2. Introducción al diseño de software

El diseño software es una disciplina de la Ingeniería de Software.

- **Diseño explícito:** realizado para planificar o documentar el software desarrollado.
- **Diseño implícito:** la estructura que el software tiene realmente, aunque no se haya diseñado formalmente.

Diseñar bien no es garantía de éxito, es decir, realizar un diseño previo no garantiza que el software resultante sea conforme a ese diseño ni a la especificación.

¿Cómo y cuánto hay que diseñar?

- **Diseño orientado a objetos:** los sistemas de software se componen de objetos con un estado privado y que interactúan entre sí.
- **Diseño orientado a funciones:** descompone el sistema en un conjunto de funciones que interactúan y comparten un estado centralizado.
- **Diseño de sistemas de tiempo real:** reparto de responsabilidades entre software y hardware para conseguir una respuesta rápida.
- **Diseño de interfaces de usuario:** tiene en cuenta las necesidades, experiencia y capacidades de los usuarios.

Fases del diseño

- **Diseño arquitectural:** estructura de alto nivel. Identificación de los componentes principales junto con los requisitos funcionales y no funcionales.
- **Diseño detallado:** los componentes se descomponen con un nivel de detalle más fino. Guiado por la arquitectura y los requisitos funcionales.

Ventajas de un buen diseño

- Aumenta las probabilidades de finalizar con éxito el proyecto.
- Ayuda a trasladar las especificaciones a los programadores de forma no ambigua.
- Ayuda a escribir código de calidad.

Herramientas de diseño orientado a objetos

UML, los diagramas más usados, no son más que un formato de representación y su uso no garantiza que el diseño sea bueno. Engloba los diagramas de: actividad, caso de uso, secuencia, clases y estados.

Se usa UML como:

- Un medio para facilitar el debate sobre un sistema existente o propuesto.
- Una forma de documentar un sistema existente.
- Una descripción detallada que se puede usar para desarrollar una implementación.
- Los modelos se pueden usar para generar código automáticamente.

La metodología impone las reglas

- En **metodologías tradicionales** hay que generar numerosos modelos antes de tocar una sola línea de código.
- En **metodologías ágiles** hay que modelar lo mínimo imprescindible, en grupo y de manera informal (se usa como herramienta de comunicación).

3. Patrones arquitecturales - Arquitectura en capas

Un **diseño arquitectural** es una abstracción que muestra únicamente los detalles relevantes. Se deben tener en cuenta los requisitos no funcionales (p. ej. rendimiento esperado) y la distribución lógica y física de sus componentes.

Las aplicaciones que **no** tienen una **arquitectura formal** normalmente son las: altamente acopladas, frágiles, difíciles de cambiar y/o de determinar sus características arquitecturales de una aplicación sin comprender el funcionamiento interno de cada componente y módulo en el sistema, y/o que no tienen una visión o dirección claras.

Los **patrones arquitecturales** son formas de capturar estructuras de diseño de eficacia probada, de manera que puedan ser reutilizadas.

Patrón de diseño

Son soluciones documentadas a problemas frecuentes en el diseño de software.

Consecuencia: El acceso controlado a la única instancia es mejor que el uso de variables globales.

Los **patrones arquitecturales** expresan esquemas de organización estructural fundamentales, proporcionan un conjunto de subsistemas predefinidos, especifican sus responsabilidades e incluyen reglas y principios para organizar sus relaciones.

Para los patrones más habituales es frecuente encontrar **frameworks**.

Frameworks

Es una aplicación reutilizable, “semi-completa”, que se puede especializar para producir aplicaciones personalizadas. Incorporan numerosos patrones de diseño en su implementación. Muchos frameworks siguen el **principio Convention over configuration**, es decir, que no es necesario crear archivos de configuración si se respetan las convenciones de nombres.

Propiedades: modularidad, reusabilidad, extensibilidad e inversión de control (Principio Hollywood).

Arquitectura en capas

Cada capa se encarga de una tarea específica, abstrayendo los detalles a las demás capas (separación de intereses).

Capas típicas (aunque el patrón no prescribe ninguna):

- **Presentación.** Interacción con el usuario.
- **Servicios.** Funcionalidades de alto nivel.
- **Lógica de negocio.** Ejecución de las reglas de negocio.
- **Persistencia / Acceso a datos.** Comunicación con la BD.
- **Base de datos.** Almacenamiento de información.

Hay que tener en cuenta que:

- La capa de lógica de negocio y la capa de persistencia no deben depender nunca de la capa de presentación.
- La capa de lógica de negocio y la capa de persistencia pueden juntarse en una única capa en algunas circunstancias.
- La capa de servicios ofrece funcionalidades compuestas a partir de las clases de la capa de lógica de negocio, ocultando su complejidad.
- Se pueden crear tantas capas como sea necesario, siempre que cada capa tenga un propósito claro y separado de las demás.

