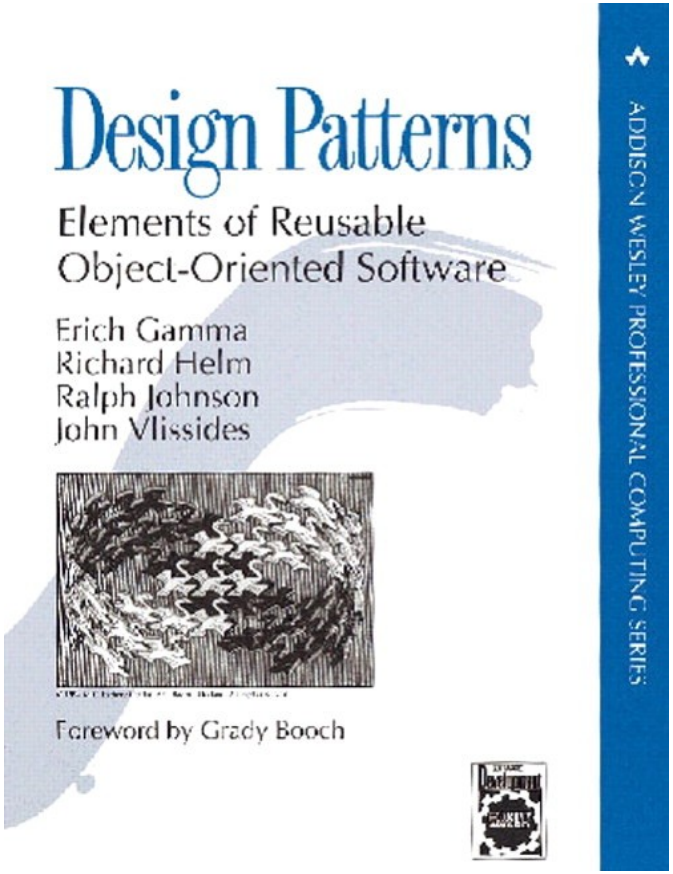
Tema 7.B. Patrones de Diseño en Java

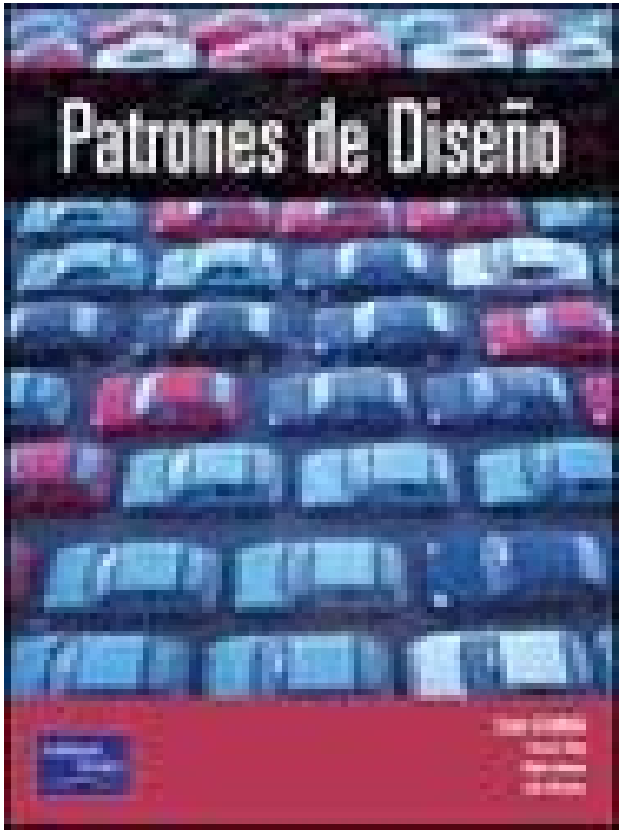
Metodología y Desarrollo de Programas

Texto básico



Design Patterns

E. Gamma, R. Helm, T. Johnson, J. Vlissides
Addison-Wesley, 1995



Patrones De Diseño.

