TEMA 093. TÉCNICAS DE DISEÑO DE SOFTWARE. DISEÑO POR CAPAS Y PATRONES DE DISEÑO.

Actualizado a 19/04/2023

1. INTRODUCCIÓN

En relación con el diseño de software existe abundante literatura que abarca desde los conceptos del diseño como la abstracción, el refinamiento, la modularidad y los patrones principales de diseño, que solucionan las principales necesidades que presenta el diseño de cualquier elemento software.

2. DISEÑO ESTRUCTURADO

El diseño estructurado de sistemas se ocupa de la **identificación, selección y organización de los módulos y sus relaciones**. Se decide que **componentes son necesarios**, y la interconexión entre los mismos, para solucionar un problema bien especificado.

Se comienza con la especificación resultante del proceso de análisis, se realiza una descomposición del sistema en módulos estructurados en jerarquías

Objetivos DE

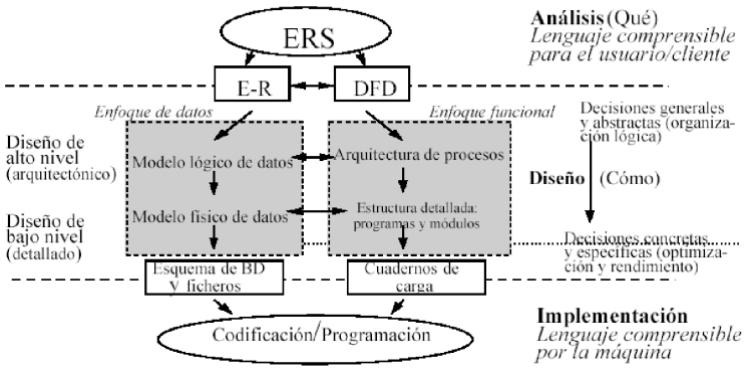
- ✓ Máxima inteligibilidad del sistema
- ✓ Minimizar el coste asociado al mantenimiento
- ✓ Facilitar la prueba
- ✓ Integración del sistema
- ✓ Que se garanticen los requisitos "no funcionales" de Reusabilidad, Fiabilidad y Portabilidad
- ✓ Utilización de herramientas gráficas para la representación de la estructura modular del sistema

Principios generales DE

- ✓ Modularización (descomposición modular)
- ✓ Jerarquía entre módulos
- ✓ Independencia modular.
- ✓ Modelización conceptual
- ✓ Principio de "caja negra"➔ abstracción. **Tipos Abstractos de Datos (TAD**) permiten generalizar y encapsular los aspectos relevantes sobre los datos que maneja un módulo o programa

Categorías de diseño:

- ✓ **Arquitectura general del sistema**: Este primer paso considerará al sistema como un todo (sw, hw y rrhh). Es de alto nivel. Es un diseño externo porque plantea las interfaces del sistema con el exterior (podremos tomar datos de la fase de análisis o incluso EVS-AGS).
- ✓ **Diseño de alto nivel o arquitectónico o estructurado**: Desarrolla una **estructura modular de programa** y representa las relaciones de control entre los módulos ➔ se utiliza el **DEC - Estructura de Cuadros de Constantine.**
  - **Tareas:** Selección de la arquitectura global del sistema, Descomposición en subsistemas de sistemas, Identificación de concurrencias, Control de los recursos compartidos, Gestión de los datos, Diseño de las interfaces
- ✓ **Diseño de bajo nivel o detallado**: Realizado a partir del de alto nivel que, en el caso de un software se puede detallar en:
  - **Diseño de datos**: Tanto lógico como físico (por ejemplo, MLRelacional y la implantación en un SGBDR concreto).
  - **Diseño procedimental:** Transforma los **elementos estructurales en una descripción de los elementos del software**.
  - **Tareas:** Identificación y manejo de excepciones, Selección de los algoritmos y de las estructuras de datos, Preparación para la migración y carga inicial de datos. Especificación técnica de las pruebas



2.1. ARQUITECTURA DE SISTEMAS MÁS COMUNES

La mayoría de los problemas se pueden resolver utilizando alguna de las arquitecturas que mencionaremos a continuación:

Transformación por lotes	Simulación dinámica
Transformación continúa	Sistema en tiempo real
Interfaz interactiva	Controlador de transacciones

2.2. EVALUACIÓN CALIDAD DEL DISEÑO

2.2.1. FAN-OUT Y FAN-IN

- ✓ El **fan-out** de un módulo es usado como una **medida de complejidad**. Es el **número de subordinados inmediatos** de un módulo (cantidad de módulos invocados).
  - Si un módulo tiene un fan-out muy grande, entonces compone el trabajo de muchos módulos subordinados, lo que supone una mayor complejidad.
  - Para tener acotada ésta se debe limitar el fan-out a no más de siete más o menos dos (7 ± 2).
- ✓ El **fan-in** de un módulo es usado como una **medida de reusabilidad**, es el **número de superiores inmediatos** de un módulo (la cantidad de módulos que lo invocan).
  - Un alto fan-in es el resultado de una descomposición inteligente.
  - Los módulos con **mucho fan-in deben tener alta cohesión**, con lo cual es muy probable que tengan buen acoplamiento con sus llamadores; Interfaz Consistente:
  - Cada **invocación para el mismo módulo** debe tener el **mismo número y tipo de parámetros**.
  - Dos características deben ser garantizadas en módulos con un alto fan-in:
    - **Buena Cohesión**: Los módulos con mucho fan-in deben tener alta cohesión, por lo que es muy probable que tengan buen acoplamiento con sus llamadores;
    - **Interfaz Consistente**: Cada invocación para el mismo módulo debe tener el mismo número y tipo de parámetros.

2.2.2. DESCOMPOSICIÓN

Se trata de **separar una función contenida en un módulo**, para que conforme un nuevo módulo. Puede hacerse por distintas razones: para reducir el tamaño del módulo que hará más sencillo el trabajar con él,

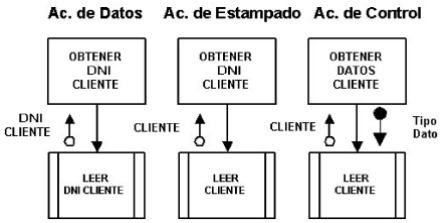
por hacer el sistema más claro permitiendo una mejor comprensión del diseño al subdividir módulos, para minimizar la duplicación de código reutilizando éste en otras partes del sistema, o bien para crear módulos más generales que serán más útiles y reutilizables en el mismo sistema y, además, pueden ser generadas bibliotecas de **módulos reutilizables en otros sistemas**.

2.2.3. ACOPLAMIENTO

Grado de interdependencia entre módulos. Se debe conseguir que los módulos sean lo más independientes entre sí.

- ✓ Cuantas menos conexiones existan entre dos módulos menor será la posibilidad de que aparezcan efectos colaterales al modificar uno de ellos.
- ✓ Posibilita que cada cambio realizado en un módulo afecte lo menos posible a otros, esto es, permite cambiar un módulo con el mínimo riesgo de tener que cambiar otro.
- ✓ Garantiza que mientras se está manteniendo un módulo no hay necesidad de preocuparse de los detalles internos (código) de cualquier otro módulo

Mínimo acoplamiento. Características: **Externo. Interfaz** (parámetros intercambiados entre módulos). De MENOR A MAYOR



- ✓ **Normal o simple**: los módulos sólo intercambian **datos**.
  - *De datos*: Los parámetros son **unidades elementales de datos**.
  - *Por estampado (de marca)*: parámetros que pueden ser registros (**estructuras**).
  - *De control*: algún parámetro es de control, i.e., que controla las decisiones de los **módulos subordinados o superiores**.
- ✓ **Externo**: los módulos están **ligados a componentes externos** (controladores E/S, …)
- ✓ **Común**: varios módulos hacen referencia a un **área común de datos** y pueden modificar los valores de los elementos de datos (p.e., variables globales).
- ✓ **De contenido**: Un módulo **hace referencia a la parte interna** de otro modificándola

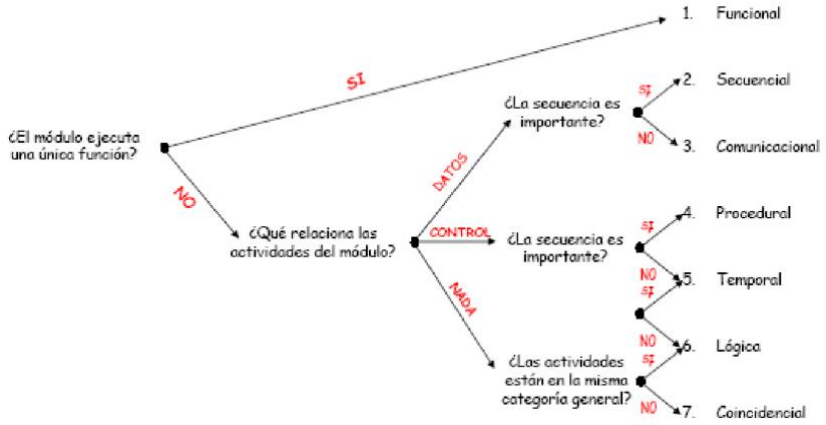
2.2.4. COHESIÓN

Definición: Relación de los elementos de un módulo. Un módulo con alta cohesión realiza una tarea concreta y sencilla.

Máxima Cohesión. Características: **Interno.** De MAYOR A MENOR

- ✓ **Funcional**: Todos los elementos del módulo están relacionados para desarrollar una única función.
- ✓ **Secuencial**: Un módulo empaqueta en secuencia varios módulos con cohesión funcional.
- ✓ **De comunicación:** Todos los elementos de procesamiento (p.e., funciones del módulo) utilizan los mismos datos de entrada y salida.
- ✓ **Procedimental o procedural:** Todos los elementos de procesamiento del módulo están relacionados y deben ejecutarse en un orden determinado. Paso de controles.
- ✓ **Temporal:** Un módulo contiene tareas relacionadas por el hecho de que todas deben realizarse en el mismo intervalo de tiempo.

- ✓ **Lógica:** Sus elementos contribuyen en actividades de una misma categoría general. La actividad o las actividades a ser ejecutadas son seleccionadas fuera del módulo mediante una señal de control
- ✓ **Casual o coincidental:** Un módulo realiza un conjunto de tareas que tienen poca o ninguna relación



2.3. Diagramas de Estructura (o Diagrama de Estructura de Cuadros)

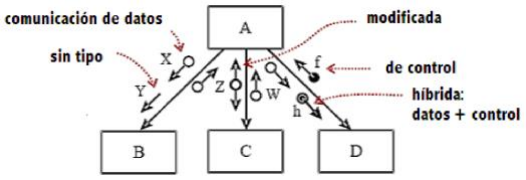
El objetivo de este diagrama es representar la **estructura modular del sistema** o de un **componente del mismo** y definir los parámetros de **entrada y salida de cada uno de los módulos**

Cada **nivel de la jerarquía** representa una descomposición más detallada del módulo del nivel superior.

La notación usada se compone básicamente de tres símbolos: módulos, invocaciones (relaciones entre módulos) y cuplas (comunicaciones entre módulos).

2.3.1. Componentes DEC

- ✓ **Módulos** Un módulo es un conjunto de instrucciones que ejecutan alguna actividad y se presenta como una **caja negra**. Deben ser pequeños e independientes y con tipos predefinidos.
  - Cuatro características básicas: **Entradas y Salidas**, **Función, Lógica Interna,** Estado **Interno**.
- ✓ **Conexión (o invocaciones):** representa una llamada de un módulo a otro.
  - Tipos de llamadas entre módulos: **Secuencial, Repetitiva**, **Alternativa**
- ✓ **Parámetro (cuplas)** Información que se intercambia entre los módulos
  - Tipos de parámetros:
    - **Control (o flags)**: valores de condición. Sincronizan la operativa.
    - **Datos**: información compartida entre módulos y procesada.
    - **Cupla Modificada**. Con una flecha doble (apuntando al módulo llamador y al módulo llamado): argumento que deberá modificar su valor.
    - **Cupla de Resultados**. Existen módulos que retornan valores sin la necesidad de que estén inicializados en el momento que se invocan.



- ✓ **Almacén de datos:** Representación física del lugar donde están almacenados los datos
- ✓ **Dispositivo físico:** cualquier dispositivo por el cual se puede recibir o enviar información que necesite el sistema

2.4. Estrategias de Diseño Estructurado

El diseño estructurado permite una **transición del DFD a una descripción de diseño de la estructura del programa**. Se definen unos pasos que están en función del tipo de flujo de información de que se trate.

a**. Flujo de transformación**: flujo de entrada, transformación, flujo de salida.

b. **Flujo de transacción**: flujo de **datos de entrada, evaluación de la transacción, selección del camino de acción** a ejecutar en función de la evaluación.

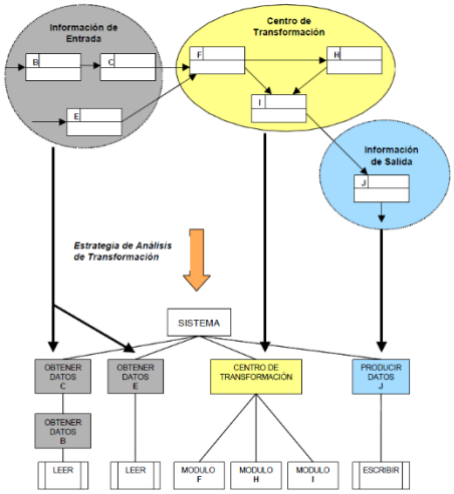
2.4.1. Análisis de transformación

Un **DFD con características de transformación** es aquél en el que se pueden distinguir:

- ✓ Flujo de llegada o entrada.
- ✓ Flujo de transformación o centro de transformación que contiene los procesos esenciales del sistema y es independiente de las características particulares de la entrada y la salida.
- ✓ Flujo de salida.

Los pasos a realizar en el análisis de transformación son (ver ejemplo de M3):

1. **Identificar el centro de transformación**.
2. Realizar el **"primer nivel de factorización"** o descomposición del diagrama de estructura. Habrá que identificar tres módulos subordinados: controlador del proceso de información de entrada, controlador del centro de transformación, controlador del proceso de la información de salida.
3. **Elaborar el "segundo nivel de factorización**
4. **Revisar y factorizar.**

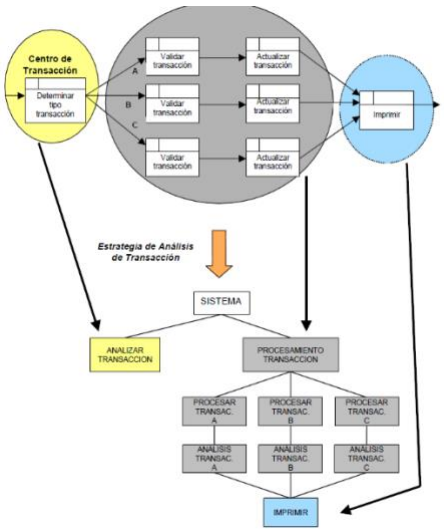


2.4.2. Análisis de transacción

El análisis de transacción se aplica cuando en un DFD existe un proceso en función del flujo de llegada, que determina la elección de uno o más flujos de información.

Se denomina centro de transacción al proceso desde el que parten los posibles caminos de información. Los pasos a realizar en el análisis de transacción son:

1. Identificar el **centro de transacción**.
2. Transformar el DFD en la **estructura adecuada al proceso de transacciones**.
3. **Factorizar la estructura** de cada camino de acción.
4. **Refinar la estructura del sistema** utilizando medidas y guías de diseño.



3. DISEÑO ORIENTADO A OBJETOS

El diseño orientado a objetos se define según Booch[1] como un "*método de diseño abarcando el proceso de descomposición orientado a objetos y una notación para representar ambos modelos lógico y físico tal como los modelos estáticos y dinámicos del sistema bajo diseño*".

En la documentación adjunta se detallan los principios fundamentales que definen el diseño orientado a objetos, incluyendo a continuación una serie de patrones y buenas prácticas relacionados con este tipo de diseños.

3.1. CARACTERÍSTICAS

- ✓ **Modularidad**. Soporte natural a la descomposición en módulos. Ej: clases
- ✓ **Ocultación de implementación (information hiding)**. La clase aporta la ocultación de detalles de implementación mediante la separación de la interfaz de la clase de su implementación
- ✓ **Acoplamiento débil**. Solo los operadores de la interfaz de la clase pueden acceder a su contenido interno.
- ✓ **Cohesión fuerte.**
- ✓ **Extensibilidad.** Se permite gracias a:
    - Herencia (reutilización de especificaciones)
    - Polimorfismo (permitir enviar mensajes sintácticamente iguales a objetos distintos, pudiendo responder estos de forma distinta, gracias al uso de la herencia)

[1] Booch, G. 1988. Object Oriented Development. Trans. of Soft. Eng. Vol. SE-12. Num. 2.Feb. 1986. pp. 211-221.

- ✓ **Integrable**. En un diseño global
- ✓ **Reusabilidad.** Técnicas de diseño y programación para reutilización de módulos software

3.2. METODOLOGÍAS

- ✓ Object-Oriented Design (OOD), Booch.
- ✓ Object Modeling Technique (OMT), Rumbaugh.
- ✓ Object Oriented Analysis (OOA), Coad/Yourdon.
- ✓ Hierarchical Object Oriented Design (HOOD), ESA.
- ✓ Object Oriented Structured Design (OOSD), Wasserman.
- ✓ Object Oriented Systems Analysis (OOSA), Shaler y Mellor
- ✓ UML: Unified Modeling Language (no es una metodología como tal)

3.3. PRINCIPIOS SOLID

SOLID es un acrónimo mnemónico introducido por Robert C. Martin[2] que define una serie de principios que favorecen el diseño y la mantenibilidad del software orientado a objetos:

- ✓ **S – The Single Responsibility Principle (SRP)**: Principio de Responsabilidad Única
  "Una clase debería tener solo una razón para cambiar"
- ✓ **O – The Open/Closed Principle (OCP)**: Principio Abierto / Cerrado
  "Una clase debería poder ser extendida, sin ser modificada"
- ✓ **L – The Liskov Substitution Principle (LSP)**: Principio de Sustitución de Liskov
  "Clases derivadas deben ser sustituibles por sus clases bases"
- ✓ **I – Interface Segregation Principle (ISP)**: Principio de Segregación de Interfaces
  "Muchas interfaces muy especializadas son preferibles a una interfaz general en la que se agrupen todas las interfaces"
- ✓ **D – The Dependency Inversion Principle (DIP)**: Principio de Inversión de Dependencias
  "Las abstracciones no deben depender de los detalles, los detalles deben depender de las abstracciones"

3.4. PRINCIPIOS GRASP

GRASP (object-oriented design General Responsibility Assignment Software Patterns) o conjunto de buenas prácticas en el diseño orientado a objetos, según lo describe Craig Larman:

- ✓ Experto en Información.
- ✓ Creador.
- ✓ Alta Cohesión.
- ✓ Bajo Acoplamiento.
- ✓ Polimorfismo.
- ✓ Fabricación pura.
- ✓ Indirección.
- ✓ Variaciones protegidas.
- ✓ Controlador.

[2] "Principles Of OOD", Robert C. Martin ("Uncle Bob"), butunclebob.com.

4. PROGRAMACIÓN ORIENTADA A ASPECTOS

Es un paradigma de programación reciente que busca la modularización y separación de responsabilidades. Tiene como ventajas un código menos enmarañado, menor dependencia, facilidad para depurar y modificar y más reusabilidad.

4.1. FUNDAMENTOS

- Lenguaje para definir la funcionalidad básica, de propósito general
- Lenguaje de aspectos, como AspectJ
- El tejedor, que combina ambos lenguajes

4.2. CONCEPTOS BÁSICOS

- Aspecto (Aspect). Funcionalidad transversal (cross-cutting) que se va a implementar de forma modular y separada del resto del sistema.
- Punto de cruce o unión (Join point). Punto de ejecución donde se puede conectar un aspecto: llamada a método, excepción, etc.
- Consejo (Advice). Implementación del Aspecto que se inserta en la aplicación de los Puntos de Cruce.
- Puntos de corte (Pointcut). Define mediante expresiones regulares, patrones de nombres o dinámicamente, los Consejos que se aplicarán a cada Punto de Cruce.
- Introducción (Introduction). Permite añadir métodos o atributos a clases ya existentes.
- Destinatario (Target). Clase aconsejada, que contiene su lógica, además de la del Aspecto.
- Resultante (Proxy). Objeto creado después de aplicar el Consejo al Objeto Destinatario.
- Tejido (Weaving). Objeto creado después de aplicar el Consejo al Objeto Destinatario, pudiendo ocurrir en tiempo de compilación, carga o ejecución.

5. DISEÑO POR CAPAS

Es la forma de lograr separación e independencia, fundamentales para el diseño arquitectónico. La funcionalidad del sistema se organiza en capas y cada una se apoya en las facilidades y servicios ofrecidos por la capa bajo ella.
Facilita el desarrollo incremental y la portabilidad.

ARQUITECTURA DE 2 CAPAS

Separación entre capa de presentación (usuario/cliente) y capa de aplicación y gestión de datos, donde se incluye la lógica de negocio y el modelo de datos.

MIDDLEWARE

Nivel de indirección entre los clientes y las demás capas del sistema. Capa adicional en la lógica de negocio que simplifica el diseño y facilita el acceso a los clientes, mediante la reducción del número de interfaces.

SISTEMA DE 3 CAPAS O MULTICAPA

Se incluyen tres capas que están completamente separadas:

- Presentación. Usuario/cliente
- Aplicación. Lógica de negocio
- Capa de gestión de datos. Modelo de datos

Permite el soporte para múltiples clientes, separación y replicación de datos y lógica de aplicación separada. Con esto se gana en modularidad.
Ventajas: reducción de tráfico, flexibilidad, independencia, fiabilidad.

ARQUITECTURA CLIENTE SERVIDOR

Modelo de aplicación distribuida en el que las tareas se reparten entre los proveedores de recursos o servicios (servidores) y los demandantes de servicios o clientes. Un ejemplo es la web.
Niveles:

- Presentación
- Aplicación
- Comunicación
- Base de datos

Puede existir middleware, módulo intermedio que actúa como conductor entre sistemas permitiendo a cualquier usuario de sistemas de información comunicarse con varias fuentes de información.
Tipos de arquitectura:

- Cliente pesado (fat client). El peso de la aplicación lo ejecuta el cliente. El servidor se encarga básicamente de las labores de administración de base de datos. Usado en DSS (soporte a la decisión) y EIS (sistema de información ejecutivo).
- Cliente ligero (thin client) y servidor pesado. El proceso cliente está restringido a la presentación. El peso de la aplicación recae en el servidor.

Alternativas en cuanto a separación de funciones:

- Presentación Distribuida.
- Presentación remota.
- Lógica o procesamiento distribuidos.
- Acceso a datos remoto.
- Bases de datos distribuidas

6. PATRONES DE DISEÑO

Es una solución “repetible” a un problema “común” de diseño. Se debe haber comprobado su efectividad y ser reutilizable.
Orígenes: 1979 por el arquitecto Chistopher Alexander.

Aplicación al mundo de la informática en la década de los 90: “Design Patterns” escrito por el grupo Gang of Four (GoF) compuesto por Erich Gamma, Richard Helm, Ralph Johnson y John Vlissides.
No pretenden imponer alternativas ni limitar la creatividad.

OBJETIVOS

- Catálogo de elementos reusables
- Evitar reiteración en búsqueda de soluciones
- Formalizar vocabulario común
- Estandarizar diseño
- Facilitar aprendizaje

CATEGORÍAS

- ✓ Patrones de arquitectura. Esquema organizativo estructural para sistemas software.
- ✓ Patrones de diseño. Esquemas para definir estructuras de diseño software.

- ✓ Dialectos. Bajo nivel, para un lenguaje de programación o entorno.

6.1. PATRONES GOF

6.1.2 PATRONES CREACIONALES

- ✓ **Object pool** (conjunto de objetos). No especificador por GoF. Se obtienen objetos nuevos a través de la clonación, al ser menos costosa que la creación.
- ✓ **Abstract Factory** (Factoría abstracta). Crear objetos de una misma familia, no mezclando entre sí los objetos de distintas familias.
- ✓ **Builder** (Constructor virtual). Permitir la creación de una variedad de objetos complejos desde un objeto fuente (Producto), centralizándose en un único punto.
- ✓ **Factory Method** (Método de Fabricación). Centraliza en una clase constructora (que contiene ya algunos métodos definidos) la creación de objetos de un subtipo de un tipo determinado, ocultando la casuística para elegir el subtipo.
- ✓ **Prototype** (Prototipo). Crea nuevos objetos clonándolos de una instancia ya existente
- ✓ **Singleton** (Instancia única). Permite restringir la creación de objetos pertenecientes a una clase o el valor de un tipo a un único objeto, garantizando la existencia de una única instancia de la clase y un mecanismo de acceso global a esta.

6.1.2 PATRONES ESTRUCTURALES

- ✓ **Adapter** (Adaptador). Adapta una interfaz en otra, para que pueda ser utilizada por una clase que de otro modo no podría utilizarla.
- ✓ **Active record** (registro activo): en el ORM para que el objeto gestione su propia persistencia en una BD relacional (incluye operaciones-> insert, update... y atributos ->columnas).
- ✓ **Bridge** (Puente): Desacopla una abstracción de su implementación, para que ambas puedan ser modificadas de forma independiente.
- ✓ **Composite** (Objeto compuesto): Permite tratar objetos compuestos como si de uno simple se tratase. Se construye el objeto a partir de otros más simples.
- ✓ **Decorator** (Envoltorio): Añade funcionalidad a una clase dinámicamente
- ✓ **Facade** (Fachada): Provee de una interfaz unificada simple para acceder a una interfaz o grupo de interfaces de un subsistema más complejo.
- ✓ **Flyweight** (Peso ligero): Reduce la redundancia cuando gran cantidad de objetos poseen idéntica información, logrando equilibrio entre flexibilidad y rendimiento (uso de recursos)
- ✓ **Proxy**: Proporciona un intermediario o representante de un objeto para controlar su acceso.

6.1.3 PATRONES DE COMPORTAMIENTO

- ✓ **Chain of Responsibility (Cadena de responsabilidad):** Permite establecer la línea que deben llevar los mensajes para que los objetos realicen la tarea indicada. Evita acoplar el emisor de una petición a su receptor dando a más de un objeto la posibilidad de responder a una petición.
- ✓ **Command** (Orden): Encapsula una operación en un objeto, permitiendo ejecutar dicha operación sin necesidad de conocer el contenido de la misma, ni el receptor real de esta.
- ✓ **Interpreter** (Intérprete): Dado un lenguaje, define una representación para su gramática, así como las herramientas necesarias para interpretarlo.

- ✓ **Iterator** (Iterador): Permite realizar recorridos sobre objetos compuestos. Define una interfaz que declara los métodos necesarios para acceder secuencialmente a un grupo de objetos de una colección.
- ✓ **Mediator** (Mediador): Define un objeto que coordine la comunicación entre objetos de distintas clases, pero que funcionan como un conjunto.
- ✓ **Memento** (Recuerdo): Permite almacenar el estado de un objeto en un momento dado de manera que se pueda restaurar en ese punto de manera sencilla.
- ✓ **Observer** (Observador): Define una dependencia de uno-a-muchos entre objetos, de forma que cuando un objeto cambie de estado se notifique y actualicen automáticamente todos los objetos que dependen de él.
- ✓ **State** (Estado): Permite que un objeto modifique su comportamiento cada vez que cambie su estado interno.
- ✓ **Strategy** (Estrategia): Permite disponer de varios métodos para resolver un problema y elegir cuál utilizar en tiempo de ejecución. Determina cómo se debe realizar el intercambio de mensajes para resolver una tarea.
- ✓ **Template Method** (Método plantilla): Define en una operación el esqueleto de un algoritmo, delegando en las subclases algunos de sus pasos, esto permite que las subclases redefinan ciertos pasos de un algoritmo sin cambiar su estructura.
- ✓ **Visitor** (Visitante): Permite separar el algoritmo de la estructura de un objeto. Permite definir nuevas operaciones sobre una jerarquía de clases sin modificar las clases sobre las que opera.

6.2. ANTIPATRONES DE DISEÑO

Es un patrón de diseño que invariablemente conduce a una mala solución para un problema.
Ejemplos:

- Antipatrones de gestión de proyectos (Humo y espejos, Software inflado)
- Antipatrones generales de diseño de software (Sistema de cañerías de calefacción, Entrada chapuza, Clase Gorda, chapado de oro)
- Antipatrones de diseño orientado a objetos (Singletonitis, Fallo de clase vacía, Objeto todopoderoso)
- Antipatrones de programación (Ancla del barco, Lógica super-booleana, Código espagueti, Programación por excepción)
- Antipatrones metodológicos (Bala de plata, Martillo de oro , Reinventar la rueda, Reinventar la rueda cuadrada)
- Antipatrones de gestión de la configuración (Conflicto de extensiones, Infierno de dependencias)

6.3. PATRONES DE DISEÑO MODERNOS

Los patrones de diseño modernos son una evolución de los patrones de diseño GOF y se enfocan en problemas de diseño más actuales y en nuevas tecnologías. A continuación, se presentan algunos ejemplos de patrones de diseño modernos:

- Inyección de dependencias (**Dependency Injection**): Se trata de una técnica para crear objetos que dependen de otros objetos. En lugar de que un objeto cree sus dependencias directamente, estas se le suministran a través de un contenedor de dependencias. Esto ayuda a *reducir el acoplamiento* y hace que el código sea más fácil de probar.

- Manejador de eventos (**Event Sourcing**): Se utiliza en sistemas que requieren la persistencia de eventos. En lugar de almacenar el estado actual del sistema, se almacenan todos los eventos que han ocurrido. El estado actual del sistema se puede calcular a partir de los eventos almacenados. Es útil para sistemas altamente concurrentes y distribuidos.

- CQRS (**Command Query Responsibility Segregation**): Separa las operaciones de lectura y escritura en dos modelos diferentes. El modelo de escritura se encarga de actualizar el estado del sistema, mientras que el modelo de lectura se encarga de leer y mostrar la información. Esto puede ayudar a mejorar el rendimiento y la escalabilidad del sistema.

- **Patrón Saga**: Se utiliza en sistemas distribuidos y es especialmente útil para el manejo de transacciones a largo plazo. Se utiliza para garantizar que todas las operaciones dentro de una transacción se completen correctamente o se anulen en caso de error. En un sistema distribuido, una transacción puede involucrar múltiples servicios y componentes que se comunican entre sí. El patrón Saga se basa en la idea de que una transacción se puede dividir en múltiples operaciones más pequeñas llamadas "*etapas*" que se pueden ejecutar de forma independiente. Cada etapa tiene dos acciones asociadas: "compensar" y "confirmar". La acción de confirmación se realiza cuando la etapa se completa correctamente, mientras que la acción de compensación se realiza cuando se produce un error. Si una etapa falla, se ejecutan las acciones de compensación de las etapas anteriores para revertir los cambios realizados en la transacción. El patrón Saga utiliza un *coordinador central* para coordinar y controlar todas las etapas de la transacción. El coordinador es responsable de iniciar las etapas de la transacción, monitorear su progreso y tomar decisiones en caso de errores.