La arquitectura puede ser:

- **Cerrada.** La comunicación va de una capa a la inmediatamente inferior. Este tipo disminuye el impacto de los cambios y la complejidad del sistema.
- **Abierta.** Las capas superiores se pueden comunicar con todas o algunas de las inferiores.

4. Lógica de negocio y acceso a datos

Patrones de lógica de negocio:

Patrón	Definición	Características
Transaction script	"Organiza la lógica de negocio en procedimientos, donde cada procedimiento gestiona una petición de la capa de presentación".	● Forma más sencilla. ● Se crea una transacción para cada una de las funcionalidades. ● Cada *transaction script* para un solo método.
Table/Row Data Gateway	"Un objeto que actúa como pasarela para una tabla de la BD. Una única instancia gestiona todas las filas de la tabla". Se haría uso de esta si queremos evitar llamar a la BD directamente.	● Se puede juntar con Transaction Script. ● Implementa el acceso a datos.
Table module	"Una única instancia gestiona la lógica de negocio para todas las filas en una tabla o vista de la BD".	● Implementa la lógica de negocio. ● Solo lo correspondiente a cada objeto individual. ● **Las operaciones complejas deben ir en la capa de servicio.** ● Se puede usar Table Date Gateway para comunicarse con la BD.
Domain model	"Un objeto del modelo de dominio incorpora tanto el comportamiento como los datos." Dependiendo de la complejidad el acceso a datos se puede hacer de dos maneras.	**ActiveRecord (M. sencillo)** "Un objeto que envuelve una fila de una tabla a vista, encapsula el acceso a la BD y añade comportamiento a esos datos." (+) Más sencillo de implementar. (-) Incrementa el acoplamiento con la BD y dificulta la refactorización. (-) Difícil optimizar el SQL y puede perjudicar el rendimiento. **Data Mapper (M. Complejo)** "Una capa de mapeadores se encarga de mover datos entre los objetos y la base de datos, manteniéndolos independientes entre ellos e independientes del mapeador." (+) Permite realizar transformaciones complejas. (+) Facilita la optimización del acceso a la BD. (-) Más difícil de entender y configurar.

5. Mapeo objeto-relacional

Cuando se decide implementar un modelo de dominio hay que encontrar la manera de adaptar la estructura del modelo a la estructura de la base de datos.

Tipos de estructura de la **base de datos**:

- **Orientadas a objetos.** Son una forma natural de persistir objetos, pero no son muy comunes.
- **Relacionales.** Son mucho más comunes, pero tienen una estructura distinta a los modelos orientados a objetos, y usa SQL para realizar las consultas.

Los **sistemas de mapeo objeto-relacional** (OMR) se encargan de almacenar y recuperar los objetos en bases de datos relacionales. Los sistemas ORM existentes normalmente implementan uno de los dos patrones para persistir modelos de dominio.

Dos estrategias:

- **Model first.** Primero se diseña el modelo de dominio, y luego se crea la base de datos con la estructura necesaria. Situación ideal para usar el patrón ActiveRecord.
- **Database first.** Se parte de una base de datos ya existente, por lo que hay que adaptar el modelo de dominio a su estructura. En estos casos puede ser más conveniente usar un Data Mapper.

Patrones de comportamiento

Los patrones de comportamiento gestionan cómo se almacenan y recuperan los objetos de una base de datos relacional.

Patrón	Problemas	Definición	Características
Unit of Work	Se debe mantener la integridad referencial cuando se crean y modifican múltiples objetos.	“Mantiene una lista de objeto afectados por una transacción y coordina la escritura de los cambios y la resolución de problemas de concurrencia”.	Llamar a la base de datos para cada cambio en el modelo de dominio puede afectar el rendimiento, por eso **se abre una transacción** con la BD **cuando es necesario**.
Identity Map	No debe haber múltiples copias del mismo objeto en memoria.	“Asegura que cada objeto se carga una única vez llevando un registro en un mapa de cada objeto. Busca los objetos en el mapa antes de recuperarlos de la BD.”	El objeto Identity Map lleva un registro de los objetos cargados y devuelve referencias para los que ya existe una instancia en memoria. También actúa de **caché** para la BD.

Lazy Load	Cargar los objetos relacionados en un modelo de dominio puede acabar recuperando la BD completa.	"Un objeto que no contiene datos, pero sabe cómo obtenerlos."	Lazy Load solo carga un objeto cuando se usa. ● **Lazy initialization:** objeto nulo y se necesita encapsular en método get. ● **Virtual proxy:** objeto vacío y puede ir cargando los datos.

Patrones de comportamiento

A veces no es necesario mapear todos los objetos de dominio a tablas en la BD.