E. Gamma, R. Helm, T. Johnson, J. Vlissides
Addison-Wesley, 1995
(Versión española de 2003)

- Catálogo de patrones GoF

		Propósito		
		Creación	*Estructural*	*Comportamiento*
Ámbito	*Herencia*	Factory Method	**Adapter**	Interpreter Template Method
	Compo-sición	Abstract Factory Builder Prototype Singleton	**Adapter** Bridge **Composite** Decorator **Facade** Flyweight Proxy	Chain of Responsability Command **Iterator** Mediator Memento **Observer** State **Strategy** Visitor

- Catálogo de patrones GoF

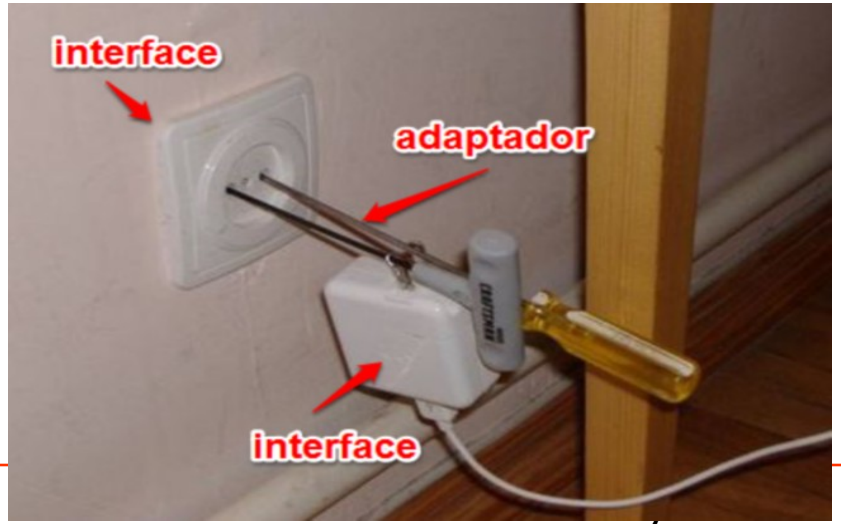
CREACIÓN	ESTRUCTURALES	COMPORTAMIENTO
Abstract Factory (O)	Adapter (C)	Chain of responsibility (O)
Builder (O)	Bridge (O)	Command (O)
Factory Method (C)	Composite (O)	Interpreter (C)
Prototype (O)	Decorator (O)	Iterator (O)
Singleton (O)	Facade (O)	Mediator (O)
	Flyweight (O)	Memento (O)
	Proxy (O)	Observer (O)
		State (O)
		Strategy (O)
		Template Method (C)
		Visitor (O)

Ámbito: (C) Clase, (O) Objeto.

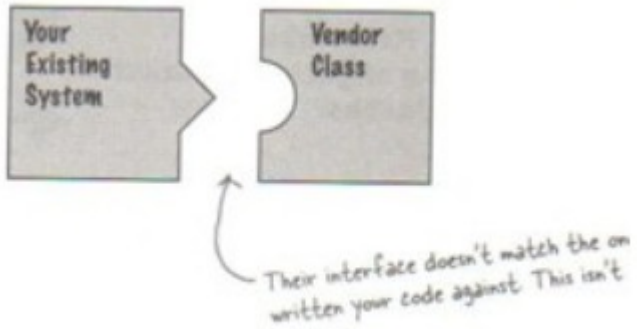
Patrones estructurales

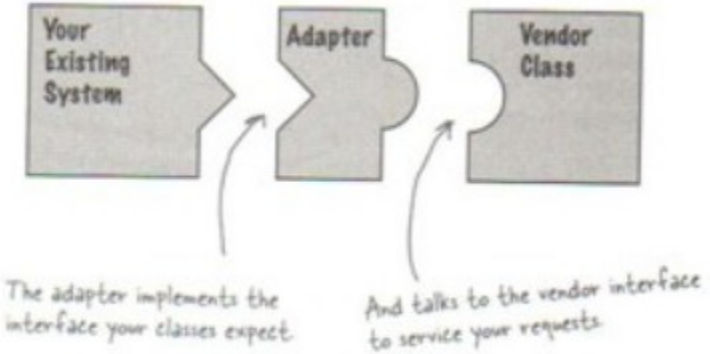
- Patrones estructurales
  - Clases y objetos que se combinan para formar estructuras más complejas.
- Tipos
  - **Adapter (Adaptador): Adapta una interfaz para que pueda ser utilizada por una clase que de otro modo no podría utilizarla.**
  - Bridge (Puente): Desacopla una abstracción de su implementación.
  - **Composite (Objeto compuesto): Permite tratar objetos compuestos como si de uno simple se tratase.**
  - Decorator (Envoltorio): Añade funcionalidad a una clase dinámicamente.
  - **Facade (Fachada): Provee de una interfaz unificada simple para acceder a una interfaz o grupo de interfaces de un subsistema.**
  - Flyweight (Peso ligero): Reduce la redundancia cuando gran cantidad de objetos poseen idéntica información.
  - Proxy: Mantiene un representante de un objeto.

