Patrones de Diseño

Índice

1. Introducción

- ¿Qué son los patrones de diseño?
  - Un patrón de diseño describe un problema que ocurre repetidamente en nuestro entorno
  - Además, describe el núcleo de la solución al problema, en una forma tal que esta solución se puede utilizar millones de veces, sin ni siquiera realizar lo mismo dos veces
- Permiten reutilizar un conocimiento de diseño ya existente
  - Ahorro de tiempo, toma de decisiones automática, …

Características principales de un patrón de diseño

- Un patrón de diseño es
  - Una solución plantilla a un problema recurrente
    - Pensar al menos una vez antes de reinventar la rueda
  - Conocimiento de diseño reusable
    - Están generalizados de sistemas existentes
    - Proporcionan un vocabulario compartido a los diseñadores
    - De más alto nivel que las clases o las estructuras de datos (listas enlazadas, árboles binarios ...)
    - De más bajo nivel que los *framework* de aplicación
  - Un ejemplo de diseño modificable (maleable)
    - Se soporta y facilita el cambio, la reutilización y la mejora mediante la utilización de clases abstractas y delegación
    - Son guías, no reglas rigurosas

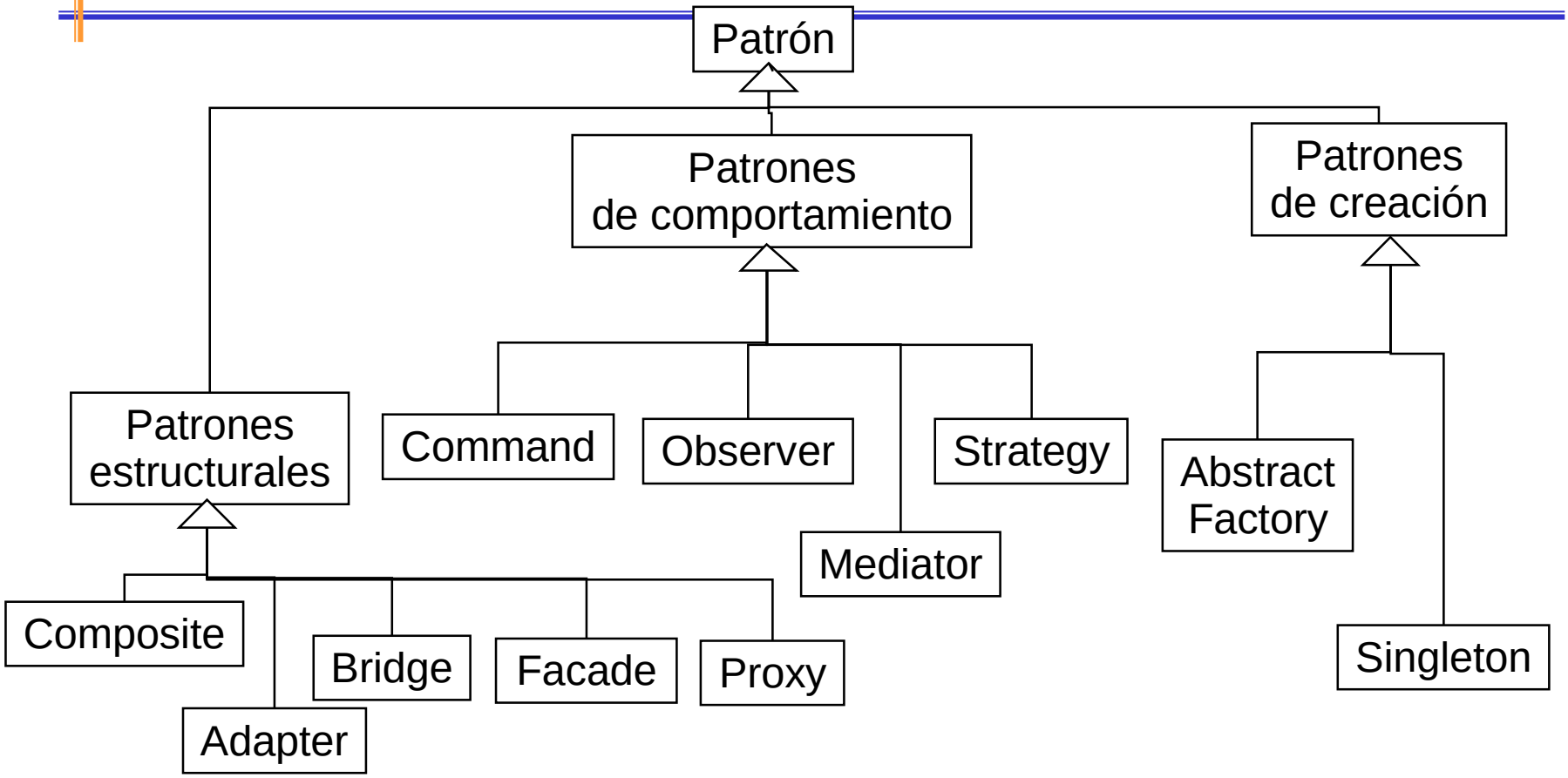
¿Cómo se describe mínimamente un patrón?

- Nombre bien conocido
- Motivación
  - Descripción del problema, a veces mediante un ejemplo real
- Solución
  - Estructura
    - A través de un diagrama de clases
  - Colaboraciones
    - Utilización de diagramas de interacción (secuencia y/o comunicación) para describir cómo colaboran los objetos para llevar a cabo sus responsabilidades
- Consecuencias
  - Ventajas y desventajas del uso del patrón

Categorías de patrones

- Patrones estructurales
  - Establecen cómo se componen clases y objetos para formar estructuras mayores que implementan nueva funcionalidad
- Patrones de comportamiento
  - Describen algoritmos y asignación de responsabilidades a objetos
  - Describen cómo se comunican los objetos, cooperan y distribuyen las responsabilidades para lograr sus objetivos
- Patrones de creación
  - Abstraen el proceso de instanciación
  - Ayudan a que el sistema sea independiente de cómo se crean, componen y representan los objetos
- Otros patrones por propósito: arquitectura, concurrencia, testeo, GUI

Categorías de patrones (II)



Gamma E, Helm R, Johnson R, Vlissides J (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley.
https://www.oodesign.com/
https://sourcemaking.com/design_patterns

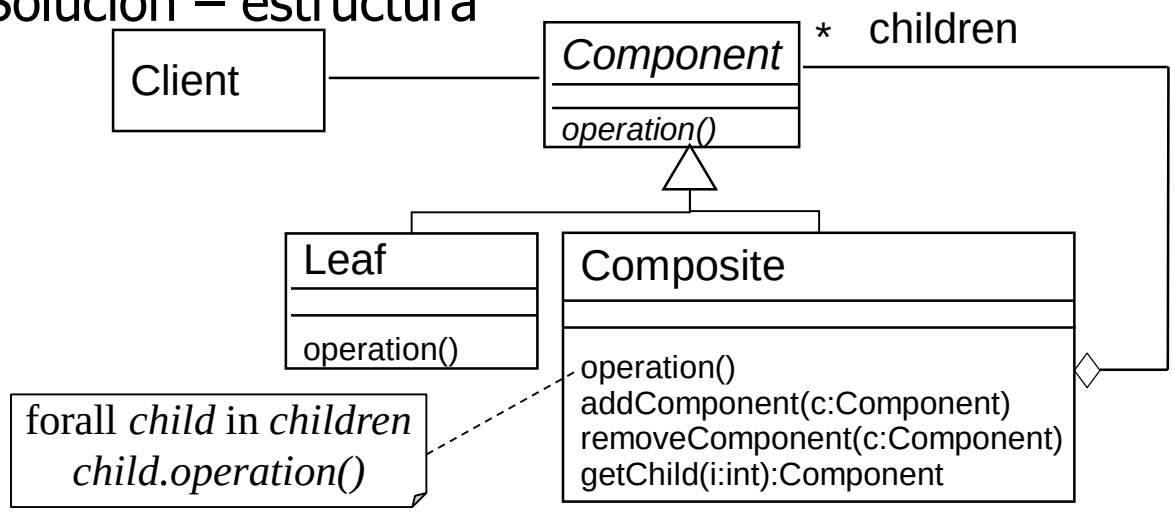
2. Patrones estructurales

- Establecen cómo se componen clases y objetos para formar estructuras mayores que implementan nueva funcionalidad

Composite	Modelan agregaciones flexibles (árboles con anchura y profundidad variables)
Adapter	Interfaz con la realidad Útiles para interactuar con sistemas existentes: sistemas legados
Bridge	Interfaz con la realidad y preparación para el futuro Útiles para interactuar con sistemas existentes y futuros
Facade	Establecen interfaces para subsistemas (arquitecturas cerradas frente arquitecturas abiertas)
Proxy	Proporcionan transparencia de localización

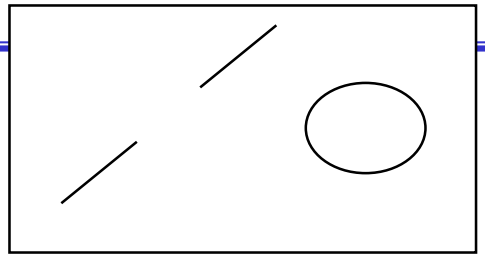
2.1. Patrón Composite

- Motivación
  - El objetivo es representar jerarquías todo-parte
    - Composiciones de objetos con dos tipos de objetos: Simples, Compuestos
    - Estructuras arborescentes que representan jerarquías todo-parte con una anchura y una profundidad arbitrarias
- Solución – estructura

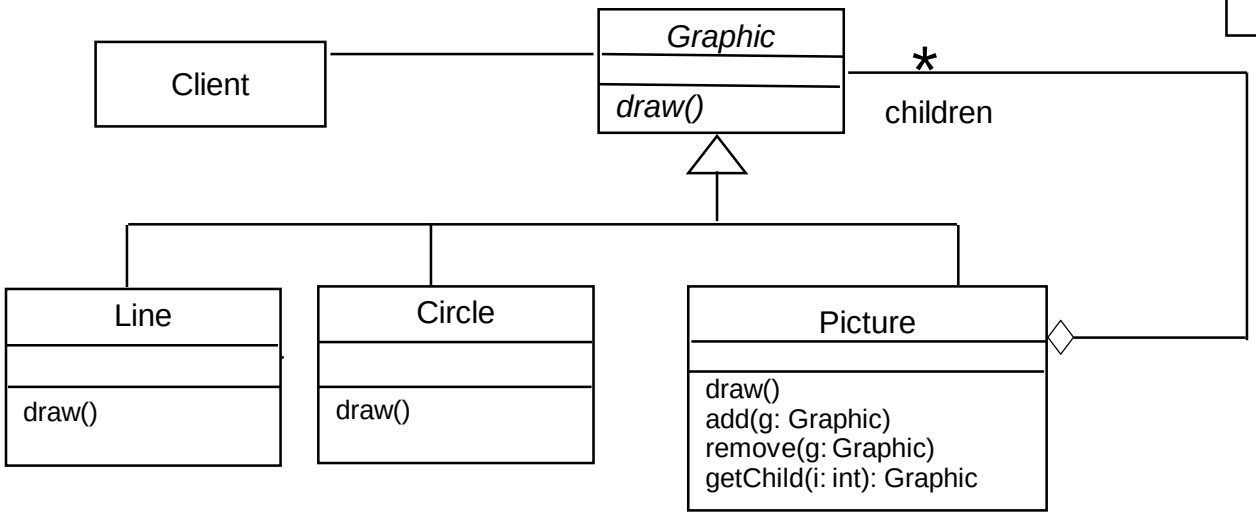


- Consecuencias: Se permite que los clientes puedan tratar a objetos individuales y a composiciones de objetos de una forma uniforme

Ejemplo: aplicaciones gráficas



La clase *Graphic* representa tanto primitivas (Line, Circle) como sus contenedores (Picture)

