Ejercicios sobre patrones GOF

Diseño de Sistemas Software
Curso 2021/2022

Carlos Pérez Sancho

Universitat d'Alacant
Universidad de Alicante

Departament de Llenguatges i Sistemes Informàtics
Departamento de Lenguajes y Sistemas Informáticos

Ejercicio 1

Enunciado

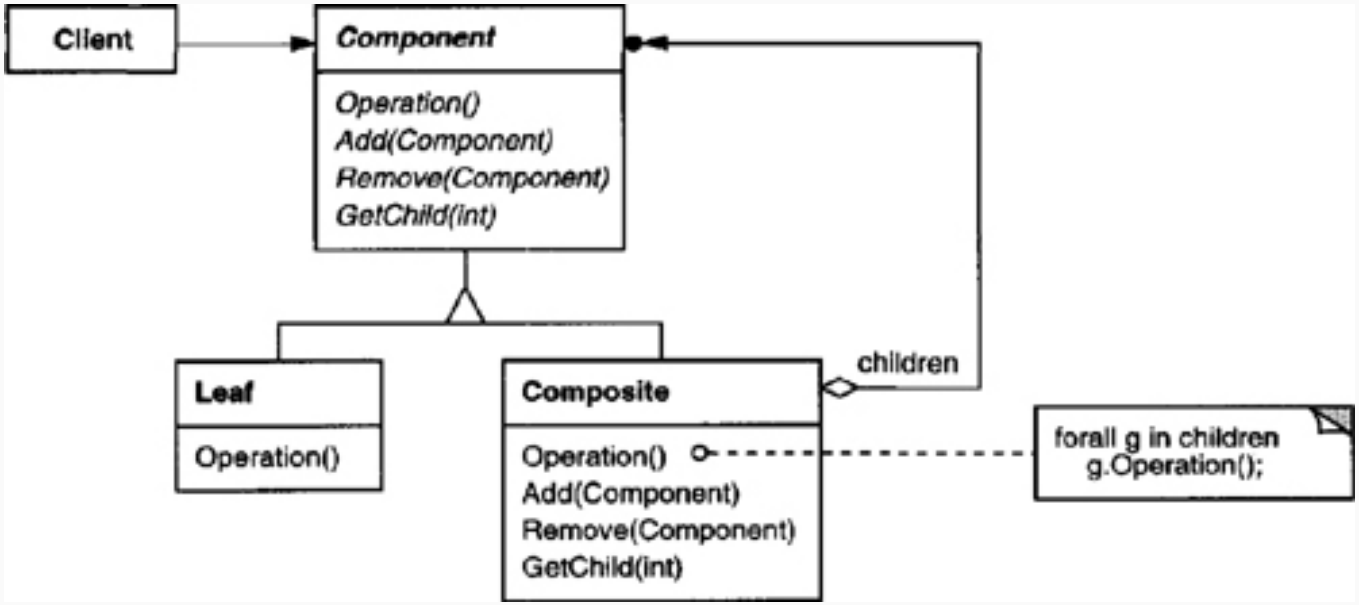
Decathlon nos ha pedido que le creemos un módulo para el manejo de la configuración de bicicletas a medida. Para ello, la tienda nos ha proporcionado la siguiente lista de partes, que podría irse especializando con nuevas partes, simples o agregadas: rueda, horquilla, eje, radio, tuercas del radio de las ruedas, cámara, tubo, llanta, armazón, manillar, ..., cada una con su precio. En un principio supondremos que Decathlon trabaja con un solo modelo de cada parte. Decathlon quiere que su software pueda llamar a nuestro módulo sin tener que preocuparse de si está tratando con una bicicleta completa o con partes aisladas.

De momento nos ha pedido un prototipo donde la única funcionalidad necesaria es `getPrecio()`. Ese precio se compone de un precio de mano de obra (sólo aplicable si el cliente pide una pieza que tiene que ser montada) y un precio base por cada una de las piezas. Plantea un diseño que resuelva este problema.

Análisis del problema

La complejidad del problema consiste en representar dos tipos de piezas (**simples y compuestas**), que deben comportarse de igual manera ante el cliente (**comparten un interfaz común**).

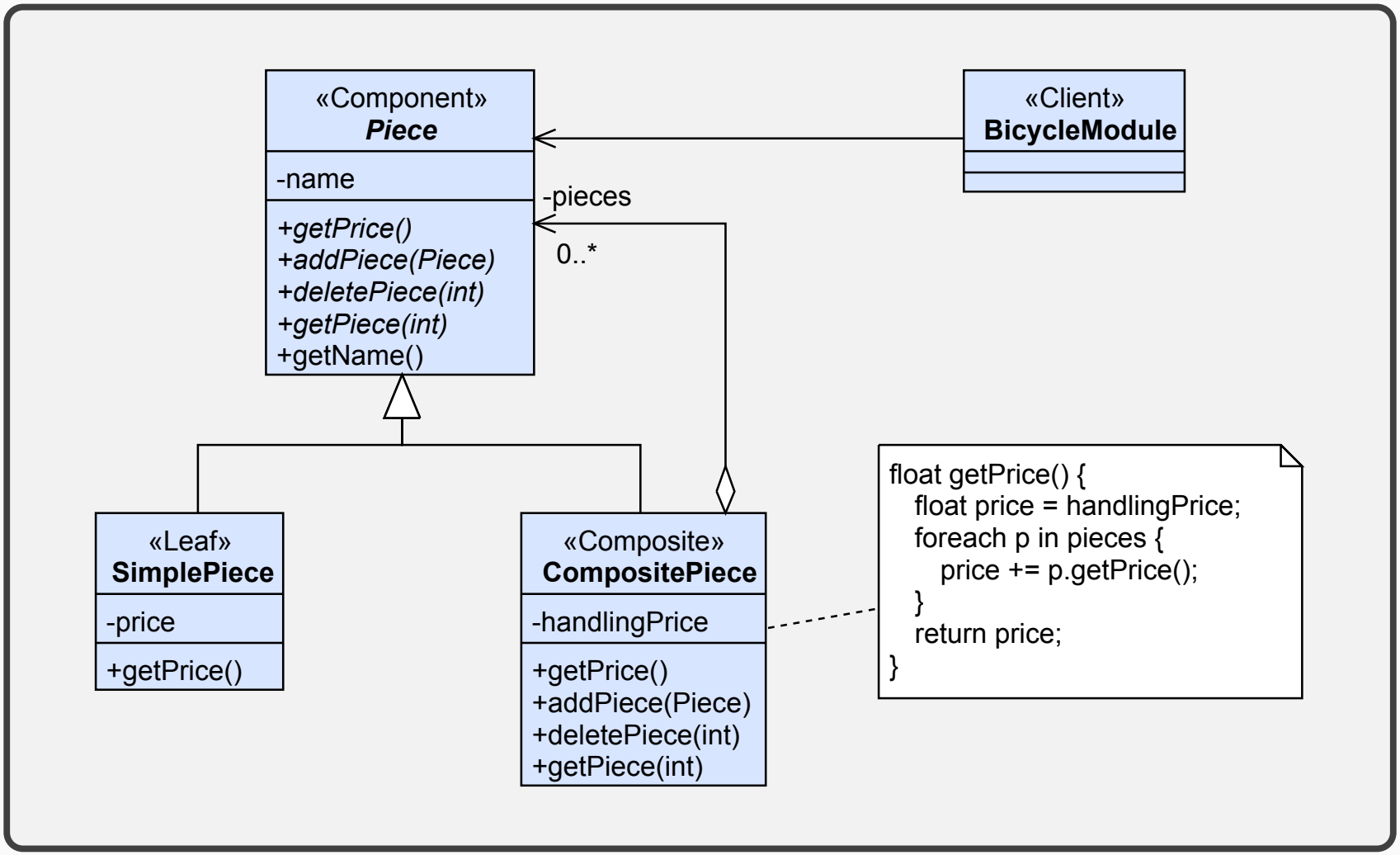
Solución: Patrón Composite (estructural)



