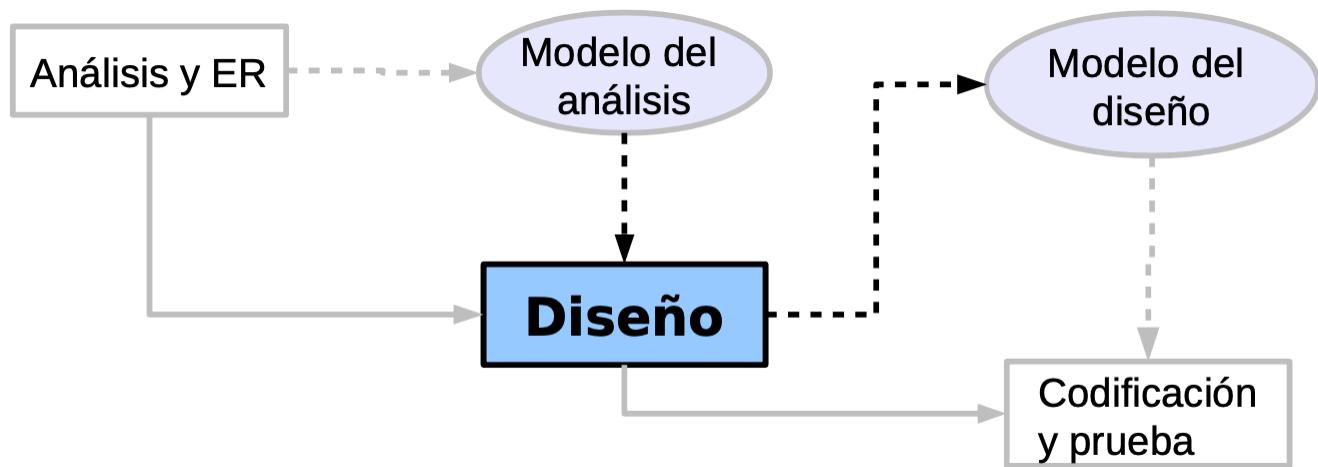


# Tema 3 - Diseño e Implementación

## 3.1 Introducción al diseño

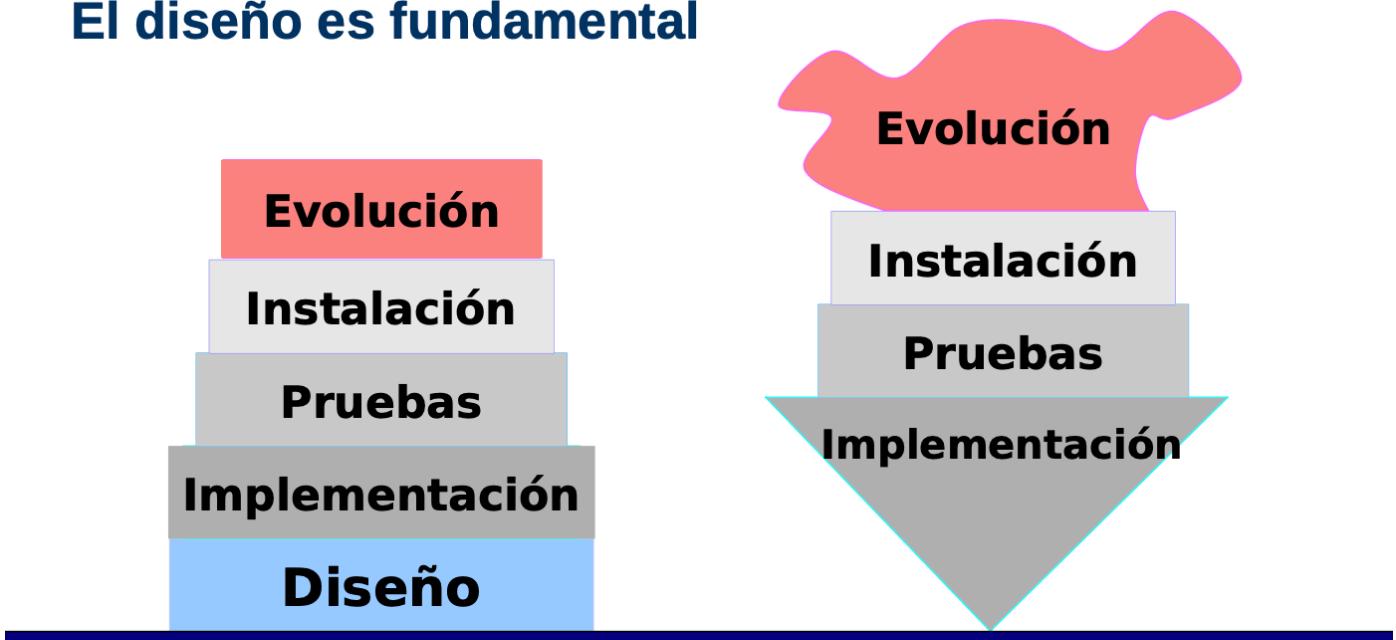
### Definición y características

El diseño es el proceso de aplicar distintas métodos, herramientas y principios con el propósito de definir un dispositivo, proceso o sistema con los suficientes detalles como para permitir su realización física.



El diseño de software es el proceso de aplicar métodos, herramientas y principios de diseño, para traducir el modelo del análisis a una representación del software (modelo del diseño) que pueda ser codificada.

# El diseño es fundamental



**“Con Diseño”**

**“Sin Diseño”**

- El diseño implica una **propuesta de solución** al problema especificado durante el análisis.
- Es una **actividad creativa** apoyada en la experiencia del diseñador.
- Apoyado por principios, técnicas, herramientas, ...
- Es una tarea **clave para la calidad** del producto software.
- **Base para el resto** de las etapas del desarrollo (figura anterior).
- Debe ser un **proceso de refinamiento**.
- El diseño va a garantizar que un programa funcione correctamente.

Hacer que un  
programa funcione



Hacer que funcione  
**CORRECTAMENTE**

## Principios de diseño : Modularidad

Divide y vencerás

"Un sistema software debe estar formado por piezas (Módulos), que deben encajar perfectamente, que interactúan entre sí para llevar a cabo algún objetivo común".

Un **Módulo Software** es una unidad básica de descomposición de un sistema software y representa una entidad o un funcionamiento específico.

Nombre //que lo identifique

**(inicio)**

// contenido del módulo

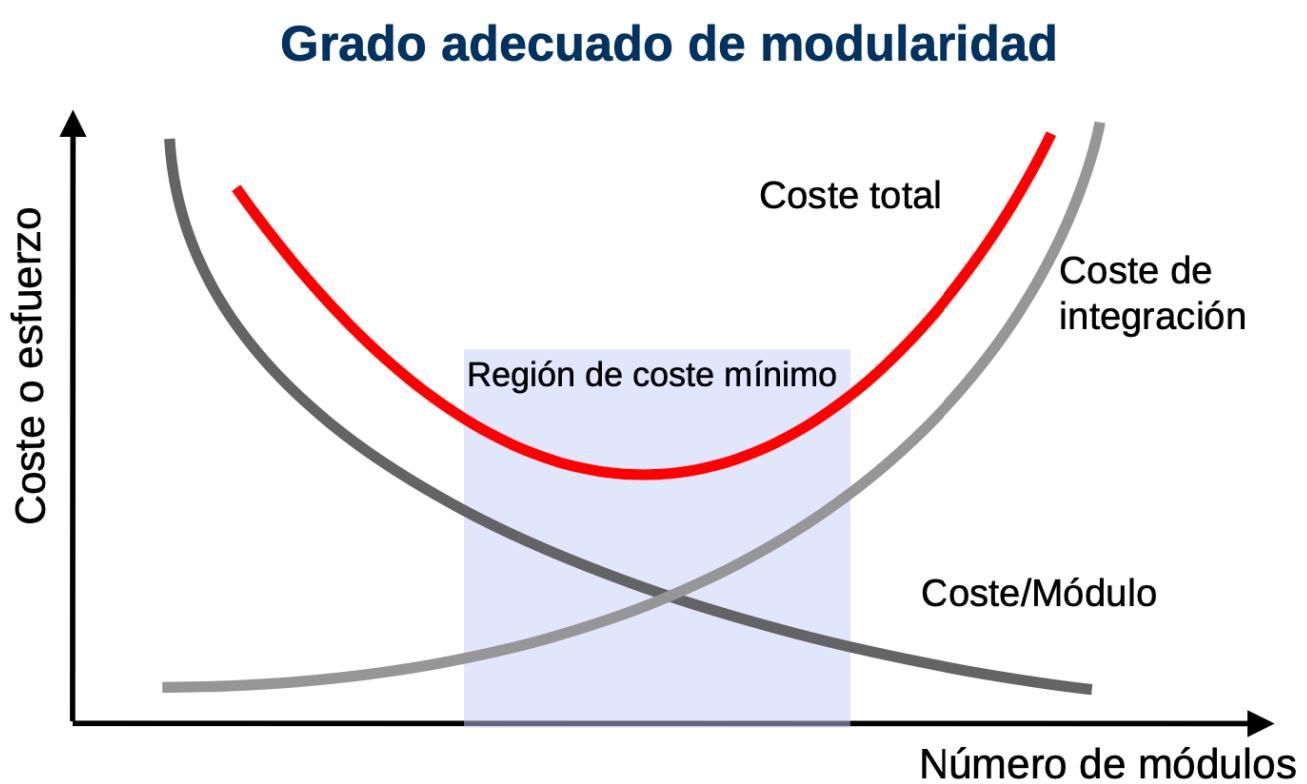
**(fin)**

Pueden ser Módulos: una función, una clase, un paquete...

## Ventajas de la modularidad

Los módulos:

- Son más fáciles de entender y de documentar que todo el subsistema o sistema,
- Facilitan los cambios.
- Reducen la complejidad.
- Proporcionan implementaciones más sencillas.
- Posibilitan el desarrollo en paralelo.
- Permiten la prueba independiente.
- Facilitan el encapsulamiento.



## Principios de diseño : Abstracción

*“Mecanismo que permite determinar qué es relevante y qué no lo es en un nivel de detalle determinado, ayudando a obtener la modularidad adecuada para ese nivel de detalle”*

Mecanismos de abstracción en el diseño:

- Abstracción procedural.
- Abstracción de datos.
- Abstracción de control.

Al proceso de ir incorporando detalles al diseño conforme vayamos bajando el nivel de abstracción se denomina **REFINAMIENTO**, propuesto por N. Wirth en 1971 ([verlo en \[PRES13 página 194\]\]](#)).

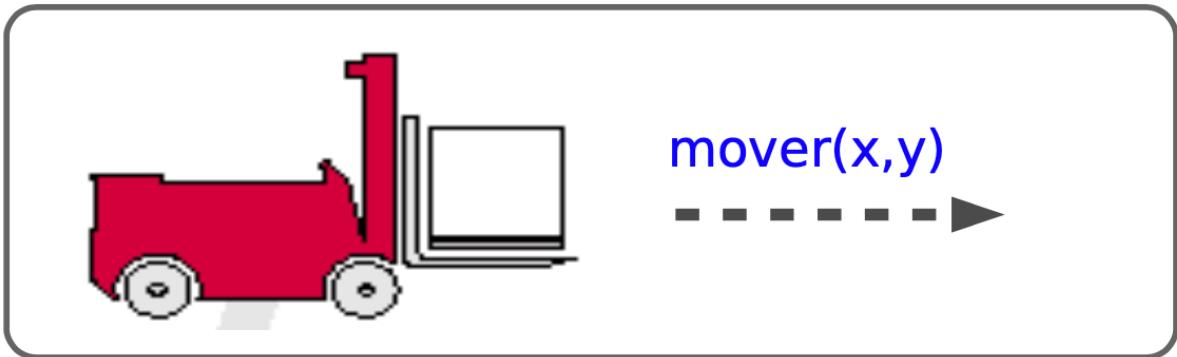
El refinamiento stepwise es una estrategia de diseño propuesta originalmente por Niklaus Wirth [Wir71]. Un programa se elabora por medio del refinamiento sucesivo de los detalles del procedimiento. Se desarrolla una jerarquía con la descomposición de un enunciado macroscópico de la función (abstracción del procedimiento) en forma escalonada hasta llegar a los comandos del lenguaje de programación.

En realidad, el refinamiento es un proceso de *elaboración*. Se comienza con un enunciado de la función (o descripción de la información), definida en un nivel de abstracción alto. Es decir, el enunciado describe la función o información de manera conceptual, pero no dice nada sobre los trabajos internos de la función o de la estructura interna de la información. Después se trae- baja sobre el enunciado original, dando más y más detalles conforme tiene lugar el refinamiento (elaboración) sucesivo.

La abstracción y el refinamiento son conceptos complementarios. La primera permite especificar internamente el procedimiento y los datos, pero elimina la necesidad de que los “extra-ños” conozcan los detalles de bajo nivel. El refinamiento ayuda a revelar estos detalles a medida que avanza el diseño. Ambos conceptos permiten crear un modelo completo del diseño conforme éste evoluciona.

## **1. Abstracción procedural:**

Se abstracta sobre el funcionamiento para conseguir una estructura modular basada en procedimientos.



Nivel de  
detalle muy  
elevado

**mover(x,y)**

*Descripción  
general del  
movimiento  
de la carretilla*

```
void moverLaCarretilla(float x, float y)
```

**//Precondición:** los valores de x e y deben estar dentro de los límites de movimiento de la carretilla.

**//Poscondicion:** la carretilla se ha desplazado desde la posición en la que se encontraba hasta la posición indicada por las coordenadas x e y.

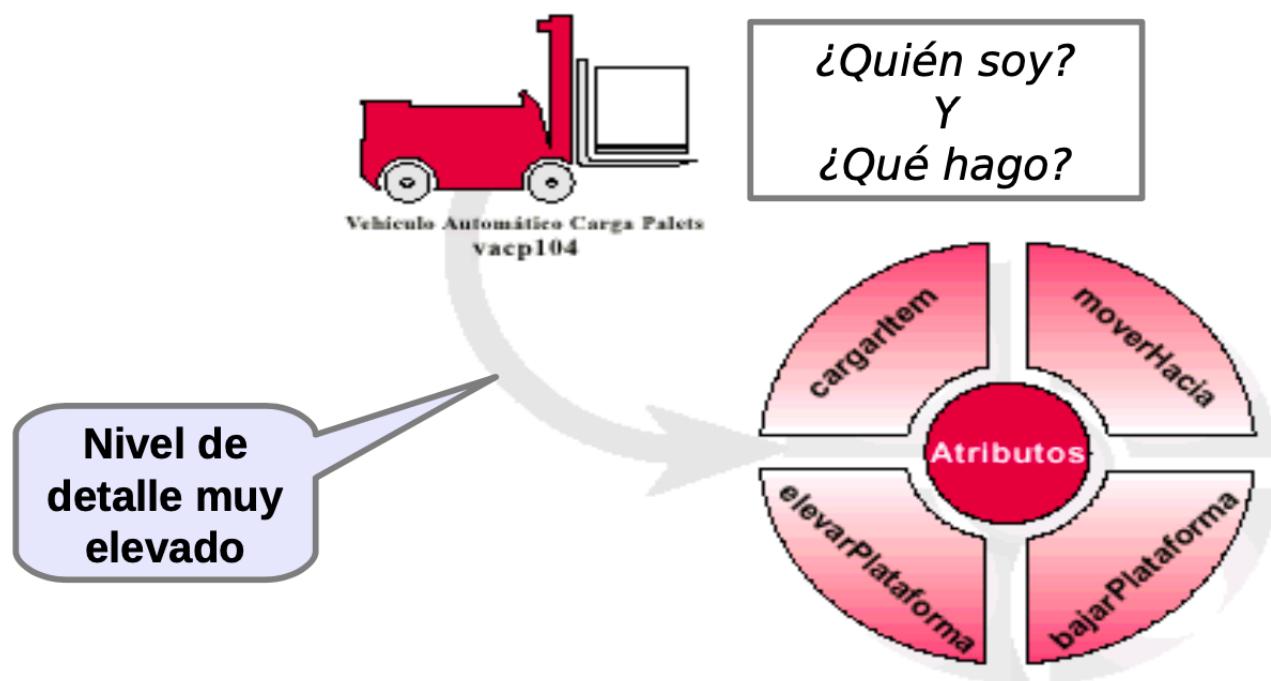
{ “Descripción del algoritmo de movimiento” }

**Especificación  
de la  
abstracción  
procedimental**

**Mediante un proceso de refinamiento llegamos a un nivel de detalle muy bajo o detallado**

## 2. Abstracción de datos:

Se abstrae tanto el funcionamiento como los atributos que definen el estado de una entidad, para obtener una estructura modular basada en el estado y funcionamiento de una entidad u objeto.