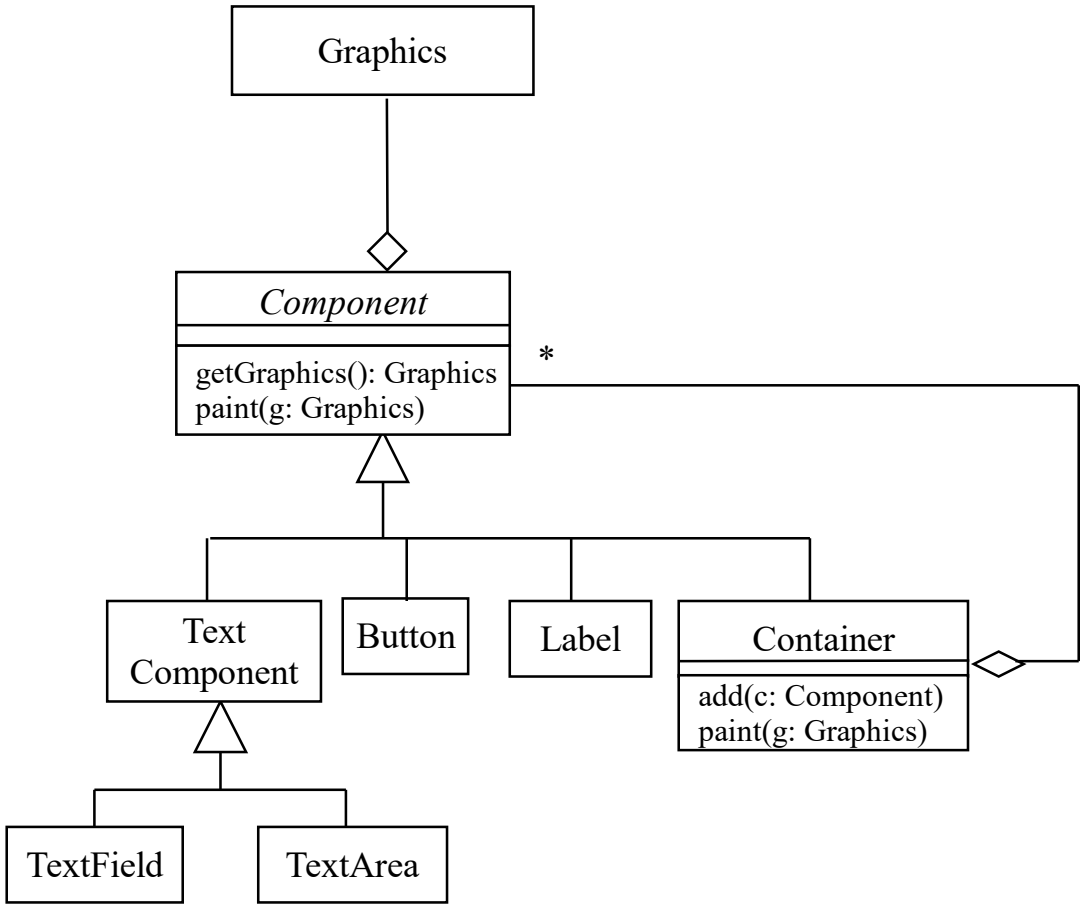



```
public class Picture extends Graphic{
 private List children;
 …
 public void draw(){
   for (Graphic g: children)
     g.draw();
  }
 …
```

```
 …
 public void add (Graphic g) {
   children.add(g);
 }
 public void remove (Graphic g) {
   children.remove(g);
 }
 ...
}
```

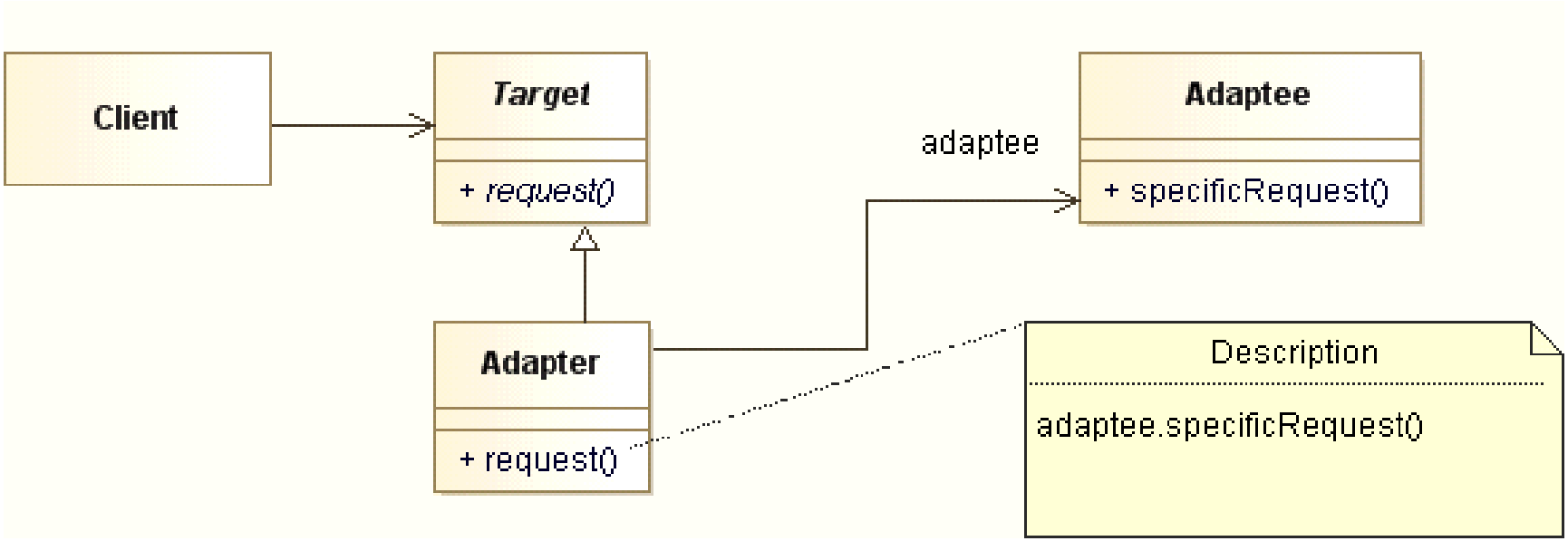
Ejemplo: la librería AWT de Java

- Se puede modelar con el patrón *Composite*



2.2. Patrón *Adapter* (I)

- Motivación
  - Se pretende proporcionar una interfaz nueva a componentes legados existentes (reingeniería)
  - El patrón se conoce también como *wrapper* (envoltorio)
- Solución - estructura



2.2. Patrón *Adapter* (II)

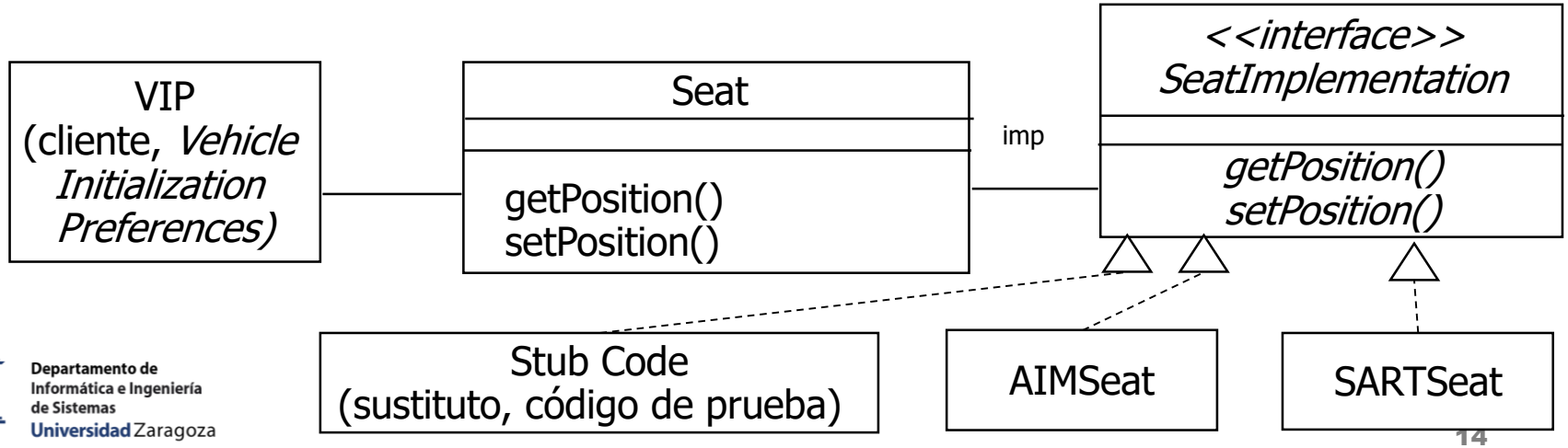
- Se usa la delegación para enlazar un *Adapter* y un *Adaptee*
- La herencia de interfaz *Target* (client interface) se usa para especificar la interfaz de la clase *Adapter*
- *Target* y *Adaptee* (adaptado, sistema legado) existen previamente a la construcción del *Adapter*

Consecuencias

- "Convierte la interfaz de una clase en otra interfaz que el cliente espera"
- El patrón *Adapter* permite que las clases trabajen juntas cuando de otra forma no podrían debido a interfaces incompatibles

2.3. Patrón *Bridge* (*Handle/Body*)

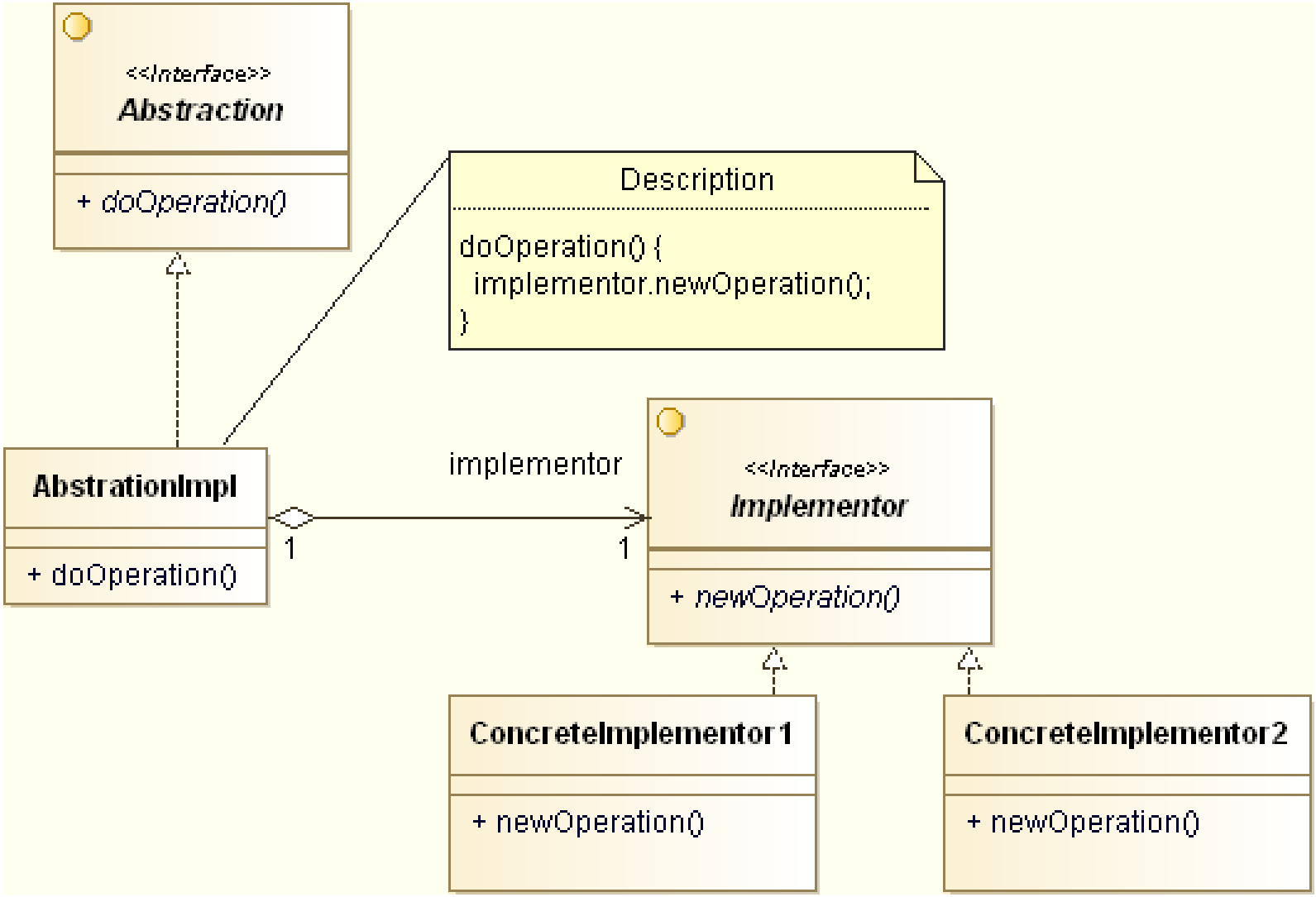
- Motivación
  - Se pretende interaccionar con un componente incompleto, no conocido todavía o no disponible durante las pruebas
  - Ejemplo: En un sistema de configuración personalizada de vehículos mediante tarjetas inteligentes se necesitan leer/modificar los datos de configuración del asiento pero la conexión con los dispositivos externos no se ha implementado todavía, o utilizamos distintas formas de conexión con los dispositivos externos



Seat Implementation

```
public interface SeatImplementation {
  public int getPosition();
  public void setPosition(int newPosition);
}
public class Stubcode implements SeatImplementation {
  public int getPosition() {
    // stub code for GetPosition
  }
  ...
}
public class AimSeat implements SeatImplementation {
  public int getPosition() {
    // actual call to the Adaptative Information Manager (AIM) system
  }
  ....
}
public class SARTSeat implements SeatImplementation {
  public int getPosition() {
    // actual call to the Seat Access Real Time (SART) system
 }
  ...
}
```