[Gamma et al., 1994]

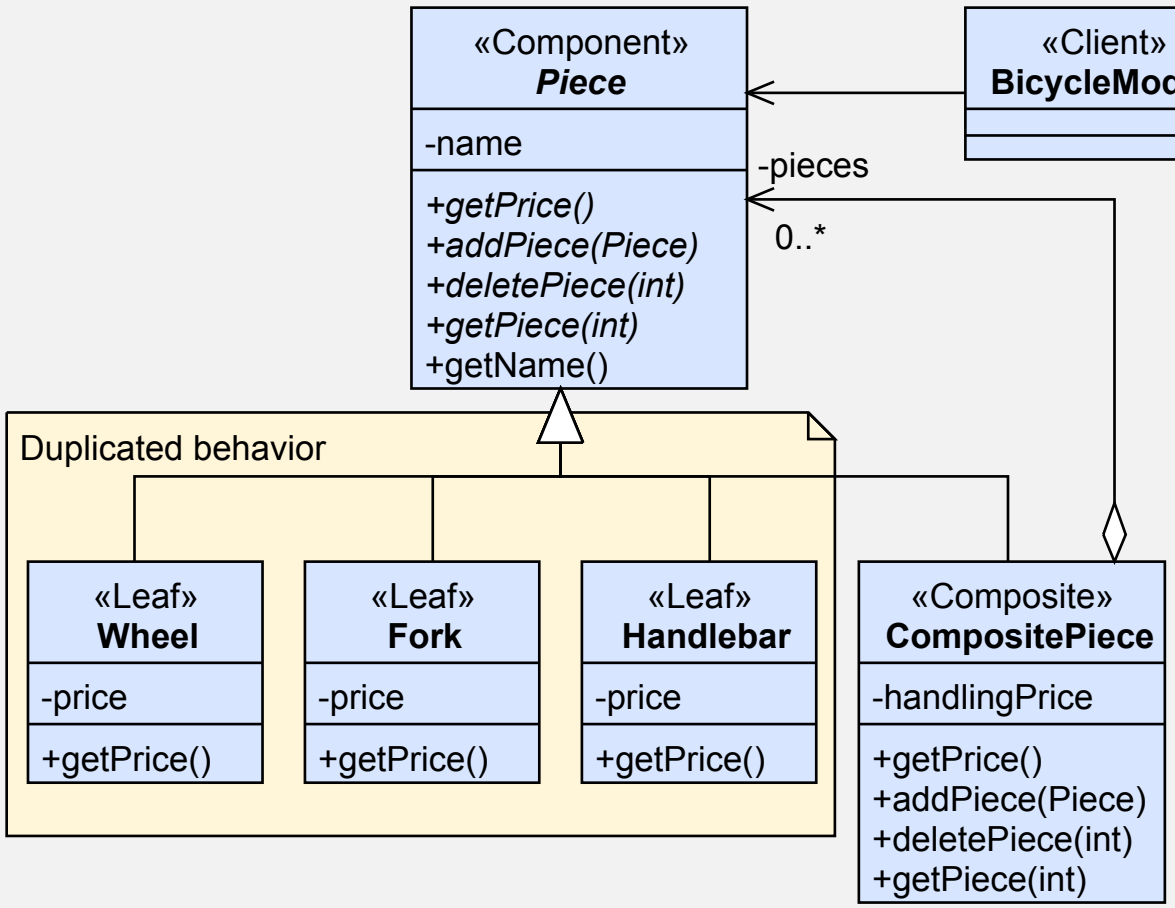
Solución

Patrón Composite



Discusión

Si todas las piezas simples tienen el mismo comportamiento debería evitarse un diseño como éste:



Discusión

Ventajas:

- El cliente ve todos los objetos como si pertenecieran a la clase `Piece` y los trata de manera uniforme. Se simplifica el código del cliente.
- Es sencillo añadir nuevos componentes sin tener que modificar el cliente.

Inconvenientes:

- Las restricciones en cuanto a la composición de las piezas compuestas se debe hacer mediante código (en tiempo de ejecución).

Discusión

Detalles de implementación: hay dos alternativas para ubicar los métodos que gestionan los componentes de una pieza compuesta.

- Para que el cliente trate a todos los componentes por igual, deben compartir el mayor número posible de métodos en el interfaz común. De esta manera las piezas simples se comportarían como piezas compuestas que nunca pueden tener componentes $\rightarrow$ **mayor transparencia**.
- Sin embargo, según el principio de segregación de interfaces, las piezas simples no deberían tener métodos que no necesitan. Los métodos para gestionar los componentes estarían sólo en las piezas compuestas, de manera que el cliente no puede usar por error las piezas simples como si fueran compuestas $\rightarrow$ **mayor seguridad**.

Ejercicio 2

Enunciado

En un sistema que ofrece servicios a clientes remotos se desea realizar una auditoría del tiempo que tardan los distintos servicios en ejecutarse, a fin de mejorar el rendimiento global del sistema. Para esto se ha implementado una clase `UtilTime`, que no es abstracta, que incluye un método estático `time()` que mide el tiempo que tarda un método cualquiera en ejecutarse. Parte del código de dicho método sería:

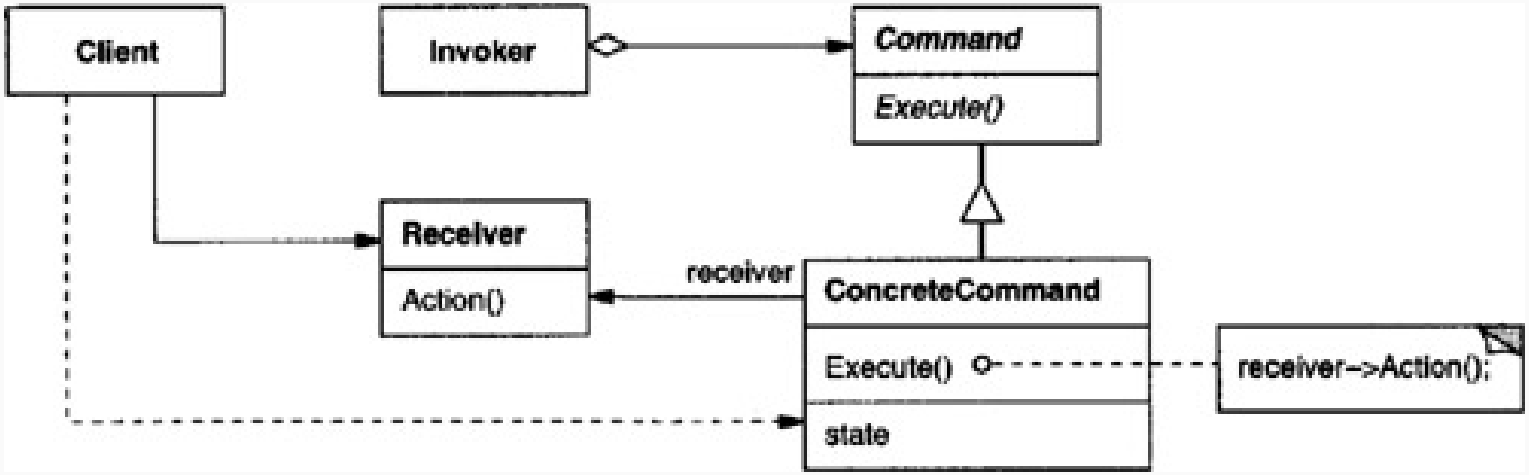
```
public static long time (/* parámetros */) {
    long t1 = System.currentTimeMillis();
    // falta aquí el código apropiado
    long t2 = System.currentTimeMillis();
    return t2 - t1;
}
```

Diseña una solución basada en alguno de los patrones GOF y completa el método: parámetros y código en la posición del comentario. Escribe un código cliente con un ejemplo de utilización del método `time()`.

