El Proceso de Diseño

PATRONES DE DISEÑO

- Resuelven problemas específicos de diseño
- Refinar Componentes o Subsistemas
- Descripciones de las comunicaciones de objetos y clases que son personalizadas para resolver un problema general de diseño en un contexto particular

GoF

- Erich Gamma, Richard Helm, Ralph Johnson y John Vlissides

- 23 Patrones

- Agrupados:
  - Patrones de creación: Guía de cómo crear objetos cuando sus creaciones requieren tomar decisiones
  - Patrones Estructurales: Formas comunes en que diferentes tipos de objetos pueden ser organizados.
  - Patrones Comportamiento: organizar, manejar y combinar comportamientos

PATRONES DE CREACIÓN

- Abstract Factory
- Builder
- Prototype
- Singleton
- Factory Method

PATRONES ESTRUCTURALES

- Adapter
- Bridge
- Composite
- Decorator
- Facade
- Proxy

PATRONES COMPORTAMIENTO

- Interpreter
- Template Method
- Chain of
- Responsibility
- Command
- Iterator
- Mediator
- Memento
- Flyweight
- Observer
- State
- Strategy
- Visitor

FACTORY

- Simplificación del Abstract Factory
- Define un objeto de fabricación Pura “factory” para crear los objetos

Problema	Quién debe ser el responsable de la creación de los objetos cuando existen consideraciones especiales, como una lógica de creación compleja, el deseo de separar las responsabilidades de la creación para mejorar la cohesión?
Solución	Crear un objeto Fabricación Pura denominado Factoría que maneje la creación

Factory Simple

```
private Pizza ordenarPizza()
{
    Pizza pizza = new Pizza();
    pizza.preparar();
    pizza.cocinar();
    pizza.cortar();
    pizza.empaquetar();
    return pizza;
}
```

```
private Pizza ordenarPizza(string p_tipo)
 {
     Pizza pizza = new Pizza();

     switch (p_tipo)
     {
         case "queso":
             pizza = new PizzaQueso();
             break;
         case "griega":
             pizza = new PizzaGriega();
             break;
         case "pepperoni":
             pizza = new PizzaPepperoni();
             break;
         default:
             break;
     }
     pizza.preparar();
     pizza.cocinar();
     pizza.cortar();
     pizza.empaquetar();
     return pizza;
 }
```

Factory Simple

```
private Pizza ordenarPizza(string p_tipo)
 {
     Pizza pizza = new Pizza();

     switch (p_tipo)
     {
         case "queso":
             pizza = new PizzaQueso();
             break;
         case "griega":
             pizza = new PizzaGriega();
             break;
         case "pepperoni":
             pizza = new PizzaPepperoni();
             break;
         default:
             break;
     }
     pizza.preparar();
     pizza.cocinar();
     pizza.cortar();
     pizza.empaquetar();
     return pizza;
 }
```

```
private Pizza ordenarPizza(string p_tipo)
{
    Pizza pizza = new Pizza();

    switch (p_tipo)
    {
        case "queso":
            pizza = new PizzaQueso();
            break;
        case "vegetariana":
            pizza = new PizzaVegetariana();
            break;
        case "griega":
            pizza = new PizzaGriega();
            break;
        case "pepperoni":
            pizza = new PizzaPepperoni();
            break;
        default:
            break;
    }
    pizza.preparar();
    pizza.cocinar();
    pizza.cortar();
    pizza.empaquetar();
    return pizza;
}
```

Crear un elemento Factory

```
private Pizza ordenarPizza(string p_tipo)
{
    Pizza pizza = new Pizza();

    switch (p_tipo)
    {
        case "queso":
            pizza = new PizzaQueso();
            break;
        case "vegetariana":
            pizza = new PizzaVegetariana();
            break;
        case "griega":
            pizza = new PizzaGriega();
            break;
        case "pepperoni":
            pizza = new PizzaPepperoni();
            break;
        default:
            break;
    }
    pizza.preparar();
    pizza.cocinar();
    pizza.cortar();
    pizza.empaquetar();
    return pizza;
}
```

