11 Arquitectura Hexagonal y la aplicación de DDD

Introducción a la arquitectura hexagonal

La **arquitectura hexagonal**, también conocida como **Arquitectura de Puertos y Adaptadores**, fue propuesta por Alistair Cockburn en 2005 como una forma de estructurar sistemas de software que fueran más fáciles de entender, mantener y modificar. Su principal objetivo es separar la lógica central del negocio de las dependencias externas, lo que permite que los sistemas sean más flexibles, modulares y fáciles de probar. Esta arquitectura también facilita la sustitución o actualización de componentes externos sin impactar el núcleo del sistema.

1.1 Definición de arquitectura hexagonal

La **arquitectura hexagonal** es un enfoque que busca desacoplar el **núcleo del dominio** (la lógica de negocio) de las dependencias externas, como bases de datos, APIs, interfaces de usuario y sistemas de mensajería. La idea principal es que el **dominio del negocio** debe estar protegido de los detalles técnicos externos y solo debe interactuar con ellos a través de **puertos** (interfaces) claramente definidos. Los **adaptadores** implementan esos puertos y sirven como intermediarios entre el dominio y el mundo externo.

- **Núcleo del dominio:** Es el corazón de la aplicación y contiene toda la lógica de negocio y las reglas del dominio. Es completamente independiente de cualquier tecnología o implementación externa.
- **Puertos:** Son interfaces que definen cómo el núcleo del dominio interactúa con el mundo exterior. Pueden ser puertos de entrada (cómo los usuarios o sistemas externos interactúan con el dominio) o puertos de salida (cómo el dominio interactúa con bases de datos o servicios externos).
- **Adaptadores:** Son implementaciones concretas de los puertos. Conectan el dominio con tecnologías externas, como bases de datos, APIs, interfaces de usuario, etc.

1.2 Origen y contexto histórico

La **arquitectura hexagonal** fue introducida por Alistair Cockburn como una respuesta a los problemas comunes que enfrentaban las arquitecturas tradicionales, como el **acoplamiento excesivo** entre la lógica de negocio y la infraestructura técnica. Cockburn observó que cuando el código del negocio está fuertemente acoplado a bases de datos, frameworks o servicios externos, se vuelve difícil de mantener, probar y evolucionar. Al diseñar un sistema con arquitectura hexagonal, se asegura que el dominio del negocio esté protegido de estos detalles técnicos, haciéndolo más flexible y adaptable a cambios.

1.3 Objetivo principal: Desacoplar el núcleo del negocio de las dependencias externas

El principal objetivo de la **arquitectura hexagonal** es permitir que el **núcleo del dominio** evolucione de manera independiente de las tecnologías externas. Esto significa que los

detalles técnicos (bases de datos, frameworks, APIs) pueden cambiar o actualizarse sin afectar el dominio, y viceversa.

- **Desacoplamiento:** La arquitectura hexagonal separa las preocupaciones técnicas (cómo el sistema interactúa con el exterior) de las preocupaciones del negocio (las reglas y lógica que definen el dominio). Esto permite que las capas técnicas y del negocio evolucionen de manera independiente.
- **Sustitución de componentes:** Un beneficio clave es la capacidad de reemplazar o actualizar fácilmente los componentes externos sin afectar la lógica de negocio. Por ejemplo, se puede cambiar una base de datos por otra o una API por otra sin alterar el núcleo del sistema.

1.4 Visión general del modelo hexagonal

La arquitectura hexagonal visualiza el sistema como un **hexágono**, donde el núcleo del dominio se encuentra en el centro, rodeado por puertos que actúan como interfaces entre el dominio y los componentes externos. Cada puerto tiene adaptadores que permiten que el sistema interactúe con las tecnologías externas. Este modelo promueve la **inversión de dependencias**, donde el dominio no depende de los adaptadores, sino que estos dependen del dominio.

- **Núcleo en el centro:** El núcleo del dominio es autónomo y no tiene dependencias directas con la infraestructura externa.
- **Puertos como puntos de entrada/salida:** Los puertos definen las interfaces para interactuar con el sistema. Los puertos de entrada son utilizados por actores externos (como usuarios o servicios) para interactuar con el sistema, mientras que los puertos de salida permiten que el sistema interactúe con componentes externos (como bases de datos o APIs).
- **Adaptadores:** Los adaptadores implementan los puertos y gestionan la interacción con tecnologías específicas, como bases de datos, mensajería, o interfaces de usuario.

1.5 Beneficios clave de la arquitectura hexagonal

La arquitectura hexagonal ofrece varios beneficios que la hacen adecuada para sistemas complejos o aquellos que deben adaptarse fácilmente a cambios tecnológicos o de negocio:

- **Mejor testabilidad:** Como la lógica del negocio está completamente desacoplada de las tecnologías externas, es más fácil probar el núcleo del sistema de manera aislada, utilizando mocks o stubs para simular las dependencias externas.
- **Modularidad y flexibilidad:** El desacoplamiento del dominio permite que el sistema sea más modular, facilitando el reemplazo o la actualización de componentes sin afectar otras partes del sistema.
- **Evolución del sistema:** La arquitectura hexagonal facilita la evolución de un sistema a medida que cambian las necesidades del negocio o la tecnología, permitiendo la integración de nuevos adaptadores sin modificar el dominio.
- **Fácil integración con microservicios:** Esta arquitectura es ideal para sistemas distribuidos, como microservicios, donde cada servicio puede tener su propio núcleo de dominio desacoplado de los detalles técnicos externos.

Principio de la arquitectura hexagonal

La **arquitectura hexagonal** se basa en un conjunto de principios que permiten desacoplar el núcleo de la lógica de negocio de los sistemas externos y las tecnologías que lo rodean. El enfoque central de esta arquitectura es organizar el sistema de tal manera que los cambios en la infraestructura o las interfaces externas no afecten la lógica central del negocio. Este principio permite crear aplicaciones más **modulares**, **flexibles**, y **fácilmente mantenibles**.

2.1 Separación entre el núcleo del negocio y las infraestructuras externas

El principio más importante de la arquitectura hexagonal es la clara separación entre la lógica del negocio (el **núcleo**) y cualquier dependencia externa, como bases de datos, interfaces de usuario, APIs, o servicios externos. Esta separación ayuda a mantener el sistema flexible, de manera que las decisiones sobre tecnología, bases de datos o frameworks puedan cambiar sin tener que modificar el núcleo del sistema.

- **Núcleo autónomo:** El núcleo del negocio debe ser completamente independiente de los detalles técnicos. Esto permite que el código del dominio esté centrado únicamente en resolver problemas de negocio sin tener que preocuparse por cómo se persisten los datos o cómo se muestran al usuario.
- **Interacción a través de puertos:** Para conectar el núcleo con las infraestructuras externas, se utilizan **puertos** (interfaces). Estos puertos definen cómo los sistemas externos interactúan con el dominio, pero mantienen el núcleo protegido de las implementaciones técnicas.

2.2 Definición de puertos y adaptadores

La **arquitectura hexagonal** introduce los conceptos de **puertos** y **adaptadores** para manejar la interacción entre el dominio y el mundo exterior. Este es uno de los principios clave que permite el desacoplamiento entre la lógica del negocio y las infraestructuras técnicas.

- **Puertos:** Son interfaces que definen cómo interactuar con el dominio. Los puertos representan tanto los puntos de **entrada** como los puntos de **salida** de la aplicación:
    - **Puertos de entrada:** Definen cómo los actores externos, como usuarios o sistemas, pueden interactuar con el dominio. Estos puertos exponen las funcionalidades del dominio a través de servicios o controladores.
        - **Ejemplo:** En un sistema de ventas, un puerto de entrada podría ser una API REST que expone operaciones para crear, modificar o consultar pedidos.
    - **Puertos de salida:** Definen cómo el dominio interactúa con sistemas externos, como bases de datos, sistemas de mensajería o servicios de terceros.
        - **Ejemplo:** Un puerto de salida podría definir una interfaz para persistir pedidos en una base de datos, sin que el dominio conozca los detalles específicos de la base de datos subyacente.

- **Adaptadores:** Son las implementaciones concretas de los puertos. Los adaptadores permiten conectar los puertos con tecnologías específicas, como bases de datos, APIs, interfaces de usuario, etc.
    - **Adaptadores de entrada:** Implementan puertos de entrada y son responsables de transformar las solicitudes externas en operaciones sobre el dominio.
        - **Ejemplo:** Un controlador REST que recibe solicitudes HTTP y las traduce a comandos para el dominio es un adaptador de entrada.
    - **Adaptadores de salida:** Implementan puertos de salida y permiten que el dominio interactúe con infraestructuras técnicas como bases de datos, servicios de mensajería o APIs externas.
        - **Ejemplo:** Un repositorio que implementa una interfaz de persistencia es un adaptador de salida que permite guardar y recuperar datos del sistema.

2.3 Inversión de dependencias

El principio de **inversión de dependencias** es fundamental en la arquitectura hexagonal. En este enfoque, las dependencias tradicionales se invierten: en lugar de que la lógica de negocio dependa de las implementaciones técnicas (por ejemplo, una base de datos o un servicio externo), son los detalles técnicos los que dependen del núcleo del negocio.

- **Dependencias dirigidas hacia el núcleo:** El núcleo del dominio nunca depende de detalles técnicos externos. En su lugar, los adaptadores externos dependen de las interfaces (puertos) definidas por el dominio para interactuar con él.
    - **Ejemplo:** El núcleo del dominio define una interfaz `RepositorioDePedidos`, pero no sabe ni le importa si esta interfaz está implementada mediante una base de datos SQL, NoSQL o cualquier otra tecnología. Es el adaptador quien debe implementar esta interfaz y encargarse de la persistencia.
- **Simplicidad en el cambio de infraestructura:** Este principio facilita el cambio de infraestructura sin tener que modificar la lógica central del negocio. Si se quiere cambiar la base de datos o la interfaz de usuario, solo se necesita cambiar el adaptador correspondiente, manteniendo intacto el núcleo del sistema.

2.4 Ejemplos de puertos en la arquitectura hexagonal

Los **puertos** en la arquitectura hexagonal son interfaces que permiten que el sistema se comunique con el mundo exterior, ya sea para recibir información o para enviar datos a otras aplicaciones o sistemas. Estos puertos permiten mantener la independencia entre la lógica de negocio y las infraestructuras externas.

- **Puerto de entrada:**
    - Un puerto de entrada puede ser una interfaz que define cómo interactúan los usuarios o sistemas externos con la aplicación.
    - **Ejemplo:** En una aplicación de pedidos, un puerto de entrada puede ser una interfaz `GestorDePedidos` que define métodos como `crearPedido()`, `actualizarPedido()`, o `consultarPedido()`. Estos métodos permiten

que los controladores o servicios externos interactúen con el dominio, sin conocer los detalles internos.

- **Puerto de salida:**
  - Un puerto de salida permite que el núcleo del dominio interactúe con servicios o sistemas externos.
  - **Ejemplo:** Un puerto de salida puede ser una interfaz `RepositorioDeClientes` que define cómo se almacenan o recuperan los datos de los clientes, sin especificar si se utiliza una base de datos SQL, NoSQL o incluso un servicio remoto.

2.5 Ejemplos de adaptadores en la arquitectura hexagonal

Los **adaptadores** son las implementaciones concretas de los puertos. Se encargan de transformar las interacciones externas en operaciones sobre el dominio y de traducir los resultados del dominio para que puedan ser entendidos por sistemas externos.

- **Adaptador de entrada:**
  - Los adaptadores de entrada convierten las solicitudes de los usuarios o sistemas externos en comandos o consultas sobre el dominio.
  - **Ejemplo:** Un controlador REST implementa un adaptador de entrada al recibir una solicitud HTTP, transformar los datos en una llamada a la lógica del dominio, y devolver una respuesta.
- **Adaptador de salida:**
  - Los adaptadores de salida implementan los puertos de salida y permiten que el dominio interactúe con infraestructuras externas.
  - **Ejemplo:** Un adaptador de salida para persistencia puede implementar un `RepositorioDePedidos` utilizando una base de datos MySQL o MongoDB para almacenar los pedidos.

2.6 Ventajas de la separación de responsabilidades

Al separar claramente las responsabilidades entre el núcleo del negocio, los puertos y los adaptadores, la arquitectura hexagonal proporciona numerosas ventajas:

- **Facilidad para realizar pruebas unitarias:** Al estar el núcleo del negocio desacoplado de las infraestructuras externas, es mucho más fácil realizar pruebas unitarias, utilizando mocks o stubs para simular los adaptadores.
- **Flexibilidad para cambiar infraestructuras:** Si se desea cambiar de base de datos, sistema de mensajería o framework de interfaz de usuario, solo es necesario modificar o sustituir los adaptadores correspondientes, sin afectar el núcleo de la aplicación.
- **Desarrollo modular:** Los desarrolladores pueden trabajar en diferentes partes del sistema de manera independiente, mejorando la productividad y facilitando la colaboración en equipos grandes.

Módulos de la arquitectura hexagonal

La arquitectura hexagonal organiza un sistema en **módulos** claramente definidos que separan la lógica de negocio de las dependencias externas. Estos módulos incluyen el **núcleo del dominio**, los **puertos** que actúan como interfaces, y los **adaptadores** que implementan esas interfaces. La división en módulos asegura que cada parte del sistema cumpla una función específica y no dependa de otras capas innecesariamente.

3.1 Capa del dominio

La **capa del dominio** es el núcleo del sistema en la arquitectura hexagonal. Contiene toda la lógica de negocio y las reglas que rigen el comportamiento de la aplicación. La capa del dominio es completamente independiente de cualquier infraestructura técnica externa, lo que garantiza que no dependa de tecnologías como bases de datos, interfaces de usuario o APIs externas.

- **Responsabilidad:** Esta capa está enfocada exclusivamente en el negocio, y sus entidades y objetos están diseñados para reflejar el comportamiento del dominio sin considerar cómo se almacenan o cómo se interactúa con ellos desde fuera.
- **Entidades y objetos de valor:** El núcleo del dominio se compone de **entidades**, **objetos de valor** y **servicios de dominio**. Estos elementos encapsulan las reglas del negocio y aseguran que las interacciones y operaciones sean coherentes con el modelo de negocio.
    - **Ejemplo:** En un sistema de pedidos, la entidad `Pedido` y los objetos de valor como `Producto` y `Dirección` forman parte del dominio. El comportamiento de estas clases define cómo se procesan los pedidos y cómo se gestionan las operaciones comerciales.
- **Servicios de dominio:** Los **servicios de dominio** encapsulan lógica de negocio que no pertenece directamente a ninguna entidad u objeto de valor específico, sino que afecta a múltiples entidades.
    - **Ejemplo:** Un `ServicioDeEnvio` podría manejar la lógica de cálculo y selección de opciones de envío basándose en las reglas del negocio, sin depender de ninguna tecnología externa.

3.2 Puertos: Interfaz entre la lógica del dominio y el mundo externo

Los **puertos** son interfaces que permiten la comunicación entre el núcleo del dominio y los sistemas externos. Actúan como un **punto de entrada** o **salida** del sistema, definiendo cómo los actores externos (usuarios, APIs) interactúan con el dominio y cómo el dominio interactúa con otras infraestructuras externas, como bases de datos o servicios de terceros.

- **Puertos de entrada:** Son los puntos por los cuales los actores externos (usuarios, APIs, sistemas) pueden acceder al núcleo del dominio. Definen las operaciones disponibles para interactuar con la aplicación.
    - **Ejemplo:** Un puerto de entrada en un sistema de comercio electrónico podría ser una interfaz `GestorDePedidos`, que define métodos como `crearPedido()`, `actualizarPedido()` o `consultarEstadoDelPedido()`.

- **Puertos de salida:** Permiten que el dominio interactúe con infraestructuras externas, como sistemas de almacenamiento, servicios de mensajería o servicios de terceros. Los puertos de salida encapsulan las interacciones con estos sistemas y aseguran que el núcleo del dominio no dependa de implementaciones específicas.
    - **Ejemplo:** En el sistema de comercio electrónico, un puerto de salida podría ser la interfaz `RepositorioDePedidos`, que define operaciones para guardar o recuperar pedidos de una base de datos, sin que el dominio sepa qué tipo de base de datos se está utilizando.

3.3 Adaptadores: Implementaciones concretas de los puertos

Los **adaptadores** son las implementaciones de los puertos, y su función es conectar el dominio con tecnologías externas. Los adaptadores pueden ser de entrada o de salida, dependiendo de si su tarea es permitir la interacción con actores externos (usuarios, APIs) o gestionar la interacción con sistemas externos (bases de datos, servicios externos, etc.).

- **Adaptadores de entrada:** Implementan los puertos de entrada y se encargan de transformar las solicitudes de los actores externos en comandos o consultas que el dominio pueda procesar. Los adaptadores de entrada son responsables de traducir la interacción externa en una operación interna sobre el dominio.
    - **Ejemplo:** Un controlador REST (`PedidoController`) que recibe solicitudes HTTP para crear o consultar pedidos es un adaptador de entrada. Convierte las solicitudes HTTP en operaciones sobre el `GestorDePedidos` (el puerto de entrada).
- **Adaptadores de salida:** Implementan los puertos de salida y permiten que el dominio interactúe con sistemas externos, como bases de datos, sistemas de mensajería o servicios de terceros. Los adaptadores de salida aseguran que el dominio no dependa directamente de la infraestructura técnica.
    - **Ejemplo:** Un adaptador de salida (`RepositorioDePedidosMySQL`) puede implementar un puerto de salida (`RepositorioDePedidos`) utilizando una base de datos MySQL para guardar y recuperar pedidos.

3.4 Distribución de los módulos en un proyecto

La **arquitectura hexagonal** facilita la organización modular de un sistema, donde cada componente tiene responsabilidades bien definidas. En un proyecto basado en **Spring Boot**, los módulos de la arquitectura hexagonal pueden organizarse de la siguiente manera:

- **Módulo de dominio:** Contiene el núcleo del dominio (entidades, objetos de valor, servicios de dominio). Este módulo no tiene dependencias con la infraestructura externa, y solo depende de las interfaces de los puertos.
- **Módulos de puertos:** Los puertos pueden organizarse en submódulos dentro del sistema. Estos submódulos contienen las interfaces que definen cómo los adaptadores externos interactúan con el dominio.
    - **Puertos de entrada:** Definen las interfaces por las cuales los actores externos (controladores, APIs) interactúan con el sistema.

    - **Puertos de salida:** Definen las interfaces que permiten al sistema interactuar con infraestructuras técnicas externas (bases de datos, servicios de mensajería).
- **Módulo de adaptadores:** Los adaptadores de entrada y salida se colocan en un módulo separado que implementa las interfaces definidas en los puertos. Este módulo contiene las dependencias externas necesarias para la interacción con tecnologías como bases de datos, APIs externas o sistemas de mensajería.
    - **Ejemplo:** Un adaptador que utiliza Spring Data JPA para implementar un `RepositorioDePedidos` sería parte del módulo de adaptadores, junto con otros adaptadores, como controladores REST o servicios de integración con APIs externas.

3.5 Ejemplos de cómo distribuir estos módulos en proyectos Spring Boot o microservicios

En un proyecto basado en **Spring Boot**, la arquitectura hexagonal se puede estructurar de la siguiente manera:

- **Módulo del dominio:** Este módulo contiene todas las entidades del negocio y los servicios de dominio.
    - **Ejemplo:** Un servicio de `GestorDePedidos` que maneja la lógica para crear y actualizar pedidos.
- **Módulo de puertos:** Aquí se definen las interfaces que actúan como puertos de entrada y salida. Estas interfaces no tienen dependencias de Spring o cualquier tecnología externa.
    - **Ejemplo:** La interfaz `GestorDePedidos` define las operaciones que permiten crear, modificar y consultar pedidos.
- **Módulo de adaptadores:** Aquí se implementan los adaptadores de entrada y salida. Los controladores REST que gestionan las solicitudes HTTP y los repositorios que interactúan con bases de datos serían parte de este módulo.
    - **Ejemplo:** Un `PedidoController` implementa el adaptador de entrada, mientras que un repositorio que utiliza **Spring Data JPA** es el adaptador de salida para el puerto de persistencia.

Convirtiendo microservicios a arquitectura hexagonal

Migrar o diseñar **microservicios** usando la **arquitectura hexagonal** tiene como objetivo maximizar la modularidad, desacoplar la lógica de negocio de los detalles técnicos, y facilitar la evolución y mantenibilidad del sistema. En una arquitectura de microservicios, cada servicio tiene su propio dominio y debe ser autónomo. Implementar la arquitectura hexagonal en microservicios permite que cada servicio mantenga su independencia mientras se asegura la flexibilidad en la elección o cambio de tecnologías externas.

4.1 Beneficios de aplicar la arquitectura hexagonal a microservicios

Los microservicios ya buscan la **descomposición** del sistema en componentes pequeños y autónomos. La **arquitectura hexagonal** lleva este principio un paso más allá al aplicar una estructura interna clara que separa el dominio de las dependencias externas. Esto tiene varios beneficios clave:

- **Desacoplamiento:** Los microservicios con arquitectura hexagonal permiten que la lógica de negocio esté completamente desacoplada de detalles técnicos como bases de datos, sistemas de mensajería o APIs externas. Esto facilita la migración o sustitución de tecnologías sin afectar el núcleo del negocio.
- **Testabilidad:** Dado que la lógica del dominio no tiene dependencias externas directas, es más fácil realizar pruebas unitarias y de integración, utilizando mocks para simular las interacciones con el exterior.
- **Modularidad:** La arquitectura hexagonal impone una organización interna que favorece la modularidad, permitiendo que cada microservicio sea un módulo autosuficiente que puede desarrollarse y desplegarse de manera independiente.
- **Flexibilidad tecnológica:** Los adaptadores en la arquitectura hexagonal permiten utilizar diferentes tecnologías (bases de datos, APIs) sin modificar el núcleo del sistema, lo que hace que los microservicios sean más flexibles y adaptables a cambios tecnológicos.

4.2 Estrategia para convertir microservicios existentes a la arquitectura hexagonal

Migrar un microservicio existente a la arquitectura hexagonal requiere un enfoque estructurado que garantice una transición sin afectar el comportamiento del sistema. A continuación se describe una estrategia en varios pasos:

- **Paso 1: Identificar el núcleo del dominio y aislarlo de las dependencias externas**
  - El primer paso es analizar el código existente y **extraer la lógica del negocio**. Esto implica identificar las reglas de negocio, las entidades clave y los servicios del dominio. Esta lógica debe aislarse en un **módulo de dominio** que no tenga dependencias con la infraestructura externa.
  - **Ejemplo:** Si tienes un microservicio de gestión de pedidos, identifica los objetos `Pedido`, `Producto` y `Cliente`, y coloca toda la lógica relacionada con la creación, actualización y cancelación de pedidos en el módulo del dominio.
- **Paso 2: Crear puertos para definir la comunicación con el dominio**
  - Define **puertos** (interfaces) que actúen como puntos de entrada y salida para la interacción con el dominio. Los puertos de entrada permiten que el sistema reciba solicitudes de APIs o usuarios, mientras que los puertos de salida gestionan la interacción con bases de datos, servicios de mensajería, o servicios externos.
  - **Ejemplo:** Crea un puerto de entrada llamado `GestorDePedidos`, que defina métodos como `crearPedido()`, y un puerto de salida `RepositorioDePedidos` para manejar la persistencia de los pedidos.
- **Paso 3: Refactorizar los adaptadores para conectar con sistemas externos**
  - Los **adaptadores** son las implementaciones concretas de los puertos que permiten al microservicio interactuar con el exterior. Durante la migración,

refactoriza las dependencias externas existentes (bases de datos, APIs, mensajería) para que interactúen con el dominio a través de los puertos.
    - **Ejemplo:** Si el microservicio utiliza una base de datos SQL para almacenar pedidos, refactoriza el código de acceso a datos en un adaptador que implemente el puerto `RepositorioDePedidos`, de manera que el dominio no tenga acceso directo a la base de datos.

4.3 Uso de Spring Boot y Spring Data para implementar puertos y adaptadores

Spring Boot facilita enormemente la implementación de la arquitectura hexagonal en microservicios. Su enfoque modular y su soporte para la inyección de dependencias encajan perfectamente con los principios de esta arquitectura.

- **Implementación de puertos con interfaces de Spring:** Los puertos de entrada y salida se definen como **interfaces Java**, utilizando las capacidades de inyección de dependencias de Spring. Por ejemplo, un puerto de salida para gestionar la persistencia de pedidos podría definirse como una interfaz `RepositorioDePedidos`.

Ejemplo:

```
public interface RepositorioDePedidos {
    Pedido guardar(Pedido pedido);
    Optional<Pedido> buscarPorId(Long id);
}
```

- **Implementación de adaptadores con Spring Data:** Los **adaptadores de salida** pueden implementarse utilizando **Spring Data**, que facilita la interacción con bases de datos sin tener que escribir código repetitivo. El adaptador puede implementar la interfaz `RepositorioDePedidos` y delegar las operaciones a una base de datos SQL, NoSQL, o cualquier otro sistema de almacenamiento.

Ejemplo:

```
@Repository
public class RepositorioDePedidosImpl implements RepositorioDePedidos {
    @Autowired
    private JpaPedidoRepository jpaPedidoRepository;

    @Override
    public Pedido guardar(Pedido pedido) {
        return jpaPedidoRepository.save(pedido);
    }
```

```
    @Override
    public Optional<Pedido> buscarPorId(Long id) {
        return jpaPedidoRepository.findById(id);
    }
}
```

4.4 Ejemplo práctico de un microservicio que gestiona pedidos utilizando la arquitectura hexagonal

Consideremos un **microservicio de gestión de pedidos** que utiliza la arquitectura hexagonal. El sistema debe gestionar la creación de pedidos, su persistencia en una base de datos y la integración con un servicio de mensajería para notificar cuando un pedido se ha creado.

- **Módulo de dominio:**
    - El núcleo del dominio contiene las entidades `Pedido`, `Cliente` y `Producto`. Además, incluye la lógica de negocio para validar y crear pedidos.

Ejemplo de entidad de dominio:

```
public class Pedido {
    private Long id;
    private Cliente cliente;
    private List<Producto> productos;
    private EstadoPedido estado;

    public Pedido(Cliente cliente, List<Producto> productos) {
        this.cliente = cliente;
        this.productos = productos;
        this.estado = EstadoPedido.PENDIENTE;
    }

    public void completarPedido() {
        if (!productos.isEmpty()) {
            this.estado = EstadoPedido.COMPLETADO;
        } else {
            throw new IllegalStateException("No se puede completar un
pedido sin productos.");
        }
    }
}
```

- **Puertos:**

- Se define el puerto de entrada `GestorDePedidos` para recibir solicitudes de creación de pedidos y el puerto de salida `RepositorioDePedidos` para manejar la persistencia.

Ejemplo de puerto de entrada:

```
public interface GestorDePedidos {
    Pedido crearPedido(Cliente cliente, List<Producto> productos);
}
```

- **Adaptadores:**
  - Un adaptador de entrada, como un **controlador REST**, interactúa con el `GestorDePedidos`. Un adaptador de salida implementa `RepositorioDePedidos` utilizando **Spring Data** para guardar los pedidos en una base de datos.

Ejemplo de controlador REST (adaptador de entrada):

```
@RestController
@RequestMapping("/pedidos")
public class PedidoController {
    private final GestorDePedidos gestorDePedidos;

    @PostMapping
    public Pedido crearPedido(@RequestBody PedidoRequest request) {
        return gestorDePedidos.crearPedido(request.getCliente(),
request.getProductos());
    }
}
```

4.5 Consideraciones al convertir microservicios a arquitectura hexagonal

- **Desarrollo iterativo:** La migración hacia la arquitectura hexagonal puede hacerse de manera iterativa, empezando por los componentes más críticos del sistema y refactorizando gradualmente el resto de los servicios.
- **Monitoreo y pruebas:** Durante la transición, es fundamental mantener un enfoque riguroso en las pruebas y el monitoreo para asegurar que el comportamiento del sistema no cambie de manera inesperada.
- **Compatibilidad con tecnologías:** La arquitectura hexagonal es compatible con una amplia variedad de tecnologías y patrones. Esto significa que se puede integrar fácilmente con microservicios basados en Spring, utilizando herramientas como **Spring Boot**, **Spring Data** y **Spring Cloud**.

Diferencia entre Arquitecturas: Clean vs Hexagonal vs Onion

La **arquitectura hexagonal**, la **arquitectura limpia (Clean Architecture)** y la **arquitectura en capas de cebolla (Onion Architecture)** comparten muchos principios comunes y tienen objetivos similares: desacoplar el dominio del negocio de los detalles técnicos, mantener la independencia de las infraestructuras externas, y facilitar la escalabilidad y mantenibilidad del sistema. Sin embargo, cada una tiene un enfoque diferente en cómo organizar y estructurar el código.

5.1 Comparación entre Arquitectura Limpia, Hexagonal y Onion

Las tres arquitecturas proponen una forma de organizar las aplicaciones de tal manera que la lógica de negocio se mantenga aislada de las infraestructuras y detalles técnicos. La **diferencia clave** entre ellas está en la forma en que organizan las dependencias, los módulos y los flujos de interacción entre las diferentes partes del sistema.

- **Arquitectura Limpia (Clean Architecture):**
  - Propuesta por Robert C. Martin (Uncle Bob), la arquitectura limpia utiliza **capas concéntricas** donde la lógica del negocio está en el centro, rodeada por capas que representan la infraestructura, interfaces de usuario, APIs, etc.
  - En la arquitectura limpia, cada capa solo puede depender de las capas más internas, asegurando una clara inversión de dependencias. La lógica de negocio no depende de ninguna tecnología externa, y la infraestructura depende de la capa de negocio.
  - **Capa central:** Contiene entidades y casos de uso (similar a la lógica del dominio).
  - **Ventajas:** Proporciona una clara separación de responsabilidades y una estructura flexible que puede adaptarse a cambios tecnológicos.
- **Arquitectura Hexagonal (Puertos y Adaptadores):**
  - Propuesta por Alistair Cockburn, esta arquitectura organiza el sistema como un **hexágono**, donde el núcleo del dominio está en el centro y los puertos (interfaces) son los puntos de entrada y salida para interactuar con sistemas externos.
  - La arquitectura hexagonal enfatiza los **puertos y adaptadores** como mecanismos para conectar el dominio con el exterior. Los puertos son interfaces que definen cómo interactuar con el dominio, mientras que los adaptadores implementan estas interfaces para integrar con tecnologías como bases de datos, sistemas de mensajería, APIs, etc.
  - **Ventajas:** Facilita la prueba y la modularidad, ya que el núcleo del negocio no depende de tecnologías externas. Es particularmente útil en sistemas distribuidos y microservicios.
- **Arquitectura Onion (Cebolla):**
  - Propuesta por Jeffrey Palermo, la arquitectura Onion es muy similar a la arquitectura limpia y organiza el sistema en **capas concéntricas** alrededor del núcleo de negocio, de manera similar a las capas de una cebolla.
  - En la arquitectura Onion, la lógica del negocio está en el centro, y las capas externas son responsables de interactuar con las interfaces de usuario,

bases de datos, sistemas de mensajería, etc. Las dependencias solo pueden ir de las capas externas hacia el núcleo.
    - **Ventajas:** Al igual que la arquitectura limpia, promueve una clara separación de responsabilidades y un alto nivel de flexibilidad, asegurando que los detalles técnicos no afecten la lógica del negocio.

5.2 Similitudes entre las arquitecturas

Aunque existen diferencias en la forma en que organizan los componentes y las dependencias, estas tres arquitecturas comparten varios principios comunes:

- **Desacoplamiento del dominio de las infraestructuras externas:** Todas las arquitecturas separan claramente la lógica de negocio de los detalles técnicos, asegurando que las decisiones tecnológicas puedan cambiar sin afectar el núcleo del sistema.
- **Independencia de frameworks:** Las tres arquitecturas evitan que la lógica de negocio dependa de frameworks o tecnologías específicas, permitiendo que el dominio sea autónomo y fácil de testear.
- **Modularidad y escalabilidad:** En todas estas arquitecturas, los sistemas se dividen en módulos o capas, lo que facilita el desarrollo modular, la escalabilidad y la mantenibilidad del sistema a lo largo del tiempo.
- **Enfoque en la inversión de dependencias:** Las tres arquitecturas aseguran que las capas internas (lógica de negocio) no dependan de las externas (infraestructura, frameworks, interfaces de usuario), sino que sean las capas externas las que dependan del núcleo.

5.3 Diferencias clave entre las arquitecturas

- **Arquitectura Limpia:**
    - Organiza el sistema en **capas concéntricas**, donde la capa más interna contiene las **entidades** (modelos del dominio) y la lógica de negocio se encuentra en una capa intermedia (casos de uso).
    - Los adaptadores (infraestructura, controladores, repositorios) se encuentran en las capas más externas, y la dependencia fluye hacia el centro. La **separación de capas** está bien definida y controlada.
- **Arquitectura Hexagonal:**
    - No utiliza un enfoque en capas concéntricas, sino que organiza el sistema en torno a **puertos y adaptadores**. Los puertos son las interfaces que permiten la interacción con el dominio, mientras que los adaptadores implementan esos puertos.
    - El enfoque es más flexible en cuanto a cómo se estructuran las dependencias externas, permitiendo que los adaptadores puedan cambiarse sin afectar al núcleo. Esto es ideal para aplicaciones donde la integración con múltiples servicios externos es crítica.
- **Arquitectura Onion:**
    - Similar a la arquitectura limpia, utiliza **capas concéntricas** donde la lógica de negocio está en el centro. La arquitectura Onion se centra en garantizar que las capas externas interactúan con el núcleo de manera controlada.

    - Las dependencias van de las capas externas (infraestructura) hacia el núcleo, lo que asegura que los detalles técnicos estén siempre separados de la lógica del dominio. Es una buena opción cuando se requiere una organización estricta en capas para la arquitectura del sistema.

5.4 ¿Cuándo usar cada una?

La elección entre estas arquitecturas depende en gran medida del tipo de sistema que se está construyendo y de las prioridades del proyecto:

- **Arquitectura Limpia (Clean Architecture):** Es ideal cuando se necesita una organización estricta de las capas y una estructura clara y comprensible. Funciona bien en aplicaciones grandes o con múltiples módulos, donde se desea mantener un fuerte control sobre las dependencias.
- **Arquitectura Hexagonal (Hexagonal Architecture):** Es particularmente útil en sistemas distribuidos y microservicios, donde es importante tener flexibilidad para cambiar los adaptadores (por ejemplo, diferentes bases de datos o APIs). Si el sistema requiere múltiples formas de interacción (APIs, eventos, etc.), la arquitectura hexagonal facilita esta integración.
- **Arquitectura Onion (Onion Architecture):** Es adecuada para proyectos en los que se busca una clara separación de responsabilidades en capas y donde el enfoque concéntrico facilita el mantenimiento del sistema. La arquitectura Onion es útil cuando se quiere asegurar que las capas externas no afecten a la lógica interna del negocio.

5.5 Ejemplos prácticos de aplicaciones

- **Aplicación web empresarial con arquitectura limpia:** Una aplicación empresarial compleja con múltiples módulos, como un sistema de gestión de inventarios, puede beneficiarse de la arquitectura limpia, donde cada módulo sigue un enfoque en capas claras y bien definidas.
- **Microservicios con arquitectura hexagonal:** Un sistema basado en microservicios que debe interactuar con múltiples servicios externos, como APIs REST, bases de datos NoSQL, y sistemas de mensajería, se beneficiaría de la flexibilidad y modularidad de la arquitectura hexagonal. Cada microservicio puede tener su propio conjunto de puertos y adaptadores, permitiendo que evolucione de manera independiente.
- **Sistema de comercio electrónico con arquitectura Onion:** En un sistema de comercio electrónico donde las reglas del negocio deben estar estrictamente separadas de las infraestructuras externas (interfaces de usuario, sistemas de pago, bases de datos), la arquitectura Onion proporciona un enfoque robusto para organizar las dependencias de forma concéntrica, manteniendo el dominio protegido de los detalles técnicos.