## 2. Abstracción de datos

**Nombre:** Carretilla

**Atributos que definen su estado:**

posiciónX:float // posición X del plano en la que se encuentra la carretilla.

posiciónY:float // posición Y del plano en la que se encuentra la carretilla.

pesoMáximo:float // peso máximo que admite la carretilla

posiciónPala:float // posición en la que se encuentra la pala de carga

Mediante un proceso de refinamiento llegamos a un nivel de detalle muy bajo o detallado

**Funcionalidad de los objetos carretilla:**

```
void moverLaCarretilla(x:float, y:float)
```

//Precondición: ....

//Poscondicion: ....

```
void cargarItem(peso:float )
```

//Precondición: ....

//Poscondicion: ....

```
void moverPlataforma(z:float )
```

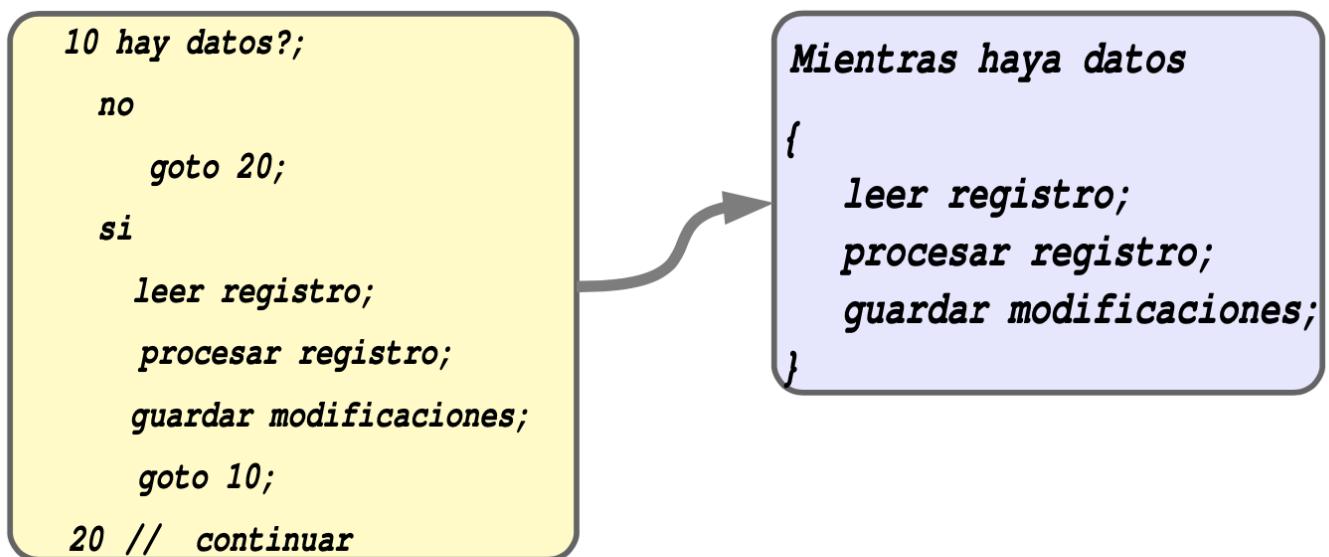
//Precondición: ....

//Poscondicion: ....

¿Cuáles serían sus precondiciones y postcondiciones ?

### 3. Abstracción de control:

Mecanismo que permite abstraer sobre el flujo de control de cualquier proceso en general.



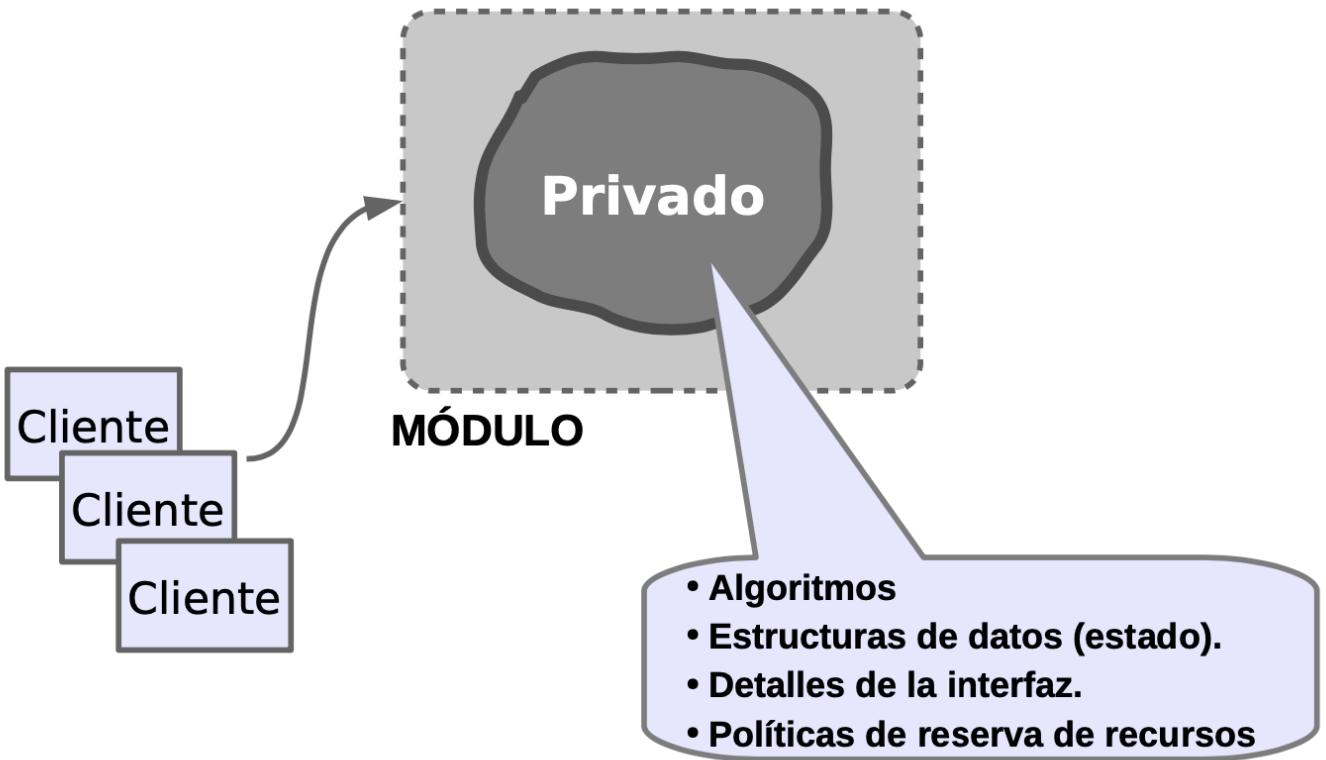
**Otros ejemplos:** semáforos en SO e iteradores sobre colecciones

### Principios de diseño: Ocultamiento de información

“Un módulo debe especificarse y diseñarse de forma que la información (procedimientos y datos) que está dentro del módulo sea inaccesible para otros módulos que no necesiten de esa información”.

Beneficios del ocultamiento de información:

- Reduce la probabilidad de “efectos colaterales”.
- Limita el impacto global de las decisiones de diseño locales.
- Enfatiza la comunicación a través de interfaces controladas.
- Disminuye el uso de datos globales.
- Potencia la modularidad.
- Produce software de alta calidad.



## Principios de diseño: Independencia Modular

*Dos parámetros miden el grado de independencia de un módulo: cohesión y acoplamiento.*

**1. Cohesión:** Grado que tiene un módulo en la realización de **un solo objetivo** y todos sus elementos deben estar ahí para llevar a cabo ese objetivo y ningún otro. Un módulo debe presentar un nivel alto de cohesión.

La alta cohesión proporciona módulos fáciles de entender, reutilizar y mantener.

- Si el módulo es un **procedimiento** verlo en Pfleeger de 2002 páginas 254-260 y PRESS13 página 193.
- Si el módulo es una **clase**, ésta presenta un nivel alto de cohesión si modela un solo concepto abstracto con un pequeño conjunto de responsabilidades íntimamente relacionadas y todas sus operaciones, atributos y asociaciones están para realizarlas”

**2. Acoplamiento:** Medida de interdependencia entre módulos dentro de una estructura de software. Un módulo debe presentar un nivel de acoplamiento, con los demás módulos, lo más bajo posible.

Un grado adecuado de acoplamiento entre módulos es indispensable, hay que:

- Tratar de reducirlo siempre que sea posible.

- Mostrarlo de forma explícita en todos los modelos del diseño.

Si el módulo es un **procedimiento** verlo en Pfleeger de 2002 páginas 254-260 y PRESS13 página 193.

Si el módulo es una **clase**, ésta debería relacionarse (mediante herencia, asociación o dependencia) sólo con las clases que necesite para llevar a cabo sus responsabilidades.

## Herramientas de diseño

Las **herramientas de diseño** son los instrumentos que ayudan a representar los modelos de diseño de software.

Algunas de las más usuales:

- Diagramas de UML: de clase, de interacción, de paquetes, de componentes, de despliegue...
- Cartas de estructura.
- Tablas de decisión.
- Diagramas de flujo de control:
  - Organigramas estructurados
  - Diagramas de NS.
- Lenguajes de diseño de programas (LDP).

## Métodos de diseño

Un **método de diseño** va a permitir obtener diseños de forma sistemática, dándonos las herramientas, las técnicas y los pasos a seguir para llevar a cabo el diseño.

Todo método de diseño **debe poseer**:

- **Principios** en los que se basa.
- **Mecanismos de traducción** del modelo de análisis al modelo de diseño.
- **Herramientas** que permitan representar los componentes funcionales y estructurales.
- **Heurísticas** que nos permitan refinar el diseño. • Criterios para evaluar la calidad del diseño.

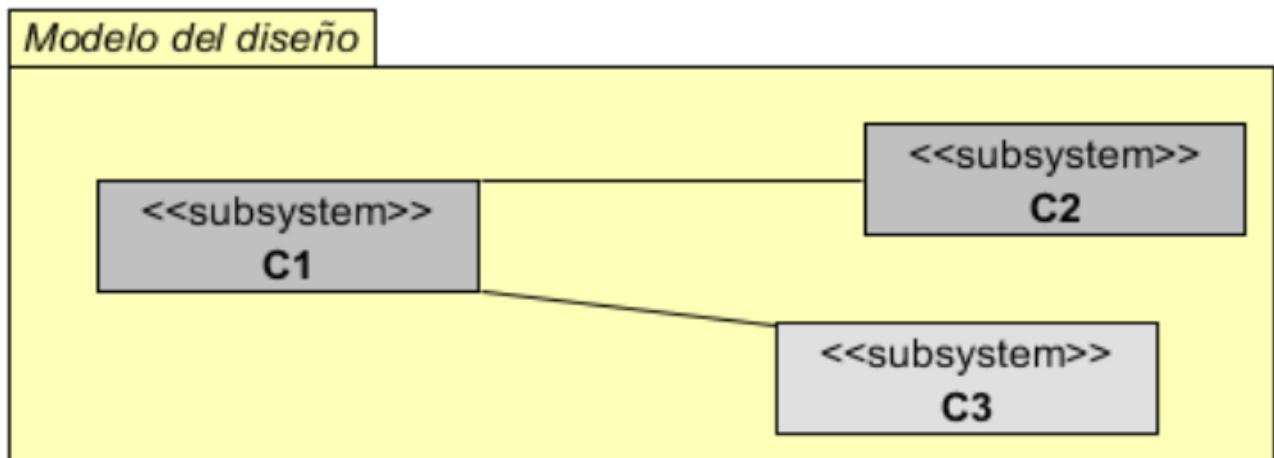
## Principales métodos de diseño

- **SSD** (Diseño estructurado de sistemas).
- **JSD** (Desarrollo de sistemas de Jackson).

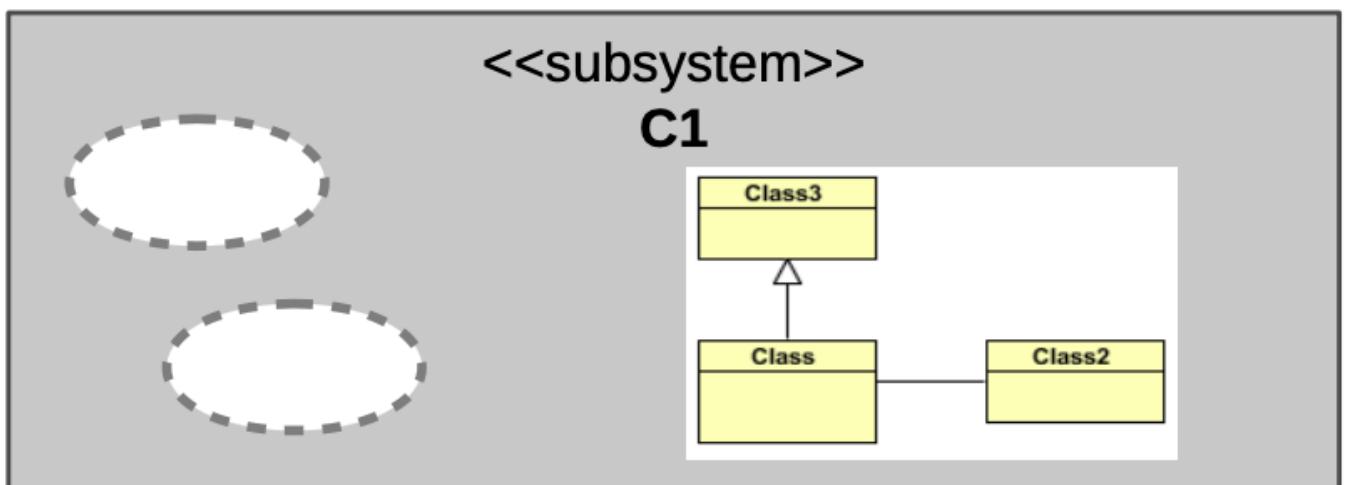
- **ERA** (Entidad–Relación–Atributo).
- **OMT** (Técnicas de modelado de objetos).
- **Metodo de Booch** (Método de diseño basado en objetos)
- **Métodos orientado a objetos:** En la actualidad existe una gran variedad de métodos orientado a objetos, aunque la mayoría de ellos usan como herramienta de modelado UML y como proceso de desarrollo el PU.

## Modelo de diseño

A nivel general está formado por varios **subsistemas** de diseño junto con las **interfaces** que requieren o proporcionan estos subsistemas.

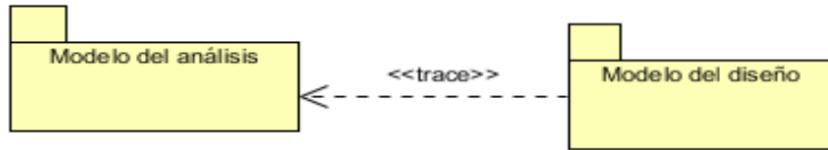


Cada subsistema de diseño a su vez pueden contener diferentes tipos de elementos de modelado del diseño, principalmente **realización de casos de uso-diseño** y **clases de diseño**.



## Relación con el modelo del análisis

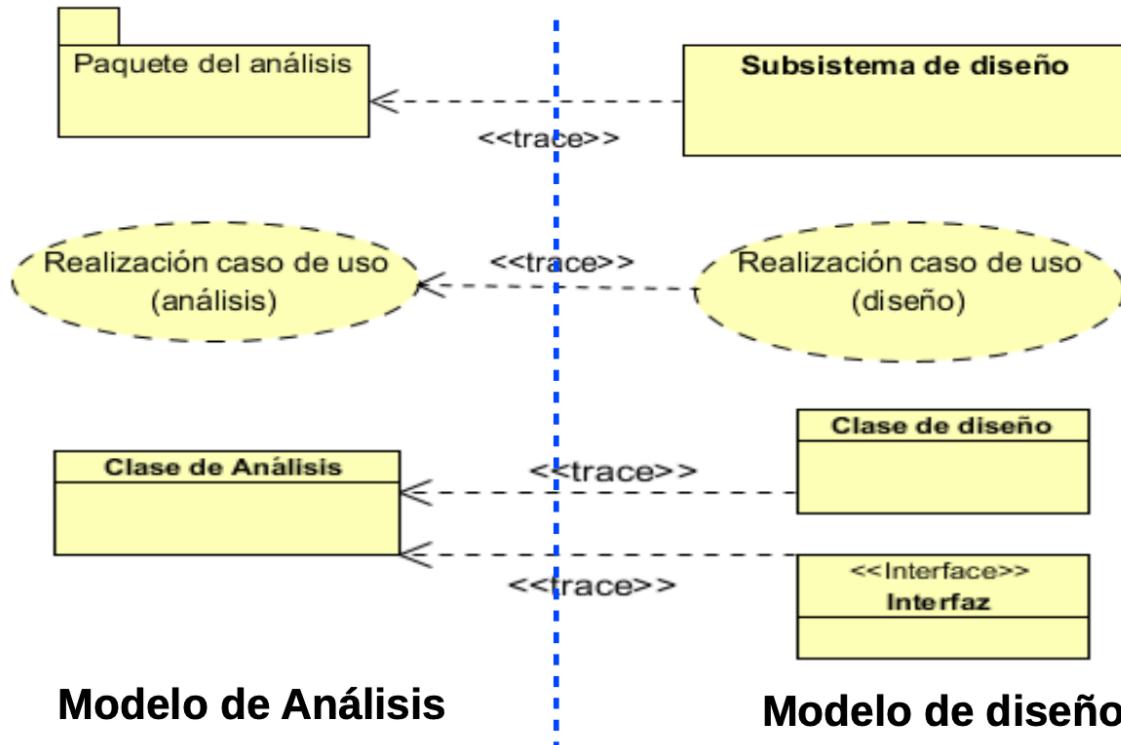
El modelo del diseño puede considerarse como una elaboración/refinamiento del modelo del análisis, en los que todos los artefactos de éste están mejor definidos e incorporan detalles técnicos que permiten su implementación.