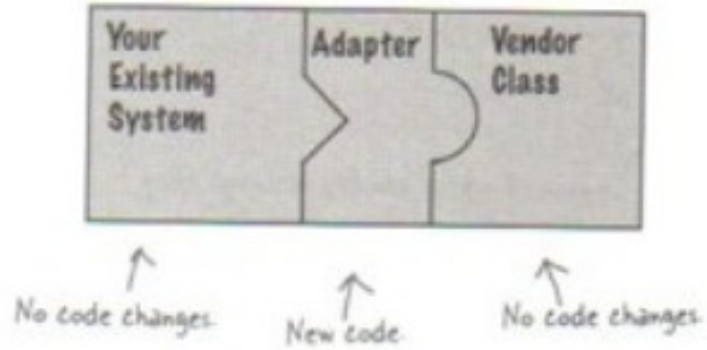
- Adapter-Adaptador (1/4)
  - Objetivo:
    - ✓ Convertir la interfaz de una clase en la interfaz que esperan los clientes (también conocido como **wrapper**)
  - Motivación
    - ✓ Cuando se necesita implementar un interface y no son necesario implementar todos sus métodos.
    - ✓ Permite que clases con interfaces incompatibles se comuniquen
    - ✓ De esta forma cuando creamos una clase abstracta (Adapter) y posteriormente heredamos de esta clase y se redefine los métodos que se necesiten.