Patrón	Definición	Características
Embedded Value	"Mapea un objeto en varios campos de otra tabla."	*Null*
Serialized LOB	"Objetos que almacenan una estructura jerárquica de pequeños objetos en un único objeto grande (LOB) de una forma eficiente."	● Forma binaria (BLOB) ● Forma textual (CLOB) ● Almacenarlo en una única columna. ● Imágenes en una carpeta y guardar su ruta en la BD.
Mapeo de relaciones	"Cuando un objeto contiene colecciones o referencias (compartidas) a otros objetos, no deben guardarse como valores en la misma tabla."	● De clave ajena. ● De tabla de asociación. ● De herencia: Las BD relacionales no la soportan, entonces usamos clases hijas como padres (poliformismo) o que todas las clases compartan un id común.
Class table inheritance	Cada tabla almacena únicamente los atributos nuevos.	**(-)** Necesita hacer join para cada consulta. **(+)** No hay columnas irrelevantes.
Concrete table inheritance	Cada tabla almacena todos los atributos (heredados + nuevos).	**(-)** Es complicado gestionar los identificadores. **(-)** No se pueden representar relaciones con las clases abstractas. **(+)** No hay columnas irrelevantes. **(+)** No necesita hacer join.
Single table inheritance	Se usa una única tabla para todas las clases. Almacena la unión de todos los atributos de las clases hijas.	**(-)** Las columnas que no usan todas las clases gastan memoria innecesaria. **(+)** Evita joins innecesarios. **(+)** Mismo campo identificador para todas las subclases.

Autentificación de usuarios

Problemas:

- No podemos cambiar el tipo de un usuario, hay que destruirlo y crear uno nuevo.
- Tendríamos que hacer comprobación de tipos (instanceof) para comprobar el tipo de usuario cada vez que se ejecuta una funcionalidad.

Cuando el usuario pide acceso a una funcionalidad se hacen dos comprobaciones: **autentificación y autorización.**

Cuando el usuario introduce las credenciales correctas se crea una instancia de la clase User y se almacena la sesión (objeto global que almacena información compartida entre todas las pantallas).

La clase User no implementa las funcionalidades, se usa para otorgar acceso a las pantallas que las ofertan y se puede simplificar el diseño usando una única clase. Si queremos tener mayor control sobre lo que pueden hacer los usuarios podemos añadir permisos.

Generalización: patrón Role-Based Access Control (RBAC).

6. Patrones de capa de presentación

Responsabilidades:

- Gestionar la interacción con el usuario.
- Comunicarse con las capas inferiores que proveen las funcionalidades deseadas.
- Mostrar/actualizar la información como resultado de las llamadas a la lógica de negocio.

Los **patrones estructurales más usados** para la capa de presentación son:

- Model-View-Controller (MVC).
- Model-View-Presenter (MVP).
- Model-View-ViewModel (MVVM).

Estos patrones son independientes del tipo de aplicación, dependiendo del framework y de las características de la aplicación será más sencillo emplear unos u otros.

Normalmente los frameworks para aplicaciones web de servidor están preparados para usar el patrón MVC. Sin embargo, la mayoría de frameworks para aplicaciones web cliente (frontend) permite usar cualquiera de los tres.

Model-View-Controller (MVC)

Características

Es uno de los patrones con más variantes y sobre el que menos consenso hay.

Cada componente gráfico tiene:

- **Vista**. Se encarga de dibujarlo a partir de los datos del modelo.
- **Controlador**. Recibe la interacción del usuario y manipula el modelo.

Funcionamiento

Hay dos posibilidades para actualizar la **vista**:

1. El controlador actualiza la vista después de manipular el modelo.
2. El modelo (activo) notifica a la vista para que se actualice.

En **apps web simples** (sin clientes enriquecidos con JavaScript) la comunicación entre objetos es distinta:

- La vista gestiona la interacción con el usuario y notifica al controlador.
- El controlador manipula el modelo y decide qué vista mostrar a continuación.

Es necesario porque la app está dividida físicamente entre el cliente y el servidor.

La **vista** tiene dos partes:

- Un objeto (dinámico) que se instancia en la capa de presentación del servidor para generar el HTML.
- El HTML (estático) mostrado en el navegador del cliente.

Varias formas de programar los **controladores**:

- **Page controller.** Un objeto que gestiona una petición para una paginación o acción específica en un sitio web. Es el patrón utilizado por ASP.NET (code-behind). Implica la creación de un controlador para cada página lógica de la aplicación. Para aplicaciones más complejas es muy difícil de gestionar.

- **Front controller.** Un controlador que gestiona todas las peticiones de un sitio web. Realiza el comportamiento común a todas las acciones, y luego delega en controladores específicos para cada acción. La principal ventaja frente al patrón Page Controller es que ahora un solo objeto controlador gestiona todas las peticiones para evitar duplicación de comportamiento.

