## CAPÍTULO 6: Diseño de Software

### Diseño

El diseño de software agrupa el **conjunto de principios, conceptos y prácticas** que llevan al desarrollo de un sistema o **producto de alta calidad**. El diseño es una representación significativa de ingeniería de algo que vamos a construir.

### Características de un buen diseño

* **Resistencia:** un programa no debe tener ningún error que impida su funcionamiento.
* **Funcionalidad:** un programa debe ser apropiado para los fines que persigue.
* **Belleza:** la experiencia de usar el programa debe ser placentera.

### Etapas o niveles de detalle del diseño

Comienza con el modelo de requerimientos. Se trabaja para transformar este modelo y obtener cuatro niveles de detalle de diseño:

* La estructura de **datos**
* La **arquitectura** del sistema
* La representación del **interfaz**
* Detalles a nivel de **componentes**

**Se produce una especificación del diseño** que se compone de los modelos del diseño que describen, los datos, la arquitectura, interfaces y componentes.

En cada etapa para saber si lo hecho es correcto, se revisan los productos del diseño del software, en cuanto a calidad, corrección, finalización y consistencia

### ¿Cómo influye el diseño (arquitectura) en el sistema?

La arquitectura del sistema afecta al rendimiento, solidez, grado de distribución y mantenibilidad de un sistema (**Requerimientos No Funcionales**).

* Rendimiento → Subsistemas
* Protección → Capas
* Seguridad → Único Subsistema
* Disponibilidad → Componentes Redundantes
* Mantenibilidad → Componentes Independientes

### Elementos del Modelo de Análisis

El modelo de análisis debe lograr tres objetivos:

* Describir lo que quiere el cliente
* Establecer una base para el diseño de software
* Definir un conjunto de requerimientos que pueda validar una vez que se construye el software.

Cada parte del modelo de análisis proporciona la información necesaria para crear los cuatro modelos de diseño. El modelo de análisis extraído de un diseño estructurado es el siguiente:

* **DISEÑO DE DATOS** → estructuras de datos
* **DISEÑO ARQUITECTÓNICO** → Define relación entre los elementos estructurales,patrones de diseño, los requisitos y las restricciones
* **DISEÑO DE INTERFAZ** → la manera de comunicarse el software dentro de sí mismo
* **DISEÑO DE NIVEL DE COMPONENTES** → descripción procedimental de los componentes

### El Proceso de Diseño

El diseño del software es un proceso iterativo mediante el cual los requisitos se traducen en un «plano». Se representa desde un nivel más global (abstracto) y va hacia lo más particular (comportamiento, funcionalidades y de datos).

### Diseño y Calidad

La calidad de la evolución del diseño se evalúa con una serie de revisiones técnicas formales:

* El diseño deberá implementar todos los requerimientos.
* El diseño proporcionará una imagen completa del software.
* El diseño es una guía para los que generan código.

Los **FACTORES DE CALIDAD EXTERNOS** son esas propiedades del software que pueden ser observadas fácilmente por los usuarios.

Los **FACTORES DE CALIDAD INTERNOS** tienen importancia para los ingenieros de software. Desde una perspectiva técnica conducen a un diseño de calidad alta.

## CAPÍTULO 7: Diseño Arquitectónico

### ¿Qué es la arquitectura del software?

La arquitectura de software es la **estructura integral de un sistema**, que se manifiesta en sus **componentes** principales, sus **interacciones** y el **entorno** en el que operan. Incluye los principios y directrices que guían su diseño, desarrollo y evolución a lo largo del tiempo.

### Vistas Arquitectónicas

Una vista es una **representación de** un conjunto de **elementos** del sistema y las **relaciones** asociadas con ellos.

Documentar una arquitectura significa **documentar las vistas relevantes bajo las cuales se puede observar un sistema**. Se tienen uno o más documentos de vista y documentación que explican cómo se relacionan las vistas entre sí. Las diferentes vistas exponen diferentes atributos de calidad en diferentes grados.

### Vistas según el modelo 4+1 de Kruchten

### Documentos para una vista

La documentación para una vista incluye una **representación gráfica** de los elementos principales y sus relaciones, **un catálogo de elementos** que define sus propiedades, **una especificación de las interfaces** y **el comportamiento de cada elemento**.