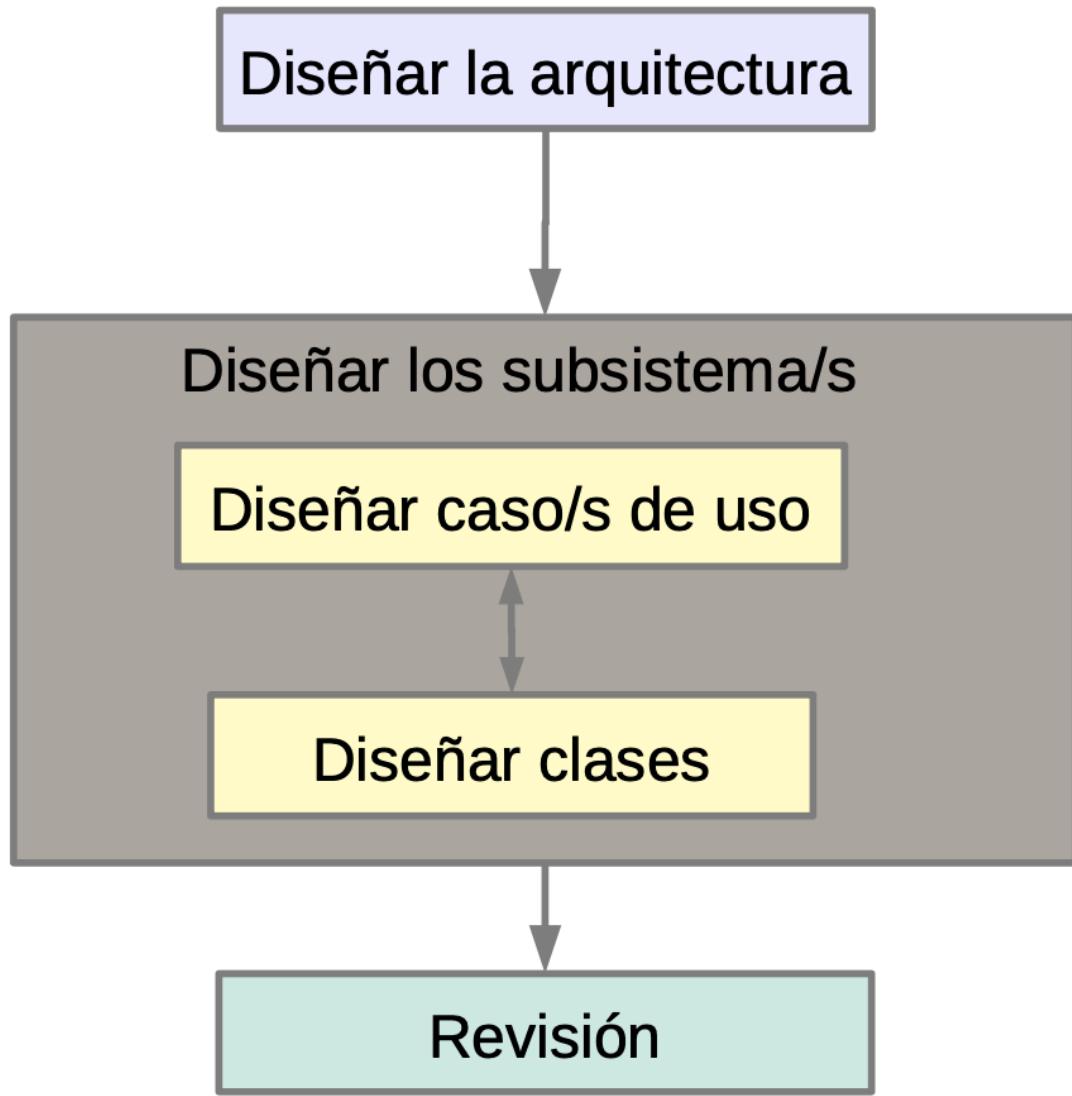
¿Cómo proceder para conseguir la trazabilidad entre modelos?

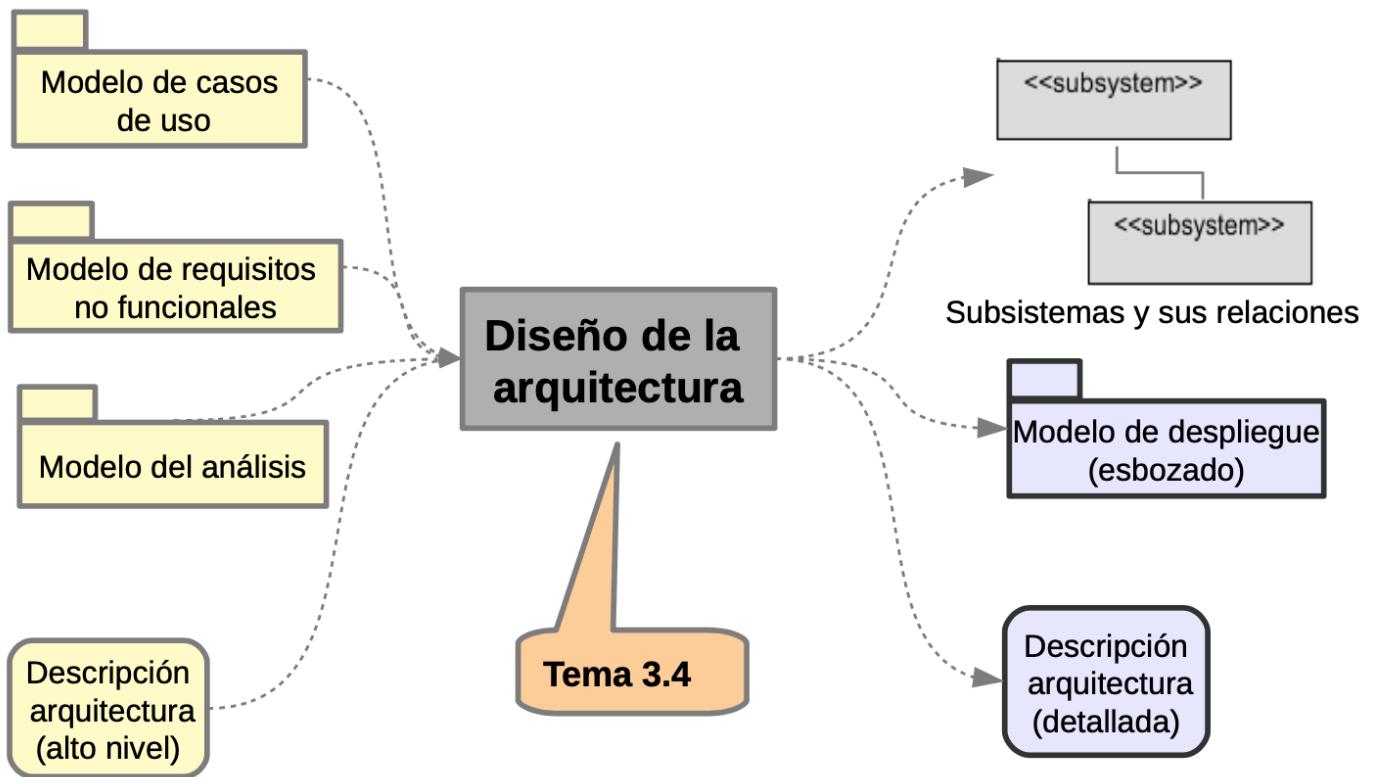
Estrategia	Consecuencias
A) Modificar el modelo del análisis	Se tiene un solo modelo pero se pierde la vista del análisis
B) Igual que A y usar una herramienta que nos recupere el modelo del análisis	Se tiene un solo modelo, pero la vista recuperada puede no ser satisfactoria
C) Congelar el modelo del análisis y hacer una copia para continuar con el diseño	Se tienen dos modelos que no van al mismo ritmo.
D) Mantener los dos modelo	Se tiene dos modelos al mismo ritmo, pero hay una sobrecarga de mantenimiento.

Correspondencia entre los componentes del modelos del análisis y del diseño

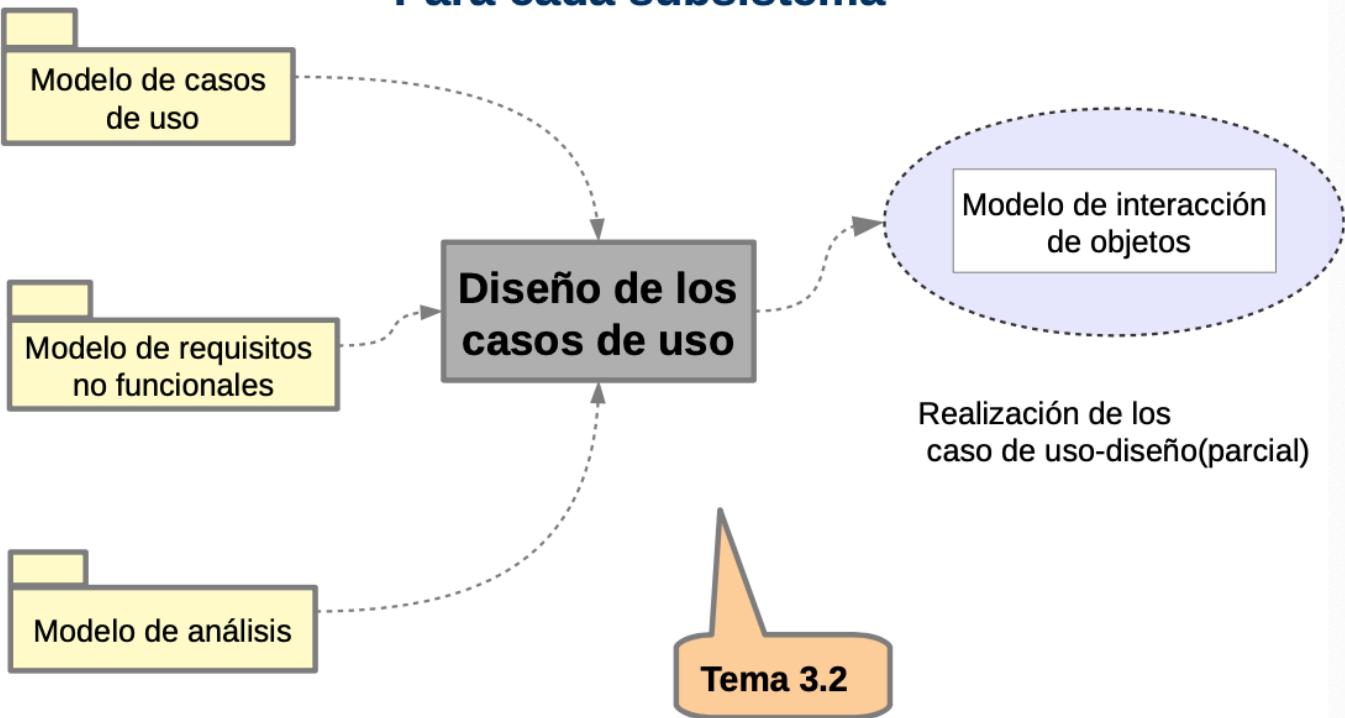


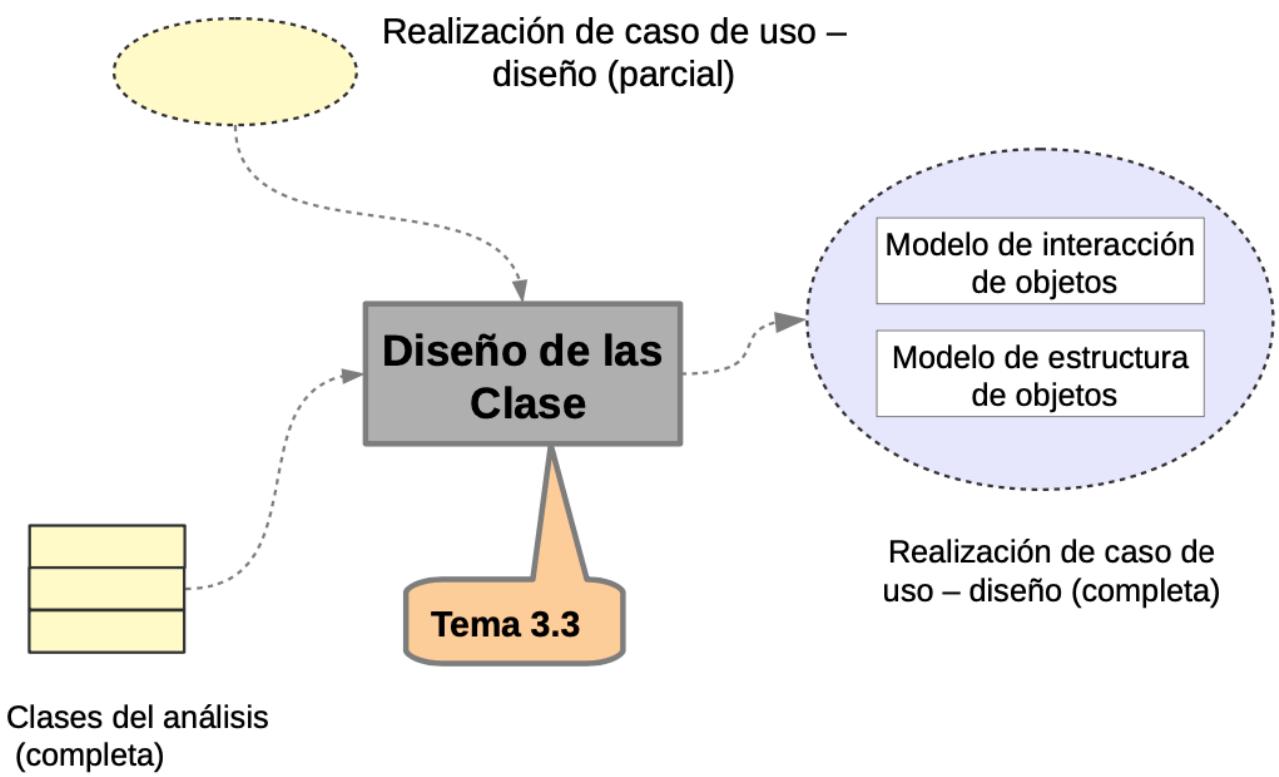
## Tareas del diseño





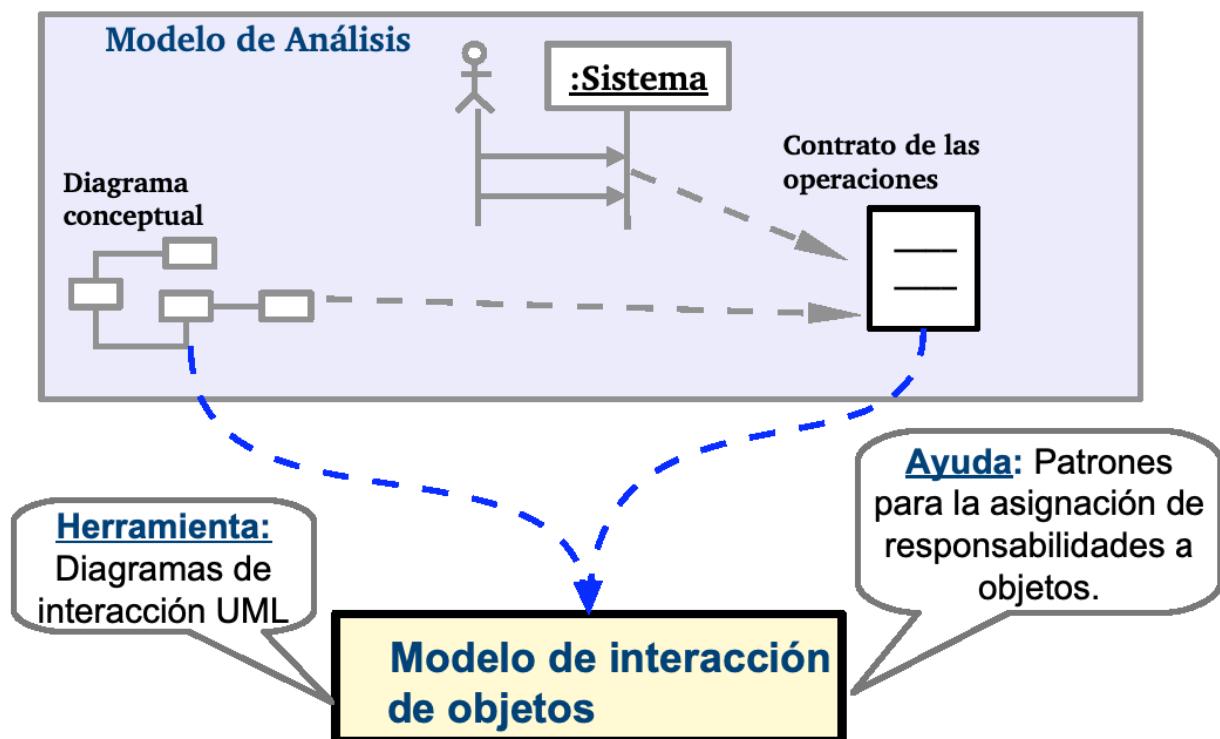
## Para cada subsistema





## 3.2 Diseño de los casos de uso

### Modelo de interacción de objetos



La herramientas para representar el modelo son los diagramas de interacción de UML:

- Diagrama de secuencia.
- **Diagrama de comunicación.**

Estos dos diagramas son semánticamente equivalentes. (Ver seminario Diagrama de comunicación)

## Patrones de diseño de Craig Larman

### Patrón experto en información

**Nombre:** Experto en información

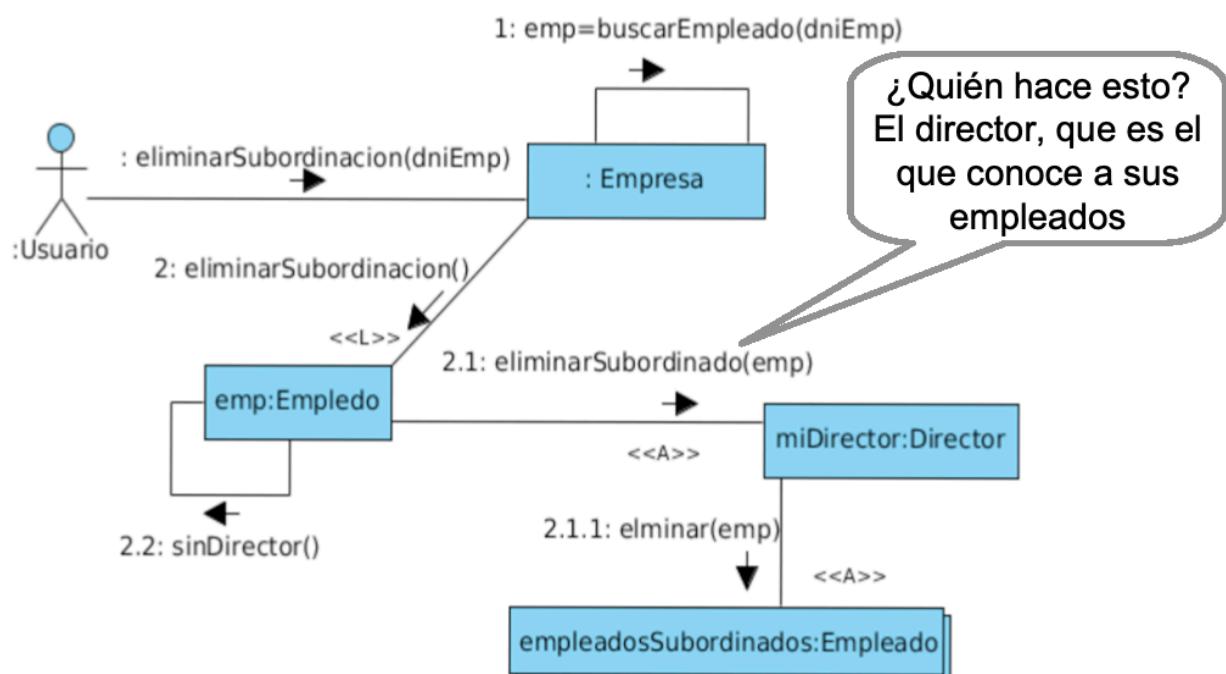
**Problema:** Complejidad en la búsqueda de información y acoplamientos fuertes entre clases en estas búsquedas.

**Solución:** Asignar responsabilidad a la clase que contiene la información necesaria para llevar a cabo la responsabilidad.

**Consecuencias:**

- **Malas:** en ocasiones va en contra de los principios de acoplamiento o de cohesión.
- **Buenas:** mantiene el ocultamiento de la información y distribuye el comportamiento.

Ejemplo:



## Patrón creador

Nombre: Creador

**Problema:** Tener acoplamientos, mala encapsulación y reutilización y poca claridad en la construcción de objetos.

**Solución:** Asignar a la clase B la responsabilidad de crear una instancia de A en los siguientes casos:

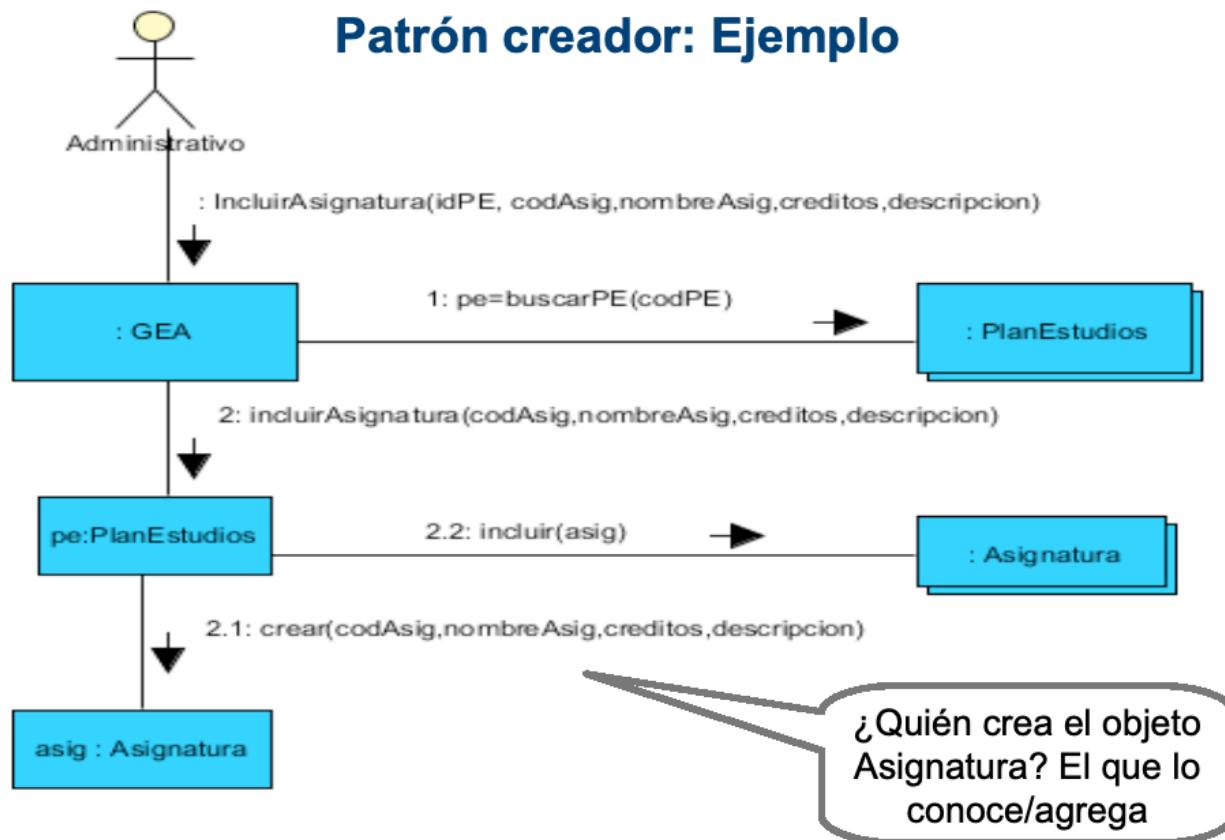
- B **agrega** objetos de A.
- B **contiene** objetos de A.
- B **registra** objetos de A.
- B **utiliza** objeto de A.
- B **tiene los datos de inicialización** de A.

**Consecuencias:**

**Malas:** No es conveniente usarlo cuando construyamos a partir de instancias que ya existen.

**Buenas:** Produce bajo acoplamientos.

Ejemplo:



## **Patrón bajo acoplamiento**

### **Nombre:** Bajo acoplamiento

**Problema:** Elementos que dependen de demasiados elementos. Una modificación conlleva demasiadas modificaciones colaterales, difíciles de entender aisladamente y difíciles de reutilizar.

**Solución:** Asignar responsabilidades de forma que tengamos elementos (clases, subsistemas,...) que dependan justo de los únicos elementos que necesite.

### **Consecuencias:**

**Malas:** Llevado a un extremo puede ocasionar diseños pobres, en unconjunto de clases debe existir un nivel de acoplamiento adecuado.

**Buenas:** No afectan los cambios en otros elementos. Fáciles de entender de manera aislada. Aumento de la reutilización.

## **Patrón alta cohesión**

### **Nombre:** Alta cohesión

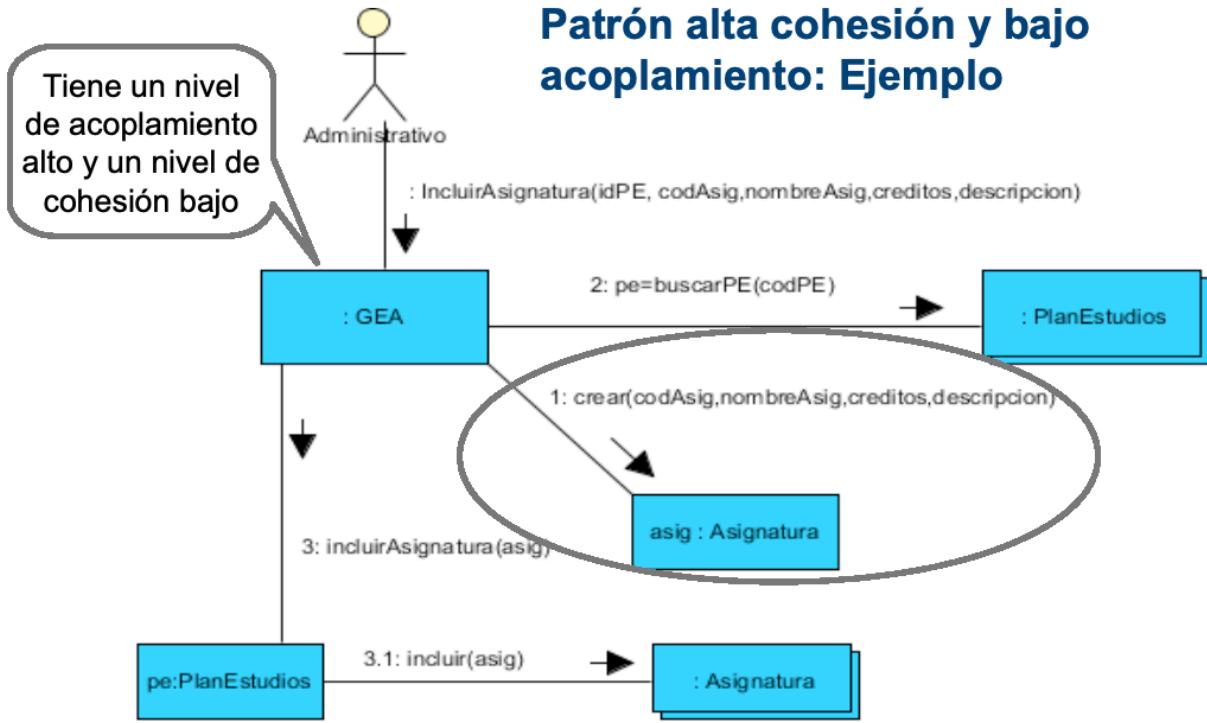
**Problema:** Elementos con pocas tareas o con muchas pero no relacionadas. Estos elementos son difíciles de entender, de reutilizar y de mantener, además se ven afectados por continuos cambios.

**Solución:** Asignar responsabilidades de forma que todas las tareas de un elemento (clase, subsistema,...) estén para lograr un objetivo común.

### **Consecuencias:**

**Malas:** Ninguna, renunciar a la alta cohesión tan sólo cuando esté muy justificado.

**Buenas:** Claridad y facilidad de entendimiento del diseño. Simplificación del mantenimiento y de las mejoras. Aumento de la reutilización.



## Patrón controlador o fachada

**Nombre:** Controlador o fachada

**Problema:** Comunicación entre los objetos de la capa del dominio de la solución y la capa de la interfaz.

**Solución:** Asignar responsabilidades de recibir o manejar un mensaje de evento del sistema a una clase que representa alguna de las siguientes opciones:

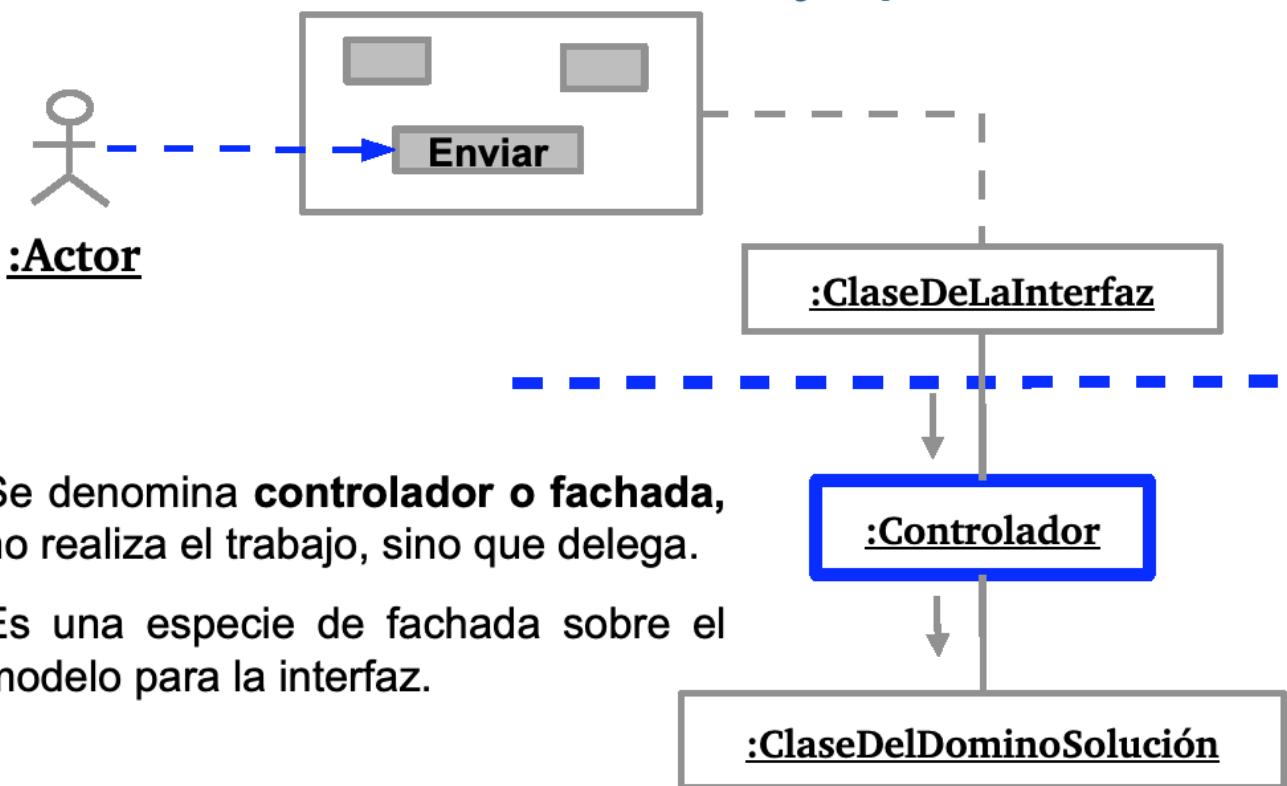
- **Al sistema global.** Único controlador que nos representa a todo el sistema.
- **Al caso de uso** en el que tiene lugar el evento del sistema. Un controlador por cada caso de uso.

**Consecuencias:**

**Malas:** Controladores saturados.

**Buenas:** Se asegura que la lógica de la aplicación no se maneja en la interfaz. Buena reutilización y bajo nivel de acoplamiento. Posibilidad de poder razonar sobre el estado de los casos de uso.