En frameworks web modernos, el Front Controller delega algunas de sus responsabilidades en otros objetos: objetos Middleware o un router y se puede usar un Application Controller.

Cada objeto **Middleware** se especializa en realizar un tipo de comprobación:

- Verificar si el usuario está autentificado.
- Comprobar si el usuario tiene permisos para ejecutar la acción.
- Limitar acceso a los recursos por número de accesos o ancho de banda (throttling).
- Comprobar y/o modificar los valores de entrada.

Los frameworks incorporan algunos Middleware por defecto, como comprobar usuario autentificado.

También se pueden crear objetos Middleware personalizados o encadenar distintos objetos Middleware.

El **Router** es el responsable de seleccionar el controlador apropiado para cada petición. Otros frameworks, como ASP.NET, para MVC están configurados para hacer una correspondencia entre la ruta y el método del controlador a ejecutar.

Un controlador no siempre puede decidir directamente cuál es la siguiente vista a mostrar, ya que puede depender del estado de los modelos o de la ejecución de una regla de negocio. En estos casos es conveniente mantener un **Application Controller** en una capa separada.

Model-View-Presenter (MVP)

Problemas:

- La vista está acoplada al modelo, pudiendo acabar con otras vistas.
- Cada vista tiene su propio controlador, hay poca reutilización de código.
- Es difícil probar los controladores porque están acoplados con vistas concretas.

Solución: Un objeto Presenter en lugar del controlador de MVC.

Características

La vista ya no depende del modelo, ahora el objeto *Presenter* lee el modelo y prepara los datos para pasarlos a la vista. Es más fácil probar los *Presenter* inyectándoles vistas mock.

Funcionamiento

Cuando se crea una instancia de la vista, esta crea un *Presenter* para que cargue la información. Si la app es distribuida, este realiza la petición en un hilo paralelo y la vista no bloquea la interfaz. Cuando el Presenter recibe la información, le pasa los datos a la vista y esta los muestra. Cada vez que una vista necesita nueva información, ocurre lo mismo.

Model-View-ViewModel (MVVM)

Características

Cuando una aplicación tiene **mucha interacción en el interfaz**:

- Una sola vista puede necesitar datos de muchos modelos.
- Hay que gestionar todos los eventos de los controles gráficos del interfaz y responder adecuadamente a cada acción.
- Un cambio en el modelo puede necesitar que se actualicen varias partes de una pantalla.

Funcionamiento

El objeto **ViewModel** es una representación del modelo en la capa de presentación, que además contiene la lógica necesaria para responder a los eventos del interfaz. Además es el responsable de comunicarse con la capa de lógica de negocio cuando se solicita desde la vista.

Ventajas

El uso de **Data Binding**, creando un enlace entre los controles del interfaz y los objetos Viewmodel.

Cuando el usuario actualiza el campo de un formulario, se actualiza automáticamente la propiedad asociada del ViewModel y, como consecuencia, los controles asociados en la interfaz.

Esto permite construir interfaces con mucho menos código.

Aplicaciones web RIA

Rich Internet Application (RIA) es una aplicación web que tiene un interfaz con características similares a las aplicaciones de escritorio. Las más extendidas actualmente son las aplicaciones de tipo **Single Page Application** (SPA), y usan JavaScript y AJAX en el cliente.

Funcionamiento de una aplicación SPA:

- El cliente carga un HTML mínimo y los scripts con la lógica de la aplicación cliente.
- El interfaz se construye dinámicamente, generando el HTML necesario a partir de los datos proporcionados por el servidor.
- El cliente realiza peticiones AJAX al servidor y la página no se recarga.
- Los datos se transmiten serializados en formato JSON.

Estas **aplicaciones clientes** necesitan **estructurar su código**. Para eso existen frameworks que favorecen el uso de los distintos **patrones MV*.**

7. Diseño web adaptable

- **Adaptive web design**: Técnicas para adaptar el contenido de una página web a algún dispositivo.
- **Responsive Web Design (RWD)**: Técnicas para que el contenido de una página web se adapte automáticamente al tamaño de pantalla del dispositivo.

Es necesario para mejor experiencia del usuario (UX) → disminuye la tasa de rebote.

Conceptos

Viewport. Es el área de tamaño similar que usan los navegadores móviles para poder mostrar las páginas web están diseñadas para escritorio con un tamaño fijo en píxeles. Este tamaño se puede controlar con la etiqueta <meta>.

Densidad de pantalla. Cantidad de píxeles que caben en la pantalla. Normalmente se mide en puntos por pulgada (dots per ingh, dpi).

