# Paradigma orientado a objetos

En este tipo de paradigma se integra la realidad, a diferencia del paradigma estructurado que desintegra la realidad o propone una separación para solucionar algún problema. En la orientación a objetos, los bloques fundamentales de construcción son las **clases** y los **objetos**.
Con el POO se obtienen software industriales más flexibles, completos y de mejor calidad.

| POO | ESTRUCTURADO |
|---|---|
| Integra la realidad. | Desintegra la realidad. |
| Permite herencia | No permite herencia |
| Logra sistemas más flexibles | Logra sistemas difíciles de mantener |

## Ventajas del paradigma orientado a objetos

☑ Cambia nuestra forma de pensar (es más natural).
☑ Los sistemas se construyen a partir de objetos existentes, esto lleva a un alto grado de reutilización, ahorro de dinero, menor tiempo de desarrollo y mayor confiabilidad del sistema.
☑ La complejidad sigue en aumento ya que se construyen nuevos objetos a partir de otros (evolucionan en el tiempo).
☑ Ayuda a explotar la potencia de los lenguajes basados y orientados a objetos.
☑ Ayuda a disminuir la complejidad del software industrial.
☑ Todos los elementos se relacionan, esto permite una mejor rastreabilidad y facilita el mantenimiento del sistema. La rastreabilidad debe ser hacia delante y hacia atrás (desde los requerimientos al código y viceversa (reingeniería)). La vista de casos de uso es la que nos proporciona la rastreabilidad.

## Clases

Una **clase** es una abstracción de atributos y comportamientos comunes de un conjunto de objetos. Posee dos vistas:
☑ Vista **externa** (interfaz), que es aquello que se ve, es la declaración de las cosas que puede hacer el objeto (¿Qué hace?)
☑ Vista **interna** (implementación), que es lo que tiene que ser accesible a otros objetos (¿Cómo lo hacen?).

### Tipos de clases

☑ **Clase base, raíz o padre:** es aquella que no depende de otra clase, pero de ella si dependen otras.
☑ **Clase hoja:** es aquella que depende de una clase base y a su vez puede o no heredar a otras clases.
☑ **Clase abstracta:** es aquella que no posee instancias directas, sino que existen para heredar atributos y operaciones.
☑ **Clase concreta:** es aquella que posee instancias directas.

### Características de las clases

☑ **Nombre:** permite identificarlas.
☑ **Atributos:** propiedades comunes al conjunto de objetos que representa la clase.
☑ **Operaciones:** servicios que se les puede pedir a los objetos de la clase.

## Objetos

Un **objeto** es un elemento, unidad o entidad identificable, ya sea real o conceptual que tiene un rol bien definido en el dominio del problema cuyas fronteras están definidas con nitidez. Es una instancia de una clase.
Los objetos están formados por:
☑ **Atributos:** son cualidades o propiedades de los objetos.
☑ **Operaciones:** son todo aquello que puede hacer un objeto.

**Características de los objetos**

☑ **Estado:** cuando el objeto asume ciertos valores en un determinado tiempo, se dice que el objeto se encuentra en un estado. El estado se refleja en el valor de los atributos. La ejecución de ciertos métodos son los que cambian el valor de los estados (¿Qué sabe?).

☑ **Comportamiento:** define el conjunto de cosas que el objeto puede hacer, como actúa y reacciona en términos de sus cambios de estados y paso de mensajes (¿Qué hace?).

☑ **Identidad:** es lo que nos permite definir lo que el objeto es. Cada uno ocupa un lugar en el espacio y en el tiempo, logrando así la identidad única (¿Qué es?). La identidad es una propiedad que no se puede heredar, ya que la herencia es conceptual.

☑ **Responsabilidad:** es aquello que el objeto conoce sumado a lo que puede hacer. Es decir, el conjunto de atributos y operaciones de la clase que instancia al objeto.

☑ **Colaboración:** se representa como la solicitud de un cliente (objeto que utiliza los recursos de otro objeto, el servidor) a un servidor para cumplir con su responsabilidad.

# Modelo de Objetos

El modelo de objetos es el marco de referencia contextual que se utiliza para todo lo relacionado a objetos. Este modelo está formado por cuatro elementos fundamentales y tres secundarios.

## Elementos fundamentales

☑ **Abstracción:** es centrarse en las características esenciales de un objeto que lo distinguen de todos los demás. Esto permite combatir la complejidad dejando de lado detalles que por el momento son irrelevantes. Se centra en la visión externa del objeto.

☑ **Encapsulamiento:** utiliza el ocultamiento de información de manera que los objetos guarden para sí atributos y operaciones. Esto permite que al modificar un objeto no se altere el sistema. Protege la abstracción.

☑ **Modularidad:** es fragmentar un programa en componentes individuales que pueden compilarse por separado, pero que tienen conexiones con otros módulos, para reducir su complejidad. Los módulos son contenedores físicos de clases y objetos. Se debe hacer lo posible para construir módulos altamente cohesivos (agrupar abstracciones que guarden cierta relación) y débilmente acoplados (no estén tan relacionados).

☑ **Jerarquía:** es una forma de organizar o clasificar las abstracciones. Las más importantes son: **estructura de clase** (“es un” o herencia) y **estructura de objetos** (“es parte de” o agregación). La jerarquía queda determinada por la herencia.

La abstracción y el encapsulamiento son conceptos complementarios, mientras la abstracción se centra en el comportamiento observable de un objeto, el encapsulamiento se centra en la implementación que da lugar a este comportamiento.

☑ **Polimorfismo:** es la propiedad que permite a los objetos de clases distintas, responder a un mismo método de manera diferente.

## Elementos secundarios

Cada uno de ellos es una parte útil del modelo de objetos, pero no es esencial.

☑ **Tipo (tipificación):** los tipos son la puesta en vigor de la clase de los objetos. Así, los objetos de tipos distintos no pueden intercambiarse libremente. Caracteriza al objeto a nivel de atributo y comportamiento de tal manera que un objeto pertenece a esa clase y no a otra.

☑ **Concurrencia:** es la propiedad que distingue un objeto activo de uno que no lo está. Se dice que un objeto esta activo si cada objeto puede representar un hilo separado de control (abstracción de un solo proceso).

☑ **Persistencia:** es la propiedad de un objeto por la que su existencia trasciende el tiempo o el espacio. Es decir que continúa existiendo después que su creador deja de existir o trasciende la posición en la cual fue creado.

### Relaciones entre objetos

☑ **Enlace:** es una conexión (asociación) entre objetos, por la cual un objeto (cliente) le pide algo a otro (servidor). Estos se representan con diagramas de colaboración. El paso de mensaje entre dos objetos es generalmente unidireccional, aunque puede ser bidireccional. Un objeto puede desempeñar tres roles en un enlace: **actor** (puede operar sobre otros, pero otros no pueden operar sobre el), **servidor** (nunca opera sobre otros objetos, pero estos pueden operar sobre el) o **agente** (puede operar sobre otros y estos, a su vez pueden operar sobre el).
☑ **Agregación:** establece una relación jerárquica del todo con las partes.

### Relaciones entre clases

☑ **Generalización:** es una relación de herencia ("es un"). La herencia es una relación en la que una clase comparte la estructura o el comportamiento de una o más clases, llamadas superclases.
☑ **Asociación:** representa una relación entre dos clases en la que la naturaleza del vínculo no es aclarada.
☑ **Agregación:** es un tipo particular de asociación. Es una relación todo-partes.
☑ **Composición:** es una relación aun más fuerte que la agregación. En este caso si se elimina el todo, se eliminan las partes.
☑ **Dependencia o Uso:** la modificación de la clase usada puede ocasionar una modificación en la clase que la utiliza.

**Acoplamiento:** nivel de relación entre clases. Debe ser mínimo o bajo.
**Cohesión:** mide si las cosas que están dentro de una clase están relacionadas o no. Debe ser alta.
**Navegabilidad:** está dada por el sentido de la flecha en las relaciones.
**Multiplicidad o Cardinalidad de una relación:** representa que tantos objetos de una clase se relacionan con cuantos de la otra. Se simboliza con 0 (ninguno), 1 (uno) o * (muchos).

## El Lenguaje Unificado de Modelado (UML 1.4)

### ¿Qué es UML?

Es un lenguaje estándar para escribir planos de software.
☑ No es una metodología.
☑ Es independiente del proceso, aunque para usarlo óptimamente sus autores aconsejan utilizar un proceso dirigido por casos de uso, centrado en la arquitectura e incremental-iterativo.
☑ Proporciona un vocabulario y una serie de reglas para combinar las palabras y facilitar la comunicación.

#### Características

UML es un lenguaje para:
☑ **Visualizar:** ver el concepto a través de un elemento. Mostrar la realidad a través de los modelos, facilita la comunicación.
☑ **Especificar**, ya que cubre todas las descripciones del análisis, diseño e implementación que deben desarrollarse en sistemas complejos.
☑ **Construir**, si bien no es un lenguaje de programación visual, sus modelos pueden conectarse en forma directa a una gran cantidad de lenguajes de programación, como ser, JAVA, C++, Visual Basic, incluso bases de datos relacionadas.

☑ **Documentar**, desde requisitos hasta pruebas, prototipos, etc. Documentar es dejar asentado las distintas características de un sistema (la documentación asegura la calidad).

UML indica cómo crear y leer modelos pero no dice que modelos se deben crear ni cuando, por eso hace falta un proceso.

## Elementos principales de UML (modelo conceptual)

### Bloques de Construcción

### Elementos

☑ **Estructurales:** en su mayoría son la parte estática de un modelo, representa cosas que son conceptuales o materiales. Sus elementos son siete:

**1- Clase:** descripción de un conjunto de objetos que comparten los mismos atributos, operaciones, relaciones y semántica. Una clase implementa una o más interfaces.

**2- Interfaz:** es una colección de operaciones que especifican un servicio de una clase o componente. Describe el funcionamiento visible externamente de ese elemento.

**3- Colaboración:** define una interacción y es una sociedad de roles y otros elementos que colaboran para proporcionar un comportamiento cooperativo mayor que la suma de los comportamientos de sus elementos.

**4- Caso de Uso:** es una descripción de un conjunto de secuencias de acciones que un sistema ejecuta y produce un resultado de interés para el actor.

**5- Clase activa:** es una clase cuyos objetos tienen uno o más procesos o hilos de ejecución y por lo tanto pueden dar origen a actividades de control.

**6- Componente:** es una parte física y reemplazable de un sistema que conforma con un conjunto de interfaces y proporciona la implementación de dicho conjunto.

**7- Nodo:** es un elemento físico que existe en tiempo de ejecución y representa un recurso computacional.

☑ **De comportamiento:** representan la parte dinámica de un modelo. Representan comportamiento en el tiempo y espacio.

1- **Interacción:** es un comportamiento que comprende un conjunto de mensajes intercambiados entre un conjunto de objetos, dentro de un contexto para alcanzar un propósito específico.
2- **Máquina de estados:** es un comportamiento que especifica la secuencia de estados por las que pasa un objeto o una interacción durante su vida en respuesta a eventos, junto con sus reacciones a estos eventos.

☑ **De agrupación:** se utiliza para organizar los modelos. Posee un elemento que se denomina paquete, que sirve para organizar elementos en grupos y se lo representa como una carpeta.

☑ **De anotación:** son la parte explicativa de los modelos. Son comentarios que sirven para describir, clarificar y hacer observaciones sobre un elemento de un modelo. Su elemento principal es la nota, que es un símbolo para mostrar restricciones y comentarios junto a un elemento o una colección de elementos.

### Relaciones

☑ **Generalización:** es una relación de especialización/generalización en la cual los objetos del elemento especializado (hijo) pueden sustituir a los objetos del elemento general (padre). El hijo comparte la estructura y el comportamiento del padre. Una generalización es una relación de clases, no de objetos.

☑ **Asociación:** es una relación estructural que describe un conjunto de enlaces, los cuales son conexiones entre objetos. Una asociación puede refinarse con la navegabilidad, la multiplicidad y un rol.

☑ **Agregación:** es un tipo especial de asociación. Es una relación estructural entre el Todo y sus Partes.

☑ **Dependencia:** es una relación semántica entre dos elementos, en la cual un cambio en un elemento independiente puede afectar a la semántica del elemento dependiente.
☑ **Realización:** relaciona casos de uso con colaboraciones
Las relaciones como herencia, asociación y agregación son estáticas, las cuales se implementan por medio de punteros. La dependencia utiliza un puntero temporal, a diferencia de las anteriores.

## Diagramas

Un **diagrama** es un grafico que representa elementos y sus relaciones. Cada uno de los diagramas mostrara al sistema desde una perspectiva diferente.
☑ **Diagrama de clases:** muestra un conjunto de clases, interfaces, colaboraciones y sus relaciones. Modela la vista estática del sistema, es decir, modela estructura.
☑ **Diagrama de objetos:** muestra un conjunto de objetos y sus relaciones. Los DO representan instantáneas de instancias de los elementos encontrados en los DC.
☑ **Diagrama de casos de uso:** muestra un conjunto de casos de uso, actores y sus relaciones. Con este diagrama se construye el modelo de casos de uso del SI.
☑ **Diagrama de interacción:** muestra un conjunto de objetos, relaciones y mensajes. Este diagrama es usado cuando los flujos alternativos son pocos y hay muchos objetos de entidad involucrados. Dentro de este diagrama hay dos tipos:

1. **Diagrama de colaboración:** resalta la organización estructural de los objetos que envían y reciben mensajes. Muestran las conexiones y mensajes de comunicación entre objetos y son mejores para la comprensión de todos los efectos de un objeto dado.
2. **Diagrama de secuencia:** resalta la ordenación temporal de los mensajes. Muestran y hacen explicita la secuencia de eventos y son mejores que los diagramas de actividad para escenarios más complejos.

Estos dos diagramas son **isomorfos**, es decir, se puede tomar uno y transformarlo en el otro.

☑ **Diagrama de estados:** muestra una maquina de estados, que consta de estados, transiciones, eventos y actividades.
☑ **Diagrama de actividades:** muestra el flujo de actividades dentro de un sistema.
☑ **Diagrama de componentes:** muestra la organización y las dependencias entre un conjunto de componentes.
☑ **Diagrama de despliegue:** muestra la configuración de los nodos de procesamiento en tiempo de ejecución y los componentes que residen en ellos.
Los diagramas de clases, de objetos, de casos de uso, de componentes y de despliegue cubren la vista **estática** del sistema. Los diagramas de interacción, de estados, de actividades cubren la vista **dinámica** del sistema.
*Clasificación de los diagramas:*

| Diagrama | Estructura | Comportamiento | | Estático | Dinámico | | Lógico | Físico |
|---|---|---|---|---|---|---|---|---|
| **de clases** | ● | | | ● | | | ● | |
| **de objetos** | ● | | | ● | | | ● | |
| **de casos de uso** | | ● | | ● | | | ● | |
| **de interacción (colaboración y secuencia)** | | ● | | | ● | | ● | |
| **de estados** | | ● | | | ● | | ● | |
| **de actividades** | | ● | | | ● | | ● | |
| **de componentes** | ● | | | ● | | | | ● |
| **de despliegue** | ● | | | ● | | | | ● |

## Reglas para combinar esos boques

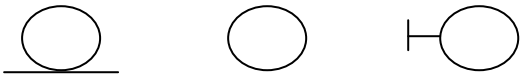
Un modelo bien formado es aquel que es autoconsistente y está en armonía con todos sus modelos relacionados.
UML establece reglas semánticas para:
☑ Los **nombres**: significa como nombrar a los elementos, relaciones y diagramas.
☑ La **visibilidad:** como se ven y usan esos elementos.

☑ La **integración**: como relacionar elementos con otros en forma apropiada.
Los modelos construidos de sistemas complejos tienden a evolucionar por esta razón es común que los desarrolladores construyan modelos **abreviados** (ocultando ciertos elementos), **incompletos** e **inconsistentes** (no se garantiza la integridad del modelo).

## Mecanismos comunes

Se usan para una mejor comprensión y claridad de los modelos. Se dividen en cuatro:
☑ **Especificaciones:** aclaraciones narrativas, que significa alguna cosa del diagrama. Ej. nota
☑ **Adornos:** diferenciación dentro de la sintaxis de los diagramas. Ej. Clases- Clases activas
☑ **Divisiones comunes:** diferenciar una clase de un objeto y una interfaz de una implementación.
☑ **Mecanismos de extensibilidad:** sirven para extender el lenguaje de manera controlada, pueden ser:

- **Estereotipos:** símbolos para diferenciar clases.



- **Valores etiquetados:** añade nueva información en la especificación de un elemento con una nota.
- **Restricciones:** reglas o limitaciones para aclarar el modelo con una nota

***El UML 2.0 agrega cuatro modelos más:**
☑ **Estructurales**

- **Estructura Compuesta:** cuando la estructura de la clase es compleja.
- **Paquetes:** en el 2.0 se formalizo este modelo, en el 1.4 ya estaba.
- **Colaboración:** representa la vista estática de los objetos, como se relacionan estructuralmente los objetos.

☑ **Comportamiento**

- **Interacción**

**Comunicación:** es el D. de colaboración en el 1.4.
**Descripción de interacción:** permite modelar escenarios de CU diferentes y relacionarlos. Usa distintos diagramas (colaboración, secuencia, DTE, actividad) y los relaciona, es decir, arma un circuito.
**Timming:** mezcla entre D. Secuencia y de Estado. Permite ver como objetos de distintas clases se restringen entre sí. Modelan restricciones.
**Secuencia**

# Modelado de Negocios

El **modelado de negocios** es una técnica para comprender los procesos de negocio que componen a la organización. Muestra como esta actúa con el ambiente.
Para construir el SI se debe saber del negocio:
☑ Los usuarios del sistema
☑ Los procesos de negocio
☑ Los tipos de documentos que se producen en cada proceso ya que este será el ambiente o contexto en donde se instalara el SI

### *Propósitos del Modelado de Negocios

☑ Entender la estructura y dinámica de la organización.
☑ Comprender los problemas e identificar potenciales mejoras.
☑ Asegurar que los clientes, usuarios finales y desarrolladores tengan un entendimiento común de la organización.
☑ Derivar los requerimientos del SI.

### *Casos del Modelado de Negocios

☑ Tener un diagrama de la organización
☑ Tener modelado el dominio de la organización (organización + información)
☑ Para subsistemas de un negocio
☑ Para aplicaciones que van a ser utilizadas por varias organizaciones (Modelado de Negocio genérico)
☑ Para un negocio nuevo
☑ Para renovar una organización (reingeniería)

### *Un Modelado de Negocio se realiza para ayudar a encontrar:

☑ Requerimientos para una aplicación especifica, que será construida desde cero.
☑ Requerimientos para mejorar una aplicación existente.
☑ Requerimientos para el sistema de información de una línea de negocios completamente nueva.

### *Vocabulario

Usuarios de Negocio = Actores de negocio
Procesos de Negocio = Casos de Uso de Negocio y Realizaciones de casos de uso
Roles = Trabajadores del negocio
Cosas = Entidades del negocio

El Modelado de Negocio esta soportado por el **Modelo de Casos de Uso** (vista externa) y por el **Modelo de Objetos del Negocio** (vista interna).

## Modelo de Casos de Uso

**Características**

☑ Se puede utilizar para modelar el contexto de un sistema (para especificar los actores y el significado de sus roles) o para modelar los requisitos de un sistema (para especificar el comportamiento deseado del sistema).
☑ Sirve para modelar el comportamiento de un sistema, subsistema o clase, cada uno de ellos muestra un conjunto de CU, actores y sus relaciones.
☑ Nos permite captar los requerimientos funcionales desde el punto de vista del usuario y facilita la construcción del sistema, porque el proceso de desarrollo está dirigido por CU.
☑ Sirven para probar sistemas ejecutables a través de la ingeniería directa y para comprender sistemas ejecutables a través de ingeniería inversa.
☑ Sirve para comunicar el comportamiento del sistema al cliente o usuario final.

**Elementos**

☑ **Actor:** representa un rol jugado en relación al negocio. Por ejemplo, cliente, proveedor, etc. Pueden ser HW, SW o personas.
Actor = Usuario + Rol
Sus **características** son:

- Cada actor humano expresa un rol, no una persona física.
- Cada actor modela algo fuera del negocio.
- Cada actor está involucrado al menos con un caso de uso.
- Cada actor tiene un nombre y una descripción.
- Cada actor no puede interactuar con el negocio de varias formas distintas.

☑ **Caso de uso:** procesos en los que se divide el negocio. Es la sumatoria de escenarios (curso normal y uno o más cursos alternativos).

**Escenario:** es una secuencia específica de acciones que ilustra un comportamiento. Un escenario es básicamente una instancia de un caso de uso.
Los casos de uso se clasifican en:
- **Caso de uso esencial:** representa actividades comercialmente importantes para el negocio.
- **Caso de uso soporte:** representa actividades menos importantes pero que deben realizarse.
- **Caso de uso de administración:** representa toma de decisiones.

☑ **Relaciones:** se usan para facilitar el entendimiento, el mantenimiento de los casos de uso y para poder reutilizar CU. Estas son las tres razones principales para estructurar el modelo de caso de uso del negocio.

Hay tres tipos de relaciones:

- **Inclusión:** se incluye en la base la funcionalidad del adicional. De lo contrario el base no podrá lograr su objetivo. Se da cuando un CU tiene una porción de comportamiento que es similar en más de un CU y no se quiere copiar la descripción de tal conducta. Si no se instancia el adicional el base fracasa.
- **Extensión:** el base puede cumplir su objetivo sin la funcionalidad del adicional. Cuando se dan determinadas condiciones se llama al adicional agregando funcionalidad al base. El objetivo del base no depende del adicional.
- **Generalización:** se da cuando hay funcionalidad en común.

La utilización de gran cantidad de estas relaciones produce acoplamiento, y el mismo debe ser bajo.

**Características de un buen Modelo de Caso de Uso del Negocio**
☑ Los casos de uso concuerdan con el negocio que ellos describen
☑ Se han detectado todos los casos de uso. Puestos juntos, los casos de uso ejecutan todas las actividades del negocio
☑ Cada actividad del negocio debería incluirse en al menos un caso de uso
☑ Debería haber un equilibrio entre el número de casos de uso y el tamaño de los casos de uso:
- Pocos casos de uso hacen el modelo fácil de entender
- Muchos casos de uso pueden hacer el modelo difícil de entender
- Casos de uso largos pueden ser complejos y difíciles de entender
- Casos de uso pequeños son fáciles de entender, sin embrago, se debe asegurar que el caso de uso describa un workflow completo que produce algo de valor para el cliente
☑ Cada caso de uso debe ser único
☑ La revisión del modelo de caso de uso debería dar una vista comprensiva de la organización

**Para construir un Diagrama de CU hay que:**
☑ **Identificar CU:** se identifican los actores y el límite del SI. Para poder deducir los CU hay que fijarse los distintos comportamientos del actor.
☑ **Describir los casos de uso:** usamos herramientas como las plantillas. Estructuramos los CU (establecemos las relaciones) y documentamos.

Los casos de uso se pueden agrupar de acuerdo al actor o por relaciones.

**Ingeniería directa:** es el proceso de transformar un modelo en código a través de una correspondencia con un lenguaje de implementación. A un diagrama de casos de uso se le puede aplicar ingeniería directa para generar pruebas del elemento al que se aplica.

**Ingeniería inversa:** es el proceso de transformar código en un modelo a través de una correspondencia a partir de un lenguaje de implementación. Conseguir un diagrama de casos de uso es impensable con la tecnología actual, porque hay perdida de información al pasar de la especificación de cómo se debe comportar un elemento al cómo se implementa. Lo que se puede hacer es estudiar un sistema existente y comprender su comportamiento de forma manual, para ponerlo después en forma de diagrama de casos de uso.

### *Modelo de Objeto de Negocio

Describe la realización de los casos de uso, es decir, como cada caso de uso es llevado a cabo por parte de un conjunto de trabajadores que utilizan un conjunto de entidades del negocio. El modelo define **como** debería relacionarse la gente que trabaja en el negocio con las cosas que maneja y usa (clases y objetos del negocio) para producir los resultados esperados. Para este modelo se utiliza el **diagrama de colaboración**, el cual muestra un conjunto de objetos y sus conexiones entre sí.