Cliente

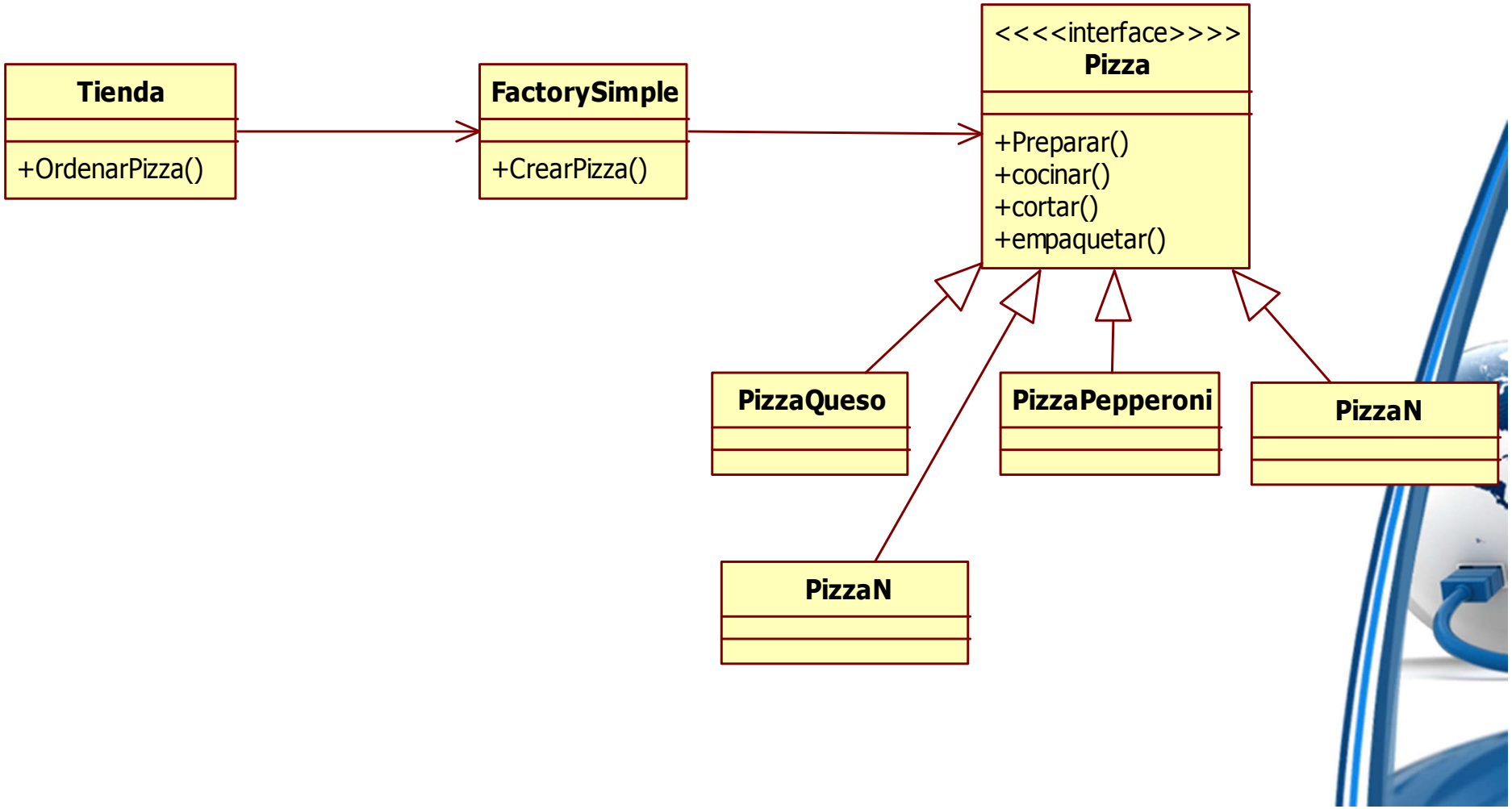
Factory

Ejemplo con el Factory

```
private static void ordenarPizza(string p_tipo) {
    Pizza pizza = null;
    FabricaSimple fabrica = new FabricaSimple();
    pizza = fabrica.ObtenerPizza(p_tipo);
    if (pizza != null) {
        pizza.preparar();
        pizza.cocinar();
        pizza.cortar();
        pizza.empaquetar();
    }
    else
        Console.WriteLine("Seleccion Invalida");
}
```

```
public class FabricaSimple {
    public Pizza ObtenerPizza(string p_tipo) {
        Pizza pizza=null;
        switch (p_tipo) {
            case "queso":
                pizza = new PizzaQueso();
                break;
            case "vegetariana":
                pizza = new PizzaVegetariana();
                break;
            case "pepperoni":
                pizza = new PizzaPepperoni();
                break;
            default:
                break;
        }
        return pizza;
    }
}
```

Estructura Factory Simple



Creación: Abstract Factory

"Proveer una interface para la creación de familias de objetos relacionados o dependientes sin especificar la clase concreta."[GoF].

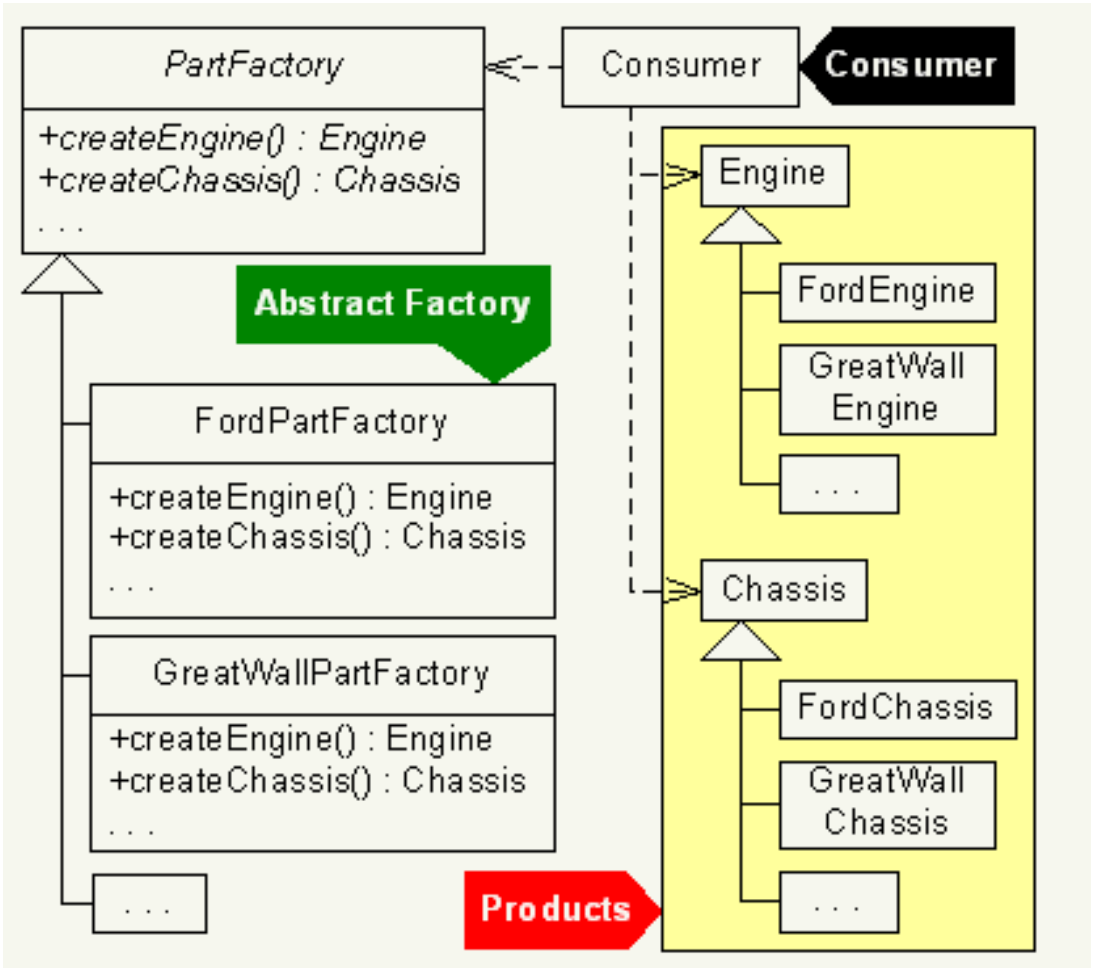
- El patrón aísla la definición del producto y sus nombres de clase del cliente el cual solo puede obtener uno a través de la fábrica

Creación: Abstract Factory

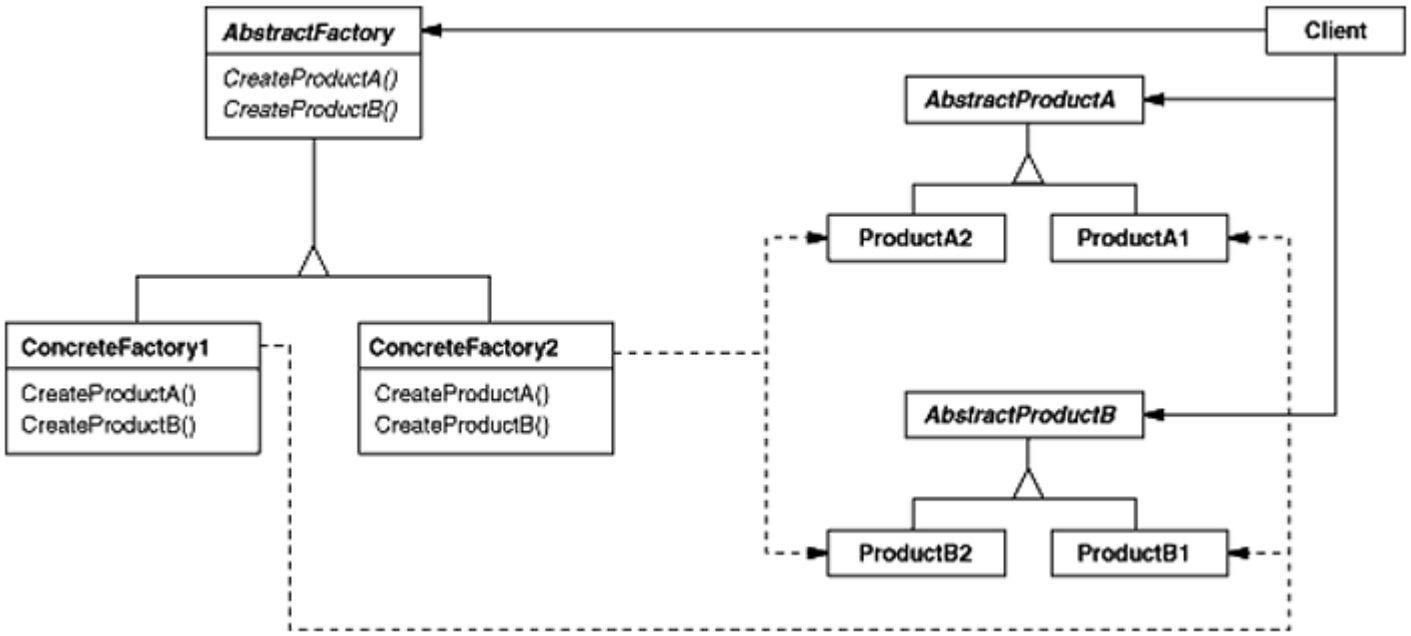
Comúnmente utilizado cuando:

- El sistema debe ser independiente de como sus objetos son creados, compuestos y representados
- Se tiene una familia de productos relacionados diseñados para ser usados en conjunto y se desea reforzar esa restricción
- Se quiere proveer una librería de clase y solo se quieren revelar sus interfaces y no sus implementaciones

Creación: Ejemplo Abstract Factory



Abstract Factory

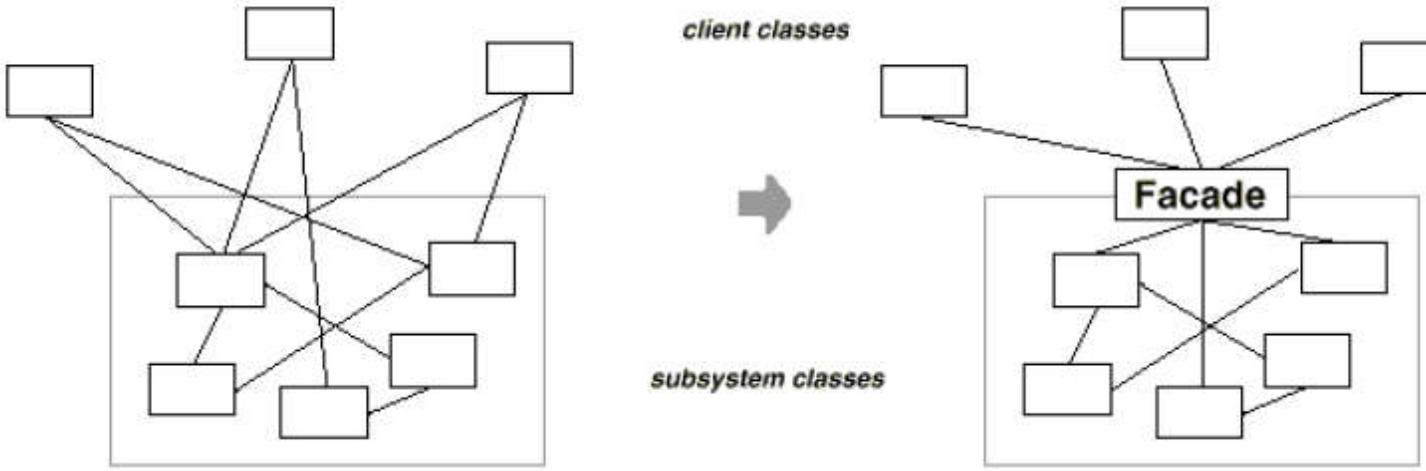


Estructurales: Fachada (Façade)

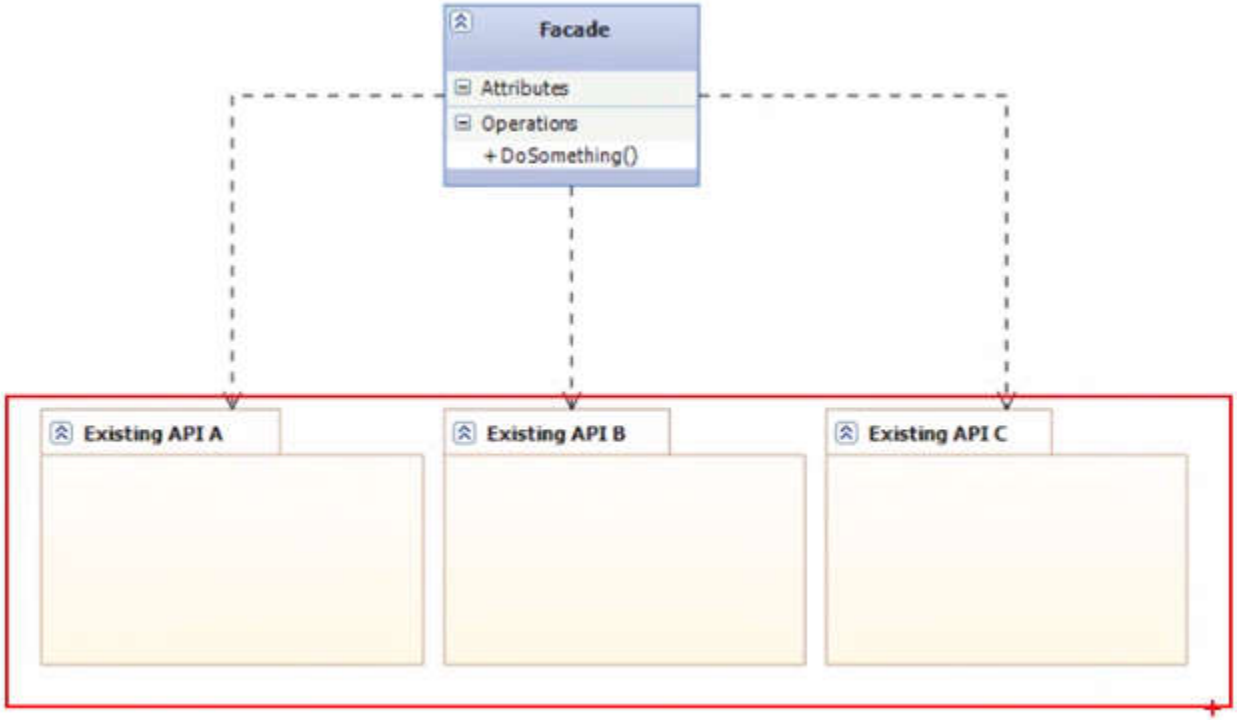
Problema	Se requiere una interfaz común, unificada para un conjunto de implementaciones o interfaces dispares. Podría no ser conveniente acoplarla con muchas cosas del subsistema o la implementación del subsistema podría cambiar. Que hacemos?
Solución	Definir un único punto de conexión con el subsistema. Este objeto fachada presenta una única interfaz unificada y es responsable de colaborar con los componentes del subsistema

Fachada

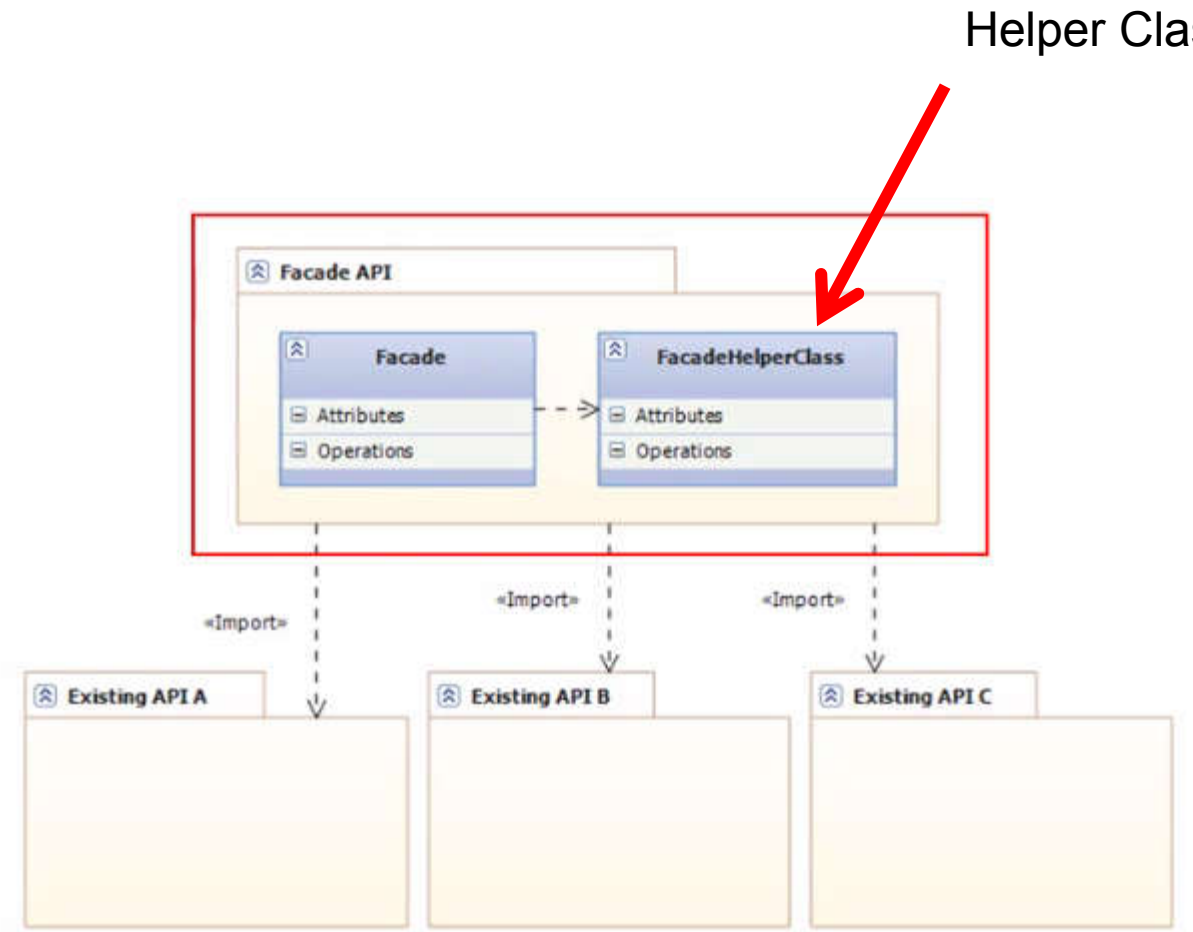
- Proveer una interfaz unificada a un conjunto de interfaces en un subsistema.
- Define una interfaz de alto nivel que hace fácil de usar a un subsistema
- Útil cuando existen muchas dependencias entre los clientes y las clases que poseen la implementación



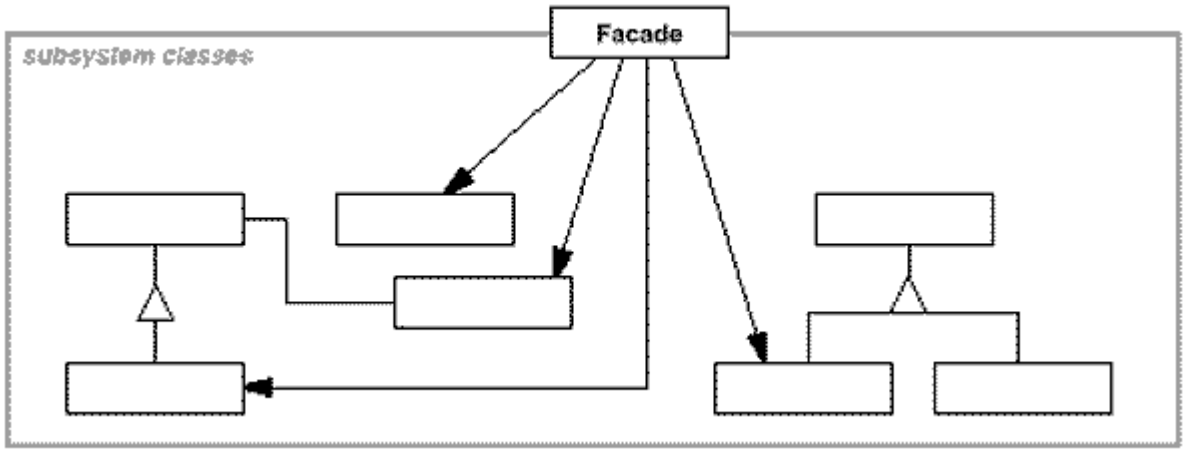
Fachada


Facade
Attributes
Operations
+DoSomething()
Existing API A
Existing API B
Existing API C

Fachada

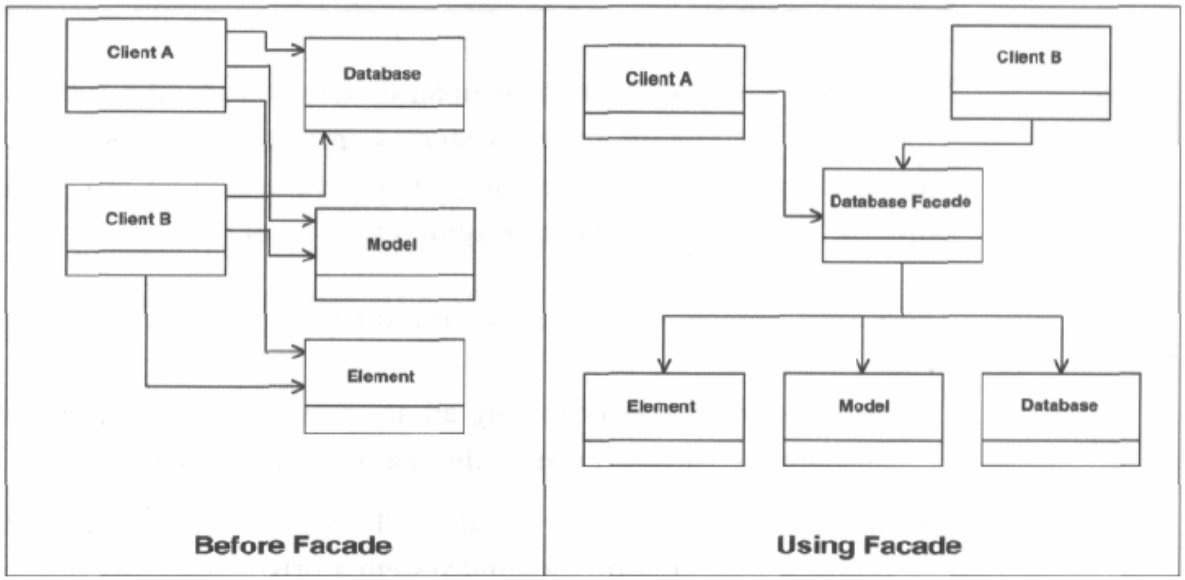

Helper Class
Facade API
Facade
Attributes
Operations
FacadeHelperClass
Attributes
Operations
«Import»
«Import»
«Import»
Existing API A
Existing API B
Existing API C

Estructura Fachada



Fachada

- Reduce el número de objetos con que un cliente debe trabajar



Estructurales: Adaptador (Adapter)