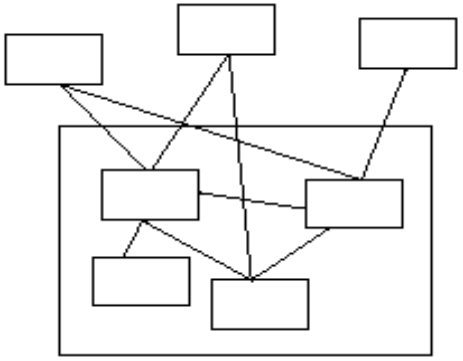
Solución - estructura



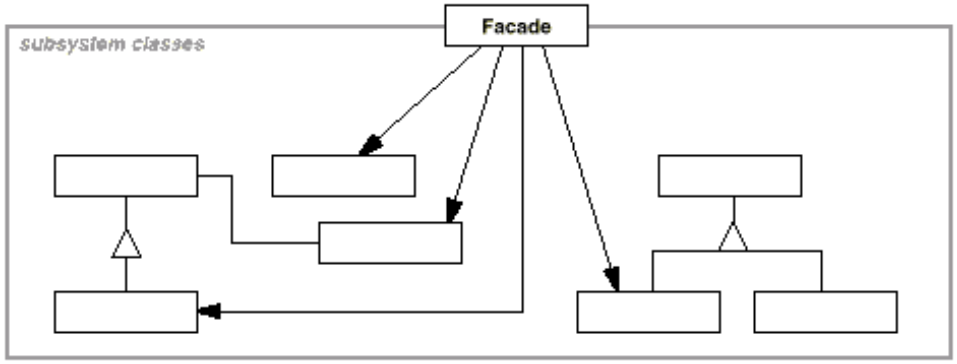
Consecuencias

- El patrón *Bridge* desacopla una abstracción de su implementación para que las dos puedan variar independientemente
- El patrón *Bridge* se utiliza para proporcionar implementaciones múltiples bajo la misma interfaz
- Similitudes con el patrón *Adapter*:
  - Ambos se utilizan para ocultar los detalles de la implementación subyacente
- Diferencias con el patrón *Adapter*:
  - El patrón *Adapter* está orientado a conseguir que componentes no relacionados trabajen juntos
    - Se aplica a sistemas que ya están diseñados (reingeniería)
  - Un *Bridge*, por otro lado, se utiliza por adelantado en un diseño para permitir que las abstracciones y las implementaciones varíen independientemente

2.4. Patrón *Façade*



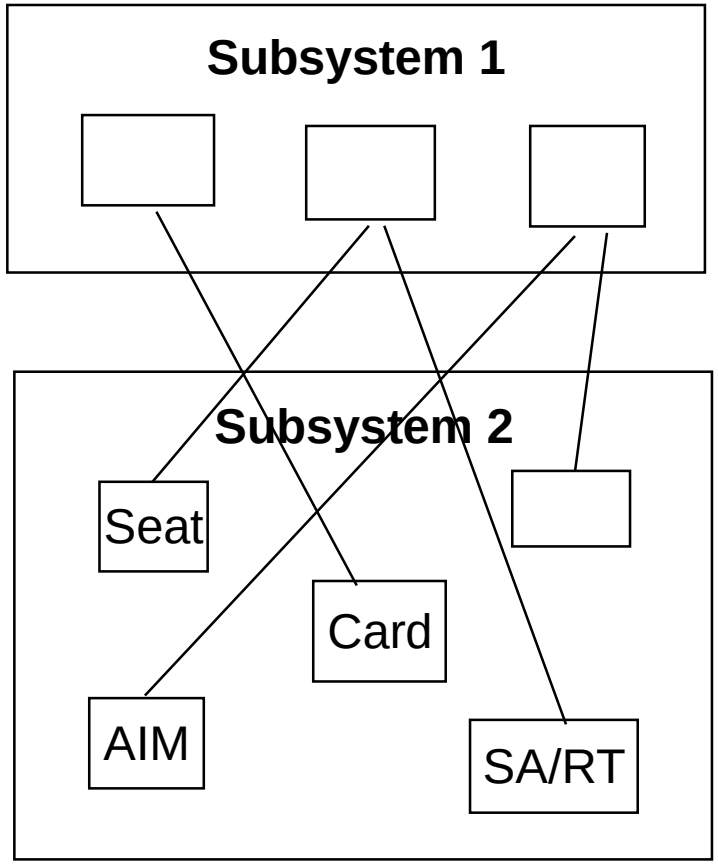
- Motivación
  - Proporciona una interfaz unificada a un conjunto de objetos dentro de un subsistema
- Solución - estructura



- Consecuencias
  - Define una interfaz de alto nivel que hace más fácil de usar el subsistema (abstrae de los detalles internos)
  - Nos facilita una arquitectura cerrada

Ejemplo de aplicación (I)

- El subsistema 1 puede observar dentro del subsistema 2 (subsistema de vehículos) y invocar sobre cualquier operación de componente o clase que desee
- Esto es un diseño caótico
- ¿Por qué es bueno?
  - Eficiencia
- ¿Por qué es malo?
  - No se puede esperar que el llamante entienda como trabaja el subsistema o las relaciones complejas dentro del subsistema
  - Podemos asegurar que el subsistema será mal empleado, conduciendo a un código no portable

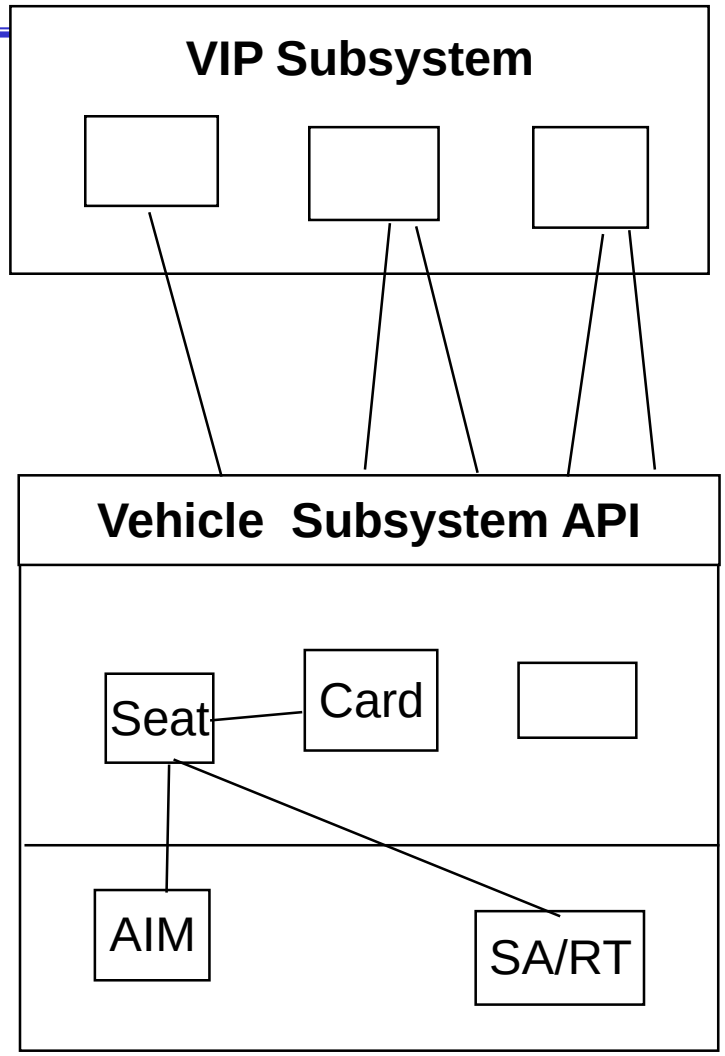


Ejemplo de aplicación (II)

- La estructura ideal de un subsistema se compone de
  - Un objeto interfaz
  - Un conjunto de objetos del dominio de la aplicación (objetos entidad) modelando entidades reales o sistemas existentes
    - Algunos objetos del dominio de aplicación son interfaces a sistemas existentes
  - Uno o más objetos de control
- Podemos utilizar patrones de diseño para realizar la estructura del subsistema
- Realización de un objeto interfaz: patrón *Façade*
  - Proporciona la interfaz al subsistema

Ejemplo de aplicación (III)

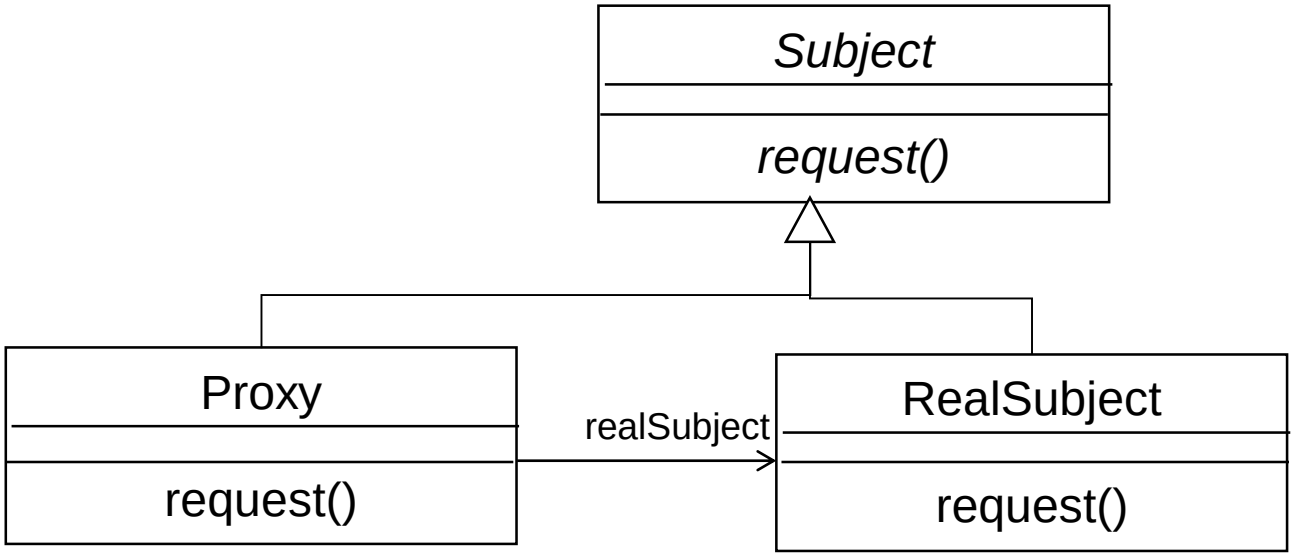
- El subsistema decide exactamente cómo debe ser accedido
- No hay por qué preocuparse acerca de un mal uso de los llamantes
- Si se utiliza un *façade*, se puede utilizar enseguida el subsistema en pruebas de integración
  - Necesitamos escribir solo un conductor



2.5. Patrón *Proxy*

- Motivación
  - Tengo una conexión de Internet intermitente y con velocidad de descarga bastante baja. Intento acceder a un sitio web que incluye gran cantidad de imágenes en una hora punta
  - Problema: Coste de operaciones como la creación de objetos y la inicialización de objetos
- Solución
  - Retrasar la creación y la inicialización al momento en que realmente se necesite
  - Con el patrón *Proxy* se utiliza otro objeto ("el *proxy*") que actúa como representante del objeto real
  - El *Proxy* crea el objeto real solo si el usuario lo pide (reduce el coste de acceder a objetos)

Solución - estructura



- La herencia de interfaz se utiliza para especificar la interfaz compartida por el *Proxy* y el *RealSubject*
- La delegación se utiliza para capturar y redirigir cualquier acceso al *RealSubject* (si se desea)
- También se puede aplicar el patrón utilizando interfaces y realizaciones, en lugar de clases abstractas y herencia

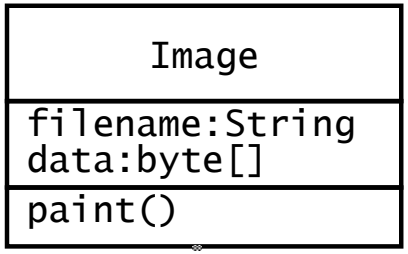
Consecuencias

- El patrón se aplica donde exista al necesidad de referenciar a un objeto de forma más versátil y sofisticada que un puntero
  - Proxy remoto
    - Representante local para un objeto en un espacio de direcciones diferente
    - Cacheo de información para minimizar comunicaciones: bueno si la información no cambia a menudo
  - Proxy virtual
    - El tamaño del objeto es demasiado grande para crear o descargar
    - El *proxy* es un sustituto que retrasa evaluaciones o cálculos costosos
  - Proxy de protección
    - El *proxy* proporciona control de acceso al objeto real
    - Útil cuando diferentes objetos deberían tener diferentes derechos de acceso y visualización al mismo documento

