Ejercicios sobre patrones GOF (2)

Diseño de Sistemas Software
Curso 2017/2018

Carlos Pérez Sancho

Universitat d'Alacant
Universidad de Alicante
Departament de Llenguatges i Sistemes Informàtics
Departamento de Lenguajes y Sistemas Informáticos

Ejercicio 8

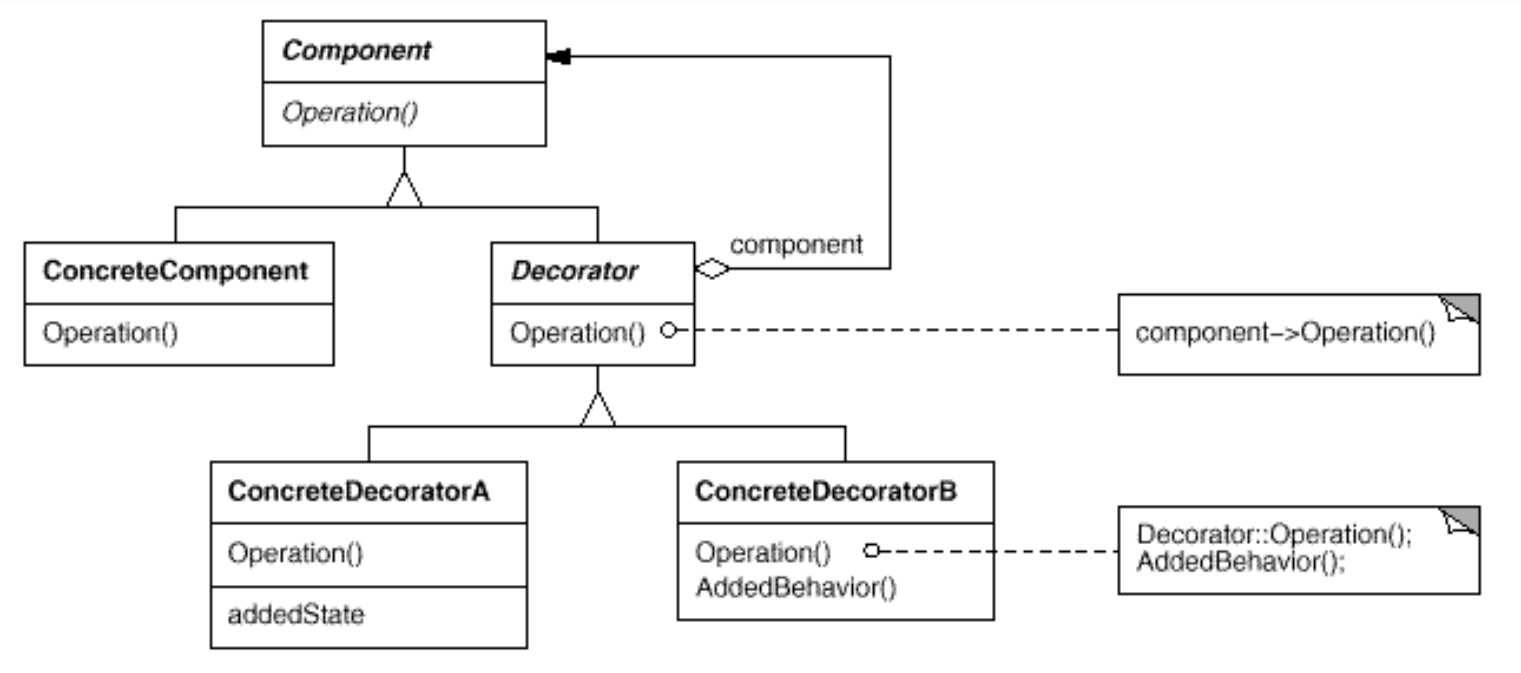
Enunciado

Muy contentos del trabajo realizado con el manejo de las partes de la bicicleta, Decathlon nos ha pedido un nuevo prototipo. Esta vez quiere que se extienda la funcionalidad de la aplicación para poder trabajar con distintas variaciones de las propiedades de las piezas. Por ejemplo, un manillar tiene un precio base, que puede verse incrementado si queremos algún “extra” (p.ej. pintura metalizada, materiales de gama superior, etc.). Partiendo del diseño anterior, enriquécelo para que soporte esta nueva variación.

Análisis del problema

Para poder calcular el precio teniendo en cuenta los extras y sin modificar el código cliente, necesitamos **añadir nuevas funcionalidades dinámicamente a las piezas**, de manera que se pueda calcular el precio de una pieza **sin modificar su interfaz**.

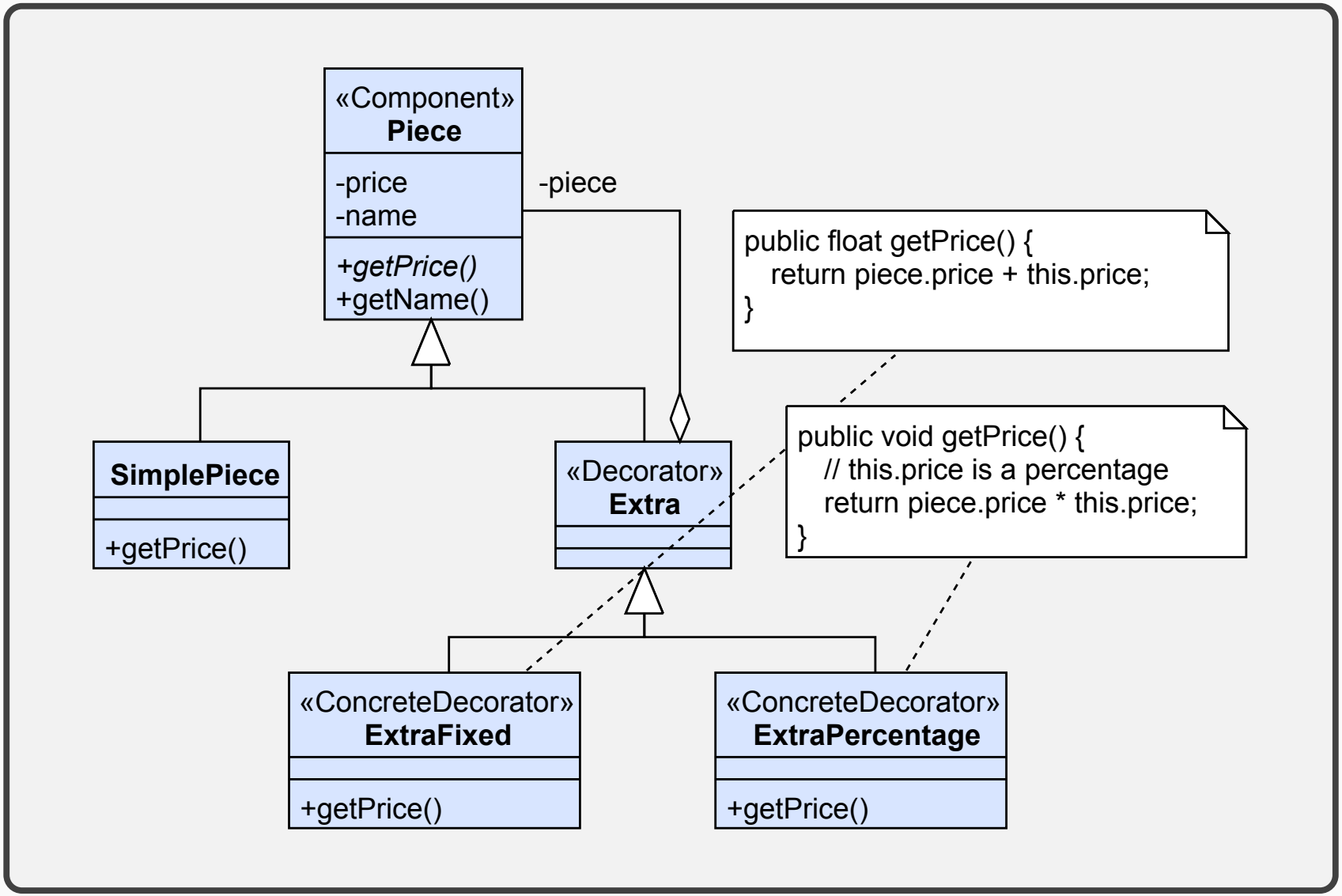
Solución: Patrón Decorator (estructural)



