Ingeniería del Software

curso 2018-19

Celia Rubio Madrigal

Universidad Complutense de Madrid

7 de junio de 2019

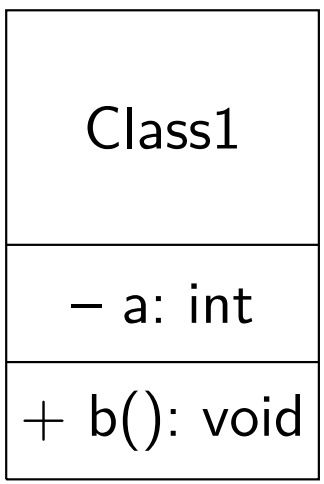
Índice

Índice

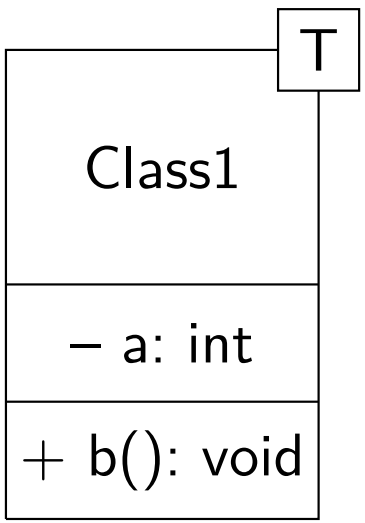
Diagramas de clase

Diagramas de clase
Entidades

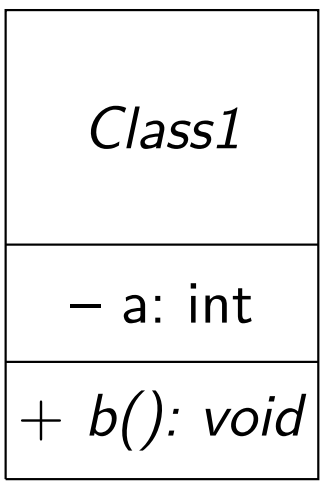
Clase



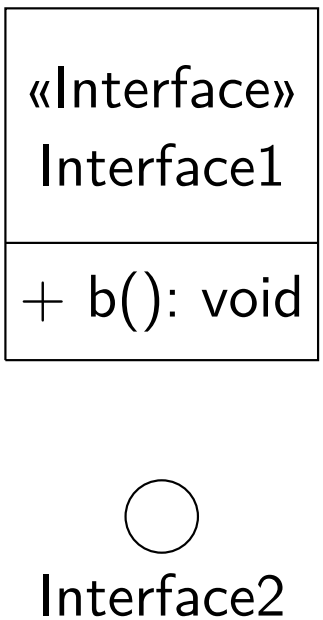
Clase con parámetros



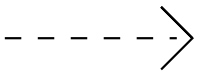
Clase abstracta

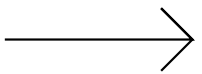


Interfaz

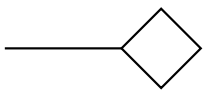


Diagramas de clase
Relaciones

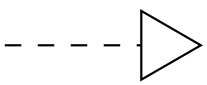

Dependencia y uso

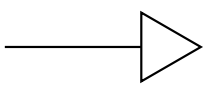

Asociación unidireccional

Asociación bidireccional


Agregación (“parte de”)


Composición (sin uno no hay el otro)


Interfaces e instancias


Herencia (“tipo de”)

Diagramas de clase
Otros

Visibilidad de atributos y métodos:

- Public: +
- Protected: #
- Private: –

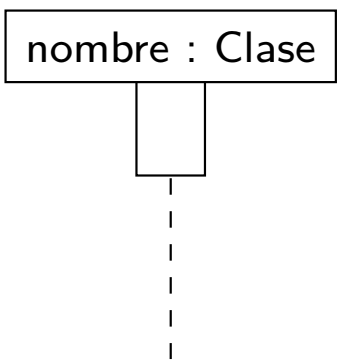
Multiplicidad de las relaciones:

- Solo M objetos: `M`
- Múltiples objetos (listas): `0..N`, `1..*`, `*`
- En general: `min..max`

Diagramas de secuencia

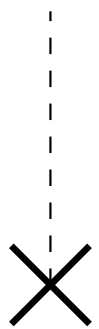
Diagramas de secuencia
Elementos

Objeto

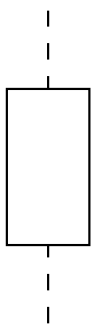


Actor

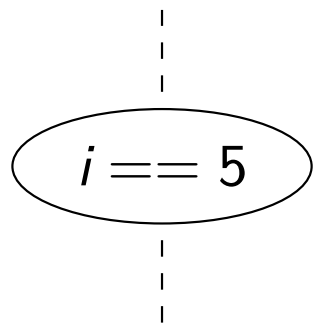
Línea de vida



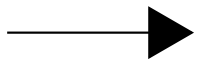
Caja de activación

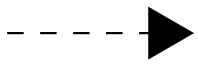


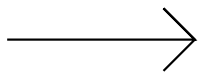
Invariante de estado

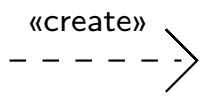


Diagramas de secuencia
Mensajes

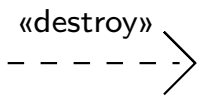

Síncrono


Retorno


Asíncrono

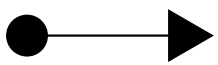


Crear instancia



Destruir instancia


Pérdida de mensaje


Mensaje encontrado

Diagramas de secuencia
Marcos de interacción

1. Opcional (if): `opt[condición]`

2. Alternativas (switch): `alt[condición]`

3. Bucles (while): `loop` `[while condición]`

4. Paralelo (al mismo tiempo): `par`

Arquitectura

Arquitectura
Capas

1. **Capa de Presentación:** Gestionan las interfaces con el usuario, es decir, la lógica de presentación necesaria para dar servicio a los clientes que acceden al sistema
2. **Capa de Negocio:** Implementan las reglas de gestión de datos y servicios de la aplicación. Todo el procesamiento.
3. **Capa de Integración:** Comunican a los otros componentes con recursos y sistemas externos, como por ejemplo, la BBDD (base de datos).

Arquitectura de una capa

La arquitectura de una capa no divide al sistema en presentación, negocio e integración (todas ellas están mezcladas).

Ventajas:

- Sencillez conceptual.

Inconvenientes:

- No se puede modificar ni la interfaz de usuario, ni la lógica del negocio, ni la representación de los datos sin afectar a las demás capas.
- Complicación del proceso.

Arquitectura de dos capas

La arquitectura de dos capas diferencia entre la capa de presentación y el resto del sistema (no diferencia negocio de integración).

Ventajas:

- Permite cambios en la interfaz de usuario o en el resto del sistema sin interferencias mutuas.
- Simplificación del proceso.

Inconvenientes:

- Mayor complicación arquitectónica que la arquitectura de una capa.
- No se puede modificar la lógica del negocio o la representación de los datos sin interferencias mutuas.

Arquitectura multicapa

Diferencia las capas de presentación, negocio e integración.

Ventajas:

- Encapsulación.
- Distribución y particionamiento.
- Desarrollo independiente y rápido.
- Empaquetamiento y configurabilidad.
- Integración y reusabilidad.
- Escalabilidad.
- Mejora de la fiabilidad.
- Manejabilidad.

Inconvenientes:

- Mayor complejidad arquitectónica.
- Posible pérdida de rendimiento.
- Riesgos de seguridad.
- Gestión de componentes.

Patrones

Modelo-Vista-Controlador

- **Modelo:** parte lógica. Maneja el estado interno y la funcionalidad.
- **Vista:** muestra y recoge la información del usuario.
- **Controlador:** media entre vistas y modelo y maneja eventos. Puede haber varios controladores según el usuario pueda manejar la app.

Tipos:

- **Activo:** el modelo sabe cómo son las vistas y a veces las actualiza.
- **Pasivo:** el modelo solo se comunica con el controlador.