Análisis del problema

Para solucionar el problema necesitamos **delegar la ejecución** de los servicios, de manera que ahora será la clase `UtilTime` la encargada de ejecutarlos. Para esto es necesario **convertir los servicios (operaciones) en objetos**, manteniendo su inicialización donde se hacía originalmente, y añadiéndoles un método que permita ejecutarlos.

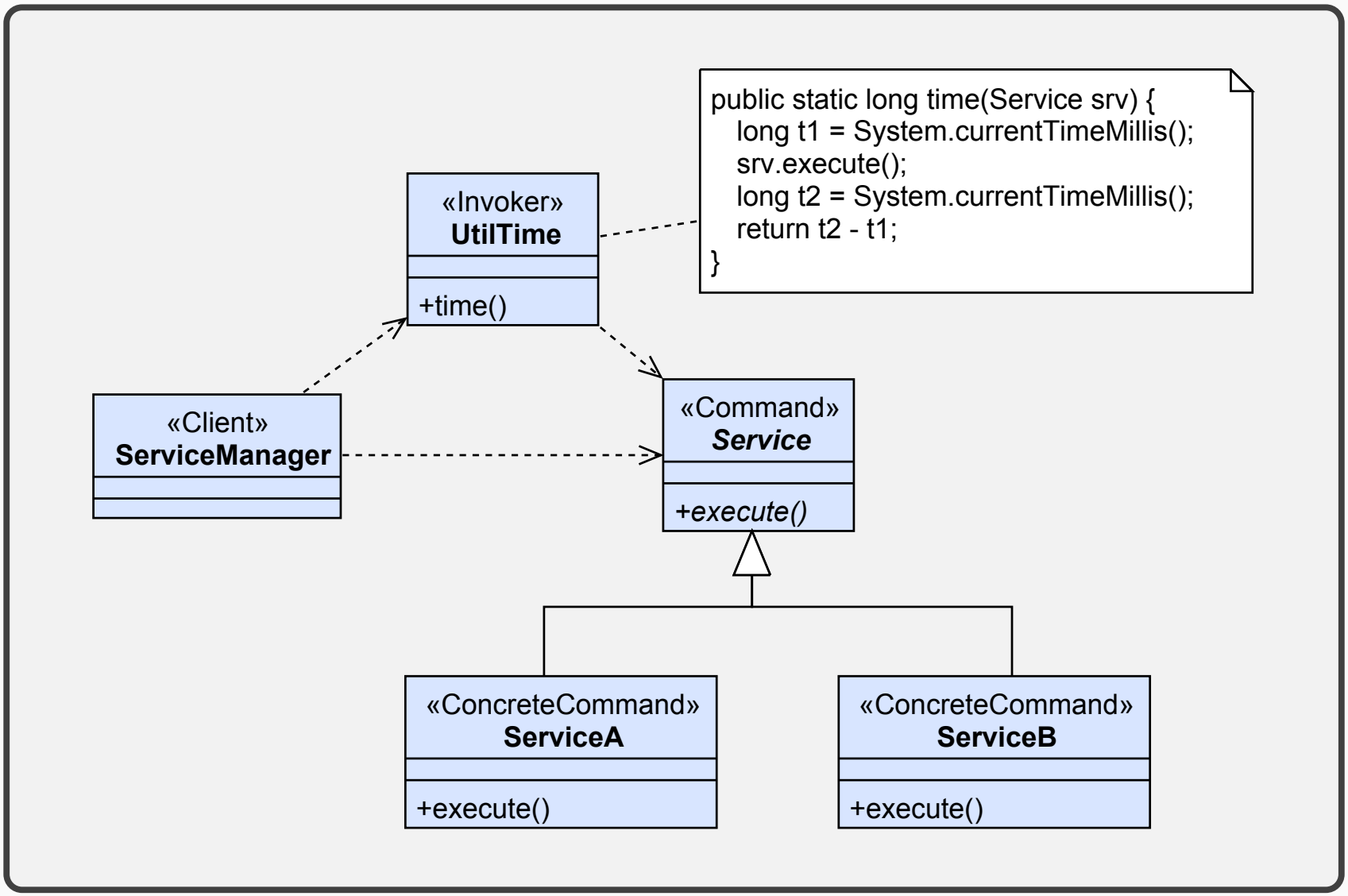
Solución: Patrón Command (de comportamiento)



[Gamma et al., 1994]

Solución

Patrón Command



Solución

```
class ServiceManager {
    public void auditService() {
        Service service = new ServiceA();
        HashMap params = new HashMap();
        params.add("param1", "value1"); // See discussion
        long time = UtilTime.time(service, params);
        // do some stuff
    }
}

class UtilTime {
    public static long time(Service srv, HashMap params) {
        long t1 = System.currentTimeMillis();
        srv.execute(params);
        long t2 = System.currentTimeMillis();
        return t2 - t1;
    }
}
```

Discusión

Detalles de implementación: al compartir todos los servicios un interfaz común se debe proporcionar un mecanismo para poder pasarles la información necesaria para ejecutarse, ya que pueden necesitar un número distinto de parámetros, o parámetros de distinto tipo.

Una implementación alternativa al uso de mapas sería:

```
class ServiceManager {
    public void auditService() {
        Service service = new ServiceA("value1");
        /* or */
        Service service = new ServiceA();
        service.setParam("param1", "value1");

        long time = UtilTime.time(service);
        // do some stuff
    }
}
```

Discusión

Ventajas:

- Al convertir operaciones en objetos podemos manipularlos con mayor flexibilidad, p.ej. delegando su ejecución o encolarlos para ejecutarlos en orden cuando los recursos son limitados.
- Permite implementar la funcionalidad ”Deshacer”, si cada comando implementa una operación `unexecute()` y se guarda un histórico de los comandos ejecutados.
- Se pueden definir comandos compuestos (macros) combinando este patrón con el patrón Composite.
- Facilita llevar un registro de las operaciones ejecutadas, añadiendo a cada comando la posibilidad de almacenar información sobre su ejecución.
- Es sencillo añadir nuevos comandos.

Discusión

Inconvenientes:

- Cuando un comando actúa sobre otro objeto (*Receiver*), es difícil decidir qué responsabilidades corresponden a cada uno de ellos. El comando puede actuar como un simple mensaje para indicar al receptor que debe realizar alguna operación, o puede implementar toda la funcionalidad él mismo.
- Para asegurar la consistencia de la aplicación cuando se ofrece la posibilidad de deshacer y rehacer operaciones, se debe almacenar información precisa que permita devolver la aplicación a su estado anterior. Esto puede aumentar significativamente la complejidad.

Ejercicio 3

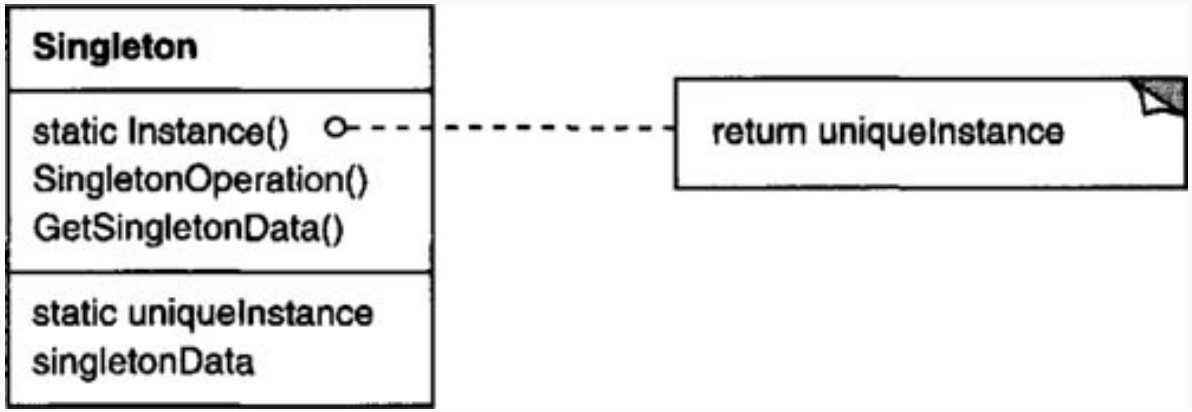
Enunciado

Nos han pedido introducir en la aplicación que estamos diseñando una clase de login que proporcione un punto de acceso global a la operación de login y password para todos los componentes de la aplicación. Propón una solución utilizando un patrón GOF.