**Características de un buen Modelo de Objetos del Negocio**

☑ Puestos juntos, los trabajadores del negocio y las entidades realizan todas las actividades descriptas en los casos de uso del negocio

☑ El modelo de objetos del negocio da una buena y comprensiva imagen de la organización

**Elementos**

☑ **Trabajador de negocio:** representan roles que los empleados actuaran, se describen igual que las clases (nombre, atributos y operaciones).

☑ **Actor**

☑ **Entidad del negocio:** representan las cosas que los trabajadores manejaran, cuando ejecutan un caso de uso de negocio.

## El proceso unificado de desarrollo de software

Un **proceso** es un conjunto de pasos ordenados para alcanzar un objetivo, en nuestro caso, el objetivo será entregar un producto que satisfaga las necesidades del usuario de forma eficiente.

### Importancia o beneficios del Proceso

1. Proporciona una guía para ordenar las actividades de un equipo.
2. Dirige las tareas de cada desarrollador por separado del usuario de forma eficiente.
3. Especifica los artefactos a desarrollar.
4. Ofrece criterios para el control y la medición de los productos y actividades del proyecto.

### El Proceso Unificado en pocas palabras

El Proceso Unificado es un proceso de desarrollo de software. Un proceso de desarrollo de software es el conjunto de actividades necesarias para transformar los requisitos de un usuario en un sistema software.
Está basado en componentes, lo cual quiere decir que el sistema software en construcción está formado por **componentes** software interconectados a través de **interfaces** bien definidas.
El Proceso Unificado utiliza el **Lenguaje Unificado de Modelado** (UML) y se centra en tres ideas básicas: casos de uso, arquitectura y desarrollo incremental e iterativo.
El PUD proporciona un marco de trabajo genérico que debe adaptarse a la complejidad del software y a la naturaleza del proyecto.

## Características del PUD

**1) Dirigido por Casos de Uso:**
Un **caso de uso** es un fragmento de funcionalidad del sistema que proporciona al usuario un resultado importante. También es un conjunto de tareas o actividades que se realizan para cumplir un objetivo medible y concreto.
Los casos de uso representan los requisitos funcionales. Todos los casos de uso juntos constituyen el **Modelo de Casos de Uso** el cual describe la funcionalidad total del sistema. Se utiliza para conseguir un acuerdo con los usuarios y clientes sobre qué debería hacer el sistema para los usuarios.
**Dirigido por casos de uso** quiere decir que el proceso de desarrollo sigue un hilo, avanza a través de una serie de flujos de trabajo que parten de los casos de uso. Los casos de uso se especifican, se diseñan, y los casos de uso finales son la fuente a partir del cual los ingenieros de prueba construyen sus casos de prueba.
La captura de requisitos tiene dos objetivos: encontrar los verdaderos requisitos y representarlos de un modo adecuado para los usuarios, clientes y desarrolladores.
Normalmente, un sistema tiene muchos tipos de usuarios. Cada tipo de usuario se representa por un actor. Los actores utilizan el sistema interactuando con los casos de uso.
***¿Por qué casos de uso?**
☑ Proporcionan un medio sistemático e intuitivo de capturar requisitos funcionales centrándose en el valor añadido para el usuario.
☑ Dirigen todo el proceso de desarrollo debido a que la mayoría de las actividades como el análisis, diseño y prueba se llevan a cabo partiendo de los casos de uso.
☑ Ayudan a idear la arquitectura. Mediante la selección correcta del conjunto correcto de casos de uso podemos implementar un sistema con una arquitectura estable, que pueda utilizarse en muchos ciclos de desarrollo subsiguientes.
☑ Se utilizan como punto de partida para escribir el manual de usuario, ya que cada caso de uso describe una forma de utilizar el sistema.

**2) Centrado en la Arquitectura:** la arquitectura en un sistema software se describe mediante diferentes vistas del sistema en construcción. Es lo que le da soporte a los casos de uso.
El concepto de arquitectura software incluye los aspectos estáticos y dinámicos más significativos del sistema (aspectos críticos). La arquitectura del sistema surge de las necesidades de la empresa, como las perciben los usuarios y los inversores, y se refleja en los casos de uso. Sin embargo, también se ve influenciada por muchos otros factores, como la plataforma en la que tiene que funcionar el software, los bloques de construcción reutilizables de que se dispone, consideraciones de implementación, sistemas heredados y requisitos no funcionales. La arquitectura es una vista del diseño completo.
¿Cómo se relacionan los casos de uso y la arquitectura? Cada producto tiene una función como una forma. La función corresponde a los casos de uso y la forma a la arquitectura. Debe haber interacción entre los casos de uso y la arquitectura; los casos de uso deben encajar en la arquitectura cuando se llevan a cabo. Por otro lado, la arquitectura debe permitir el desarrollo de todos los casos de uso requeridos. Los casos de uso y la arquitectura deben evolucionar en paralelo.
Los arquitectos moldean el sistema para darle una forma. Para ello deben trabajar sobre los casos de uso claves del sistema.
Una arquitectura se necesita para:
☑ Comprender el sistema
☑ Organizar el desarrollo
☑ Fomentar la reutilización
☑ Hacer evolucionar el sistema

¿Qué es primero, la arquitectura o los casos de uso?
La mejor forma de resolver estos problemas es mediante una iteración. Primero, construimos una arquitectura tentativa básica a partir de una buena comprensión del área del dominio, pero sin considerar los casos de uso detallados. Entonces, escogemos un par de casos de uso y ampliamos la arquitectura adaptándola para que soporte esos casos de uso. Después, escogemos algunos casos de uso más y construimos una arquitectura mejor y así sucesivamente.

**3) Incremental-Iterativo:** las iteraciones hacen referencia a pasos en el flujo de trabajo, y los incrementos, al crecimiento del producto. Es decir, va creciendo de a pasos. No se larga todo junto, se va largando de a partes funcionales, cada ciclo produce un nueva versión del sistema, y cada versión es un producto preparado para su entrega.
Las iteraciones deben estar controladas, esto es, deben seleccionarse y ejecutarse de una forma planificada.
La arquitectura proporciona la estructura sobre la cual guiar las iteraciones, mientras que los casos de uso definen los objetivos y dirigen el trabajo de cada iteración.
Son muchos los beneficios de un proceso iterativo controlado, como ser:
☑ Reduce el costo del riesgo a los costos de un solo incremento.
☑ Reduce el riesgo de no sacar al mercado el producto en el calendario previsto.
☑ Acelera el ritmo del esfuerzo de desarrollo en su totalidad debido a que los desarrolladores trabajan de manera más eficiente para obtener resultados claros a corto plazo.
La eliminación de una de las tres ideas reduciría el valor del Proceso Unificado.
El producto terminado no solo debe ajustarse a las necesidades de los usuarios, sino que también a las de todos los interesados, es decir, toda la gente que trabajara con el producto. Incluye los requisitos, casos de uso, especificaciones no funcionales y casos de prueba.

## Ciclo de vida del PUD

El Proceso Unificado se repite a lo largo de una serie de ciclos que constituyen la vida de un sistema. Cada ciclo concluye con una versión del producto para los clientes. Cada ciclo consta de cuatro fases. Cada fase termina con un hito. Cada hito se determina por la disponibilidad de un conjunto de artefactos, es decir, ciertos modelos o documentos han sido desarrollados hasta alcanzar un estado predefinido. Los hitos tienen muchos objetivos y permiten a la dirección, y a los mismos desarrolladores, controlar el progreso del trabajo según pasa por esas cuatro fases.
☑ **Fase de inicio:** se desarrolla una descripción del producto final a partir de una buena idea y se presenta el análisis de negocio para el producto, es decir, analizar los casos de uso, la arquitectura del sistema y realizar un plan de proyecto.
☑ **Fase de elaboración:** se especifican en detalle la mayoría de los casos de uso del producto y se diseña la arquitectura del sistema. La arquitectura se expresa en forma de vistas de todos los modelos del sistema, los cuales representan juntos al sistema entero.
Al final de la fase de elaboración, el director del proyecto está en condiciones de planificar las actividades y estimar los recursos necesarios para terminar el proyecto.
☑ **Fase de construcción:** se crea el producto. En esta fase, la línea base de la arquitectura crece hasta convertirse en el sistema completo. Al final de esta fase, el producto contiene todos los casos de uso que la dirección y el cliente han acordado para el desarrollo de esta versión. Sin embargo, puede que no esté libre de defectos. Muchos de estos defectos se descubrirán y solucionaran durante la fase transición.
☑ **Fase de transición:** cubre el período durante el cual el producto se convierte en versión beta. En la versión beta un número reducido de usuarios con experiencia prueba el producto e informa de defectos y deficiencias. Los

desarrolladores corrigen los problemas e incorporan mejoras. La fase de transición conlleva actividades como la fabricación, formación del cliente y proporcionar ayuda y asistencia.
El equipo de mantenimiento suele dividir esos defectos en dos categorías: los que tienen suficiente impacto en la operación para justificar una versión incrementada (versión delta) y los que pueden corregirse en la siguiente versión normal.

## Las cuatro "P" en el desarrollo de software: Personas, Proyecto, Producto y Proceso

El resultado final de un proyecto software es un producto que toma forma durante su desarrollo gracias a la intervención de muchos tipos distintos de personas.

**1) Personas:** los principales autores de un proyecto software son los arquitectos, desarrolladores, ingenieros de prueba, y el personal de gestión que les da soporte, además de los usuarios, clientes, y otros interesados. Las personas financian el producto, lo planifican, lo desarrollan, lo gestionan, lo prueban, lo utilizan y se benefician de él. Por lo tanto, el proceso que guía este desarrollo debe orientarse a las personas que lo utilizan.
El modo en que se organiza y gestiona un proyecto software afecta a las personas implicadas en :
☑ **Viabilidad del proyecto:** la mayoría de la gente no disfruta trabajando en proyectos que parecen imposibles.
☑ **Gestión del riesgo:** las personas se sienten incomodas cuando sienten que los riesgos no han sido analizados y reducidos.
☑ **Estructura de los equipos:** la gente trabaja de manera más eficaz en grupos pequeños.
☑ **Planificación del proyecto:** cuando la gente cree que la planificación de un proyecto no es realista, la moral se hunde.
☑ **Facilidad de comprensión del proyecto:** a la gente le gusta saber o que está haciendo, quieren tener una comprensión global.
☑ **Sensación de cumplimiento:** un ritmo de trabajo rápido combinado con conclusiones frecuentes aumenta la sensación de progreso de la gente.
Las **herramientas** que emplean deben ser las adecuadas. Deben ayudar a los trabajadores a llevar a cabo las actividades y deben aislarles de la información que no les sea relevante.

**2) Proyecto:** elemento organizativo a través del cual se gestiona el desarrollo de software. El resultado de un proyecto es una versión de un producto.
A través de su ciclo de vida, un equipo de proyecto debe preocuparse de:
☑ **Una secuencia de cambio:** cada ciclo, cada fase y cada iteración modifican el sistema de ser una cosa a otra distinta.
☑ **Una serie de iteraciones:** dentro de cada fase de un ciclo, los trabajadores llevan a cabo las actividades de la fase a través de una serie de iteraciones. Cada iteración implementa un conjunto de casos de uso relacionados o atenúa algunos riesgos.
☑ **Un patrón organizativo:** la gente trabaja como diferentes trabajadores. La idea de "proceso" es proporcionar un patrón dentro del cual las personas en su papel de trabajadores ejecuten un proyecto.

**3) Producto:** artefactos que se crean durante la vida del proyecto, como los modelos.

**4) Proceso:** un proceso de ingeniería de software es una definición del conjunto completo de actividades necesarias para transformar los requisitos de usuario en un producto.
En el contexto del PUD, el término se refiere a los procesos de negocio claves en una empresa de desarrollo de software.
Un proceso es una plantilla para crear proyectos. Una instancia del proceso es un sinónimo de proyecto.

Un proceso cubre no solamente el primer ciclo de desarrollo (la primera versión) sino que también los ciclos posteriores más comunes.
Es importante que el proceso sea completamente consistente dentro de una organización. Esto permite intercambios de componentes, una transición eficaz de personas y directivos entre proyectos.
Los factores principales que influyen en cómo se diferenciara el proceso son:

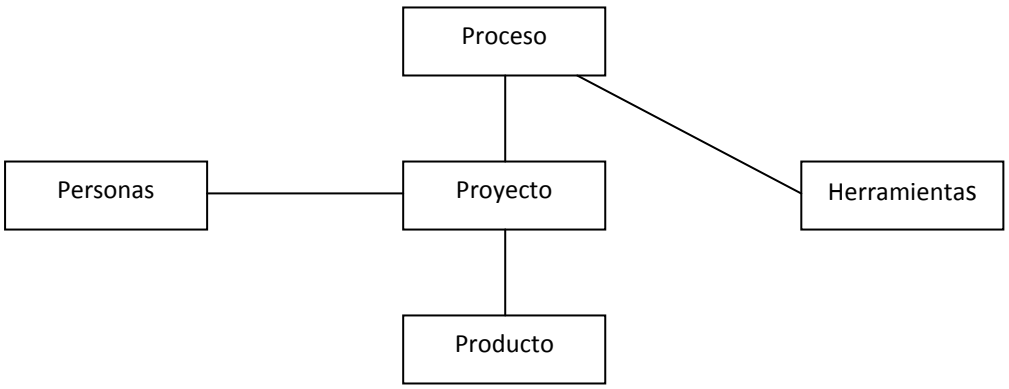
☑ **Factores organizativos:** la estructura organizativa, la cultura de la empresa, etc.
☑ **Factores del dominio:** procesos de negocio que se deben soportar, la comunidad de usuarios.
☑ **Factores del ciclo de vida:** el tiempo de salida al mercado, el tiempo esperado del software, la tecnología, etc.
☑ **Factores técnicos:** lenguaje de programación, herramientas de desarrollo, base de datos, etc.

**Herramientas:** software que se utiliza para automatizar las actividades definidas en el proceso.
Sirven también para:

- Gestionar grandes cantidades de información
- Guiarnos a lo largo de un camino de desarrollo concreto.
- Incrementar la productividad y la calidad.
- Reducir el tiempo de desarrollo.

El proceso especifica la funcionalidad de las herramientas, es decir, los casos de uso de las herramientas.
Las herramientas deben ser fáciles de usar.
El desarrollo del proceso y su soporte por herramientas debe tener lugar de manera simultánea. En cada versión del proceso debe haber también una versión de las herramientas.
Hay herramientas que soporte al ciclo de vida del software completo:

- **Gestión de requisitos:** se utiliza para almacenar, examinar, revisar, hacer el seguimiento y navegar por los diferentes requisitos de un proyecto software.
- **Modelado visual:** se utiliza para automatizar el uso de UML, es decir, para modelar y ensamblar una aplicación visualmente.
- **Herramientas de programación:** como editores, compiladores, depuradores, detectores de errores y analizadores de rendimiento.
- **Aseguramiento de la calidad:** se utiliza para probar aplicaciones y componentes, es decir, para registrar y ejecutar casos de prueba.




**¿Qué es un sistema software?**

Un sistema es todos los artefactos que se necesitan para representarlo en una forma comprensible por maquinas u hombres, para las maquinas, los trabajadores y los interesados. Las maquinas son las herramientas, compiladores u ordenadores destinatarios.

**¿Qué es un trabajador?**

Son personas que desarrollan el sistema y hacen los artefactos, son responsables de un conjunto de actividades. Puede ser un individuo o un conjunto de personas, como por ejemplo, un trabajador arquitecto puede ser un grupo de arquitectura.

Las **actividades** son trabajos significativos para una persona que actúa como trabajador.

**¿Qué es un artefacto?**

Es un término general para cualquier tipo de información creada, producida, cambiada o utilizada por los trabajadores en el desarrollo del sistema, por ejemplo, prototipos, modelos, diagramas, etc.

Hay dos tipos de artefactos:

☑ Artefactos de ingeniería, ej: modelo de CU

☑ Artefactos de gestión, por ejemplo, plan para la asignación de personas concretas a trabajadores y el diseño de las actividades de los trabajadores en el plan. Incluyen las especificaciones del entorno de desarrollo.

**¿Qué es un modelo?**

Es una abstracción del sistema, especificando el sistema modelado desde un cierto punto de vista y en un determinado nivel de abstracción. Un punto de vista, es por ejemplo, una vista de especificación o una vista de diseño del sistema.

El modelo es el tipo de artefacto más interesante utilizado en el PUD. Cada trabajador necesita una perspectiva diferente del sistema. Las perspectivas recogidas de todos los trabajadores se estructuran en unidades más grandes, es decir, modelos, de modo que un trabajador puede tomar una perspectiva concreta del conjunto de modelos.

Un modelo es una vista autocontenida en el sentido de que un usuario de un modelo no necesita para interpretarlo más información (de otros modelos).

La idea de ser autocontenido significa que los desarrolladores trataron de que hubiera una sola interpretación de lo que ocurrirá en el sistema. Además un modelo debe describir las interacciones entre el sistema y los que le rodean.

Un sistema no es solo la colección de sus modelos sino que contiene también las relaciones entre ellos. El hecho de que los elementos en dos modelos estén conectados no cambia lo que hacen en los modelos a los que pertenecen.

**¿Qué es calidad?**

Conjunto de propiedades y características que debe poseer un producto y servicio confiriéndole aptitud al satisfacer las necesidades del cliente, implícitas y explícitas.

Elementos de la calidad: Todas las características no comportamentales del soft podemos agruparlas en 3 términos:

1. Funcionamiento: que el sistema siempre funcione y que no cese su actividad.
2. Funcionalidad: que el sistema haga lo que tiene que hacer.
3. Usabilidad: que realice lo que tenga que hacer en forma adecuada y normal.

**¿Qué es un flujo de trabajo (workflows)?**

Es el modo en que describimos un proceso. Un **flujo de trabajo** es un conjunto de actividades en las que se divide el PUD y determina como fluye el trabajo de un trabajador a otro, para ello se identifica primero los trabajadores que participan y luego los artefactos que se necesitan crear.

En términos de UML, un flujo de trabajo es un **estereotipo** de colaboración, en el cual los trabajadores y los artefactos son los participantes.

Los trabajadores y artefactos que participan en un flujo de trabajo pueden participar también en otros flujos de trabajo.

**Flujos de trabajo fundamentales en los que se divide el proceso**

F.T. Requisitos……Modelo de Casos de Uso
F.T. Análisis……Modelo de Análisis
F.T. Diseño……Modelo de Diseño
Modelo de Despliegue
F.T. Implementación……Modelo de Implementación
F.T. Prueba……Modelo de Prueba

**Flujos de trabajo de soporte**

F.T. Administración de las configuraciones……Modelo de Prueba
F.T. Gestión de Proyecto……Proyecto
F.T. Entorno……Modelo de Entorno

**Modelo de Casos de Uso:** define todos los casos de uso y su relación con los usuarios.
**Modelo de Análisis:** sirve para refinar los casos de uso con más detalle.
**Modelo de Diseño:** define la estructura estática del sistema en la forma de subsistemas, clases e interfaces y los casos de uso reflejados como colaboraciones entre los subsistemas, clases e interfaces.
**Modelo de Implementación:** incluye componentes (código fuente) y la correspondencia de las clases con los componentes.
**Modelo de Despliegue:** define los nodos físicos (ordenadores) y la correspondencia de los componentes con esos nodos.
**Modelo de Prueba:** describe los casos de prueba que verifican los casos de uso.

## Calidad de Software

Es la concordancia de los requisitos funcionales con los estándares de desarrollo documentados, y las características implícitas que se espera de todo sistema desarrollado por un profesional.
Algunos factores que determinan la calidad son: fiabilidad, integridad, facilidad de uso, facilidad de mantenimiento, flexibilidad, portabilidad, reusabilidad.
La calidad no comienza a la hora de codificar, es una actividad de protección que debe aplicarse en cada paso del proceso, engloba:

1. Métodos y herramientas de análisis, diseño, codificación y pruebas.
2. Revisiones formales (formularios).
3. Estrategias de prueba.
4. Control de la documentación y de los cambios realizados.
5. Mecanismos de revisión.

La calidad está dada cuando cubre las exigencias del cliente.
Se busca reducir la complejidad del software. Estos son complejos porque existe:

- dificultad a la hora de entender el dominio
- problemas a la hora de gestionar el desarrollo
- la flexibilidad del SW
- las características de los sistemas discretos (no se sabe cómo va a reaccionar ante los posibles eventos)

# Flujo de Trabajo de Requisitos (WorkFlow de Requerimientos)

El propósito fundamental del flujo de trabajo de requisitos es guiar el proceso de desarrollo hacia el sistema correcto. Esto se consigue mediante una descripción de los requerimientos del sistema suficientemente buena como para que pueda llegarse a un acuerdo entre el cliente (incluyendo a los usuarios) y los desarrolladores sobre que debe y que no debe hacer el sistema.
El cliente debe ser capaz de leer y comprender el resultado de la captura de requisitos (se usa el lenguaje del cliente).
Los resultados del flujo de trabajo de los requisitos también ayudan al jefe del proyecto a planificar las iteraciones y las versiones del cliente.

| ENTRADA | PROCESO | SALIDA |
|---|---|---|
| • Modelo de Negocio<br>• Lista de requerimientos | • Encontrar actores y CU<br>• Priorizar los CU<br>• Detallar los CU<br>• Prototipar la interfaz<br>• Estructurar el modelo de CU | • Modelo de CU<br>  • Diagrama de CU<br>  • Descripción de CU<br>  • Descripción de trabajadores<br>  • Descripción y prototipo de interfaz de usuario<br>  • MODP<br>  • Lista de requerimientos adicionales<br>• Descripción de la arquitectura (Vista de CU) |

Hay una serie de pasos para el descubrimiento de requisitos que son:
**1. Enumerar los requerimientos candidatos** que aparecen de las ideas de todos los involucrados en el sistema. Esta lista de características se utiliza solo para la planificación del trabajo. Cada característica tiene un nombre corto y una breve explicación, para poder hablar de ella en la planificación del producto. Además tiene un conjunto de valores de planificación que son:
☑ Estado (propuesto, aprobado, incluido o validado)
☑ Costo estimado de implementación (en términos de tipos de recursos y horas-persona)
☑ Prioridad (critico, importante o secundario)
☑ Nivel de riesgo asociado a la implementación de la característica (critico, significativo u ordinario)
**2. Comprender el contexto del sistema**, para ello se utilizan dos formas de modelar, el **Modelado del Dominio** que describe los conceptos importantes del contexto como objetos del dominio (diagrama de clases) y el **Modelado de Negocio** que describe los procesos, formado por el modelo de CU del negocio y el modelo de objeto de colaboración.
**3. Encontrar requerimientos funcionales:** la técnica para identificar los requerimientos del sistema se basa en los CU. Estos CU capturan tanto los requerimientos funcionales como los no funcionales. Si los analistas pueden describir todos los CU que necesita el usuario entonces saben lo que debe hacer el sistema. Accesorio al CU los analistas deben especificar cuál será la apariencia de la interfaz cuando éstos se lleven a cabo.
Cada caso de uso representa una forma de usar el sistema.
**4. Encontrar requerimientos no funcionales:** especifican propiedades del sistema, como restricciones del entorno o de la implementación, rendimiento, dependencias de la plataforma, facilidad de mantenimiento, extensibilidad y fiabilidad.

| Trabajo a realizar | Artefactos resultantes |
|---|---|

| | |
|---|---|
| Enumerar requisitos candidatos | Lista de características |
| Comprender el contexto del sistema | Modelo del dominio o del negocio |
| Capturar los requisitos funcionales | Modelo de casos de uso |
| Capturar los requisitos no funcionales | Requisitos adicionales o casos de uso concretos |

## Ciclo de vida

- **Inicio:** los analistas identifican la mayoría de los CU para conocer el alcance del sistema y detallan los más críticos (solo el 10%).
- **Elaboración**: se describen la mayoría de los CU y con ello los desarrolladores pueden estimar el tamaño y el esfuerzo que requerirá.
- **Construcción:** se capturan requisitos restantes, son muy pocos.
- **Transición:** casi no hay captura a menos que los requisitos cambien.

## ¿Qué es un Modelo del Dominio (MODP)?

Un modelo del dominio captura los tipos más importantes de objetos en el contexto del sistema. Los objetos del dominio representan las cosas que existen o los eventos que suceden en el entorno en el que trabaja el sistema. Muchos de los objetos del dominio o clases pueden obtenerse de una especificación de requisitos o mediante la entrevista con los expertos del dominio. Las clases del dominio aparecen en tres formas típicas:

☑ Objetos del negocio que representan cosas que se manipulan en el negocio, como pedidos, cuentas, etc.