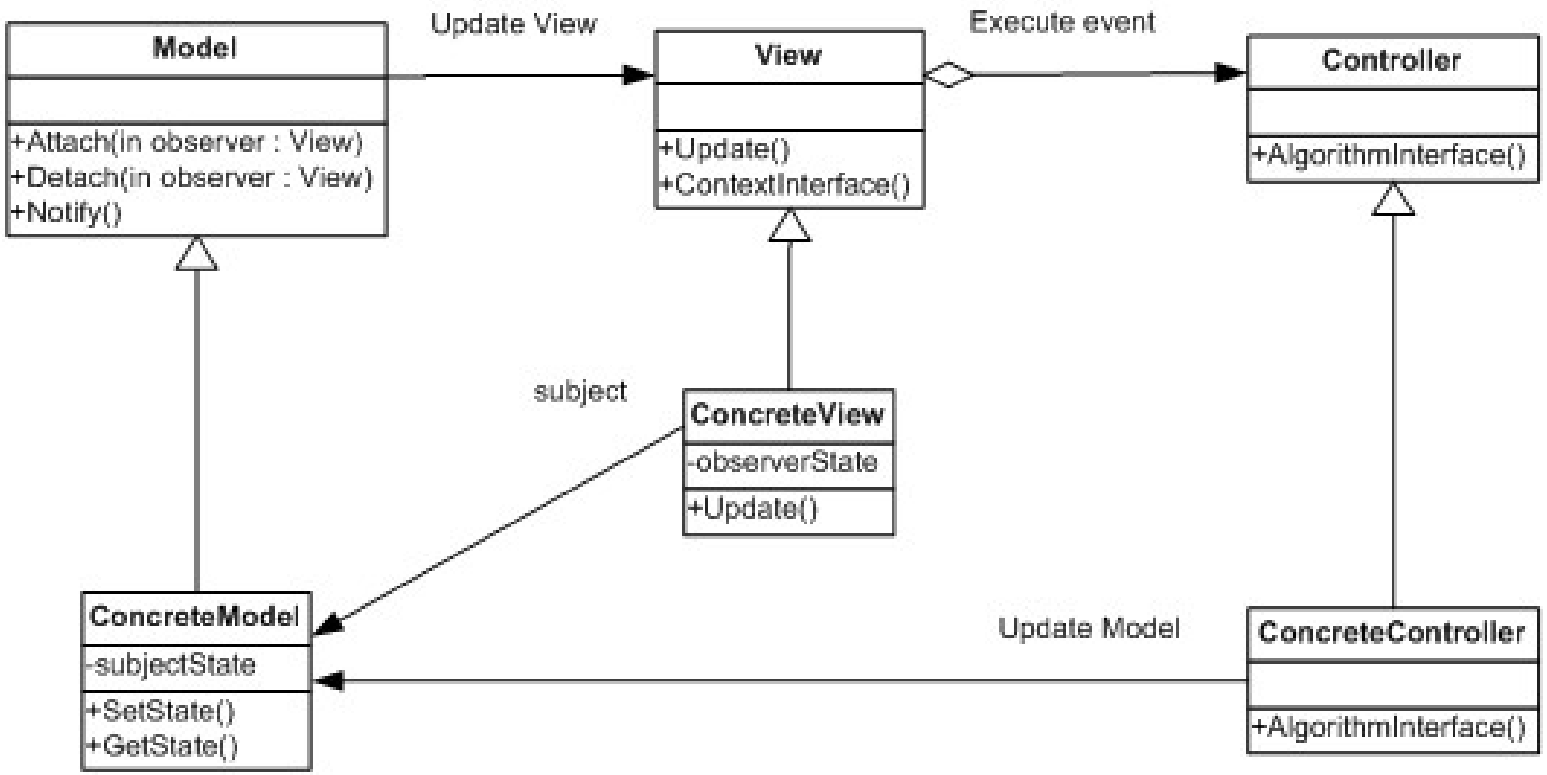
Ventajas:

Reutilizar código.
Componentes separados para trabajo simultáneo.
Desacoplamiento. Posibilidad de tener diferentes vistas. Es fácil modificar.

Desventajas:

Complejidad adicional de clases. Puede que unas partes se alejen mucho de otras. Es más complicado de programar.

Diagrama de clases:



Servicio de Aplicación

Problema:
En arquitectura multicapa, hay que asegurar que no se desdibujen las fronteras entre capas. ¿Dónde se instancian las clases de Negocio? El Controller es de Presentación.

Solución:
Clase `SA` (Interfaz e instanciación) puente entre capas. Los eventos de Presentación pasan por SA y se actúa llamando a Negocio. También guarda DAOs y llama a Integración.

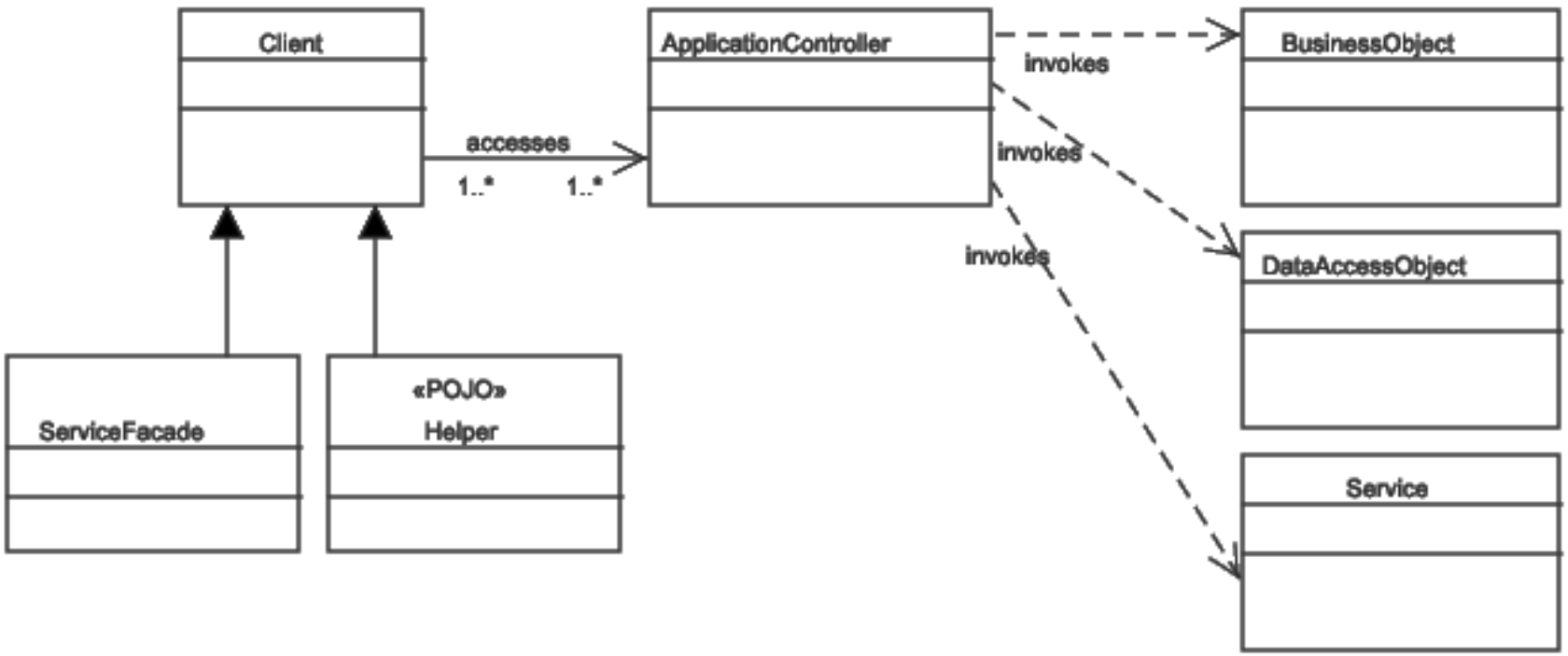
Ventajas:

Centralización. Encapsulación. Separa capas.