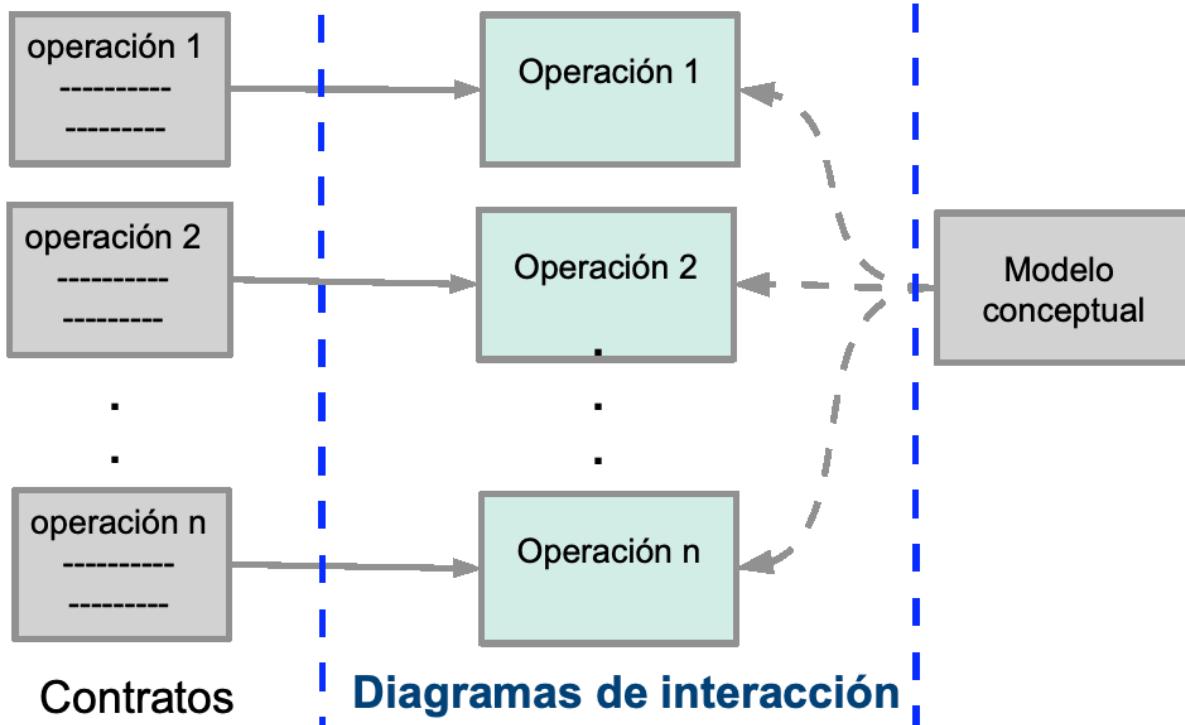
## Patrón controlador o fachada: Ejemplo



- Se denomina **controlador o fachada**, no realiza el trabajo, sino que delega.
- Es una especie de fachada sobre el modelo para la interfaz.

## Elaboración del modelo de interacción

### De dónde partimos y hacia dónde vamos



## Directrices generales

Las **bases** principales para obtener los Diagramas de comunicación son los **contratos** y el **modelo conceptual**.

El modelo conceptual nos sirve como guía para saber qué objetos pueden interaccionar en una operación.

Todo lo especificado en el contrato, especialmente las poscondiciones, las excepciones y las salidas tiene que ser satisfecho en el correspondiente diagrama de comunicación.

Para la elaboración de cada diagrama de comunicación nos ayudamos de los **patrones de diseño de Craig larman** vistos.

Vamos a ver el proceso de elaboración del modelo de interacción para la operación **definirProyecto(idProfesor, titulo, numAlum, descripcion, listaIdAsg)**

### Pasos a seguir

A) Elaborar los diagramas de interacción. Para cada operación especificada en los DSS:

1. Tener presente el diagrama de conceptos y el contrato de dicha operación.
2. Representar las relaciones del controlador con los objetos que intervienen en la interacción.
3. Asignar responsabilidades a objetos.
4. Establecer tipo de enlaces entre objetos.

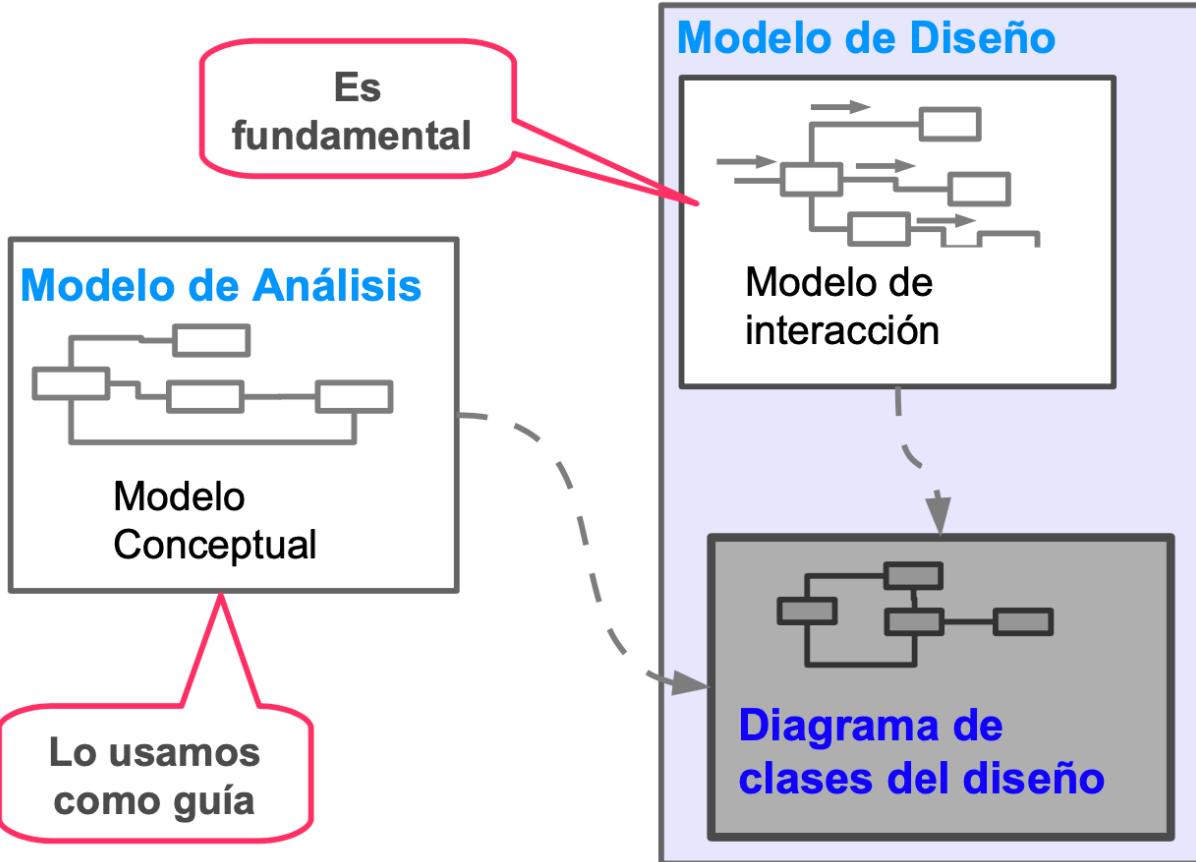
B) Inicialización del sistema.

C) Establecer las relaciones entre el modelo y la Interfaz de Usuario.

Ver ejemplo en diapositivas

## 3.3 Diseño de la estructura de clases

### 3.3.1 Diagrama de clases del diseño



## ¿Qué lo compone?

Un **diagrama de clases del diseño** describe gráficamente las especificaciones de las clases e interfaces software y las relaciones entre éstas en una aplicación. A diferencia del Modelo Conceptual representa la solución a nuestro problema.

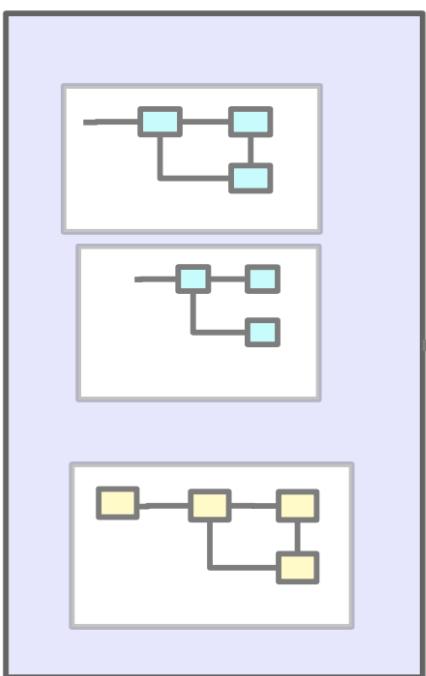
Puede contener los siguiente elementos:

- Clases con sus atributos y sus operaciones.
- Interfaces con sus operaciones y constantes.
- Relaciones entre Clase/Clase, Clase/Interface o Interface/Interface.
- Información sobre el tipo de los atributos y parámetros.
- Navegabilidad de las asociaciones.
- (Cualquier elemento que forma parte de la solución)

Herramienta para su representación **Diagrama de Clases de UML**

## Pasos a seguir

1. Identificar y representar las clases



Modelo de interacción de objetos y modelo conceptual

Todos los objetos que estén en los DC's tendrán su correspondiente clase en el DCD.

Las clases identificadas tomarán sus atributos del MC y de los DC's.

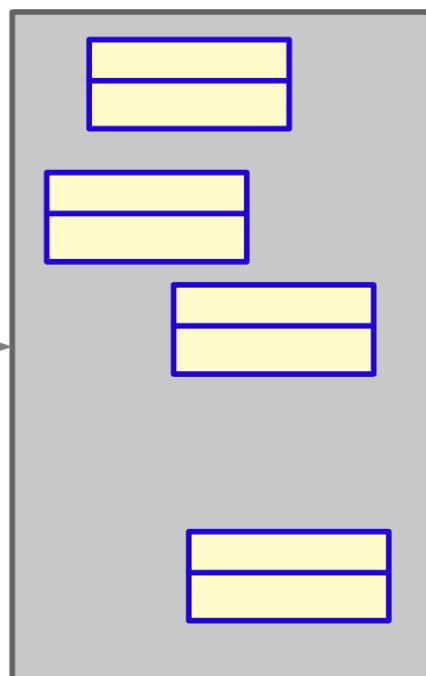
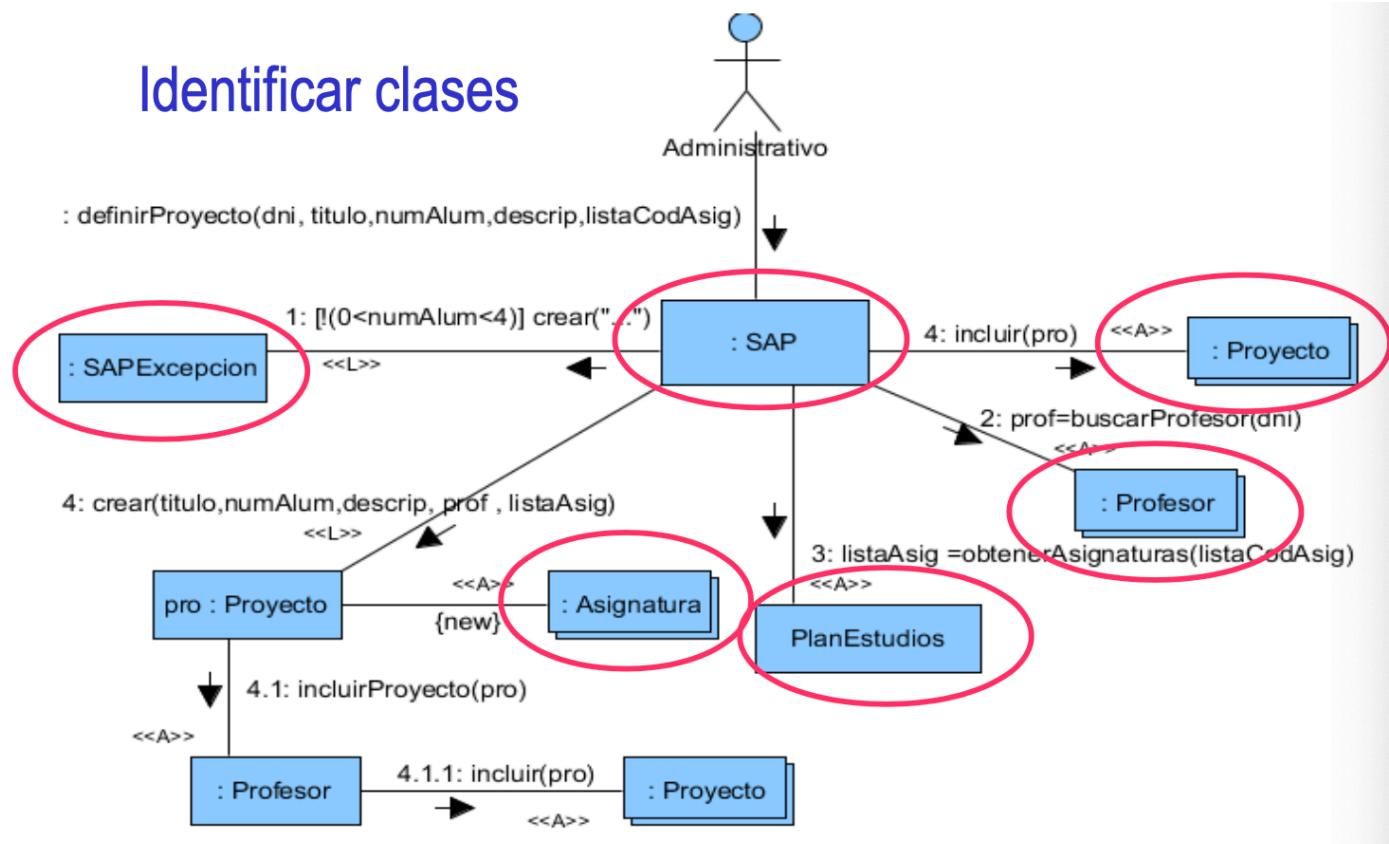
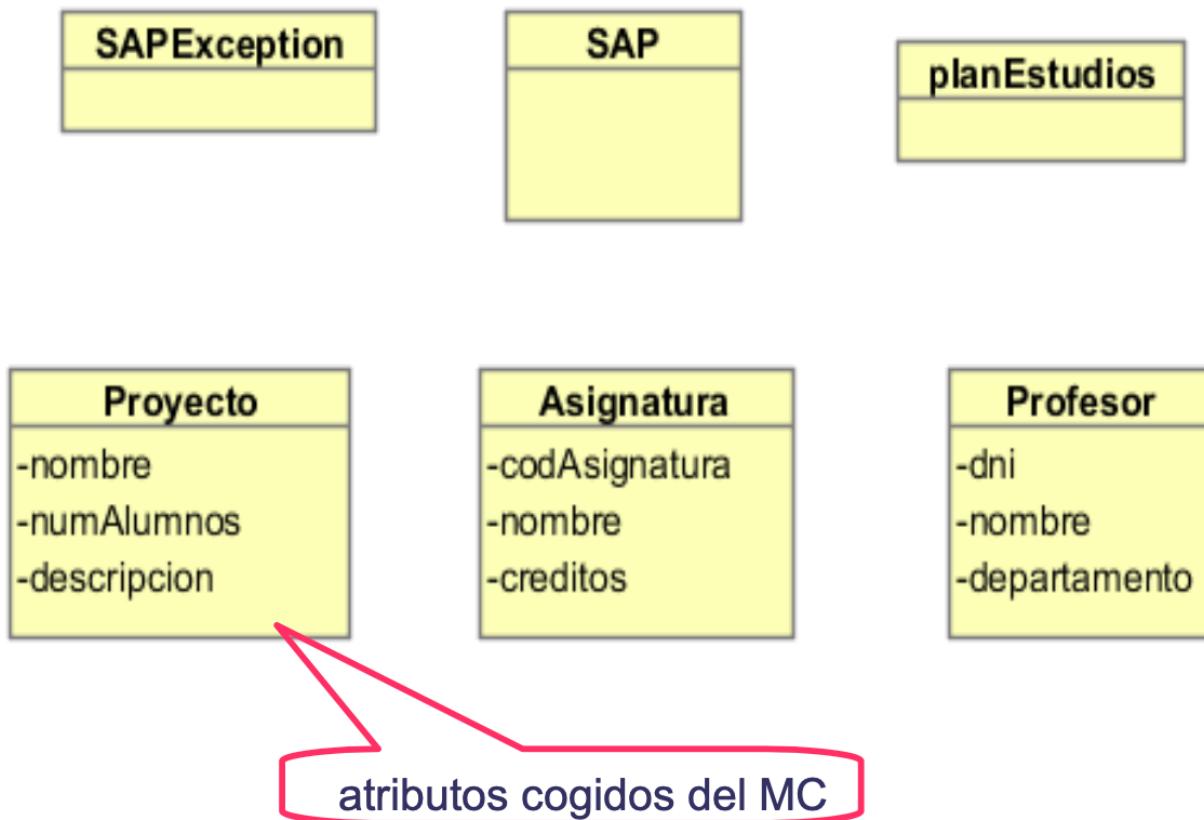


Diagrama de clases del diseño

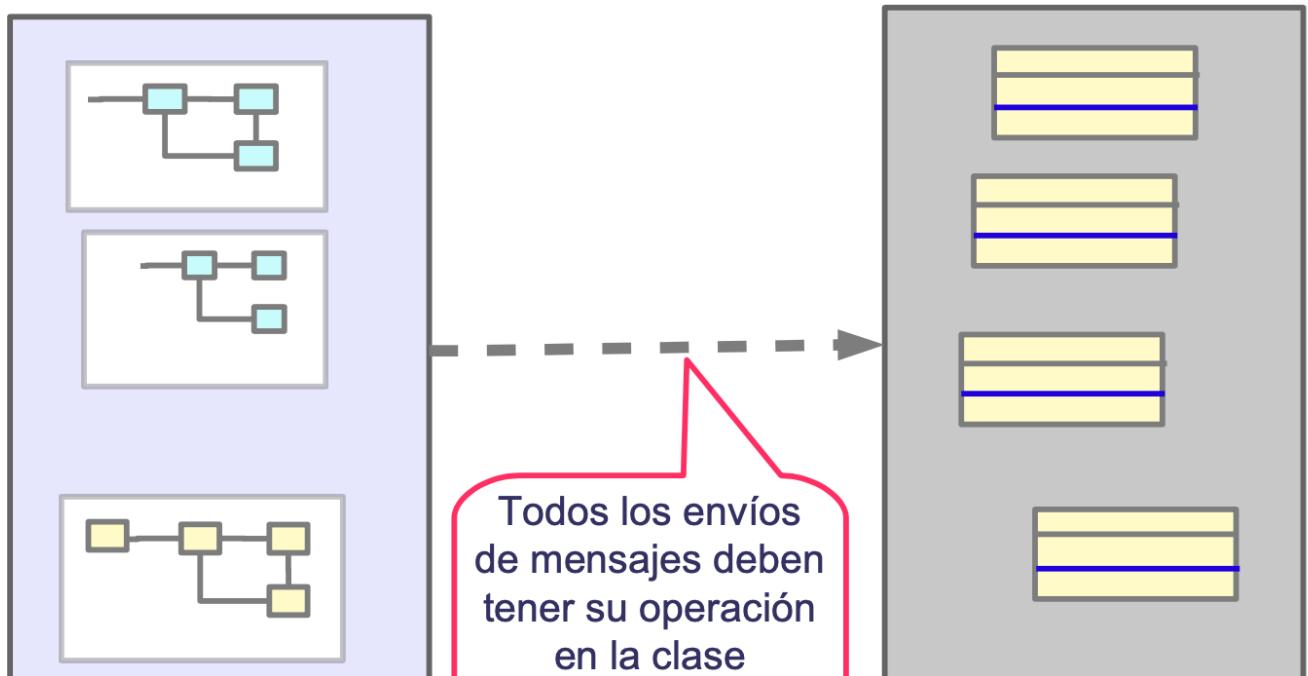
## Identificar clases



# Representar clases y sus atributos



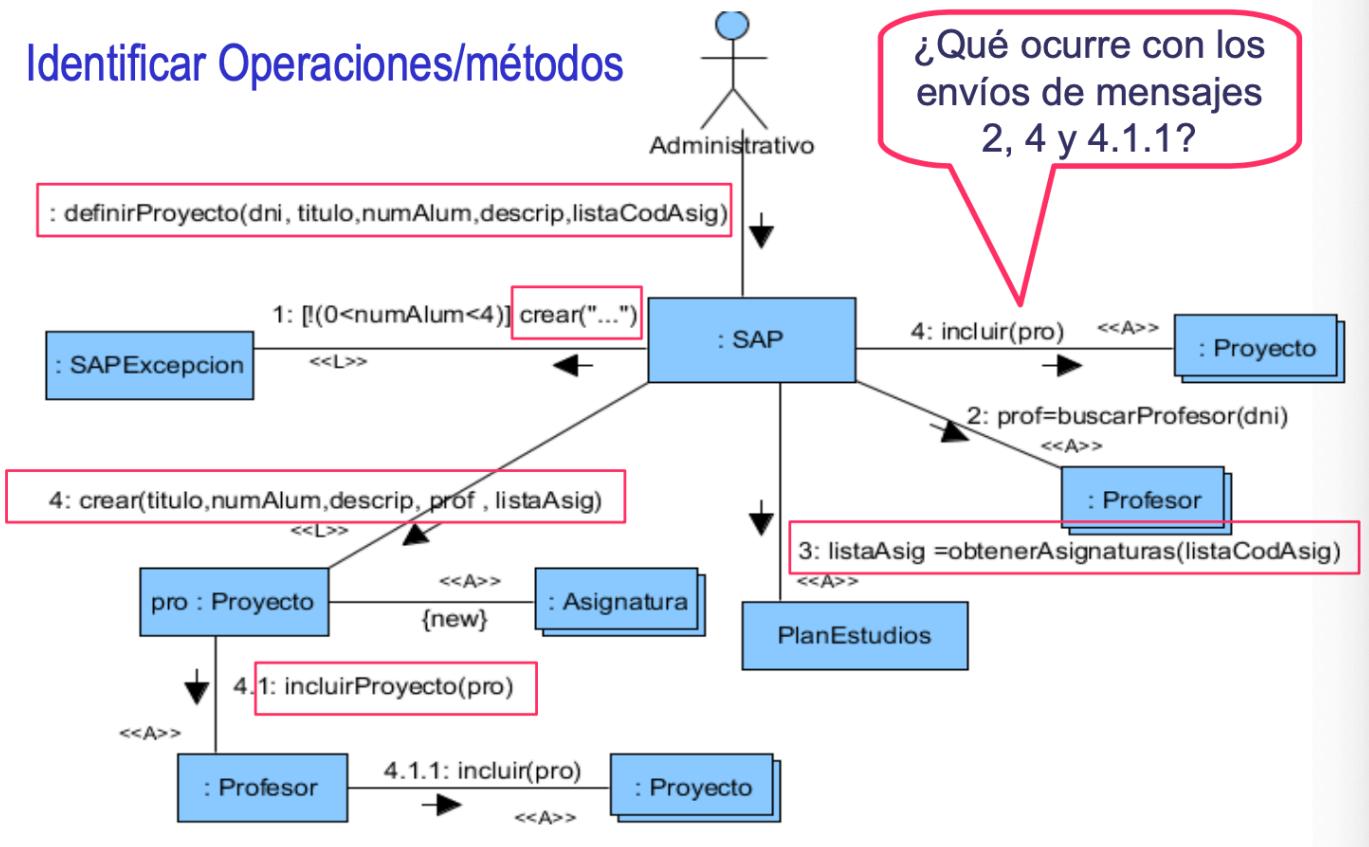
## 2. Añadir las operaciones



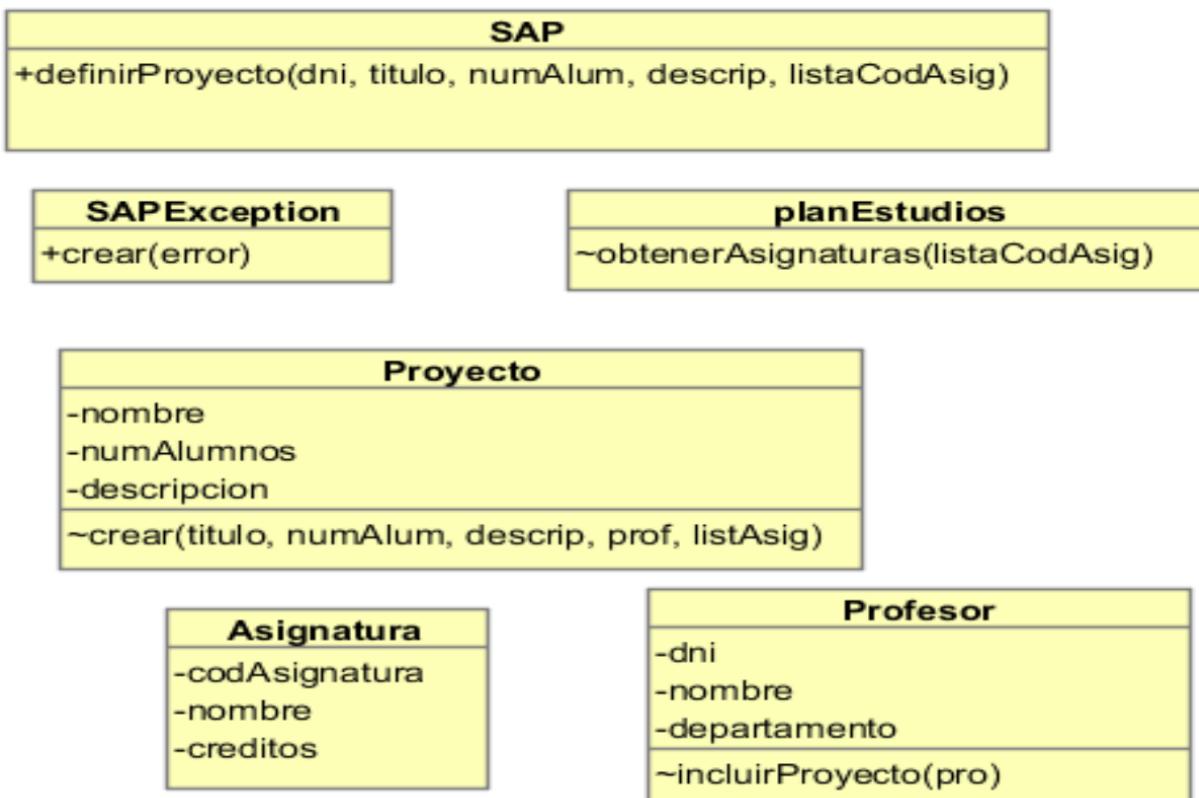
Modelo de interacción de objetos y modelo conceptual

Diagrama de clases del diseño

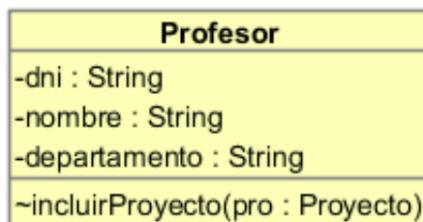
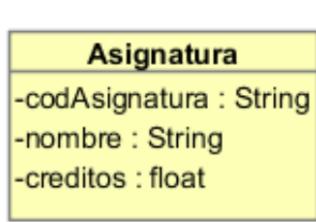
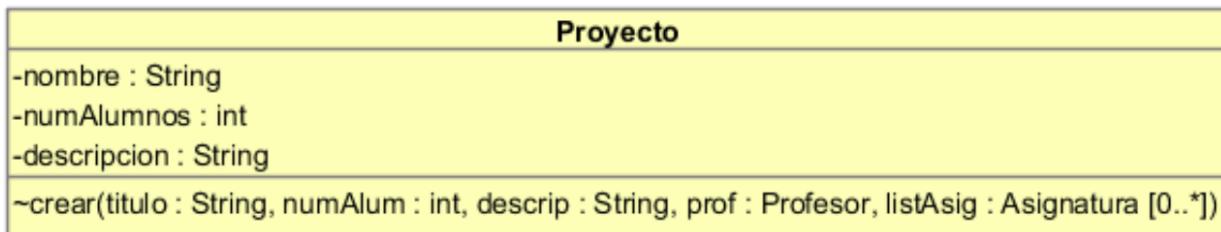
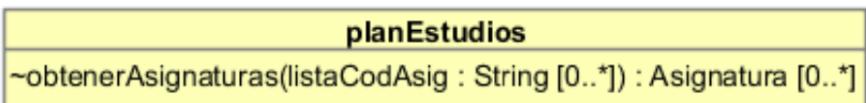
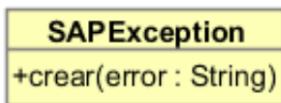
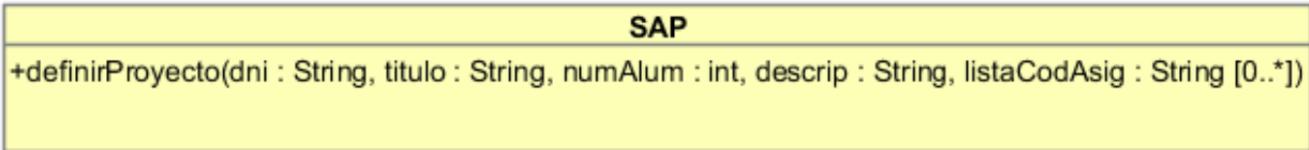
## Identificar Operaciones/métodos



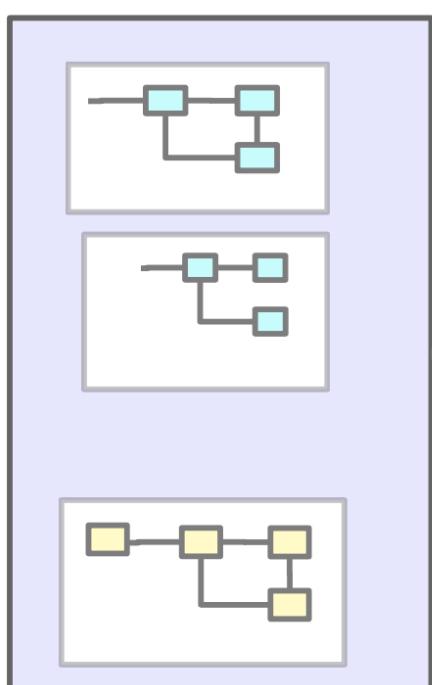
## Representar las Operaciones/métodos



3. Añadir tipos de atributos y de parametros

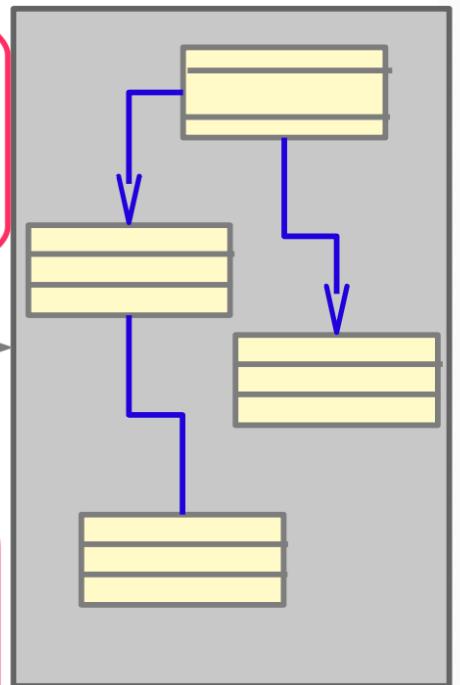


#### 4. Incluir asociaciones y navegabilidad



Todos los enlaces estereotipados con <<A>> deben tener su correspondiente **asociación**

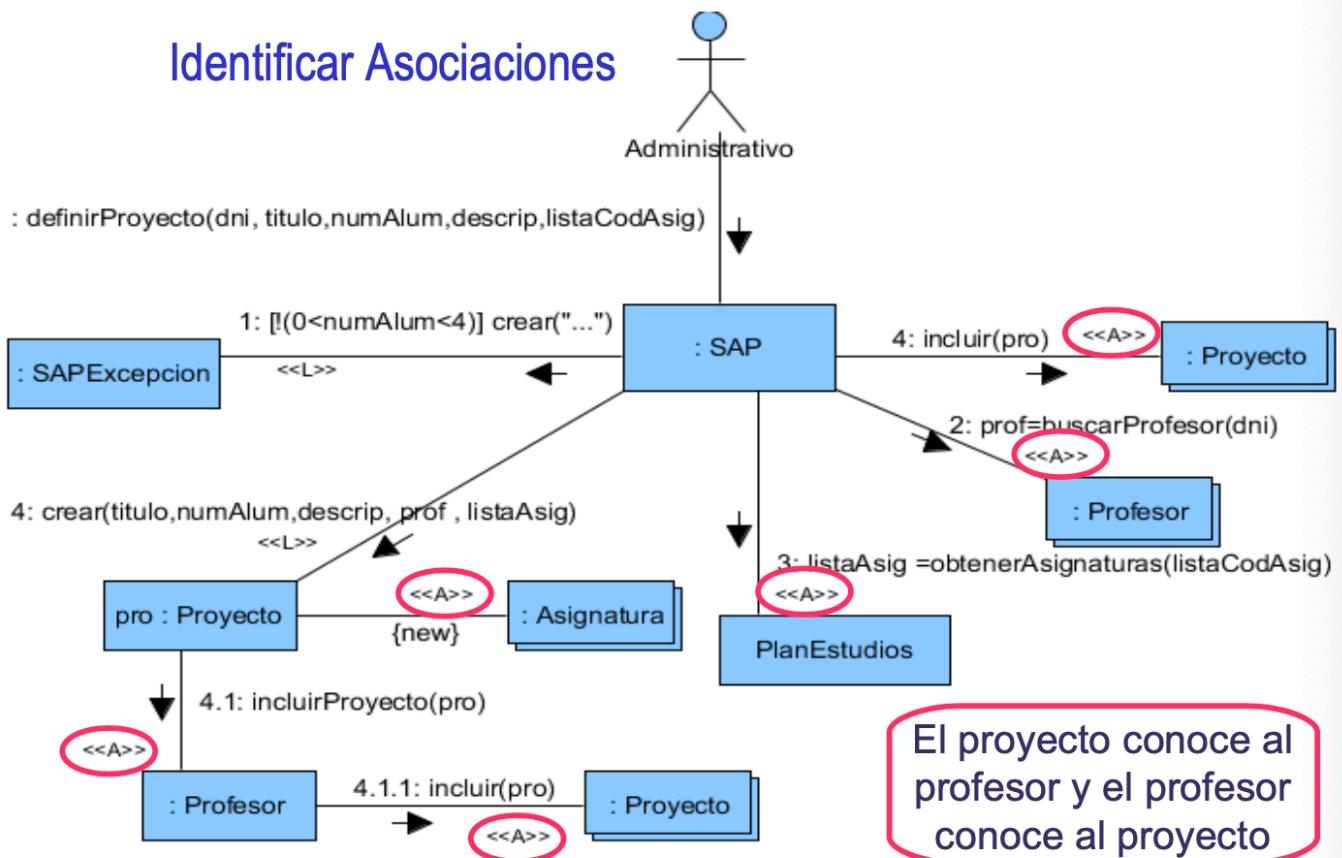
La **navegabilidad** nos la da la dirección del envío de mensaje y la **multiplicidad** la existencias de multiobjetos



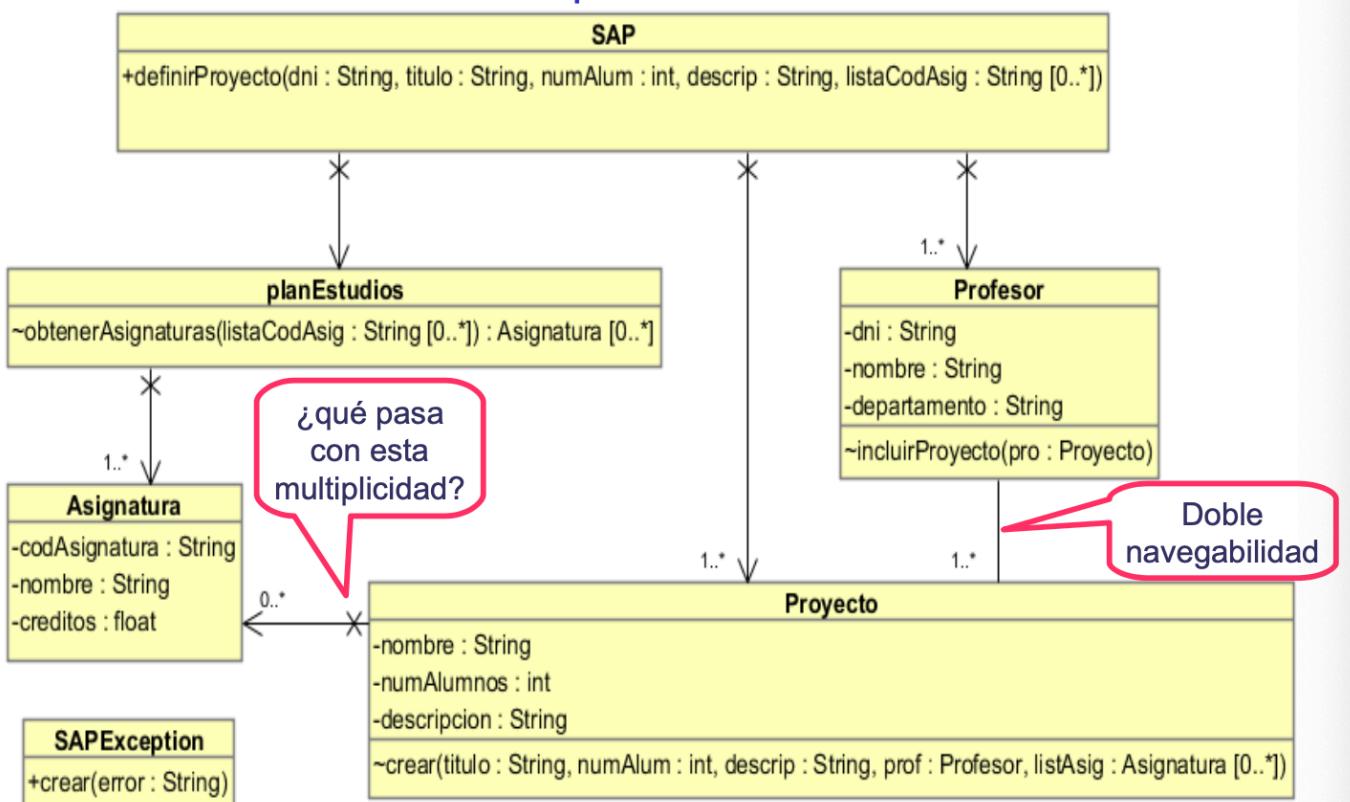
Modelo de interacción de objetos y modelo conceptual

Diagrama de clases del diseño

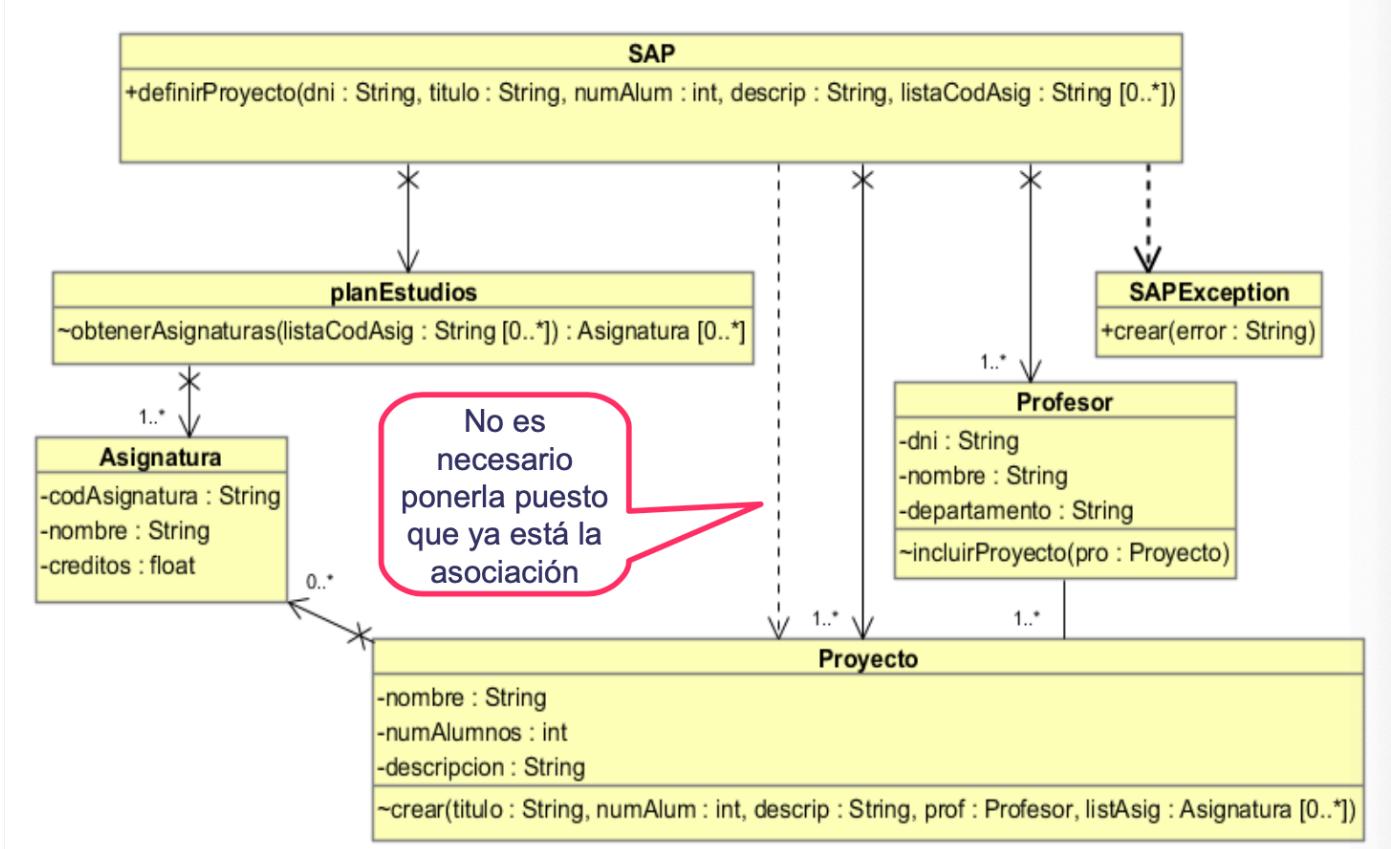
## Identificar Asociaciones



## Representar las asociaciones



## 5. Incluir dependencias



## ¿Qué tenemos?

**Tenemos:** Un único diagrama de clases del diseño, con toda la información que hemos ido obteniendo de los diagramas de comunicación, en el que todas las clases están totalmente diseñadas.

**Nos queda:** Incluir relaciones de generalización.

Sospechar que las generalizaciones que hay en el modelo conceptual también pueden aparecer en el diagrama de clases del diseño. Y proceder de la siguiente forma:

En el diagrama de clases del diseño obtenido hasta este punto, observar:

- Clases con nombre que nos identifiquen las distintas clasificaciones de un conjunto de objetos.
- Clases con los mismos atributos.
- Clases con la misma asociación con una clase.
- Clases con operaciones con el mismo nombre o parecido. Para asegurarnos que se corresponde con igual o semántica parecida, mirar la similitud de estructura de los diagramas de colaboración correspondientes.

Cuando tenemos alguna de estas situaciones o las tres, proceder a establecer una generalización entre esas clases, llevándose a la superclase atributos, operaciones y asociaciones comunes.

## 3.4 Diseño de la arquitectura

### Conceptos de diseño de la arquitectura

El **diseño de la arquitectura** va a proporcionar una imagen global de la estructura del sistema software, esbozando los artefactos más importantes que lo componen, principalmente **subsistemas y sus relaciones (interfaces)**.

La **determinación de la arquitectura del software** consiste en la toma de **decisiones** respecto a:

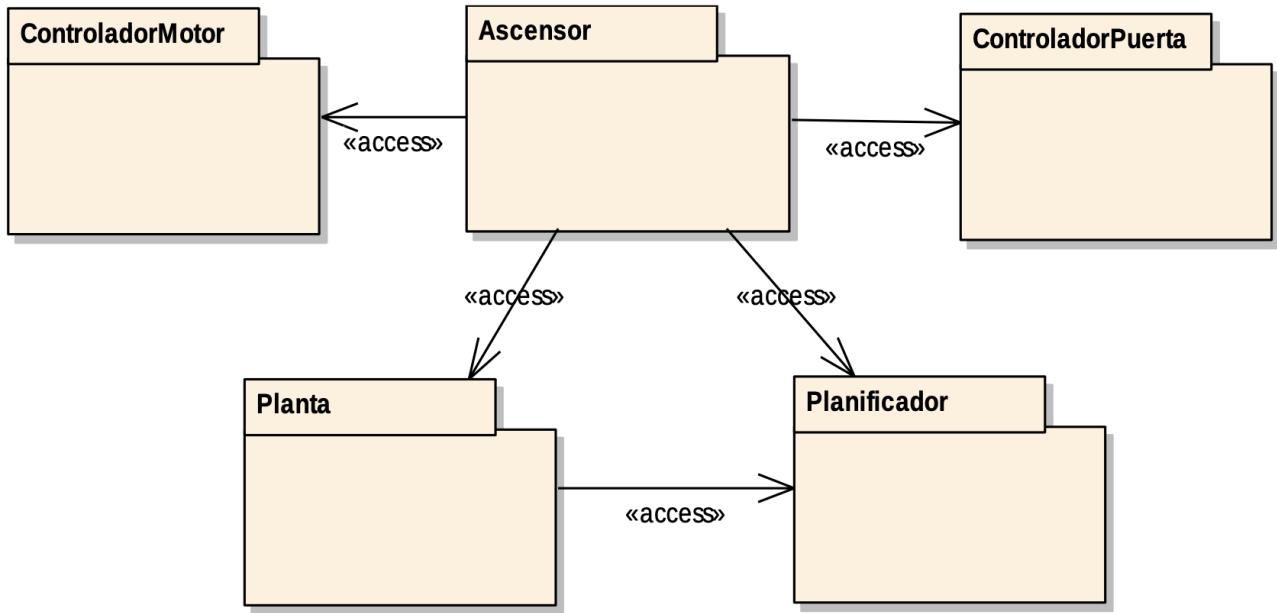
- Cómo se va a dividir el sistema en subsistemas.
- Cómo deben interactuar dichos subsistemas.
- Cuál va a ser la interfaz de cada uno de estos subsistemas.
- Cuál va a ser el estilo arquitectónico usado.

### Importancia de la arquitectura

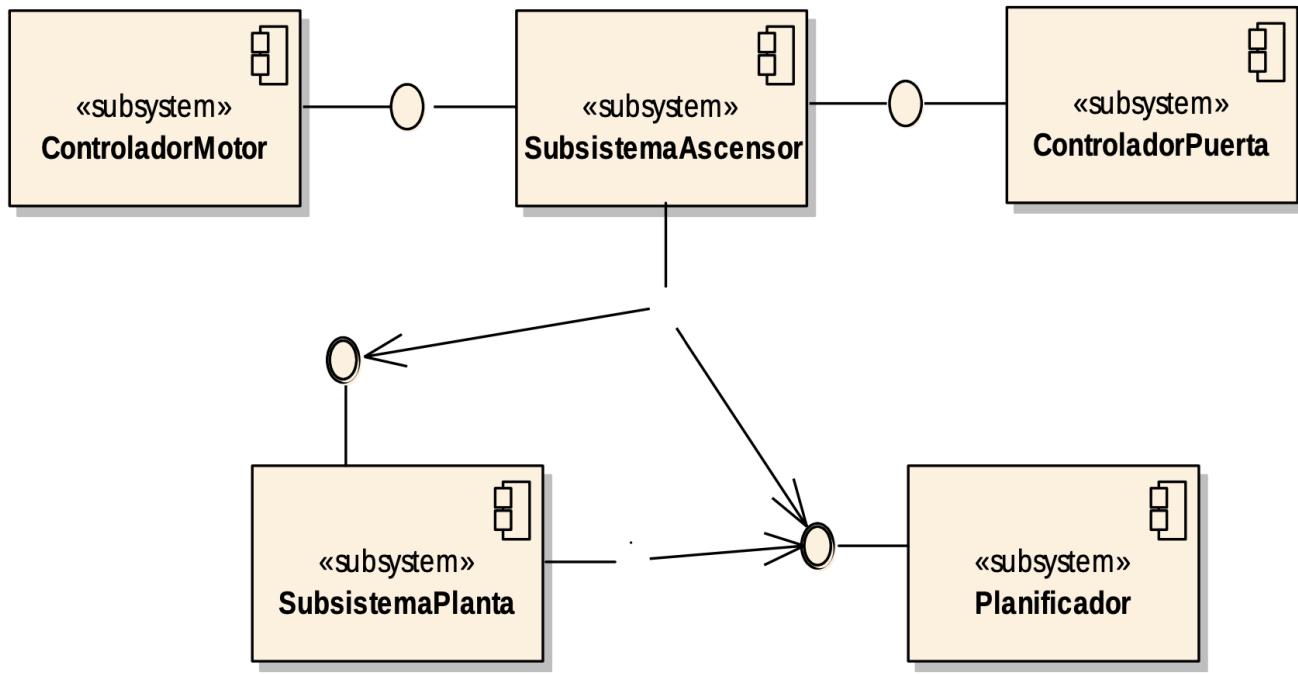
- Facilita la comprensión de la estructura global del sistema.
- Permite trabajar en los subsistemas de forma independiente.
- Facilita las posibles extensiones del sistema.
- Facilita la reutilización de los distintos subsistemas.