☑ Objetos del mundo real y conceptos de los que el sistema debe hacer un seguimiento.

☑ Sucesos que ocurrirán o han ocurrido.

El modelo del dominio se describe mediante el diagrama de clases. Estos diagramas muestran a los clientes, usuarios, revisores y a otros desarrolladores las clases del dominio y como se relacionan unas con otras mediante asociaciones.

## ¿Qué es un Modelo del Negocio?

Un modelo de casos de uso del negocio describe los procesos de negocio de una empresa en términos de casos de uso del negocio y actores del negocio que se corresponden con los procesos de negocio y los clientes. El modelo de casos de uso del negocio presenta un sistema desde la perspectiva de su uso, y esquematiza como proporcionar valor a sus usuarios.

El Modelo de Negocios se desarrolla, en dos pasos:

**1.** Los modeladores del negocio deben confeccionar un modelo de casos de uso del negocio que identifique los actores del negocio y los casos de uso del negocio que utilicen los actores.

**2.** Los modeladores deben desarrollar un modelo de objetos del, negocio compuesto por trabajadores, entidades del negocio, y unidades de trabajo que juntos realizan los casos de uso del negocio. Se asocian a estos diferentes objetos las reglas del negocio y otras normas impuestas por el negocio.

## *Requisitos adicionales

Son fundamentalmente requisitos no funcionales que no pueden asociarse a ningún caso de uso concreto.

☑ **Requisito de interfaz:** especifica la interfaz con un elemento externo con el cual debe interactuar el sistema o que establece restricciones.

☑ **Requisito físico:** especifica una característica física que debe poseer el sistema, como su forma, tamaño, etc.

☑ **Restricción de diseño:** limita el diseño de un sistema, como lo hacen las restricciones de extensibilidad y mantenibilidad, o las restricciones relativas a la reutilización de sistemas heredados o partes esenciales de los mismos.

☑ **Restricción de implementación:** especifica o limita la codificación o construcción del sistema.

## Tipos de Requerimientos

Los requerimientos son capacidades que debe tener y las que no el sistema.
**Requerimientos funcionales:** son las cosas que el SI va a hacer (¿Qué?). También declaran lo que el sistema no debe hacer. El incumplimiento de estos requisitos degrada al sistema.
**Requerimientos no funcionales:** son limitaciones o restricciones del SI (¿Cómo?). Una falla en un requerimiento no funcional del sistema lo inutiliza. Surgen de las necesidades del usuario y son difíciles de verificar.
**Requerimientos del dominio:** aparecen no por una necesidad propia del cliente sino por el ámbito en donde va a estar el SI.

## Propósitos de los requerimientos

**1.** Para que el desarrollador pueda explicarle lo que ha entendido que el cliente quiere que el sistema haga.
**2.** Indica a los diseñadores la funcionalidad y características del sistema.
**3.** Permite al equipo de prueba demostrar que el sistema hace lo que el cliente solicito.

La descripción de las funciones o servicios y restricciones forman los **Requerimientos**. El proceso por el cual se extraen, analizan, documentan y validan estos servicios y restricciones se llama **Ingeniería de Requerimientos**.

Para describir el flujo de trabajo de requisitos utilizaremos los **artefactos**, los **trabajadores** participantes y las **actividades** llevadas a cabo.

## Artefactos

☑ **Modelo de casos de uso:** permite que los desarrolladores de software y los clientes lleguen a un acuerdo sobre los requisitos, es decir, sobre las condiciones y posibilidades que debe cumplir el sistema. Sirve como acuerdo entre clientes y desarrolladores, y proporciona la entrada fundamental para el análisis, el diseño y las pruebas. Es un modelo del sistema que contiene actores, casos de uso y sus relaciones.
☑ **Actor:** los actores representan terceros fuera del sistema que colaboran con el sistema. Una vez que hemos identificado todos los actores del sistema, tenemos identificado el entorno externo al sistema. Una instancia de un actor es un usuario concreto que interactúa con el sistema.
☑ **Caso de uso:** cada forma en que los actores usan el sistema se representan con un caso de uso. Un caso de uso es una *especificación,* es un fragmento de funcionalidad que el sistema ofrece. Especifica el comportamiento de cosas dinámicas, en este caso, de instancias de los casos de uso. Una instancia de caso de uso es un escenario, es una secuencia de acciones que realiza un negocio para producir un resultado (éxito o fracaso) observable de valor para un actor individual del negocio. Un caso de uso define un conjunto de instancias de caso de uso de negocio.
☑ **Descripción de la arquitectura (vista del modelo de casos de uso):** la descripción de la arquitectura contiene una vista de la arquitectura del modelo de casos de uso, que representa los casos de uso significativos para la arquitectura. Esta vista incluye los casos de uso que describen alguna funcionalidad importante y crítica, o que impliquen algún requisito importante que deba desarrollarse pronto dentro del ciclo de vida del software.
☑ **Glosario:** sirve para definir términos comunes importantes que los analistas utilizan al describir el sistema. Un glosario es muy útil para alcanzar un consenso entre los desarrolladores relativo a la definición de los diversos conceptos y nociones, y para reducir el riesgo de confusiones. Es fácil de mantener y es más intuitivo para utilizarlo con terceras personas externas, como usuarios y clientes.
Un glosario tiende a estar más centrado en el sistema que se va a construir, en lugar de su contexto.
☑ **Prototipo de interfaz:** nos ayudan a comprender y especificar las interacciones entre actores humanos y el sistema durante la captura de requisitos. Nos ayuda también a comprender mejor los casos de uso.

## Trabajadores

☑ **Analista de sistemas:** es el responsable del conjunto de requisitos que están modelados en los casos de uso (requisitos funcionales y no funcionales). Delimita el sistema, encontrando los actores y los casos de uso y asegurando que el modelo de casos de uso es completo y consistente. Para la consistencia, el analista puede utilizar un glosario. También es el que dirige el modelado y el que coordina la captura de requisitos. No es responsable de cada CU en particular.

☑ **Especificador de casos de uso:** describe en forma detallada uno o más casos de uso.

☑ **Diseñador de interfaz de usuario:** dan forma visual a las interfaces de usuario.

☑ **Arquitecto:** describe la vista de la arquitectura del modelo de casos de uso. Esta vista es una entrada importante para planificar las iteraciones.

## Flujo de trabajo

Primero el analista ejecuta la actividad de encontrar CU y actores, asegurando que el modelo de CU captura todos los requisitos. Entonces el arquitecto identifica los CU relevantes arquitectónicamente, y así priorizar los CU y proporcionar aquellos que van a ser desarrollados en la iteración actual. Hecho esto, los especificadores describen los CU que se han priorizado y los diseñadores sugieren la interfaces de usuario para cada actor. Entonces el analista de sistemas reestructura el modelo de CU definiendo generalizaciones para hacerlo más comprensible.

☑ **Encontrar actores y CU** para:

- Delimitar el sistema de su entorno
- Saber quién y que (actores) interactúan con el sistema, y que funcionalidad (casos de uso) se espera del sistema
- Para capturar y definir un glosario de términos comunes esenciales para la creación de descripciones detalladas de las funcionalidades del sistema.

Es la actividad más decisiva para obtener adecuadamente los requisitos, y es responsabilidad del analista de sistemas.

Esta actividad consta de cuatro pasos:

- Encontrar los actores
- Encontrar los casos de uso
- Describir brevemente cada caso de uso
- Describir el modelo de casos de uso completo

☑ **Priorizar los CU:** cuales son los necesarios para el desarrollo en las primeras iteraciones y cuales se dejan para más adelante.

☑ **Detallar los CU:** es describir su flujo de sucesos en detalle, es decir, como comienza, termina e interactúan con los actores. Existen diferentes herramientas para realizar esto como pueden ser el modelo de CU y los diagramas de CU.

☑ **Prototipar la interfaz:** consiste en diseñar la interfaz que permita llevar a cabo los CU de manera eficiente.

☑ **Estructurar el modelo de CU:** el modelo de casos de uso se estructura para:

- Extraer descripciones de funcionalidad generales y compartidas que pueden ser utilizadas por descripciones mas especificas.
- Extraer descripciones de funcionalidad adicionales u opcionales que pueden extender descripciones mas especificas.

## Calidad en el WF de Requerimientos

Se asegura calidad con la validación entre:

- *diagrama de CU y las descripciones*: a través de la complejidad, la consistencia con el objetivo y lo descripto.
- *MODP y las descripciones de CU*: la estructura de la clase (atributos)
- *Prototipos de interfaz y las descripciones de CU*

- de los *requerimientos*:
  - Con prototipos de interfaz
  - Si se contemplan todos ellos en el modelo de CU
  - Con la trazabilidad de los mismos, ya que los modelos guardan una armonía entre ellos.

Además también la estamos asegurando cuando aplicamos patrones, a nivel conceptual (que nos permite construir MODP y diagramas de CU), a nivel abstracciones (evitando nombrar tecnología, trabajando con encapsulamiento, abstracción, modularidad y jerarquía) y a nivel de descripciones de CU.
También al aplicar la reusabilidad y al utilizar UML y el PUD.

# Flujo de trabajo de Análisis

Durante el análisis, analizamos los requisitos que se describieron en la captura de requerimientos refinándolos (incorporar detalle) y estructurándolos (organizar los requerimientos para que me facilite el trabajo) para poder comprenderlos mejor y hacer que sea fácil mantener y estructurar el sistema entero.

| ENTRADA | PROCESO | SALIDA |
|---|---|---|
| • Requisitos adicionales<br>• Modelo de CU<br>• Modelo de Negocio<br>• Descripción de la arquitectura (vista CU) | • Analizar la arquitectura<br>• Analizar los CU<br>• Analizar clases de análisis<br>• Analizar paquetes | • Modelo de Análisis<br>  • Diagrama de clases de análisis<br>  • Paquetes de análisis<br>  • Realizaciones de CU<br>• Descripción de la arquitectura (Vista de Análisis) |

## Ventajas del análisis

- Al estructurar el sistema hace que sea más fácil practicar las iteraciones del diseño y la implementación, así también asignar responsabilidades del grupo de desarrollo.
- Es más rápida la capacitación al incorporar nuevas personas.
- El Modelo de Análisis es el elemento unificador a la hora de que el negocio tenga que elegir entre alternativas del sistema.
- Se pueden comprender los sistemas heredados a partir del Modelo de Análisis.

La captura de requerimientos se lleva a cabo mediante los CU con un lenguaje que entiende el cliente, intuitivo, pero impreciso. Esto lleva a que quizá queden aspectos sin resolver que no nos hayamos dado cuenta por usar ese lenguaje del cliente. La captura de requisitos se hace de la siguiente forma, pero puede dejar también aspectos sin resolver:

1. *Los CU deben mantenerse tan independientes unos de otros como sea posible*, para ello, no hay que quedarse atrapado en detalles relativos a interferencias, concurrencia y conflictos entre CU cuando estos compiten por recursos compartidos, que son internos al sistema.
2. *Los CU deben describirse usando el lenguaje del cliente*. Hay que usar lenguaje natural en las descripciones de CU, pero de esta forma, perdemos poder expresivo y en la captura de requisitos pueden quedar sin tratar (o quedar apenas descriptos) muchos detalles que podríamos haber precisado con notaciones más formales.
3. *Debe estructurarse cada CU para que forme una especificación de funcionalidad completa e intuitiva.* Para ello hay q estructurar los CU de manera que reflejen intuitivamente los CU "reales" que el sistema proporcional. Por lo tanto, los aspectos relativos a las redundancias y a la abstracción entre los requisitos descriptos puede quedar sin resolver durante la captura de requisitos.

El análisis debe resolver estos temas sin tratar, analizando los requisitos con mayor profundidad, pero utilizando un lenguaje de los desarrolladores para describir los resultados (resuelve punto 2).

En el análisis podemos razonar más sobre los aspectos internos del sistema y resolver aspectos relativos a las interferencias entre CU y aspectos internos del sistema. A esto se le llama "refinar los requerimientos" (resuelve punto 1).

Además, en el análisis se pueden estructurar los requerimientos para facilitar su comprensión, preparación, modificación y mantenimiento (resuelve punto 3). Esta estructura (basada en clases de análisis y paquetes) es independiente a la estructura que se le dio a los requisitos (basada en CU)

| Modelo de casos de uso | Modelo de análisis |
|---|---|
| Descrito con el lenguaje del cliente | Descrito con el lenguaje del desarrollador. |
| Vista externa del sistema. | Vista interna del sistema. |
| Estructurado por los casos de uso; proporciona la estructura a la vista externa. | Estructurado por clases y paquetes estereotipados; proporciona la estructura a la vista interna. |
| Utilizado fundamentalmente como contrato entre el cliente y los desarrolladores sobre qué debería y qué no debería hacer el sistema. | Utilizado fundamentalmente por los desarrolladores para comprender cómo deberá darse forma al sistema, es decir, cómo debería ser diseñado e implementado. |
| Puede contener redundancias, inconsistencias, etc., entre requisitos. | No debería contener redundancias, inconsistencias, etc., entre requisitos. |
| Captura la funcionalidad del sistema, incluida la funcionalidad significativa para la arquitectura. | Esboza cómo llevar a cabo la funcionalidad dentro del sistema, incluida la funcionalidad significativa para la arquitectura; sirve como una primera aproximación al diseño. |
| Define casos de uso que se analizarán con más profundidad en el modelo análisis. | Define realizaciones de casos de uso, y cada una de ellas representa el análisis de un caso de uso del modelo de casos de uso |

## Ciclo de vida

- **Inicio:** ayuda a planificar las iteraciones.
- **Elaboración**: en las iteraciones iniciales de la elaboración se centra el análisis para lograr una arquitectura robusta y estable.
- **Construcción:** -
- **Transición:** corregir errores.

## *Importancia del análisis

Algunas empresas no realizan análisis pero si hacen las actividades del mismo en los WF de requerimientos y de diseño.

Incorporarlo en el primero involucra más formalidad en el lenguaje, lo cual es muy probable que el cliente no lo entienda.

Incorporarlo en el segundo, sucede que al momento d tomar decisiones debo tener en claro lo que hace el sistema y puedo tomar caminos erróneos.

Hacer el análisis es bueno porque puedo obtener una comprensión clara y precisa del sistema necesario para el diseño.

## *Análisis según el tipo de proyecto

- Aquellos en los que se utiliza el modelo de análisis en el diseño y la implementación, y se lo mantiene actualizado al mismo
- Aquellos en los que se hace el modelo de análisis hasta que termina la fase de elaboración y después no se lo actualiza.
- Aquellos en los que no se hace análisis debido a que el dominio es muy simple.

Al momento de elegir entre estas alternativas influye el factor costo-beneficio de mantener el análisis durante todo el proyecto o dejar de actualizarlo.

## Artefactos

☑ **Modelo de análisis:** El modelo de análisis se representa mediante un sistema de análisis que denota el paquete de más alto nivel del modelo. Dentro del modelo de análisis, los CU se describen mediante clases de análisis y sus objetos. Esto se representa mediante colaboraciones dentro del modelo de análisis que llamamos *realizaciones de CU de análisis*.
☑ **Clase de análisis:** representa una abstracción de una o varias clases y/o subsistemas del diseño del sistema. La clase de análisis posee las siguientes características:
**1.** Se centra en el tratamiento de los requisitos funcionales y pospone los no funcionales.
**2.** Esto hace que la clase de análisis sea más evidente en el contexto del dominio del problema.
**3.** Raramente define u ofrece un interfaz en términos de operaciones y sus signaturas. En cambio, su comportamiento se define mediante responsabilidades en un nivel más alto y menos formal. Una responsabilidad es una descripción textual de un conjunto cohesivo del comportamiento de la clase.
**4.** Define atributos, aunque los atributos también son de un nivel bastante alto (no se definen tipos de atributos, etc).
**5.** Participa en relaciones.
**6.** Las clases de análisis encajan en 3 estereotipos: de interfaz, de control o de entidad. Cada estereotipo implica una semántica específica, lo cual contribuye a la robustez del sistema.

*Clases de interfaz:* se usan para modelar la interacción entre el sistema y sus actores. Esta interacción implica recibir y presentar información y peticiones desde y hacia los usuarios y los sistemas externos. Estas clases deberían mantenerse en un nivel bastante alto y conceptual, es decir, que es suficiente con que describan lo que se obtiene con la interacción. La clase de interfaz debe relacionarse al menos con un actor, y viceversa. Representan a menudo abstracciones de de ventanas, formularios, paneles.

*Clase de entidad:* se usan para modelar información que posee una vida larga y que a menudo es persistente. Estas clases modelan la información y el comportamiento asociado de algún fenómeno o concepto, como una persona, objeto del mundo real o un suceso del mundo real. Por lo general, derivan directamente de una clase del dominio. La diferencia entre una clase de entidad y una clase del dominio es que la primera representa un objeto manejado por el sistema, mientras que las segundas representan objetos presentes en el negocio en general. La clase entidad refleja la información de una forma que beneficia a los desarrolladores al diseñar e implementar el sistema,

incluyendo su soporte de persistencia. La clase entidad suele mostrar una estructura de datos lógica y contribuyen a entender de qué información depende el sistema.

*Clases de control:* representan coordinación, secuencia, transacciones y control de otros objetos y se usan para encapsular el control de un CU en concreto. También se usan para representar derivaciones y cálculos complejos, como la lógica del negocio, pero no pueden asociarse como ninguna información concreta, ni de larga duración almacenada por el sistema. Los aspectos dinámicos del sistema se modelan con clases de control, debido a que ellas manejan y coordinan las acciones y los flujos de control principales, y delegan trabajo a otros objetos (de interfaz y de entidad).

☑ **Realización de CU de análisis:** es una colaboración dentro del modelo de análisis, que describe como se lleva a cabo y se ejecuta un CU determinado en términos de clases de análisis y de sus objetos del análisis en interacción. Una realización de CU posee una descripción textual del flujo de sucesos, diagramas de clases que muestran sus clases de análisis participantes y diagramas de interacción que muestran la realización de un flujo o escenario particular del CU en término de interacciones de objetos del análisis. Se centra en los requisitos funcionales y puede posponer el tratamiento de los requisitos no funcionales a las actividades de diseño e implementación.

1. *Diagramas de clases:* una clase de análisis y sus objetos normalmente participan en varias realizaciones de CU, y algunas de las responsabilidades, atributos y asociaciones de una clase concreta suelen ser solo relevantes para una única realización de CU, por lo tanto es importante coordinar todos los requisitos sobre una clase y sus objetos que pueden tener diferentes CU.

2. *Diagramas de interacción:* la secuencia de acciones en un CU comienza cuando un actor invoca el CU mediante el envío de algún tipo de mensaje al sistema, un objeto interfaz lo recibirá y lo enviara a su vez a otro objeto, y de esta forma interactuarán para llevar a cabo el CU. En el análisis preferimos mostrar esto con diagramas de colaboración, ya que nuestro objetivo fundamental es identificar requisitos y responsabilidades sobre los objetos, y no identificar secuencias de interacción detalladas y ordenadas cronológicamente.
   Ciclos de vida de los objetos:
   - *De interfaz:* a menudo se crean y finalizan dentro de una realización de CU.
   - *De entidad:* suele tener una vida larga y participa en varias realizaciones de CU.
   - *De control:* están asociados con un CU concreto, lo que implica que deben destruirse al finalizar el CU, pero hay excepciones, cuando un objeto participa en varias realizaciones de CU.

3. *Flujo de sucesos del análisis:* Los diagramas (especialmente los de colaboración) de una realización de CU pueden ser difíciles de leer, por eso es útil poner un texto adicional que los explique. Este texto debería escribirse en términos de objetos y no debería mencionar atributos y responsabilidades, ya que estos cambian a menudo.

4. *Requisitos especiales:* son descripciones textuales que recogen todos los requisitos no funcionales sobre una realización de CU.

☑ **Paquete de análisis:** proporcionan un medio para organizar los artefactos del modelo de análisis en piezas manejables. Puede constar de clases de análisis, realizaciones de CU y de otros paquetes de análisis (recursivamente). Los paquetes de análisis deberían ser cohesivos y débilmente acoplados. Tienen las siguientes características:
1. Pueden representar una separación de intereses de análisis.
2. Deberían crearse basándonos en los requisitos funcionales y en el dominio del problema, y deberían ser reconocibles por las personas con conocimientos del dominio.
3. Probablemente se convertirán en subsistemas en el diseño.

*Paquetes de servicio:* se utilizan en un nivel más bajo de la jerarquía de paquetes de análisis para estructurar al sistema de acuerdo a los servicios que proporciona. Los paquetes de servicio son indivisibles, u obtengo todas las clases que lo conforman o ninguna, están atravesados por varios CU (un CU puede requerir varios paquetes de

servicio), tienen bajo acoplamiento y alta cohesión, pueden ser mutuamente excluyentes o representar distintas variantes del mismo servicio. Es un instrumento fundamental para la reutilización durante el análisis.

☑ **Descripción de la arquitectura:** contiene una vista de la arquitectura del modelo de análisis que muestra sus artefactos significativos para la arquitectura. Los siguientes artefactos del modelo de análisis, normalmente se consideran significativos para la arquitectura:
1. La descomposición del modelo de análisis en paquetes de análisis y sus dependencias.
2. Las clases fundamentales del análisis, como las clases de entidad que encapsulan un fenómeno importante del dominio del problema, las clases de interfaz que encapsulan interfaces de comunicación importantes y mecanismos de interfaz de usuario, las clases de control que encapsulan importantes secuencias con una amplia cobertura y las clases del análisis que son generales, centrales y que tienen muchas relaciones con otras clases del análisis.
3. Realizaciones de CU que describen cierta funcionalidad importante y crítica.

## Trabajadores

☑ **Arquitecto:** es responsable de la integridad del modelo de análisis, garantizando que este sea correcto, consistente y legible como un todo. También es responsable de la arquitectura del modelo de análisis, es decir, de la existencia de sus partes significativas para la arquitectura tal y como se muestran en la vista de la arquitectura del modelo.
☑ **Ingeniero de CU:** es responsable de la integridad de una o más realizaciones de CU, garantizando que cumplen con los requisitos que recaen sobre ellos. Una realización de CU debe llevar a cabo correctamente el comportamiento de su correspondiente CU del modelo de CU, y solo ese comportamiento. Esto incluye garantizar que todas las descripciones textuales y los diagramas que describen la realización del CU son legibles y se ajustan a su objetivo.
☑ **Ingeniero de Componentes:** define y mantiene las responsabilidades, atributos, relaciones y requisitos especiales de una o varias clases del análisis, asegurándose de que cada clase cumple los requisitos que se esperan de ella. También mantiene integridad de uno o varios paquetes del análisis. Esto incluye garantizar que sus contenidos son correctos y que sus dependencias con otros paquetes del análisis son correctas y mínimas.
Si existe una traza directa entre los paquetes de análisis y los subsistemas de diseño correspondientes, éste también debería encargarse de esos subsistemas.

## Flujo de Trabajo

Los arquitectos comienzan la creación del modelo de análisis, identificando los paquetes de análisis principales, las clases de entidad evidentes y los requisitos comunes. Después los ingenieros de CU realizan cada caso de uso en términos de las clases de análisis participantes exponiendo los requisitos de comportamiento de cada clase. Los ingenieros de componentes especifican luego estos requisitos y los integran dentro de cada clase creando responsabilidades, atributos y relaciones consistentes para cada clase.
Continuamente durante el análisis, el arquitecto identifica nuevos paquetes del análisis, clases y requisitos comunes a medida que el modelo de análisis evoluciona, y los ingenieros de componentes responsables de los paquetes de análisis concretos continuamente los refinan y mantienen.

### Actividades del análisis:

☑ **Análisis de la arquitectura:** su propósito es esbozar el modelo de análisis y la arquitectura mediante la identificación de paquetes del análisis, clases de análisis evidentes y requisitos especiales comunes.
1. *Identificación de paquetes del análisis:* los paquetes del análisis proporcionan un medio para organizar el modelo de análisis en piezas más pequeñas y más manejables. Se puede llevar a cabo las siguientes asignaciones de casos de uso a un paquete en concreto:
- Los casos de uso requeridos para dar soporte a un determinado proceso de negocio.
- Los casos de uso requeridos para dar soporte a un determinado actor del sistema.
- Los casos de uso que están relacionados mediante relaciones de generalización y de extensión.