Desventajas:

Aporta complejidad. Puede que la clase central acabe siendo muy grande. Difícil de modificar cualquier cosa.

Diagrama de clases:



Transfer

Problema:
Enviar datos conjuntamente de una capa a otra.

Solución:
Objeto `Transfer` con `get` y `set` que encapsula la información y la envía.

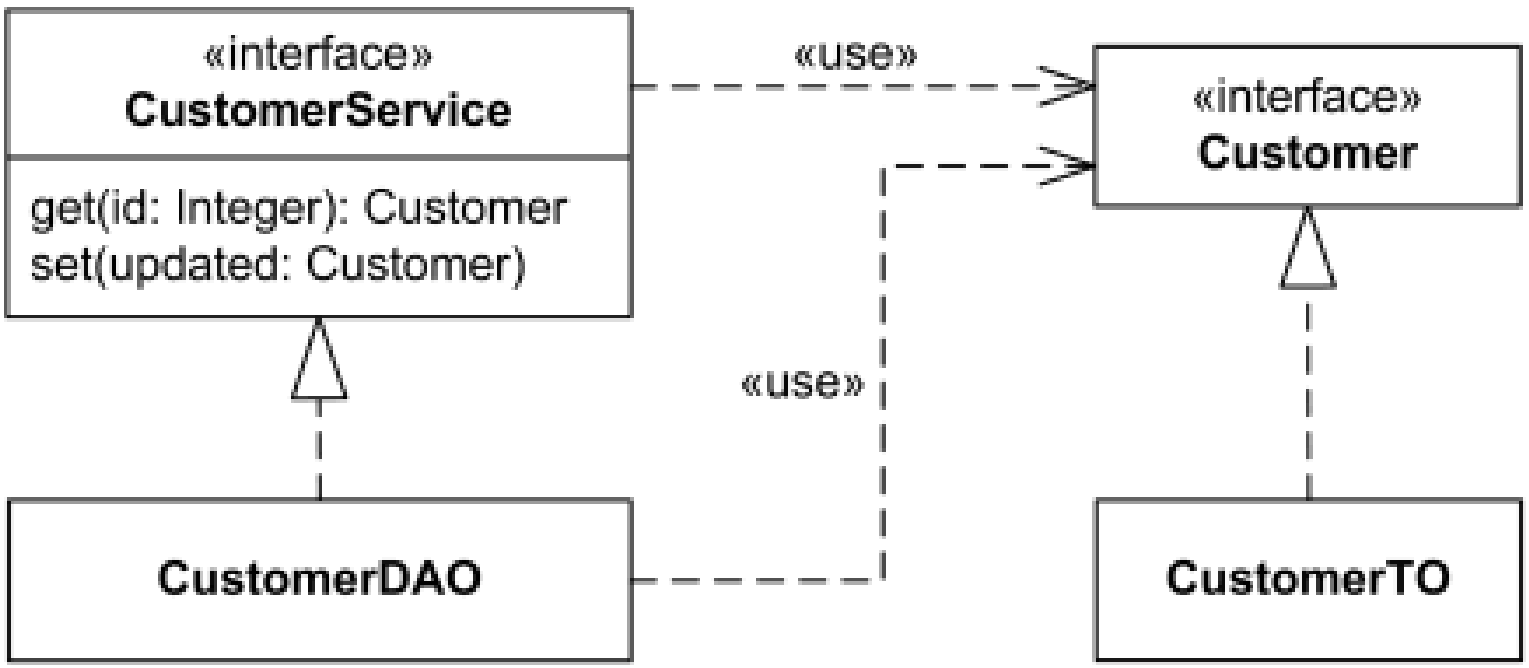
Ventajas:

Reduce el número de llamadas. Ayuda a la separación de capas. Reduce redundancia de código.

Desventajas:

Incrementa complejidad. A veces no se quiere toda la información a la vez pero está empaquetada.

Diagrama de clases:



Data Access Object

Problema:
Llamar igual a los métodos aun teniendo distintas bases de datos.

Solución:
Interfaz `DAO`, cada tipo de base de datos la implementa rellenando sus métodos según la variedad.

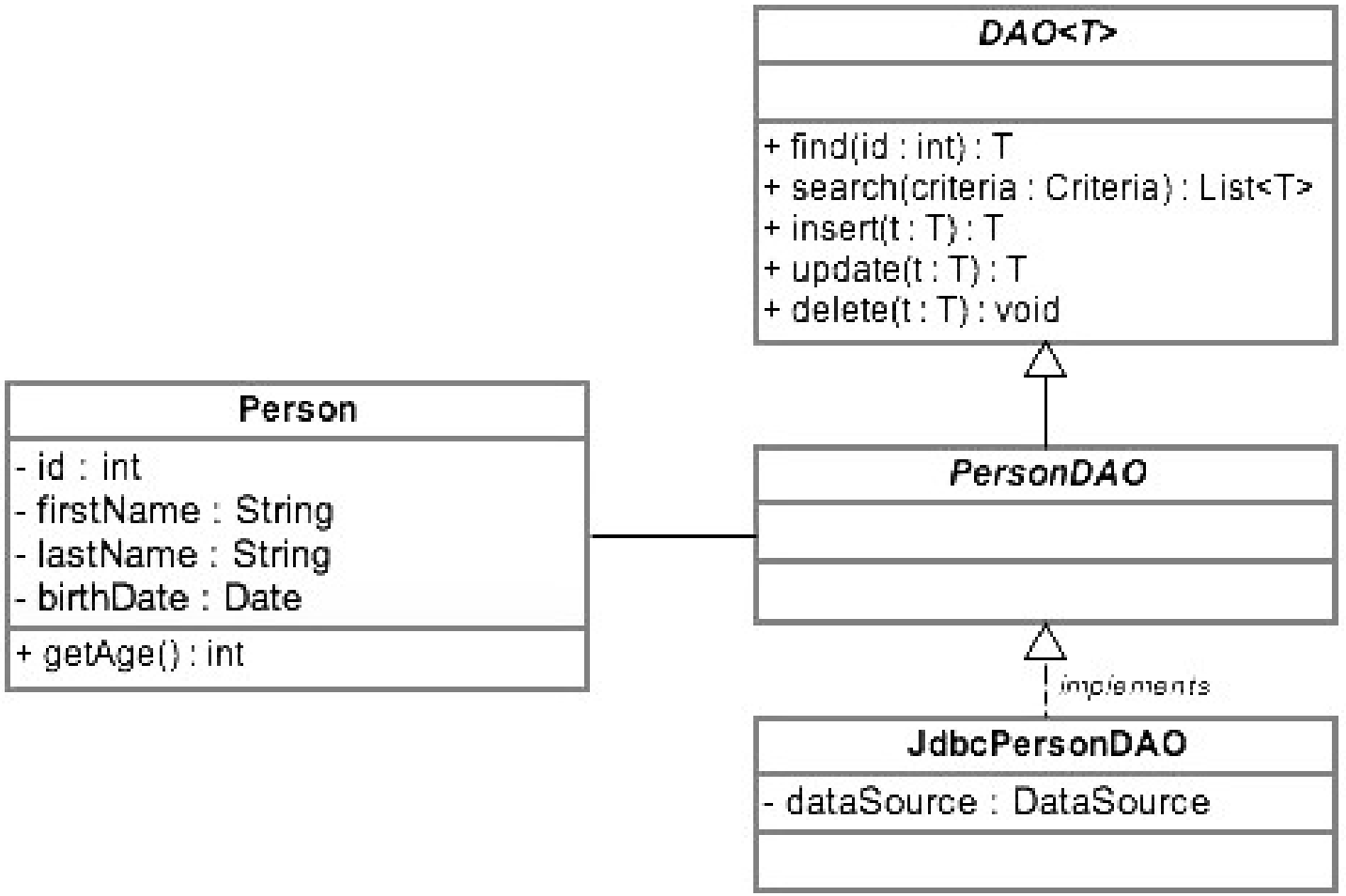
Ventajas:

Es fácil modificar código, añadirlo y cambiar de base de datos por completo. No afecta al resto del programa ni a Negocio.

