Arquitectura del Software

Jesús Sánchez Cuadrado
Última actualización: 10/02/2026

Este tema se introduce el concepto de arquitectura de software y su importancia en el desarrollo de sistemas. Se presentaran los elementos fundamentales de la arquitectura de software, así como los estilos arquitectónicos más comunes.

Esta primera versión contiene:

- Breve introducción a la arquitectura de software.
- Arquitectura Hexagonal (Ports & Adapters) y su relación con Domain-Driven Design (DDD).

Índice

1. Introducción

La arquitectura del software es la estructura fundamental de un sistema de software, compuesta por sus componentes, las relaciones entre ellos y los principios y decisiones que guían su diseño y evolución.

En otras palabras, describe cómo está organizado el software y cómo interactúan sus partes para cumplir los requisitos funcionales y no funcionales (rendimiento, seguridad, escalabilidad, mantenibilidad, etc.).

En la arquitectura el foco ya no son unidades pequeñas del código (ej., clases) sino que el foco está puesta en unidades más grandes (ej., módulos, servicios, capas, componentes). Por tanto, la arquitectura tiene un granularidad mayor que el diseño de software.

Los elementos implicados en la arquitectura de software son:

- Paquetes
- Módulos
- Componentes
- Capas
- Servicios
- Sistemas y subsistemas

1.1. Definiciones de arquitectura del software

Existen multitud de definiciones de arquitectura del software, más o menos formales. A continuación se presentan algunas de las más conocidas.

1.1.1. Basada en las relaciones entre los elementos

> La arquitectura del software de un sistema es el conjunto de estructuras necesarias para razonar sobre el sistema, las cuales comprenden los elementos de software, las relaciones entre ellos y las propiedades de ambos."
>
> – Software Architecture in Practice, Bass, Clements y Kazman (2013)

1.1.2. Basada en las decisiones principales

> "La arquitectura del software trata de las decisiones importantes, aquellas que son difíciles de cambiar."
>
> – Philippe Kruchten

> "Architecture is about the important stuff. Whatever that is."
>
> – Ralph Johnson

Esta definición pone el foco en la arquitectura refleja las decisiones críticas del proyecto, aquellas que deben ser correctas desde el principio del proyecto.

Esencialmente, la arquitectura del software se refiere a las decisiones fundamentales sobre la estructura y organización de un sistema de software y el impacto de estas decisiones.

1.2. Elementos de la arquitectura del software

Por tanto, la **arquitectura del software** es el conjunto de estructuras necesarias para razonar sobre un sistema software (a alto nivel). Cada estructura está formada por elementos software, relaciones entre ellos y ciertas propiedades de los elementos y relaciones. Estas estructuras principales reflejan las decisiones críticas del proyecto, aquellas que deben ser correctas desde el principio del proyecto.

"Los planos del software", que incluyen:

- Estructura
- Comportamiento
- Interacciones
- Propiedades no funcionales

La arquitectura del software tiene como objetivo abordar los requisitos no funcionales. La arquitectura del software trata con los factores de calidad del software (ver Introducción), como por ejemplo si el software es extensible, tolerante a fallos, seguro, etc.

Pero, ¿por qué los requisitos no funcionales deben abordarse a nivel de arquitectura del software? Por que los requisitos funcionales se pueden abordar agregando, modificando o eliminando código. Sin embargo, los requisitos no funcionales suelen ser "cross-cutting concerns", esto es, afectan a múltiples partes del sistema. Por tanto, requieren una planificación y diseño cuidadosos a nivel arquitectónico para garantizar que se cumplan de manera efectiva en todo el sistema.

1.2.1. Utilidad de la arquitectura del software

Nos ayuda a razonar sobre el sistema (responder preguntas)

- ¿Cuáles son las partes del sistema que pueden cambiar?
- ¿Realizar este cambio "romperá" (afectará) a otra parte del sistema?
- ¿Dónde debo incluir cierta funcionalidad?

- ¿Qué significa esta decisión que hay en el código? Las decisiones arquitecturales terminan llegando al código

2. Arquitectura hexagonal

La arquitectura hexagonal fue propuesta por Alistair Cockburn en 2005, que la ha ido refinando hasta su versión actual denominada "Arquitectura de puertos y adaptadores" (Ports & Adapters). La última versión está documentada en [1].

El objetivo que pretende conseguir esta arquitectura es aislar el núcleo de la aplicación (la lógica de negocio) de las tecnologías externas (bases de datos, interfaces de usuario, servicios externos, etc.), para ello hay que diseñar la aplicación para que cumpla con las siguientes características:

- Pueda ejecutarse sin interfaz de usuario.
- Pueda ejecutarse sin base de datos.
- Pueda ser controlada por tests automáticos o scripts.
- Sea posible permitir cambiar tecnologías sin tocar el dominio.