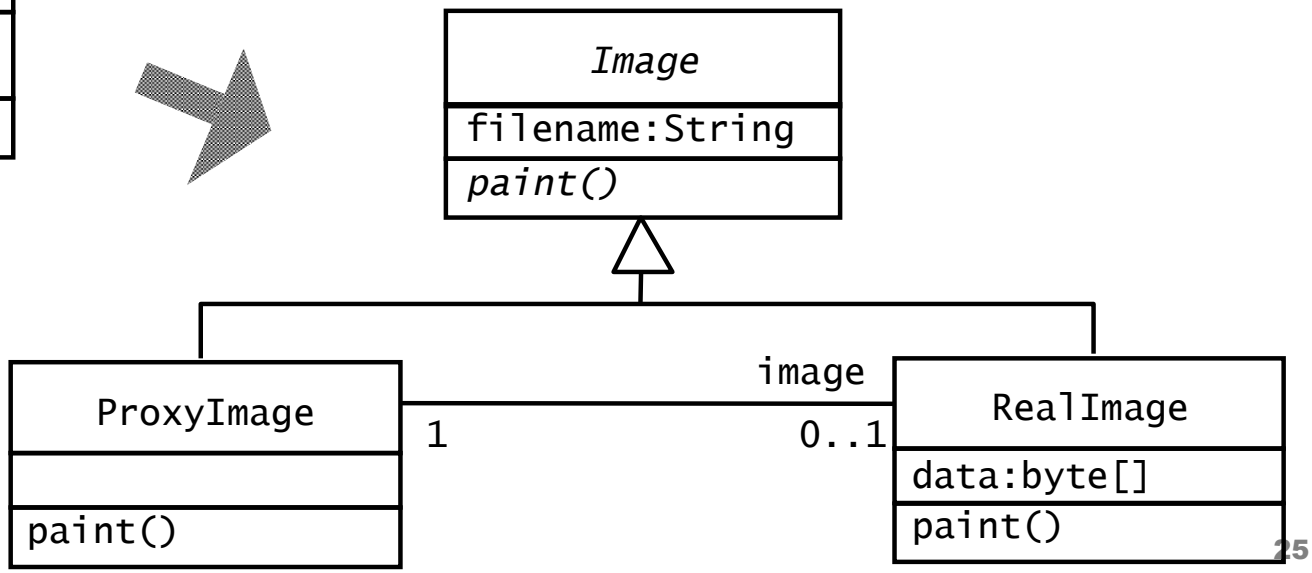
Ejemplo de proxy virtual (I)

- Un programa de visualización muestra un listado de imágenes de alta resolución
  - El programa tiene que mostrar un listado de todas las fotos
  - Pero no necesita mostrar (pintar) la imagen real hasta que el usuario selecciona una imagen del listado

Modelo original

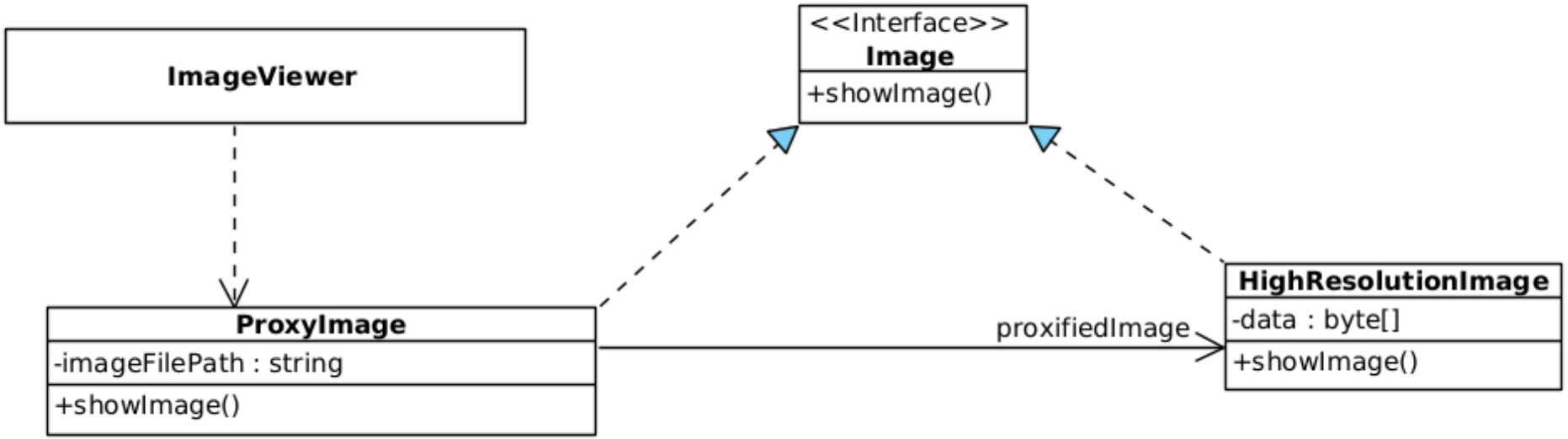


Después de aplicar patrón *Proxy*



Ejemplo de proxy virtual (II)

- También se puede aplicar el patrón utilizando interfaces



Ejemplo de proxy virtual (III)

```
public class ProxyImage implements Image {
    // Private Proxy data
    private String imageFilePath;

     // Reference to RealSubject
    private Image proxifiedImage;

    public ProxyImage (String imageFilePath) {
            this.imageFilePath= imageFilePath;
    }

    public void showImage() {
            // create the Image Object only when the image is required to be shown
            proxifiedImage = new HighResolutionImage(imageFilePath);
            // now call showImage on realSubject
            proxifiedImage.showImage();
    }

}
```

3. Patrones de comportamiento

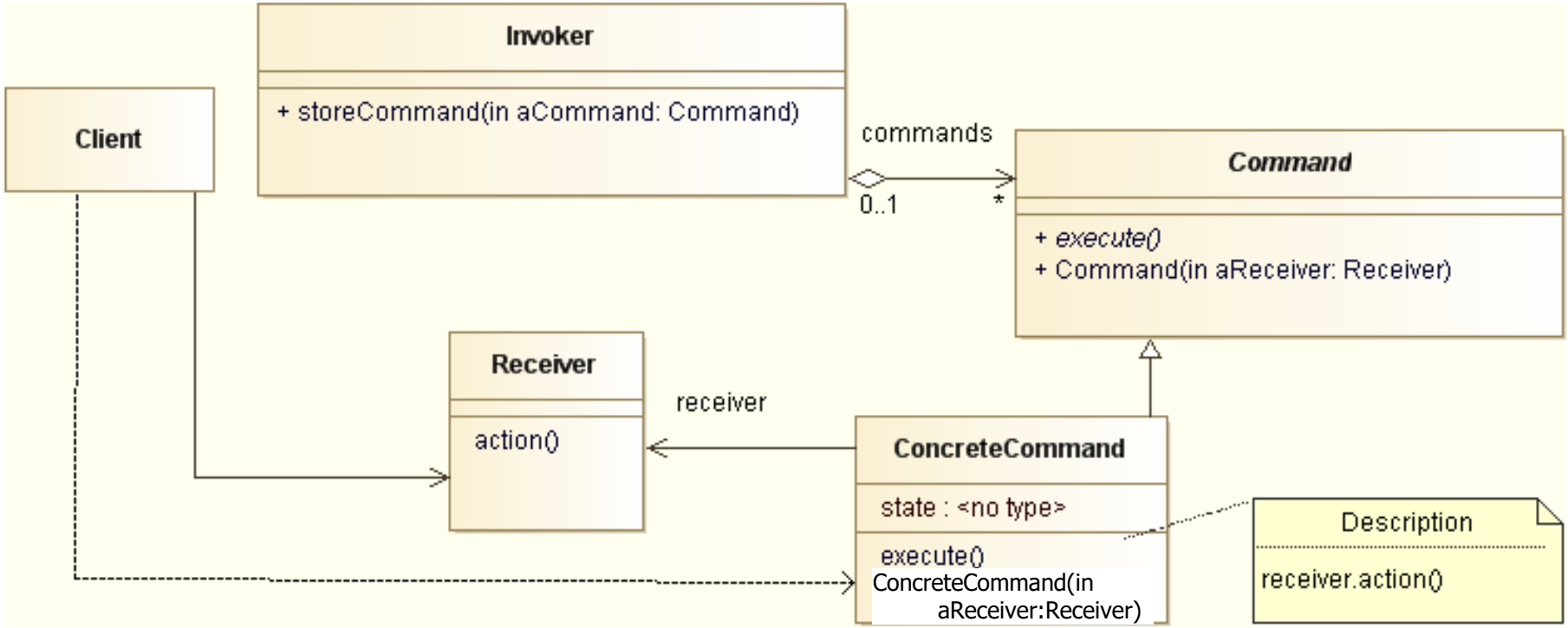
- Describen algoritmos y asignación de responsabilidades a objetos
- Describen patrones de comunicación entre objetos
- Caracterizan flujos complejos de control. Viendo el patrón es fácil comprender el flujo (es difícil comprender el flujo haciendo un seguimiento del mismo en tiempo de ejecución)

Command	Encapsula el flujo de control
Observer	Proporcionan el mecanismo publicar/suscribir
Mediator	Encapsula una interacción compleja entre objetos
Strategy	Soporta una familia de algoritmos, separados de la política y el mecanismo

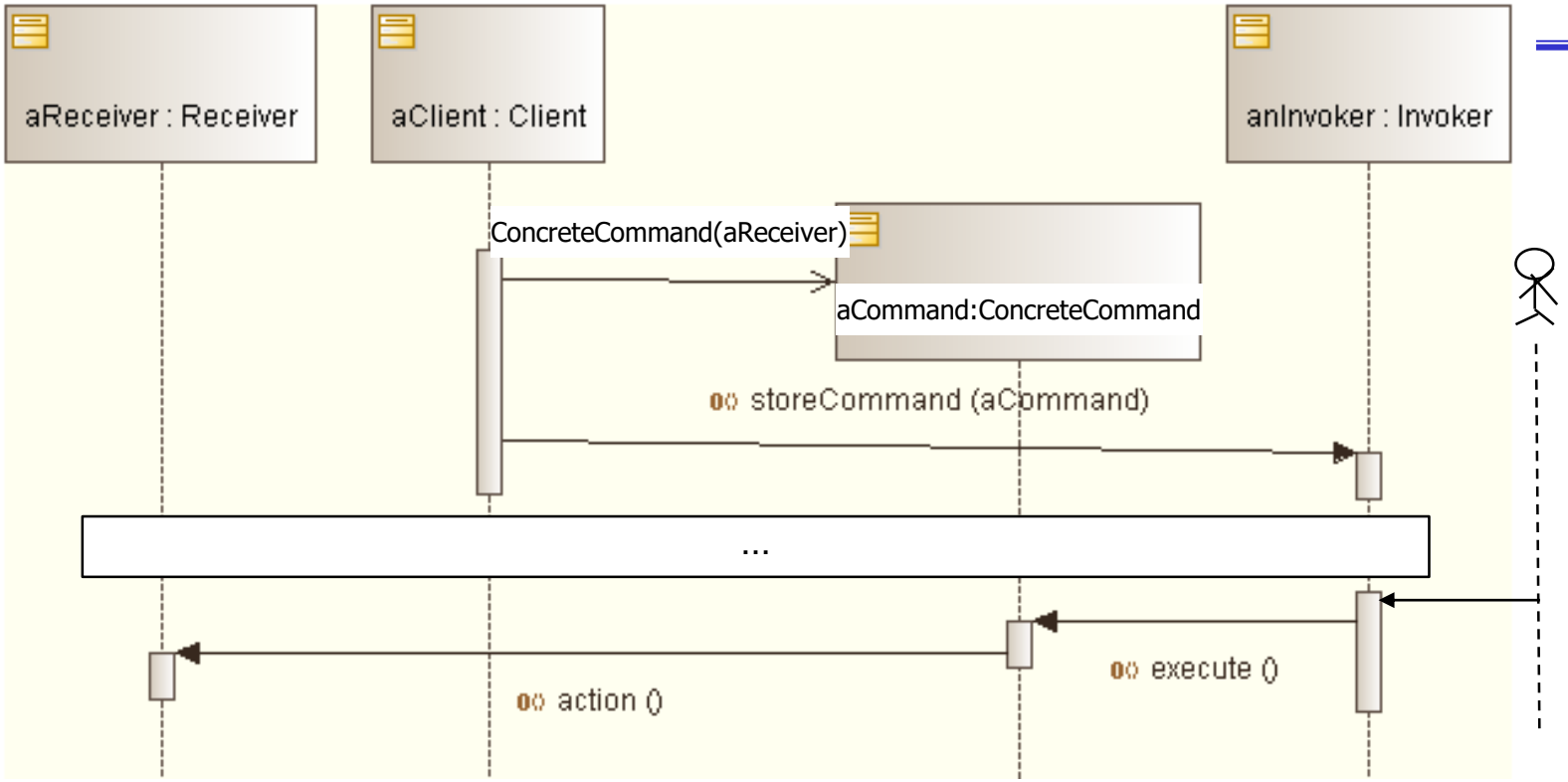
3.1. Patrón *Command*

- Motivación
  - Se quiere construir una interfaz de usuario y facilitar menús
    - Se quiere que la interfaz de usuario sea reusable a través de muchas aplicaciones
      - No se puede codificar el significado de los menús para varias aplicaciones
      - Las aplicaciones solo saben lo que hay que hacer cuando se selecciona el menú
  - Se quiere construir una cola de tareas genéricas
- Solución
  - Este tipo de menús se puede implementar fácilmente con el patrón *Command*