[Gamma et al., 1994]

Solución

Patrón Decorator



Discusión

Ventajas:

- El uso de los decoradores es transparente al cliente, ya que implementan el mismo interfaz que las piezas.
- Se pueden anidar tantos decoradores como sea necesario.
- Se pueden añadir y quitar decoradores en tiempo de ejecución.

Inconvenientes:

- Se crean muchos objetos pequeños, puede dificultar la comprensión del sistema.

Ejercicio 9

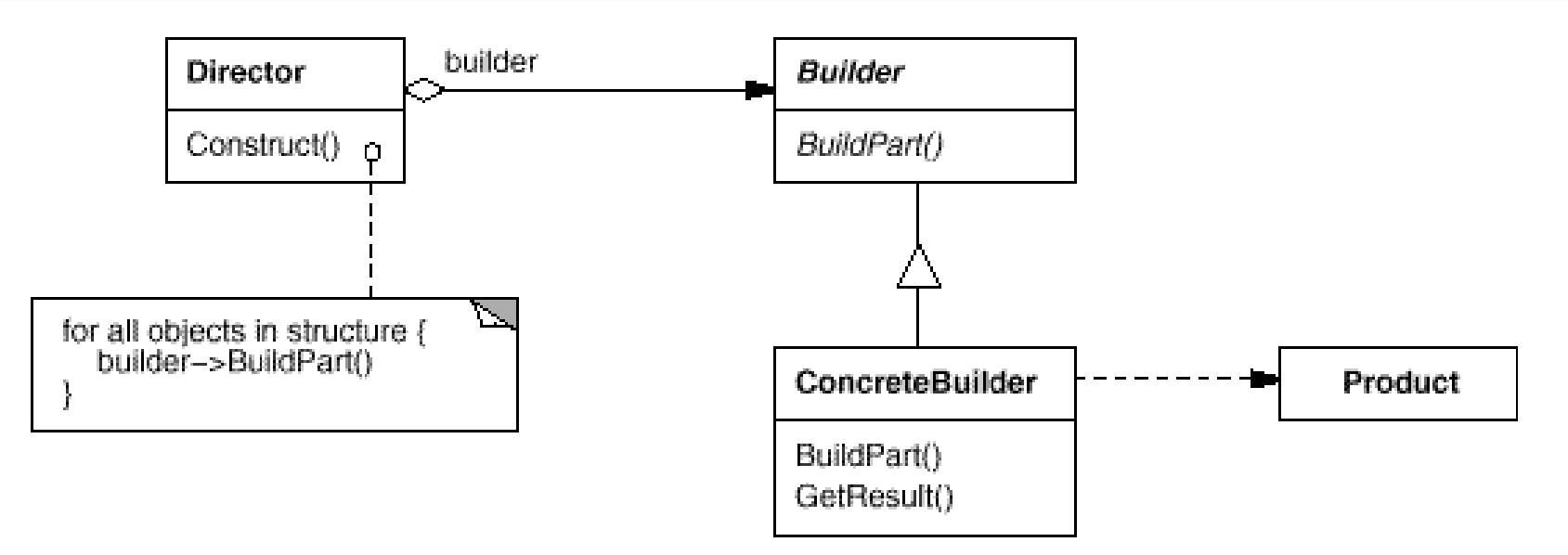
Enunciado

Imaginad que estamos construyendo una aplicación para restaurantes de comidas rápidas que preparan menús infantiles. Para ello hemos creado una clase Menú, de la que hereda Menú infantil. Generalmente estos menús están formados de un plato principal, un acompañamiento, una bebida y un juguete. Si bien el contenido del menú puede variar, el proceso de construcción es siempre el mismo: el cajero indica a los empleados los pasos a seguir. Estos pasos son: preparar un plato principal, preparar un acompañamiento, incluir un juguete y guardarlos en una bolsa. La bebida se sirve en un vaso y queda fuera de la bolsa. Completad el diseño.

Análisis del problema

Queremos **separar el proceso de construcción del menú de la lógica condicional** que hay en cada paso de la construcción. En cada paso hay que comprobar el tipo de menú, así que la solución es usar **objetos constructores para cada tipo de menú** que implementen todos los pasos de la construcción.

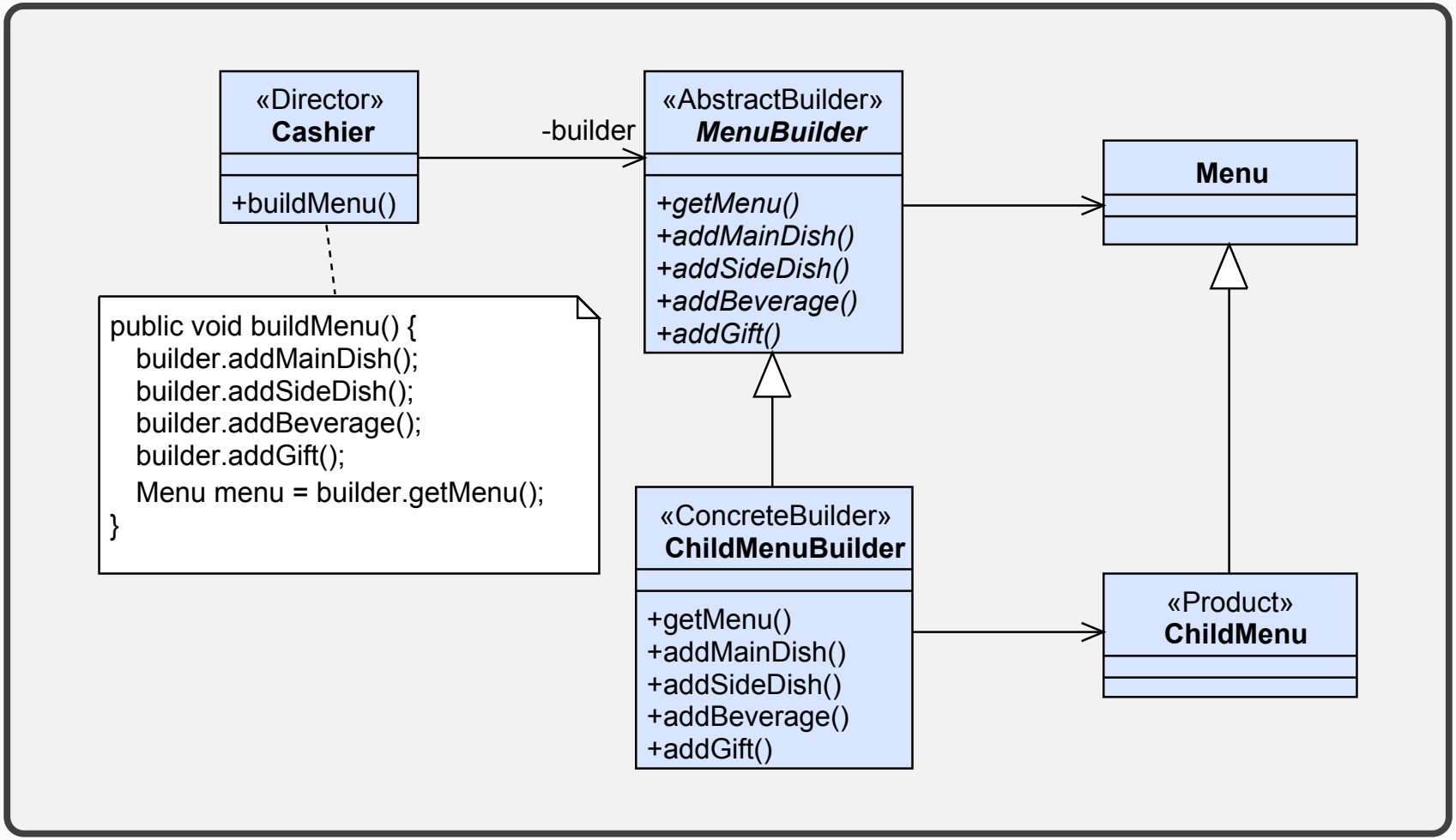
Solución: Patrón Builder (creacional)



[Gamma et al., 1994]

Solución

Patrón Builder



Discusión

Ventajas:

- Permite separar el algoritmo de construcción de los detalles sobre cómo crear las partes del objeto.
- Permite crear distintos tipos de objetos siguiendo el mismo algoritmo.

Inconvenientes:

- Puede haber productos que no necesiten alguna de las partes, en ese caso el ConcreteBuilder debe tener una implementación vacía del método.

Ejercicio 10

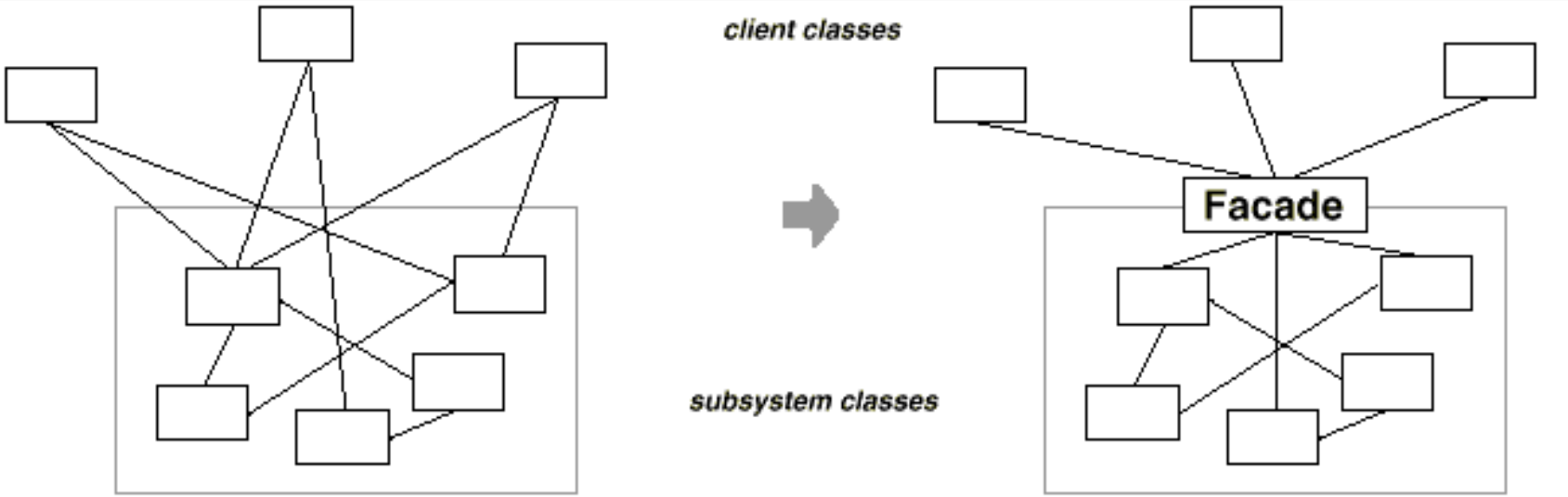
Enunciado

Sea un conjunto de clases que permiten la creación y envío de mensajes de correo electrónico y que entre otras incluye clases que representan el cuerpo del mensaje, los anexos, la cabecera, el mensaje, la firma digital y una clase encargada de enviar el mensaje. El código cliente debe interactuar con instancias de todas estas clases para el manejo de los mensajes, por lo que debe conocer en qué orden se crean esas instancias; cómo colaboran esas instancias para obtener la funcionalidad deseada y cuáles son las relaciones entre las clases. Idea una solución basada en algún patrón tal que se reduzcan las dependencias del código cliente con esas clases y se reduzca la complejidad de dicho código cliente para crear y enviar mensajes. Dibuja el diagrama de clases que refleje la solución e indica qué patrón has utilizado.

Análisis del problema

Queremos **evitar un acoplamiento excesivo** con todas las clases involucradas para el envío de un correo electrónico. Para esto introducimos un nivel de indirección, **reduciendo el acoplamiento a una única clase fachada**.

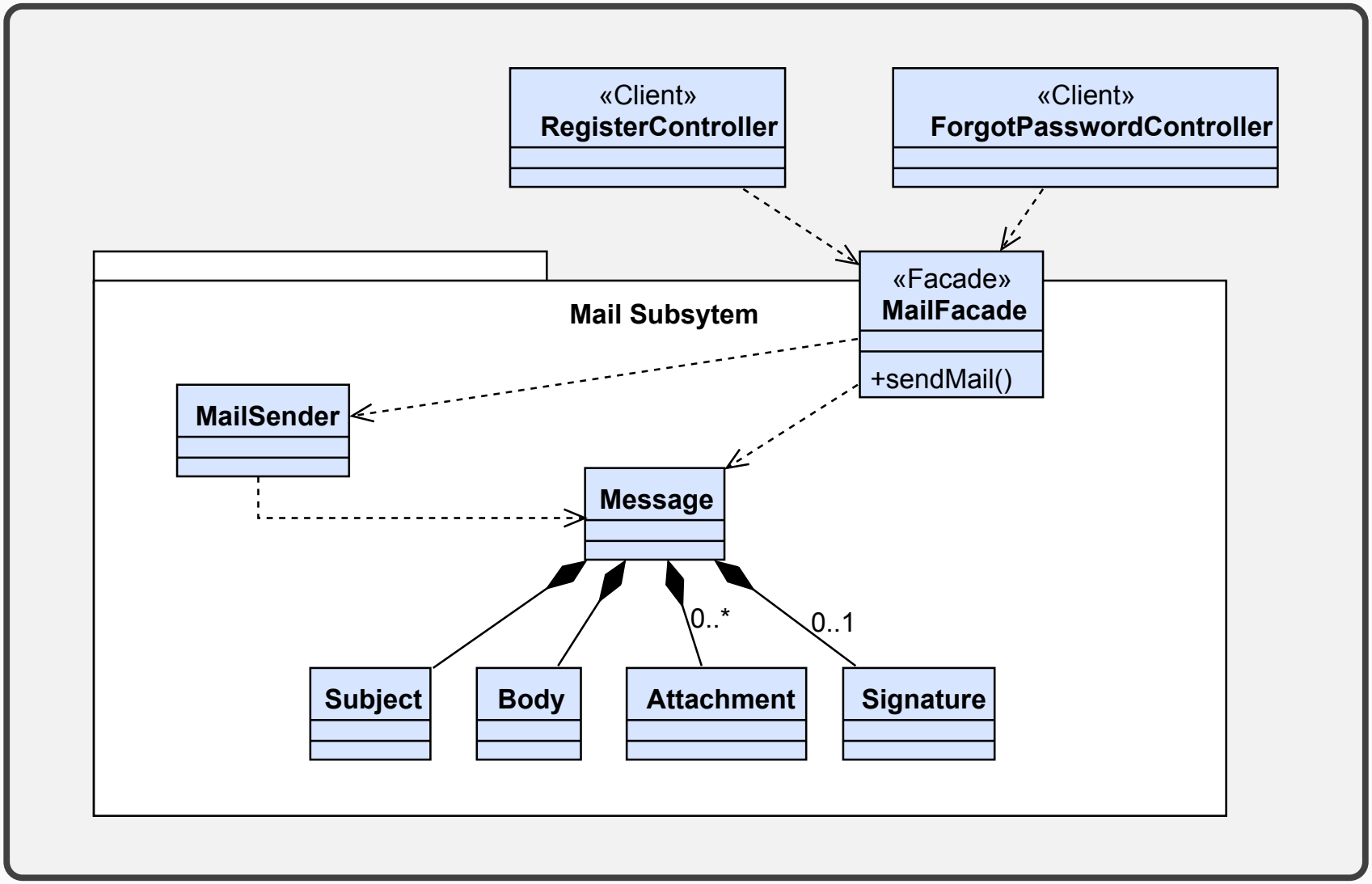
Solución: Patrón Facade (estructural)



[Gamma et al., 1994]

Solución

Patrón Facade



Discusión

Ventajas:

- Proporciona un interfaz fácil de usar para un subsistema complejo.
- Desaparecen las dependencias entre numerosos clientes y las clases del subsistema.
- Permite reducir el acoplamiento entre distintas capas, haciendo que las capas se comuniquen mediante llamadas a sus fachadas.
- Aún así, las clases internas pueden ser utilizadas si es necesario.

Ejercicio 11

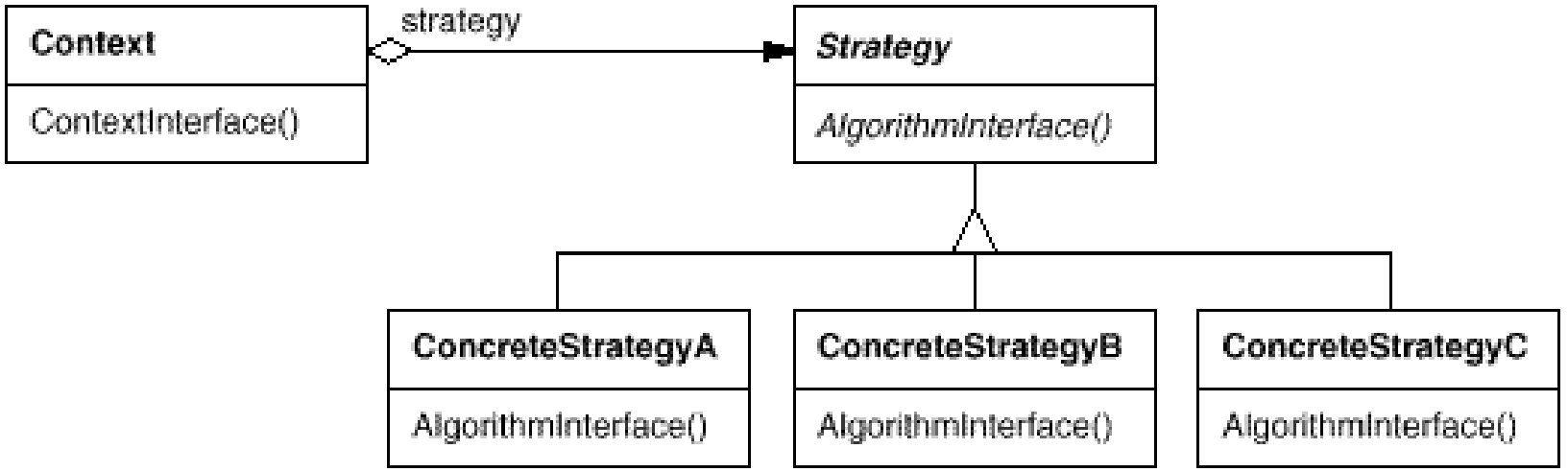
Enunciado

Suponed que estamos desarrollando un juego interactivo basado en la serie Héroes. El juego cuenta con distintos personajes, y cada uno tiene una serie de habilidades especiales: volar, lanzar rayos, leer la mente, regenerarse, teletransportarse, ... Los personajes pueden adquirir/perder poderes de forma dinámica según se desarrolla la trama del juego. Por ejemplo, el padre de Peter y Nathan Petrelli puede desposeer a cualquier personaje de sus poderes. Sylar, por otro lado, va adquiriendo poderes según va analizando a sus víctimas. Además, cualquier personaje puede ver inhabilitados sus poderes cuando están capturados o cuando el Haitiano está cerca o hay un eclipse. Los personajes también ven modificado el comportamiento de sus habilidades cuando están aturdidos.

Análisis del problema

Para poder **intercambiar distintos comportamientos en tiempo de ejecución** es necesario sacar la implementación de la clase, de manera que se ofrecen **distintas implementaciones en clases separadas con el mismo interfaz**.

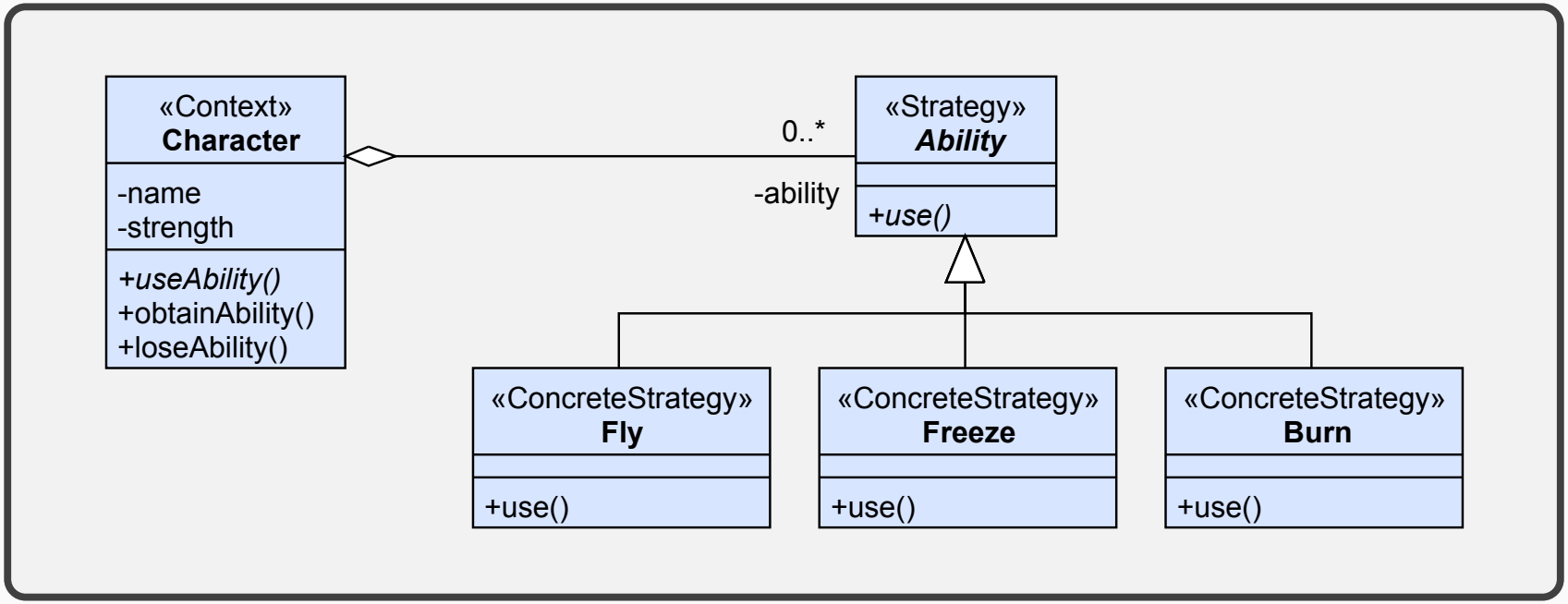
Solución: Patrón Strategy (de comportamiento)



[Gamma et al., 1994]

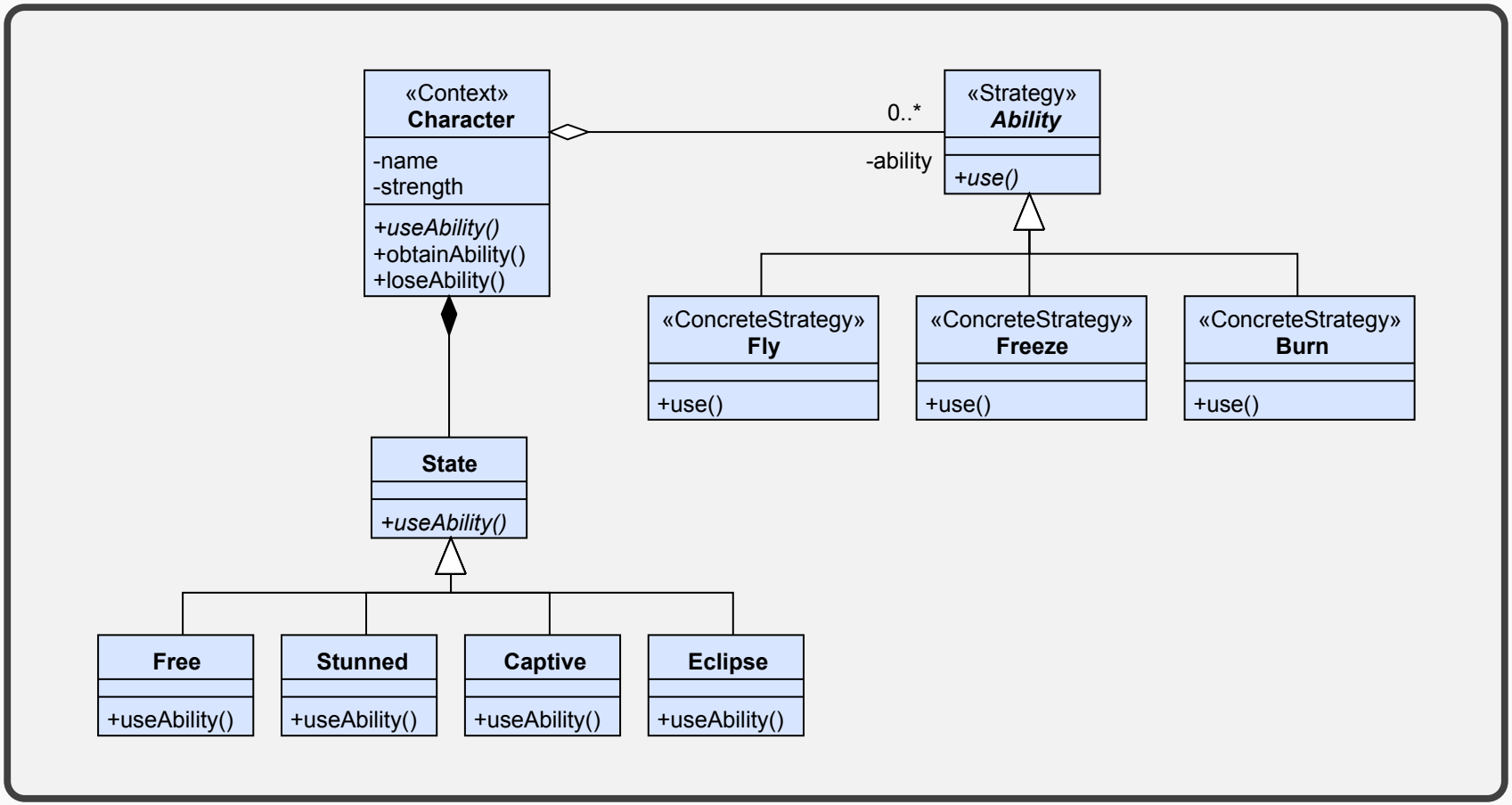
Solución

Patrón Strategy



Solución

Patrón Strategy + State



Discusión

Ventajas:

- Los distintos comportamientos se pueden sustituir en tiempo de ejecución, sustituyendo la instancia que guarda la clase cliente por otra implementación del algoritmo.
- Permite ocultar detalles de implementación del comportamiento que son irrelevantes para el cliente.
- Elimina lógica condicional en el cliente.

Inconvenientes:

- La clase cliente debe conocer todas las subclases que implementan el comportamiento. Se puede resolver usando una clase Factory para crear las instancias.

Ejercicio 12

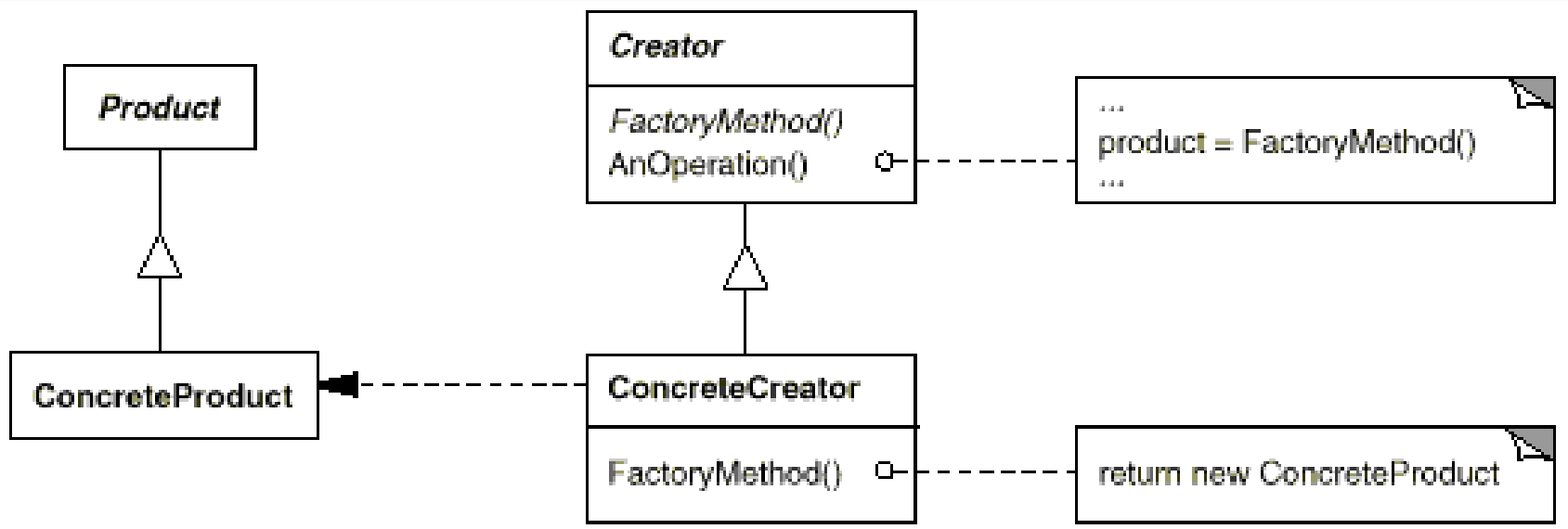
Enunciado

Nos han pedido colaborar en la elaboración de un framework para gestionar aplicaciones de manejo de documentos. Este framework debe asegurar, entre otras cosas, el correcto manejo de las peticiones de apertura, creación y salvado de documentos de cualquier tipo, donde el tipo concreto dependerá de la aplicación que extienda nuestro framework. Por ejemplo, una aplicación de dibujo contendrá una clase `AplicaciónDibujo`, y una clase `DocumentoDibujo`.

Análisis del problema

Tenemos una funcionalidad común a todas las aplicaciones (abrir, crear, guardar), que depende de la **creación de objetos que no se conocen a priori**. Se puede implementar esta funcionalidad delegando la creación de estos objetos en **subclases que implementen el método encargado de crearlos**.

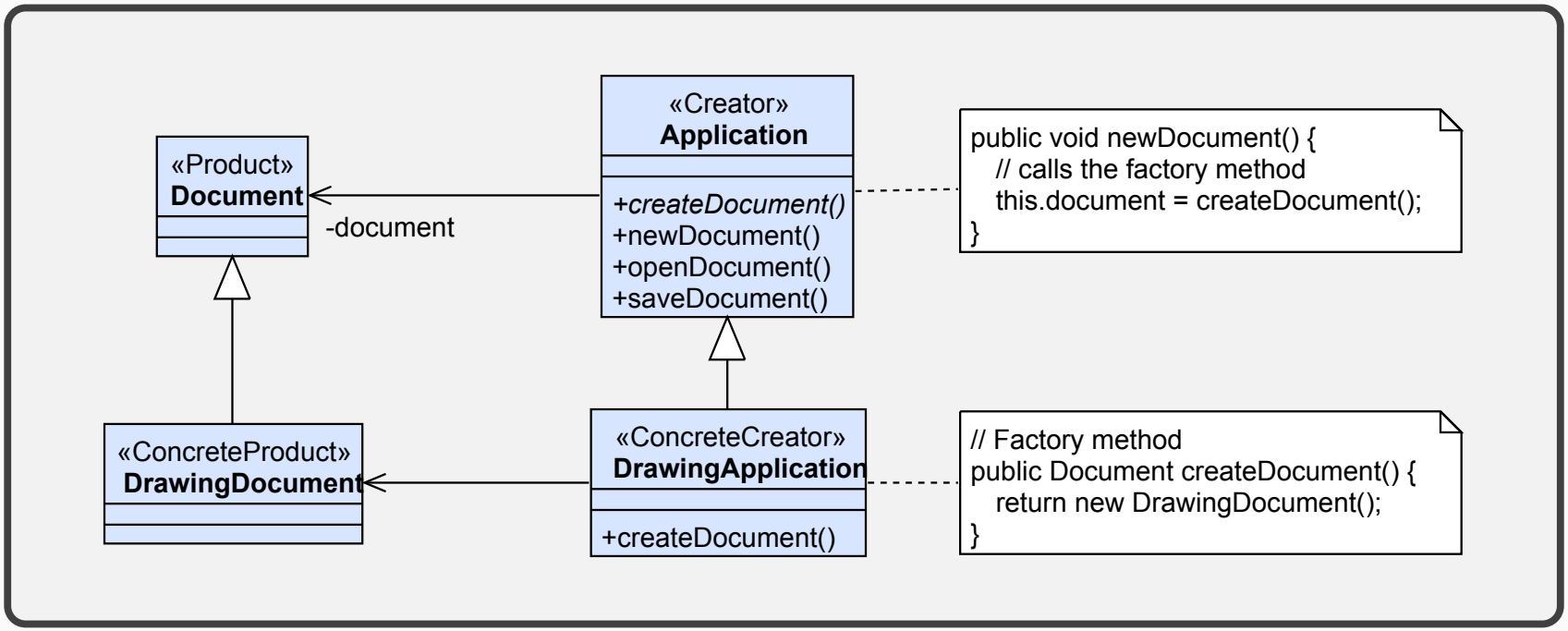
Solución: Patrón Factory Method (creacional)



[Gamma et al., 1994]

Solución

Patrón Factory Method



Discusión

Ventajas:

- Permite implementar una funcionalidad que depende de objetos que no se conocen.

Inconvenientes:

- No se puede aplicar si los distintos objetos a crear son muy diferentes y no se puede extraer un interfaz común.

Ejercicio 13

Enunciado

Nos han pedido implementar una aplicación para una agencia de viajes. La agencia maneja distintos tipos de paquetes; todos ellos comparten un protocolo común, que incluye lo siguiente:

- Todos los turistas son transportados al lugar de destino por el medio de transporte seleccionado
- Cada día tienen una actividad programada
- Todos los turistas vuelven (si quieren)