La Figura 1 muestra el objetivo de la arquitectura hexagonal, que es aislar el núcleo de la aplicación de las tecnologías externas. La aplicación establece un límite claro dentro del cual (el hexágono) se encuentra la lógica de negocio, mientras que las tecnologías externas se encuentran fuera del hexágono. La aplicación se comunica con el exterior a través de interfaces bien definidas (puertos) y adaptadores que traducen las llamadas entre la aplicación y el exterior. De esta manera, se puede desarrollar y probar la lógica de negocio de manera independiente de las tecnologías externas, lo que facilita el desarrollo incremental e independiente, la creación de pruebas y la evolución del sistema a lo largo del tiempo.

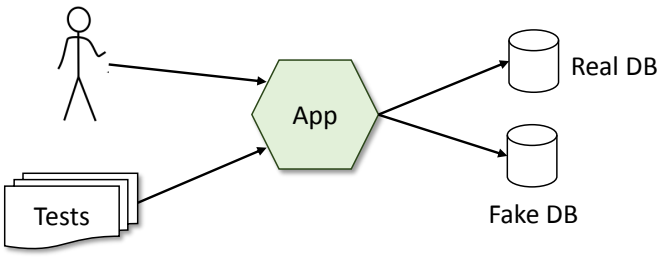

Figura 1: Objetivo de la arquitectura hexagonal: aislar el núcleo de la aplicación de las tecnologías externas.

2.1. Elementos

La arquitectura tiene cuatro elementos básicos:

- La aplicación o sistema en sí (lo que está dentro del hexágono).
- Los puertos que definen la interfaz de comunicación entre la aplicación y el exterior.
- Los actores externos que dirigen la aplicación o son dirigidos por la aplicación.
- Los adaptadores que se encargan de traducir las llamadas entre la aplicación y los actores externos a través de los puertos.

Además, desde un punto de vista práctico debe haber un elemento adicional que configure la conexión entre los cuatro elementos anteriores, pero eso

está fuera de la arquitectura propiamente dicha puesto que es un elemento de infraestructura.

En esta arquitectura un **aplicación** se ve como un componente reutilizable que no tiene dependencias tecnológicas externas. Esto quiere decir que esencialmente tiene lógica de negocio y lógica de coordinación (servicios de aplicación), pero no tiene lógica de infraestructura (bases de datos, interfaces de usuario, servicios externos, etc.)[1]..

[1]La aplicación puede ser fácilmente distribuida como una librería.

Esencialmente cualquier elemento físico con el que interacciona la aplicación está fuera de la aplicación. Esto incluye bases de datos, interfaces de usuario, servicios externos, redes, etc.

Para comunicarse con el exterior la aplicación define **puertos** (ports) que son interfaces que definen cómo la aplicación interactúa con el exterior. Los puertos son de dos tipos: puertos de entrada (***driving ports***) y puertos de salida (***driven ports***). Los puertos de entrada son los que utilizan los actores externos para interactuar con la aplicación, mientras que los puertos de salida son los que utiliza la aplicación para interactuar con los actores externos.

Esencialmente un puerto declara una o más interfaces que definen cómo la aplicación interactúa con el exterior. Hay dos tipos de interfaces: provided interfaces y required interfaces.

- La provided interface es la interfaz que la aplicación ofrece al exterior para que el pueda "dirigir" la aplicación.
- La required interface es la interfaz que la aplicación requiere del exterior para que la aplicación pueda "conducir" ciertos elementos.

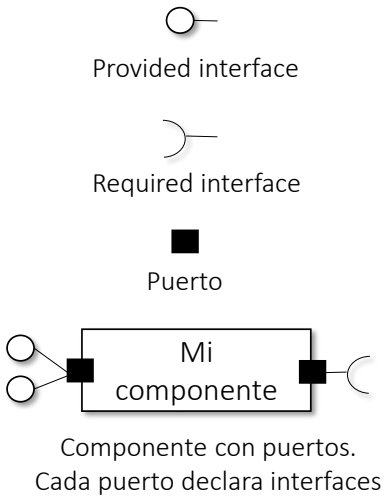

Figura 2: Símbolos UML para describir componentes.

> ⚠️ **¿Quién define las interfaces?**
>
> La aplicación define ambos tipos de interfaces puesto las interfaces representan el contrato que establece con el exterior. La diferencia es que las "provided interfaces" son el contrato que la aplicación se compromete a cumplir, mientras que las "required interfaces" son el contrato que la aplicación necesita o exige que el exterior cumpla.

Para interactuar con la aplicación se crean **adaptadores**. Un adaptador es un componente que implementa una interfaz de puerto (provided o required) y se encarga de transformar las llamadas entre la aplicación y el actor externo.