Solución - estructura



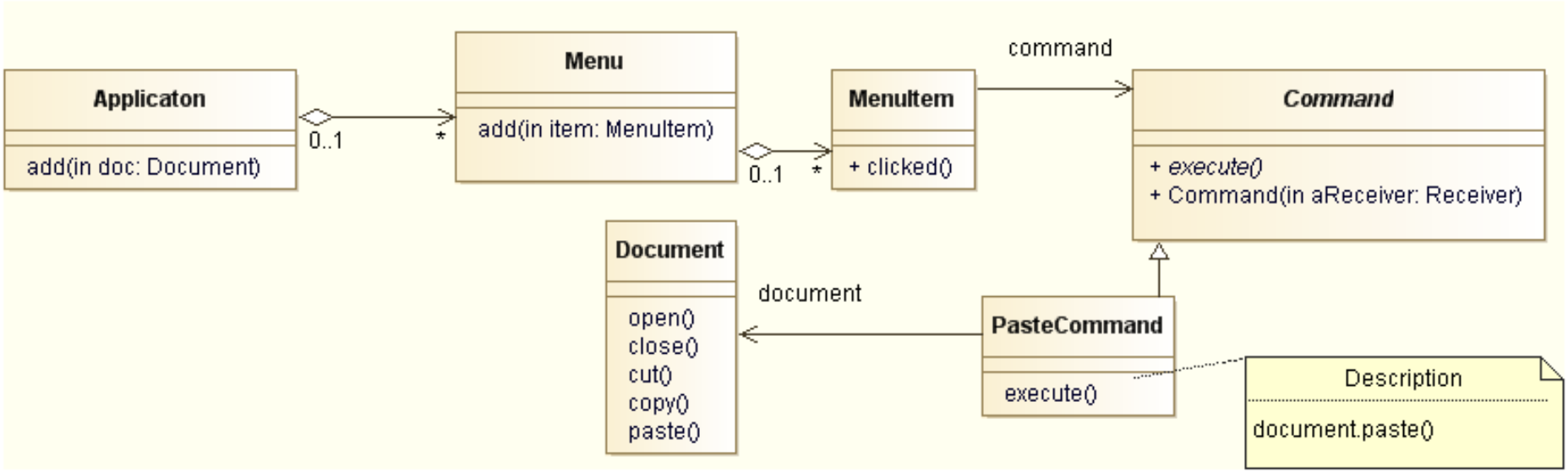
Solución - colaboraciones



- La aplicación cliente (*Client*) crea un *ConcreteCommand* y lo enlaza con el *Receiver* (la clase que sabe ejecutar las operaciones asociadas a una petición)
- La aplicación cliente (*Client*) pasa la referencia del *ConcreteCommand* al Invoker (menú item) que lo almacena
- El *Invoker* tiene la responsabilidad de ("execute" o "undo")

Consecuencias

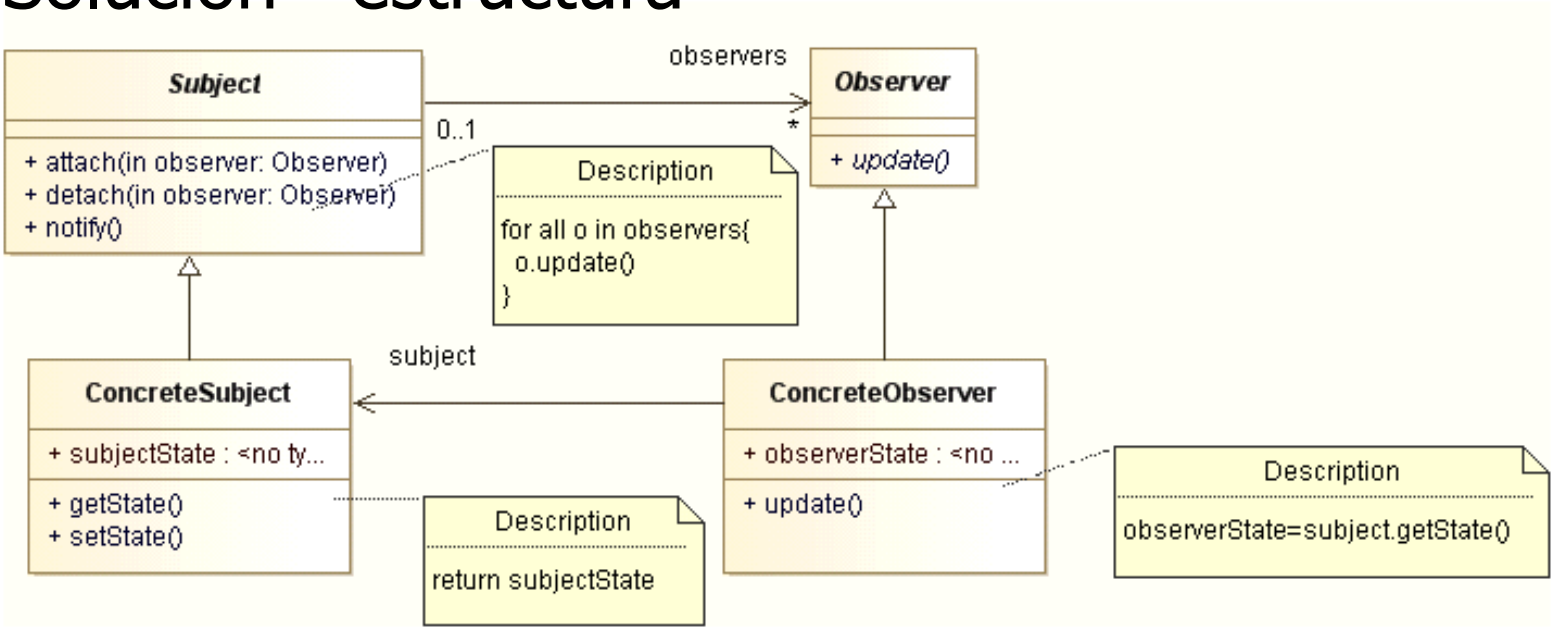
- El patrón *Command* encapsula una petición como un objeto, permitiendo
  - parametrizar los clientes con diferentes peticiones,
  - encolar o registrar las operaciones, y
  - soportar operaciones que se pueden deshacer



3.2. Patrón *Observer* (*Publish and Subscribe*)

- Motivación
  - Definir una asociación 1:n entre objetos de forma que cuando un objeto (sujeto) cambia de estado, todos los objetos dependientes (observadores) son notificados y actualizados automáticamente

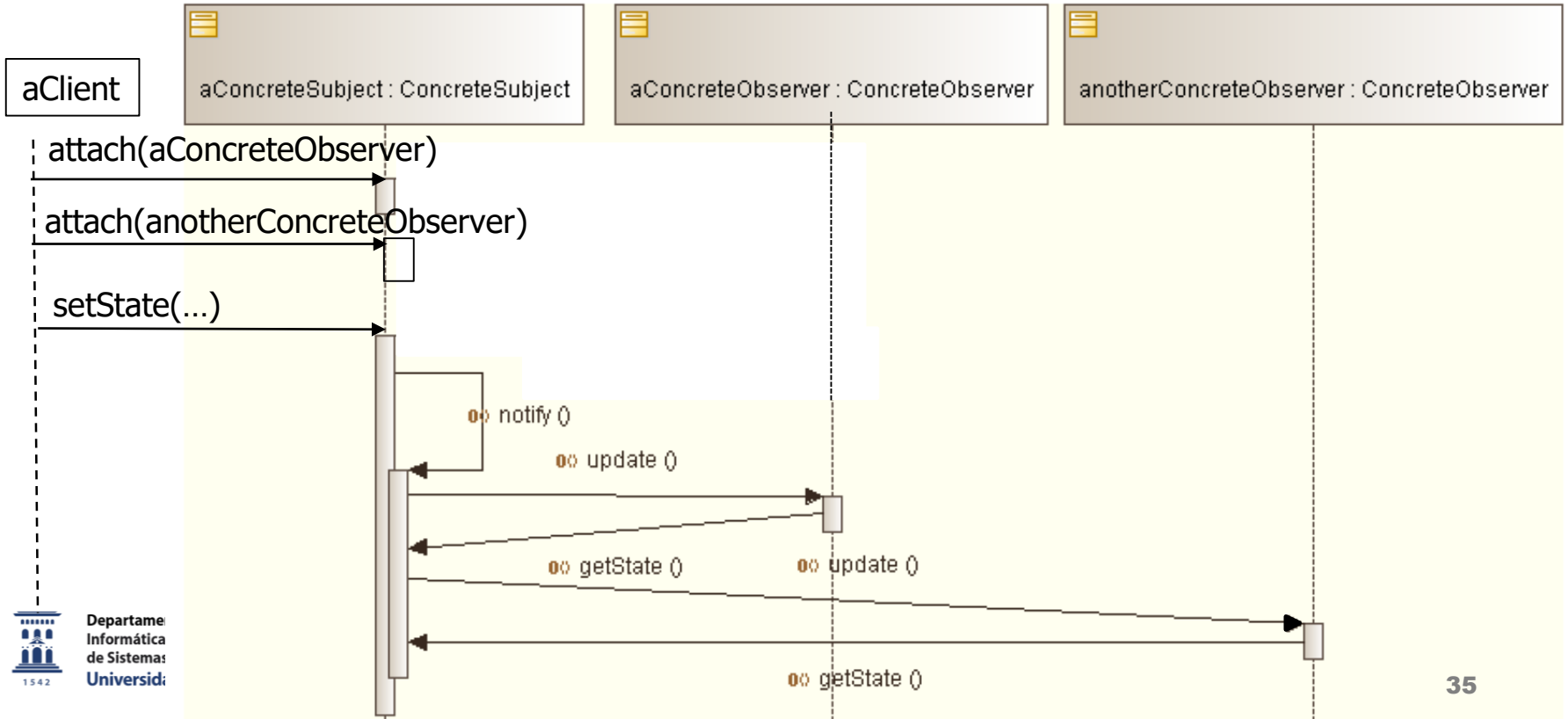
Solución - estructura



- La clave del patrón son dos objetos:
  - Sujeto (*Subject*): mantiene el estado que otros objetos comparten
    - Se debe implementar como una clase (necesita almacenar la lista de observadores)
  - Observador (*Observer*): accede al estado del sujeto (los observadores representan distintas vistas del estado)
    - se puede implementar como una interfaz Java
- El sujeto no sabe si tiene o no observadores. El sujeto sólo conoce al observador abstracto y no conoce los detalles de la clase concreta. Por lo tanto existe un mínimo acoplamiento entre los objetos...

Solución - colaboraciones

- Los observadores se registran con el sujeto en el que están interesados mediante **attach(Observer)**
- Cuando el sujeto cambia, se notifica el cambio a todos los observadores registrados con ese sujeto.
- Los observadores preguntan al sujeto su nuevo estado



Consecuencias

- Separa la visualización del estado de un objeto del objeto mismo y permite visualizaciones alternativas
- Cuando el objeto cambia, se notifica y actualiza automáticamente a todas las visualizaciones para reflejar el cambio