Problema	Cómo resolver interfaces incompatibles o proporcionar una interfaz estable para componentes parecidos con diferentes interfaces?
Solución	Convertir la interfaz original de un componente en otra interfaz, mediante un objeto adaptador intrmedio

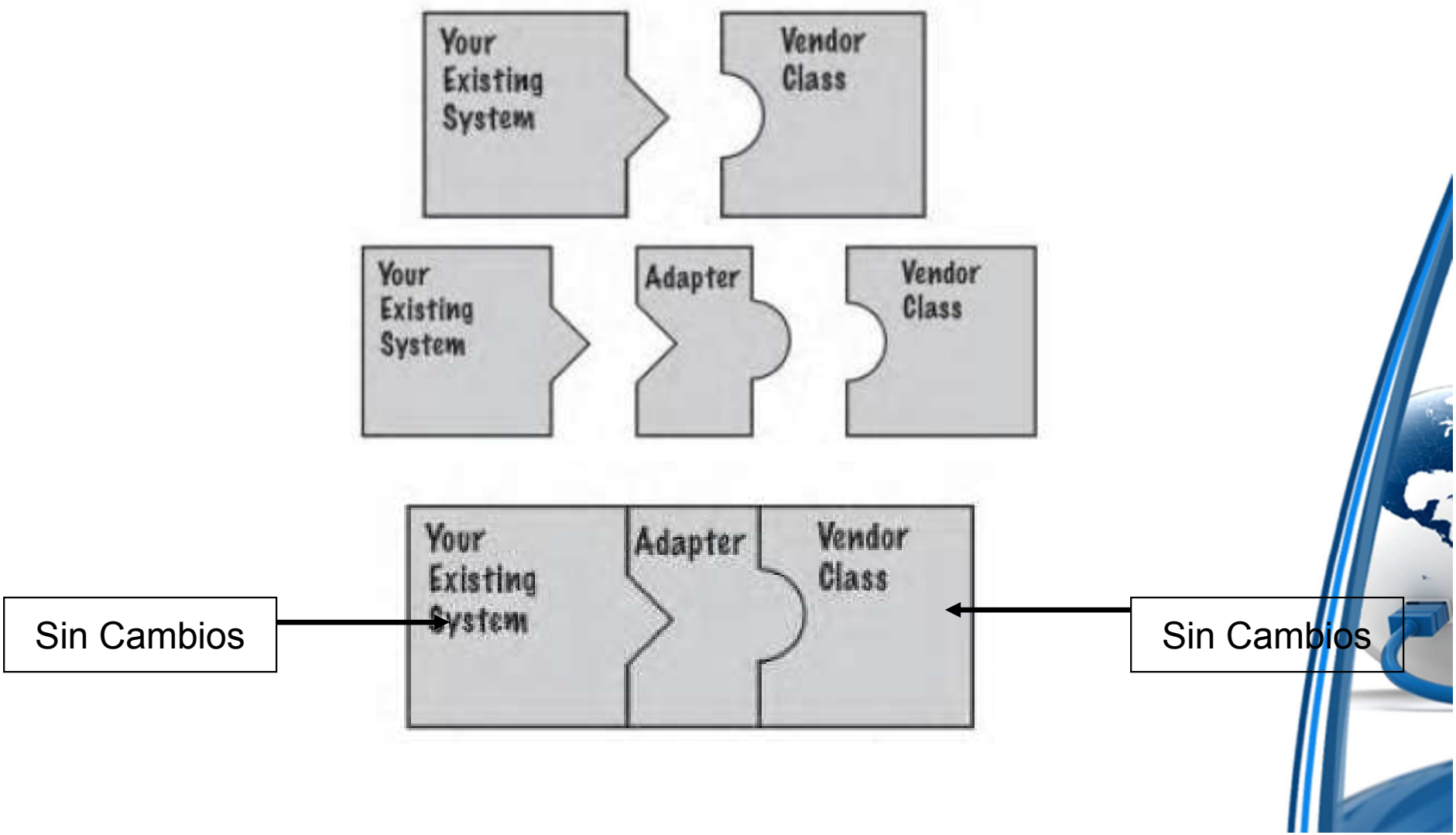
Estructurales: Adaptador (Adapter)

- A class that would be useful to your application does not implement the interface you require
- You are designing a class or a framework and you want to ensure it is usable by a wide variety of as-yet-unwritten classes and applications
- Adapters are also commonly known as *Wrappers*
- In this module, we will refer only to *object adapters*, which do not require multiple inheritance (as *class adapters* do)

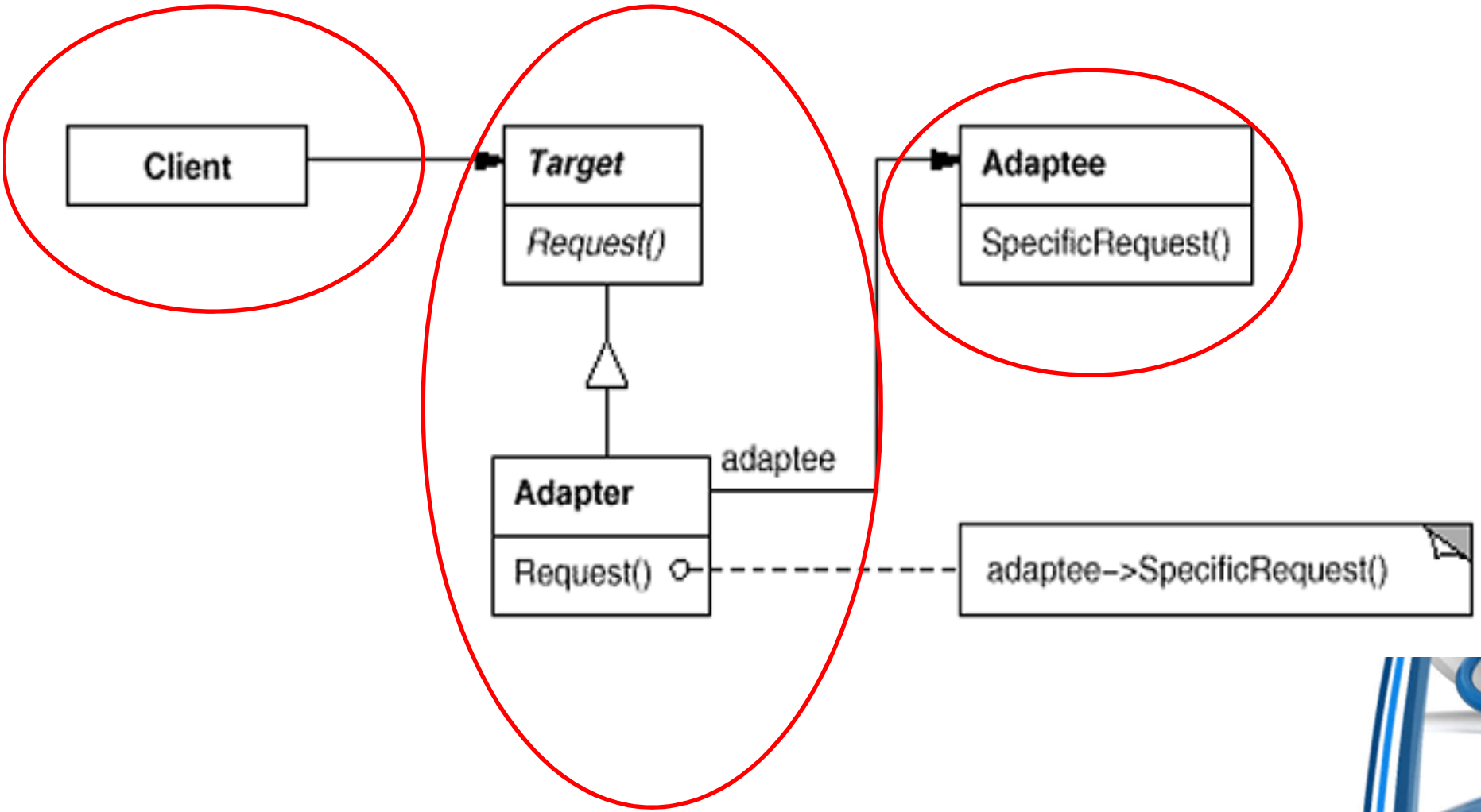
Estructurales: Adaptador (Adapter)