Algunos ejemplos de adaptadores para los puertos conducidos son:

- Un adaptador de base de datos que implementa la required interface del puerto de salida para permitir a la aplicación almacenar y recuperar datos de una base de datos ej., MySQL.
- Un adaptador de servicio web que implementa la required interface del puerto de salida para permitir a la aplicación comunicarse con una API externa (ej., una API de un LLM).

Estos adaptadores se implementan fuera de la aplicación en sí pero se despliegan junto con la aplicación.

Algunos ejemplos de adaptadores para los puertos dirigidos son:

- Una interfaz de usuario (UI) que invoca la provided interface del puerto de entrada para permitir a un usuario interactuar con la aplicación.
- Un conjunto de pruebas automatizadas que invoca la provided interface del puerto de entrada para probarlas

Es importante observar que la implementación concreta de la "provided interface" **está dentro** de la aplicación. El adaptador no la conoce porque se comunica a través de la interfaz. Su labor es traducir las llamadas del actor externo a la interfaz de la aplicación.

2.2. Ejemplo de arquitectura hexagonal y DDD

Supongamos que se quiere implementar una aplicación de pedidos para una tienda online. La funcionalidad principal es la de crear un pedido y cuando esté listo procesar el pago y guardar el pedido en una base de datos. Para ello necesita:

- Implementar el modelo de dominio que representa los pedidos y la lógica de negocio asociada.
- Definir un puerto de entrada para la operación de crear pedido, que será invocado por una interfaz de usuario o por pruebas automatizadas.
- Definir puertos de salida para interactuar con la base de datos para guardar los pedidos y para interactuar con un servicio de pago externo para procesar los pagos.
- Implementar adaptadores para la base de datos (ej., Postgresql) y para el servicio de pago (ej., Stripe).
- Implementar una interfaz de usuario (ej., una API REST) que invoque el puerto de entrada para crear pedidos.

2.2.1. Diseño e implementación de la aplicación con DDD

En la arquitectura hexagonal para el modelo de dominio se puede seguir el enfoque que se desee. Sin embargo, en nuestro caso utilizaremos Domain-Driven Design (DDD) de manera que el dominio se organizará según los patrones tácticos de DDD.

Definiremos entidades como `Pedido`, `LineaPedido`, `Cliente`, etc., junto con la lógica de negocio asociada. Además habrá que definir servicios de aplicación para representar las operaciones que pueden realizar los usuarios de la aplicación.

```
public class Pedido {

  private final PedidoId id;
  private final ClienteId clienteId;
  private final List<LineaPedido> items;
  private final LocalDateTime fechaCreacion;
  private EstadoPedido estado;

  // Resto del código
  public void addLineaPedido(ProductoId productoId, int qty) {
    // Lógica para añadir una línea de pedido
  }

  public void marcarComoPagado() {
    // Lógica para marcar el pedido como pagado
  }

}
```

Para poder realizar pagos necesitamos interactuar con un **servicio de pago externo**. Este es un servicio de infraestructura, para el cual definiremos una interfaz dentro de la aplicación pero que será implementada por un adaptador externo. Este servicio se usará en el servicio de aplicación correspondiente (ver debajo).

```
/** Definición de un servicio de infrastructura */
public interface ServicioDePago {

    EstadoPago procesarPago(PedidoId pedidoId, Dinero total);
}
```

El **servicio de aplicación** se encarga de coordinar la lógica de negocio para gestionar los pedidos. Tiene dos dependencias (ver constructor): el repositorio de pedidos para guardar y recuperar los pedidos de la base de datos, y el servicio de pago para procesar los pagos.

```
public class ServicioDePedidos {

  // Dependencias del servicio. Durante la configuración
  // de la aplicación se inyectarán las implementaciones
  // concretas de estas dependencias.
  private final RepositorioDePedidos repositorio;
  private final ServicioDePago servicioDePago;

  public ServicioDePedidos(RepositorioDePedidos repositorio,
                           ServicioDePago servicioDePago) {
      this.repositorio = repositorio;
      this.servicioDePago = servicioDePago;
  }

  // ...

  @Override
  public Pedido pagarPedido(String pedidoId)
    throws PagoRechazadoException {

    Pedido pedido = repositorio.findPorId(pedidoId);
    Preconditions.checkNotNull(pedido, "Pedido no encontrado");

    EstadoPago estadoPago = servicioDePago.procesarPago(pedidoId,
pedido.getTotal());

    if (estadoPago == EstadoPago.APROBADO) {
        pedido.marcarComoPagado();
        repositorio.guardar(pedido);
    } else {
        throw new PagoRechazadoException("El pago fue rechazado
para el pedido: " + pedidoId);
    }

    return pedido;
  }
}
```

2.2.2. Puertos y adaptadores

La aplicación definida en las clases anteriores es independiente de las tecnologías externas, pero para que pueda funcionar es necesario definir

los puertos y adaptadores correspondientes. Como se muestra en la Fig. Figura 3, la aplicación tiene:

Un puerto de entrada (***driving port***):

- `ServicioDePedidos` que define la interfaz para gestionar los pedidos.