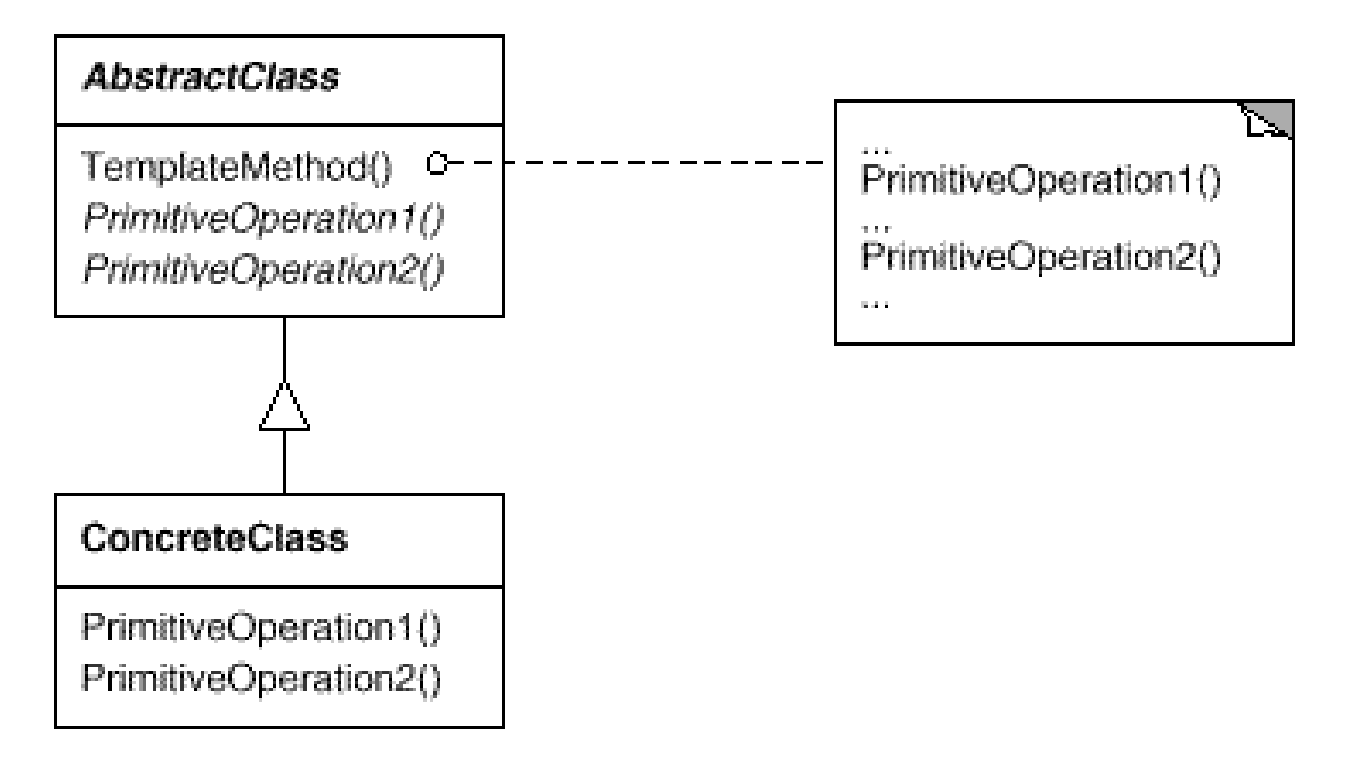
Sin embargo, según el paquete elegido, hay ciertas variaciones:

- Los turistas pueden ser llevados en transfer desde el hotel al lugar donde cojan su transporte o ir por su cuenta.
- Las actividades programadas pueden estar incluidas o no en el precio del paquete
- ...

Análisis del problema

Tenemos un **protocolo común, cuyos pasos se pueden implementar de distintas maneras**. La implementación de los pasos se hará en clases hijas que heredan el comportamiento común e implementan de forma diferente cada uno de los pasos.

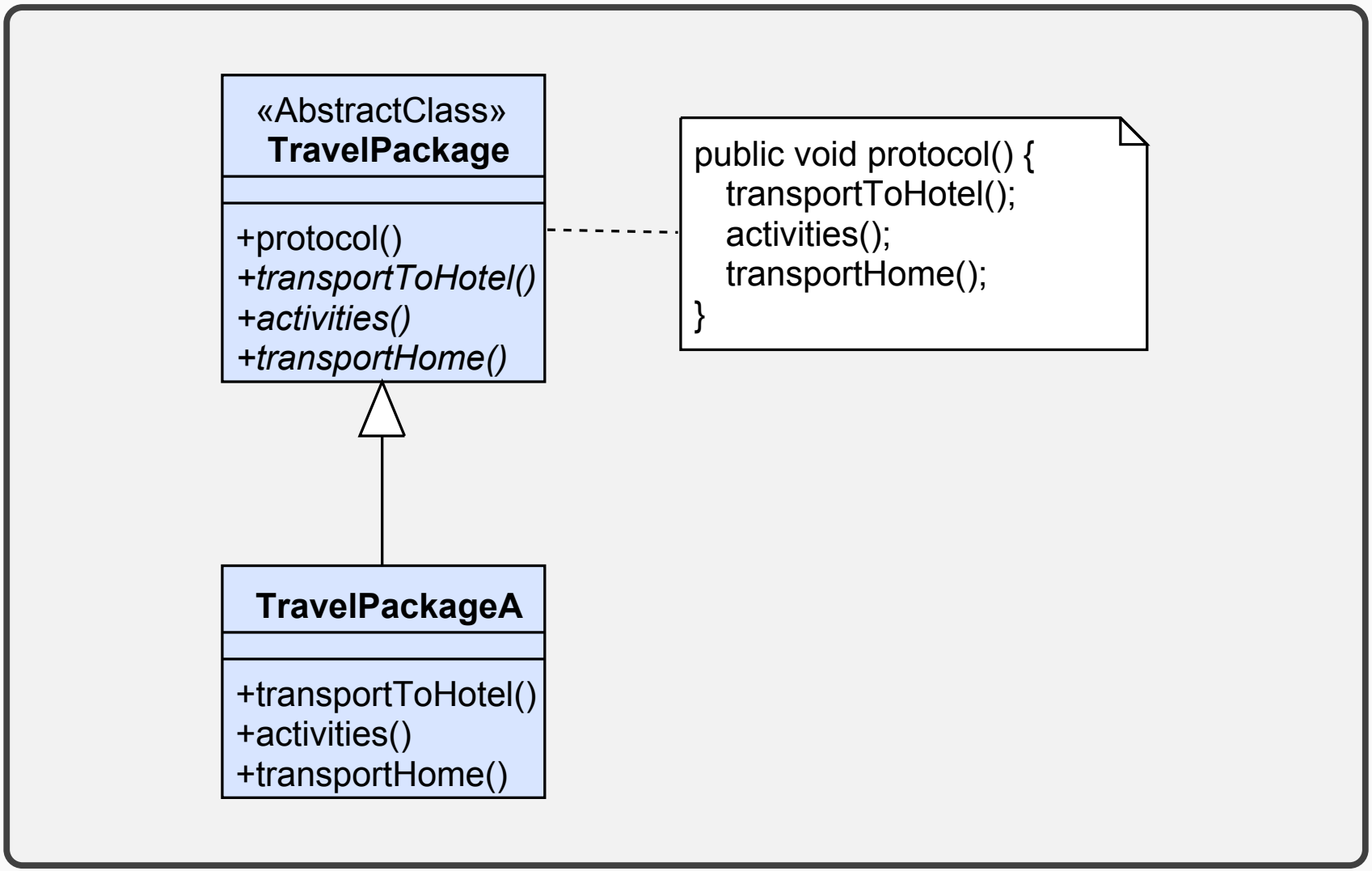
Solución: Patrón Template Method (de comportamiento)



[Gamma et al., 1994]

Solución

Patrón Template Method



Discusión

Ventajas:

- Permite implementar las partes invariantes de un algoritmo, delegando en las subclases la implementación de los detalles.
- Proporciona un mecanismo de inversión de control, muy usado en frameworks (p.ej. controladores y middleware en Laravel).

Detalles de implementación:

- No se debería permitir sobreescribir el comporamiento del template method.

Ejercicio 14

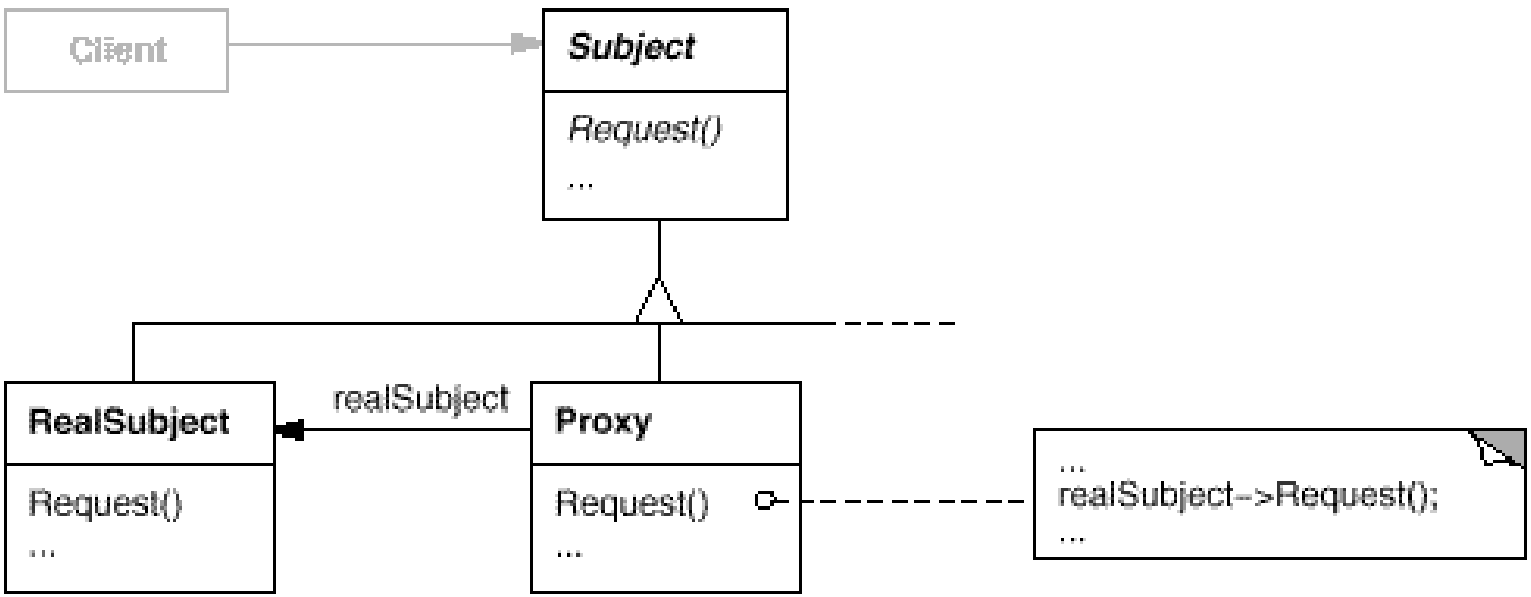
Enunciado

Se desea almacenar en una base de datos todas las transacciones realizadas por un terminal punto de venta (*point of sale*, POS). El terminal puede trabajar en modo conectado o no conectado, dependiendo de si la conexión a internet está en funcionamiento en el momento de almacenar los datos. Como es muy importante que no se pierda ninguna transacción, en el caso de que se esté trabajando en modo sin conexión, los datos deben almacenarse en el disco hasta que la conexión se restablezca.

Análisis del problema

Para no aumentar la complejidad del cliente podemos situar un **objeto intermediario** entre éste y el objeto que se comunica con la base de datos, **imitando su interfaz**. Este nuevo objeto se encargará de gestionar los datos en caso de que falle la conexión para que no se pierdan transacciones.

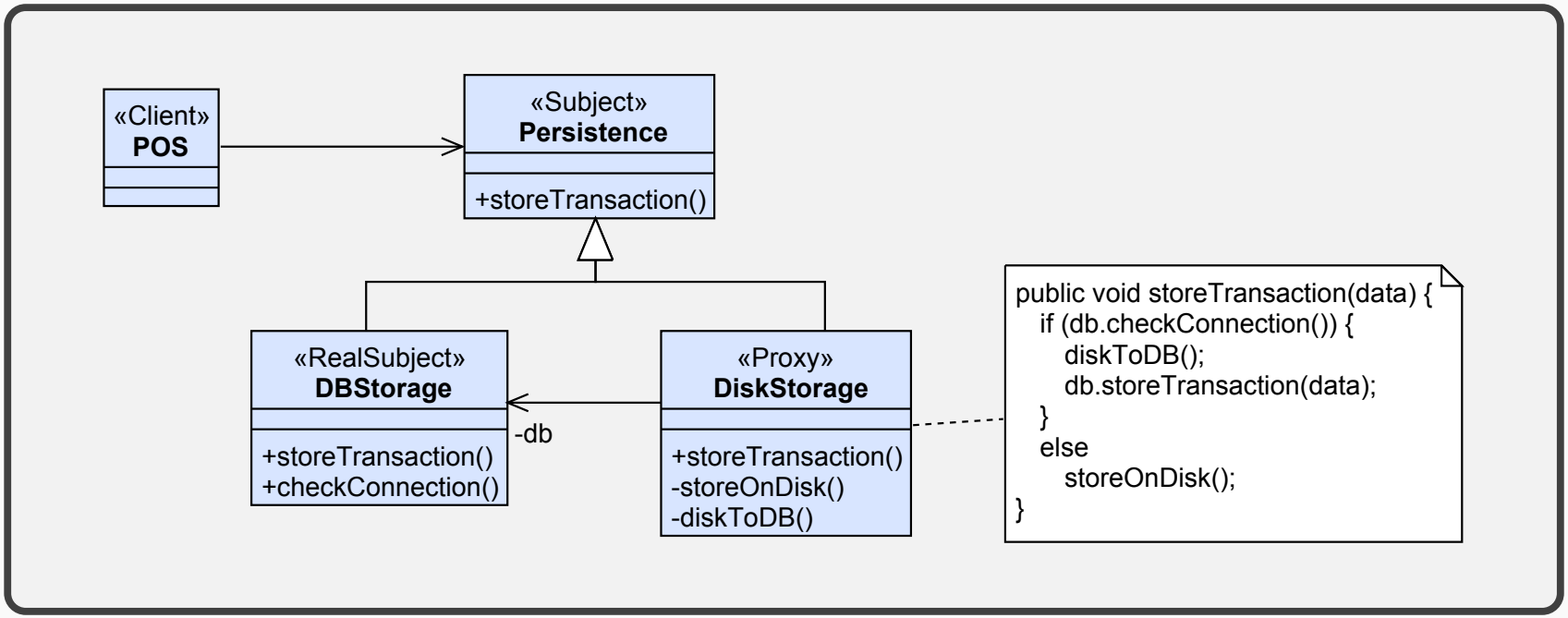
Solución: Patrón Proxy (estructural)



[Gamma et al., 1994]

Solución

Patrón Proxy



Discusión

Combinación de los patrones Decorator y Adapter.

Tipos de proxies:

- **Proxy virtual:** sustituyen objetos “caros” hasta que son necesarios. P.ej. para implementar la estrategia Lazy Loading.
- **Proxy remoto:** representa un objeto ubicado en un servidor remoto y se encarga de comunicarse con él.
- **Proxy de protección:** permite controlar el acceso a un objeto.
- ***Smart reference*:** permite realizar operaciones adicionales cuando se accede a un objeto. Por ejemplo:
  - Registrar todos los accesos al objeto real.
  - Liberar el objeto real cuando ya no hay objetos referenciándolo (*smart pointers*).
  - Bloquear el objeto real para evitar acceso concurrente.

¿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.