Análisis del problema

Para proporcionar un punto de acceso global debe haber **una única instancia de la clase**, accesible mediante un método estático. De esta manera todos los componentes controlan el acceso a través de la misma instancia, evitando inconsistencias.

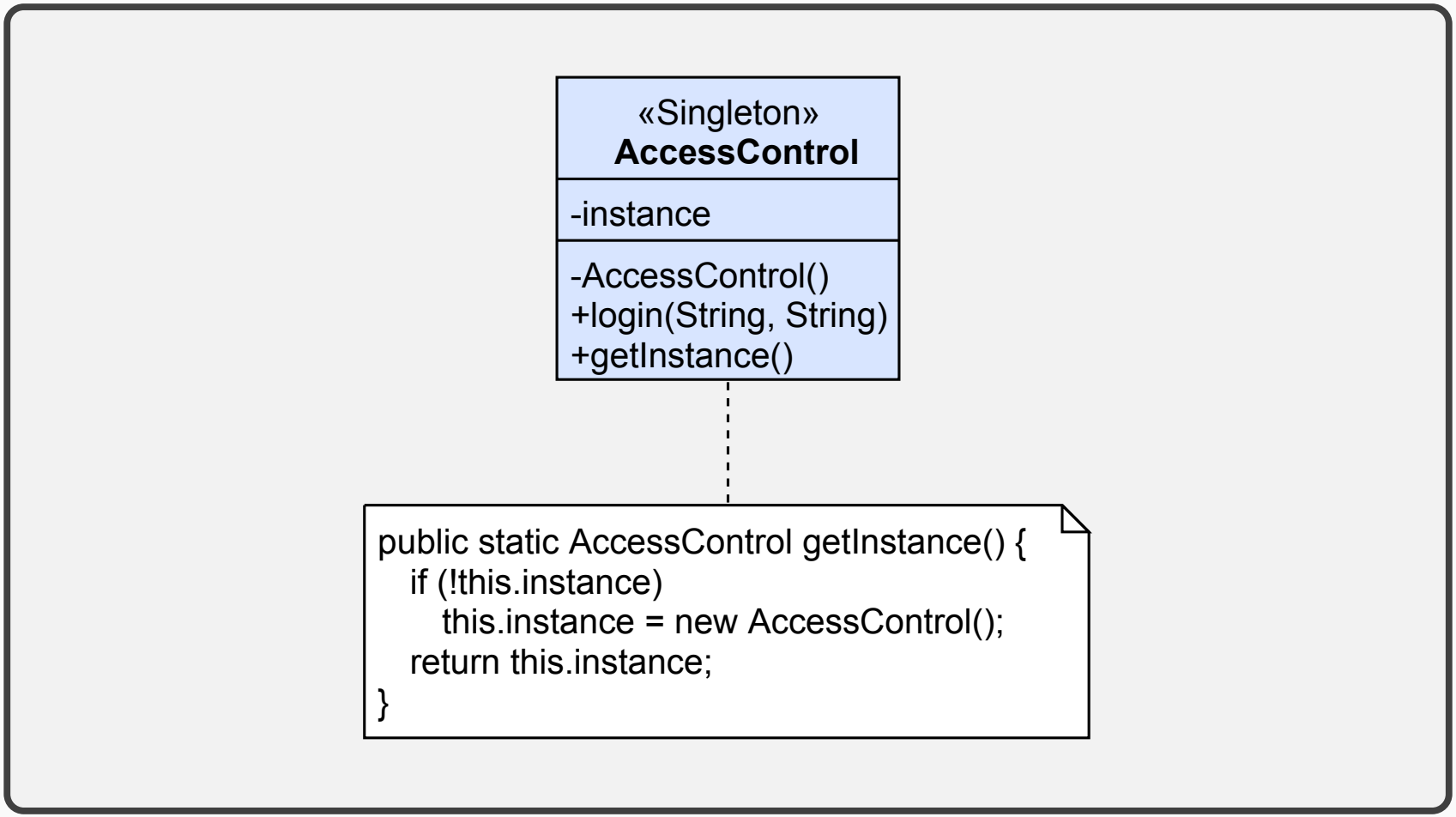
Solución: Patrón Singleton (creacional)



[Gamma et al., 1994]

Solución

Patrón Singleton



Discusión

Ventajas:

- Permite controlar el acceso a la única instancia.
- Se puede modificar fácilmente para permitir un número limitado de instancias reutilizables (*pooling*).
- Más flexible que usar un método estático para implementar la funcionalidad.

Inconvenientes:

- Si se quieren crear subclases de la clase Singleton la solución no es trivial, ya que todas comparten el atributo que almacena la única instancia.