Media queries. Introducidas en CSS3. Permiten aplicar un conjunto de reglas solo si se cumples unas determinadas condiciones e identificar el tipo de dispositivo, el tamaño y la orientación de la pantalla.

Sistemas de rejillas. Muchas páginas estructuran el contenido mediante rejillas (grid), que divide la página en columnas, normalmente 12, que se redimensionan automáticamente al cambiar el tamaño de la pantalla. Esto facilita el posicionamiento de los elementos en la pantalla.

Fluid grids. Una rejilla fluida reestructura el contenido apilando las columnas cuando el tamaño de la pantalla es demasiado pequeño.

HTML semántico. El uso de etiquetas semánticas en HTML5 ayuda a posicionar de forma más inteligente el contenido de la página web.

Imágenes responsive. Uso de técnica básica (max-widht), distintas imágenes para distintas densidades de pantalla y/o adaptar el nivel de detalle al tipo de pantalla.

Rendimiento de la página. Servir el mismo contenido independientemente del dispositivo puede afectar negativamente al tiempo de carga en conexiones móviles.

Estrategias de diseño

Planificación

- No diseñar para un ancho de pantalla específico.
- Identificar qué contenidos son importantes y cuales son prescindibles, con el fin de ocultar estos últimos en las versiones móviles.
- Realizar mockups para las distintas versiones.
- Probar el resultado en dispositivos reales.
- Evitar plugins siempre que sea posible.

Diseñar **mockups** primero para móviles:

- Mejora el rendimiento en dispositivos pequeños.
- En lugar de cambiar los estilos de pantalla cuando esta es más pequeña, lo hace cuando es más grande.
- Intenta cargar el contenido que se ve a simple vista en un segundo, por lo que hay que situar primero el HTML y CSS imprescindible de no más de 14KB para conexiones 3G y, a continuación, cargar el JavaScript e imágenes secundarias.
- Usar minificadores de código y optimizadores de imágenes.
- Usar carga condicional de contenido e imágenes con JavaScript, devolviendo imágenes de distinto tamaño desde el servidor.

Herramientas

Frameworks. Paquete de ficheros CSS y JavaScript con clases CSS predefinidas para formatear el contenido. Se adaptan automáticamente al tamaño de pantalla. Los más utilizados son Bootstrap y Foundation.

Plantillas. Páginas prediseñadas para modificar únicamente el contenido.

Herramientas de desarrollo. Incluidas en los navegadores como Firefox Developer Tools, Firefox Developer Edition o Chrome DevTools. Permiten inspeccionar el contenido de una página, las reglas CSS que se aplican a cada elemento y ver la página en distintos tamaños de pantalla.

Optimización para móviles. Prueba de optimización para móviles de Google e Yslow de Yahoo.

8. Otros patrones arquitecturales

Arquitectura tuberías y filtros

Los datos pasan por una serie de filtros, que transforman la información. Muy usado en sistema Unix, aunque no tiene por qué implementarse necesariamente mediante tuberías y línea de comandos. Ejemplo: Apertium.

Filosofía Unix:

- Modularidad.
- Simplicidad.
- Composición mediante el encadenamiento de programas sencillos para realizar tareas complejas.
- “Haz una cosa y hazla bien”.

Beneficios

- Facilita el diagnóstico.
- Permite añadir fácilmente nuevos filtros entre módulos.
- Desarrollo de aplicaciones derivadas mediante la reutilización de módulos.

Análisis del patrón

- **Agilidad alta.** Debido a la modularidad.
- **Despliegue lento.** Normalmente se trata de aplicaciones que se ejecutan en las máquinas de los usuarios finales.
- **Pruebas fáciles.** Se pueden probar los filtros de forma independiente.
- **Rendimiento alto.** Aunque depende de la complejidad de los filtros.
- **Escalabilidad baja.** Distribuir los filtros en distintos nodos añade complejidad.
- **Desarrollo fácil.** Usando los mecanismos de comunicación del sistema operativo (tuberías).

Arquitectura microkernel (de plugins)

Arquitectura muy adecuada para el diseño y el desarrollo evolutivo e incremental, y aplicaciones “producto”. Ejemplo: Eclipse IDE.

Consiste en un núcleo central que se puede extender mediante módulos. Este contiene la funcionalidad mínima para que el sistema funcione. Además contiene plugins que son unos componentes independientes que añaden funcionalidades al núcleo central, puede haber dependencia entre ellos y se suelen organizar en un registro o repositorio para que el núcleo pueda obtenerlos.

Análisis del patrón

- **Agilidad alta.** Es sencillo añadir nuevos componentes.
- **Despliegue sencillo.** Se pueden añadir nuevos plugins en tiempo de ejecución.
- **Pruebas fáciles.** Se pueden probar los plugins de forma independiente.
- **Rendimiento alto.** Se pueden instalar únicamente los plugins necesarios.
- **Escalabilidad baja.** Normalmente diseñado como un único ejecutable.
- **Desarrollo difícil.** El diseño del interfaz de los plugins debe plantearse cuidadosamente. La gestión de versiones y repositorio de plugins añade complejidad.