*Gestión de los aspectos comunes entre paquetes del análisis:* con frecuencia se da el caso de encontrar aspectos comunes entre los paquetes identificados. Por ejemplo, cuando dos o más paquetes del análisis necesitan compartir la misma clase del análisis. Una manera apropiada de tratar es extraer la clase compartida, colocarla dentro de su

propio paquete, o simplemente fuera de cualquier otro paquete y hacer que los otros paquetes sean dependientes de ese paquete o clase más general.
*Identificación de paquetes de servicio:* se suele hacer cuando el trabajo del análisis esta avanzado. Lo que se hace es que todas las clases del análisis dentro del mismo paquete de servicio contribuyan al mismo servicio.
*Definición de dependencias entre paquetes del análisis:* deberían definirse dependencia entre los paquetes si sus contenidos están relacionados. El objetivo es conseguir paquetes débilmente acoplados y altamente cohesivos.
2. *Identificación de clases de entidad obvias*: suele ser adecuado preparar una propuesta preliminar de las clases de entidad más importantes basada en las clases del dominio que se identificaron durante la captura de requisitos (10 o 20) pero la mayoría de las clases se identificaran al crear las realizaciones de los CU.
3. *Identificación de requisitos especiales comunes*: un requisito especial es un requisito que aparece durante el análisis y es importante anotar de forma que pueda ser tratado adecuadamente en las siguientes actividades de diseño e implementación. Ej: restricciones sobre la persistencia, la seguridad.

☑ **Analizar un CU:** analizamos un CU para:
- Identificar las clases del análisis cuyos objetos son necesarios para llevar a cabo el flujo de sucesos del CU.
- Distribuir el comportamiento del CU entre los objetos del análisis que interactúan
- Capturar requisitos especiales sobre la realización de los CU

1. *Identificación de clases de análisis:* identificamos las clases de control, entidad e interfaz necesarias para realizar los CU y esbozamos sus nombres, responsabilidades, atributos y relaciones.
   Formas de identificar las clases del análisis:
   - Identificar clases de entidad mediante el estudio en detalle de la descripción del caso de uso y de cualquier modelo del dominio que se tenga, y después considerar qué información debe utilizarse y manipularse en la realización de caso de uso.
   - Identificar una clase de interfaz central para cada actor humano u sistema externo, y dejar que esta clase represente la ventana principal del interfaz de usuario con el cual interactúa el actor o la de comunicación.
   - Identificar una clase de control responsable del tratamiento del control y de la coordinación de la realización de caso de uso, y después refinar esta clase de control de acuerdo a los requisitos del caso de uso.

2. *Descripción de interacciones entre objetos del análisis:* mediante diagramas de colaboración que contienen instancias de actores y objetos de análisis describimos cómo interactúan los objetos del análisis entre sí. Se hace un diagrama de colaboración por cada escenario deseado.
   Diagrama de Comunicación (de Colaboración)
   Tiene como propósito mostrar la organización estructural de los objetos y los mensajes entre ellos.
   Elementos: objetos o roles, enlaces(interacción) y los mensajes.

3. *Captura de requisitos especiales*

☑ **Analizar una clase:** los objetivos de analizar una clase son:
- Identificar y mantener las responsabilidades de una clase del análisis, basadas en su papel en las realizaciones de CU.
- Identificar y mantener los atributos y relaciones de la clase del análisis
- Capturar requisitos especiales sobre la realización de la clase del análisis.

1. *Identificar responsabilidades:* las responsabilidades de una clase pueden recopilarse combinando todos los roles que cumple en diferentes realizaciones de CU.
2. *Identificación de atributos:* un atributo especifica una propiedad de una clase del análisis y normalmente es necesaria para las responsabilidades de su clase.
3. *Identificación de asociaciones y agregaciones:* los objetos del análisis interactúan unos con otros mediante enlaces en los diagramas de colaboración, estos enlaces suelen ser instancias de asociaciones entre sus correspondientes clases. Hay que determinar que asociaciones son necesarias y su multiplicidad, minimizando el número de relaciones entre clases.
4. *Identificación de generalizaciones:* las generalizaciones deberían usarse durante el análisis para extraer comportamiento compartido y común entre varias clases de análisis diferentes.

5. *Captura de requisitos especiales:* recogemos todos los requisitos de una clase del análisis que se identificaron en el análisis, pero que deberían tratarse en el diseño e implementación (es decir, requisitos no funcionales)

☑ **Analizar un paquete:** los objetivos de analizar un paquete son:
1. Garantizar que el paquete del análisis es tan independiente de otros paquetes como sea posible
2. Garantizar que el paquete del análisis cumple su objetivo de realizar algunas clases del dominio o CU
3. Describir las dependencias de forma que pueda estimularse el efecto de los cambios futuros.

### Calidad en WF de Análisis

Gracias que el PUD es dirigido por CU, podemos garantizar la calidad a través de la trazabilidad entre los distintos modelos. Otro aspecto que la asegura es el que los mensajes (en el diag. de colaboración) tengan el mismo nombre de la operación a la que invocan y la aplicación de patrones.

A nivel de paquetes se asegura la calidad cuando vemos que son altamente cohesivos y débilmente acoplados.

## Workflow de Diseño

Modelamos el sistema y encontramos su forma para que soporte todos los requisitos que se le suponen.

### Propósitos

- Comprender más profundamente todos los aspectos referidos a los requerimientos no funcionales y a los que se refieren al ambiente de implementación.
- Refinar los requerimientos a través de los subsistemas, clases de diseño e interfaz.
- Preparar en forma conceptual cómo va la implementarse el sistema.
- Descomponer el sistema en piezas manejables que permitan realizar el trabajo.

| ENTRADA | PROCESO | SALIDA |
|---|---|---|
| • Requisitos adicionales<br>• Modelo de Análisis<br>• Descripción de la arquitectura (vista Análisis)<br>• Diagrama de clases del análisis (MODSolución)<br>• Especificación del ambiente de implementación | • Diseñar la arquitectura<br>• Diseñar los CU<br>• Diseñar clases de diseño<br>• Diseñar subsistemas | • Modelo de Diseño<br>• Diagrama de clases de diseño<br>• Subsistemas<br>• Interfaces<br>• Realizaciones de CU de diseño:<br>  ○ Descripción de flujos de sucesos<br>  ○ Diag. de interacción (secuencia)<br>  ○ Diag. de clases (parcial)<br>• Descripción de la arquitectura (Vista de Diseño)<br>• Modelo de despliegue |

| Modelo de Análisis | Modelo de Diseño |
|---|---|
| Modelo Conceptual | Modelo Físico |
| Modelo genérico respecto al diseño | No genérico, específico para una implementación |
| 3 estereotipos: Control, Entidad e Interfaz | Cualquier cantidad de estereotipos |
| Menos formal | Más formal |
| Menos caro de desarrollar | Más caro de desarrollar |
| Menos capas | Más capas |
| Dinámico (no muy centrado en la secuencia) | Dinámico (muy centrado en la secuencia) |
| Bosquejo del diseño del sistema | Manifiesto del diseño del sistema |
| Creado como "trabajo de a pie" | Creado como "programación visual". Realizado según la ingeniería de ida y vuelta. |
| Puede no estar mantenido | Debe ser mantenido |
| Define una estructura | Da forma al sistema mientras preserva la estructura |

## Ciclo de vida

El diseño es el centro de atención al final de la fase de elaboración y el comienzo de la de construcción. Esto contribuye a una arquitectura estable y sólida, y a crear un plano del modelo de implementación.

## Niveles de diseño

- **Sistema:** se emplean diagramas de despliegue, clases y de objetos.
- **CU:** se emplean diagramas de interacción, de componentes y de estados
- **Clase:** se refinan los atributos (visibilidad, tipo y tamaño) y los métodos (con el diagrama de secuencia, de estado y de actividad)

## Artefactos

☑ **Modelo de diseño**: El modelo de diseño es un modelo de objetos que describe la realización física de los casos de uso centrándose en cómo los requisitos funcionales y no funcionales, junto con otras restricciones relacionadas con el entorno de implementación, tienen impacto en el sistema a considerar.
El modelo de diseño se representa por un sistema de diseño compuesto por subsistemas interrelacionados mediante interfaces, se define una jerarquía de subsistemas para organizar el modelo en porciones más manejables.
Los subsistemas de diseño y clases del diseño representan **abstracciones** del subsistema y componentes de la implementación del sistema y poseen una correspondencia directa entre el diseño y la implementación.

☑ **Clase de diseño:** Una clase de diseño es una abstracción sin costuras de una clase o construcción similar en la implementación del sistema:
- El lenguaje utilizado para especificar una clase del diseño es el mismo que el lenguaje de programación.
- A diferencia de las clases de análisis, en las clases de diseño se especifican la visibilidad, tipo de datos y largo de los atributos
- Las relaciones de aquellas clases del diseño implicadas con otras clases, a menudo tienen un significado directo cuando la clase es implementada.
- Los métodos de una clase del diseño tienen correspondencia directa con el correspondiente método en la implementación de las clases. Si los métodos se especifican en el diseño, se suelen especificar en lenguaje natural, o en pseudocódigo, y por eso pueden ser utilizados como comentarios en las implementaciones del método.
- Una clase de diseño suele aparecer como un estereotipo que se corresponde con una construcción en el lenguaje de programación.
- Una clase de diseño puede realizar y proporcionar interfaces.
- Una clase de diseño puede activarse, aunque no están normalmente activas.

☑ **Realización de caso de uso-diseño:** Una realización de caso de uso-diseño es una colaboración en el modelo de diseño que describe cómo se realiza un caso de uso específico, y cómo se ejecuta, en términos de clases de diseño y sus objetivos. Una realización de caso de uso-diseño proporciona una traza directa a una realización de caso de uso-análisis.
Una realización de caso de uso-diseño proporciona una realización física de una realización de caso de uso-análisis para la que es trazada, y también gestiona muchos requisitos no funcionales (es decir, requisitos especiales) capturados de la realización de caso de uso-análisis. Podemos posponer algunos requisitos hasta las actividades de implementación.
Una realización de caso de uso-diseño compuesta por:

- Flujo de sucesos (explicación y descripción textual).
- Diagramas de clases (parcial)
- Diagramas de interacción (específicamente Diagrama de secuencia: tiene como propósito mostrar cómo se da el paso de mensajes entre una sociedad de objetos a lo largo del tiempo).
- Subsistemas asociados a los objetos de la sociedad con sus interfaces y relaciones.
- Requerimientos de implementación.

Elementos de un diagrama de secuencia:

- Objetos: actor, gestor, múltiple instancia, objeto concreto
- Mensajes: mensaje(parámetro:tipoParametro):retorno:tipoRetorno
- Línea de vida
- Notas

☑ **Subsistema de diseño:** Los subsistemas de diseño son una forma de organizar los artefactos del modelo de diseño en piezas más manejables. Un subsistema puede constar de clases del diseño, realizaciones de caso de uso, interfaces y otros subsistemas, un subsistema puede proporcionar interfaces que representan la funcionalidad que exportan en términos de operaciones.
Un subsistema debería ser cohesivo; es decir, sus contenidos deberían encontrarse fuertemente asociados. Además, los subsistemas deberían ser débilmente acoplados; esto es, sus dependencias entre unos y otros, o entre sus interfaces, deberían ser mínimas.
Podemos encontrar subsistemas de servicio, que se basan en los paquetes de servicio del modelo de análisis, pero tratan más aspectos que sus correspondientes paquetes de servicio.
☑ **Interfaz:** Las interfaces se utilizan para especificar las operaciones que proporcionan las clases y los subsistemas del diseño.
Una clase del diseño que proporcione una interfaz debe proporcionar también métodos que realicen las operaciones de la interfaz. Las interfaces constituyen una forma de separar la especificación de la funcionalidad en términos de operaciones de sus implementaciones en términos de métodos. La mayoría de las interfaces se consideran relevantes para la arquitectura, debido a que definen las interacciones entre subsistemas.
☑ **Descripción de la arquitectura (vista del modelo de diseño):** La descripción de la arquitectura contiene una vista de la arquitectura del modelo de diseño, que muestra sus artefactos relevantes para la arquitectura.
Suelen considerarse significativos para la arquitectura los siguientes artefactos del modelo de diseño:
- La descomposición del modelo de diseño en subsistemas, sus interfaces, y las dependencias entre ellos. Esta descomposición es muy significativa para la arquitectura en general, debido a que los subsistemas y sus interfaces constituyen la estructura fundamental del sistema.
- Clases del diseño fundamentales, como clases que poseen una traza con clases del análisis significativas, clases activas, y clases del diseño que sean generales y centrales, que representan mecanismos de diseño genéricos, y que tengan muchas relaciones con otras clases del diseño. Basta con considerar significativas para la arquitectura a las clases abstractas, y no a sus subclases, a menos que las subclases representen algún comportamiento interesante y significativo para la arquitectura diferente al de la clase abstracta.
- Realizaciones de caso de uso-diseño que describan alguna funcionalidad importante y crítica que debe desarrollarse pronto dentro del ciclo de vida del software, que impliquen muchas clases del diseño y por tanto tengan una cobertura amplia, posiblemente a lo largo de varios subsistemas, o que impliquen clases del diseño fundamentales.

☑ **Modelo de despliegue:** El modelo de despliegue es un modelo de objetos que describe la distribución física del sistema en términos de cómo se distribuye la funcionalidad entre los nodos de cómputo.
Podemos observar lo siguiente sobre el modelo de despliegue:
- Cada nodo representa un recurso de cómputo, normalmente un procesador o un dispositivo hardware similar.
- Los nodos poseen relaciones que representan medios de comunicación entre ellos, tales como Internet, intranet, bus, y similares.
- El modelo de despliegue puede describir diferentes configuraciones de red, incluidas las configuraciones para pruebas y para simulación.
- La funcionalidad (los procesos) de un nodo se definen por los componentes que se distribuyen sobre ese nodo.
- El modelo de despliegue en sí mismo representa una correspondencia entre la arquitectura software y la arquitectura del sistema (hardware).

☑ **Descripción de la arquitectura (vista del modelo de despliegue):** La descripción de la arquitectura contiene una vista de la arquitectura del modelo de despliegue, que muestra sus artefactos relevantes para la arquitectura.

## Trabajadores

☑ **Arquitecto:** En el diseño el arquitecto es el responsable de la arquitectura y de la integridad de los modelos de diseño y de despliegue, garantizando que los modelos son correctos, consistentes y legibles en su totalidad.
Los modelos son correctos cuando realizan la funcionalidad, y sólo la funcionalidad, descrita en el modelo de casos de uso, en los requisitos adicionales, y en el modelo de análisis.
El arquitecto no es responsable del desarrollo y mantenimiento continuos de los distintos artefactos del modelo de diseño. Estos se encuentran bajo la responsabilidad de los correspondientes ingenieros de casos de uso y de componentes.

☑ **Ingeniero de casos de uso:** El ingeniero de casos de uso es responsable de la integridad de una o más realizaciones de casos de uso-diseño, y debe garantizar que cumplen los requisitos que se esperan de ellos. Una realización de caso de uso-diseño debe realizar correctamente el comportamiento de su correspondiente realización de caso de uso-análisis del modelo de análisis, así como el comportamiento de su correspondiente caso de uso del modelo de casos de uso, y sólo esos comportamientos.
El ingeniero de casos de uso no es responsable de las clases, subsistemas, interfaces y relaciones de diseño que se utilizan en la realización del caso de uso. Éstas son responsabilidad del correspondiente ingeniero de componentes.

☑ **Ingeniero de componentes:** El ingeniero de componentes define y mantiene las operaciones, métodos, atributos, relaciones y requisitos de implementación de una o más clases del diseño, garantizando que cada clase del diseño cumple los requisitos que se esperan de ella según las realizaciones de caso de uso en las que participa.

## Flujo de Trabajo

Los arquitectos inician la creación de los modelos de diseño y de despliegue. Ellos esbozan los nodos del modelo de despliegue, los subsistemas principales y sus interfaces, las clases del diseño importantes como las activas, y los mecanismos genéricos de diseño del modelo de diseño. Después, los ingenieros de casos de uso realizan cada caso de uso en términos de clases y/o subsistemas del diseño participantes y sus interfaces. Las realizaciones de caso de uso resultantes establecen los requisitos de comportamiento para cada clase o subsistema que participe en alguna realización de caso de uso. Los ingenieros de componentes especifican a continuación los requisitos, y los integran dentro de cada clase. A lo largo del flujo de trabajo del diseño, los desarrolladores identificarán, a medida que evolucione el diseño, nuevos candidatos para ser subsistemas, interfaces, clases y mecanismos de diseño genéricos, y los ingenieros de componentes deberán refinarlos y mantenerlos.

## Actividades

☑ **Diseño de la arquitectura:** El objetivo del diseño de la arquitectura es esbozar los modelos de diseño y despliegue y su arquitectura mediante la identificación de los siguientes elementos:
- Nodos y sus configuraciones de red.
- Subsistemas y sus interfaces.
- Clases del diseño significativas para la arquitectura, como las clases activas.
- Mecanismos de diseño genéricos que tratan los requisitos comunes, como los requisitos especiales sobre persistencia, distribución, rendimiento y demás, tal y como se capturaron durante el análisis sobre las clases y las realizaciones de caso de uso-análisis.

A lo largo de esta actividad los arquitectos consideran las distintas posibilidades de reutilización, además mantiene y refina las vistas arquitectónicas.

*Identificación de nodos y configuraciones de red*: las configuraciones de red habituales utilizan un patrón de tres capas en el cual los clientes (las interacciones de los usuarios) se dejan en una capa, la funcionalidad de base de datos en otra, y la lógica del negocio o de la aplicación en una tercera.

*Identificación de subsistemas y de sus interfaces*: Pueden ser identificados al inicio o bien a medida que el sistema va creciendo. Por otro lado identificamos subsistemas de las capas específicas y general de la aplicación, así como también subsistemas intermedios y de software del sistema. Deberían definirse dependencias entre subsistemas si sus contenidos están relacionados. Si utilizamos interfaces entre subsistemas, las dependencias van hacia las interfaces, no hacia los subsistemas. Las interfaces definen operaciones que son accesibles "desde afuera" del subsistema. *ver en el libro 223-231.

*Identificación de clases del diseño relevantes para la arquitectura*: evitar identificar demasiadas clases en esta etapa, solo se necesita un esbozo de clases significativas par la arquitectura. Algunas clases del diseño pueden ser el resultado de un esbozo con clases del análisis.

*Identificación de mecanismos genéricos de diseño*: se estudian los requisitos comunes y cómo tratarlos, teniendo en cuenta las tecnologías de diseño e implementación disponibles.
☑ **Diseño de un caso de uso:** Los objetivos del diseño de un caso de uso son:
- Identificar las clases del diseño y/o los subsistemas cuyas instancias son necesarias para llevar a cabo el flujo de sucesos del caso de uso.
- Distribuir el comportamiento del caso de uso entre los objetos del diseño que interactúan y/o entre los subsistemas participantes.
- Definir los requisitos sobre las operaciones de las clases del diseño y/o sobre los subsistemas y sus interfaces.
- Capturar los requisitos de implementación del caso de uso.

*Identificación de clases del diseño participantes:*
- Estudiar las clases del análisis que participan en la correspondiente realización de caso de uso-análisis. Identificar las clases del diseño que poseen una traza hacia esas clases del análisis, creadas por el ingeniero de componentes en el diseño de clases o por el arquitecto en el diseño arquitectónico.
- Estudiar los requisitos especiales de la correspondiente realización de caso de uso-análisis. Identificar las clases del diseño que realizan esos requisitos especiales.
- Identificar clases necesarias y asignar su responsabilidad a un ingeniero de componentes.
- Si aún falta alguna clase del diseño, el ingeniero de CU debe avisar de dicha situación a los arquitectos y a los ingenieros de componentes.

Descripción de las interacciones entre objetos del diseño:
Debemos describir cómo interactúan los objetos del diseño. Esto se logra mediante diagramas de secuencia que contienen las instancias de los actores, los objetos del diseño y las transmisiones de mensajes entre éstos. Para crear el diagrama de secuencia, debemos seguir paso a paso desde el inicio el flujo del CU, decidiendo qué objetos y qué interacciones son necesarias para realizar el CU. Suele ser útil crear uno por cada subflujo distinto.

Identificación de subsistemas e interfaces participantes:
Algunas veces es mejor diseñar un CU en términos de los subsistemas y/o interfaces. Debemos identificar los subsistemas necesarios para realizar el CU.

Descripción de interacciones entre subsistemas:
Se realiza mediante diagramas de secuencia que contiene instancias actores, subsistemas y transmisiones de mensajes. Las líneas de vida denotan ahora subsistemas en lugar de objetos.

Captura de requisitos de implementación:
Aquí incluimos todos los requisitos que deben tratarse durante la implementación.

☑ **Diseño de una clase**
*Esbozar la clase del diseño*
- Diseñar clases de interfaz es dependiente de la tecnología de interfaz específica que se utilice.
- Diseñar clases de entidad que representen información persistente (o clases que tienen otros requisitos persistentes) a menudo implica el uso de tecnologías de BD específicas.
- Diseñar clases de control es una tarea delicada. Debido a que encapsulan secuencias, coordinación de otros objetos o algunas veces pura lógica del negocio, es necesario considerar los siguientes aspectos:
    - Aspectos de distribución: si la secuencia necesita ser distribuida y manejada por diferentes nodos de una red, se puede requerir separar las clases del diseño en diferentes nodos para realizar la clase de control.
    - Aspectos de rendimiento: puede que no sea justificable tener clases del diseño separadas para realizar la clase de control. En cambio, la clase de control podría realizarse por las mismas clases del diseño que están realizando algunas clases de interfaz o clases de entidad relacionadas.
    - Aspectos de transacción: las clases de control a menudo encapsulan transacciones. Sus correspondientes diseños deben incorporar cualquier tecnología de manejo de transacción existente que se esté utilizando.

*Identificar Operaciones:*
Identificamos las operaciones que las clases de diseño van a necesitar y las describimos utilizando la sintaxis de los lenguajes de programación.

*Identificar Atributos:*
Identificamos los atributos que las clases de diseño van a necesitar y los describimos utilizando la sintaxis de los lenguajes de programación.

- Considerar los atributos sobre cualquier clase de análisis.
- Los tipos de los atributos están restringidos por el lenguaje de programación.
- Intentar reutilizar tipos de atributos.

*Identificar asociaciones y agregaciones:*
Se estudia la transmisión de mensajes en los diag. de secuencia para determinar que asociaciones son necesarias. El numero de relaciones entre clases debe estar minimizado.

- Considerar las asociaciones y agregaciones involucrando la correspondiente clase de análisis (o clases). Algunas veces estas relaciones (en el modelo de análisis) implican la necesidad de una o varias relaciones correspondientes (en el modelo de diseño) que involucre la clases de diseño.
- Refinar la multiplicidad de las asociaciones, nombres de rol, cales de asociación, roles de ordenación, roles de cualificación y asociaciones n-arias de acuerdo con el soporte del lenguaje de programación utilizado.
- Refinar la navegabilidad de las operaciones. La dirección de las transmisiones de mensajes entre objetos se debe corresponder con la navegabilidad de las asociaciones entre sus clases.

*Identificar las generalizaciones:*
Las generalizaciones deben ser usadas con la misma semántica definida en el lenguaje de programación. Si el lenguaje no admite generalización, usar asociación y/o agregación.

*Describir los métodos:*
Los métodos se usan en el diseño para especificar cómo se deben realizar las operaciones. Los describimos utilizando lenguaje natural o pseudocódigo.

*Describir estados:*
Es importante el uso de diagramas de estado para describir las transiciones de estado de un objeto del diseño. Algunos objetos, de acuerdo a su estado, cambian su comportamiento al recibir un mensaje.

*Tratar requisitos especiales:*
Acá se trata a cualquier requisito no considerado anteriormente. De ser posible, utilizar los mecanismos de diseño identificados por el arquitecto. Se pueden posponer algunos requisitos para ser tratados en la implementación.

☑ **Diseño de un subsistema:**
Los objetivos del diseño de un subsistema son:
- Garantizar que el subsistema es tan independiente como sea posible de otros subsistemas y/o de sus interfaces.
- Garantizar que el subsistema proporciona las interfaces correctas.
- Garantizar que el subsistema cumple su propósito de ofrecer una realización correcta de las operaciones tal y como se definen en las interfaces que proporciona.

*Mantenimiento de las dependencias entre subsistemas:* Es mejor ser dependiente de una interfaz que serlo de un subsistema, ya que un subsistema podría ser sustituido por otro que posea un diseño interno distinto, mientras que en ese mismo caso, no estaríamos obligados a sustituir la interfaz. (No interesa cómo un subsistema lleva a cabo las colaboraciones para dar soporte a una interfaz, siempre y cuando lo haga correctamente).
*Mantenimiento de interfaces proporcionadas por el subsistema:* Las operaciones definidas por las interfaces que proporciona un subsistema deben soportar todos los roles que cumple el subsistema en las diferentes realizaciones de caso de uso.

*Mantenimiento de los contenidos de los subsistemas:* Un subsistema cumple sus objetivos cuando ofrece una realización correcta de las operaciones tal y como están descritas por las interfaces que proporciona.
Tener en cuenta que:
- Por cada interfaz que proporcione el subsistema, debería haber clases del diseño u otros subsistemas dentro del subsistema que también proporcionen la interfaz.
- Para clarificar cómo el diseño interno de un subsistema realiza cualquiera de sus interfaces o casos de uso, podemos crear colaboraciones en términos de los elementos contenidos en el subsistema. Podemos hacer esto para justificar los elementos contenidos en el subsistema.

## Diagrama de Transición de Estados

Muestra el comportamiento del sistema ante eventos y va marcando cómo reacciona el objeto a través de estados. Se realiza sobre CU, clases del MODP, del DCD, a interfaces y al sistema completo. El nivel de detalle lo determina quien realiza el DTE y de sus necesidades.
Elementos:
- Estado: condición del objeto en un instante de tiempo.
- Evento: hecho relevante. Tipos:
  - Externos: la incidencia viene desde afuera
  - Interno: es mejor utilizar el diagrama de secuencia.
  - De tiempo: transcurrido un período provoca una reacción en el sistema.
- Transición: relación entre dos estados
- Actividad: reacción del objeto
- Condición de guarda: restricción. Tiene que cumplirse si se da esta condición.
- Estados anidados: adentrarse en un estado y ver qué sucede.
- Condición inicial: pseudoestado, pasa al estado siguiente l crearse el objeto.
- Estado final: puede o no tenerlo.

## Calidad en el diseño

- El PUD plantea un modelo implícito que permite la trazabilidad de requisitos
- Mientras más patrones apliquemos mayor será la calidad, logrando modelos altamente cohesivos y débilmente acoplados
- Utilizando abstracciones: interfaces, subsistemas, clases de diseño
- A través de las verificaciones:
  - Que los mensajes de los diagramas de secuencia y estado tengan la misma asignatura que la operación
  - Que la clase a la cual se le realiza el DTE posea el atributo estado

# Workflow de Implementación

Propósitos:

- Planificar las integraciones de sistema necesarias en cada iteración.
- Distribuir el sistema asignando componentes ejecutables a nodos en el diagrama de despliegue. Esto se basa fundamentalmente en las clases activas encontradas durante el diseño.
- Implementar las clases y subsistemas encontrados durante el diseño.
- Probar componentes individualmente, y a continuación integrarlos compilándolos y enlazándolos en uno o más ejecutables, antes de ser enviados para ser integrados y llevar a cabo las comprobaciones de sistema.

| ENTRADA | PROCESO | SALIDA |
|---|---|---|
| • Requisitos adicionales<br>• Modelo de CU<br>• Modelo de Diseño<br>• Modelo de Despliegue<br>• Descripción de la arquitectura (vista Diseño) | • Implementación de la Arquitectura<br>• Integrar el Sistema<br>• Implementar un Subsistema<br>• Implementar una Clase<br>• Realizar pruebas de unidad | • Modelo de Implementación<br>• Componente<br>• Descripción de la arquitectura (Vista de Implementación)<br>• Plan de Integración |

## Ciclo de vida

- **Inicio:** -
- **Elaboración**: se realiza la línea base ejecutable de la arquitectura.
- **Construcción:** es el centro de esta fase.
- **Transición:** se realiza la corrección de defectos.

## Artefactos

☑**Modelo de implementación:** Describe como los elementos del diseño, como las clases, se implementan en términos de componentes, como ficheros de código fuente, ejecutables, etc. También cómo se organizan los componentes de acuerdo con los mecanismos de estructuración y modularización disponibles en el entorno de implementación y en el lenguaje de programación utilizados, y cómo dependen los componentes unos de otros.

☑**Componente*:*** Un componente es el empaquetamiento físico de los elementos de un modelo, como son clases del modelo de diseño.
Los componentes tienen las siguientes características:

- Los componentes tienen relaciones de traza con los elementos del modelo que implementan.
- Es normal que un componente implemente varios elementos, por ejemplo, varias clases.
- Los componentes proporcionan las mismas interfaces que los elementos del modelo que implementan.
- Puede haber dependencias de compilación entre componentes, denotando qué componentes son necesarios para compilar un componente determinado.

Stubs: es un componente que puede ser utilizado minimizar el número de componentes nuevos necesarios en cada nueva versión (intermedia) del sistema, simplificando así los problemas de integración y las pruebas de integración. Favorece la reutilización.

☑**Subsistema de implementación:** Deberían seguir la traza uno a uno de sus subsistemas de diseño correspondientes.

- El subsistema de implementación debería definir dependencias análogas hacia otros subsistemas de implementación o interfaces.
- El subsistema de implementación debería proporcionar las mismas interfaces.
- El subsistema de implementación debería definir qué componentes o, recursivamente, qué subsistemas de implementación dentro del subsistema deberían proporcionar las interfaces proporcionadas por el subsistema. Además, estos componentes contenidos deberían seguir la traza de las clases correspondientes en el subsistema de diseño que implementan.

☑**Interfaz:** Un componente que implementa una interfaz ha de implementar correctamente todas las operaciones definidas por la interfaz. Un subsistema de implementación que proporciona una interfaz tiene también que contener componentes que proporcionen la interfaz y otros subsistemas que proporcionen la interfaz.

☑**Descripción de la arquitectura (vista del modelo de implementación):**
La descripción de la arquitectura contiene una **vista del modelo de implementación**, el cual representa sus artefactos significativos arquitectónicamente.
Los siguientes artefactos son considerados en el modelo de implementación significativos arquitectónicamente:

- La descomposición del modelo de implementación en subsistemas, sus interfaces y las dependencias entre ellos.
- Componentes claves, como los componentes que siguen la traza de las clases de diseño significativas arquitectónicamente, los componentes ejecutables y los componentes que son generales, centrales, que implementan mecanismos de diseño genéricos de los que dependen muchos otros componentes.

☑**Plan de Integración de Construcciones:** Es importante construir el software incrementalmente en pasos manejables, de forma que cada paso dé lugar a pequeños problemas de integración o prueba. El resultado de cada paso es llamado "construcción", que es una versión ejecutable del sistema, usualmente una parte específica del sistema. Cada construcción es entonces sometida a pruebas de integración antes de que se cree ninguna otra construcción.

La integración incremental es para la integración del sistema lo que el desarrollo iterativo controlado es para el desarrollo de software en general. Ambos se centran en un incremento bastante pequeño y manejable de la funcionalidad.
Un *plan de integración de construcciones describe* la secuencia de construcciones necesarias en una iteración.
Describe lo siguiente para cada construcción:

- La funcionalidad que se espera que sea implementada en dicha construcción. Consiste en una lista de casa de uso o escenarios o parte de ellos, como se discutió en capítulos anteriores. Esta lista puede también referirse a otros requisitos adicionales.
- Las partes del modelo de implementación que están afectadas por la construcción.

## Trabajadores

☑ **Arquitecto:** es responsable de la integridad del modelo de implementación y asegura que el modelo como un todo es correcto, competo y legible.
El modelo es correcto cuando implementa la funcionalidad descrita en el modelo de diseño y en los requisitos adicionales, y sólo esta funcionalidad.
Un resultado importante de la implementación es la asignación de componentes ejecutables a nodos. El arquitecto es responsable de esta asignación, la cual se representa en la vista de la arquitectura del modelo de despliegue.

☑**Ingeniero de componentes:** Define y mantiene el código fuente de uno o varios componentes, garantizando que cada componente implementa la funcionalidad correcta.
El ingeniero de componentes necesita garantizar que los contenidos de los subsistemas de implementación son correctos, que sus dependencias son otros subsistemas o interfaces son correctas y que implementan correctamente las interfaces que proporcionan.

☑**Integrador de sistemas:** Tiene como responsabilidad planificar la secuencia de construcciones necesarias en cada iteración y la integración de cada construcción cuando sus partes han sido implementadas.

## Flujo de Trabajo

Este proceso es iniciado por el arquitecto esbozando los componentes clave en el modelo de implementación. A continuación, el integrador de sistemas planea las integraciones de sistema necesarias en la presente iteración como una secuencia de construcciones. Para cada construcción el integrador de sistemas describe la funcionalidad que debería ser implementada y qué partes del modelo de implementación se verán afectadas. Los requisitos sobre los subsistemas y componentes en la construcción son entonces implementados por ingenieros de componentes. Los componentes resultantes son probados y pasados al integrador de sistemas para su integración. Entonces, el integrador de sistemas integra los nuevos componentes en una construcción y la pasa a los ingenieros de pruebas de integración para llevar a cabo pruebas de integración. Luego, los desarrolladores inician la implementación de la siguiente construcción, tomando en consideración los defectos de la construcción anterior.

☑**Implementación de la arquitectura:**
El propósito de la implementación de la arquitectura es esbozar el modelo de implementación y su arquitectura mediante:

- La identificación de componentes significativos arquitectónicamente, tales como componentes ejecutables.
- La asignación de componentes a los nodos en las configuraciones de redes relevantes.

El mayor reto durante la implementación es crear dentro de los subsistemas de implementación los componentes que implementen los subsistemas de diseño correspondientes.
La asignación de componentes a nodos es muy importante para la arquitectura del sistema, y debería ser representada en una vista de la arquitectura del modelo de despliegue.

☑**Integrar el sistema:**
Los objetivos de la integración del sistema son:
- Crear un plan de integración de construcciones que describa las construcciones necesarias en una iteración y los requisitos de cada construcción.
- Integrar cada construcción antes de que sea sometida a pruebas de integración.

Planificación de una construcción**:** Se debe tener en cuenta los siguientes criterios:

1. Una construcción debería añadir funcionalidad a la construcción previa implementando casos de uso completos o escenarios de éstos. Las pruebas de integración de la construcción estarán basadas en estos casos de uso y escenarios; es más fácil probar casos de uso completos que fragmentos de éstos.
2. Una construcción no debería incluir demasiados componentes nuevos o refinados. Si no es así, puede ser muy difícil integrar la construcción y llevar a cabo las pruebas de integración. Si fuera necesario algunos componentes pueden implementarse como *stubs* para minimizar el número de nuevos componentes introducidos en la construcción.
3. Una construcción debería estar basada en la construcción anterior, y debería expandirse hacia arriba y hacia los lados de la jerarquía de subsistemas. Esto significa que la construcción inicial debería empezar en las capas inferiores; las construcciones subsiguientes se expanden entonces había arriba a las capas general de la aplicación y específica de aplicación. La razón fundamental de esto es simplemente que es difícil implementar componentes en las capas superiores antes de que estén colocados y funcionando adecuadamente los componentes necesarios en las capas inferiores.

Manteniendo estos criterios en mente, uno puede empezar a evaluar los requisitos, tales como los casos de uso, que han de ser implementados.
En cualquier caso, es importante identificar los requisitos apropiados a implementar en una construcción y dejar el resto de los requisitos para construcciones futuras.

☑**Implementar un subsistema:**
El propósito de implementar un subsistema es el de asegurar que un subsistema cumple su papel en cada construcción, tal y como se especifica en el plan de integración de la construcción.

Mantenimiento de los contenidos de los subsistemas: Un subsistema cumple su propósito cuando los requisitos a ser implementados en la construcción actual y aquéllos que afectan al subsistema están implementados correctamente por los componentes dentro del subsistema.

☑**Implementar una clase:**
El propósito de la implementación de una clase es implementar una clase de diseño en una componente fichero. Esto incluye lo siguiente:

- Esbozo de los componentes de fichero: Es decir, especificar el componente fichero y considerar su ámbito. Los componentes fichero elegidos deberían facilitar la compilación, instalación y mantenimiento del sistema.
- Generación de código a partir de las clases de diseño
- Implementación de operaciones**:** Incluye la elección de un algoritmo y unas estructuras de datos apropiadas de las acciones requeridas por el algoritmo. Los estados descritos para la clase de diseño pueden influenciar el modo en que son implementadas las operaciones, ya que sus estados determinan su comportamiento cuando ésta recibe un mensaje.
- Comprobar si los componentes ofrecen las interfaces apropiadas

☑ **Realizar prueba de unidad:**
El propósito de realizar la prueba de unidad es probar los componentes implementados como unidades individuales. Se llevan a cabo los siguientes tipos de prueba de unidad:

- La *prueba de especificación*, o **prueba de caja negra**, que verifica el comportamiento de la unidad observable externamente.
- La *prueba de estructura*, o **prueba de caja blanca**, que verifica la implementación interna de la unidad.

También se realizan pruebas sobre las unidades, como pruebas de rendimiento, utilización de memoria, carga y capacidad. Además, se realizan pruebas de integración y sistema para asegurar que los diversos componentes se comportan correctamente cuando se integran.

Realización de pruebas de especificación**:** Se realiza para verificar el comportamiento del componente sin tener en cuenta *cómo* se implementa dicho comportamiento en el componente.
Realización de prueba de estructura**:** El ingeniero de componentes debería asegurarse de probar todo el código durante las pruebas de estructura, lo que quiere decir que cada sentencia ha de ser ejecutada al menos una vez.

# Workflow de Prueba

El principal objetivo de la prueba es realizar y evaluar las pruebas como se describe en el modelo de prueba. Los ingenieros de pruebas inician esta tarea planificando el esfuerzo de prueba en cada iteración, y describen entonces los casos de prueba necesarios y los procedimientos de prueba correspondientes para llevar a cabo las pruebas. Si es posible, los ingenieros de componentes crean a continuación los componentes de prueba para automatizar algunos de los procedimientos de prueba. Todo esto se hace para cada construcción entregada como resultado del flujo de trabajo de implementación.
Con estos casos, procedimientos y componentes de prueba como entrada, los ingenieros de pruebas de integración y de sistema prueban cada construcción y detectan cualquier defecto que encuentren en ellas. Los defectos sirven como realimentación tanto para otros flujos de trabajo, como el de diseño y el de implementación, como para los ingenieros de pruebas para que lleven a cabo una evaluación sistemática de los resultados de las pruebas.

| ENTRADA | PROCESO | SALIDA |
|---|---|---|
| • Modelo de Casos de Uso<br>• Modelo de Análisis<br>• Modelo de Diseño<br>• Modelo de Implementación<br>• Descripción de la arquitectura (vista de todos los WF) | • Planificar Prueba<br>• Diseñar prueba<br>• Implementar una prueba<br>• Realizar pruebas de integración<br>• Realizar prueba del sistema<br>• Evaluar una prueba | • Plan de prueba<br>• Modelo de prueba:<br>- Caso de prueba<br>- Procedimiento de prueba<br>- Componente de prueba<br>• Defecto<br>• Evaluación de prueba (para una iteración) |

## Ciclo de vida

- **Inicio:** se hace la parte de la planificación inicial de las pruebas.
- **Elaboración**: se realizan pruebas de la línea base ejecutable de la arquitectura.
- **Construcción:** se realizan las pruebas, cuando el grueso del sistema está implementado.
- **Transición:** se realiza la corrección de defectos durante los primeros usos y las pruebas de regresión.

## Artefactos

☑ **Modelo de pruebas:** Describe cómo se prueban los componentes ejecutables (como construcciones) en el modelo de implementación con pruebas de integración y de sistema.
El modelo de pruebas es una colección de casos de prueba, procedimientos de prueba y componentes de prueba. Si el modelo es grande puede introducirse paquetes para manejar su tamaño.
☑ **Caso de prueba:** Un caso de prueba específica una forma de probar el sistema, incluyendo la entrada o resultado con la que se ha de probar y las condiciones bajo las que ha de probarse.
Los siguientes son casos de prueba comunes:
- Un caso de prueba que especifica cómo probar un caso de uso o un escenario específico de un caso de uso. Especifica una prueba del sistema como "caja negra", es decir, una prueba del comportamiento observable externamente del sistema.
- Un caso de prueba que especifica cómo probar una realización de caso de uso-diseño o un escenario específico de la realización. Especifican una prueba del sistema como una "caja blanca", es decir, una prueba de interacción interna entre los componentes del sistema.
☑ **Procedimiento de prueba:** Especifica cómo realizar uno o varios casos de pruebas o partes de éstos. Se puede reutilizar un procedimiento de prueba para varios casos de prueba, así como también reutilizar varios procedimientos de prueba para un caso de prueba.
☑ **Componente de prueba:** Automatiza uno o varios procedimientos de prueba o partes de ellos.
Los componentes de prueba pueden ser desarrollados utilizando un lenguaje de guiones o un lenguaje de programación, o pueden ser grabados con una herramienta de automatización de pruebas.
Los componentes de prueba se utilizan para probar los componentes en el modelo de implementación, proporcionando entradas de prueba, controlando y monitorizando la ejecución de los componentes a probar e informando de los resultados de pruebas.
☑ **Plan de Pruebas:** describe las estrategias, recursos y planificación de la prueba.
☑ **Defecto:** Un defecto es una anomalía del sistema, como por ejemplo un síntoma de un fallo software o un problema descubierto en una revisión.
☑ **Evaluación de prueba:** Es una evaluación de los resultados de los esfuerzos de prueba, tales como la cobertura del caso de prueba, la cobertura de código y el estado de los defectos.

## Trabajadores

☑ **Diseñador de pruebas:** es el responsable de la integridad del modelo de pruebas, asegurando que cumpla con su propósito. También planea las pruebas (deciden los objetivos apropiados y la planificación de las pruebas). Además, seleccionan y describen los casos de pruebas y los procedimientos de prueba necesarios y son los responsables de la

evaluación de las pruebas de integración y de sistema cuando estas se ejecutan. O sea que no lleva a cabo las pruebas, sino que las preparan y las evalúan.
☑ **Ingeniero de Componentes:** son responsables de los componentes de prueba que automatizan algunos de los procedimientos de prueba.
☑ **Ingeniero de pruebas de integración:** son responsables de realizar las pruebas de integración que se necesitan para cada construcción producida en el workflow de implementación. Estas pruebas de integración se realizan para verificar que los componentes integrados en una construcción funcionan correctamente juntos. También documenta los defectos en los resultados de las pruebas de integración.
☑ **Ingeniero de pruebas de sistema:** es responsable de realizar las pruebas de sistema necesarias sobre una construcción que muestra el resultado de una iteración completa. Estas pruebas de sistema se llevan a cabo principalmente para verificar las interacciones entre los actores y el sistema, por ello, se derivan a menudo de los casos de prueba que especifican cómo probar los CU.
El ingeniero de pruebas de sistema se encarga de documentar los defectos en los resultados de las pruebas de sistema.

## Flujo de Trabajo

Los ingenieros de pruebas inician esta tarea planificando el esfuerzo de prueba en cada iteración, y describen entonces los casos de prueba necesarios y los procedimientos de prueba correspondientes para llevarlos a cabo. Si es posible, los ingenieros de componentes crean a continuación los componentes de prueba para automatizar algunos de los procedimientos de prueba. Todo esto se hace para cada construcción entregada como resultado del flujo de trabajo de implementación.
Con estos casos, procedimientos y componentes de prueba como entrada, los ingenieros de pruebas de integración, y de sistema prueba cada construcción y detectan cualquier defecto que encuentren en ellos. Los defectos sirven como realimentación tanto para otros flujos de trabajo, como el de diseño y el de implementación, como para los ingenieros de pruebas para que lleven a cabo una evaluación sistemática de los resultados de las pruebas.

☑ **Planificar Prueba:**
Se planifican los esfuerzos de prueba en una iteración llevando a cabo estas tareas:
- Describir una estrategia de prueba.
- Estimar los requisitos para el esfuerzo de la prueba.
- Planificar el esfuerzo de la prueba.

Ningún sistema puede ser probado completamente; por tanto, se debería identificar los casos, procedimientos y componentes de prueba con un mayor retorno a la inversión en términos de mejora de calidad.
☑ **Diseñar prueba:**
- Identificar y describir los casos de prueba.
- Identificar y estructurar los procedimientos de prueba

- Diseño de los casos de prueba de integración: estos casos de prueba se usan para verificar que los componentes interaccionan entre sí de la manera correcta. Se consideran como entrada los diagramas de interacción de las realizaciones de CU.
- Diseño de los casos de prueba de sistema: las pruebas de sistema se usan para probar que el sistema funciona correctamente como un todo. Se prueban combinaciones de CU instanciados bajo condiciones diferentes.
- Diseño de los casos de prueba de regresión: Algunos casos de prueba de construcciones anteriores pueden ser usados para pruebas de regresión en construcciones subsiguientes.
- Identificación y estructuración de los procedimientos de prueba: se intentan reutilizar procedimientos de prueba existentes tanto como sea posible

☑ **Implementar una prueba:** Se refiere a automatizar los procedimientos de prueba creando componentes de prueba si fuera posible.
☑ **Realizar pruebas de integración:** Se realizan las pruebas de integración necesarias para cada una de las construcciones creadas en una iteración, se recopilan los resultados de las pruebas y se informa a los ingenieros de componentes y a los diseñadores de pruebas los defectos encontrados.

☑ **Realizar prueba del sistema:** Se realizan las pruebas del sistema necesarias para cada una de las construcciones creadas en una iteración y se recopilan los resultados de las pruebas.
☑ **Evaluar una prueba:** Su propósito es evaluar los esfuerzos de las pruebas en una iteración. Los diseñadores de pruebas evalúan los resultados de la prueba comparando los resultados obtenidos con los esbozados en el plan de la prueba. Metricas: % de casos de prueba completados y la fiabilidad del sistema.

## *Tipos de Tests (Pruebas)

- Test de Operación: Es el más común. El sistema es probado en operación normal.
- Tests de Escala completa: Se ejecuta el sistema al máximo: valores máximos, muchos equipos conectados, muchos usuarios, etc.
- Tests de Performance o de capacidad: Mide la capacidad de procesamiento del sistema.
- Tests de Sobrecarga: determina cómo se comporta el sistema cuando es sobrecargado.
- Tests Negativos: El sistema se usa intencionalmente de forma incorrecta. Se prueban casos especiales.
- Tests basados en requerimientos: Se rastrean directamente desde la especificación de requerimientos.
- Tests Ergonómicos: Son importantes si el sistemas será usado por gente inexperta.
- Tests de Documentación del Usuario: Se prueba la documentación del sistema.
- Tests de Aceptación: Es ejecutado por la organización que solicita el sistema.

## Niveles de Prueba

☑ Pruebas de Unidad: Es el nivel de prueba más bajo. Involucra: clases, bloques, paquetes de servicio, procedimientos y subrutinas. Consiste de:
- Prueba de especificación o Caja Negra: verifican el comportamiento de la interfaz de la unidad. Es importante ver si se produce una salida, y si es la correcta.
- Prueba estructural o de caja blanca: Se verifica que la estructura interna se la correcta. Es preferible hacer esta prueba al último.
- Prueba basada en estados: Prueba la interacción entre las operaciones de una clase, monitoreando los cambios que tienen lugar en los atributos de los objetos. Basarse en los diagramas de transición de estados. Se puede utilizar una "matriz de estados", en donde tenemos los estados como columnas y los estímulos en las filas. Se debe verificar que todos los estados se pueden alcanzar con alguna combinación de operaciones, en caso contrario hay alguna falla en el diseño de la clase.

☑ Pruebas de Integración: Involucra: CU. Determinan si las unidades desarrolladas trabajan correctamente juntas. Se realizan varias de estas pruebas a distintos niveles. Estas pruebas son necesarias porque al combinar las unidades pueden aparecer nuevas fallas, y porque la combinación aumenta el número de caminos posibles. Estas pruebas se realizan probando los CU, desde dos puntos de vista: uno interno (basado en diagramas de interacción) y uno externo (basado en las descripciones del modelo de requerimientos). Para un CU se hacen las siguientes pruebas:
- Pruebas del curso básico (curso normal).
- Pruebas de cursos alternativos.
- Pruebas de documentación de usuarios.

☑ Prueba de Sistema: se prueba el sistema completo. Se divide en:
- Pruebas de instalación: verifican que el sistema puede ser instalado en la plataforma del cliente y que funcionara correctamente.
- Pruebas de configuración: verifican que el sistema funciona correctamente en diferentes configuraciones(ej:de red)
- Pruebas negativas: intentan provocar que el sistema falle y así mostrar sus debilidades.
- Pruebas de estrés: identifican problemas con el sistema cuando hay recursos insuficientes.

## *Estrategias de Prueba

Lo más común es realizar las estrategias de prueba en el orden inverso al que se realiza el diseño y la implementación. Podemos ir realizando pruebas conforme vamos terminando los diseños, es decir la implantación y la prueba se realizan en forma intercalada e incremental. Hay varias maneras:

☑ Top-Down: Primero desarrollamos las interfaces entre subsistemas y luego se reemplaza con el código real, esto permite probar el flujo completo en niveles superiores en primer lugar, y en segundo lugar ir a niveles inferiores.
☑ Botton-Up: Es preferible este modo en los niveles más bajos. Minimiza las necesidades de implementar clases piloto sólo para pruebas, ya que las unidades certificadas trabajan como servidores.

# Workflow de Despliegue

El propósito del despliegue es volcar el producto finalizado a sus usuarios, presentar el software.
El centro de su actividad se encuentra al final de la fase de transición.

## Trabajadores

☑ **Gerente de despliegue:** planifica y organiza el despliegue. Es responsable del programa de soporte a las pruebas beta y de asegurarse que el producto esta empaquetado apropiadamente para su envio.
☑ **Gerente de proyecto:** es el que funciona de intermediario con los clientes, el responsable de aprobar los despliegues basados en retroalimentación y pruebas de evaluación resultantes , así también como de la aceptación del cliente para la entrega del producto.
☑ **Escritor técnico:** planea y produce el material de soporte al usuario final.
☑ **Desarrollador de curso:** planea y produce el material de capacitación.
☑ **Artista Grafico:** es el responsable por todos los trabajos de arte relacionados al producto.
☑ **Tester:** ejecuta pruebas de aceptación y es el responsable de asegurarse que el producto fue probado adecuadamente.
☑ **Implementador:** crea los scripts de instalación y los artefactos relacionados que ayudaran al usuario final a instalar el producto.

## Artefactos

El despliegue tiene un campo muy amplio, desde el despliegue de sistemas a medida hasta software que puede ser descargado de la web. Dado este amplio rango de productos, los artefactos listados debajo pueden ser requeridos o no, dependiendo del modo de despliegue.

☑ **Release:** el artefacto clave es el producto terminado, que puede consistir en:
- El software ejecutable, en todos los casos.
- Artefactos de instalación: scripts, herramientas, archivos, guías, información de licencia.
- Release notes: notas que describen el producto terminado al usuario final.
- Material de Soporte: como los manuales de usuario, operaciones y mantenimiento.
- Manuales de Capacitación

En el caso de **productos manufacturados**, se requieren artefactos adicionales para crear el “producto” como:
☑ La lista completa de ítems a ser incluidos en el producto manufacturado
☑ Empaquetamiento
☑ Dispositivos de almacenamiento: el material en el cual el producto será vendido, como por ejemplo: CDs

Otros artefactos de despliegue usados en el desarrollo del producto, pero no necesariamente entregados al cliente son:
☑ Prueba de resultados
☑ Resultados de retroalimentación (para prueba de betas)
☑ Resumen de pruebas de evaluación

## Flujo de Trabajo

☑ **Planear el Despliegue:** dado que un despliegue exitoso está definido por la voluntad del cliente de usar el nuevo software, el planeamiento del despliegue no solo debe tener en cuenta cómo y cuándo entregar el nuevo software, sino también debe asegurarse que el usuario final tiene toda la información necesaria para recibir el nuevo software adecuadamente y comenzar a usarlo. Para asegurarse de eso, los planes de despliegue incluyen una prueba de la

beta del programa, para ir asesorando al usuario de forma temprana a través de las betas que todavía están en construcción.
el planeamiento de despliegue general del sistema requiere un alto grado de colaboración y preparación del cliente. Una conclusión exitosa del proyecto del software puede ser impactada severamente por factores fuera del desarrollo mismo del software, tales como el edificio donde se instalará el software, la infraestructura de hardware fuera de lugar y usuarios que no están lo suficientemente preparados para adaptarse al nuevo software.
☑ **Desarrollo de material de soporte:** el material de soporte cubre toda la información que será requerida por el usuario final para instalar, operar, usar y mantener el sistema. También incluye material de capacitación para que todos los usos posibles que se le pueda dar al sistema sean efectivos.
☑ **Prueba del producto en el lugar de desarrollo:** esta prueba determina si el producto tiene la madurez suficiente para ser entregado como producto final o como distribución para beta-testers.
El testeo de betas se hace para pulir un amplio rango de detalles, permitiendo al usuario final retroalimentar y mejorar el producto antes de su entrega final y en el caso de sistemas desarrollados a medida, la prueba beta puede ser una instalación piloto en el lugar donde se instalara el sistema.
☑ **Crear el producto final:** hay que asegurarse que el producto está preparado para la entrega al cliente. El producto final consiste en todo lo que el usuario final necesitara para instalar y ejecutar el software finalmente.
☑ **Prueba Beta del producto final:** esto requiere que el producto sea instalado por el cliente, quien proveerá información sobre su performance y usabilidad.
En el contexto del desarrollo iterativo, el beta testing es esencial para asegurarse que las expectativas del cliente fueron satisfechas y la información provista por el usuario, se retroalimente a la próxima iteración del desarrollo.
☑ **Probar el producto en el lugar de instalación:** el producto debe ser instalado y probado por el cliente. Basándonos en las iteraciones y pruebas anteriores, en esta prueba no deberíamos encontrarnos con sorpresas y debería ser solo una formalidad para que el cliente acepte el sistema.
☑ **Empaquetar el producto:** estas actividades opcionales describen lo que debe pasar para producir un producto de "software empaquetado". En este caso, el producto final se guarda como producto maestro para su producción en masa y luego empaquetado en cajas con la lista de contenidos para su envío al cliente.

---

# Patrones

Un patrón es el conocimiento de una buena práctica en distintos aspectos del sistema. Permite reusar conocimiento. Realizan una descripción de un problema y la forma de solucionarlo.
Los patrones permiten diseños más flexibles, elegantes y reutilizables, mejorar la documentación y el mantenimiento del sistema, reutilizar buenos diseños y así realizarlos rápidamente.

**Elementos de un patrón**

- *Nombre*: permite incrementar nuestro vocabulario, introducirlo en la documentación y mejorar la comunicación con colegas.
- *Problema*: describe el contexto en donde aplicar el patrón.
- *Solución*: describen los elementos que constituyen el diseño y sus relaciones.
- *Consecuencias*: ventajas e inconvenientes de aplicar el patrón.

# Patrones de CU

- **Patrones de Estructura:** describe componentes básicos de los CU, explicando cómo deberían ser organizados y ofrece criterios para juzgar un CU.
  - **Conjunto de CU:** describe el comportamiento del sistema y consiste en un conjunto de CU que prestan un servicio útil para el actor.
    - **Compartir una visión clara:** definir el objetivo del sistema.
    - **Límite visible:** establecer un límite entre el sistema y su ambiente.
    - **Reparto claro de roles:** identificar actores del sistema, su rol y describirlo.
    - **Transacciones de valor:** identificar CU para el usuario.
    - **Siempre contar una historia:** reagrupar CU y definir una descripción general.
  - **Casos de Uso:** cada CU es una colección de escenarios que describen la forma en que un actor alcanza o fracasa un objetivo.
    - **Completar un objetivo simple:** definir objetivo del CU
    - **Nombrar con una frase verbal:** dar nombre al CU con un verbo activo.
    - **Escenarios más fragmentos:** describir el escenario de éxito del CU y pensar en alternativos.
    - **Alternativas exhaustivas:** describir escenarios alternativos.
    - **Adornos:** determinar información adicional del CU.
    - **Preciso y legible:** se describe el CU a trazo fino, con esto se valida el diagrama.
  - **Escenarios y pasos:** describir los pasos (acciones) del sistema o del actor para alcanzar el objetivo.
    - **Condiciones detectables:** definir claramente el escenario.
    - **Pasos nivelados:** buscar un equilibrio en el nivel de detalle de los escenarios.
    - **Actor intenta cumplir:** mostrar el resultado y el protagonista de cada paso.
    - **Progresar hacia adelante:** eliminar o simplificar pasos que no contribuyen con el actor.
    - **Tecnología neutral:** no hacer referencia a tecnología alguna.
  - **Relaciones entre CU:** técnicas para manejar el comportamiento repetitivo o complejo de los CU.
    - **Comportamiento común subordinado:** establecer CU con relaciones de inclusión.
    - **Interrumpir con extensiones:** establecer CU con relaciones de extensión.
    - **Promover alternativa:** promover alternativas complejas a CU separados.
    - **Abstracción capturada:** establecer CU con relaciones de generalización.
- **Patrones de Desarrollo:** describe características para el desarrollo efectivo de CU.
  - **El Proceso:** técnicas para crear un conjunto de CU.
    - **Respirar antes de profundizar:** realizar una descripción general (trazo grueso) antes de profundizar.
    - **Desarrollo en espiral:** desarrollar los CU de manera iterativa incremental.
    - **Formas múltiples:** seleccionar el formato más conveniente (plantillas).
    - **Revisión de dos capas:** llevar dos tipos de revisiones con un grupo interno y uno completo.
    - **Tiempo de dejar:** dejar de desarrollar el CU una vez que están comprendidos y completos.
    - **Licencia de escritor:** estilo de escritura del escritor.

- **El Equipo:**
  - **Equipos de escritura pequeños:** grupos de 2 o 3 personas.
  - **Equipos balanceados:** el perfil de los integrantes que tengan nexos con todos los interesados.
  - **Audiencia participante:** a la hora de describir CU tener la retroalimentación con el cliente.
- **De Edición:**
  - **Redistribuir la salud:** mover pasajes complicados o voluminosos a otro CU.
  - **Mezclar gotitas:** unir CU diminutos o fragmentos con CU relacionados.
  - **Limpiar la casa:** remover CU que no agregan valor al sistema.

# Patrones GRASP

Patrones para Asignación de Responsabilidades. Se aplican durante la construcción de diagramas de interacción y colaboración al asignar responsabilidades a los objetos y al diseñar la colaboración entre ellos.
Responsabilidad es un contrato u obligación de un tipo o clase. Las responsabilidades se relacionan con las obligaciones de un objeto respecto a su comportamiento. Hay dos tipos de responsabilidades:
☑ Relacionadas con el ***hacer****:*
- Hacer algo en uno mismo
- Iniciar una acción en otros objetos
- Controlar y coordinar actividades en otros objetos

☑ Relacionadas con el ***conocer***:
- Estar enterado de los datos privados encapsulados
- Estar enterado de la existencia de objetos nuevos
- Estar enterado de cosas que se pueden derivar o calcular

**Patrón Experto**

Lo hace quien conoce. Asigna una responsabilidad al experto en información: los objetos hacen cosas relacionadas con la información que poseen.
*Ejemplo:*
Venta conoce le total de la venta. DetalleDeVenta conoce el subtotal. EspecificacionProducto conoce el precio del producto
*Ventajas*:
- Se conserva bajo acoplamiento, lo que favorece a tener sistemas robustos y de fácil mantenimiento.
- El comportamiento se distribuye con las clases que cuentan con la información requerida, fomentando la creación de clases sencillas y cohesivas.

**Patrón Creador**

Asigna a la clase B la responsabilidad de crear una instancia de la clase A en uno de los siguientes casos:
- B agrega los objetos de A
- B contiene los objetos de A
- B registra las instancias de los objetos de A
- B usa específicamente los objetos de A
- B tiene los datos de inicialización que serán transmitidos a A cuando sea creado (así B es un experto respecto de la creación de A)

*Ejemplo:*
¿Quién debe crear el detalle de venta? → La Venta crea el Detalle de venta.
*Ventajas:*
Da soporte al bajo acoplamiento, lo cual supone menos dependencias respecto al mantenimiento y mejores oportunidades de reutilización. Como la clase creada tiende a ser visible al creador, el acoplamiento no aumenta.

**Patrón Bajo Acoplamiento**

Asignar una responsabilidad para mantener el bajo acoplamiento y aumentar la reutilización. Soporta el diseño de clases más independientes, que reducen el impacto de los cambios y también clases más reutilizables que aumentan la oportunidad de mayor productividad.

Ejemplo:
En vez de que el gestor cree el pago y luego el gestor agregue el pago a la venta, lo que se hace es que la venta al efectuar el pago, cree el pago
*Ventajas:*
- No se afectan componentes por cambios de otros componentes
- Fáciles de entender por separado
- Fáciles de reutilizar

**Patrón Alta Cohesión**
Asignar una responsabilidad de modo que la cohesión siga siendo alta.
La cohesión funcional es la medida de que tan relacionadas están las responsabilidades de una clase. Una alta cohesión hace que las clases muy relacionadas no hagan un trabajo enorme.
Una clase de alta cohesión posee un número relativamente pequeño de operaciones con una importante funcionalidad relacionada y poco trabajo por hacer. Colabora con otros objetos para compartir esfuerzo si la tarea es grande.
*Ventajas:*
- Mejoran la claridad y facilidad con que se entiende el diseño
- Se simplifica el mantenimiento y las mejoras en funcionalidad
- A menudo se genera el bajo acoplamiento
- La ventaja de una gran funcionalidad es que soporta una mayor capacidad de reutilización, porque una clase muy cohesiva puede destinarse a un propósito muy especifico.

**Patrón Controlador**
Asigna la responsabilidad del manejo de un mensaje de los eventos de un sistema, a una clase que represente una de las siguientes opciones:
- El sistema "global" (controlador de fachada)
- La empresa u organización global (controlador de fachada)
- Algo en el mundo real que es activo (por ej el papel de una persona) ya que pueda participar en la tarea (controlador de tareas)
- Un manejador artificial de todos los eventos del sistema de un C.U., generalmente denominado "manejador <nombre del C.U.>" (controlador del Use Case)

Usar la misma clase de controlador con todos los eventos del sistema en el mismo C.U.
No usar clases como: ventana, vista, documento, generalmente las reciben y delegan al controlador.

Un controlador es un objeto de interfaz no destinada al usuario que se encarga de manejar un evento del sistema. (gestor)
En todos los CU en un diseño orientado a objetos hay que elegir los controladores que manejan los eventos de entrada externa. La misma clase debería usarse con todos los eventos de un CU, de modo que podamos conservar la información referente al estado del CU, lo que servirá para identificar eventos fuera de secuencia. No se le debe delegar demasiada responsabilidad al controlador.

Tipos de controladores:
- *Controlador de fachada*: representan al sistema global. Son adecuados cuando el sistema tiene solo unos cuantos eventos o cuando es imposible redirigir los mensajes a otros controladores.
- *Manejador artificial de CU*: se plantea un controlador para cada CU, que es un concepto artificial no un objeto del dominio, cuyo objetivo es dar soporte al sistema. Este manejador debería usarse cuando un controlador empieza a saturarse con demasiadas responsabilidades, lo que produce alto acoplamiento o baja cohesión.

*Ventajas:*
- Mayor potencial de los componentes reusables: garantiza que la empresa o los procesos del dominio sean manejados por la capa del dominio y no por la de interfaz.

- Reflexionar sobre el estado de un CU: a veces no es necesario asegurar que las operaciones ocurran en una cierta secuencia legal o poder saber el estado actual de la actividad y las operaciones del CU.

*Problemas y soluciones:*
Controladores saturados: un controlador mal diseñado presentara baja cohesión, dispersa y con demasiada responsabilidad, se lo denomina controlador saturado. Entre los signos de saturación podemos mencionar:
- Hay una sola clase controlador que recibe todos los eventos del sistema y estos son excesivos.
- El controlador realiza el mismo muchas tareas necesarias para cumplir el evento del sistema, sin delegar el trabajo, lo que suele ser una violación de los patrones Experto y Alta Cohesión.
- Un controlador posee muchos atributos, y conserva información importante sobre el dominio, información que debería haber sido distribuida entre otros objetos y también duplica información en otra parte.

Soluciones:
- Agregar más controladores: un sistema no necesariamente debe tener uno solamente. Además del controlador de fachada, se recomienda los de papeles o de CU.
- Diseñe el controlador de modo que delegue fundamentalmente u otros objetos el desempeño de las responsabilidades de la operación del sistema.

# Patrones de Diseño

Objetivos de los patrones:
Los patrones de diseño pretenden:
☑ Proporcionar catálogos de elementos reusables en el diseño de sistemas software.
☑ Evitar la reiteración en la búsqueda de soluciones a problemas ya conocidos y solucionados anteriormente.
☑ Formalizar un vocabulario común entre diseñadores.
☑ Estandarizar el modo en que se realiza el diseño.
☑ Facilitar el aprendizaje de las nuevas generaciones de diseñadores condensando conocimiento ya existente.

Asimismo, no pretenden:
☑ Imponer ciertas alternativas de diseño frente a otras.
☑ Eliminar la creatividad inherente al proceso de diseño.
No es obligatorio utilizar los patrones siempre, solo en el caso de tener el mismo problema o similar que soluciona el patrón.

**Tipos de patrones**
- De comportamiento
- De creación
- Estructurales
- De sistemas

**Nivel de un patrón**
- Clase
- Componente
- Arquitectónico

| | **PROPÓSITO**<br>¿Qué hace el patrón? | | |
|---|---|---|---|
| | ***De creación***<br>Proceso de creación de un objeto | ***Estructurales***<br>Composición de clases u objetos | ***De comportamiento***<br>Modo en que las clases y objetos interactúan y se reparten las responsabilidades |

| **Ámbito** ¿donde se aplica? | | | | |
|---|---|---|---|---|
| | ***CLASE***<br>Trata relaciones entre clases y subclases | Delegan alguna parte del proceso de creación de objetos en subclases.<br>Factory Method | Usan la herencia para componer clases.<br>Adapter | Usan la herencia para describir algoritmos y flujos de control.<br>Template Method |
| | ***OBJETO***<br>Trata con relaciones entre objetos | Delegan alguna parte del proceso de creación de objetos en otro objeto.<br>Singleton | Describen formas de ensamblar objetos.<br>Facade | Describe como cooperan un grupo de objetos para realizar una tarea que ninguno de ellos la puede realizar por sí solo.<br>Command, Iterator, State, Strategy |

Los patrones nos ayudan a:
- Encontrar los objetos apropiados
- Determinar la granularidad de los objetos
- Especificar las interfaces (métodos) e implementaciones (cuerpo del método) de los objetos
- Favorecer la reutilización a través de la composición de objetos

## Patrones de Creación

***Abstract Factory**

Crea diferentes familias de objetos. El uso más común es el de creación de interfaces gráficas de distinto tipo, para lo cual se suele combinar con los patrones Singleton y Adapter. La estructura típica del patrón **Abstract Factory** es la siguiente:
☑ La definición de interfaces para la familia de productos *genéricos* (ej: ventana, menú, botón...)
☑ Implementación de las interfaces de los productos para cada una de las distintas familias concretas.
☑ La definición de los métodos de creación de los productos genéricos en la interfaz de la fábrica (ej: construir_ventana, construir_menú, construir_botón...) cuyo tipo de retorno serán las interfaces genéricas.
☑ Implementación de una fábrica para cada una de las familias concretas (ej: fábrica_gtk, fábrica_qt).
El patrón **Abstract Factory** está aconsejado cuando se prevé la inclusión de nuevas familias de productos, pero puede resultar contraproducente cuando se añaden nuevos productos o cambian los existentes.

***Builder**

El patrón builder(Constructor) es usado para permitir la creación de una variedad de objetos complejos desde un objeto fuente(Producto), el objeto fuente se compone de una variedad de partes que contribuyen individualmente a la creación de cada objeto complejo a través de un conjunto de llamadas a interfaces comunes de la clase Abstract Builder, centralizando dicha creación en un único punto, para que el mismo proceso de creación pueda crear diferentes representaciones.

**Factory Method**

*Propósito*: Centralizar en una clase constructora la creación de objetos de un subtipo de un tipo determinado, ocultando al usuario la casuística para elegir el subtipo que crear.
Las clases principales en este patrón son el *creador* y el *producto*. El creador necesita crear instancias de productos, pero el tipo concreto de producto no debe ser forzado en las subclases del creador, porque entonces las posibles subclases del creador deben poder especificar subclases del producto para utilizar.
*Aplicabilidad*: clase que no puede prever la clase de objetos que va a crear.

***Prototype**

El patrón Prototype (Prototipo), tiene como finalidad crear nuevos objetos duplicándolos, clonando una instancia creada previamente. Este patrón propone la creación de distintas variantes del objeto que nuestra aplicación necesite, en el momento y contexto adecuado. Toda la lógica necesaria para la decisión sobre el tipo de objetos que usará la aplicación en su ejecución debería localizarse aquí. Una copia significa otra instancia del objeto. El único requisito que debe cumplir este objeto es suministrar la prestación de clonarse. Cada uno de los objetos prototipo

debe implementar el método Clone(). Este patrón es motivo donde en ciertos escenarios es preciso abstraer la lógica que decide qué tipos de objetos utilizará una aplicación, de la lógica que luego usarán esos objetos en su ejecución.

**Singleton**
*Propósito*: Garantizar que una clase sólo tenga una instancia y proporciona un punto de acceso global a ella.
*Aplicabilidad*: debe existir una única instancia de la clase accesible globalmente.
*Consecuencias*:

- Acceso controlado a una única instancia
- Evitar el uso de variables globales
- Manejar las subclases de la clase singleton

*Ventaja*: permite un acceso controlado a una única instancia.

## Patrones Estructurales

***Adapter**
Se utiliza para adaptar o transformar una interfaz en otra, de tal modo que una clase que no pudiera utilizar la primera, haga uso de ella a través de la segunda. *Adapter* permite a las clases trabajar juntas, lo que de otra manera no podría hacerlo debido a sus interfaces incompatibles.
Se usa cuando:
☑ Se desea usar una clase existente, y su interfaz no se iguala con la necesitada.
☑ Se desea crear una clase reusable que coopera con clases no relacionadas, es decir, las clases no tienen necesariamente interfaces compatibles.

***Bridge**
Desacopla una abstracción de su implementación, de manera que ambas puedan ser modificadas independientemente sin necesidad de alterar por ello la otra. También llamado Handle/Body.
Se usa cuando:
☑ Se desea evitar un enlace permanente entre la abstracción y su implementación. Esto puede ser debido a que la implementación debe ser seleccionada o cambiada en tiempo de ejecución.
☑ Cambios en la implementación de una abstracción no deben impactar en los clientes, es decir, su código no debe tener que ser recompilado.
☑ Se desea compartir una implementación entre múltiples objetos (quizá usando contadores), y este hecho debe ser escondido a los clientes.

***Composite**
Permite tratar objetos compuestos como si se tratase de uno simple, gracias a la composición recursiva y a una estructura en forma de árbol. Esto simplifica el tratamiento de los objetos creados, ya que al poseer todos ellos una interfaz común, se tratan todos de la misma manera.

Problema que soluciona. Ejemplo
Imaginemos que necesitamos crear una serie de clases para guardar información acerca de una serie de figuras que serán círculos, cuadrados y triángulos. Además necesitamos poder tratar también grupos de imágenes porque nuestro programa permite seleccionar varias de estas figuras a la vez para moverlas por la pantalla. En principio tenemos las clases *Círculo*, *Cuadrado* y *Triángulo*, que heredarán de una clase padre que podríamos llamar *Figura* e implementarán todas la operación **pintar()**. En cuanto a los grupos de *Figuras* podríamos caer en la tentación de crear una clase particular separada de las anteriores llamada *GrupoDeImágenes*, también con un método **pintar()**.
Problema: Esta idea de separar en clases privadas componentes (figuras) y contenedores (grupos) tiene el problema de que, para cada uno de los dos atributos, el método **pintar()** tendrá una implementación diferente, aumentando la complejidad del sistema.

***Decorator**
Responde a la necesidad de añadir dinámicamente funcionalidad a un Objeto. Esto nos permite no tener que crear sucesivas clases que hereden de la primera incorporando la nueva funcionalidad, sino otras que la implementan y se asocian a la primera.

Aplicabilidad
☑ Añadir objetos individuales de forma dinámica y transparente
☑ Responsabilidades de un objeto pueden ser retiradas
☑ Cuando la extensión mediante la herencia no es viable
☑ Hay una necesidad de extender la funcionalidad de una clase, pero no hay razones para extenderlo a través de la herencia.
☑ Hay una necesidad de extender dinámicamente la funcionalidad de un objeto y quizás quitar la funcionalidad extendida

**Facade**
*Propósito*: Proveer de una interfaz unificada simple para acceder a una interfaz o grupo de interfaces de un subsistema. Es intermediario entre un cliente y una interfaz o grupo de interfaces más complejas.
*Consecuencias*:
- Oculta a los clientes los componentes del subsistema.
- Promueve el débil acoplamiento entre el subsistema y los clientes.

*Ventajas*:
☑ Hacer una biblioteca de software más fácil de usar y entender, ya que *facade* implementa métodos convenientes para tareas comunes
☑ Hacer el código que usa la librería más legible, por la misma razón
☑ Reducir la dependencia de código externo en los trabajos internos de una librería, ya que la mayoría del código lo usa *Facade*, permitiendo así más flexibilidad en el desarrollo de sistemas;
☑ Envuelve una colección mal diseñada de APIs con un solo API bien diseñado
Facade debe utilizarse para crear clases sencillas, no clases que "sirvan para todo" o "lo hagan todo".

***Flyweight**
Reduce la redundancia cuando gran cantidad de objetos poseen idéntica información.
Problema que soluciona. Ejemplo
Necesitamos representar gráficamente muchas pelotas idénticas que rebotan en los bordes de una ventana, así que creamos una clase que tenga por atributos las coordenadas, el radio y el color con que se dibujará la pelota.
Problema: Aunque las coordenadas son distintas, como queremos que nuestras pelotas sean iguales, el radio y el color se repetirán en cada instancia, desperdiciando memoria. Crear una clase PelotaFlyweight, que contendrá la información común (radio y color) y otra clase PelotaConcreta, que contendrá las coordenadas concretas de cada pelota y una referencia a un objeto de tipo PelotaFlyweight.
Al crearse instancias de PelotaConcreta, se les deberá proveer de referencias a la instancia de PelotaFlyweight adecuada a nuestras necesidades. En este caso solamente tendríamos una instancia de PelotaFlyweight, puesto que hemos dicho que todas nuestras pelotas tienen el mismo radio y color, pero pensando en un ejemplo en el que tuviéramos varios grupos de pelotas, y dentro de cada uno de los cuales se compartieran el radio y el color, se puede utilizar Flyweight conjuntamente con el patrón Factory, de tal modo que este último, en el momento en que se le soliciten instancias de PelotaConcreta con determinadas características (mismo radio y color que el solicitado), compruebe si ya existe un PelotaFlyweight con ese radio y color, y devuelva esa referencia o, en caso de que no exista, la cree y la registre. El patrón Factory se encargaría de gestionar los PelotaFlyweight existentes.

***Proxy**
Se utiliza como intermediario para acceder a un objeto, permitiendo controlar el acceso a él.
Problema que soluciona. Ejemplo
Necesitamos crear objetos que consumen muchos recursos, pero no queremos instanciarlos a no ser que el cliente lo solicite o se cumplan otras condiciones determinadas. Tenemos un objeto padre Asunto del que heredan otros dos:AsuntoReal y Proxy, todos ellos tienen un método petición(). El cliente llamaría al método petición() de Asunto, el cual pasaría la petición a Proxy, que a su vez instanciaría AsuntoReal y llamaría a su petición(). Esto nos permite controlar las peticiones a AsuntoReal mediante el Proxy, por ejemplo instanciando AsuntoReal cuando sea necesario y eliminándolo cuando deje de serlo.

## Patrones de Comportamiento

***Chain of Responsibility**

Permite establecer una cadena de objetos receptores a través de los cuales se pasa una petición formulada por un objeto emisor. Cualquiera de los objetos receptores puede responder a la petición en función de un criterio establecido.
Se utiliza, por ejemplo, cuando en función del estado del sistema las peticiones emitidas por un objeto deben ser atendidas por distintos objetos receptores.

**Command**

Permite solicitar una operación a un objeto sin conocer realmente el contenido de esta operación, ni el receptor real de la misma. Para ello se encapsula la petición como un objeto, con lo que además se facilita la parametrización de los métodos.
*Propósito:*
☑ Encapsula un mensaje como un objeto, con lo que permite gestionar colas o registro de mensaje y deshacer operaciones.
☑ Soportar restaurar el estado a partir de un momento dado.
☑ Ofrecer una interfaz común que permita invocar las acciones de forma uniforme y extender el sistema con nuevas acciones de forma más sencilla
*Aplicaciones:*
☑ Facilitar la parametrización de las acciones a realizar.
☑ Independizar el momento de petición del de ejecución.
☑ Implementar CallBacks, especificando qué órdenes queremos que se ejecuten en ciertas situaciones de otras órdenes. Es decir, un parámetro de una orden puede ser otra orden a ejecutar.
☑ Soportar el "deshacer".
☑ Desarrollar sistemas utilizando órdenes de alto nivel que se construyen con operaciones sencillas (primitivas).

*Ventajas:* Es fácil añadir nuevas ordenes

***Interpreter**

Dado un lenguaje, define una gramática para dicho lenguaje, así como las herramientas (intérprete del lenguaje) necesarias para interpretarlo. Se usa para definir un lenguaje para representar expresiones regulares que representen cadenas a buscar dentro de otras cadenas. Además, en general, para definir un lenguaje que permita representar las distintas instancias de una familia de problemas.

**Iterador**

*Propósito*: Define una interfaz que declara los métodos necesarios para acceder secuencialmente a un grupo de objetos de una colección.
Algunos de los métodos que podemos definir en la Interfase Iterator son: Primero(), Siguiente(), haymas() y elementoactual() .

*Aplicabilidad*: acceder a un objeto agregado sin exponer su representación interna y permitir varias formas de recorrer el agregado.
*Consecuencias*:

- Permite modificaciones en el recorrido de un agregado.
- Simplifican la interfaz del agregado.

*Ventajas:* no es necesario conocer la estructura interna de la colección para poder utilizarla.

***Mediator**

Coordina las relaciones entre sus asociados. Permite la interacción de varios objetos, sin generar acoples fuertes en esas relaciones. Su intención es definir un objeto que encapsule como interactúa un conjunto de objetos. Cuando muchos objetos interactúan con otros objetos, se puede formar una estructura muy compleja, con objetos con muchas conexiones con otros objetos. Para evitar esto el patrón Mediator encapsula el comportamiento de todo un conjunto de objetos en un solo objeto.
Usar el patrón Mediator cuando:

☑ Un conjunto grande de objetos se comunica de una forma bien definida, pero compleja.
☑ Reusar un objeto se hace difícil porque se relaciona con muchos objetos.
☑ El comportamiento de muchos objetos que está distribuido entre varias clases, puede resumirse en una o varias por subclasificación.

***Memento**
Almacena el estado de un objeto (o del sistema completo) en un momento dado de manera que se pueda restaurar a ese punto de manera sencilla. Para ello se mantiene almacenado el estado del objeto para un instante de tiempo en una clase independiente de aquella a la que pertenece el objeto (pero sin romper la encapsulación), de forma que ese recuerdo permita que el objeto sea modificado y pueda volver a su estado anterior. Se usa este patrón cuando se quiere poder restaurar el sistema desde estados pasados y por otra parte, es usado cuando se desea facilitar el hacer y deshacer de determinadas operaciones, para lo que habrá que guardar los estados anteriores de los objetos sobre los que se opere (o bien recordar los cambios de forma incremental).

**State**
*Propósito*: Permitir que un objeto modifique su comportamiento cada vez que cambie su estado interno.
*Aplicabilidad:* se utiliza cuando el comportamiento de un objeto cambia dependiendo del estado del mismo.
*Consecuencias*:

- Coloca todo el comportamiento asociado a un estado particular de la clase
- Ayuda a evitar estados inconsistentes
- Incrementa el número de objetos

*Ventaja*: se eliminan las sentencias case utilizando polimorfismo y permite el crecimiento de la lógica de negocio en función de los cambios de estado.

***Observer**
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. Desacopla la clase de los objetos clientes del objeto, aumentando la modularidad del lenguaje, así como evitar bucles de actualización (espera activa o polling).
Este patrón suele observarse en los marcos de interfaces gráficas orientadas a objetos, en los que la forma de capturar los eventos es suscribir 'listeners' a los objetos que pueden disparar eventos.

**Strategy**
Permite disponer de varios métodos para resolver un problema y elegir cuál utilizar en tiempo de ejecución. Mantiene un conjunto de algoritmos de los que el objeto cliente puede elegir aquel que le conviene e intercambiarlo según sus necesidades. Los distintos algoritmos se encapsulan y el cliente trabaja contra un objeto contexto o Context. Como hemos dicho, el cliente puede elegir el algoritmo que prefiera de entre los disponibles o puede ser el mismo objeto Context el que elija el más apropiado para cada situación. Cualquier programa que ofrezca un servicio o función determinada, que pueda ser realizada de varias maneras, es candidato a utilizar el patrón Strategy. Puede haber cualquier número de estrategias y cualquiera de ellas podrá ser intercambiada por otra en cualquier momento, incluso en tiempo de ejecución.

**Template Method**
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.
Usando el Template Method, se define una estructura de herencia en la cual la superclase sirve de plantilla ("Template" significa plantilla) de los métodos en las subclases. Una de las ventajas de este método es que evita la repetición de código, por tanto la aparición de errores. Este patrón se vuelve de especial utilidad cuando es necesario realizar un algoritmo que sea común para muchas clases, pero con pequeñas variaciones entre una y otras.

***Visitor**
Permite definir nuevas operaciones sobre una jerarquía de clases sin modificar las clases sobre las que opera.
El patrón visitor también especifica cómo sucede la interacción en la estructura del objeto. En su versión más

sencilla, donde cada algoritmo necesita iterar de la misma forma, el método *accept* de un elemento contenedor, además de una llamada al método *visitor*, también pasa el objeto visitor al método *accept* de todos sus elementos hijos. Este patrón es ampliamente utilizado en intérpretes, compiladores y procesadores de lenguajes, en general.

# Criterios para un buen diseño GUI

☑ **Control de usuario**: La aplicación debe indicar claramente si obtiene el control a costa del usuario. Esto se logra frecuentemente cambiando el apuntador del ratón a un reloj de arena o un indicador de espera, y atrapando cualquier clic del ratón adicional que haga el usuario y que no sea atrapado por el ambiente operativo.
☑ **Sensibilidad**: Si la aplicación está procesando durante mucho tiempo, considere la adición de una escala deslizante para indicar el porcentaje de trabajo que se ha realizado, o distráigalos con algún mensaje informativo en la barra de estado.
☑ **Personalización**: Es muy útil permitir que los usuarios reordenen y redimensionen las columnas en un conjunto de resultados grande.
Los colores pueden usarse como pistas visuales para informar al usuario de si un campo es opcional, requerido o prohibido.
☑ **Dirección**: Cuando se ha seleccionado el objeto deseado puede iniciar una diversidad de acciones. Cada acción que puede realizar debe estar disponible ya sea en un elemento de menú o en un botón de comando.
☑ **Consistencia**: Una aplicación de negocios deberá ser consistente con el mundo en que los usuarios viven y trabajan diariamente.
El analista tiene dos alternativas, personalizar la aplicación para mapear la cultura histórica de cada lugar o aplicar reingeniería al negocio para obtener un consenso sobre la terminología y los procesos del negocio.
Las aplicaciones de negocios también deberán ser consistentes con el aspecto, sensación y lenguaje de otras aplicaciones.
☑ **Claridad**: La información que se presenta en la interfaz debe ser comprensible, y el uso de la aplicación debe ser visualmente evidente.
El concepto de consistencia dice que hay que usar el lenguaje del mundo real de los usuarios. El concepto de claridad toma en cuenta que algunos de los lenguajes de los usuarios son surrealistas, y por lo general es resultado del diseño de sistemas heredados.
☑ **Estética**: La composición y disposición de una ventana debe ser visualmente agradable. Estudios sobre la fijación y movimiento del ojo humano muestran que la mayoría de la gente ve primero la parte izquierda superior del centro de la pantalla y hace un barrido en dirección en sentido del reloj.
☑ **Indulgencia**: Un buen diseño de interfaz debe motivar la exploración. El usuario debe sentirse libre para recorrer la aplicación y dar vistazos rápidos en las diversas ventanas y características.
☑ **Limitaciones humanas**: En una interfaz gráfica de usuario no se requiere memorizar. Cada comando o acción posible debe estar disponible ante el usuario por medio de un botón, elemento del menú o icono.
Conforme se incrementa la cantidad de temas diferentes manejados por una ventana, disminuye la habilidad de la persona para usarla en forma efectiva. La complejidad de la ventana también es inversamente proporcional a la habilidad del programador para mantener y probar la aplicación a lo largo del tiempo.

# Cohesión de ventanas

Niveles de cohesión, ordenados desde el más deseado hasta el menos deseado:
☑ **Funcional**: Una ventana o conjunto de ventanas funcionalmente cohesivas manejan un evento a nivel negocio.
La ventaja de las ventanas funcionalmente cohesivas es que se tienen ventanas eficientes y especializadas que son menos complejas y, por lo tanto, más fáciles de usar.
El potencial de reutilización también se incrementa en gran medida.
La desventaja de la cohesión funcional es que frecuentemente incrementa la cantidad de ventanas de la interfaz.
☑ **Secuencial**: Es aquella en donde todos los eventos están agrupado debido a que suceden en secuencia.
La recompensa de la cohesión secuencial es que se mapea muy de cerca con el flujo de trabajo manual. La cohesión secuencial es adecuada para las ventanas que manejan tareas altamente repetitivas en donde la *velocidad de captura* y la *economía de tecleado* son los principales vectores de calidad.

La desventaja de este tipo de ventana es que ahora es una mezcla muy compleja de código no relacionado.
☑ **Comunicativa o comunicacional**: Es aquella en la que los eventos han sido agregados porque afectan al mismo objeto.
☑ **Procedural**: Organiza las tareas de acuerdo a la descripción de trabajo de un usuario particular.
☑ **Temporal**: La cohesión temporal sucede cuando los eventos están agrupados en una ventana porque suceden al mismo tiempo.
☑ **Lógica**: Sucede cuando los eventos están agrupados para compartir el mismo código. El propósito es prevenir la proliferación de ventanas de selección de reporte similares por toda la aplicación, y se pretende que reduzca el efecto de propagación de onda de los cambios que puedan afectar varias ventanas.
☑ **Coincidental:** Cualquier relación que haya entre eventos en una ventana *coincidentalmente cohesiva* existe solamente en la mente del programador que la creo.

# Diseño de Interfaces Externas

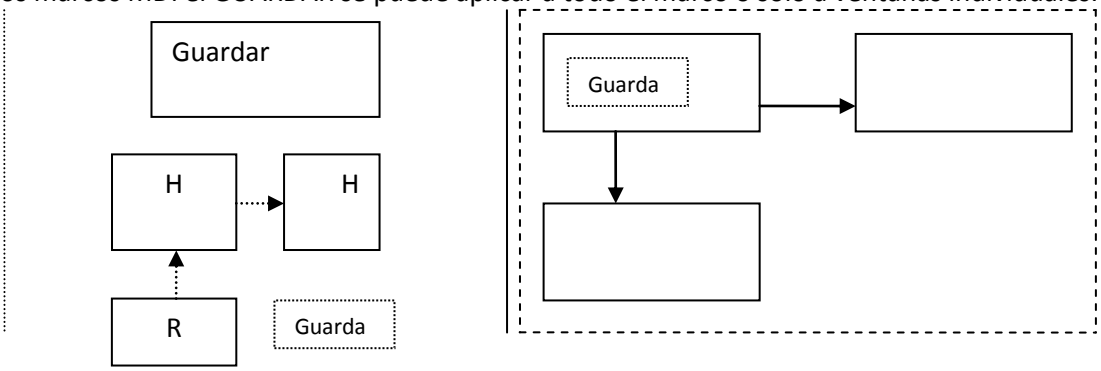
El diseño de la interfaz externa especifica el aspecto, sensación y comportamiento de la parte del sistema que ve el usuario. Es un refinamiento del prototipo de interfaz que incluye elementos tales como el tipo de ventana, navegación, especificación de botones y definición de la unidad de trabajo adecuada para el usuario. El diseño externo del sistema depende en gran medida de las convenciones de la industria para la tecnología de interfaz seleccionada.

Los componentes de un diseño de interfaz externa para una interfaz gráfica de usuario incluyen:

**Panoramas del sistema y aplicación**: El panorama del sistema es una descripción escrita del objetivo del negocio y la función del sistema completo.

**Diagramación de navegación de ventanas**: El objetivo de esta iteración del prototipo es determinar el tipo de ventana correcto y las rutas de navegación, y definir la unidad de trabajo adecuada para el usuario.
La ruta de navegación entre dos ventanas se muestra trazando una flecha de una ventana a la siguiente. Una flecha con una sola punta indica que no se requiere el regreso a la ventana que llama. Se usa una flecha de doble punta para mostrar que el usuario está obligado a regresar a la ventana que llama después de terminar su tarea.
☑ **Ventana principal o de aplicación**: Puede traslapar y ser traslapada por otras ventanas. Es movible, lo que significa que puede ser arrastrada a un lado para que revele lo que se encuentra bajo ella. Es redimensionable por el usuario y puede ser minimizada como un icono en el escritorio.
☑ **Ventana desplegable o de aparición súbita**: Se abre desde una ventana existente, a la cual se le menciona como ventana *madre*. A diferencia de una ventana principal, no puede ser traslapada por su *madre*.
La ventana desplegable también puede existir después de que se cierra la ventana *madre*. Esto puede ser problemático cuando la ventana desplegable contiene cualquier dato que no se ha guardado y que es requerido por la ventana *madre*.
☑ **Ventana hija**: También se abre a partir de una ventana *madre*, y al igual que la ventana desplegable, una ventana hija no puede ser traslapada por la ventana *madre*. La principal diferencia entre ellas es que una ventana hija no puede ser arrastrada afuera de los confines del marco de la ventana *madre*.
La ventana hija no puede existir después de que la ventana *madre* se haya cerrado. Cuando se cierra la ventana *madre* también se cerrarán todas sus ventanas hijas.
☑ **Ventana de Respuesta**: El tipo de ventana con comportamiento más restrictivo es la *ventana de respuesta*, ya que obtiene el enfoque cuando es abierta y no lo libera sino hasta que se cierra.
☑ **Marco MDI / hoja MDI**: El marco MDI es un espacio de trabajo redimensionable y autocontenido que opera en una forma muy similar a la ventana principal, pero tiene integradas algunas características especiales. El marco MDI viene con un menú. Las ventanas que se abren dentro del marco son llamadas *hojas MDI*. Las hojas MDI se comportan como ventanas hijas. No pueden ser arrastradas fuera de los confines del marco MDI y se minimizan a un icono dentro del marco. El usuario es capaz de tener varias aplicaciones abiertas al mismo tiempo, sin tener peligro de entrelazar ventanas no relacionadas.
☑ **Carpeta con fichas o pestañas**: La *carpeta con fichas* es un tipo de ventana especial que se comporta en forma muy similar a un conjunto en capas de hojas MDI o ventanas hijas. La apariencia del marco de ventana ha sido

modificada para que se parezca a una carpeta de archivo antigua con una pestaña saliente sobre la cual está escrito el nombre. Las carpetas con ficha son útiles cuando hay demasiados elementos de datos a desplegar en una ventana y el tema a presentar se divide en forma lógica en áreas de temas distintos.
☑ **Unidad de trabajo**: Ésta se define como qué tanto se decide permitirles avanzar antes de que deban guardar su trabajo en la base de datos. El diagrama de navegación permite que se juegue con la colocación de GUARDAR. Si se aplica sólo a una ventana, puede ser anotado directamente en la ventana. Si se aplica al trabajo realizado en varias ventanas abiertas, se puede trazar una línea punteada alrededor del conjunto completo de ventanas cuyo contenido es registrado con GUARDAR.
En los marcos MDI el GUARDAR se puede aplicar a todo el marco o sólo a ventanas individuales.




**Especificación de ventanas:** cada ventana del diagrama de navegación de ventanas, requiere una especificación. La especificación de ventana le sirve tanto a los usuarios como a los desarrolladores. Hay cuatro partes de una especificación externa de ventanas:
☑ **Disposición de ventanas:** para cada ventana del diagrama de navegación, una disposición de ventana muestra la manera en que esta aparecerá ante el usuario
☑ **Descripción de la ventana:** el texto que acompaña a cada disposición de ventana define claramente la función y características de esta, en forma tal que un usuario potencial pueda comprender el comportamiento del diseño.
☑ **Mini especificación de ventana:** la especificación técnica de la ventana define el comportamiento para la apertura y cierre de la ventana y la activación y ejecución de cada botón, control y elemento de menú.
☑ **Especificación de campo:** define los campos y ediciones asociadas para todos los datos que aparecen en la ventana. La especificación de campo debe incluir mini especificación sobre la manera en que se adquieren los datos, listando nombres de tablas, nombres de columnas, cómo unir tablas y describiendo la manera de aplicar cualquier criterio de selección.

# *Especialización de Base de datos

La necesidad de usar una BD proviene de la capacidad limitada de la memoria primaria, por lo tanto las BD se almacenan sobre *almacenamiento secundario*, proveyendo maneras eficientes de acceder a los objetos. Otra necesidad es la de almacenar elementos que *persistan* más que la ejecución del programa.
Un DBMS es quien se ocupa del almacenamiento para que el usuario no tenga que molestarse en saber cómo se almacenan los objetos. El programador de aplicación solo quiere una vista lógica de la BD y no quiere tomar parte de cómo se hace el almacenamiento físico.

Un DBMS también debería proveer:
☑ **Concurrencia**: permite a múltiples usuarios trabajar con una BD común simultáneamente.
☑ **Recuperación**: si ocurre una falla, el DBMS debería ser capaz de volver la BD a un estado consistente anterior.
☑ **Facilidad de consulta**: debería soportar una manera fácil de acceso a los datos en la BD.

La especialización de la BD significa que un producto DBMS se integra al desarrollo de un sistema. Las propiedades típicas que indican una necesidad de un DBMS en el desarrollo de una aplicación son:
☑ La información necesita ser persistente.
☑ Más de una aplicación compartiendo los datos.
☑ La estructura de la información tiene un gran número de instancias.
☑ Búsquedas complejas en la estructura de la información.

☑ Generación avanzada de informes desde la información almacenada.
☑ Manipulación de transacciones de usuario.
☑ Un registro para el reinicio del sistema.

Por lo general los objetos que van a ser persistentes son los de entidad, quienes retienen información que sobrevive los casos de uso. En algunas aplicaciones también hay necesidad de almacenar otro tipo de objetos, como aquellos para manejar interfaces.

## DBMS's Relacionales

*Aspectos de problema:* En una BD relacional, la información se almacena en tablas. Los tipos a ser usados en las tablas, son mayormente los tipos primitivos tales como char, int, etc. Esto trae algunos problemas a nuestra ambición de almacenar objetos.
En primer lugar, solo los datos pueden almacenarse, no el comportamiento y en segundo lugar, solo los tipos de datos primitivos pueden almacenarse, y no las estructuras complejas de nuestros objetos.

Cuando un lenguaje de programación se conecta a un DBMS, un número de problemas proviene. El primero es que en nuestro sistema toda la información se almacena en los objetos, por tanto, necesitamos transformar nuestra estructura de información de objeto a una estructura orientada a tablas. A este problema se lo llama *problema de impedancia.*
El problema de impedancia trata a un otro problema: crea un fuerte acoplamiento entra la aplicación y el DBMS. Para hacer que el diseño sea mínimamente afectado por el DBMS, tan pocas partes de nuestro sistema como sea posible, deberían saber sobre la interface del DBMS.
Un tercer problema, es cómo expresar la herencia en la BD. Dado que todos los objetos son persistentes, tienen esta capacidad (papel) en común, el comportamiento de persistencia también puede heredarse.
*Objetos en tablas:* La primera cosa para hacer es determinar que clases y que variables de la clase deben almacenarse en la BD. Cada una de estas clases, será representada por lo menos, por una tabla en la BD.

**1.** Asignar una tabla para la clase.
**2.** Cada atributo (primitivo) se transformará en una columna en la tabla. Si el atributo es complejo (es decir, debe componerse de tipos de DBMS), o agregamos una tabla adicional para el atributo, o distribuimos el atributo en varias columnas de la tabla de la clase.
**3.** La columna de la clave primaria será el identificador único de la instancia, llamado identificador.
**4.** Cada instancia de la clase ahora será representada por una fila en esta tabla.
**5.** Cada asociación de conocimiento con una cardinalidad mayor que 1 (por ejemplo, [0..N]) se transformará en una nueva tabla. Esta nueva tabla conectará las tablas que representan los objetos que van a ser asociados.
*Herencia:* Los objetos que deberían ser persistentes, deberían ser mapeados en tablas de la BD. Si las clases se heredan unas a otras, hay principalmente dos formas de resolver esto:
**1.** Los atributos heredados se copian a todas las tablas que representan las clases descendientes. Ninguna tabla representará la clase abstracta. Esta alternativa es normalmente la más rápida, dado que no hay ningún joining o búsqueda en varias tablas. Sin embargo el tamaño de la BD aumentará dado que las columnas heredadas deben duplicarse. Además, si los cambios ocurren en los atributos heredados, estos cambios afectaran a las tablas de todas las clases descendientes.
**2.** La clase abstracta está en su propia tabla, a la que las tablas de las clases descendiente hacen referencia. Esta alternativa incluye menos redundancia, pero puede conducir a problemas si los identificadores son comunes en la tabla de la clase abstracta (por ej, si tenemos una clase abstracta persona y estudiante y profesor son descendientes, entonces ninguna persona puede ser ambos, profesor y estudiante al mismo tiempo). Además, si queremos cambiar la clave primaria, estos cambios deben hacerse en varios lugares. Generalmente esta alternativa es la recomendada.

---

Problema: obtener datos de DBMS y traerlos a la memoria local durante el uso de la aplicación.
Solución: utilizar un *framework de persistencia*: es un conjunto de tipos de propósito general, reutilizable y extensible, que proporciona funcionalidad para dar soporte a los objetos persistentes. Debería proporcionar funciones para almacenar y recuperar los objetos y commit y rollback de las transacciones.

*Frameworks:*
☑ Es un conjunto extensible de clases e interfaces que colaboran para proporcionar servicios de la parte central e invariable de un subsistema lógico.
☑ Contiene clases concretas y (especialmente) abstractas que definen las interfaces a las que ajustarse, interacciones de objetos en las que participar, y otras variantes.
☑ Puede requerir que el usuario del framework defina subclases de las clases del framework para utilizar, adaptar y extender los servicios del framework.
☑ Tiene clases abstractas que podrían contener tanto métodos abstractos como concretos.
☑ Confía en el **Principio de Hollywood**: "*No nos llame, nosotros lo llamaremos"*.
☑ Ofrecen un alto grado de reutilización, mucho más que con clases individuales.

**Conceptos**
*OID*: identificador único del objeto.
*Materialización y desmaterialización*: la materialización es el acto de transformar una representación de datos no OO de un almacenamiento persistente de objetos.
*Caché*: los servicios persistentes almacenan en una caché los objetos materializados por razones de rendimiento.

## Patrones para DBMS

**Representación de objetos como tablas**
Definir una tabla por cada clase de objetos persistentes. Los atributos simples de los objetos contienen datos primitivos.
**Identificador de objeto**
Propone asignar in identificador de objeto a cada registro. Un OID se puede representar mediante una clase OID.
**Representación de las relaciones de los objetos en tablas**
Asociaciones:
- 1a1**:** colocar una referencia externa en una o ambas tablas.
- 1aN**:** los N tendrán una referencia externa (FK) al 1.
- NaN**:** crear una tabla intermedia.

**Fachada**
Se podría utilizar este patrón para proporcionar una interfaz para recuperar o almacenar objetos en la base de datos.
**Conversor de Base de Datos**
Propone crear una clase que sea responsable de materializar y desmaterializar un objeto almacenado. Se define una clase diferente que establece la correspondencia para cada clase de los objetos persistentes. Puede haber distintos tipos de conversores para cada mecanismo de almacenamiento.
**Template method**
La idea es crear un método (plantilla) en una superclase, y este invoca a otros métodos los cuales se redefinen en una subclase.
**Gestión de Caché**
Propone que sea el conversor de base de datos el responsable de mantener este caché y así aumentar el rendimento.
**State**
Crea clases de estado para cada estado, que implementan una interfaz común, que se encarguen de realizar las tareas correspondientes a cada uno de los estados.
**Command**
Define una clase por cada tarea que implemente una interfaz común. Ej.: tareas de insert, update.

# Diseño Arquitectónico

El proceso de diseño inicial para identificar estos subsistemas y establecer un marco de trabajo para el control y comunicación de los subsistemas se llama *diseño arquitectónico* y lo que produce este proceso de diseño es una descripción de la *arquitectura del software*.
Representa un vínculo crítico entre el diseño y los procesos de ingeniería de requerimientos. Una especificación no debe incluir ninguna información del diseño.
La descomposición arquitectónica es necesaria para estructurar y organizar la especificación.
Existen tres ventajas sobre la especificación del diseño y la documentación de una arquitectura de software:
**1. Comunicación entre stakeholders**: La arquitectura es una presentación de alto nivel del sistema que es utilizada como punto de discusión por un rango de stakeholders diferentes.
**2. Análisis del sistema**: Hacer explícita la arquitectura del sistema en una etapa inicial del desarrollo del sistema, significa que se debe llevar a cabo cierto tipo de análisis. Las decisiones en el diseño arquitectónico tienen un efecto profundo sobre cuándo el sistema puede cumplir los requerimientos críticos como el desempeño, la fiabilidad y la mantenibilidad.
**3. Reutilización a gran escala**: Una arquitectura del sistema en una descripción compacta y manejable de cómo se organiza el sistema y cómo interoperan los componentes. La arquitectura se puede transferir a lo largo de los sistemas con requerimientos similares y así poder reutilizar software a gran escala. Es posible desarrollar arquitecturas de líneas de productos donde la misma arquitectura se utilice en carios sistemas relacionados.

Las siguientes actividades son comunes para todos los procesos de diseño arquitectónico:
**1. Estructuración del sistema**: el sistema se estructura en varios subsistemas principales, donde un subsistema es una unidad de software independiente. Se identifican las comunicaciones entre los subsistemas.
**2. Modelado de control**: Se establece un modelo general de las relaciones de control entre las partes del sistema.
**3. Descomposición modular**: Cada subsistema identificado se descompone en módulos. El arquitecto debe decidirse sobre los tipos de módulos y sus interconexiones.

***Diferencia entre un subsistema y un módulo:***
1. Un *subsistema* es un sistema por sí mismo cuya operación no depende de los servicios suministrados por otros subsistemas. Los subsistemas se componen de módulos y tienen interfaces definidas que se utilizan para la comunicación con otros subsistemas.
2. Un *módulo* es por lo regular un componente del sistema que suministra uno o más servicios a otros módulos. Utiliza los servicios suministrados por otros módulos. Por lo general no se considera un sistema independiente. Los módulos están compuestos de varios componentes simples del sistema.

La arquitectura del sistema afecta al desempeño, la robustez, la distribución y la mantenibilidad del sistema. Por lo tanto, el estilo particular y la estructura elegida para una aplicación pueden depender de los requerimientos no funciones del sistema:
**1. Desempeño**: Si el desempeño es un requerimiento crítico, esto sugiere que la arquitectura se debe diseñar para localizar las operaciones críticas dentro de un número reducido de subsistemas con poca comunicación, hasta donde sea posible, entre estos subsistemas. Esto significa utilizar componentes de grano grueso más que de grano fino para reducir los componentes de comunicación.
**2. Seguridad**: Si el seguridad es un requerimiento crítico, esto sugiere utilizar una estructura en capas para la arquitectura con los recursos más críticos protegidos por las capas más internas y con un alto nivel de validación de la seguridad aplicado a esas capas.
**3. Protección**: Si la protección es un requerimiento crítico, esto sugiere que la arquitectura se debe diseñar de tal forma que las operaciones relacionadas con la protección se localicen en un solo subsistema o en un número reducido de subsistemas. Esto reduce los costos y los problemas de validación y hace posible crear sistemas de protección relacionados.
**4. Disponibilidad**: Si la disponibilidad es un requerimiento crítico, esto sugiere que la arquitectura debe diseñarse para incluir componentes redundantes de tal forma que se posible reemplazar y actualizar los componentes sin detener el sistema.

**5. Mantenibilidad**: Si la mantenibilidad es un requerimiento crítico, esto sugiere que la arquitectura se debe diseñar utilizando componentes autocontenidos de grano fino que puedan cambiarse con facilidad. Los productores de datos deben estar separados de los consumidores y las estructuras de datos compartidas deben evitarse.

# Estructuración del Sistema

***El modelo de depósito***

Existen dos formas fundamentales para lograr esto:

1. Todos los datos compartidos se ubican en una base de datos central que puede ser accedida por todos los subsistemas. Un modelo del sistema basado en una de datos compartida se denomina algunas veces *modelo de depósito*.
2. Cada subsistema tiene su propia base de datos. Los datos se intercambian con otros subsistemas pasando mensajes entre ellos.

| Ventajas | Desventajas |
|---|---|
| Es una forma eficiente de compartir grandes cantidades de datos. No existe la necesidad de transmitir datos explícitamente de un subsistema a otro. | Los subsistemas deben estar acordes al modelo de depósito de datos. Esto es un compromiso entre las necesidades especificas de cada herramienta. |
| Los subsistemas que producen datos no necesitan saber cómo son utilizados esos datos por otros subsistemas. | Si se genera un gran volumen de información, será difícil evolucionar si se ha acordado un modelo de datos. Traducir esto a un nuevo modelo será costoso; puede ser difícil o incluso imposible. |
| Las actividades como las de respaldo, seguridad, control de acceso y recuperación de errores están centralizadas. | Varios subsistemas tienen diferentes requerimientos de políticas de seguridad, recuperación y respaldo. |
| El modelo de compartición es visible a lo largo del esquema de depósito. | Es difícil distribuir el depósito en varias máquinas. |

***El modelo Cliente -Servidor***

El modelo arquitectónico Cliente-Servidor es un modelo de sistemas distribuido que muestra cómo los datos y el procesamiento se distribuyen a lo largo de varios procesadores.

Los componentes principales de este modelo son:

1. Un conjunto de servidores independientes que ofrecen servicios a otros subsistemas.
2. Un conjunto de clientes que llaman a los servicios ofrecidos por los servidores. Por lo general, éstos son subsistemas. Existen instancias de un programa cliente que se ejecuta de forma concurrente
3. Una red que permite a los clientes acceder a estos servicios. En principio, ésta no es realmente necesaria pesito que tanto los clientes como los servidores podrían ejecutarse en una sola máquina.

La ventaja más importante del modelo Cliente-Servidor es que es una arquitectura distribuida. Cada servidor debe tener responsabilidad de las actividades de administración de datos como la de respaldo y recuperación.

***El modelo de máquina abstracta***

Organiza un sistema en una serie de capas cada una de las cuales suministra un conjunto de servicios. Cada capa define una *máquina abstracta* cuyo lenguaje de máquina se utiliza para implementar el siguiente nivel de la máquina abstracta.

Esta arquitectura también es cambiable y portable. Si su interfaz se preserva, una capa se puede reemplazar por otra capa. Cuando las interfaces de las capas cambian, sólo se afecta la capa adyacente.

Una desventaja del enfoque de capas es que estructurar a los sistemas de esta forma es difícil.

**Modelos de control**

Se pueden identificar dos enfoques generales para el control:
**1. Control centralizado**: Un subsistema tiene completa responsabilidad para controlar, iniciar y detener otros subsistemas. También puede regresar el control a toros subsistemas, pero espera que se le regrese esa responsabilidad de control. Se divide en dos clases:
☑ **El modelo de llamada-retorno**: Éste es modelo familiar de subrutina descendente en el que el control inicia en la parte superior de una jerarquía y, por medio de las llamadas a la subrutina, pasa a los niveles inferiores del árbol. El modelo de subrutina sólo es aplicable a sistemas secuenciales.
☑ **El modelo del administrador**: Éste se aplica a los modelos concurrentes. Un componente del sistema se designa como un sistema administrador y controla el inicio, la detención y la coordinación de toros procesos del sistema.
**2. Control basado en eventos**: En lugar de que la información de control esté contenida en un sistema, cada subsistema puede responder a eventos generados en el exterior. Estos eventos pueden provenir de otros subsistemas o del entorno del sistema.
La distinción entre un evento y una entrada simple es que la duración del evento está fuera del control del proceso que maneja ese evento.

**Modelos de Transmisión**: En estos modelos, un evento se transmite a todos los subsistemas. Cualquier subsistema que pueda manejar ese evento responde a él.

**Modelos dirigidos por interrupciones**: Éstos se utilizan en sistemas de tiempo real donde las interrupciones externas son detectadas por un controlador de interrupciones. Después éstas se pasan a algún otro componente para su procesamiento.

Los modelos de transmisión son efectivos para integrar subsistema distribuidos a los lardo de diferentes computadoras de una red. Los modelos dirigidos por interrupciones se utilizan en sistemas de tiempo real, con requerimientos rígidos.
Los subsistemas deciden qué eventos requieren y el controlador de eventos y mensajes asegura que estos eventos sean envidados a dichos subsistemas.
La ventaja de este enfoque de transmisión es que la evolución es relativamente sencilla. Un nuevo subsistema que maneje clases particulares de eventos se puede integrar registrando sus eventos en el controlador de eventos. Cualquier subsistema puede activar otros subsistemas sin conocer su nombre o ubicación. Los subsistemas se pueden incrementar en máquinas distribuidas. Esta distribución es transparente para otros subsistemas.
La desventaja de este modelo es que los subsistemas no saben si los eventos se manejarán, ni cuándo lo harán. Cuando un subsistema genera un evento, no sabe qué otros subsistemas han registrado un interés en ese evento.

***Descomposición modular***
1. Un modelo orientado a objetos: El sistema se descompone en un conjunto de objetos que se comunican entre ellos.
2. Un modelo de flujo de datos: El sistema se descompone en módulos funcionales que afectan entradas de datos y las transforman, de alguna manera, en datos de salida.

En un modelo orientado a objetos, los módulos son objetos con estado privado y con operaciones definidas sobre ese estado. En el modelo de flujo de datos, los módulos son transformaciones funcionales. En ambos casos, los módulos se implementan como componentes secuenciales o como procesos.

***Modelo de objetos***
Un modelo orientado a objetos de una arquitectura de sistema estructura el sistema en un conjunto de objetos débilmente acoplados con interfaces bien definidas.

| Ventajas | Desventajas |
|---|---|
| La implementación de objetos se puede modificar sin afectar a otros objetos. A menudo los objetos representan entidades del mundo real por lo que la estructura del sistema es bastante comprensible. Debido a estas entidades del mundo real se utilizan en varios sistemas, los objetos se pueden reutilizar. | Para utilizar los servicios, los objetos deben hacer referencia explícita al nombre y a la interfaz de otros objetos. Si se requiere cambiar una interfaz para satisfacer los cambios del sistema, se debe evaluar el efecto de ese cambio sobre todos los usuarios del objeto cambiado. |

***Modelo de flujo de datos***

Las ventajas de esta arquitectura son:

1. Permite la reutilización de transformaciones.
2. Es intuitiva puesto que muchas personas visualizan su trabajo en términos de procesamiento de entradas y salidas.
3. Permite que el sistema evolucione al agregar nuevas transformaciones en forma directa.
4. Es sencilla de implementar ya sea como un sistema concurrente o uno secuencial.

La desventaja principal del modelo proviene de la necesidad de un formato para transferir datos que sea reconocido por todas las transformaciones.
Los sistemas interactivos son difíciles de describir utilizando un modelo de flujo de datos debido a la necesidad de procesar una gran cantidad de datos.

***Arquitecturas de dominio específico***

Existen dos tipos de modelos arquitectónicos de dominio específico:

1. Modelos genéricos que son abstracciones de varios sistemas reales
2. Modelos de referencia que son modelos abstractos y describen a una clase mayor de sistemas.

Los modelos genéricos se pueden utilizar directamente en el diseño. Los modelos de referencia se utilizan normalmente para comunicar conce3ptos del dominio y comparar las posibles arquitecturas.
Los modelos genéricos se derivan de forma ascendente a partir de los sistemas existentes, mientras que los modelos de referencia se derivan “de forma descendente”.
Los modelos de referencia no son considerados como la ruta para la implementación. La función principal es servir como medio de comparación de diversos sistemas en un dominio.

## Sistemas Distribuidos

***Seis características importantes de los sistemas distribuidos:***

**1. Compartición de recursos**: Un sistema distribuido permite compartir los recursos de hardware y software como los discos, impresoras, archivos, compiladores, etc., asociados a diversas computadoras de la red.
**2. Apertura:** La apertura de un sistema es el grado al cual se puede extender agregándole nuevos recursos no propietarios. Los sistemas distribuidos son sistemas abiertos que incluyen hardware y software de diferentes compañías.
**3. Concurrencia**: En un sistema distribuido, varios procesos operan al mismo tiempo en diferentes computadoras de la red.
**4. Escalabilidad**: Son escalables en el sentido de que las capacidades del sistema se pueden incrementar agregando nuevos recursos para cubrir las nuevas demandas de dicho sistema.
**5. Tolerancia a fallos**: La disponibilidad de varias computadoras y el potencial para replicar la información significa que los sistemas distribuidos pueden tolerar algunas fallas de hardware y software.
La pérdida completa del servicio ocurre sólo cuando existe una falla en la red

**6. Transparencia**: Esconder al usuario la naturaleza distribuida del sistema. Una meta del diseño del sistema es que los usuarios tengan acceso transparente a los recursos y no tengan necesidad de conocer algo sobre la distribución del sistema.

Los sistemas distribuidos tienen algunas desventajas:
☑ **Complejidad**: Los sistemas distribuidos son más complejos que los centralizados. Esto hace más difícil comprender las propiedades emergentes y probar estos sistemas.
☑ **Seguridad**: El sistema se puede acceder desde varias computadoras diferentes y el tráfico sobre la red debe estar sujeto a inspecciones. Esto hace más difícil administrar la seguridad en un sistema distribuido
☑ **Manejabilidad**: Las diversas computadoras de un sistema pueden ser de diferentes tipos o ejecutar diferentes sistemas operativos. Las fallas en una máquina se pueden propagar a otras máquinas con consecuencias inesperadas.
☑ **Impredecibilidad**: Ésta depende de la carga total del sistema, de su organización y de la carga de la red.

Existen dos tipos genéricos de arquitecturas de sistemas distribuidos:
**1. Arquitecturas cliente-servidor**: En este enfoque, el sistema se visualiza como un conjunto de servicios que se suministran a los clientes que lo utilizan. Los servidores y los clientes se tratan de forma diferente en estos sistemas.
**2. Arquitecturas de objetos distribuidos**: En este caso, no existe diferencia entre los servidores y los clientes y el sistema se visualiza como un conjunto de objetos que interactúan cuya ubicación es irrelevante. No existe diferencia entre un proveedor y un usuario de estos servicios.

Un sistema distribuido requiere cierto software que pueda administrar estas partes, asegurar que se comuniquen y que intercambien datos. El término *middleware* se utiliza para referirse a este software.

***Arquitecturas multiprocesador***
El modelo más simple de un sistema distribuido es un sistema multiprocesador en el cual consiste de varios procesos diferentes que pueden ejecutarse en procesadores diferentes.

***Arquitecturas cliente-servidor***
Los clientes deben saber qué servidores están disponibles, pero por lo general no conocen la existencia de otros clientes. Los clientes y servidores son procesos diferentes, dentro de un modelo lógico de una arquitectura cliente-servidor distribuida (modelo de tres capas).

Las arquitecturas cliente-servidor de dos capas tienen dos formas:
**1. Modelo de cliente delgado**: En este modelo, todo el procesamiento de la aplicación y la administración de datos se lleva a cabo en el servidor. El cliente es responsable de ejecutar el software.
**2. Modelo de cliente grueso**: En este modelo, el servidor sólo es responsable de la administración de datos. El software en el cliente implementa la lógica de la aplicación y las interacciones con el usuario del sistema.

Una gran desventaja del modelo de cliente delgado es que coloca una gran carga de procesamiento tanto en el servidor como en la red.
En contraste, el modelo de cliente grueso utiliza un alto poder de procesamiento disponible en las PCs modernas y distribuye tanto el procesamiento lógico de la aplicación como la presentación del cliente.
Aunque modelo de cliente grueso utiliza distribuye el procesamiento de forma más efectiva que uno de cliente delgado, la administración del sistema es más complejo.

El problema fundamental con un enfoque cliente-servidor de dos capas es que las tres capas lógicas –presentación, procesamiento de la aplicación, administración de datos– deben asociarse a dos sistemas de cómputo. Pueden existir problemas ya sea de escalabilidad y desempeño si se elige el modelo de cliente delgado o problemas de administración del sistema si se utiliza el modelo cliente grueso.

Una arquitectura de software cliente-servidor de tres capas no implica necesariamente que existan tres sistemas de cómputo conectado a la red. Una única computadora servidor puede ejecutar tanto el procesamiento y la administración de datos de la aplicación como servidores lógicos diferentes.
Los sistemas de múltiples capas se utilizan cuando las aplicaciones necesitan acceder y utilizar datos de diferentes bases de datos.

***Arquitecturas de objetos distribuidos***
Los clientes reciben servicios de los servidores y no de los otros clientes, los servidores pueden actuar como clientes recibiendo servicios de otros servidores pero no solicitan servicios de los clientes; y los clientes deben conocer los servicios que ofrece cada uno de los servidores y cómo contactar a estos servidores.
Los objetos se distribuyen a lo largo de varias computadoras sobre una red y se comunica a través de middleware. Éste provee un conjunto de servicios que permiten la comunicación, agregación y destrucción de los objetos del sistema. Su papel es proveer una interfaz transparente entre los objetos.
Las ventajas de este modelo en una arquitectura de sistemas distribuidos son:
☑ Permite al diseñador del sistema retrasar las decisiones sobre dónde y cómo se deben suministrar los servicios. Los objetos proveedores de servicios se pueden ejecutar en cualquier nodo de la red. Por lo tanto, la distinción entre los modelos de cliente grueso y delgado son irrelevantes puesto que no existe necesidad de decidir por adelantado dónde se localizan los objetos lógicos de la aplicación.
☑ Ésta es una arquitectura de sistemas muy abierta que permite agregar nuevos recursos si es necesario.
☑ El sistema es flexible y escalable. Se pueden crear diferentes instancias del sistema con el mismo servicio suministrado diferentes o por réplicas de objetos para hacer frente a las diversas cargas del sistema. Si esta carga se incrementa, se pueden cargar nuevos objetos sin perturbar a otros del sistema.
☑ Es posible reconfigurar el sistema de forma dinámica conforme a lo requerido, migrando los objetos a lo largo de la red.

Una arquitectura de objetos distribuidos se utiliza de dos formas en el diseño de sistemas:
1. Como un modelo lógico que permite estructurar y organizar el sistema. En este caso, se debe pensar en cómo proveer de funcionalidad a la aplicación en términos y combinaciones de servicios. Además se debe identificar cómo suministrar estos servicios utilizando varios objetos distribuidos. Los objetos que se diseñan son por lo regular objetos de negocios (de grano grueso) que suministran servicios específicos del dominio.
2. Con un enfoque flexible para la implementación de sistemas cliente-servidor. En este caso, tanto los clientes como los servidores se realizan como objetos distribuidos que se comunican a través de un bus de software.
La principal desventaja de las arquitecturas de objetos distribuidos es que son más complejas para diseñar que los sistemas cliente-servidor.
Es más difícil pensar en el abastecimiento de servicios generales si se carece de experiencia en el diseño y desarrollo de objetos de negocio de grano grueso.