Ejercicio 4

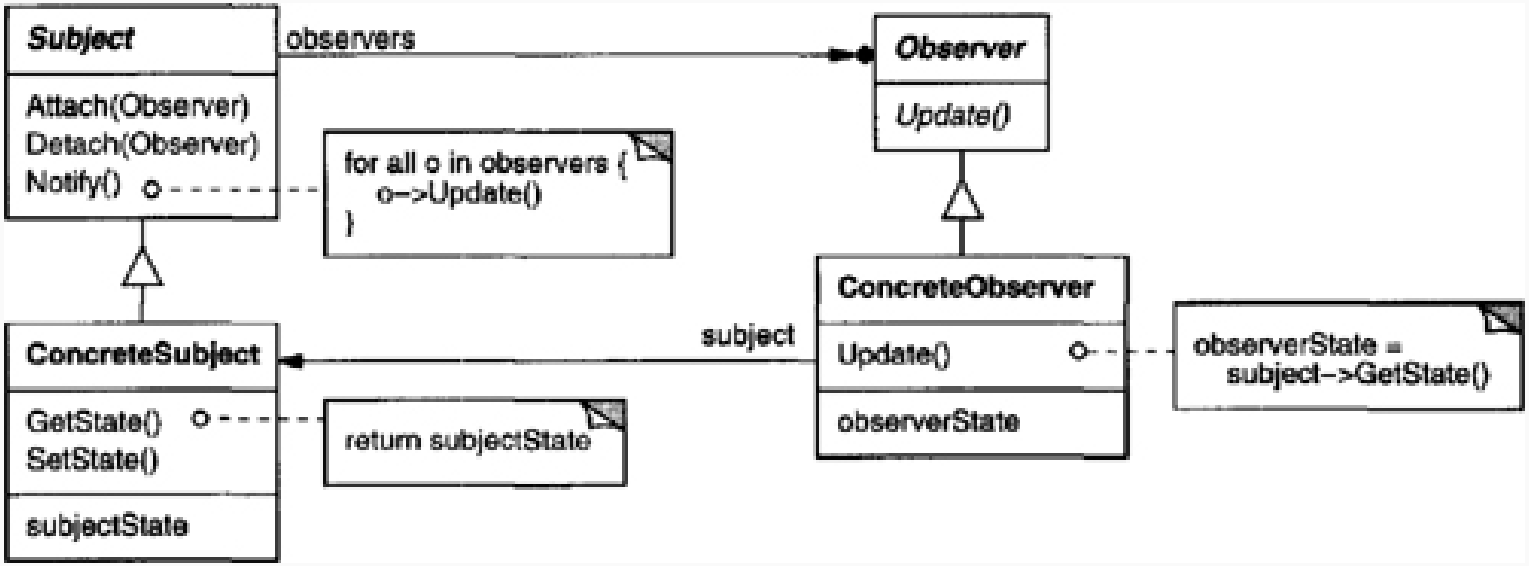
Enunciado

Nos ha contactado la Agencia F de noticias. Esta agencia recoge noticias y comunicados de todo el mundo, y las distribuye a todos sus clientes (periódicos, cadenas de televisión, etc.). Nos han pedido que creemos un framework para que la agencia pueda informar inmediatamente, cuando ocurre cualquier evento, a cualquiera de sus clientes. Estos clientes pueden recibir las noticias de distintas maneras: Email, SMS, etc. Además, nos han pedido que la solución sea lo suficientemente extensible para soportar nuevos tipos de subscripciones (p.ej. recientemente ya hay algún cliente que les ha pedido ser notificado a través de Twitter).

Análisis del problema

Para este problema necesitamos crear objetos que representan las suscripciones e informen a los clientes cuando haya alguna novedad. Sin embargo, no es viable hacer que estos objetos vigilen activamente el objeto que contiene las noticias, ya que afectaría negativamente al rendimiento. En lugar de esto los objetos “subscripción” **se registrarán para que se les notifique cuando haya alguna novedad**.

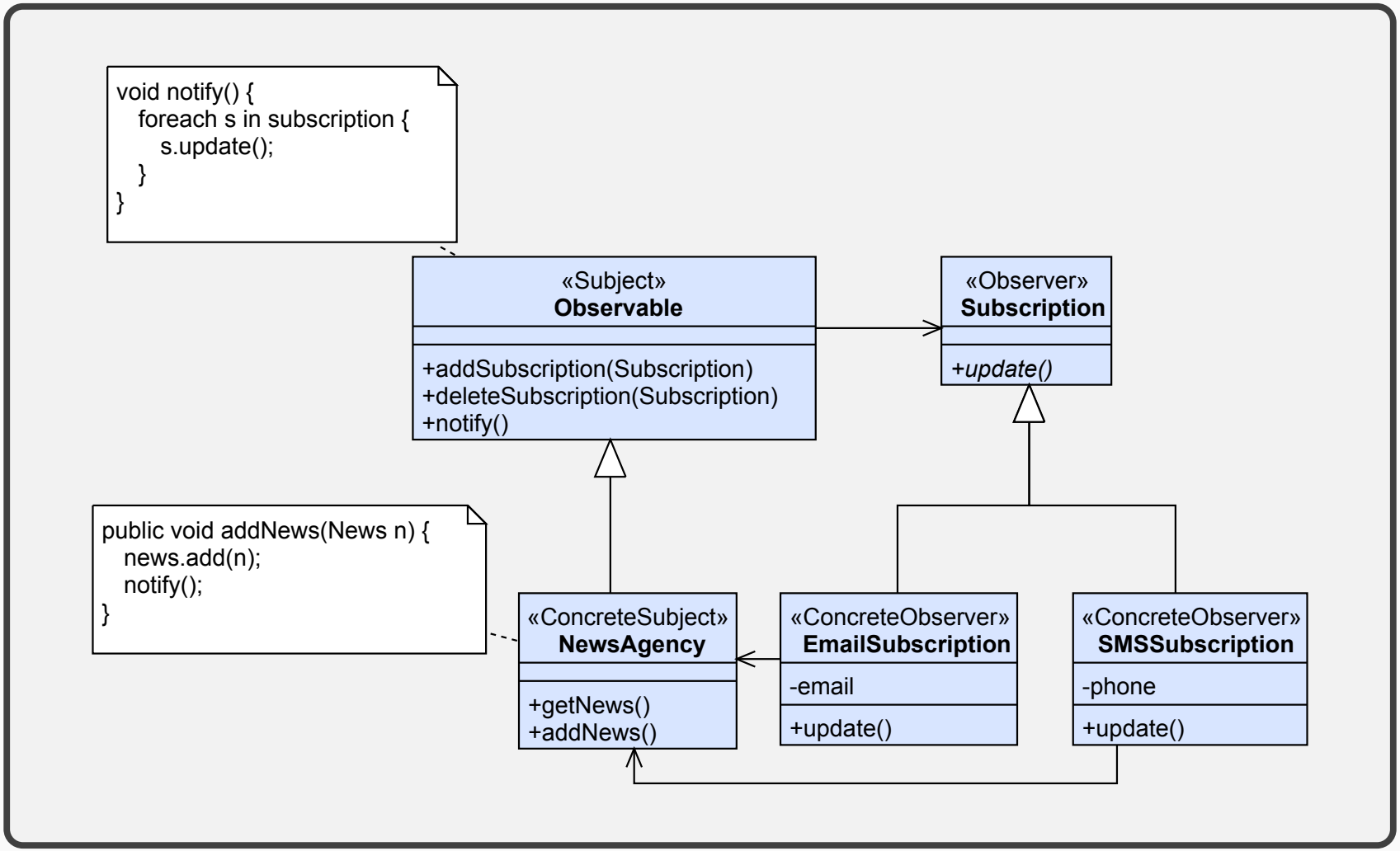
Solución: Patrón Observer (de comportamiento)



[Gamma et al., 1994]

Solución

Patrón Observer



Discusión

Ventajas:

- El objeto observado no necesita saber a priori cuántos objetos debe notificar ni a qué tipo pertenecen.
- Al estar muy poco acoplados, estos objetos (Object y Observer) pueden estar en capas distintas de la aplicación.

Inconvenientes:

- Cada pequeña actualización en el objeto Subject desencadena una cascada de notificaciones a los observadores. Se puede evitar haciendo que otro objeto Cliente (el encargado de modificar el Subject) dispare las notificaciones después de realizar todas las modificaciones necesarias.

Discusión

Detalles de implementación: existen dos alternativas para que los observadores obtengan la información que les interesa:

- El objeto Subject pasa la información al Observer a través del método `update()` $\rightarrow$ **modelo push**. Requiere que el Subject conozca mejor a los Observers, aumenta el acoplamiento.
- El objeto Observer se encarga de recuperar la información que necesita después de ser notificado $\rightarrow$ **modelo pull**. El Observer debe hacer comprobaciones adicionales para averiguar qué ha cambiado, es más ineficiente.