- Convert the interface of a class into another interface clients expect.
- Allow classes to work together that couldn't otherwise due to incompatible interfaces.
- *Future-proof* client implementations by having them depend on Adapter interfaces, rather than concrete classes directly

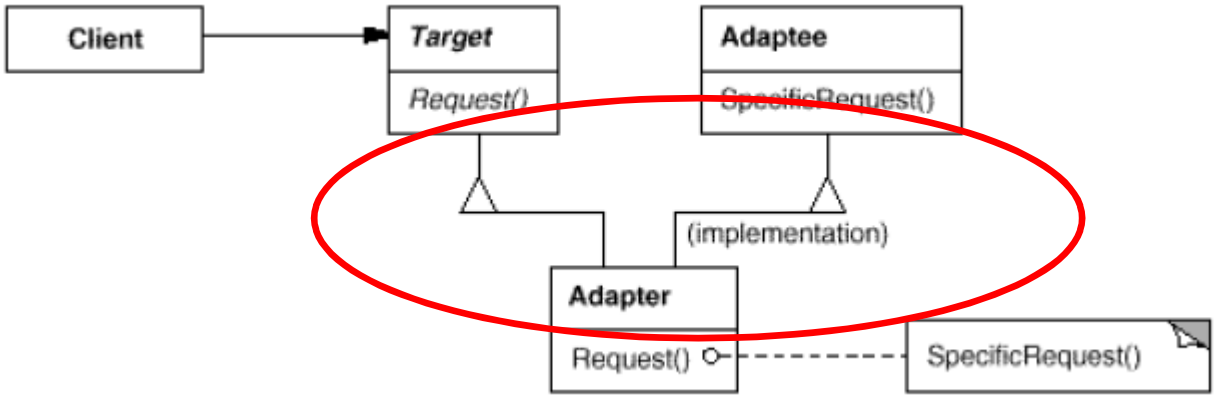
Adaptador


Your Existing System
Vendor Class
Your Existing System
Adapter
Vendor Class
Your Existing System
Adapter
Vendor Class
Sin Cambios
Sin Cambios

Adaptador


Client
Target
Request()
Adaptee
SpecificRequest()
adaptee
Adapter
Request()
adaptee->SpecificRequest()

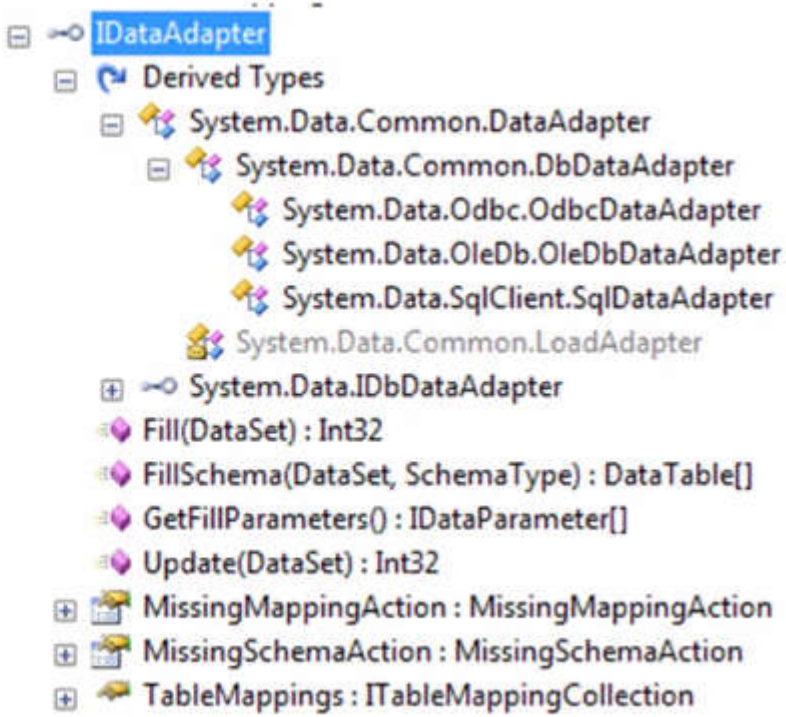
Adaptador


Client
Target
Request()
Adaptee
SpecificRequest()
(implementation)
Adapter
Request()
SpecificRequest()

Estructurales: Adaptador (Adapter)

ADO.NET

- IDataAdapter
  - DbDataAdapter
    - OdbcDataAdapter
    - OleDbDataAdapter
    - SqlClientDataAdapter



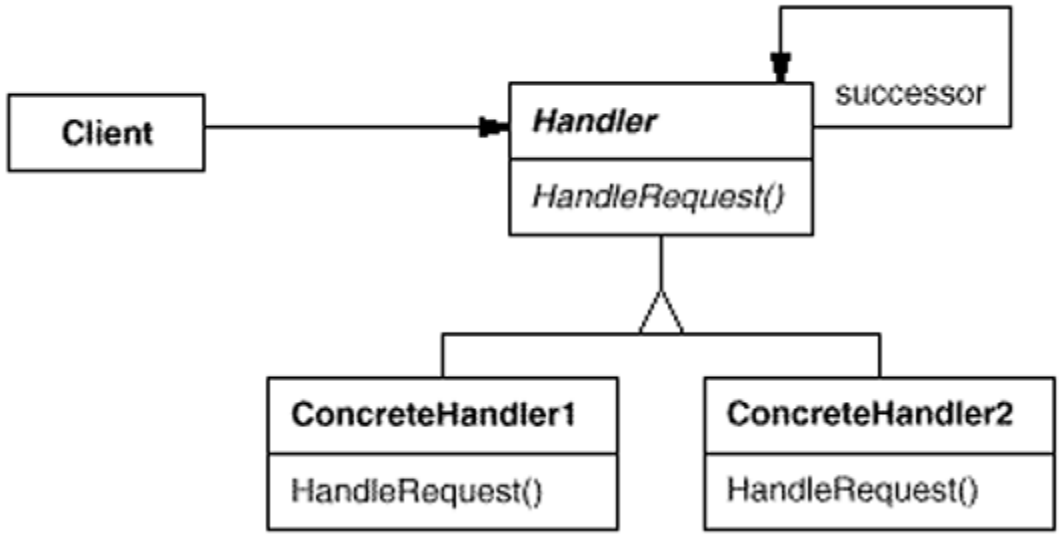
Comportamiento: **Chain of Responsibility**