Desventajas:

No se conoce la implementación interna, y normalmente se accede siempre a la base de datos entera, aumentando memoria y recursos.

Diagrama de clases:



Observador

Problema:
Tener clases que sufren cambios (`Observable`) y clases que necesitan conocer esos cambios a tiempo real (`Observers`).

Solución:
Que el Observable tenga una lista de Observers.
Notificarlos a todos en cada cambio con un `update()` de manera uniforme.

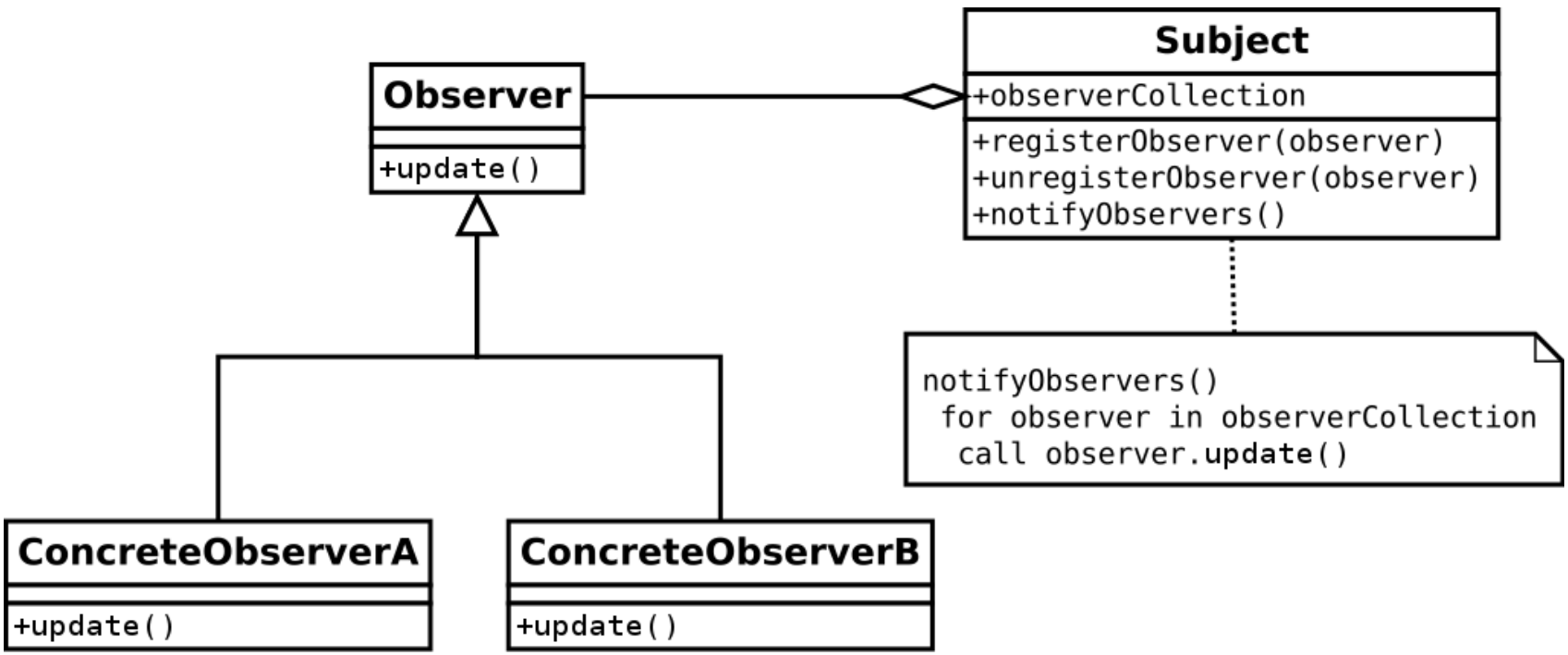
Ventajas:

Reutilizar código. Es fácil introducir nuevos Observers. Acoplamiento abstracto. Permite modificar clases independientemente.

Desventajas:

No distingue entre distintos tipos de Observers. El Observable no conoce a sus Observers. Si hay varios Observables se vuelve complicado.

Diagrama de clases:



Singleton

Problema:
Tener una clase de la que haya una sola instancia.

Solución:
Guardar una instancia estática. Constructora privada, y getInstance() estático público que devuelve la misma instancia siempre.

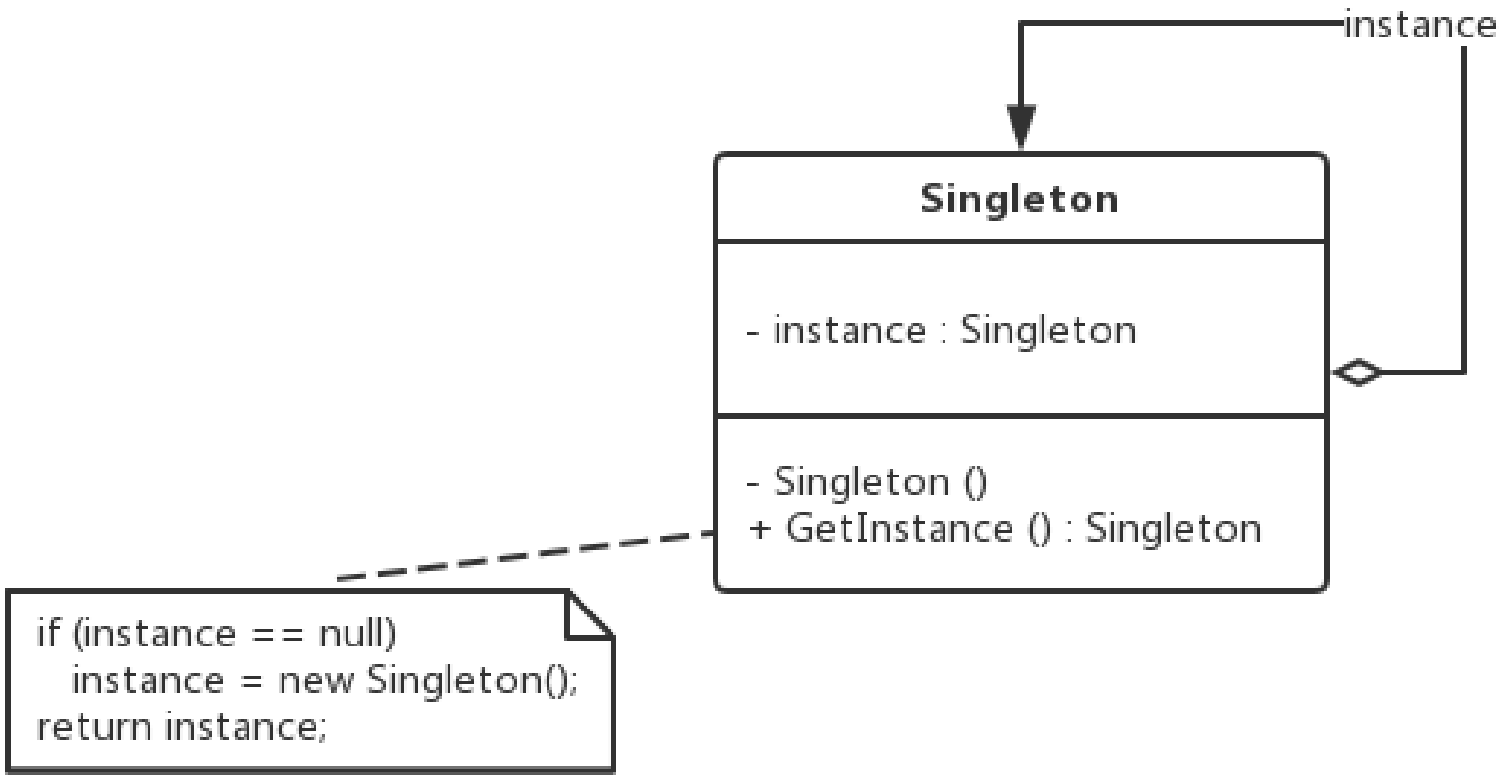
Ventajas:

Sencillo y arregla el problema.

