Desarrollo rápido de aplicaciones: Patrones de Diseño

J. Baltasar García Perez-Schofield

Grupo IMO - SI1

Empleo de patrones

- Los patrones de diseño son, símplemente, en lenguaje llano, “maneras de hacer las cosas”.
- Su propósito es doble:
  - Proporcionar una solución a un problema adaptable a variaciones del mismo y escalable a diferentes parámetros. Por ejemplo, un soporte para una colección de objetos que permita tamaños tanto del orden de decenas como de miles.
  - Proporcionar soluciones robustas y bien conocidas.

¿Patrones de diseño = RAD?

- Los patrones de diseño no se corresponden exactamente con el RAD.
- Sin embargo, proporcionan soluciones a problemas de diseño ya sobradamente conocidos.
- Reducen las necesidades de retroalimentación en el ciclo de vida entre la fase de implementación y la de diseño, puesto que las soluciones que proponen son fácilmente escalables y adaptables.

¿Cuándo utilizar patrones?

- Los patrones de diseño, como su nombre indica, se aplican en la fase de diseño. Esto implica que:
  - La fase de análisis (¿qué es necesario hacer?) NO es cubierta por los patrones de diseño en absoluto.
  - La fase de diseño (¿cómo lo vamos a hacer?), es cubierta en buena parte por esta técnica.
  - Las fases de implementación y pruebas no se ven afectadas directamente por el uso de patrones, si bien se benefician de ellos (el *software* es más fácil de modificar y ampliar).

Tipos de patrones

- Creación
  - Abstract Factory, Factory Method, Prototype
- Estructurales
  - Bridge, Facade, Composite, Proxy
- Comportamiento
  - Chain of responsibility, Iterator, Observer, TemplateMethod

Referencias:

- http://www.dofactory.com/patterns/
- Gamma, et al. (2003). *Patrones de diseño*. Addison Wesley.

Enumeración de patrones

- A continuación, se exponen varios patrones.
- Son una selección de los patrones reales, en número mayor del expuesto aquí.

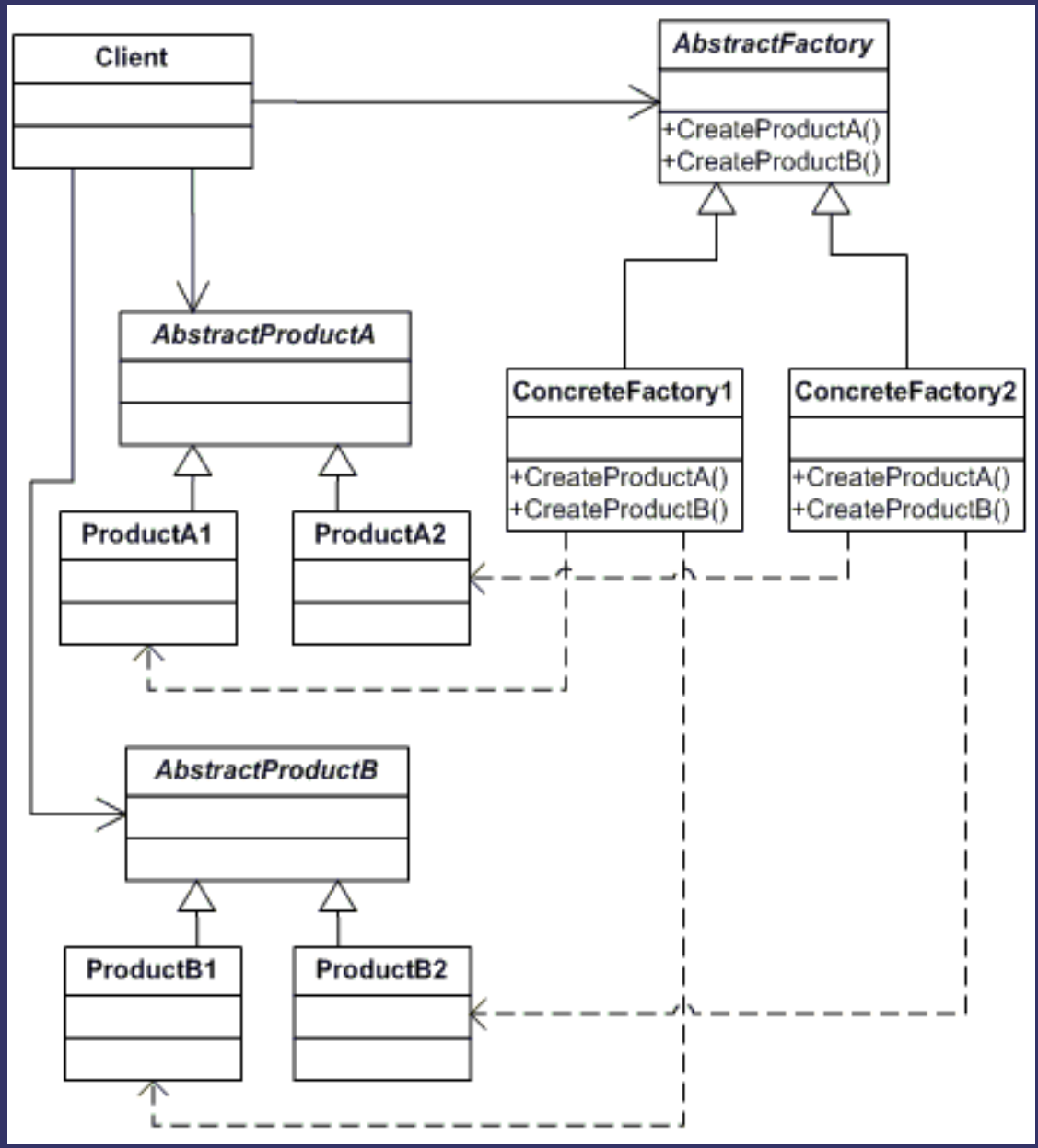
Patrones de creación

Abstract Factory

- Se trata de presentar una separación entre las clases de la interfaz de usuario y el programa en sí.
- Así, pone énfasis en la creación de objetos. Se trata de que, en un método de una clase A, una línea como:
  - `B * ptrb = new B;`
- ... crea un acoplamiento excesivo entre ambas clases.
- Esto se soluciona mediante una jerarquía de clases encargadas sólo de crear objetos (factorías).

Abstract Factory

- La clase cliente ("Client"), selecciona la clase factoría concreta en el momento de la creación.
- Sólo guarda una referencia a la clase base abstracta de *ProductA* y *ProductB*.
- De esta forma el Cliente y los productos son totalmente independientes.



Abstract Factory

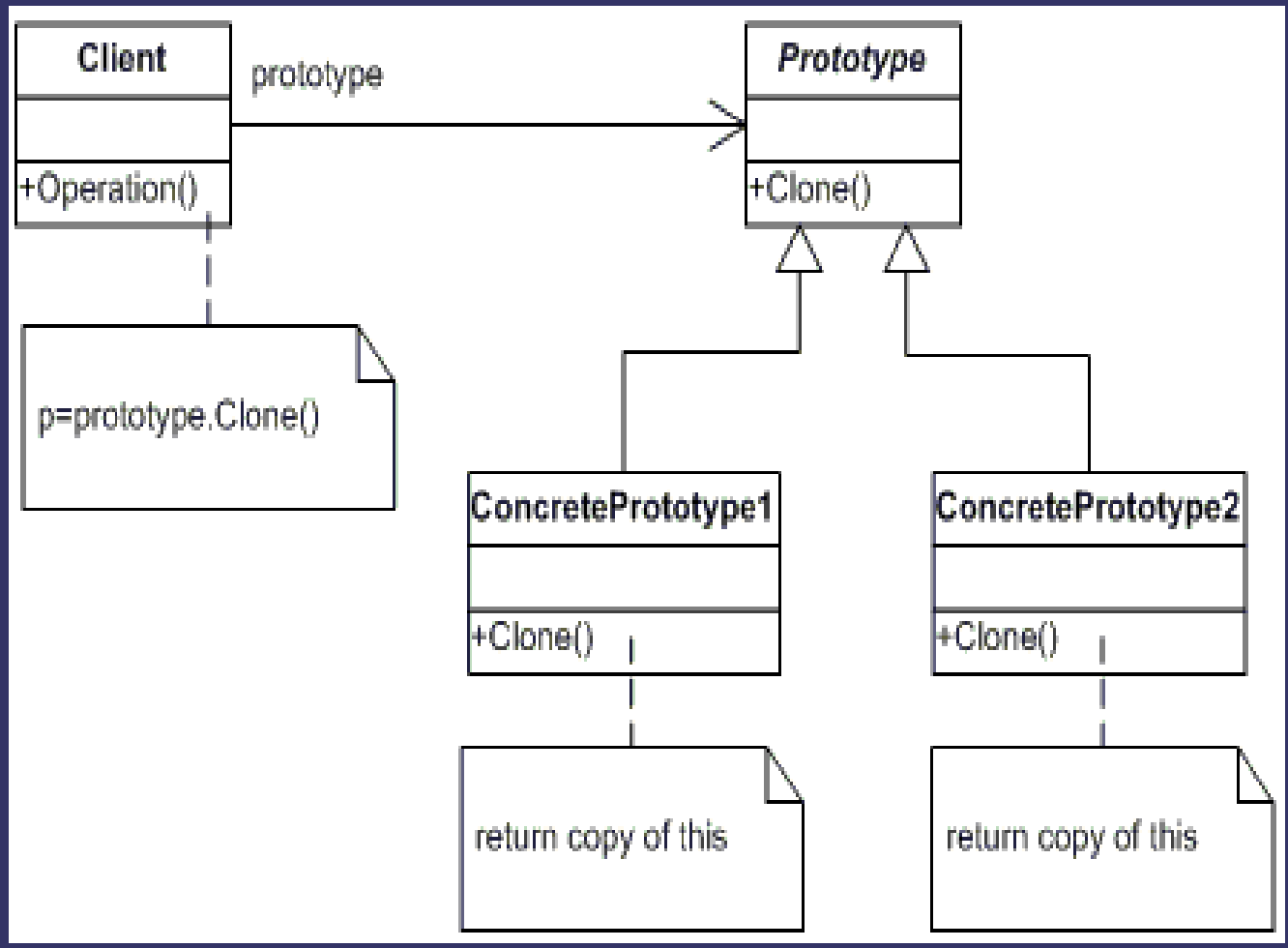
- El ejemplo típico para este patrón es el de dos entornos de ventanas distintos, con los cuales se desea ser compatible al mismo tiempo.
- Así, el uso de estos dos GUI's son intercambiables: basta seleccionar cuál de las dos jerarquías de clases se van a emplear.

Prototype

- Este patrón consiste en que, en ciertas circunstancias es posible tener una misma clase que representará distintos objetos, que sin embargo, es posible *subclasificar* de alguna forma.
- Las piezas del juego de las damas son un buen ejemplo. Cada pieza se diferencia de las otras en su posición y su color. En lugar de crear la pieza desde cero, puede ser más sencillo *clonar* un objeto **PiezaDama** con una nueva posición.

Prototype

- El cliente ("Client"), en lugar de crear objetos, clona otros ya creados.
- Nótese que `clone()` debe ser un método polimórfico, por lo que no será necesario conocer exáctamente al objeto.

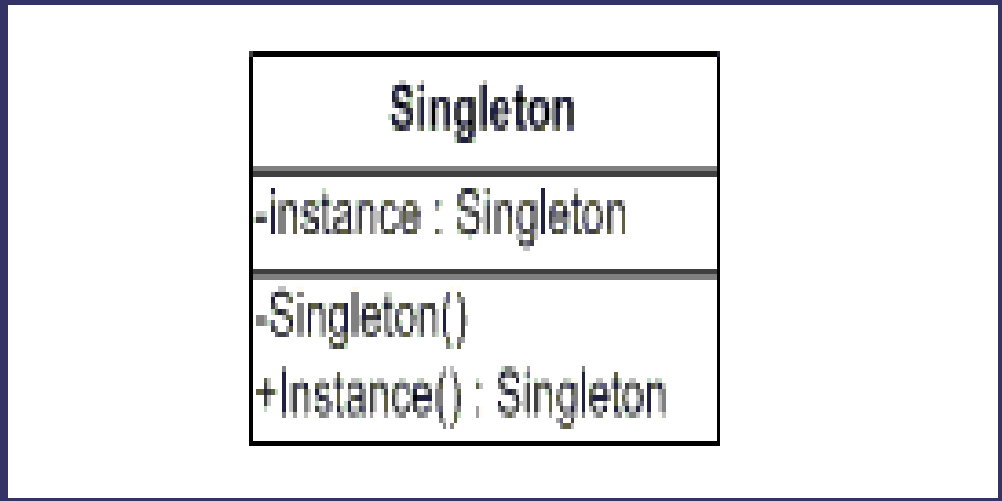


Singleton

- Este patrón asegura que existe una y sólo una instancia de una clase determinada.
- Esto es típico cuando un recurso común, como podría ser, por ejemplo, la memoria, es tratado a lo largo de todo el programa. El recurso se encapsula en una clase, pero es necesario asegurarse de que sólo habrá un objeto de esa clase.

Singleton

- Obsérvese que el constructor es privado.
- El método `instance()` es el que devuelve la única instancia de esta clase, o la crea si no existe.



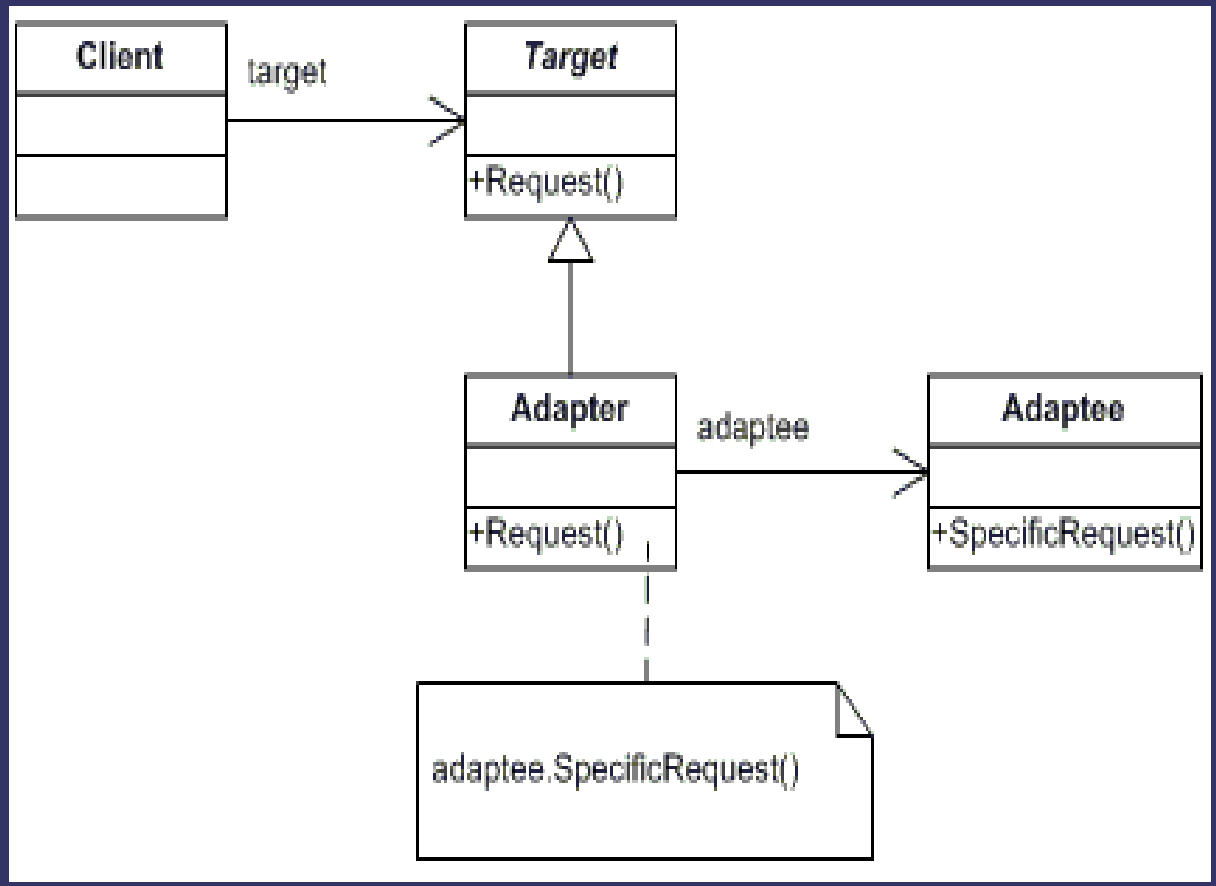
Patrones estructurales

Adapter

- Encapsula una clase, de manera que es posible normalizar su interfaz para su uso con otras clases.
- El problema suele presentarse cuando en una aplicación se reutilizan clases que no tienen exactamente la interfaz requerida.
- La clase adaptadora se interpone entre el cliente y la clase reutilizada. La clase adaptadora presenta la interfaz que espera la aplicación, y se comunica con la clase reutilizada, "*traduciendo*" los mensajes.

Adapter

- El cliente ("Client"), se comunica con la clase adaptadora que actúa de intermediaria, en lugar de con la clase que realmente proporciona la funcionalidad ("Adaptee").

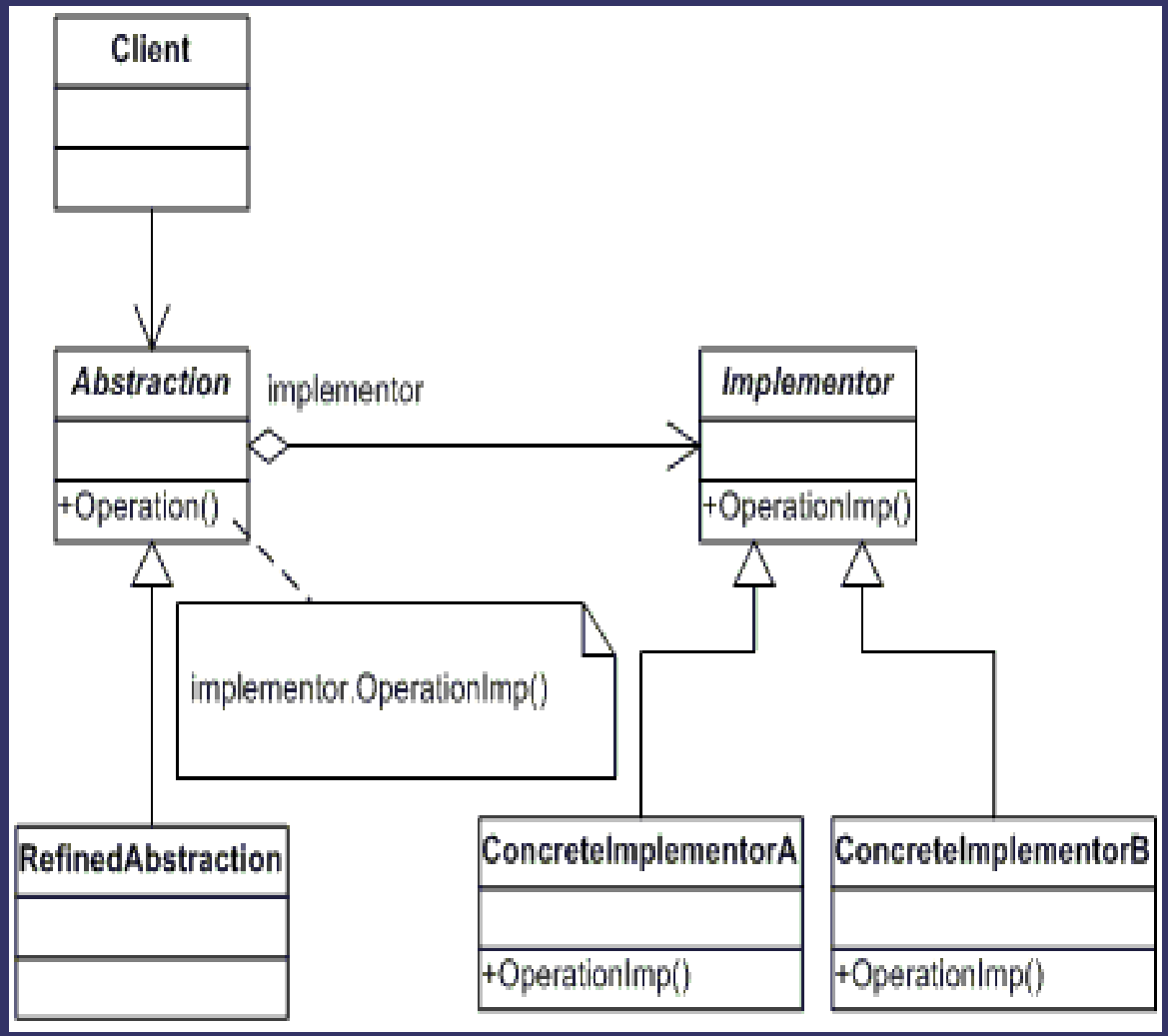


Bridge

- El puente permite separar dos aspectos distintos de una jerarquía de clases.
- Básicamente, mientras una jerarquía implementa la funcionalidad real, la(s) otra(s) representan diversos aspectos, como representación en diferentes medios ... etc.

Bridge

- En el ejemplo, existen varias implementaciones distintas para un método dado (ConcreteImplementorA y ConcreteImplementorB).
- Se trata de separar diferentes *aspectos*.



Bridge

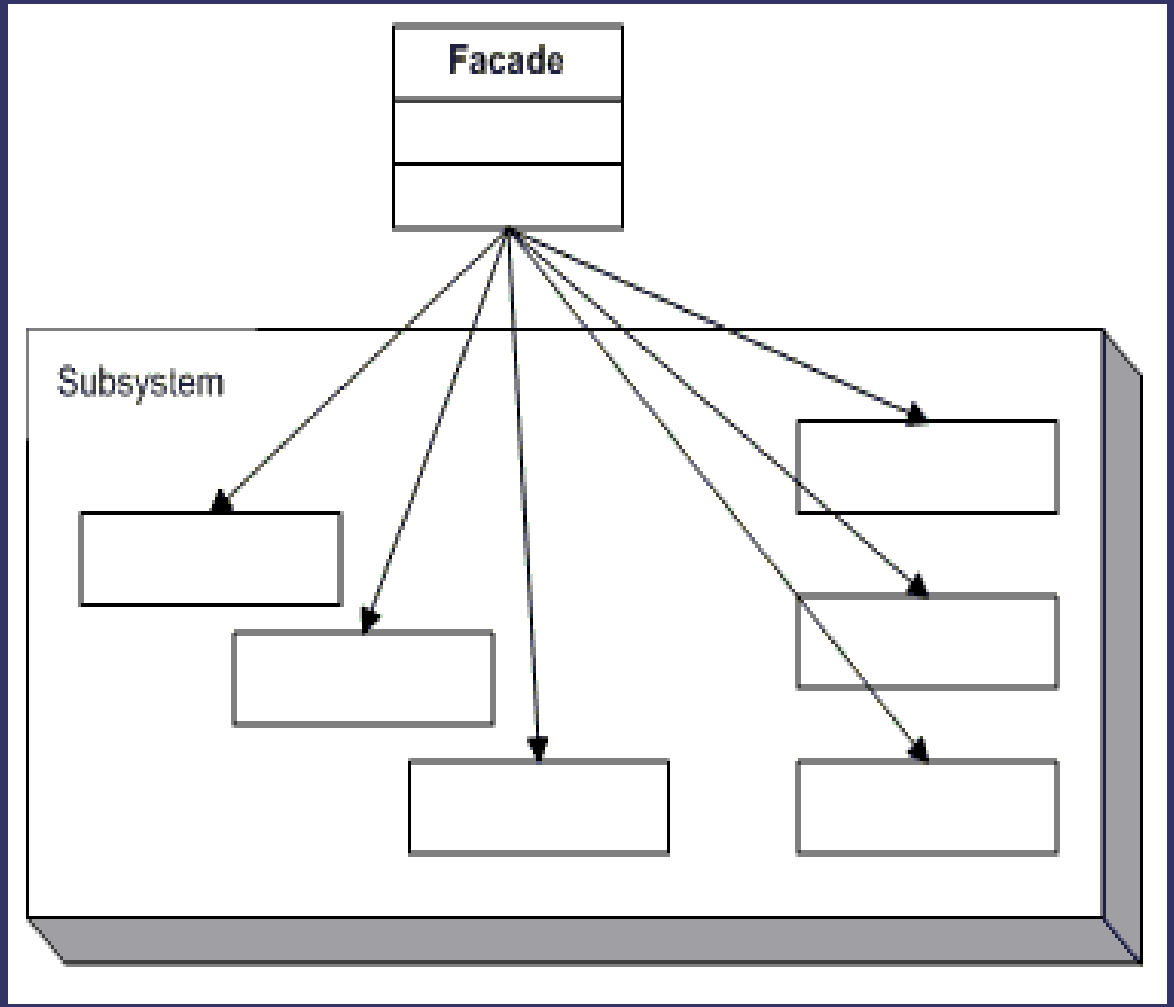
- Un ejemplo bastante típico sería el de separar el aspecto de persistencia de una jerarquía de clases.
- Mientras en la propia jerarquía sólo se implementa la funcionalidad de estas clases, en otra, relacionada, como ya hemos visto, se implementa su representación en disco.
- En algún momento será necesario *enlazar* la clase adecuada, de manera que será necesario crear la clase de persistencia adecuada para cada una de las case de funcionalidad.

Facade

- Típicamente, un diseño adecuado nos llevará a una factorización en clases más allá de la estrictamente necesaria para una sistema determinado.
- En ese caso, es posible construir una clase *facade*, (fachada), que tenga la interfaz esperada y sea el que realmente se comunique con la estructura de clases real, cuyas interfaces pueden variar.

Facade

- La clase fachada se interpone entre el cliente (no mostrado) y el subsistema que proporcione la funcionalidad deseada.
- *Facade* simplifica la interfaz del subsistema.



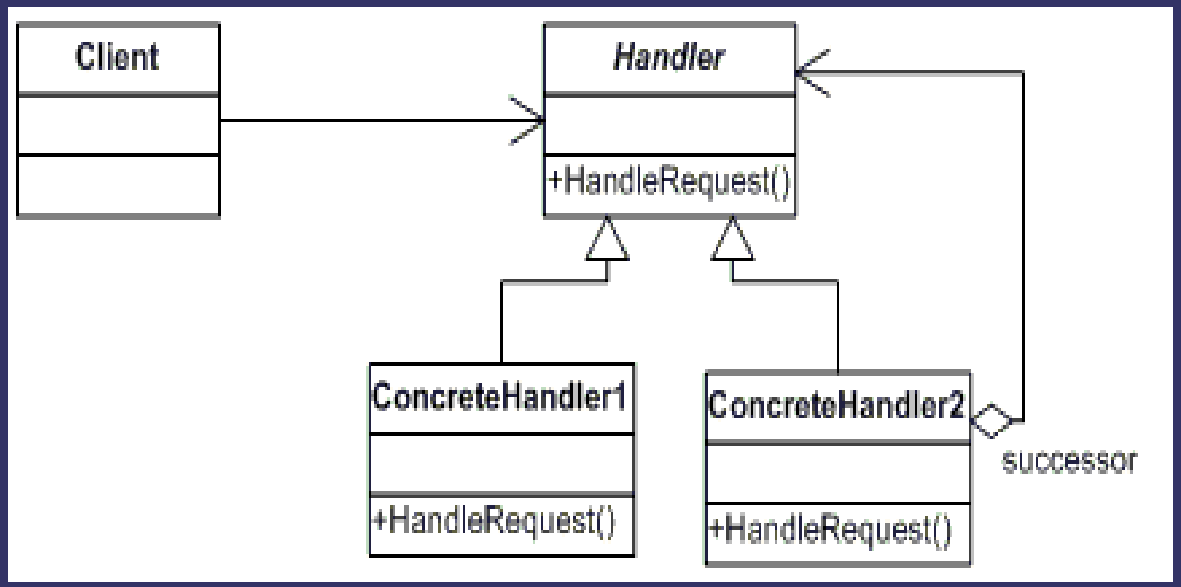
Patrones de comportamiento

Chain of responsibility

- En múltiples ocasiones, un cliente necesita que se realice una función, pero o no conoce al servidor concreto de esa función o es conveniente que no lo conozca para evitar un gran acoplamiento entre cliente y servidor.
- La petición se lanza a una cadena de objetos que se "la van pasando" hasta que uno de los objetos la maneja.

Chain of responsibility

- El cliente ("Client"), conoce a un gestor que es el que lanza la petición a la cadena hasta que alguien la recoge.



Chain of responsibility

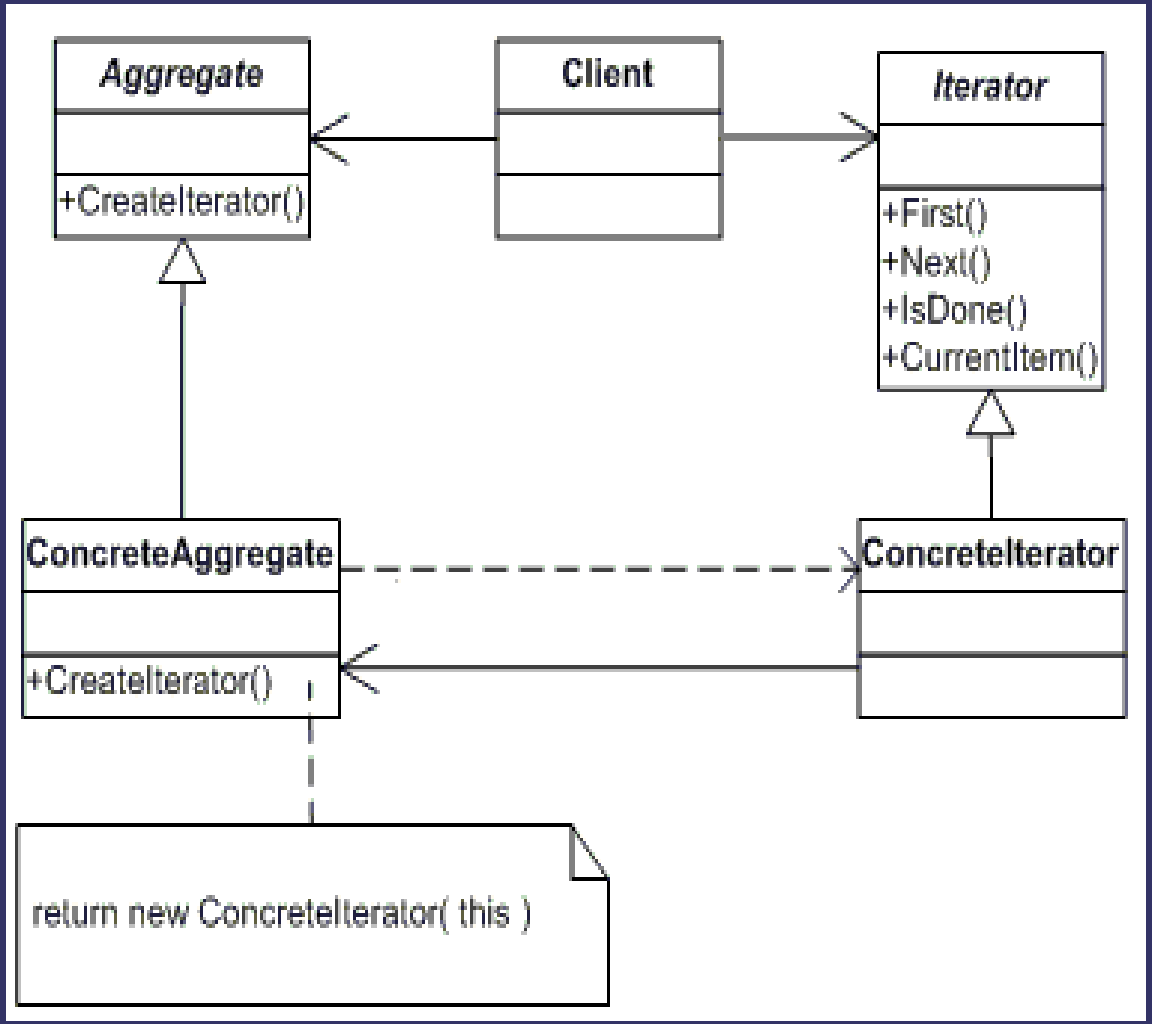
- Un ejemplo típico podría ser el lanzar un trabajo de impresión. El cliente no sabe siquiera qué impresoras están instaladas en el sistema, símplemente lanza el trabajo a la cadena de objetos que representan a las impresoras. Cada uno de ellos lo deja pasar, hasta que alguno, finalmente lo ejecuta.
- Hay un desacoplamiento evidente entre el objeto que lanza el trabajo (el cliente) y el que lo realiza (servidor).

Iterator

- Es un patrón ampliamante utilizado.
- Se trata de tener varias colecciones de objetos (contenedores como lista, vector ...), que pueden ser recorridas utilizando la misma abstracción: el *iterador*.
- Permite recorrer cada elemento de un contenedor secuencialmente.
- Es ampliamente utilizado en la STL de C++.

Iterator

- El cliente ("Client"), puede incluso utilizar un iterador sin saber sobre qué colección concreta se está ejecutando.



Observer

- Se trata de poder notificar de un cambio a varios objetos.
- Al fin y al cabo, es una manera de establecer una relación de uno a muchos.
- Un ejemplo claro puede encontrarse cuando, teniendo abiertas dos ventanas de exploración sobre un mismo directorio, desde una de ellas borramos un archivo, y la segunda se actualiza automáticamente. Podrían definirse ambos exploradores como dos *observadores* del sistema de ficheros.

Observer

- En cada observador puede tomarse una acción distinta según se haya producido un cambio.