Arquitectura microservicios

Arquitectura distribuida. El cliente se comunica con los servicios mediante algún protocolo de acceso remoto (REST, SOAP, RMI…). Usada para el desarrollo de aplicaciones y servicios web. La implementación más frecuente utiliza HTTP como protocolo de comunicación, definiendo interfaces REST para acceder a los servicios. Ejemplo: Netflix OSS.

Análisis del patrón

- **Agilidad alta.** Los cambios afectan a componentes aislados.
- **Despliegue sencillo.** Favorece la integración continua.
- **Pruebas fáciles.** Debido a la independencia de los servicios.
- **Rendimiento bajo.** Debido a la naturaleza distribuida.
- **Escalabilidad alta.** Permite escalar los servicios por separado.
- **Desarrollo fácil.** La independencia de los servicios reduce la necesidad de coordinación y el uso de protocolos de comunicación estándar facilita el desarrollo.

Arquitectura orientada a eventos

Arquitectura compleja de naturaleza asíncrona y distribuida. El sistema se compone de pequeños componentes que responden a eventos y de algún mecanismo para gestionar las colas de eventos que se reciben.

Dos alternativas:

- **Topología mediador.** El procesamiento de eventos implica varios pasos que deben ejecutarse de manera orquestada. Componentes:
  - **Procesador de eventos.** Contiene la lógica de negocio, y son pequeñas unidades autocontenidas y altamente independientes del resto.
  - **Mediador.** Recoge los eventos de inicio y los envía a los procesadores en el orden apropiado según el tipo de evento. Se puede implementar mediante soluciones open source y definir usando lenguajes de definición de procesos.

- **Topología broker.** Los procesadores de eventos se encadenan unos con otros mediante eventos que pasan a través del broker. Cuando un procesador de eventos termina su trabajo, genera un evento para que se ejecuten los siguientes componentes. Componentes:
  - **Procesador de eventos.** El mismo que el de la topología mediador.
  - **Broker.** Más ligero que el mediador y se encarga únicamente de gestionar las colas de eventos para que los procesadores no tengan que preocuparse de los detalles de implementación.

Análisis del patrón

- **Agilidad alta.** Los cambios afectan a uno o pocos componentes.
- **Despliegue sencillo.** Debido al bajo acoplamiento. Más complicado en el caso del mediador, ya que se debe actualizar cada vez que hay un cambio en los procesadores de eventos.
- **Pruebas complicadas.** Debido a la naturaleza asíncrona y la necesidad de herramientas especializadas para generar eventos.
- **Rendimiento alto.** Debido a la posibilidad de paralelizar la ejecución de componentes.
- **Escalabilidad alta.** Permite escalar los componentes por separado.
- **Desarrollo difícil.** La gestión de errores es compleja.

9. Diseño detallado y patrones GRASP

Perspectiva de análisis *vs* Perspectiva de diseño

UML incluye los diagramas de clases para ilustrar clases, interfaces y sus asociaciones. Estos se utilizan para el modelado estático de objetos y se pueden usar desde dos perspectivas:

- Conceptual → Modelo de Dominio.
- De implementación → Modelado de la Capa de Dominio.

Además define una responsabilidad como “un contrato u obligación de una clase” y estas se asignan a las clases durante el diseño de objetos (hacer y saber o conocer).

Los diagramas de diseño de clases se derivan del modelo de dominio, añadiendo los método y secuencias de mensajes necesarios para satisfacer los requisitos. Se decide qué operaciones se asignan a qué clases y cómo los objetos deberían interactuar para dar respuesta a los casos de uso.

El **Modelo de Diseño** es el artefacto más importante del flujo de trabajo de diseño e incluye el diagrama de clases software (no conceptuales) y los diagramas de interacción.

Patrones GRASP

GRASP = General Responsibility Assignment Software Patterns. Estos patrones describen 9 principios fundamentales del diseño de objetos y de la asignación de responsabilidades, expresados en términos de patrones y se dividen en dos grupos.

Básicos:

- Creador.
- Experto en Información.
- Bajo Acoplamiento.
- Controlador.
- Alta Cohesión.

Avanzados:

- Poliformismo.
- Fabricación Pura.
- Indirección.
- Protección de Variaciones.

Creador

- **Crear una nueva instancia de alguna clase:** Se asigna a la clase B la responsabilidad de crear una instancia de la clase A si B agrega, contiene y/o graba objetos de tipo A, y/o contiene datos inicializadores que serán pasados a A cuando sea necesario crear un objeto de dicho tipo, siendo así B un Experto con respecto a la creación de A.