Desventajas:

Puede haber problemas con varios hilos creando múltiples instancias a la vez. Son en realidad una variable global.

Diagrama de clases:



Fachada

Problema:
Interfaz unificada para un conjunto de interfaces.
Reducir el número de llamadas distintas.

Solución:
Una gran interfaz que unifique todas las interfaces.
Se llama solo a esa, como un mando a distancia.

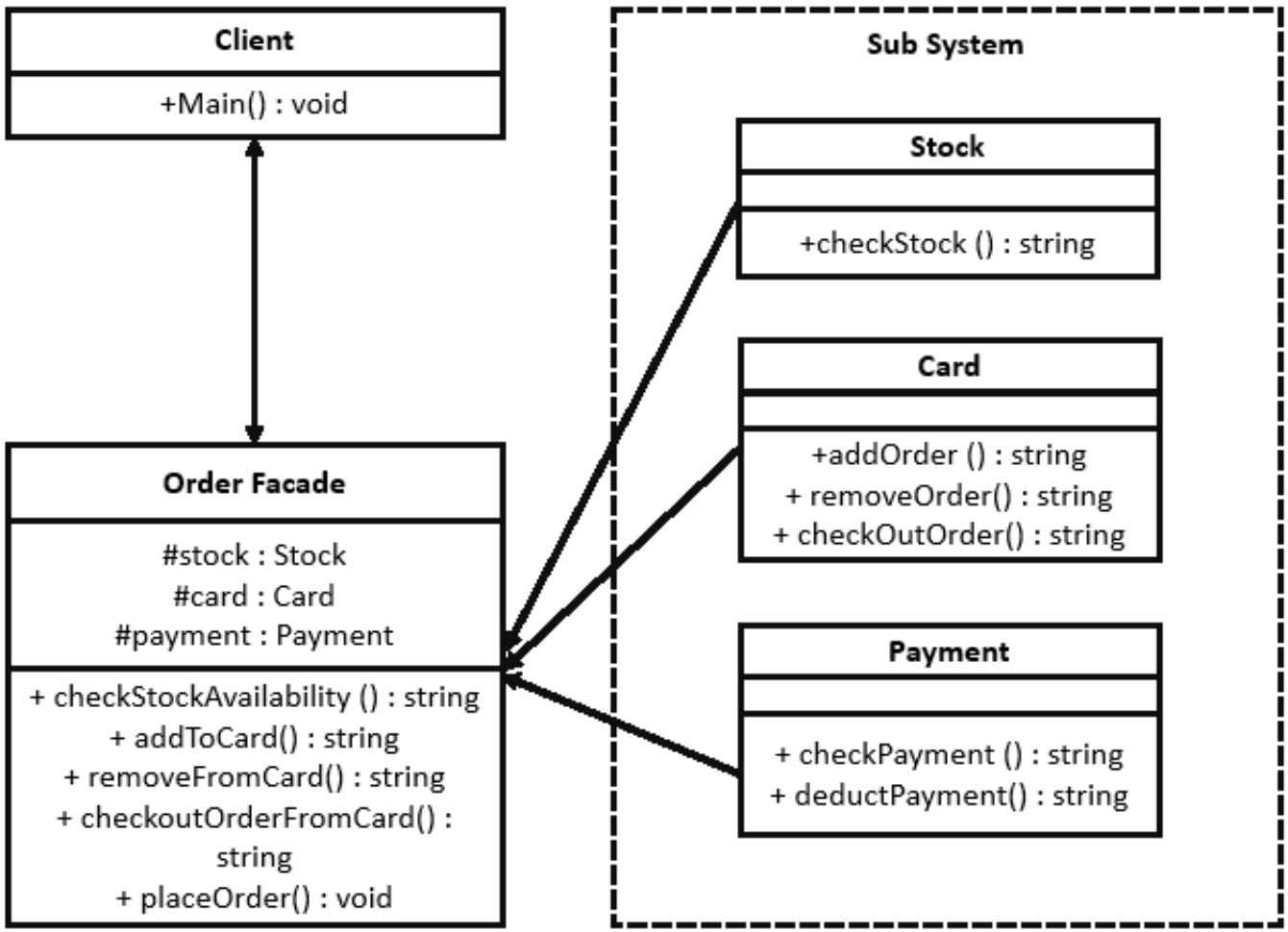
Ventajas:

La ocultación del núcleo al usuario, la facilidad de uso, la desacoplación, preserva la encapsulación mientras permite acceder a partes distintas a la vez.

Desventajas:

Cada cambio implicará un cambio de la fachada, y en programas pequeños es contraproducente su uso. Puede volverse enorme.

Diagrama de clases:



Factoría abstracta

Problema:
Tener varias clases `Producto` que hay que crear, y distintas formas de crearlas según varias clases `Factoría`.

Solución:
Hacer una Factoría abstracta de la que extiendan las factorías concretas. Hacer los Productos interfaces para que cumplan con sus funciones.

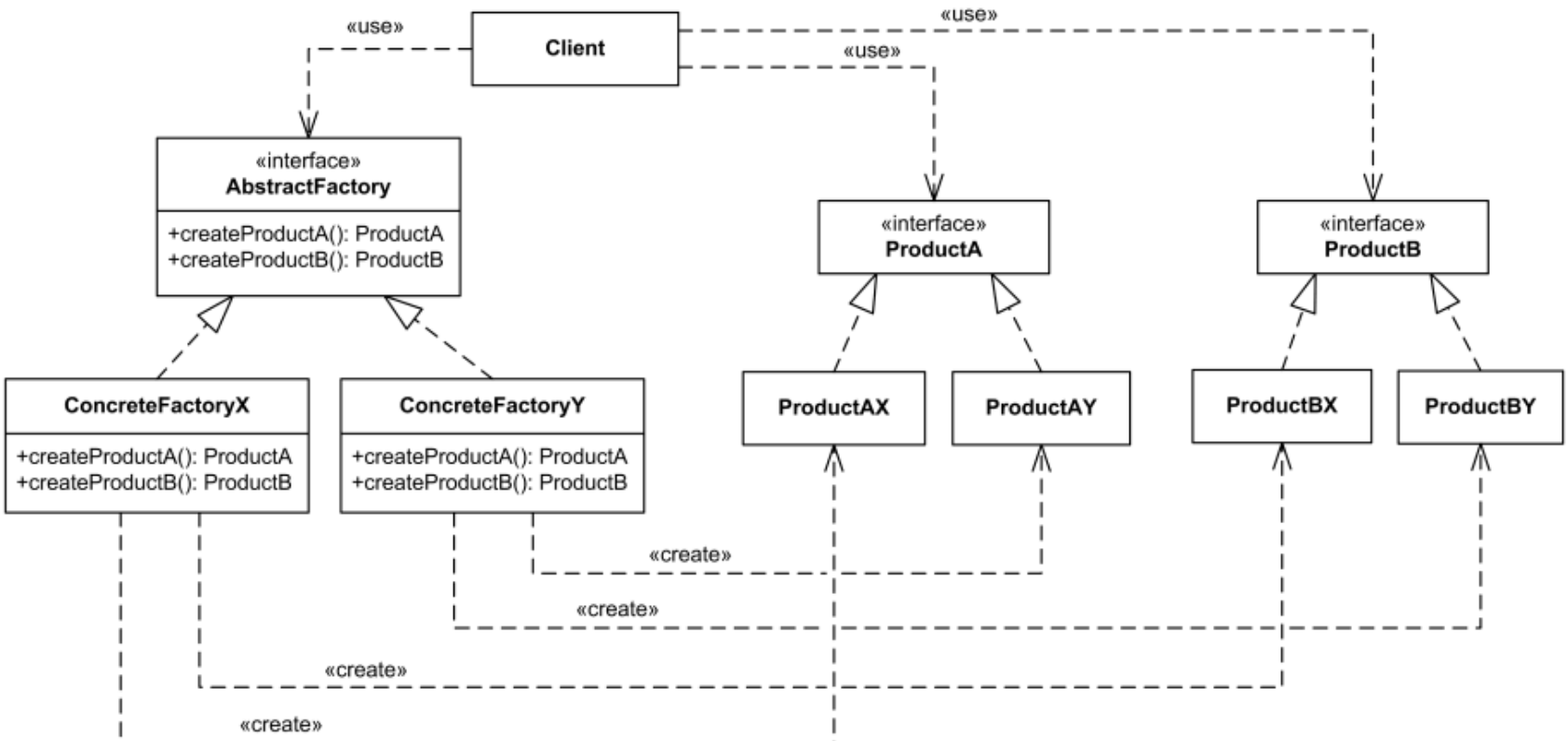
Ventajas:

Al ocultar la implementación, estas se mantienen más controladas y se evitan errores entre las partes de implementaciones distintas de una clase.