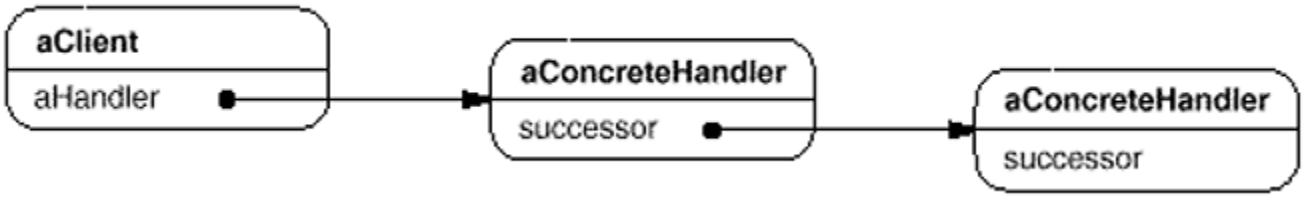
Problema	Cómo evitar acoplar el emisor de un request al receptor dándole a más de un objeto la oportunidad de manejar el request?
Solución	Una cadena de objetos receptores y pasar el request a lo largo de la cadena hasta el objeto que lo maneja

Comportamiento: **Chain of Responsibility**

- El primer objeto en la cadena recibe el request y lo maneja o lo pasa al próximo candidato. El objeto que emite el request no tiene ningún conocimiento explicito de quien manejara la solicitud.
- Útil cuando:
  - Más de un objeto puede manejar la solicitud y el manejador no es conocido de antemano
  - Se desea enviar una solicitud a uno o más objetos sin especificar explícitamente quién es el receptor
  - El set de objetos que pueden manejar la solicitud debe ser especificada de manera dinámica

Comportamiento: **Chain of Responsibility**





Factory Method

"Definir una interface para la creación de objetos, pero dejando a la clase que implementa dicha interface decidir que objeto instanciar"[GoF].

- Utilizado cuando:
  - Una clase no puede anticipar la clase de objeto que debe crear
  - Una clase quiere que sus subclases especifiquen el objeto a crear

Ejemplo Factory Method

- Suite de aplicaciones
- Cada aplicación tiene un tipo de documento específico

Se determina que mucho del código utilizado puede ser colocado en una clase padre genérica y así evitar la duplicidad de funcionalidades!

- Problema: Se necesita encontrar una manera de asegurar que el documento correcto sea asociado con cada tipo de aplicación

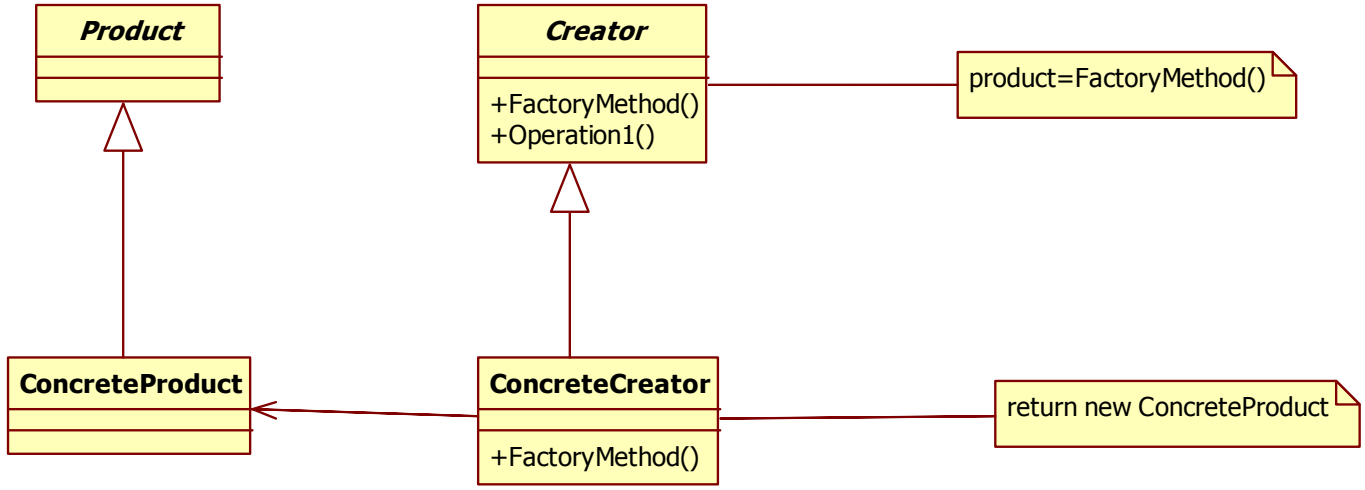
Ejemplo Factory Method

- Como el documento particular es específico de la aplicación la clase padre genérica no puede predecir el tipo de documento a instanciar
- La clase padre sólo sabe cuando debe crearlo no el tipo especifico
- El Factory Method encapsula el conocimiento de que tipo de documento crear y lo mueve fuera de la clase padre
- Se define una operación abstract CrearDocumento que retorna el documento apropiado para la subclase. El método CrearDocumento es un Factory Method por que es responsable de la manufactura del objeto

Ejemplo Factory Method

```
abstract public class Application {
        private Document document;
    public Application() {
        initializeApplication();
    }
    public void initializeApplication() {
        setDocument(createDocument());
    }
    abstract protected Document createDocument();
    public void setDocument(Document document) {
        this.document = document;
    }
    public Document getDocument() {
        return document;
    }
}
```

Estructura Factory Method



Estrategia (Strategy)

- **Define una familia de algoritmos, encapsula cada una y la hace intercambiable.**

- **El patron Strategy deja que el algoritmo varíe independientemente del cliente que lo usa**

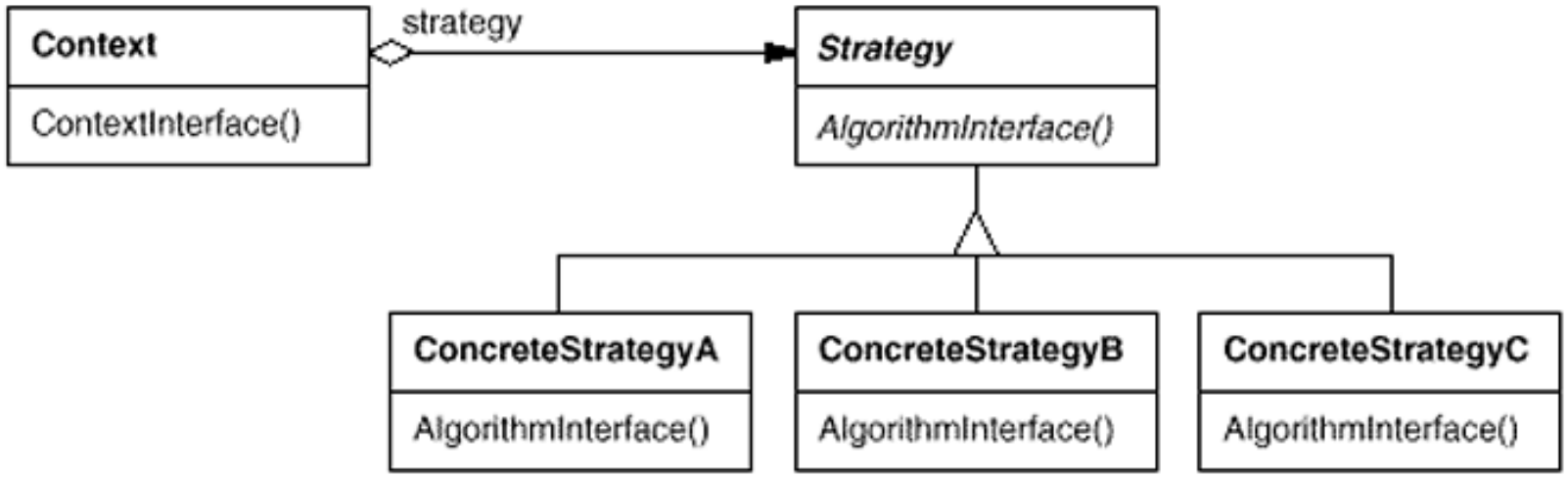
Patron Strategy

- Útil cuando se tienen muchas clases que difieren solo en su comportamiento. Strategy provee una forma de configurar una clase con muchos comportamientos