- **Beneficio:** Promueve el bajo acoplamiento, lo que implica menos dependencias (mejor mantenimiento) y mayores oportunidades de reutilización.
- **Contraindicaciones:** A veces la creación de un objeto requiere cierta complejidad, como por ejemplo el uso de instancias recicladas para aumentar el rendimiento o la creación condicional de una instancia perteneciente a una familia de clases. En estos casos es preferible delegar la creación a una clase auxiliar, mediante el uso de los patrones *Concrete Factory* o *Abstract Factory*.

Experto en información

- **Principio general de asignación de responsabilidades a objetos:** Asignar cada responsabilidad al experto de información, es decir, la clase que tiene la mayor parte de la información necesaria para cubrir la responsabilidad. A veces hay que aplicar el *Information Expert* en cascada.
- **Beneficios:** Se representa la encapsulación de información, ya que los objetos usan su propia información para completar las tareas. Esto implica normalmente bajo acoplamiento. Además, el comportamiento se distribuye entre las clases que contienen la información necesaria, consiguiendo clases más ligeras.
- **Contraindicaciones:** Aumentar las responsabilidades de una clase, añade lógica de acceso a datos y disminuye así su cohesión.

Bajo acoplamiento

- **Propiedad:** Soporta una baja dependencia, un bajo impacto de cambios en el sistema y un mayor reuso asignando una responsabilidad de manera que el acoplamiento permanezca bajo.
- **Definición acoplamiento:** Medida que indica cómo de fuertemente un elemento está conectado a, tiene conocimiento o depende de otros elementos.
- **Tipos de acoplamiento:** El acoplamiento dentro de los diagramas de clases puede ocurrir por definición de atributos, interfaces de métodos, subclases y/o tipos.
- **Problemas de un alto acoplamiento:** Cambios en clases relacionadas fuerzan a cambios en las clases afectadas. La clase afectada por el alto acoplamiento es más difícil de entender por sí sola y necesita entender otras clases y es más difícil de reutilizar, porque requiere la presencia adicional de las clases de las que depende.
- **Beneficio:** Las clases con bajo acoplamiento normalmente son sencillas de comprender de forma aislada, son fáciles de reutilizar y no les afectan los cambios en otros componentes.
- **Contraindicaciones:** El alto acoplamiento con elementos estables es raramente un problema, ya que raramente habrá cambios que puedan afectar a estas clases, por lo que no merece la pena el esfuerzo de evitar este acoplamiento.

Controlador

- **Responsable de recibir o manejar un evento del sistema:** Una clase que represente el sistema completo (control 'fachada') o un escenario de un caso de uso.
- **Controlador:** Este NO es el mismo que el del patrón MVC. Normalmente ventanas, applets, etc. reciben eventos mediante sus propios controladores de interfaz y los delegan al tipo de controlador del que hablamos aquí.

Alta cohesión

- **Definición cohesión:** Medida de cómo de fuertemente se relacionan y focalizan las responsabilidades de un elemento, que pueden ser clases, subsistemas, etc.
- **Baja cohesión:**
    - Una clase con baja cohesión hace muchas actividades poco relacionadas o realiza demasiado trabajo.
    - Un diseño con baja cohesión es difícil de entender, de usar y de mantener. Además de ser delicado y fácilmente afectables por el cambio.
- **Mantener la complejidad manejable:** Asignando las responsabilidades de manera que la cohesión permanezca alta. Clases con baja cohesión a menudo representan abstracciones demasiado elevadas o han asumido responsabilidades que deberían haber sido asignadas a otros objetos.
- **Beneficios:** Aumenta la claridad y comprensibilidad del diseño, y se simplifican tanto el mantenimiento como las mejoras.
- **Contraindicaciones:** En ocasiones se puede aceptar una menor cohesión. Agrupar responsabilidades o código en una única clase o componente puede simplificar el mantenimiento por una persona (p.ej. experto en BD). También, debido al sobrecoste y las implicaciones en el rendimiento, a veces es deseable crear objetos menos cohesivos que provean un interfaz para numerosas operaciones, evitando objetos distribuidos entre servidores.
- **Conclusión:** Una mala cohesión normalmente implica un mal acoplamiento y viceversa. Una clase con demasiadas responsabilidades (baja cohesión) normalmente se relaciona con demasiadas clases (alto acoplamiento).

Poliformismo

- **Manejo de alternativas basadas en un tipo sin usar sentencias condicionales:** Asignar la responsabilidad del comportamiento usando operaciones polimórficas a los tipos para los cuales el comportamiento varía y (corolaria) no preguntes por el tipo de objeto usando lógica condicional.
- **Beneficio:** Fácil de extender el sistema con nuevas variaciones y se pueden introducir nuevas implementaciones sin afectar a las clases cliente.
- **Contraindicaciones:** Dedicar demasiado tiempo a la realización de diseños preparados para cambios poco probables, mediante el uso de herencia y poliformismo. Esto no es nada raro, por lo que suele pasar.