Desventajas:

Si se modifica la interfaz abstracta se tienen que modificar todas sus implementaciones. Para programas sencillos es demasiado complejo. Añade muchas clases y complicación.

Diagrama de clases:



Mediador

Problema:
Muchos elementos `Colegas` deben mandarse información entre sí. Se necesita una centralización que reciba toda la información y comunique el resultado en función de ello.

Solución:
La interfaz `Mediador` tiene métodos para cuando una clase `Colega` cambie. La clase que la implementa debe tener a todos los Colegas, y cada Colega debe tener un Mediador.

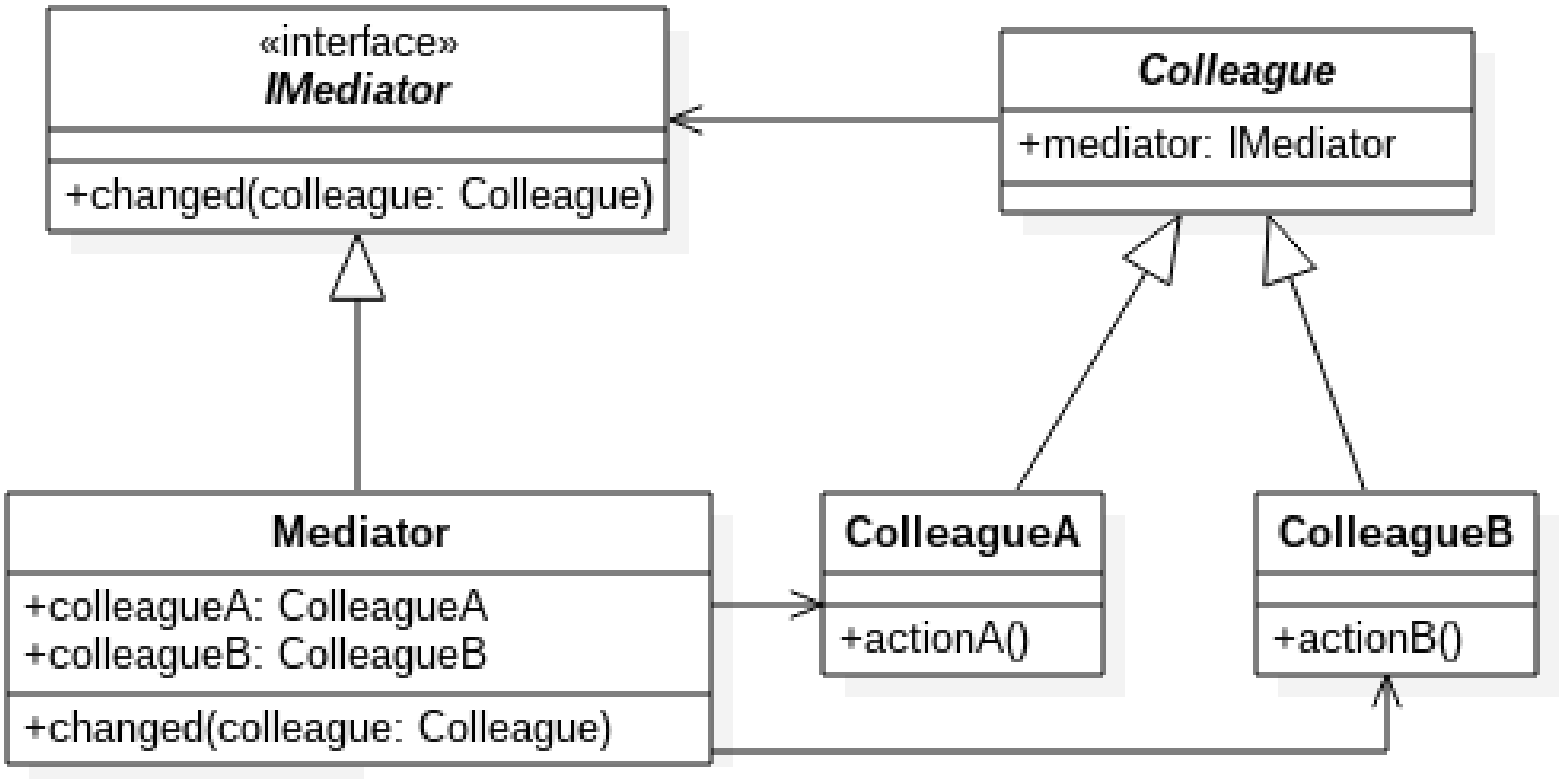
Ventajas:

Reduce la herencia. Desacoplamiento. Abstrae la cooperación. Simplifica la comunicación.

Desventajas:

La clase Mediador acaba siendo muy compleja y amplia. Centraliza el control.

Diagrama de clases:



Comando

Problema:
Mandar tareas sin saber quién las hace.

Solución:
Interfaz `Comando` con el método `execute()`. Todas las órdenes se podrán llamar de esta forma.

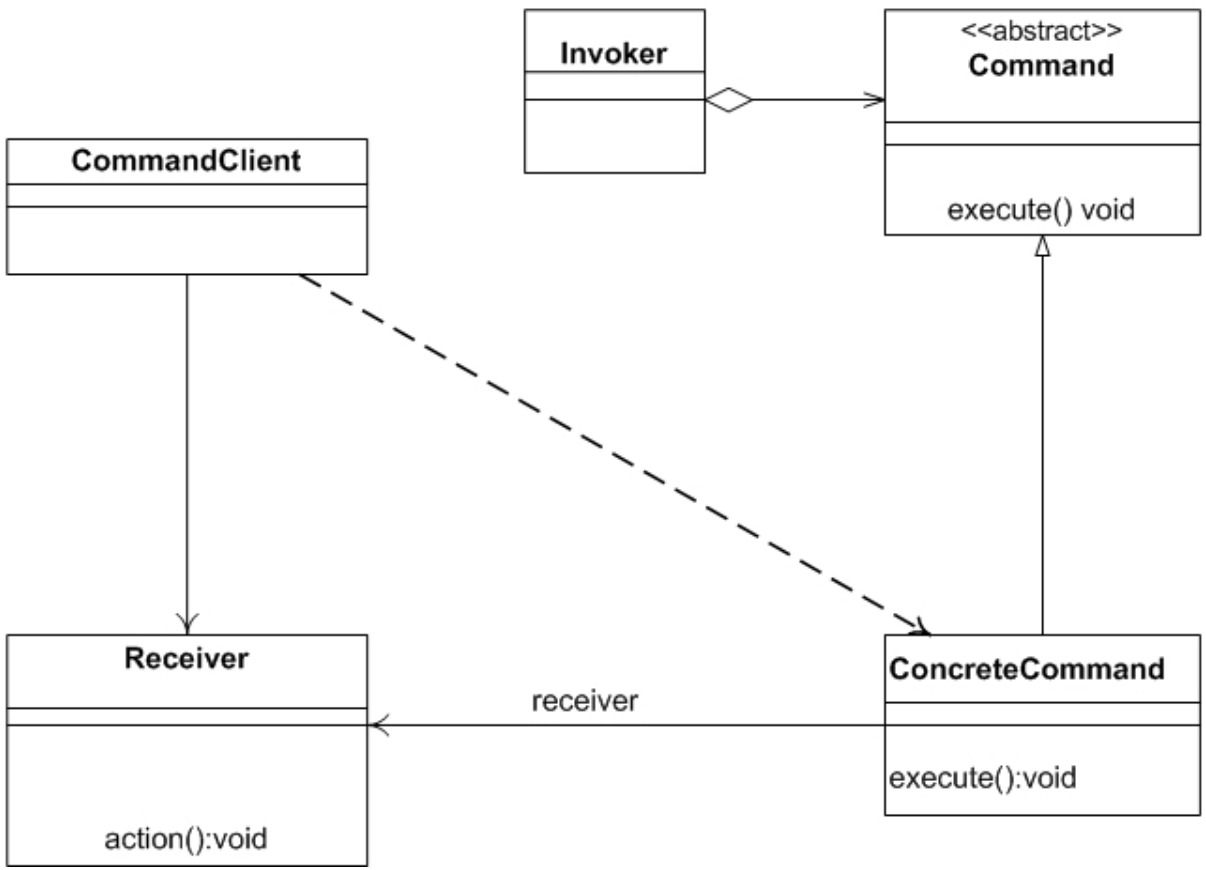
Ventajas:

Es fácil añadir órdenes nuevas. Encapsulación. Simular que todo es un objeto. Hacer y deshacer. Cada clase hace solo una cosa.

Desventajas:

Añade muchas clases y complicación para una sola acción.

Diagrama de clases:



Composite

Problema:
Estructura de árbol para clases. Si llamas a una llamas a sus hijos. Estructura contenedor – contenido.

Solución:
Clases abstractas `Component`, que tienen una `operation()`. De ella extienden algunas clases hoja, y otras clases nodo, o `Composite`. Éstas tienen un array de Componentes, y su `operation()` consiste en llamar al de sus hijos.

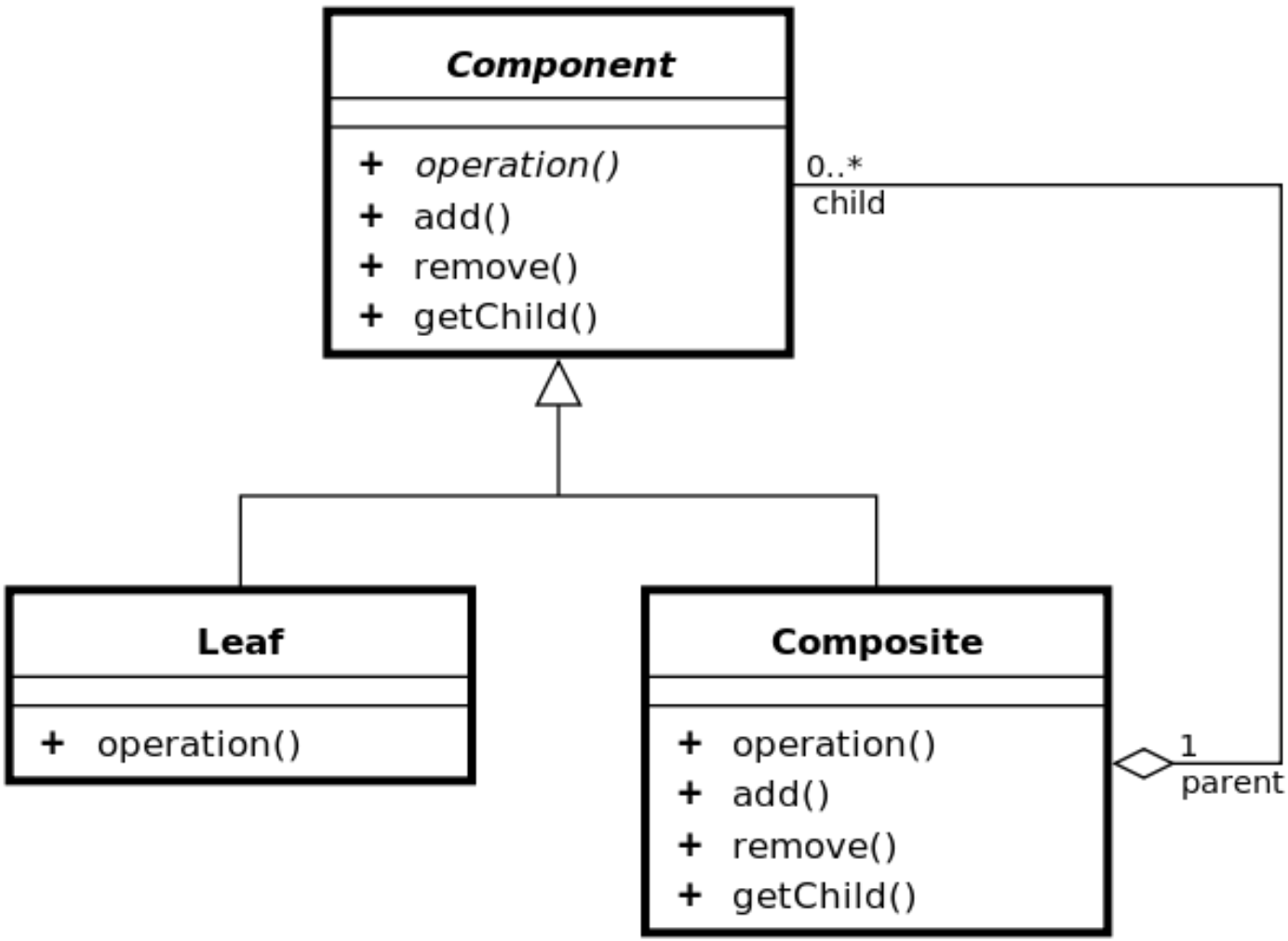
Ventajas:

No te preocupas del interior del objeto. Se trata a todos igual. Añades fácilmente objetos.

Desventajas:

No distingue entre significados. Un coche puede estar dentro de sí mismo. Es difícil que cuadre la estructura.

Diagrama de clases:



Decorador

Problema:
Añadir capas a un componente. Añadir métodos y atributos de forma dinámica. Al querer añadir muchos elementos a una componente, la herencia no nos vale, tampoco tener un array con los componentes extras.

Solución:
Crear interfaz `Componente`. Crear la clase abstracta `Decorador` que la implementa, y además tiene una instancia de ella, pasada por constructora. En los métodos heredados se llama a los de su atributo. Una vez que se tiene un Concrete-Component, se crea con él el ConcreteDecorator.