Dos puertos de salida (***driven ports***):

- `RepositorioDePedidos` que define la interfaz para almacenar de manera persistente los pedidos.
- `ServicioDePago` que define la interfaz para interactuar con el servicio de pago externo

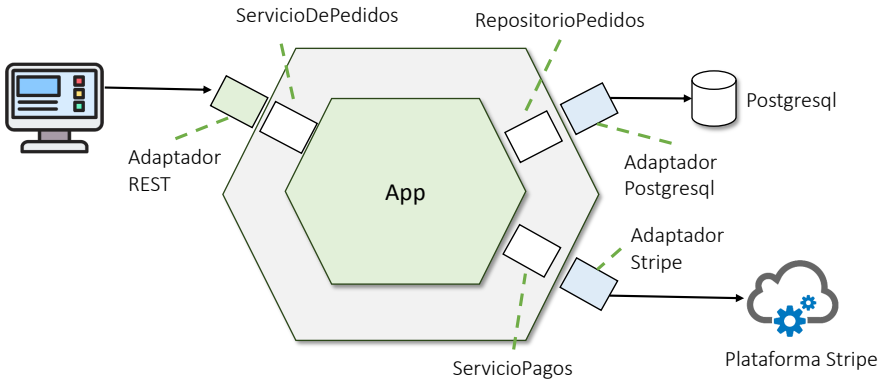

Figura 3: Ejemplo de arquitectura hexagonal cuyo nucleo es la aplicación creada con DDD para pedidos.

2.2.3. Correspondencia entre DDD y la arquitectura hexagonal

Los elementos de DDD tienen una correspondencia directa con los elementos de la arquitectura hexagonal:

- El modelo de dominio (entidades, objetos valor, servicios de dominio, etc.) se encuentra dentro del núcleo de la aplicación.
- Los servicios de aplicación (como `ServicioDePedidos`) también se encuentran dentro de la aplicación ("dentro del hexágono"), pero se intepretan como un puerto de entrada para que los actores externos puedan interactuar con la aplicación.
- Los repositorios corresponden a puertos de salida o ***driven ports*** que la aplicación utiliza para interactuar con la base de datos.
- Los servicios de infraestructura (como `ServicioDePago`) que necesite la aplicación también corresponden a puertos de salida o ***driven ports***.

2.3. Ventajas y desventajas

Los beneficios de esta arquitectura son los siguientes:

1. **Pruebas**. Se pueden escribir y ejecutar pruebas a nivel de sistema sin necesidad de configurar o conectar a las tecnologías de producción, lo cual facilita la creación de pruebas.
2. **No hay contaminación de detalles tecnológicos**. La propia arquitectura asegura que las dependencias externas no afecten a la lógica de negocio. Esto es así porque las pruebas van a demostrar que los detalles de la UI o de tecnología (ej., base de datos) no contaminan la lógica de negocio. ¿Por qué? Porque si así fuera no podrían ejecutarse las pruebas.

3. **Desarollo incremental e independiente**. Diferentes equipos pueden desarrollar sus partes del sistema de manera independiente, probarlas por separado y conectarlas mediante interfaces bien definidas y probadas. Además, el sistema se puede empezar a construir sin necesidad de tener todas las conexiones externas listas.
4. **Mantenibilidad**: Se pueden reemplazar las conexiones externas a medida que cambian las necesidades tecnológicas y de negocio a lo largo de los años.
5. **Diseño centrado en el dominio**: Al mantener los elementos tecnológicos fuera del núcleo de la aplicación, se puede centrar en el diseño del dominio, por ejemplo utilizando Domain-Driven Design (DDD).
6. **No hay necesidad de recompilar**. El núcleo de la aplicación no depende de las tecnologías externas, por lo que no es necesario recompilar el sistema para cambiar entre entornos de prueba y producción o para actualizar las tecnologías externas. En otras palabras el sistema es "externamente extensible" como un sistema de plugins.

Sin embargo, existen algunos costes asociados a esta arquitectura:

1. Es necesario diseñar y definir las interfaces entre el núcleo de la aplicación y las tecnologías externas, lo cual requiere tiempo y esfuerzo adicional.
2. Hay una sobrecarga inicial en la configuración del proyecto y la estructura del código para implementar la arquitectura hexagonal.
3. Puede haber una ligera disminución del rendimiento debido a la capa adicional de abstracción entre el núcleo de la aplicación y las tecnologías externas.

Bibliografía

[1] A. Cockburn y J. M. G. de Paz, «Hexagonal architecture explained», *How the ports & adapters architecture simplifies your life, and how to implement it. Humans and Technology Inc*, 2024.