Ejercicio 5

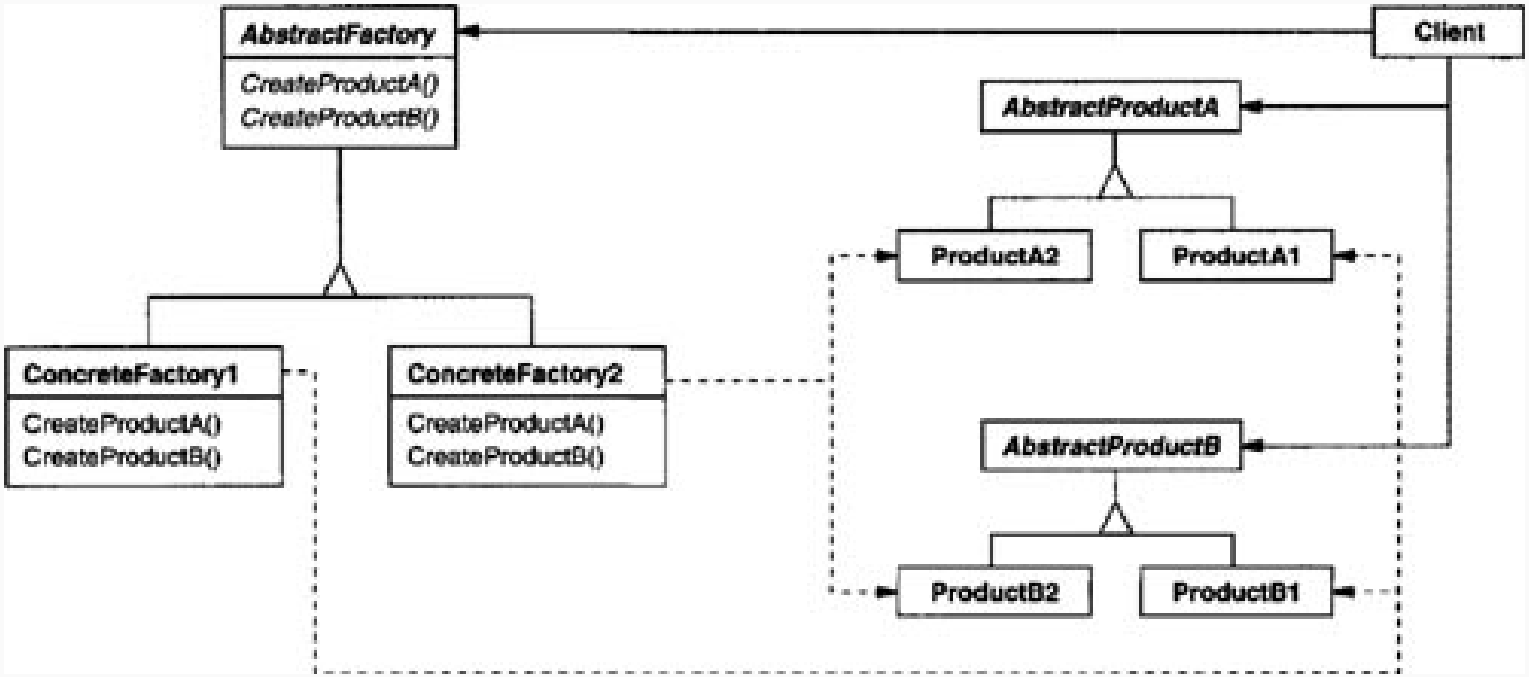
Enunciado

Queremos implementar un manejador de información personal, que controla, entre otras cosas, números de teléfono y direcciones. El alta de números de teléfono sigue una regla particular, que depende de la región y país a la que pertenezca el número. Sabemos que el número de países manejados por la aplicación va a crecer en un futuro, y queremos proteger nuestro código de esa variación. Propón un diseño que, utilizando un patrón GOF, solucione este problema.

Análisis del problema

Los datos siguen unas reglas de formato que dependen del país. Es conveniente proporcionar un mecanismo que **asegure la consistencia** y permita formatear correctamente los datos, indicando una única vez el país.

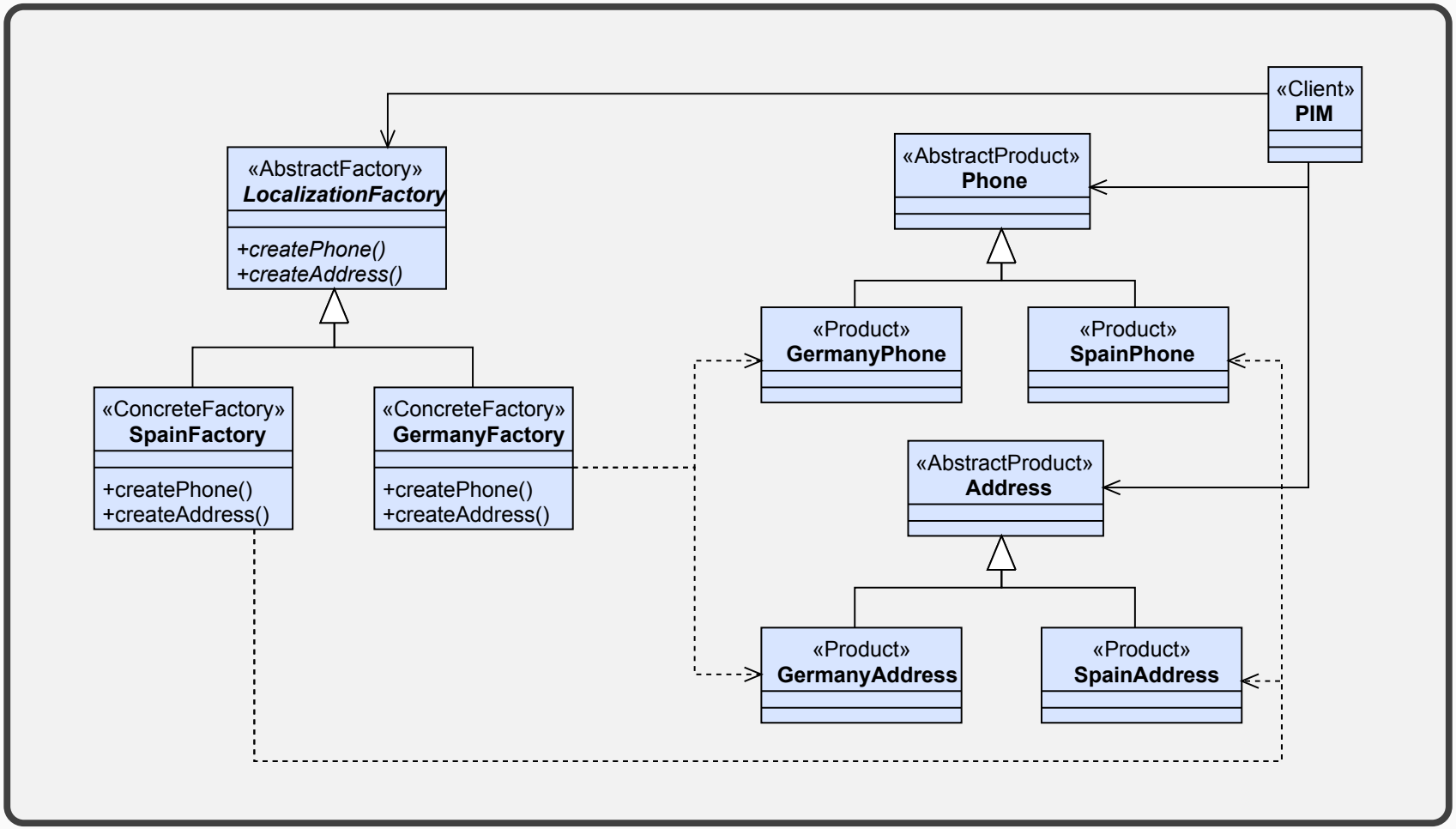
Solución: Patrón Abstract Factory (creacional)



[Gamma et al., 1994]

Solución

Patrón Abstract Factory



Discusión

Ventajas:

- Asegura la consistencia de los productos.
- El cliente sólo trata con clases abstractas, no necesita conocer las clases concretas.
- Es sencillo cambiar familias de productos (países en este ejemplo).

Inconvenientes:

- Involucra un número muy elevado de clases.
- Para añadir nuevos productos debemos modificar los objetos ConcreteFactory, lo que implica modificar también la clase abstracta AbstractFactory.