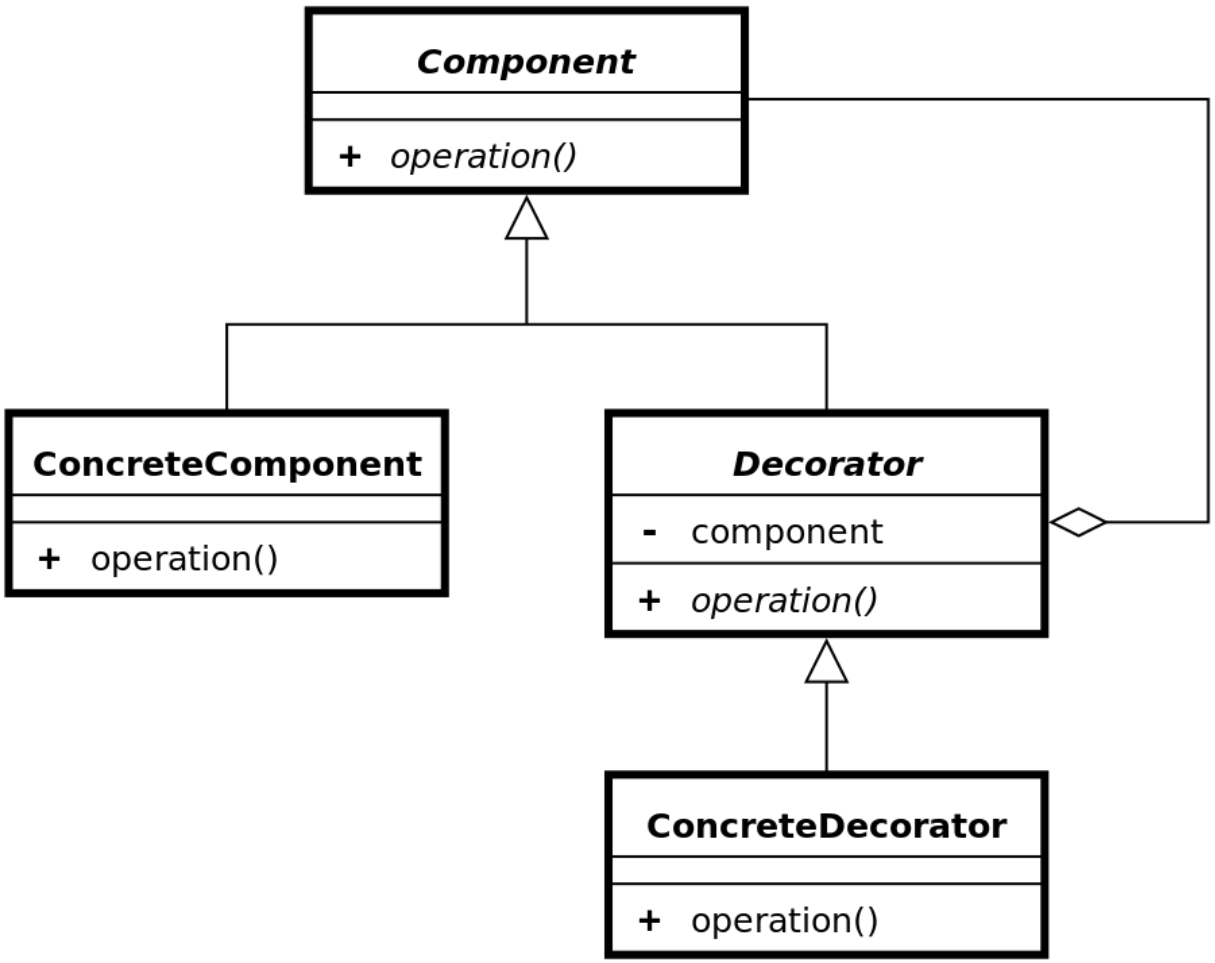
Ventajas:

Reutilización de código. Añadir funcionalidad dinámicamente. Cada clase realiza pocas funciones cada una.

Desventajas:

Es añadir por añadir. Es difícil quitar decoraciones una vez puestas. Dependen del orden en el que se añaden. Puede que tu objeto inicial sea complicado.

Diagrama de clases:



Adapter

Problema:
Reutilizar una interfaz ya hecha, cambiándole el nombre a los métodos. Reutilizar código de otros.

Solución:
Definir una clase `Adapter` que convierte la clase incompatible `Adaptee` a lo que quiere el cliente, la interfaz `Target`. La clase Adaptor tiene una instancia de Adaptee para llamar a sus métodos.

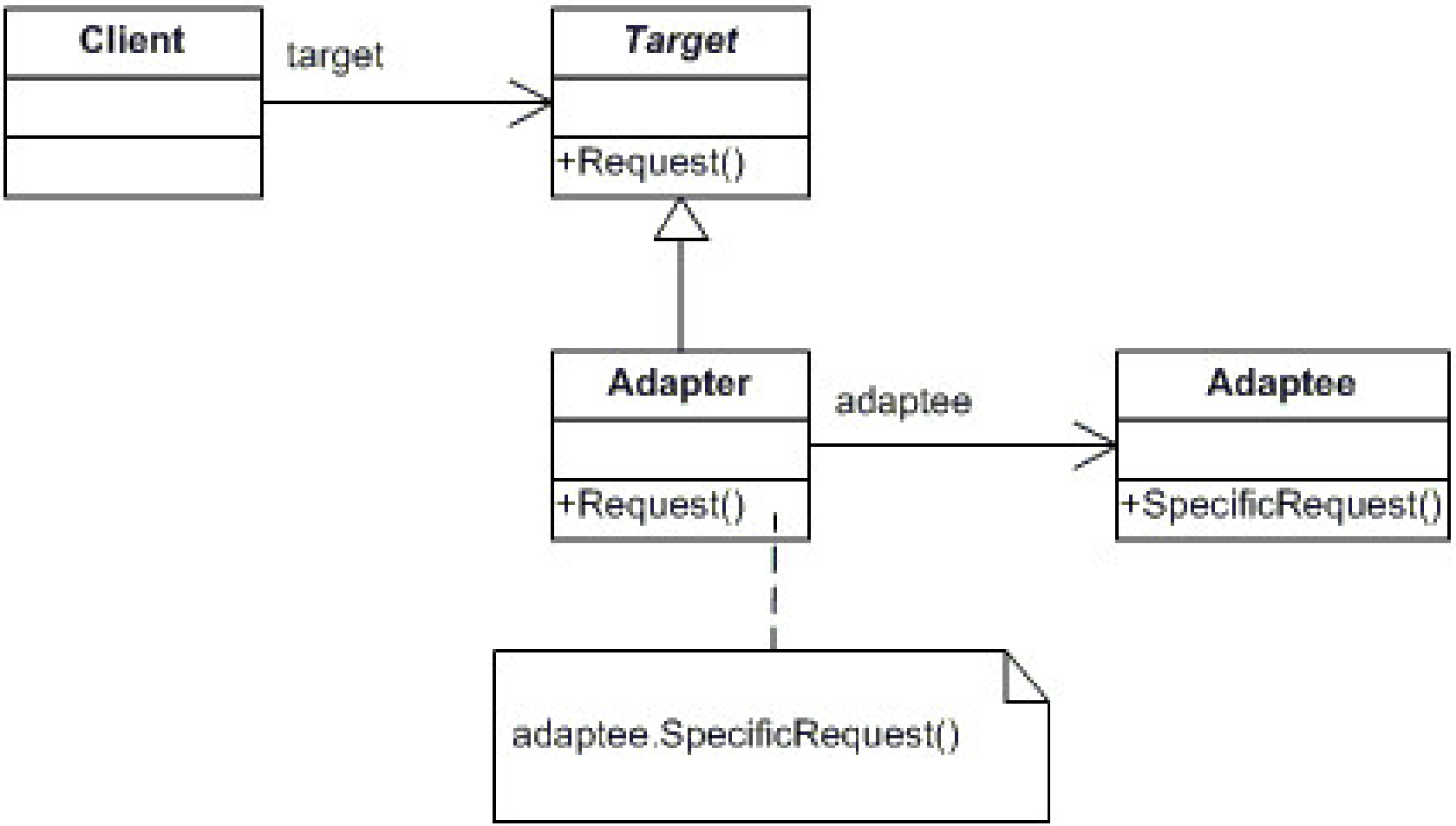
Ventajas:

Reutilización de código. Separa el algoritmo de su llamada. Se pueden añadir fácilmente otros adaptadores.

Desventajas:

Súper complejo, añades muchas clases extra. A veces conviene reescribir el código.

Diagrama de clases:



Iterador

Problema:
Recorrer cualquier tipo de estructura contenedor de la misma forma y acceder a sus elementos.

Solución:
Crear clase abstracta `Iterable`, o Aggregate, que cree su propio iterador para ser recorrida. A su vez la interfaz `Iterador` debe tener métodos típicos de recorrer (siguiente, último...).

Ventajas:

No expone la implementación interna. Se pueden recorrer de distintas maneras si se implementan distintos iteradores. Separa algoritmo de estructura. Se añaden nuevas cosas fácilmente.

Desventajas:

A veces demasiada complejidad para lo que sirve. A lo mejor menos eficiente.

Diagrama de clases: