POLITÉCNICA
"Ingeniamos el futuro"

CAMPUS
DE EXCELENCIA
INTERNACIONAL

MiW

Patrones de Diseño

6. *Adapter*

Luis Fernández Muñoz
https://www.linkedin.com/in/luisfernandezmunyoz
setillofm@gmail.com

INDICE

1. Problema

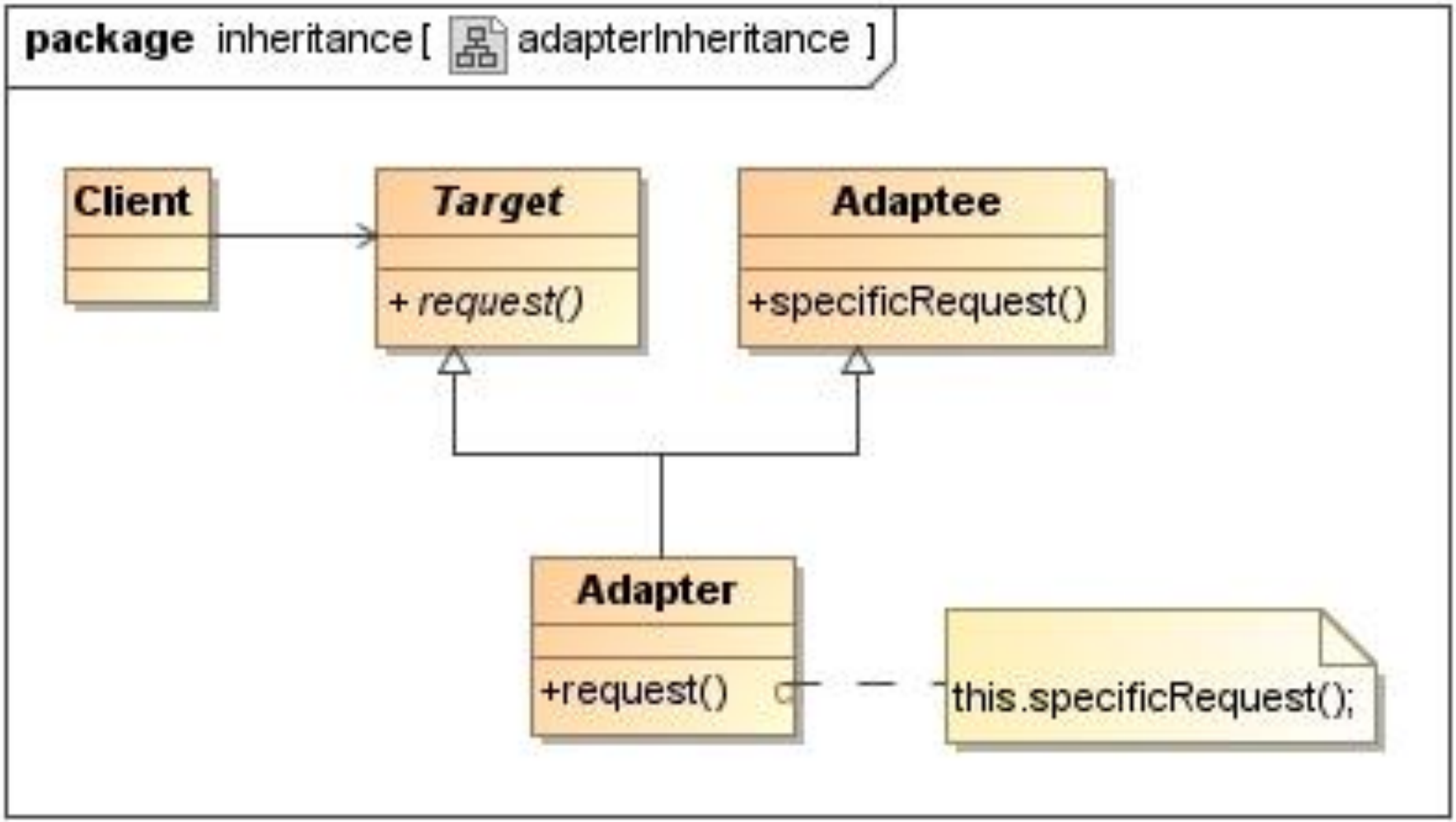
- *Cuando vas de viaje al extranjero, existen problemas al enchufar los aparatos eléctricos (p.e. secador del pelo, maquinilla de afeitar, …) porque el numero, forma y/o disposición de las patillas de los enchufes no son iguales en todos los países, hay varios estándares.*
- *Un solución costosa y engorrosa sería comprar nada más llegar al destino todos los aparatos eléctricos en el país de destino y guardarlo en el trastero hasta que se vuelva a viajar a un país del mismo estándar.*
- *Otra solución barata y sencilla es comprar adaptadores de enchufes antes de viajar y guardarlos a mano para el próximo viaje a un país con el mismo estándar.*

- ***Convierte la interfaz de una clase en otra interfaz que el cliente espera y permite trabajar con clases conjuntamente que no podrían de otra forma por la incompatibilidad de sus interfaces***

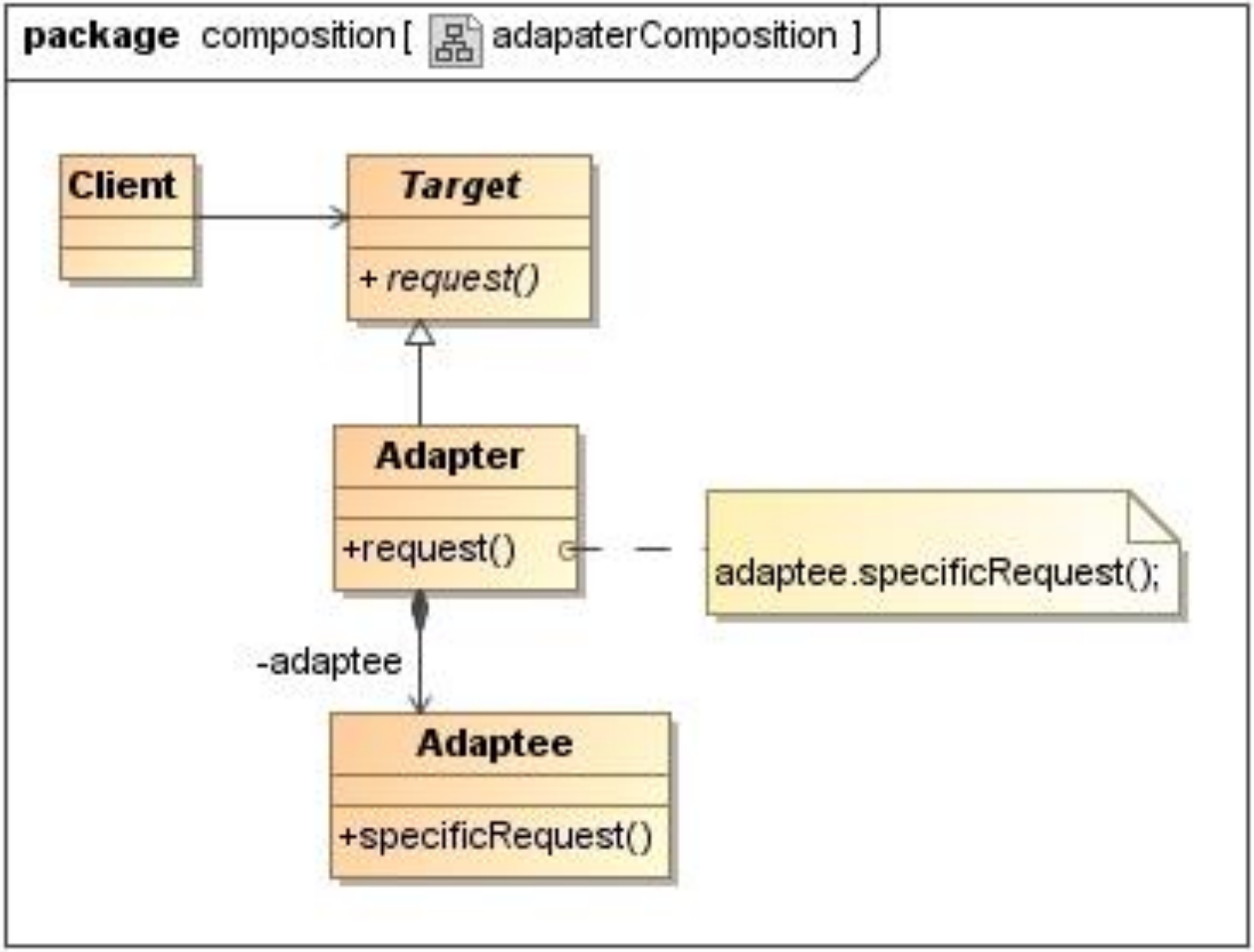
1. Problema

- Debería usarse cuando:
  - Se quiere usar una clase existente (p.e. de una biblioteca que ha sido diseñada para reutilizarse) y su interfaz no concuerda con la que se necesita (específica del dominio que requiere la aplicación)
  - Se quiere crear una clase reutilizable que coopere con clases no relacionadas o que no han sido previstas, es decir, clase que no tienen por qué tener interfaces compatibles

2. Solución


package inheritance [adapterInheritance]
Client
Target
+ request()
Adaptee
+specificRequest()
Adapter
+request()
this.specificRequest();

2. Solución


package composition [adapaterComposition]
Client
Target
+ request()
Adapter
+request()
adaptee.specificRequest();
-adaptee
Adaptee
+specificRequest()

2. Solución

- ***Adapter* de clases**: en una implementación en C++ de un *Adapter* de clases, el *Adapter* debería heredar públicamente de *Target* y privadamente de *Adaptee*. Así, *Adapter* sería un subtipo de *Target* , pero no de *Adaptee*
  - Definir las correspondientes operaciones abstractas para la interfaz reducida de *Adaptee*. Las subclases deben implementar las operaciones abstractas y adaptar el objeto estructurado jerárquicamente.
- Adapta una clase *Adaptee* a *Target*, pero se refiere únicamente a una clase *Adapter* concreta. Por tanto, un *Adapter* de clases no nos servirá cuando lo que queremos es adaptar una clase y todas sus subclases

2. Solución

- ***Adapter* de objetos**: es necesario usar varias subclases existentes, pero no resulta práctico adaptar su interfaz heredando de cada una de ellas. Un *Adapter* de objetos puede adaptar la interfaz de su clase padre
  - El cliente reenvía las peticiones para acceder a la estructura jerárquica a un objeto delegado.
  - El cliente puede usar una estrategia de adaptación diferente cambiando el objeto delegado

3. Consecuencias

- Un adaptador de clases:
  - Permite que *Adapter* redefina parte del comportamiento de *Adaptee*, por ser *Adaptee* una subclase de *Adaptee*
  - Introduce un solo objeto y no se necesita ningún puntero de indirección adicional para obtener el objeto adaptado
- Un adaptador de objetos:
  - Permite que un mismo *Adapter* funcione con muchos *Adaptee* – es decir, con el *Adaptee* en sí y todas sus subclases, en caso de que las tenga -. El *Adapter* puede añadir funcionalidad a todos los *Adaptee* a la vez
  - Hace que sea más fácil redefinir el comportamiento de *Adaptee*. Se necesitará crear una subclase de *Adaptee* y hacer que el *Adapter* se refiera a la subclase en vez de a la clase *Adaptee* en sí.

3. Consecuencias

- Una clase es más reutilizable cuando minimizamos las asunciones que deben hacer otra clases para usarla. Al hacer la adaptación de interfaces en una clase estamos eliminando la asunción de que otras clases ven la misma interfaz. Dicho de otro modo, la adaptación de interfaces nos permite incorporar nuestra clase a sistemas existentes que podrían esperar interfaces diferentes a la de la clase.
- Los adaptadores difieren en la cantidad de trabajo necesaria para adaptar *Adaptee* a la interfaz *Target*.
  - Hay una amplia gama de tareas posibles, que van desde la simple conversión de interfaces – por ejemplo, cambiar los nombres de las operaciones – a permitir un conjunto completamente nuevo de operaciones. La cantidad de trabajo que hace el *Adapter* depende de lo parecida que sea la interfaz de *Target* a la de *Adaptee*

3. Consecuencias

- Un problema potencial de los adaptadores es que no son transparentes a todos los clientes. Un objeto adaptado ya no se ajusta a la interfaz Adaptable, por lo que puede usarse tal cual en donde pudiera ir un objeto Adaptable. Los Adaptadores bidireccionales pueden proporcionar esa transparencia. En concreto, resultan útiles cuando dos clientes distintos necesitan ver un objeto de distinta forma.
  - La solución consiste en un adaptador que adapta cada una de las dos interfaces a la otra. La herencia múltiple es una solución viable en esta caso porque las interfaces de las clases adaptadas son sustancialmente diferentes. El adaptador de clases bidireccional se ajusta a las dos clases adaptadas y puede trabajar en cualquiera de los dos sistemas

4. Comparativa

- ***Adapter** vs **Bridge**.*
  - Tienen atributos comunes:
    - promueven la flexibilidad al proporcionar un nivel de indirección a otro objeto.
    - implican reenviar peticiones a este objeto desde una interfaz distinta de la suya propia
  - Tienen una estructura similar pero con un propósito diferente:
    - *Adapter* se centra en resolver incompatibilidades entre dos interfaces existentes. No presta atención a cómo se implementan dichas interfaces, ni tiene en cuenta cómo podrían evolucionar de forma independiente. Es un modo de lograr que dos clases diseñadas independientemente trabaje juntas sin tener que volver a implementar una a otra.
    - *Bridge* une una implementación con sus implementaciones (que pueden ser múltiples). Proporciona una interfaz estable a los clientes permitiendo, no obstante, que cambien las clases que la implementan. También permite incorporar nuevas implementaciones a medida que evoluciona el sistema

4. Comparativa

- ***Adapter** vs **Bridge*** (cont.).
  - Se usan en diferentes puntos del ciclo de vida del software:
    - *Adapter* suele hacerse necesario cuando se descube que deberían trabajar juntas dos clases incompatibles, generalmente para evitar duplicar código, y este acoplamiento no había sido previsto. *Adapter* hace que las cosas funcionen después de que han sido diseñadas
    - *Bridge* sabe de antemano que una abstracción debe tener varias implementaciones, y que una y otras pueden variar independientemente. *Bridge* hace que las cosas funcionen antes de que han sido diseñadas
- ***Adapter** vs **Facade***.
  - *Facade* es como un *Adapter* para un conjunto de objetos.
  - *Facade* define una nueva interfaz, mientras que un *Adapter* reutiliza una interfaz existente.

4. Comparativa

- ***Adapter** vs **Decorator** .*
  - *Decorator* permite la composición recursiva, lo que no es posible con *Adapter* puro.
  - *Decorator* sólo cambia las responsabilidades de un objeto, no su interfaz, mientras que un *Adapter* le da a un objeto una interfaz completamente nueva.
  - Por tanto, *Decorator* es más transparente a la aplicación que un *Adapter*.
- ***Adapter** vs **Proxy.***
  - *Proxy* define un representante o sustituto de otro objeto sin cambiar su interfaz. *Adapter* proporciona una interfaz diferente para el objeto que adapta.
  - No obstante, un *Proxy* utilizado para protección de acceso podría rechazar una operación que el sujeto sí realiza, de modo que su interfaz puede ser realmente un subconjunto de las del sujeto como con *Adapter*.