Ejemplo de aplicación

Observers

Subject

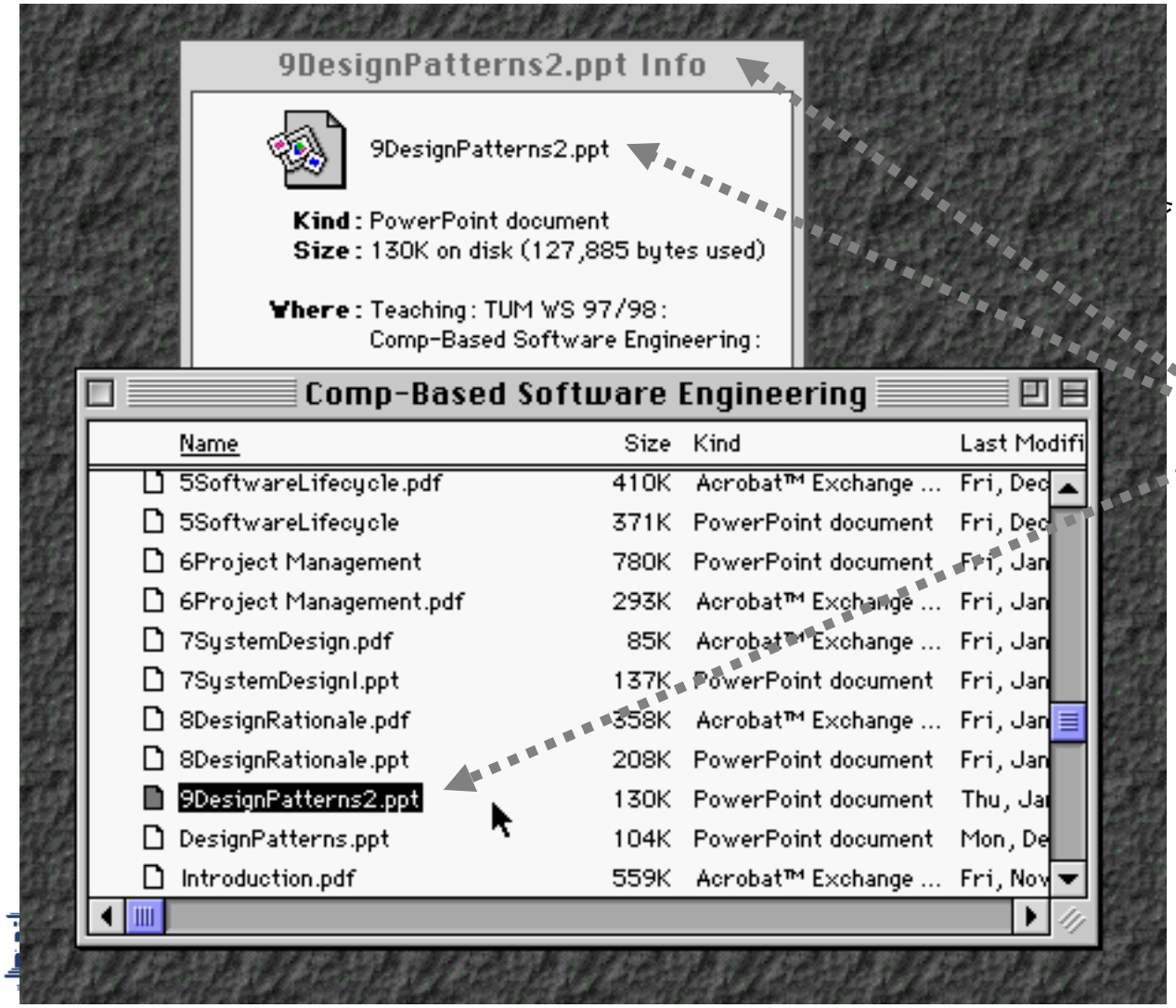
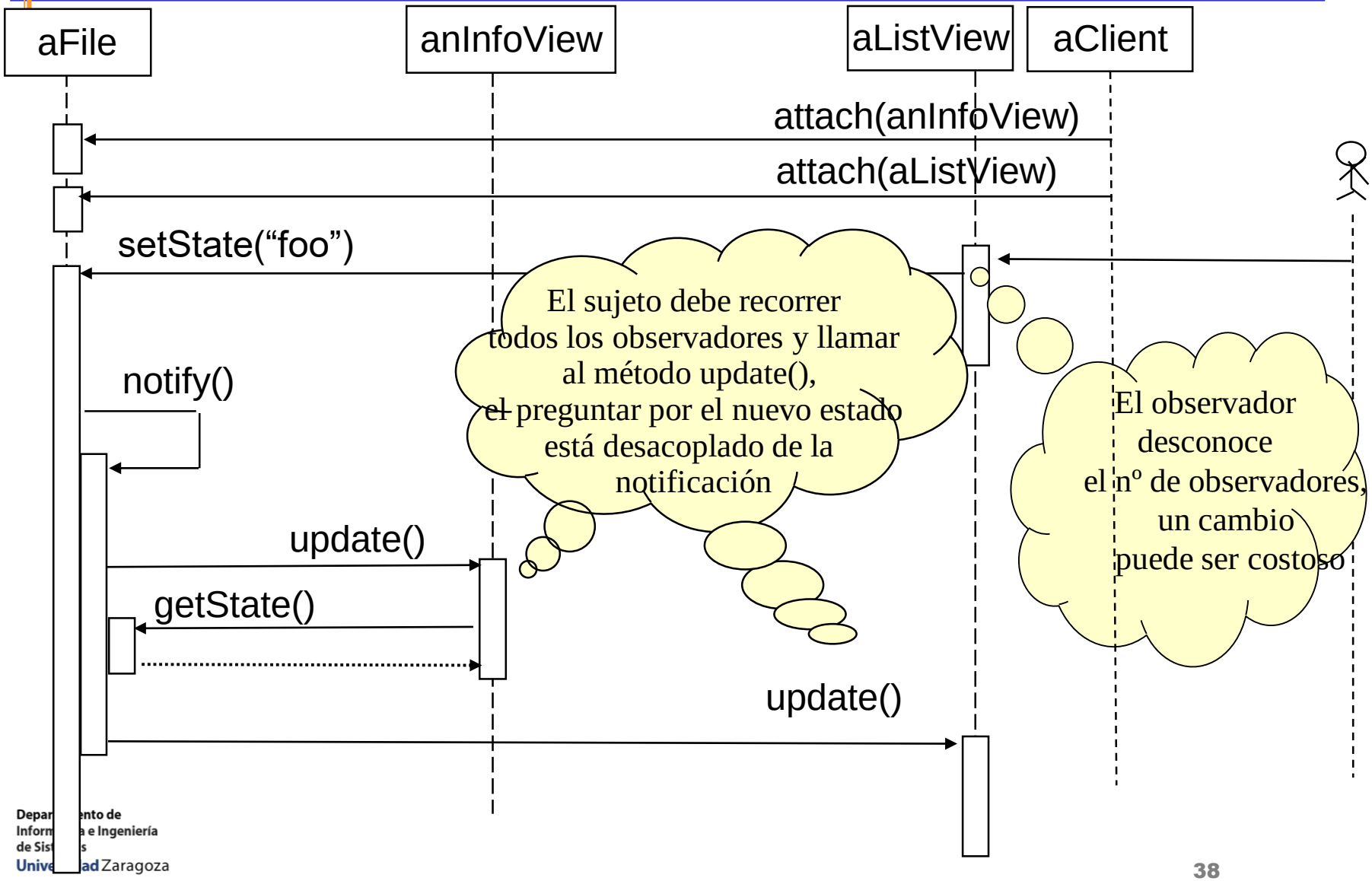


Diagrama de secuencia para el escenario: cambiar el nombre de fichero a “foo”

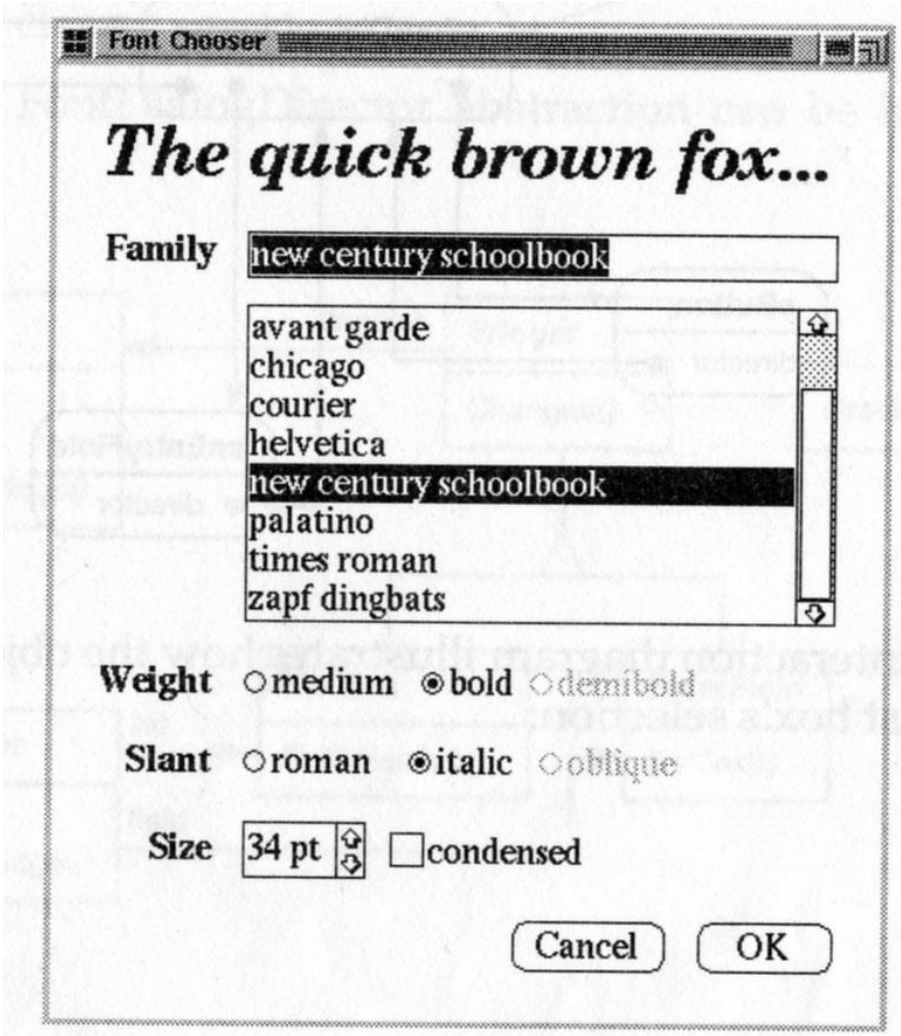


3.3. Patrón *Mediator*

- Motivación
  - En ciertos tipos de sistemas se puede tener una estructura de objetos con muchas conexiones. Al final todos los objetos terminan conociéndose
  - Particionar un sistema en muchos objetos facilita la reusabilidad, pero la proliferación de interconexiones entre ellos la reduce

Ejemplo: Implementación de un diálogo en una GUI

- Existen *dependencias* entre los elementos del diálogo:
  - Seleccionar un elemento en la *list box* cambia el contenido en otro campo
- Reusabilidad
  - Diálogos diferentes (fuentes, imprimir) tendrán dependencias diferentes entre sus elementos
  - Incluso utilizando los mismos elementos, NO se pueden reutilizar las clases
  - Adaptar las clases para contemplar las *dependencias específicas* del diálogo → Tedioso

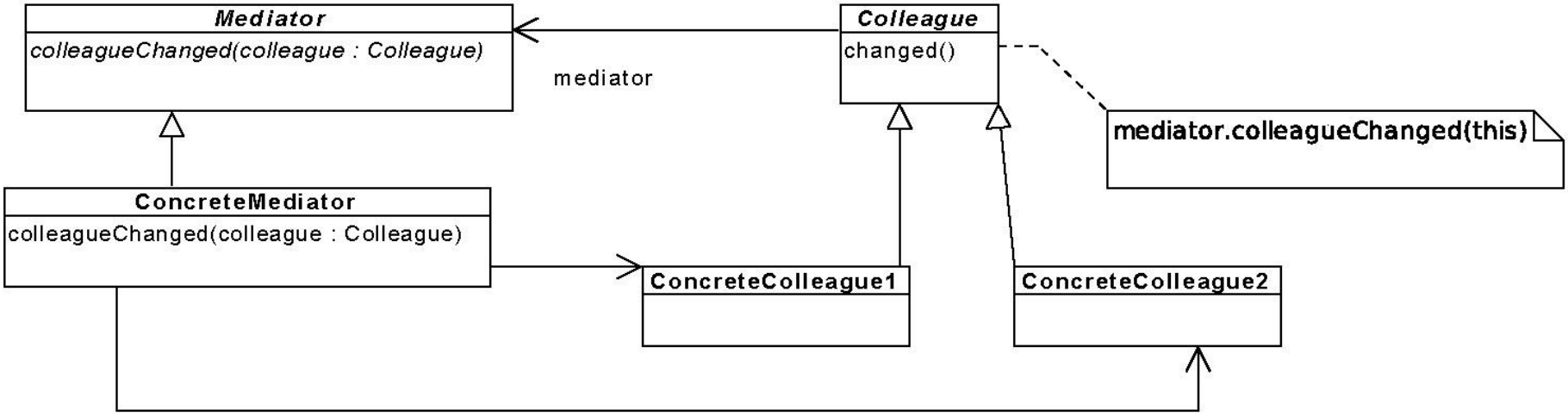


Solución

- Definir un objeto que encapsula cómo interaccionan un conjunto de objetos
  - Será responsable de coordinar y controlar las interacciones
- Promueve un bajo acoplamiento evitando que los objetos se comuniquen (interaccionen) entre sí explícitamente
- Permitirá modificar las interacciones

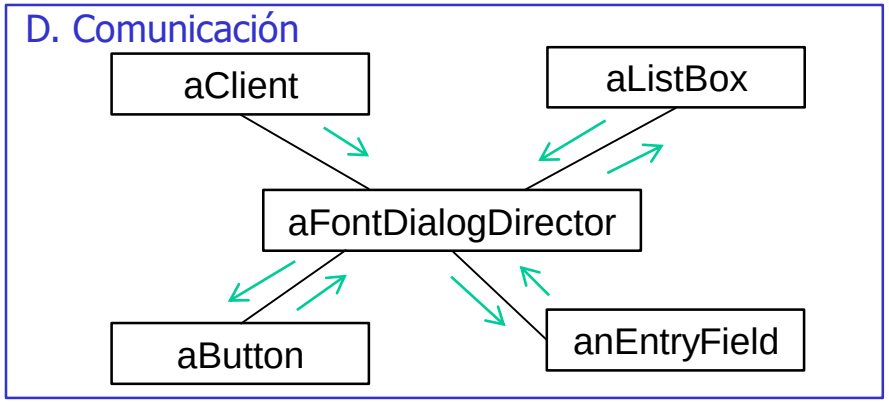
Solución - estructura

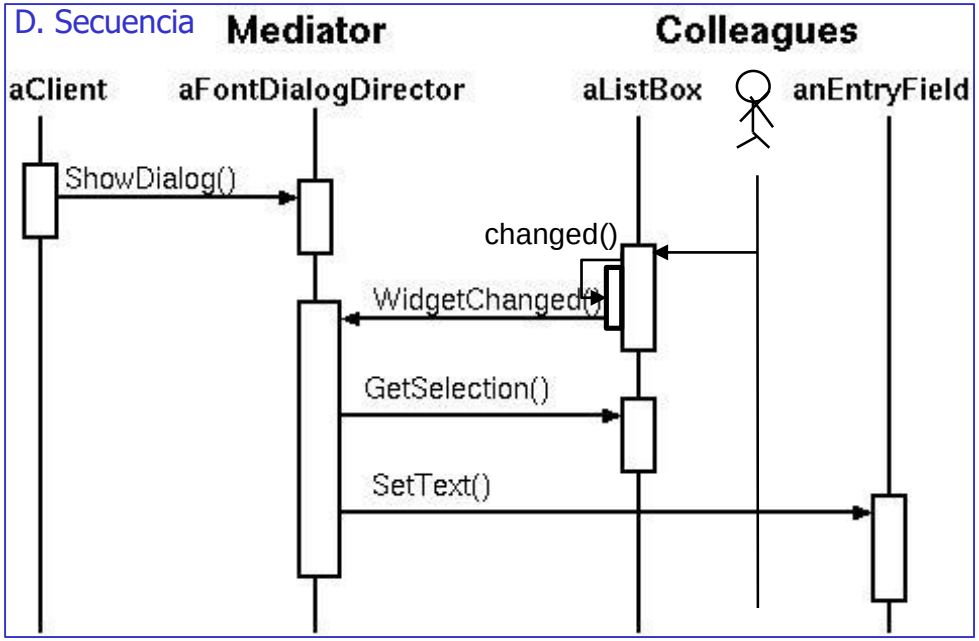
- Mediator: Define la interfaz para comunicar objetos colegas
- ConcreteMediator
  - Implementa el comportamiento cooperativo coordinando objetos colegas
  - Conoce y mantiene a los colegas que dependen de él
- Colleague
  - Cada clase colega conoce a su objeto mediador
  - Cada colega comunica con su mediador cuando debería comunicarse con otro colega