Ejercicio 6

Enunciado

Nos han pedido implementar un sistema para manejar el envío de artículos a conferencias. Una conferencia tiene distintas fases; por simplicidad vamos a asumir cuatro: **Call4Abstracts, Call4Papers, Revisión, Call4CameraReady**. El diseño inicial contiene tres clases: autor, artículo y conferencia. El sistema debe soportar cinco llamadas de interfaz:

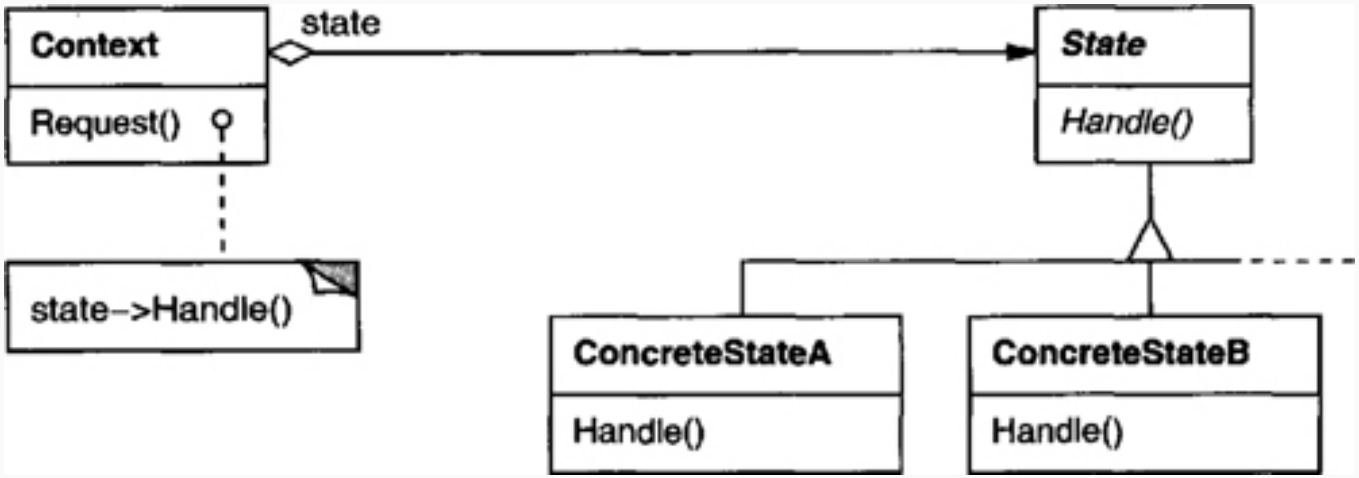
- `enviaAbstract()`
- `enviaArtículo()`
- `revisaArtículo()`
- `notificaRevisión()`
- `enviaCameraReady()`

El comportamiento de estas operaciones variará según la fase del proceso de revisión en que nos encontremos. Plantea una solución a este problema.

Análisis del problema

Para cambiar fácilmente el comportamiento de una clase dependiendo de su estado podemos **externalizar su comportamiento** fuera de la clase, proporcionando **distintas implementaciones que representan los distintos estados**.

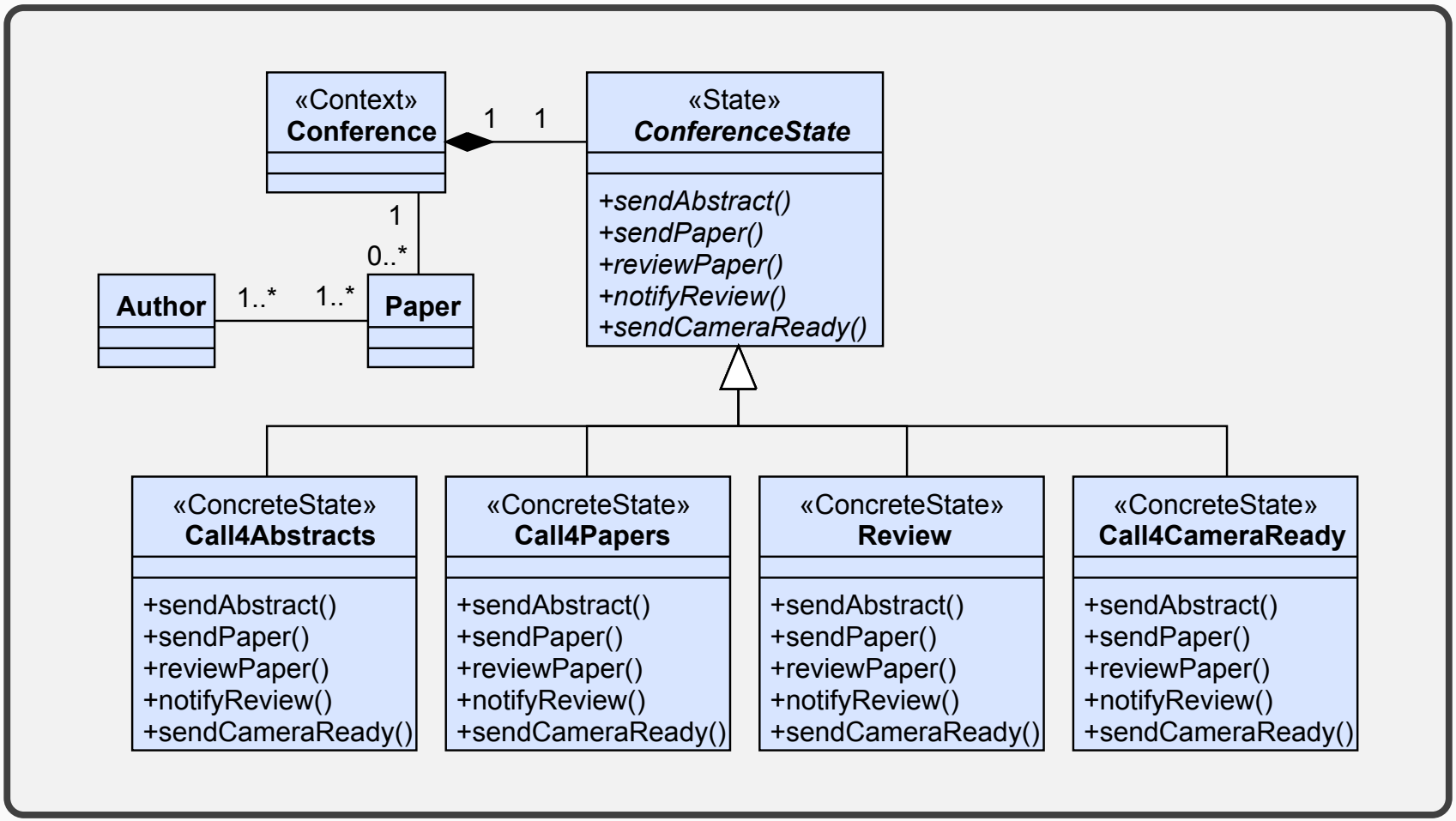
Solución: Patrón State (de comportamiento)



[Gamma et al., 1994]

Solución

Patrón State



Discusión

Ventajas:

- Las operaciones ya no necesitan lógica condicional para comprobar el estado del objeto.
- Toda la lógica asociada a un estado se encuentra en el mismo objeto (mayor cohesión).
- Hace que el cambio de estado sea una operación explícita y atómica, evitando posibles problemas de consistencia interna.

Inconvenientes:

- Todos los estados deben proporcionar todas las operaciones, aunque no las necesiten.

Discusión

Detalles de implementación: los cambios de estado se pueden implementar de dos maneras:

- El objeto principal puede ser el encargado de decidir cuál es el estado siguiente. Permite mantener la gestión de los estados en un punto centralizado.
- Los propios objetos State pueden ser los encargados de proporcionar el siguiente estado, p.ej. mediante un método `nextState()`. Reduce la complejidad del objeto principal pero introduce acoplamiento entre los distintos estados.

Ejercicio 7

Enunciado

Imagine que el profesor Jackson llama al profesor Ernst a media noche porque alguien ha descubierto que hay un problema relacionado con el sistema que están desarrollando para la famosa cadena de tiendas Decathlon: éste debe ser capaz de soportar bicicletas que se puedan volver a pintar (para cambiar su color). Los profesores dividen el trabajo: el profesor Jackson escribirá la clase ColorPalette con un método que, dado un nombre como “rojo”, “azul” o “ceniza”, devuelva un array con tres valores RGB, y el profesor Devadas escribirá un código que utilice esta clase. Los profesores hacen esto, pasan los tests al trabajo realizado, se van de fin de semana, y dejan los archivos `.class` para que los becarios del proyecto los integren.