- Una clase que define muchos comportamientos y se muestra como múltiples declaraciones de condicionales en sus operaciones. En lugar de tener los condicionales mueva la lógica a su propia clase de estrategia

- Se necesitan multiples variantes de un algoritmo

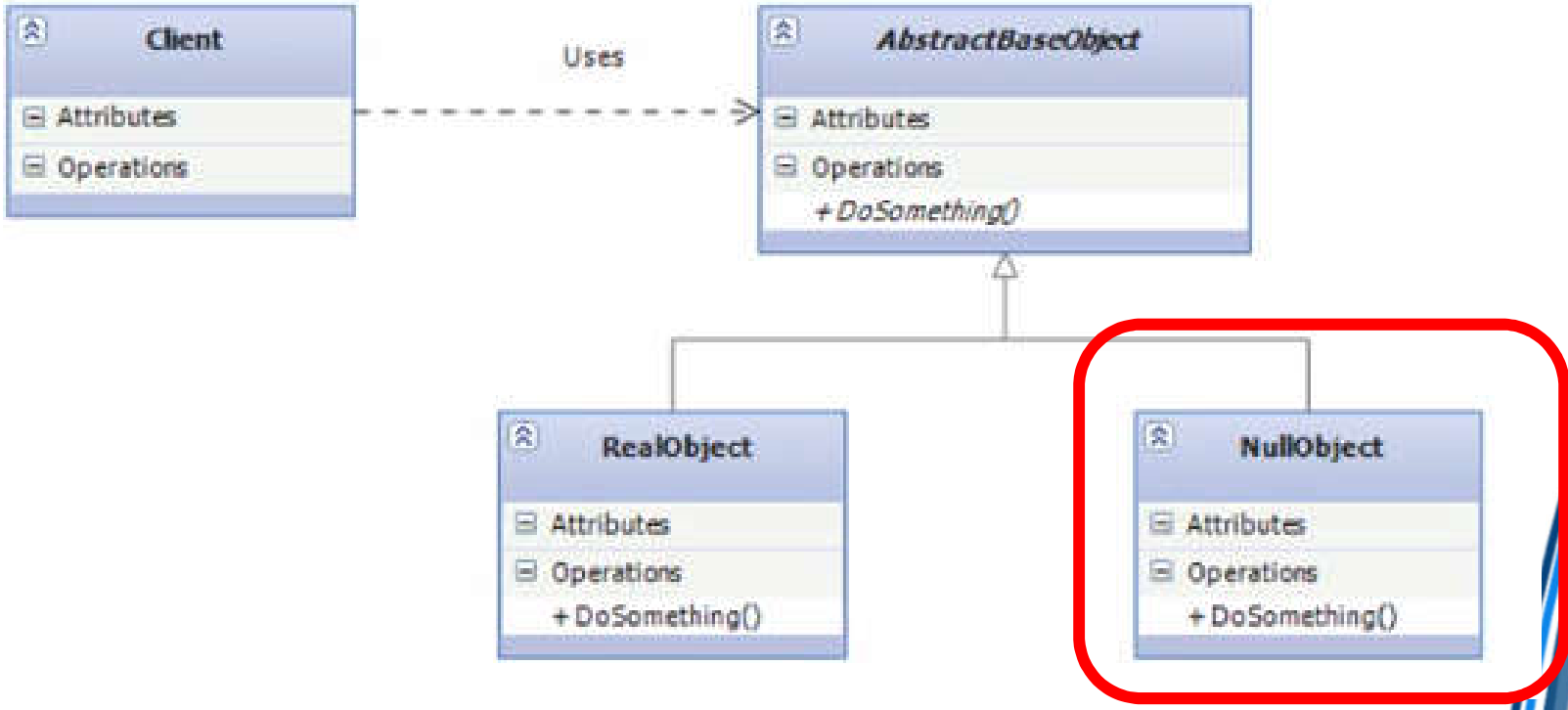
Estructura Strategy



Patrón: **Null Object**

- El rol del objeto nulo es remplazar la referencia nula, implementando la misma interface de la clase objetivo pero no su comportamiento
- El objeto nulo compartirá la misma interface o heredará de la misma clase base que la utilizada por el resultado esperado, esto busca aliviar la preocupación de tener que estar validando por casos de soluciones nulas
- Permite en cierta medida, eliminar de nuestro código las constantes validaciones de objetos nulos poniendo en un objeto no funcional la referencia nula

Patrón: **Null Object**



Creación: Singleton

Problema	Se admite exactamente una instancia de una clase (singleton). Los objetos necesitan un único punto de acceso global
Solución	Definir un método estático de la clase que devuelva el singleton

- Otorga Visibilidad global o un único punto de acceso a una única instancia de una clase
- Participan: la clase singleton, y la instancia cliente

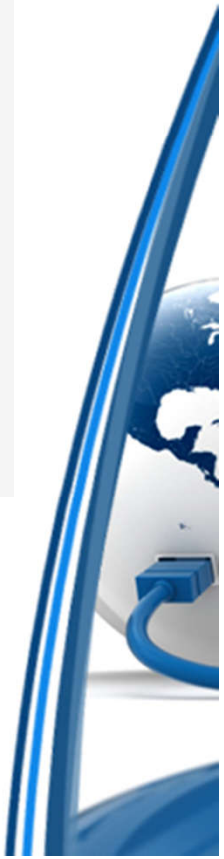
Ejemplo

```
public class ClassicSingleton {
   private static ClassicSingleton instance = null;
   protected ClassicSingleton() {
      // Exists only to defeat instantiation.
   }
   public static ClassicSingleton getInstance() {
      if(instance == null) {
         instance = new ClassicSingleton();
      }
      return instance;
   }
}
```

Thread-safe Singleton

La sincronización evita el problema de concurrencia, no obstante se necesita únicamente la primera vez que se llama la instancia

```
public synchronized static Singleton getInstance() {
    if(singleton == null) {
        simulateRandomActivity();
        singleton = new Singleton();
    }
    logger.info("created singleton: " + singleton);
    return singleton;
}
```



Cambio para mejorar el performance

```
public static Singleton getInstance() {
    if(singleton == null) {
        synchronized(Singleton.class) {
            singleton = new Singleton();
        }
    }
    return singleton;
}
```

Double-checked Singleton

```
public static Singleton getInstance() {
  if(singleton == null) {
      synchronized(Singleton.class) {
        if(singleton == null) {
          singleton = new Singleton();
        }
      }
  }
  return singleton;
}
```

Singleton Enum

```
01. public enum SingletonEnum {
02.  INSTANCE;
03.  public void doStuff(){
04.    System.out.println("Singleton using Enum");
05.  }
06. }
07. And this can be called from clients :
08. public static void main(String[] args) {
09.    SingletonEnum.INSTANCE.doStuff();
10.
11.   }
```