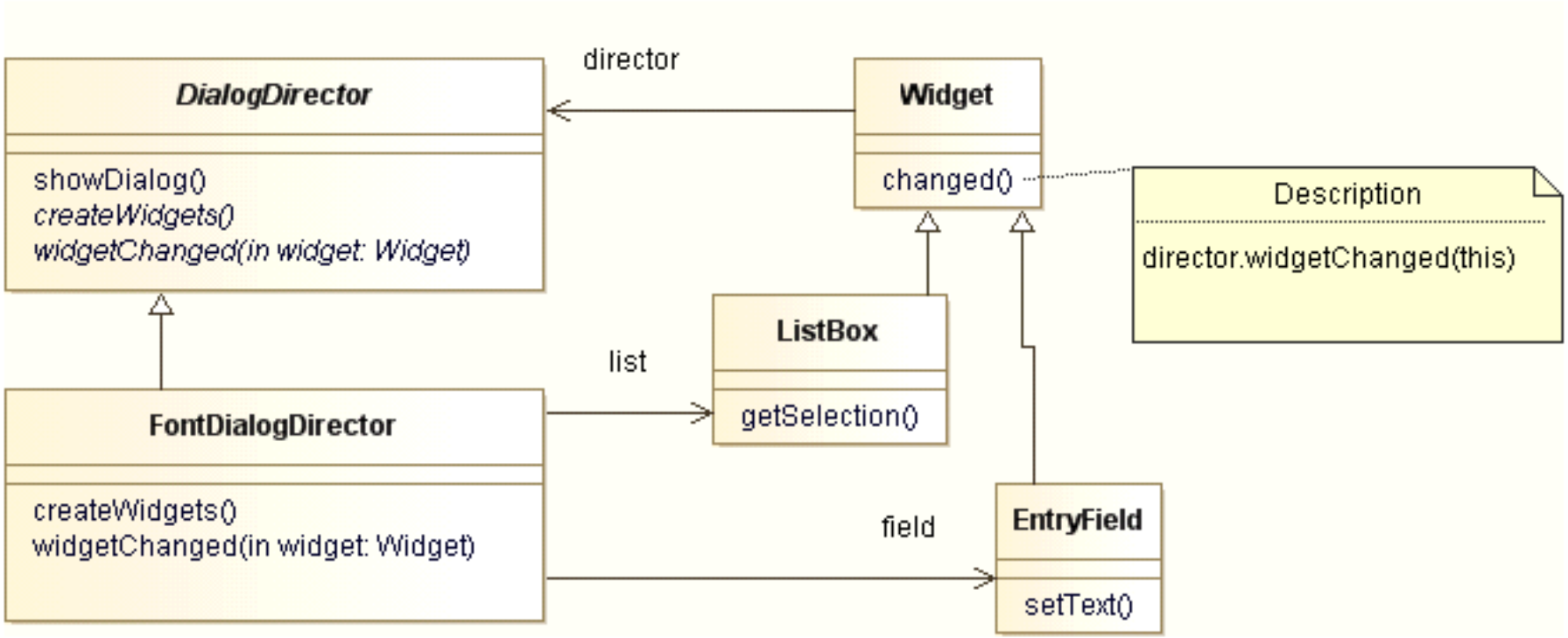
Aplicabilidad del patrón al ejemplo (I)

- El mediador puede ser el mismo Diálogo
  - Conoce sus widgets y coordina sus interacciones
  - Actúa como centro de comunicaciones
  - El diálogo media entre la list box y el campo de texto (se comunican de forma indirecta)





Aplicabilidad del patrón al ejemplo (II)



Notas:

- Se puede omitir la clase abstracta cuando las clases colega trabajan con un único mediador
- La clase abstracta mediador introduce un acoplamiento abstracto que posibilita a los colegas usar diferentes mediadores

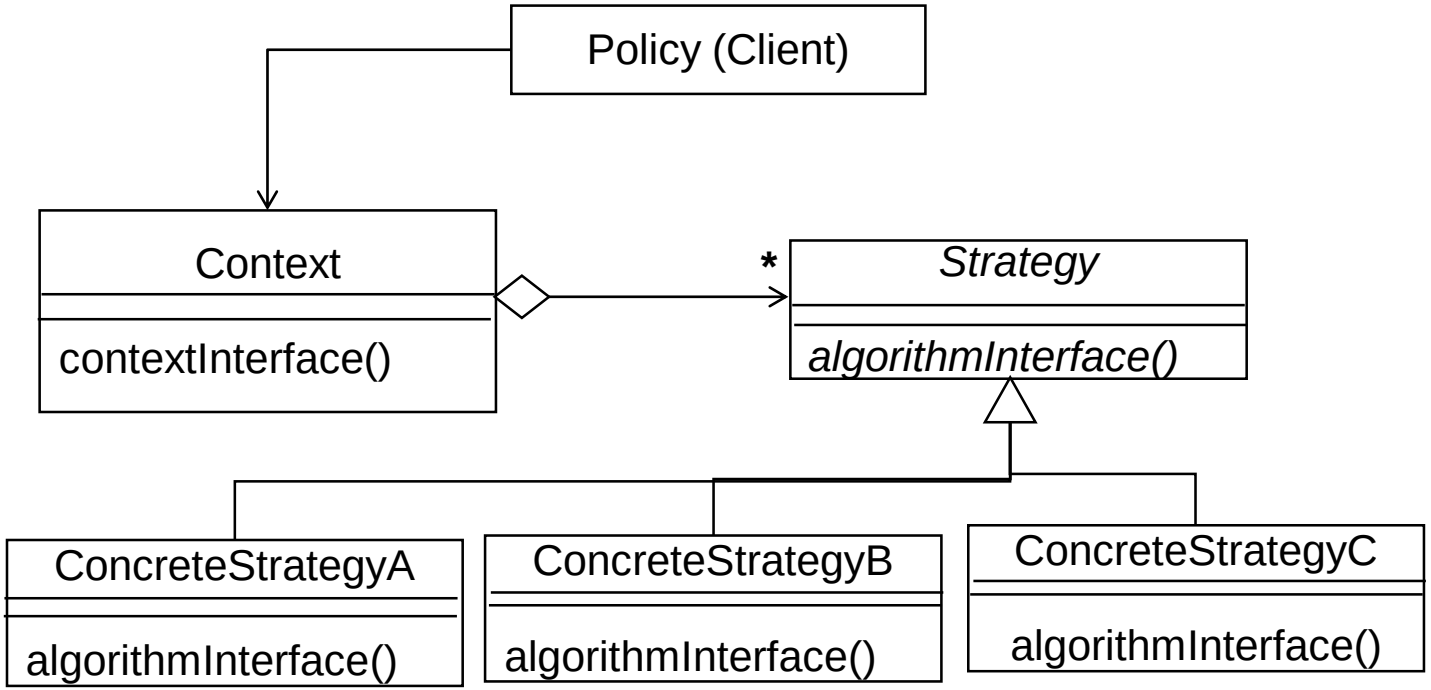
Consecuencias

Ventajas	***Desventajas***
Abstrae cómo cooperan los objetos	Centraliza el control Como el mediador encapsula toda la comunicación puede convertirse en algo *monolítico* y difícil de mantener
Desacopla a los colegas se pueden reutilizar independientemente las clases mediador y colegas	
Simplifica los protocolos de los objetos reemplaza comunicaciones muchos a muchos por comunicaciones uno a muchos que son más fáciles de entender y mantener	
Limita la generación de subclases si se necesita extender el comportamiento, solo tengo que especializar el mediador (en ausencia del patrón, tendría que haber extendido cada colega)	

3.4. Patrón *Strategy*

- Motivación
  - Existen muchos algoritmos para la misma tarea
    - Ej: Partir un flujo de texto en líneas, Ordenar la lista de clientes
  - Los diferentes algoritmos pueden ser apropiados en distintos momentos
    - Ej: prototipado rápido vs entrega final del producto
  - No queremos soportar todos los algoritmos si no los necesitamos
  - Si necesitamos un nuevo algoritmo, queremos añadirlo fácilmente sin que sea una molestia para la aplicación que usará el algoritmo
- Solución
  - Definir una interfaz común para los algoritmos soportados y elegir el algoritmo según el contexto

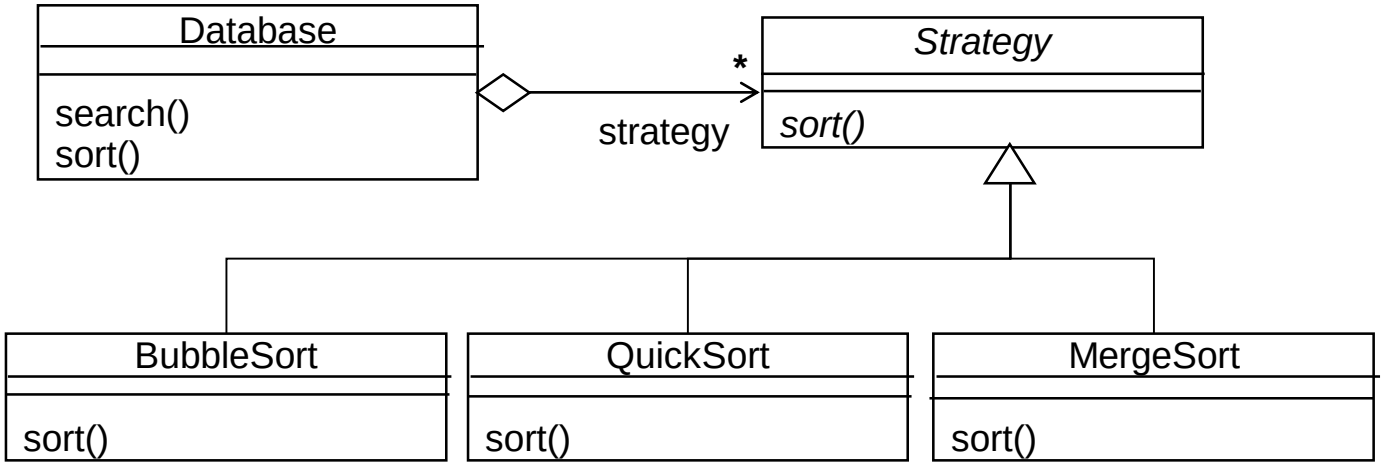
Solución - estructura



- El objeto *Context* recibe peticiones del cliente (*Policy / Client*) y delega el trabajo en los objetos de tipo estrategia.
- Normalmente, la estrategia concreta es creada por el cliente y se envía al contexto (objeto *Context*). Desde este punto, el cliente interacciona solo con el contexto.