### Herramientas para su representación

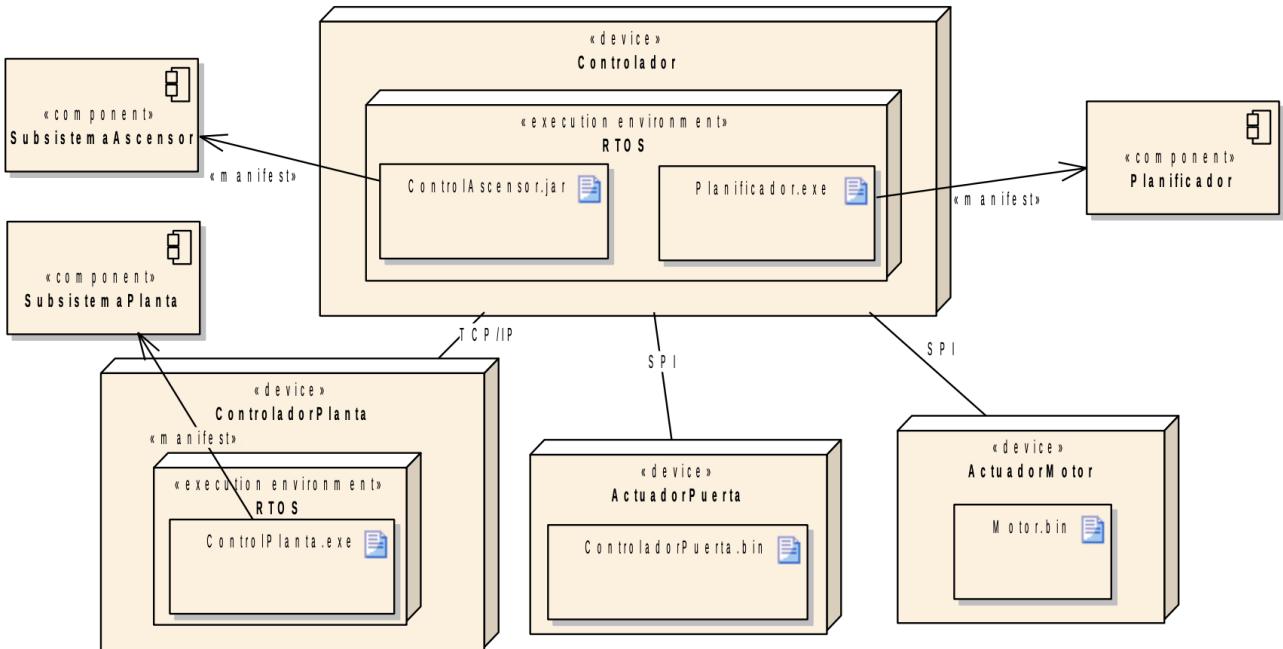
**Diagrama de paquetes:** Describe el sistema en torno a agrupaciones lógicas y proporciona una primera estructuración del sistema.



**Diagrama de componentes:** Representa una estructuración concreta del sistema a partir de los componentes software (sistemas) y su interrelación (interfaces).



**Diagrama de despliegue:** Especifica el hardware físico sobre el que se ejecutará el sistema software y cómo cada uno de los subsistemas software se despliega en ese hardware.



## Estilos arquitectónicos

Un **estilo arquitectónico** proporciona un conjunto de subsistemas predefinidos, especificando sus responsabilidades e incluyendo reglas y guías para organizar las relaciones entre ellos. No proporcionan la arquitectura del sistema, sino una guía para obtener dicha arquitectura.

Estilos arquitectónicos más generalizados:

- Arquitectura multicapa (Microkernel).
- Arquitectura cliente-servidor.
- Arquitectura de repositorio (Repository).
- Arquitectura MVC (Model-View-ControllerVC).
- MDA(ModelDrivenArchitecture)

## Arquitectura multicapa

Distribuye los subsistemas de un sistema en capas, de tal forma que:

- Cada capa presta servicios a la capa inmediatamente superior y actúa como cliente sobre las capas más internas.
- El diseño incluye los protocolos que establecen cómo interactuará cada par de capas.
- Las capas más bajas proporcionan servicios de bajo nivel como protocolos de comunicación en red, acceso a base de datos, ...

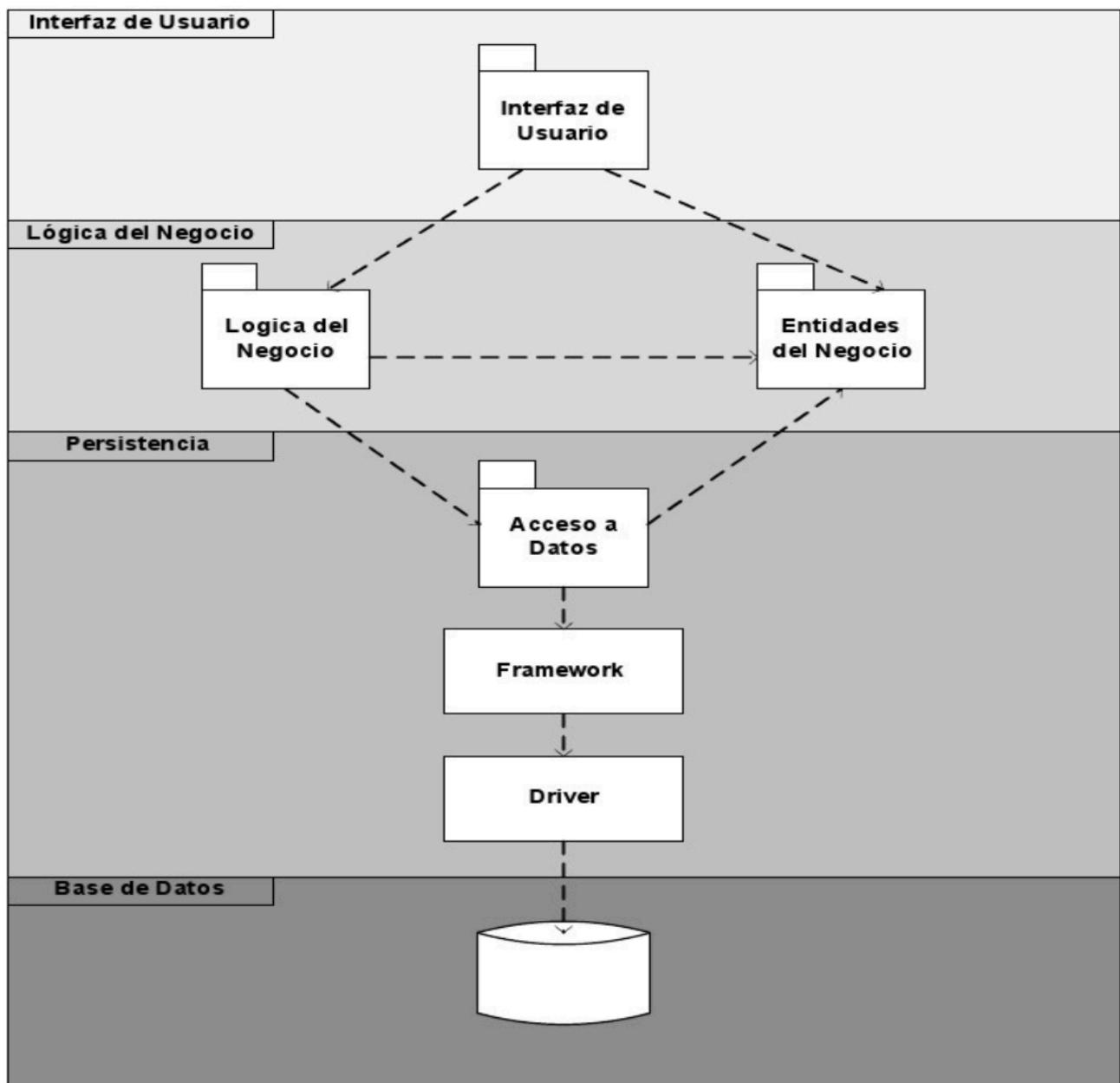
## Ventaja:

- Cada capa puede diseñarse, implementarse y probarse de forma independiente

- Arquitectura fácilmente cambiante y portable:
  - Preservando la interfaz, una capa puede ser reemplazada por otra.
  - El cambio en la interfaz de una capa sólo afecta a la capa adyacente.

## Desventajas:

- Dificultad para decidir cuáles son los servicios de cada capa.
- El rendimiento puede resultar afectado, debido a los múltiples niveles que hay que pasar para llegar a la que proporciona el servicio.



## Arquitectura multicapa: Principios de diseño

Las **capas** pueden diseñarse, construirse y probarse **independientemente**.

Una capa bien diseñada presenta **alta cohesión**.

Las capas deben estar lo más **desacopladas** posible:

- Dependencias en un sentido
- Todas las dependencias expresadas en las interfaces.

Una capa de un determinado nivel bien diseñada no tienen conocimiento de las capas superiores (buen **ocultamiento de información**).

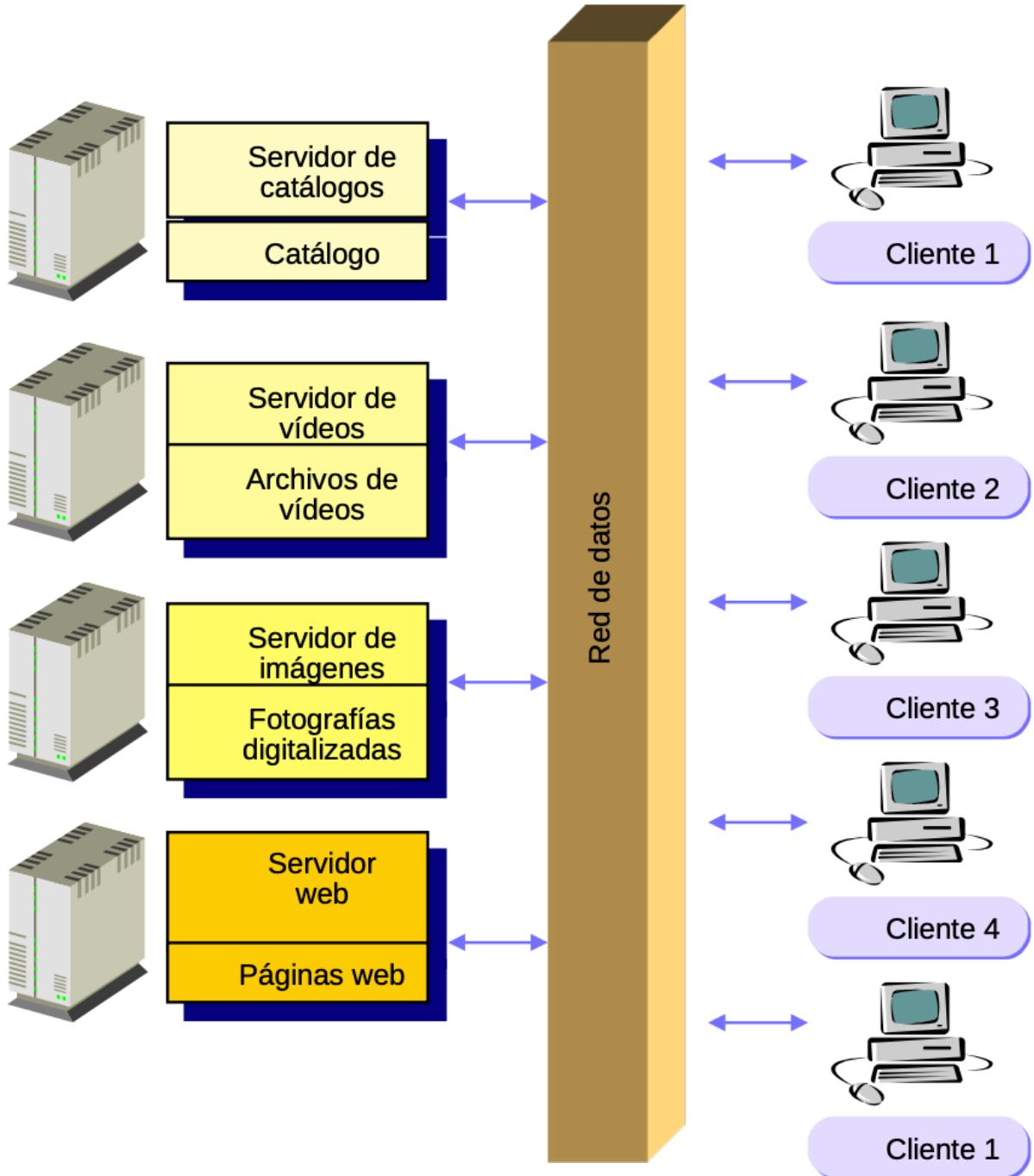
Las capas más inferiores deben diseñarse para prestar servicios de bajo nivel (**bajo acoplamiento**).

## Arquitectura cliente-servidor

Subsistemas distribuidos que muestra cómo datos y procesamiento se distribuyen a lo largo de varios procesadores.

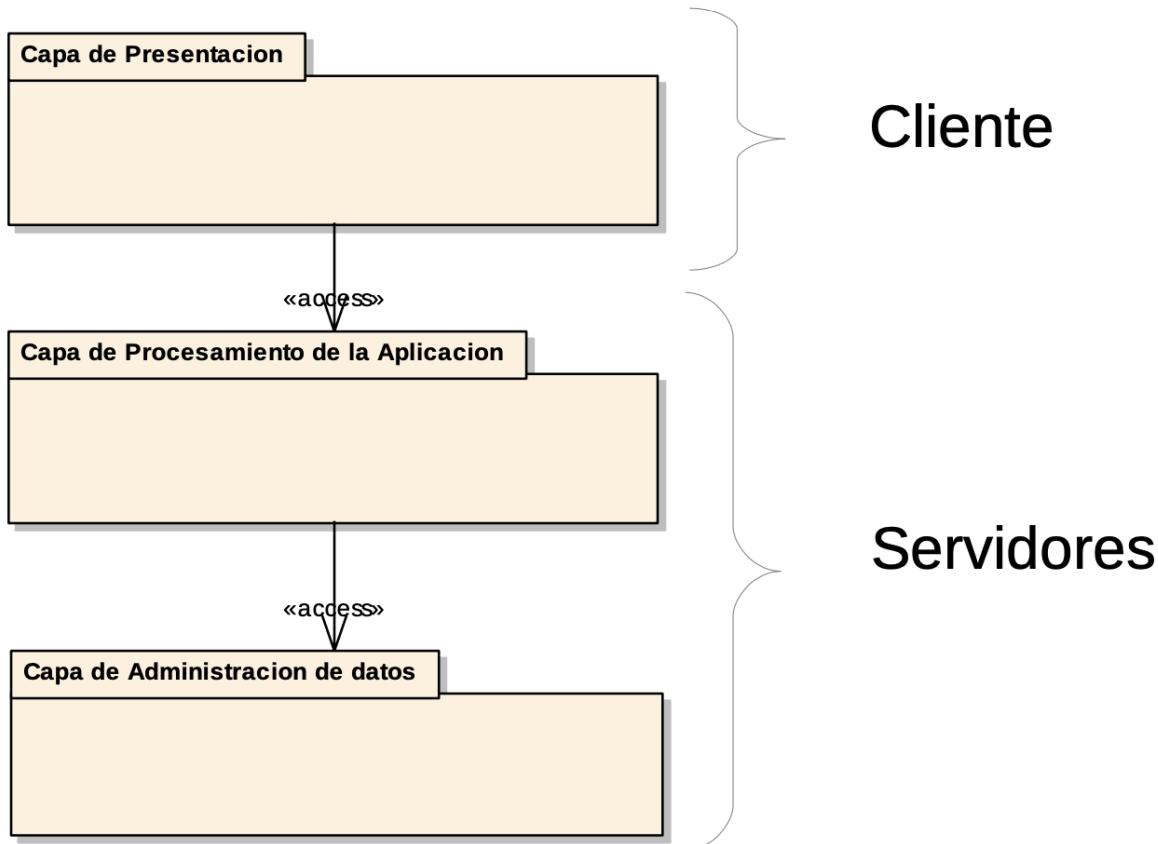
### Componentes:

- **Servidores** independientes que ofrecen servicios a otros subsistemas.
- **Clientes** que pueden solicitar esos servicios.
- Una **red** que permite a los clientes acceder a los servicios ofrecidos por los servidores.



### Arquitectura Cliente-servidor en capas:

Se puede diseñar esta arquitectura por capas:



### Arquitectura Cliente-servidor: principios de diseño

- La descomposición en clientes y servidores es una división fuerte del sistema que permite realizar el diseño, implementación y prueba de ambas partes **independientemente**.
- Mejora la **cohesión** de los subsistemas al proporcionar servicios específicos a los clientes.
- Reduce el **acoplamiento** al establecer un canal de comunicación para el intercambio de mensajes.
- Aumenta el nivel de **abstracción** al tener subsistemas o componentes distribuidos en nodos separados.
- Facilita la **reutilización** dado que se pueden usar marcos de desarrollo para sistemas distribuidos (CORBA, RMI, ...).
- Los sistemas distribuidos pueden **reconfigurarse fácilmente** añadiendo servidores o clientes extras.
- Pueden escribirse clientes para nuevas plataformas sin necesidad de modificar a los servidores.