PROYECTO FINAL

TALLER: PATRONES DE DISEÑO

Análisis de Clean Arquitecture

Profesor: Enrique Aguilar

Por: Daniel Rubi ALvarado

18/05/2024

Índice

Definiciones

Patrones de Diseño

Los patrones de diseño son soluciones típicas a problemas comunes en el diseño de software. Son como plantillas que se pueden aplicar a situaciones recurrentes para ayudar a los desarrolladores a resolver problemas de manera más eficiente y efectiva.

Los patrones de diseño no son implementaciones concretas, sino más bien descripciones de cómo estructurar el código para resolver un problema particular dentro de un contexto dado.

Existen 3 tipos de patrones de diseño principales

Patrones Creacionales: Estos patrones proporcionan mecanismos para crear objetos de manera que se pueda asegurar la flexibilidad y reutilización del código. Se enfocan en la forma en que los objetos son creados y en cómo pueden ser desacoplados del sistema.

Patrones Estructurales: Estos patrones se ocupan de la composición de clases y objetos. Utilizan la herencia para componer interfaces y definen formas para componer objetos para obtener nuevas funcionalidades.

Patrones de Comportamiento: Estos patrones se centran en cómo las clases y los objetos interactúan y se comunican entre sí. Describen no solo los patrones de objetos o clases, sino también los patrones de comunicación entre ellos.

Principios SOLID

Los principios SOLID son un conjunto de cinco principios de diseño de software que tienen como objetivo mejorar la calidad del código, haciendo que sea más mantenible, extensible y flexible. Estos principios fueron popularizados por Robert C. Martin (Uncle Bob) y se utilizan ampliamente en la programación orientada a objetos.

Dichos principios son:

1. **Single Responsibility Principle (SRP) - Principio de Responsabilidad Única**

Cada clase debe tener una única responsabilidad, es decir, una única razón para cambiar. En otras palabras, una clase debe estar encargada de una sola tarea o funcionalidad.

2. **Open/Closed Principle (OCP) - Principio de Abierto/Cerrado**

Las entidades de software (clases, módulos, funciones, etc.) deben estar abiertas para la extensión, pero cerradas para la modificación. Esto significa que el comportamiento de una entidad se puede extender sin modificar su código fuente.

3. **Liskov Substitution Principle (LSP) - Principio de Sustitución de Liskov**

Los objetos de una clase derivada deben ser reemplazables por objetos de la clase base sin alterar el correcto funcionamiento del programa. En otras palabras, las clases hijas deben ser capaces de reemplazar a sus clases padres sin que el comportamiento del sistema cambie.

4. **Interface Segregation Principle (ISP) - Principio de Segregación de Interfaces**

Los clientes no deberían verse obligados a depender de interfaces que no utilizan. Es mejor tener muchas interfaces específicas y pequeñas en lugar de una sola interfaz grande y monolítica.

5. **Dependency Inversion Principle (DIP) - Principio de Inversión de Dependencia**

Los módulos de alto nivel no deben depender de módulos de bajo nivel, ambos deben depender de abstracciones. Las abstracciones no deben depender de los detalles, los detalles deben depender de las abstracciones.

Clean Arquitecture

La Clean Architecture, propuesta por Robert C. Martin (también conocido como "Uncle Bob"), es un estilo de arquitectura de software que enfatiza la organización de código de una manera que separa claramente las responsabilidades y facilita la mantenibilidad, escalabilidad y testabilidad de las aplicaciones.

La idea central de la Clean Architecture es que el sistema debe estar estructurado en capas, con una dependencia clara de adentro hacia afuera y con el núcleo de la aplicación siendo independiente de los detalles externos como la interfaz de usuario, bases de datos y frameworks.

Estructura de la Clean Arquitecture.

La Clean Architecture se representa comúnmente como una serie de anillos concéntricos, cada uno con un conjunto de responsabilidades distinto:

1. **Entidades (Entities)**: Representan las reglas de negocio más generales y de más alto nivel. Son independientes de cualquier detalle técnico y se utilizan en toda la aplicación. Ejemplos incluyen clases de modelo y reglas de validación de negocio.
2. **Casos de Uso (Use Cases)**: Contienen la lógica de aplicación específica. Los casos de uso orquestan la interacción entre las entidades y otros componentes para realizar operaciones específicas de la aplicación. Cada caso de uso representa una funcionalidad completa y bien definida del sistema.
3. **Adaptadores (Interface Adapters)**: Incluyen los componentes que convierten los datos de un formato que puede ser utilizado por los casos de uso y entidades a un formato que puede ser utilizado por la interfaz de usuario, bases de datos, o cualquier otro sistema externo. Aquí es donde se implementan los controladores y presentadores.
4. **Frameworks y Drivers**: Es la capa más externa que contiene los detalles de implementación de frameworks, bases de datos, APIs externas, y otros componentes de infraestructura. La idea es que cualquier cambio en esta capa tenga un impacto mínimo en las capas internas.

SRC
Index

Patrones de Diseño

- Singleton
  La exportación de ‘app’ sirve para crear un único punto de acceso a la instancia del servidor express, si bien no es un uso explícito de Singleton, sigue un principio similar al generar una única instancia del servidor.

- Adapter
  ‘v1Routes’ funciona como un adaptador en ‘startExpressServer’ permitiendo que las rutas definidas en v1Routes se conecten con la aplicación Express, sin necesidad de hacer modificaciones.

Principios SOLID

- Principio de Responsabilidad Única.
  src/indext.ts tiene una única responsabilidad: iniciar y configurar el servidor de Express. Toda la lógica se encuentra dentro de la función ‘startExpressServer’.

- Principio de Abierto/Cerrado.
  El uso de la función ‘startExpressServer’ nos permite que la configuración del servidor sea extensible sin modificar su implantación interna, por ejemplo, agregar nuevas rutas sin cambiar el código interno a través de la configuración ‘startExpressServer’.

- Principio de Inversión de Dependencias
  El uso de ‘reflect-metadata’ y los contenedores de inyección de dependencias indican que las dependencias están siendo invertidas adecuadamente. En lugar de dejar que las clases definan sus dependencias estas se inyectan.

Common

Principios SOLID

- Principio de Responsabilidad Única
  ‘AuthService.ts’ y ‘CacheService.ts’ siguen este principio, ya que cada uno tiene una responsabilidad específica: el servicio de autenticación y el servicio de caché, respectivamente.

  ‘util.ts’ contiene funciones de una única responsabilidad, como transformar cadenas y generar claves hash.

- Principio de Abierto/Cerrado
  En caso de requerirse otros métodos de autentificación, podrían ser perfectamente añadidos dentro de los archivos, sin necesidad de modificar los existentes, ya que cada uno tiene una tarea especifica asignada.

- Principio de Segregación de Interfaces
  En el archivo AuthService.ts, la interfaz IAuthService proporciona una metodos específicos para las clases que necesitan implementar solo métodos relacionados con la autenticación, evitando así que se implementen métodos innecesarios.

- Principio de Inversión de Dependencias
  El código proporcionado hace uso de la inversión de dependencias mediante el uso de interfaces. Por ejemplo, AuthService implementa IAuthService, lo que permite que las clases dependan de la interfaz en lugar de una implementación concreta, lo que facilita la sustitución y extensión de los servicios.

Config

Patrones de Diseño

- Patron Singleton
  Tanto ‘db/redis.ts’ y ‘db/sql.ts’ utilizan una variable global para almacenar la conexión (redisClient y client, respectivamente). Esto asegura que solo haya una instancia de la conexión a la base de datos en toda la aplicación.

Principios SOLID

- Principio de Responsabilidad Única
  Este principio se puede observar en los archivos ‘db/redis.ts’ y ‘db/sql.ts’, donde cada archivo tiene una única responsabilidad: manejar la conexión y las operaciones relacionadas con la base de datos.

  En el caso de inversify/index.ts, su responsabilidad es configurar las dependencias de la aplicación.

- Principio de Abierto/Cerrado
  Este principio aparece en el archivo ‘**inversify/index.ts’**, donde se define la configuración para la inversión de dependencias. Las dependencias se definen a través de interfaces o símbolos en lugar de clases concretas, lo que permite una inyección de dependencias flexible y facilita la sustitución de implementaciones

Constants

Principios SOLID

- Principio de Responsabilidad Única
  Cada archivo dentro de la carpeta "constants" tiene una única responsabilidad: definir y exportar constantes relacionadas con un aspecto específico de la aplicación. Por ejemplo, code-errors.ts se encarga de definir códigos de error.

- Principio de Inversión de Dependencias
  Este principio se puede ver en el archivo index.ts, donde se exportan las constantes utilizando la sintaxis de exportación. Esto permite que los que hagan uso de estas constantes dependan de las abstracciones en lugar de las implementaciones concretas.

Controllers

Patrones de Diseño

- Patron Observer
  Dentro de user/sing-up se puede observar una especie de patrón similar a observer o que sigue un concepto similar ya que permite el registro o suscripción de un usuario a las solicitudes HTTP.

Principios SOLID

- Principio de Responsabilidad Única
  Cada controlador en esta carpeta tiene una única responsabilidad: manejar una acción específica relacionada con las solicitudes HTTP. Por ejemplo, UserSignupController maneja la lógica de registro de usuarios y UserSigninController maneja la lógica de inicio de sesión de usuarios. Esto sigue el principio SRP al tener una única razón para cambiar.

- Principio de Abierto/Cerrado
  Se pueden agregar nuevos controladores con nuevas funcionalidades sin alterar el comportamiento de los controladores existentes

- Principio de Inversión de Dependencias
  Los controladores dependen de abstracciones en lugar de implementaciones concretas. Se utilizan los símbolos definidos en @example-api/config/symbols para realizar la inyección de dependencias.

Data-access

Patrones de Diseño

- Patron Proxy
  En el archivo cache/redis-impl.ts, la clase RedisImpl actúa como un proxy para interactuar con Redis. En lugar de interactuar directamente con el cliente de Redis, otros componentes de la aplicación interactúan con RedisImpl, que a su vez maneja las operaciones de cacheado.

- Patrón Memoization
  Este patrón de diseño no fue visto en clase, pero sirve para almacenar los resultados de funciones que requieren cálculos muy costosos para no tener que volver a calcularlos, En el método memoize de la clase RedisImpl, se implementa el patrón de memoization para cachear los resultados de las funciones. Esto mejora el rendimiento al evitar la ejecución repetida de funciones con los mismos argumentos, devolviendo el resultado almacenado en la caché si está disponible.

- Patron Singleton:
  la configuración de la clase ‘RedisImpl’ y la inyección de dependencias a través de aseguran que solo haya una instancia de la clase RedisImpl en toda la aplicación

Principios SOLID

- Principio de Responsabilidad Única
  Cada clase dentro de esta carpeta tiene una única responsabilidad. Por ejemplo, RedisImpl interactua con Redis para operaciones de caché, mientras que UserRepositoryImpl interactua con la base de datos para operaciones relacionadas con los usuarios.

- Principio de Sustitución de Liskov
  En ambas clases (RedisImpl y UserRepositoryImpl), las implementaciones pueden ser utilizadas en lugar de sus abstracciones sin cambiar el comportamiento del sistema. Por ejemplo, si se quisiera cambiar el uso Redis por otra forma de almacenamiento en caché, se podría hacer sin modificar el código que utiliza la interfaz CacheService.

- Principio de Segregación de Interfaces
  Las interfaces (CacheService y UserRepository) están diseñadas para ser específicas, proporcionando solo los métodos necesarios para tarea. Esto evita que las clases dependan de métodos que no utilizan.

- Principio de Inversión de Dependencias
  Ambas clases (RedisImpl y UserRepositoryImpl) dependen de abstracciones en lugar de implementaciones concretas. Esto se ve en la inyección de dependencias y el uso de interfaces para las dependencias.

Domain

Patrones de Diseño

- Patron Observer:
  Al igual que en la carpeta controllers se puede observar una especie de patrón similar a observer dentro de su propio código de ‘sign-up’ donde podemos registrar o suscribir a un nuevo usuario al sistema.

Principios SOLID

- Principio de Responsabilidad Única
  Cada clase en la carpeta domain tiene una responsabilidad específica y bien definida. Por ejemplo, en ‘sign-in.ts’ y ‘sign-up.ts’ se encuentran las clases ‘Signin’ y ‘Signup’ que se encargan de la lógica de inicio de sesión y de registro de usuarios respectivamente.

- Principio de Abierto/Cerrado
  Las interfaces y clases en domain se pueden extender para añadir nuevas funcionalidades sin modificar el código existente.

- Principio de Segregación de Interfaces
  Las interfaces en domain son específicas y enfocadas, asegurando que las clases dependan solo de lo que realmente necesitan. Por ejemploUserRepository define métodos específicos para las operaciones que deben realizarse sobre los usuarios, sin incluir métodos innecesarios.

- Principio de Inversión de Dependencias
  Ambos deben depender de abstracciones. En este caso, Signin y Signup dependen de abstracciones (UserRepository, AuthService)

Interfaces

Principios SOLID

- Principio de Responsabilidad Única
  Cada interfaz tiene una responsabilidad definida. Por ejemplo, ICustomError se encarga de definir la estructura de los errores personalizados

- Principio de Abierto/Cerrado
  Las interfaces permiten agregar nuevas implementaciones sin modificar las existentes. Por ejemplo, se pueden definir nuevas clases que implementen ICustomError sin cambiar la definición de la interfaz.

- Principio de Sustitución de Liskov
  Las interfaces permiten asegurar que cualquier implementación de ICustomError o IGetEntityOlimpoRepository puede ser utilizada indistintamente sin romper la funcionalidad esperada del programa.

- Principio de Segregación de Interfaces
  Se crean interfaces específicas como ICustomError, IJsonObject, IGetEntityOlimpoRepository y IRequestOlimpoRepository, cada una con un propósito específico.

Platform

Patrones de Diseño

- Patron adapter
  La función startExpressServer configura y adapta varias funcionalidades y middleware de Express para iniciar el servidor con las configuraciones necesarias.

- Singlenton
  En ‘config/index.ts’ se utiliza un patron singleton para exportar una única instancia de configuración, asegurando que en toda la aplicación se esté utilizando la misma

Principios SOLID

- Principio de Responsabilidad Única
  Cada una de las clases de los archivos se encargan de una única tarea, por ejemplo, en ‘general-error.ts’ se encuentra la clase ‘CustomError’ que tiene como única tarea manejar errores personalizados, o en ‘index.ts’ y ‘types.ts’ que se encargan de manejar la configuración de la aplicación y los tipos de configuración respectivamente.

- Principio de Abierto/Cerrado
  En ‘general-error.ts’ la clase ‘CustomError’ puede extenderse mediante la herencia para crear nuevos tipos de errores sin modificar la clase original.

- Principio de Sustitución de Liskov
  En ‘general-error.ts’ La clase ‘CustomError’ extiende de Error y puede ser utilizada en cualquier lugar donde se espere un Error estándar.

- Principio de Segregación de Interfaces
  Definición de interfaces específicas como ‘CustomRequest’ y ‘IStartOptions’ que tienen campos claramente definidos y específicos para su propósito, asegurando que las interfaces sean pequeñas y enfocadas.

- Principio de Inversión de Dependencias
  La configuración que se encuentra en ‘index.ts’ depende de la abstracción Config, que está definida en types.ts

  validate-token y error-handler dependen de las abstracciones CustomRequest y ICustomError en lugar de depender de implementaciones específicas.

Routes/v1

Principios SOLID

- Principio de Responsabilidad Única
  Cada archivo cumple con una única responsabilidad, El ‘index.ts’ se encarga de definiar y exportar las rutas de la API, mientras que el archivo ‘users.ts’ se centra en las rutas relacionadas con las operaciones del usuario.

- Principio de Inversión de Dependencias
  Se puede observar en el archivo ‘user.ts’ de util se utiliza la inyección de dependencias para obtener instancias de los controladores ‘UserSignupController’ y ‘UserSignupController’.

Útil

Patrones de Diseño

- Patrón Memoization
  En el caso de la carpeta util dentro de ‘util/promises.ts’ existe una función llamada ‘memoizePromise’ que sirve para llamar un resultado anterior a una consulta si se ingresan los mismos resultados, esto con ayuda de cache.

- Patrón Promise
  Este patrón nos permite que el programa se siga ejecutando aun cuando se está realizando una solicitud de conexión, en este caso igual en ‘util/promises.ts’ existen dos funciones que aprovechan este patrón.

  Primero la función ‘sequentialPromises’ que realiza de manera secuencial (uno a uno) diferentes promise almacenándolas en un acumulador. Esto ayuda a no sobrecargar el sistema con solicitudes, permitiendo que siga ejecutándose sin interrupciones.

  Después en la función ‘executeArrayPromises’ se ejecuta un array de promise de forma asíncrona, registra los errores de las promise y retorna únicamente las que se ejecutaron correctamente, aplanando las promise que regresen como arreglo.

Principios SOLID

- Principio de Responsabilidad Única
  En ‘Util/promises’ se cumple este principio ya que cada función se encarga exclusivamente de una tarea. ‘memoizePromise’ se encarga de memoizar resultados de una función, ‘sequentialPromises’ de ejecutar promise de manera secuencial y ‘executeArrayPromises’ ejecuta un array de promesas de forma asincrona.

- Principio de Abierto/Cerrado
  Al existir funciones con tareas especificas nos permite generar nuevas funciones que cumplan con las nuevas actividades que se quieran implementar, sin afectar a las ya existentes

- Principio de Segregación de Interfaces
  Cada función proporciona una interfaz clara y especifica para realizar una tarea en específico, con esto se evita la dependencia de interfaces grandes, esto se relaciona con el principio de tener interfaces pequeñas que no afecten a otras.

Clean Arquitecture

Entidades

Las entidades representan los objetos del negocio y las reglas de negocio fundamentales. Estas deben ser independientes de cualquier otra capa.

Definición de entidades: Las entidades se encuentren en la carpeta src/domain. Allí se encuentran los usuarios, sus repositorios y los servicios de inicio de sesión e inscripción. Aunque en src/constants también se podrían encontrar entidades relacionadas al juego

Casos de Uso

Los casos de uso contienen la lógica de aplicación y coordinan el flujo de datos entre las entidades y otras capas.

Servicios: Los servicios en src/service representan los casos de uso del sistema. Aquí es donde se debe poner la lógica de negocio específica de la aplicación.

Controladores: En src/controllers, los controladores actúan como intermediarios entre las rutas (o la capa de presentación) y los servicios (o la capa de casos de uso).

Adaptadores de interfaces

Esta capa se encarga de convertir datos entre el formato que maneja el sistema y el formato necesario para las capas externas, como la web o la base de datos.

Repositorios: En src/repository, los repositorios actúan como adaptadores para acceder a la base de datos. Implementan interfaces para interactuar con las entidades.

Controladores: También en src/controllers, los controladores adaptan las peticiones HTTP para interactuar con los servicios.

Framworks y Drivers

Esta es la capa más externa, que contiene los detalles específicos del framework o tecnología utilizados (por ejemplo, Express, TypeORM, etc.).

Rutas: En src/routes, se definen las rutas de la API, que son específicas del framework (por ejemplo, Express).

Configuración de Inversify: En src/config/inversify, se configura la inyección de dependencias, lo que permite el desacoplamiento entre las diferentes capas.

Configuración de base de datos: En src/config/db, se maneja la configuración y conexión a la base de datos.

Servidor Express: En src/platform/index, se define la configuración y el inicio del servidor Express.

Migraciones y seeding de base de datos: En template-nodejs/db/migrations, se encuentran las migraciones que configuran la estructura de la base de datos.

Conclusión

El uso correcto y aprovechamiento de la Clean Architecture, los principios SOLID y los patrones de diseño es esencial para desarrollar sistemas de software robustos, mantenibles y escalables. Estas metodologías y prácticas proporcionan un marco sólido para estructurar y organizar el código, lo que resulta en un desarrollo más eficiente y una base de código más sostenible a largo plazo.