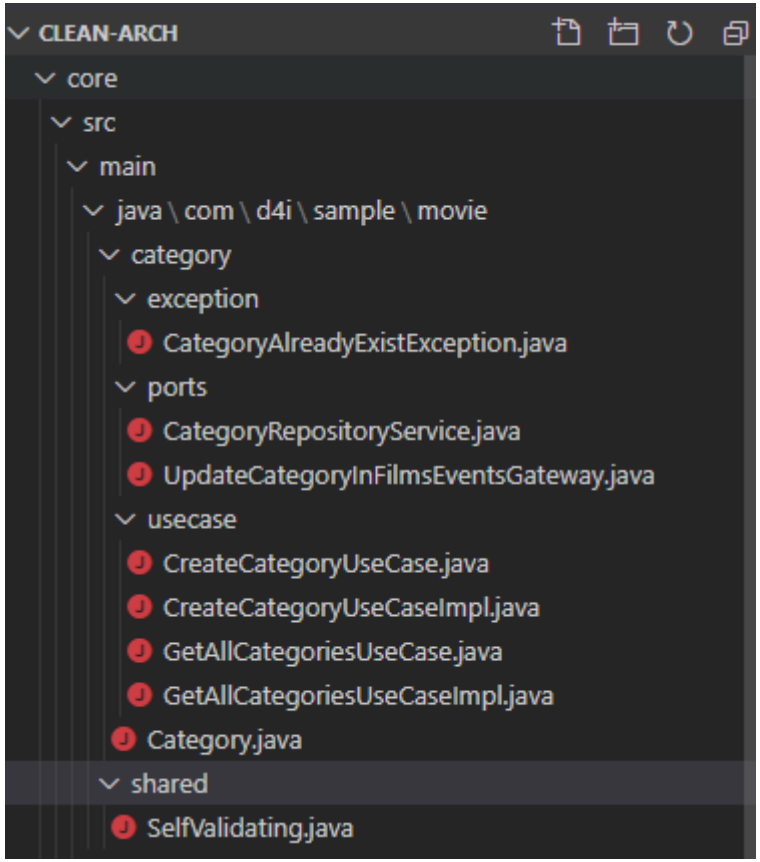
Finalmente, la clase principal termina siendo muy aburrida:

```
1   @SpringBootApplication
2    Solicitud de clase  pública {
3
4       public  static  void  main ( String [] args ) {
5           SpringApplication . ejecutar ( aplicación . clase, args);
6       }
7
8   }
```

Main.java alojado con ♡ por GitHub ver en bruto

Estructura del paquete

Esta arquitectura pragmática está separada en dos subproyectos, núcleo e infraestructura:

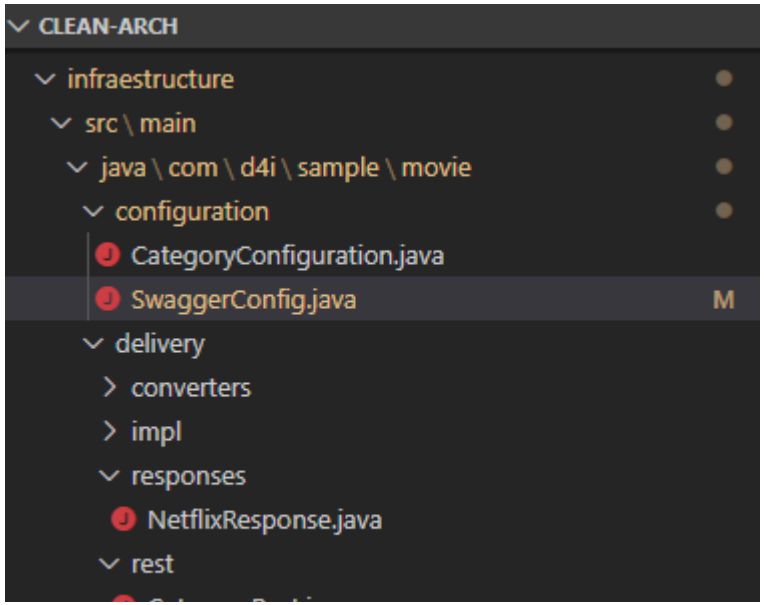


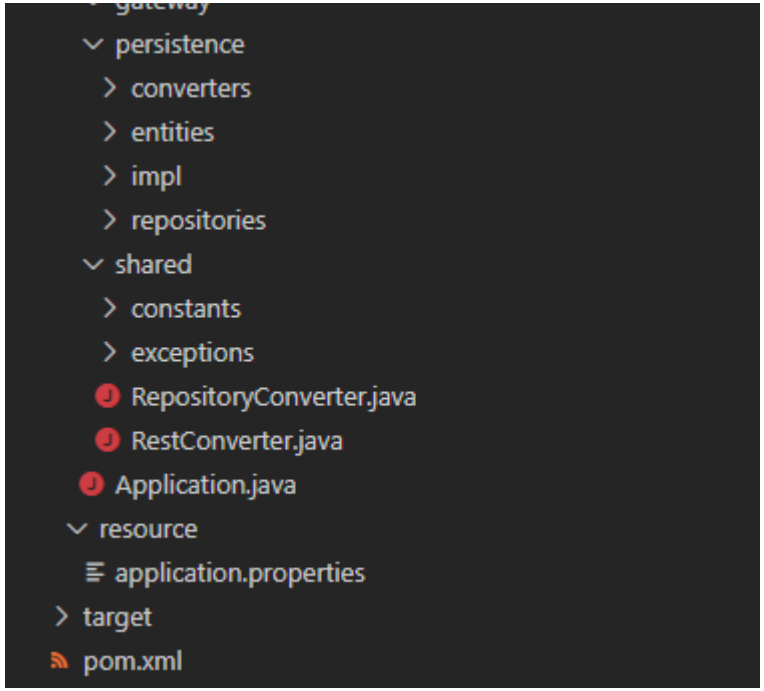
Núcleo

Las capas de dominio y aplicación representan el núcleo, sin embargo, su

Lógica comercial de dominio : aquí encontrará los "modelos" de su aplicación, que pueden ser de diferentes tipos (Raíces agregadas, Entidades, Objetos de valor) y que implementan reglas comerciales para toda la empresa (no solo contienen datos sino también procesos).

Lógica empresarial de aplicaciones : aquí se encuentran los llamados *"casos de uso",* situados en la parte superior de los modelos y los *"puertos"* para la capa de datos (utilizados para la inversión de dependencia, generalmente **interfaces de repositorio**), recuperan y almacenan modelos de dominio utilizando cualquiera de los repositorios u otros casos de uso.





Infraestructura

Es necesario mucho apoyo para permitir que el núcleo funcione en un entorno real. Estos componentes se conocen como la " ***Infraestructura*** ".

La persistencia puede ser un adaptador para una base de datos SQL (una clase de Active Record en Rails o JPA en Java), un adaptador de búsqueda elástica, API REST o incluso un adaptador para algo simple como un archivo

CSV o un Hash. Una fuente de datos implementa métodos definidos en el repositorio y almacena la implementación de obtención y envío de datos.

La entrega puede activar un interactor para realizar la lógica empresarial. Lo tratamos como una entrada para nuestro sistema. La capa de transporte más común para microservicios es el HTTP y un conjunto de controladores que manejan las solicitudes. Al tener la lógica de negocio extraída en el caso de uso, no estamos acoplados a una capa particular de transporte o implementación de controlador. El caso de uso puede ser activado no solo por un controlador, sino también por un evento, un trabajo cron o desde la línea de comandos.

Ejemplo de repositorio

soyjuanmalopez / clean-architecture

Un ejemplo de arquitectura limpia en Java 8 y Spring Boot 2.0 - soyjuanmalopez / clean-architecture

github.com

Estamos en una excelente posición cuando se trata de intercambiar fuentes de datos a diferentes microservicios. Uno de los beneficios clave es que podemos retrasar algunas de las decisiones sobre si queremos y cómo queremos almacenar datos internos de nuestra aplicación. Según el caso de uso de la función, incluso tenemos la flexibilidad de determinar el tipo de almacén de datos, ya sea relacional o de documentos.
Al comienzo de un proyecto, tenemos la menor cantidad de información sobre el sistema que estamos construyendo. No debemos encerrarnos en una arquitectura con decisiones desinformadas que conduzcan a una paradoja del proyecto.
Las decisiones que tomamos tienen sentido para nuestras necesidades ahora y nos han permitido avanzar rápidamente. La mejor parte de la arquitectura limpia es que mantiene nuestra aplicación flexible para futuros requisitos por venir.

Referencia:

- Tom Hombergs: https://8thlight.com/blog/uncle-bob/2012/08/13/the-clean-architecture.html
- Libro del tío Bob: https://www.amazon.com/Clean-Architecture-Craftsmans-Software-Structure/dp/0134494164

Gracias a:

- Clàudia Aymerich por sus valiosos comentarios y colaboración en el código fuente.

Java Bota de primavera Desarrollo de software Arquitectura limpia Arquitectura de software