Fabricación pura

- **Responsabilidad sin violar principios de "Alta Cohesión" y "Bajo Acoplamiento":** Recae sobre una clase artificial conveniente a la que se le asigna un conjunto "altamente cohesivo" de responsabilidades, que no represente un concepto del dominio del problema, algo producto de la "imaginación" para soportar dichos principios, más el de reusabilidad.
- **Los objetos pueden dividirse en:** Los diseñados por/mediante descomposición representacional y los diseñados por/mediante descomposición conductual. Estos últimos son el caso más común para objetos de Fabricación pura.

Indirección

- **Responsabilidad para evitar acoplamiento directo entre 2 o más cosas:** Recae sobre un objeto intermedio que medie entre otros componentes o servicios, de tal manera que los objetos no están directamente acoplados. Este crea una indirección entre los componentes.

Protección de variaciones

- **Problema:** ¿Cómo diseñar objetos, subsistemas y sistemas de tal manera que las variaciones o inestabilidad en estos elementos no tenga un impacto indeseable sobre otros elementos?
- **Solución:** Identificar los puntos de variación o inestabilidad y asignar responsabilidades para crear una interfaz estable a su alrededor (en su sentido más ámplio: métodos que expone una clase).

10. Principios generales de diseño

Composición sobre herencia

Supone un acoplamiento muy fuerte que puede ocasionar estos **problemas**:

- Las relaciones "es un …" pueden dejar de cumplirse al incorporar nuevos cambios.
- No se puede heredar de dos o más clases.
- Los cambios de comportamiento en tiempo de ejecución son difíciles.
- Es difícil de implementar cuando los modelos persisten en una BD relacional.
- Baja la cohesión de la clase padre.

Una herencia puede NO ser adecuada si hay numerosos métodos heredados que no se usan, demasiados métodos sobreescritos y/o demasiados niveles de herencia.

En general, **la herencia solo es imprescindible cuando necesitamos el polimorfismo.** Se puede evitar la herencia simplificando el problema, a costa de sacrificar otros aspectos del diseño.

Otra forma de "heredar" un comportamiento es usando la composición, delegando la implementación de la responsabilidad a otra clase. Si estamos diseñando el modelo de dominio, la nueva clase no tiene por qué persistir en la BD. Esto ofrece mayor flexibilidad en tiempo de ejecución, ya que permite cambiar fácilmente el comportamiento de una clase.

Principios SOLID

5 principios fundamentales de diseño:

- **Principio de responsabilidad única.** Una clase solo debería tener un motivo para cambiar.
- **Principio abierto/cerrado.** Las entidades de software (clases, módulos, funciones...) deberían estar *abiertas para la extensión, pero cerradas a la modificación*.
- **Principio de sustitución de Liskov.** Si S es un subtipo de T, entonces los objetos de tipo T en un programa deberían poder ser sustituidos por objetos de tipo S. Cada clase que hereda de otra puede usarse como su padre sin necesidad de conocer las diferencias entre ellas.
- **Principio de segregación de interfaces.** Ninguna clase debería depender de métodos que no usa.
- **Principio de inversión de dependencias.** Módulos de alto nivel no deberían depender de módulos de bajo nivel. Ambos deberían depender de abstracciones y estas no deberían depender de los detalles, sino a la inversa.

Inyección de dependencias

Es una forma de inversión de control. Un objeto distinto se encarga de crear la instancia de una implementación concreta y proporcionársela al objeto que la usará.

Tipos de inyección:

- Inyección en el constructor.
- Inyección con método setter.
- Uso de un proveedor de servicios.

Inyección *vs* Service Locator

- La inyección mediante constructor o método setter permite identificar mejor las dependencias, con un Service Locator hay que buscar sus llamadas en el código.
- El Service Locator centraliza la inyección de dependencias en un único objeto, son más fáciles de gestionar.

Constructor *vs* Método setter

- La inyección mediante constructor permite crear objetos válidos desde la inicialización y proteger campos que no deben cambiar.
- El método setter permite cambiar el comportamiento del objeto una vez inicializado.
- Lo importante es mantener la consistencia.
- Se pueden proporcionar los dos mecanismos, no son excluyentes.

Existen frameworks que permiten hacer la inyección de dependencias de forma automática, como Laravel. La configuración se realiza mediante archivos XML. El *Service Container* se encarga de resolver las dependencias en los constructores. Para poder resolver las dependencias hay que registrarlas mediante *ServiceProviders*.