Consecuencias

- El patrón estrategia permite unificar el interfaz del servicio para el cual se ofrecen distintas alternativas (algoritmos)
- Las diferentes variantes del algoritmo se pueden implementar como una jerarquía de clases
- Ejemplo de aplicación:
  - Distintas alternativas para la ordenación en una base de datos dependiendo de los requisitos de espacio y procesamiento (tiempo de ejecución)



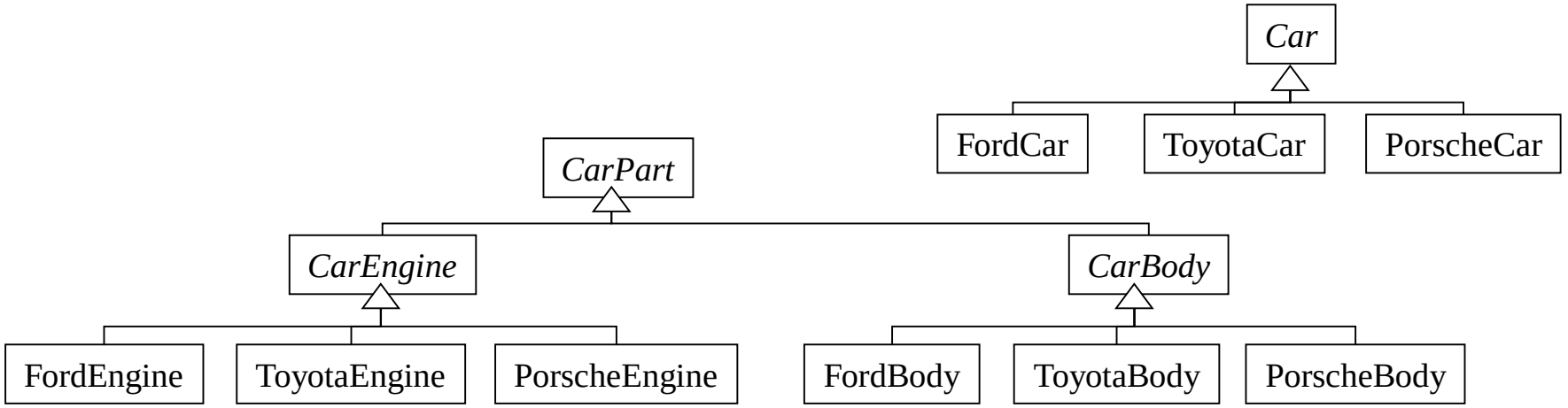
4. Patrones de creación

- Abstraen el proceso de instanciación
- Ayudan a que el sistema sea independiente de cómo se crean, componen y representan los objetos.
- Flexibilizan:
  - qué se crea, quién lo crea, cómo se crea, cuándo se crea
- Permiten configurar un sistema:
  - estáticamente (*compile-time*), dinámicamente (*run-time*)

Abstract Factory	Facilita la independencia respecto al proveedor (implementador)
Singleton	Solo necesitamos una instancia

4.1. Patrón Abstract Factory

- Motivación
  - Construir un coche a partir de un motor, una carrocería, ...
    - todos los componentes de la misma marca (familia)
    - hay múltiples marcas (Ford, Toyota, Opel, ...)
    - es responsabilidad del cliente ensamblar las piezas



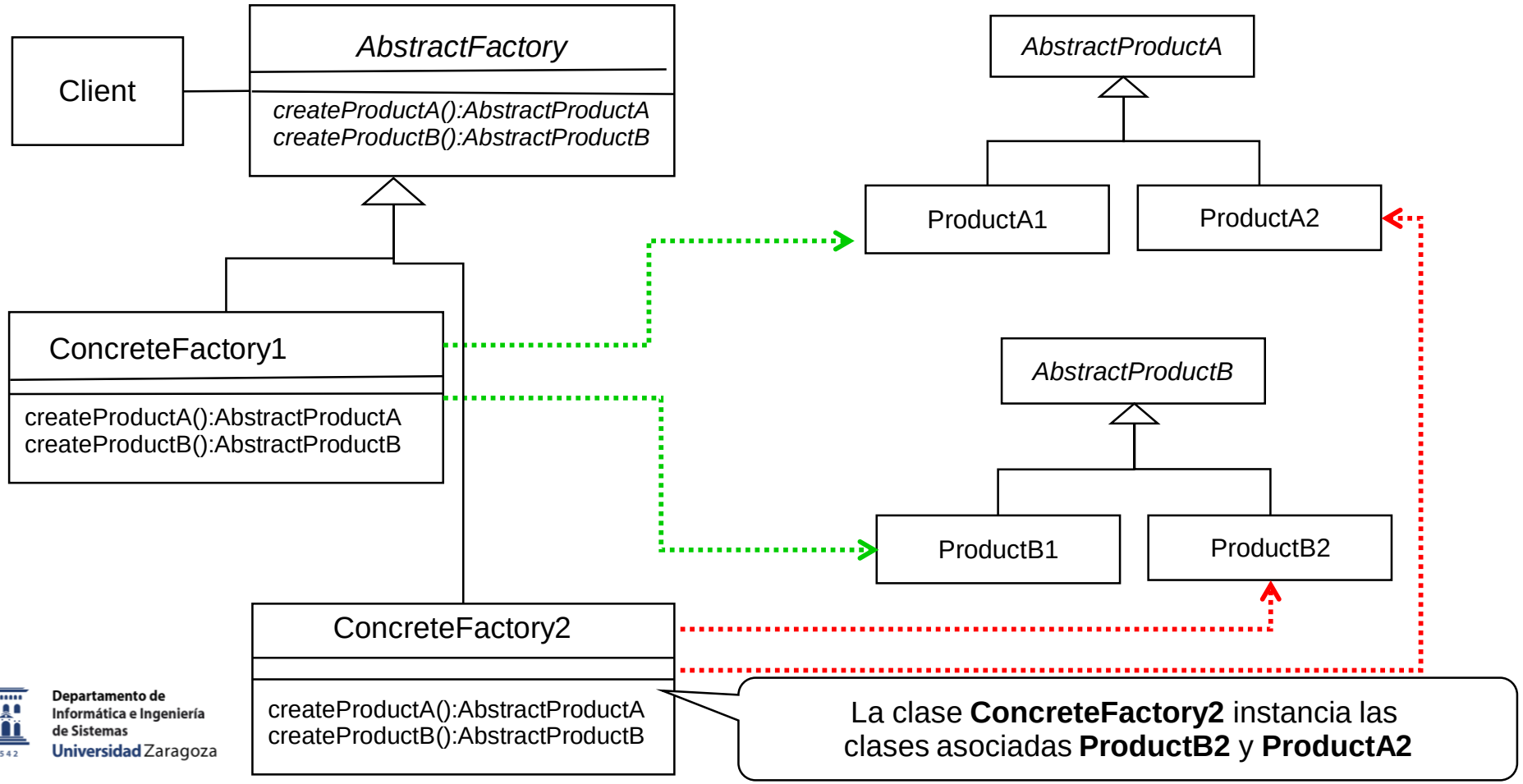
… motivación

- ¿Cómo construiríamos un coche de una marca en un método del cliente?
- ¿Qué ocurre si queremos añadir una nueva marca?
  - Modificar *createCar* en el cliente
  - Modificar las jerarquías anteriores
- ¿O si queremos añadir un nuevo elemento?

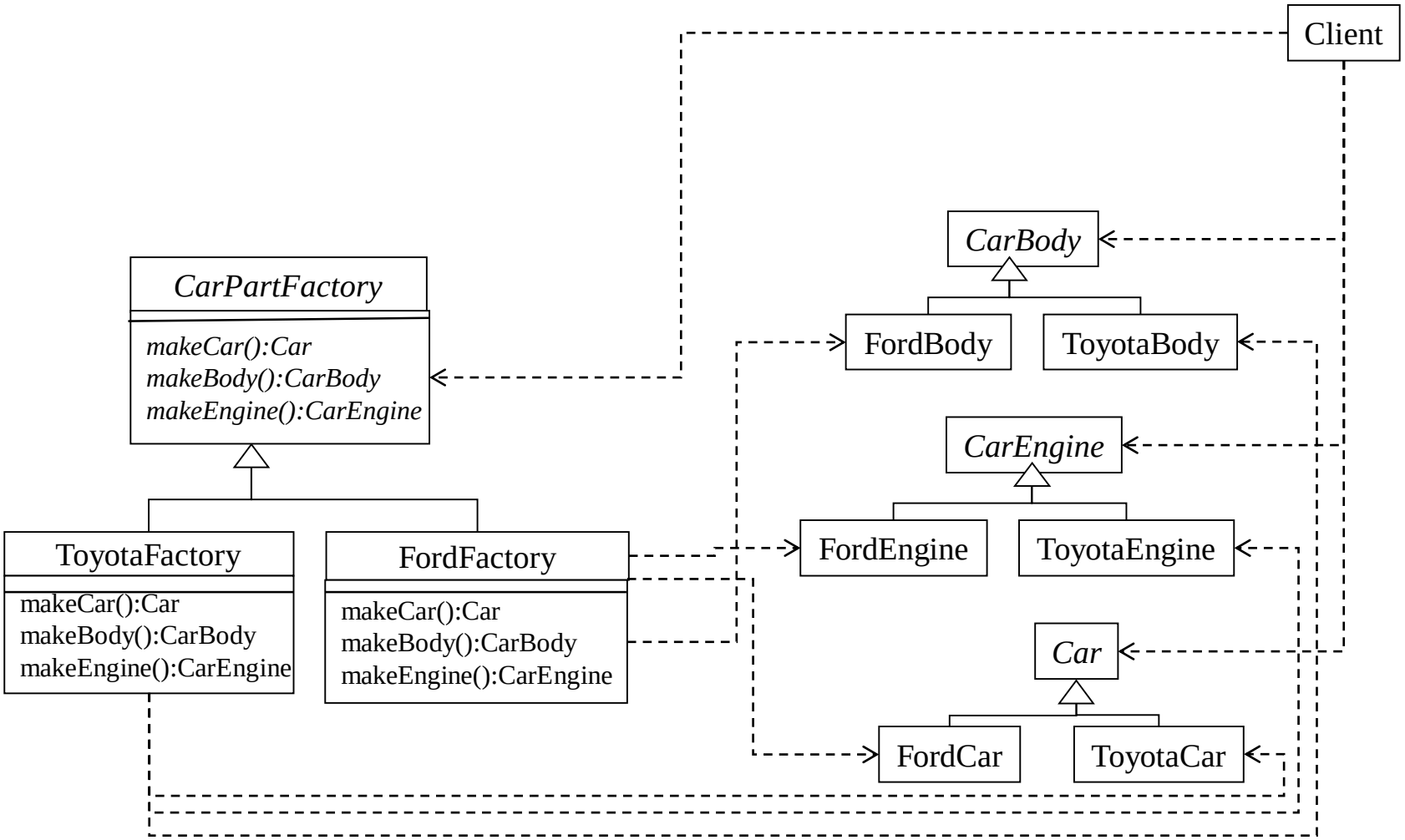
```
Car createCar (String marca) {
  Car coche;
  CarEngine motor;
  if (marca.equals("ford"))
    coche = new FordCar ()
  else
    if (marca.equals("toyota"))
      coche = new ToyotaCar();
    else …;
  if (marca.equals("ford")
    motor = new FordEngine();
  else
    if (marca.equals("toyota"))
      motor = new ToyotaEngine();
    else …;
    coche.addEngine (motor);
    ...
    return coche;
}
```

Solución - estructura

- Se permite al cliente crear productos en cualquier familia de productos, sin especificar sus clases concretas



Ejemplo de aplicación del patrón



Consecuencias

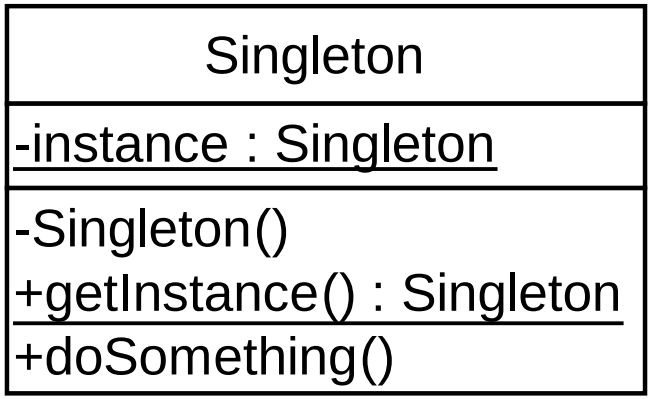
- Se independiza al cliente de cómo se crean/representan los productos
- Utilizando el patrón se fuerza a que todos los productos sean de la misma familia, una vez establecida ésta por el cliente
  - El patrón permite separar la elección de la familia (la marca del coche) del proceso de instanciar las partes.
  - En el cliente:
- Posible problema:
  - Decidir ofrecer un nuevo producto requiere modificar *AbstractFactory* y sus subclases

```
//clase cliente
class Client {
  CarPartFactory familia;
  Car createCar () {
    Car coche=familia.makeCar();
    coche.addEngine(familia.makeEngine());
    coche.addBody (familia.makeBody ());
    ...
    return coche;
  }
  void setFactory (CarPartFactory f) {
    familia=f;
  }
}
```

4.2. Patrón *Singleton*

- Motivación
  - Hay situaciones donde sólo debe existir una instancia:
    - Una única cola de impresión
    - Un único sistema de ficheros
- Solución
  - Ofrecer una clase que asegure que hay una sola instancia y proporcione un punto de acceso global a dicha instancia
  - ¿Cómo asegurarse de que la clase tiene una única instancia?
    - Actuar sobre el método de creación de instancias

Solución - estructura




```
public class Singleton {
      private static Singleton instance;

      private Singleton() {
                ...
      }

      public static synchronized Singleton getInstance() {
                if (instance == null)
                          instance = new Singleton();

                return instance;
      }
      ...
      public void doSomething()
      {
                ...
      }
}
```

5. Pistas para identificar patrones de diseño (I)

- Los requisitos no funcionales pueden dar una pista para el uso de patrones de diseño
- Leer el establecimiento del problema de nuevo. Utilizar pistas textuales para identificar patrones de diseño

Texto	***Patrón***
"independiente del proveedor", "independiente del dispositivo", "debe soportar una familia de productos"	Abstract Factory
"debe interaccionar con un objeto existente"	Adapter
"debe interaccionar con varios sistemas, algunos de ellos a desarrollarse en el futuro", "se debe demostrar con un prototipo rápido"	Bridge

5. Pistas para identificar patrones de diseño (II)

Texto	***Patrón***
“estructura compleja”, “profundidad y anchura compleja”	Composite
“debe interaccionar con un conjunto de objetos existentes” (en general, un patrón facade debería usarse en todos los subsistemas de un sistema software para definir los servicios del subsistema)	Façade
“debe ser transparente a la localización”	Proxy
“debe ser extensible”, “debe ser escalable”, “vistas alternativas de un mismo objeto”	Observer
“debe proporcionar una política independiente del mecanismo”	Strategy