Enunciado

Estos se dan cuenta de que el profesor Devadas ha escrito un código que depende de:

```
interface ColorPalette {
    // devuelve valores RGB
    int[] getColor(String name);
}
```

Sin embargo, el profesor Jackson implementa una clase que se adhiere a:

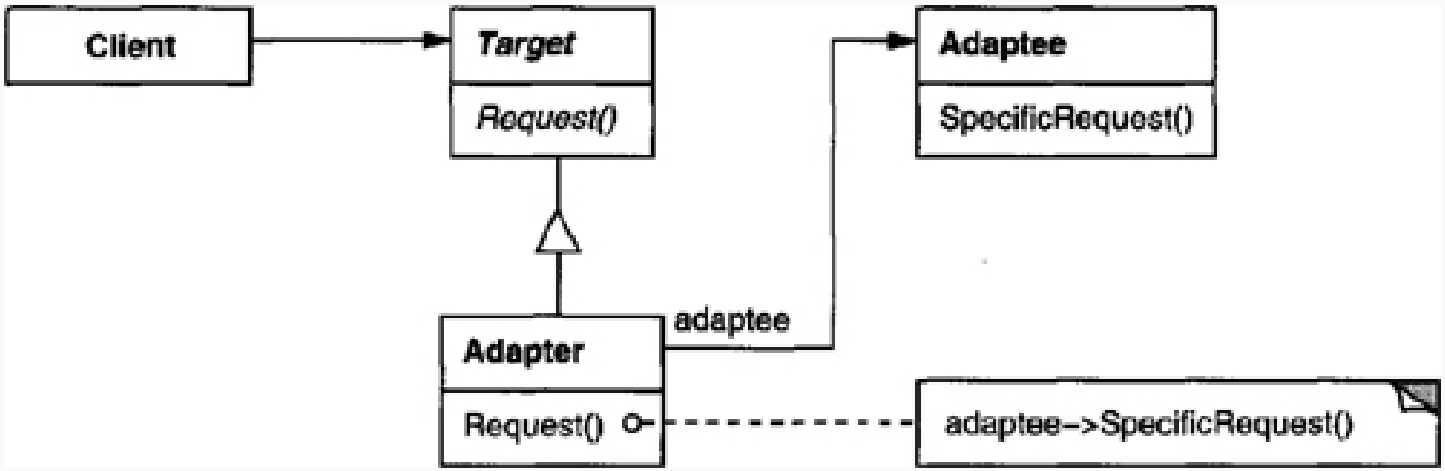
```
interface ColourPalette {
    // devuelve valores RGB
    int[] getColores(String name);
}
```

¿Qué es lo que los becarios tienen que hacer? Ellos no tienen acceso a la fuente, y no disponen de tiempo para volver a implementar y volver a pasar las pruebas.

Análisis del problema

Para solucionar el problema hay que proporcionar una clase que implemente el interfaz que espera el cliente, y se encargue de **adaptar este interfaz** al que proporciona la clase que implementa la funcionalidad.

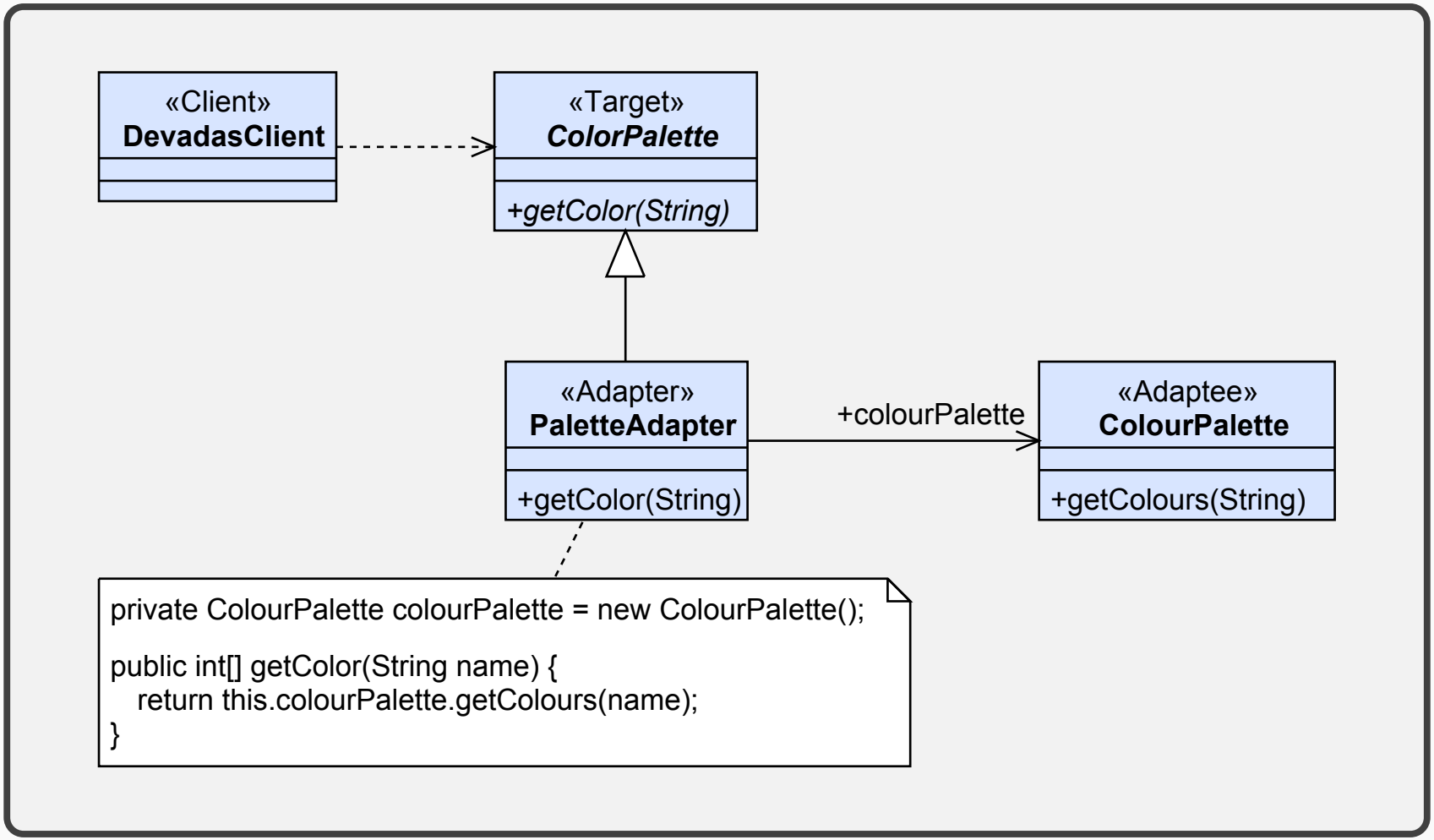
Solución: Patrón Adapter (estructural)



[Gamma et al., 1994]

Solución

Patrón Adapter



Discusión

Ventajas:

- Permite conectar las dos clases sin tener que cambiar su implementación.
- Permite añadir funcionalidad adicional si fuera necesario.

Inconvenientes:

- Al añadir un nivel de indirección puede afectar ligeramente al rendimiento.

¿Preguntas?

Referencias i


Gamma, E., Helm, R., Johnson, R., and Vlissides, J. M. (1994).
***Design Patterns: Elements of Reusable Object-Oriented Software*.**
Addison-Wesley Professional.
Safari Books Online.