### Documentación de un Arquitectura

En la documentación de una arquitectura de software, es importante considerar lo siguiente:

* **Representación del sistema en el contexto:** Se debe representar el sistema objetivo en relación con su entorno, identificando sistemas superiores, sistemas subordinados, sistemas entre iguales y actores que interactúan con el sistema.
* **Identificación de arquetipos de la arquitectura:** Los arquetipos son clases o patrones fundamentales para el diseño de la arquitectura. Representan abstracciones críticas y pueden ser componentes genéricos o módulos especializados con funciones específicas.
* **Refinamiento hacia los componentes:** Los arquetipos se refinan en componentes más específicos, definiendo interfaces e interacciones entre

**Errores más comunes al documentar:**

* Demasiado elaborado
* Incógnitas
* Se hace difícil de mantener

## CAPÍTULO 8: Patrones de Diseño

### Patrones

Un patrón es un par problema/solución con un nombre que se puede aplicar en nuevos contextos, con consejos acerca de cómo aplicarlo en nuevas situaciones y discusiones sobre sus compromisos.

### Patrones GRASP

**GRASP** es una acrónimo de **G**eneral **R**esponsibility **A**ssignment **S**oftware **P**atterns.

**GRASP Básicos:**

* Experto en información
* Creador
* Alta Cohesión
* Bajo Acoplamiento
* Controlador

**GRASP Avanzados:**

* Polimorfismo
* Fabricación Pura
* Indirección
* Variaciones Protegidas

### GRASP Básicos

#### Experto en información

* **Problema:** ¿Quién asigna responsabilidades a los objetos?
* **Solución:** Asignar una responsabilidad al experto en información (la clase que tiene la información necesaria para realizar la responsabilidad). Siguiendo la solución del patrón deberíamos buscar clases y objetos que tienen la información necesaria para determinar el total. Este patrón es un principio de guía básico en el diseño de objetos. Expresa la “intuición” de lo que creemos que los objetos del mundo real hacen y por eso es muy fácil de aplicar.
* **Contraindicaciones:** En algunas circunstancias la solución de “experto” no es deseable. Por ejemplo para almacenar en una BD la información.
* **Beneficios:** Se mantiene el encapsulamiento de la información (los objetos utilizan su propia información para trabajar) Se distribuye el comportamiento entre clases obteniendo clases más pequeñas, fáciles de entender y manejar. Se logra una alta cohesión.

#### 

#### Creador

* **Problema:** ¿Quién debería ser el responsable de la creación de una nueva instancia de alguna clase?
* **Solución:** Asignar a la clase B la responsabilidad de crear una instancia de A si se cumple uno o más de los siguientes casos: B **agrega** objetos de A, B **contiene** objetos de A, B **registra** instancias de objetos de A, B es un creador de los objetos A, etc. La intención del patrón Creador es encontrar el creador que necesita a futuro conectarse al objeto creado en alguna situación. El Creador sugiere que las clase contenedoras son buenas para tener la responsabilidad de la creación de sus clases contenidas.
* **Contraindicaciones:** Cuando la creación requiere alta complejidad (reciclo para mayor rendimiento) es complicado la creación de instancias.
* **Beneficios:** Se soporta el bajo acoplamiento.

#### Bajo Acoplamiento

* **Problema**: ¿Cómo soportar bajas dependencias, bajo impacto en el cambio e incremento de la reutilización?
* **Solución**: Asignar una responsabilidad de manera que el acoplamiento permanezca bajo. El acoplamiento es una medida de la fuerza con que un elemento está conectado a otros elementos. Un elemento con bajo acoplamiento no depende de demasiados otros elementos. La no existencia de acoplamiento (clase independientes) no es deseable ya que atenta contra el objetivo de los diseños orientados a objetos donde el sistema es un conjunto interconectado de objetos trabajando entre sí.
* **Contraindicaciones**: La aplicación de bajo acoplamiento para “Futuras Necesidades” donde no hay motivos realistas para hacerlo puede ser contraproducente.
* **Beneficios**: No afectan los cambios en otros componentes Fácil de Entender de manera aislada Conveniente para reutilizar .

#### Alta Cohesión

* **Problema**: ¿Cómo mantener la complejidad manejable?
* **Solución**: Asignar una responsabilidad de manera que la cohesión permanezca alta. La cohesión es una medida de fuerza con la que se relacionan y del grado de focalización de las responsabilidades de un objeto. Una clase con baja cohesión hace muchas cosas no relacionadas o hace demasiado trabajo. Las clases con baja cohesión representan un grano grande de abstracción (deberían haberse delegado en otros objetos) Cohesión y acoplamiento el yin y el yang à Una mala cohesión causa normalmente un mal acoplamiento y viceversa debido a que uno influye en el otro.
* **Beneficios**: Se incrementa la claridad y comprensión del diseño. Se simplifica el mantenimiento y mejoras. Se soporta a menudo bajo acoplamiento. El grado fino de funcionalidad incrementa la reutilización.

#### 

#### Controlador

* **Problema:** ¿Quién debe manejar eventos del sistema?
* **Solución**: Asignar la responsabilidad de recibir o manejar eventos del sistema a una clase. Un controlador es un objeto responsable del manejo de los eventos del sistema. Se pueden asignar a una o más clases controlador. Normalmente los controladores deben delegar las tareas a otros objetos, su misión solo debería ser coordinar o controlar las actividades. Esto para no sobrecargar la clase y perder cohesión. Es decir un controlador saturado.
* **Beneficios**: Aumenta el potencial para reutilizar y permite el intercambio de interfaces fácilmente.

### GRASP Avanzados

#### Polimorfismo

En la programación orientada a objetos el polimorfismo es permitir que varias clases se comporten de manera distinta dependiendo del tipo que sean. Siempre que se tenga que llevar a cabo una responsabilidad que dependa de un tipo, se tiene que hacer uso del polimorfismo.

* **Problema:** ¿Cómo manejar alternativas basadas en el tipo? ¿Cómo crear componentes de software conectables?
* **Solución**: Cuando las alternativas o comportamientos relacionados varían con el tipo de clase, asigne la responsabilidad del comportamiento (utilizando operaciones polimórficas) a los tipos para los que varía el comportamiento. Es un principio fundamental para gestionar variaciones similares.
* **Contraindicaciones**: Se corre el riesgo de diseñar sistemas con interfaces para “futuras necesidades” frente a variaciones desconocidas.
* **Beneficios**: Se añaden fácilmente extensiones necesarias de nuevas variaciones y Las nuevas implementaciones no afectan a los clientes.

#### Fabricación Pura

* **Problema**: ¿Qué hacer cuando se quiere mantener los objetivos de Alta Cohesión y Bajo Acoplamiento pero las soluciones que ofrece Experto no son adecuadas?
* **Solución**: Asigne un conjunto de responsabilidades altamente cohesivo a una clase artificial o de conveniencia, que no representa un concepto del dominio del problema. Muchas veces la elección de Experto trae aparejado problemas de poca reutilización, alto acoplamiento y baja cohesión como por ejemplo el caso del almacenamiento de datos en una base de datos.
* **Beneficios**: Soporta Alta cohesión y el potencial de reutilización aumenta debido a la presencia de clases Fabricación Pura que tienen aplicación en otros dominios.
* **Contraindicaciones**: La inexperiencia puede llevar a abusar de la creación de clases Fabricación Pura convirtiendo en muchos casos funciones en objetos.

#### Indirección

* **Problema**: ¿Dónde asignar una responsabilidad para evitar el acoplamiento directo entre dos o más cosas? ¿Cómo desacoplar objetos de manera que se soporte el bajo acoplamiento y el potencial para reutilizar permanezca alto?
* **Solución**: Asigne la responsabilidad a un objeto intermedio que funcione como mediador entre otros componentes o servicios de manera que no se acoplen directamente.
* **Beneficios**: Disminuir acoplamiento entre componentes.

#### Variaciones Protegidas

* **Problema**: ¿Cómo diseñar objetos, subsistemas, y sistemas de manera que las variaciones o inestabilidades en estos elementos no tengan un impacto no deseable en otros elementos?
* **Solución**: Asignar responsabilidades para crear una interfaz estable alrededor de puntos de variaciones previstas o de inestabilidad.
* **Beneficios**: Se añaden fácilmente las extensiones que se necesitan para nuevas variaciones, se pueden introducir nuevas implementación es sin afectar a los clientes. Se reduce el acoplamiento.

# Catálogo de Arquitecturas

## Estilo Flujo de Datos

### Tuberías y filtros

El estilo de tuberías y filtros es uno de los ejemplos más claros de las arquitecturas de flujo de datos. En esta arquitectura, los componentes (**filtros) están conectados por conductos (tuberías**), permitiendo que los datos fluyan de un filtro a otro. **Cada filtro realiza una transformación** en los datos que recibe y luego los pasa al siguiente filtro en la secuencia. Este estilo es muy adecuado para tareas que requieren procesamiento secuencial de datos, como los compiladores o sistemas operativos basados en UNIX​​.

## Estilo Centrado en Datos

### Arquitecturas de pizarra o repositorio

En una arquitectura de pizarra o repositorio, existen dos componentes principales: una **estructura de datos central** que representa el estado actual y una colección de **componentes independientes que operan sobre esa estructura**. Dependiendo de cómo se manejan las transacciones y los datos, el repositorio puede ser una base de datos tradicional o una pizarra pura, donde el estado de los datos dispara los procesos a ejecutar​​.

## Estilo Llamada y Retorno

### Model View Controller (MVC)

El patrón MVC divide una aplicación en **tres componentes principales**: **Modelo** (gestiona los datos), **Vista** (presenta los datos al usuario) y **Controlador** (maneja la entrada del usuario y actualiza el modelo y la vista). Este estilo permite una separación clara de las preocupaciones, facilitando el mantenimiento y la escalabilidad de las aplicaciones.

### Arquitectura Orientada a Objetos

En este estilo, el sistema está compuesto por **objetos que interactúan entre sí**. Cada objeto **encapsula datos y comportamiento**, y la comunicación entre objetos se realiza mediante **mensajes**. Este enfoque promueve la reutilización de código y la modularidad.

### Arquitectura en Capas

La arquitectura en capas organiza el sistema en **niveles jerárquicos**, donde cada capa **proporciona servicios a la capa superior y utiliza los servicios de la capa inferior**. Esto facilita el mantenimiento y la escalabilidad, ya que cada capa puede desarrollarse y modificarse de manera independiente.

### Arquitectura Basada en Componentes

Similar a la arquitectura orientada a objetos, pero a una escala mayor, donde los **componentes son unidades de implementación independientes que interactúan a través de interfaces bien definidas**. Esto permite una alta reutilización de componentes y una fácil integración de nuevas funcionalidades.

## Estilo de Código Móvil

### Arquitectura de Máquinas Virtuales

En este estilo, se utiliza una **máquina virtual para ejecutar código que puede moverse entre diferentes entornos de ejecución**. Esto proporciona **portabilidad y aislamiento**, permitiendo que el mismo código se ejecute en diferentes plataformas sin modificación.

## Estilos Heterogéneos

### Sistema de Control de Procesos

Este estilo se emplea en sistemas donde el control de procesos es crucial, como en la automatización industrial. Combina **diferentes enfoques arquitectónicos para manejar tanto el control en tiempo real como el procesamiento de datos y la interfaz de usuario**.

### Arquitectura Basada en Atributos

Esta arquitectura se enfoca en **satisfacer ciertos atributos de calidad como la seguridad, el rendimiento o la disponibilidad**. Los componentes y sus interacciones se diseñan para optimizar estos atributos específicos.

## Estilos Peer-to-Peer

### Arquitectura Basada en Eventos

En esta arquitectura, los **componentes se comunican mediante el envío y recepción de eventos**. Es adecuada para sistemas altamente distribuidos y descentralizados, donde la escalabilidad y la flexibilidad son esenciales.

### Arquitectura Orientada a Servicios (SOA)

SOA organiza el sistema en **servicios independientes que interactúan entre sí a través de interfaces bien definidas**. Cada servicio realiza una función específica y puede ser reutilizado en diferentes aplicaciones, facilitando la integración y la escalabilidad.

### Arquitectura Basada en Recursos

Este estilo se centra en **recursos** identificables y sus representaciones. Los **componentes interactúan mediante operaciones estándar sobre estos recursos**, lo cual es común en arquitecturas RESTful utilizadas en el desarrollo de servicios web.