- Adapter-Adaptador

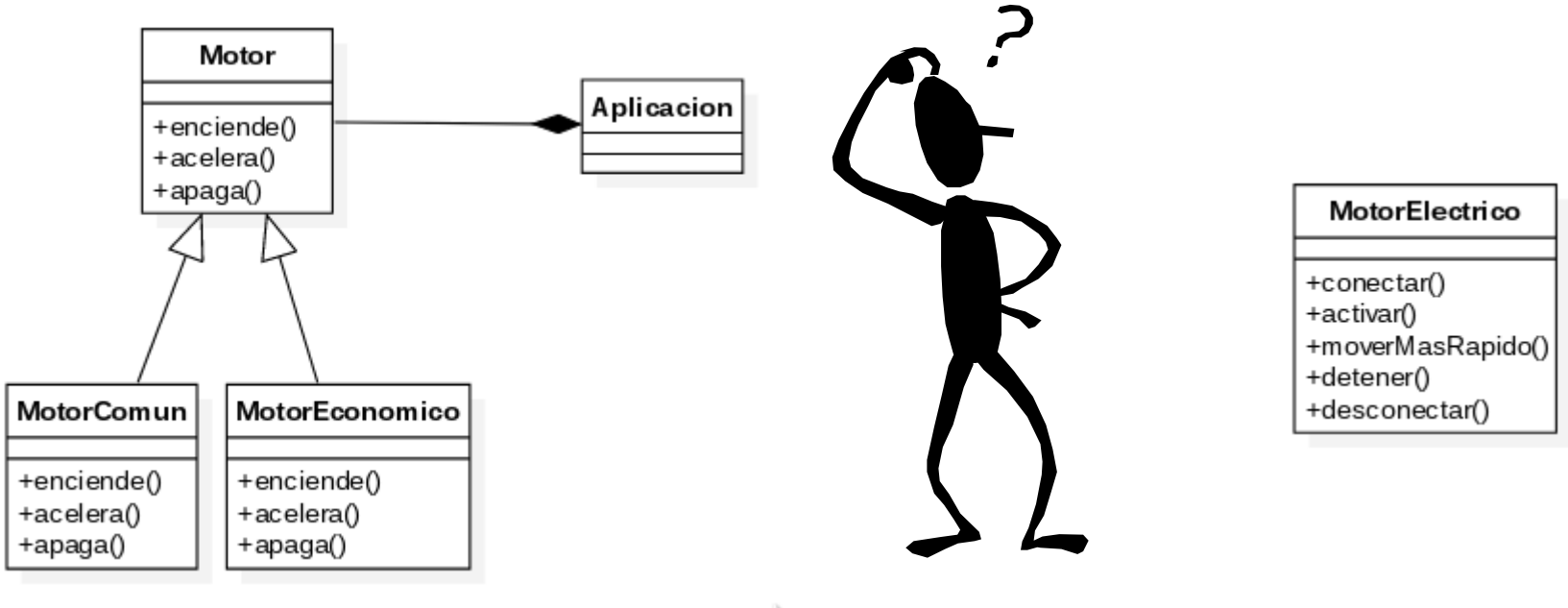




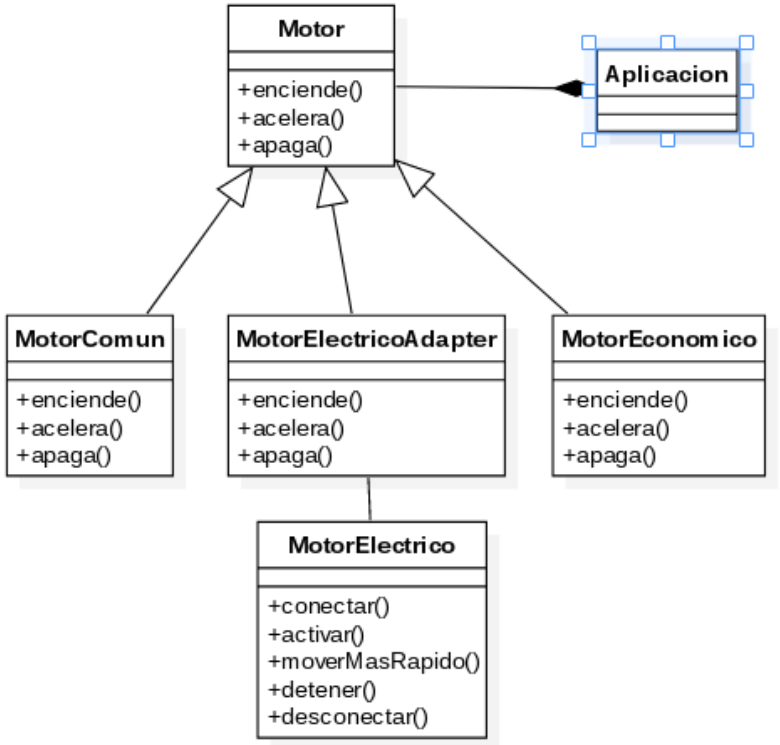


Obtenido del libro Heard First Design Patterns

- Adapter-Adaptador
  - Tenemos un sistema que trabaja con diferentes tipos de motores (Común, Económico)
  - Se desea vincular al sistema una clase de tipo motor Eléctrico con un funcionamiento diferente al de los demás, se debe adaptar la nueva clase sin que esto afecte la lógica inicial de la aplicación...



- Adapter-Adaptador
  - Tenemos un sistema que trabaja con diferentes tipos de motores (Común, Económico)
  - Se desea vincular al sistema una clase de tipo motor Eléctrico con un funcionamiento diferente al de los demás, se debe adaptar la nueva clase sin que esto afecte la lógica inicial de la aplicación...



Obtenido de: http://codejavu.blogspot.com.es/2013/08/ejemplo-patron-adapter.html

Adapter-Adaptador

```
public class MotorElectricoAdapter extends Motor{
    private MotorElectrico motorElectrico;

    public MotorElectricoAdapter(){
        super();
        this.motorElectrico = new MotorElectrico();
        System.out.println("Creando motor Electrico adapter");
    }
    @Override
    public void encender() {
        System.out.println("Encendiendo motorElectricoAdapter");
        this.motorElectrico.conectar();
        this.motorElectrico.activar();
    }

    @Override
    public void acelerar() {
        System.out.println("Acelerando motor electrico...");
        this.motorElectrico.moverMasRapido();
    }

    @Override
    public void apagar() {
        System.out.println("Apagando motor electrico");
        this.motorElectrico.detener();
        this.motorElectrico.desconectar();
    }
}
```

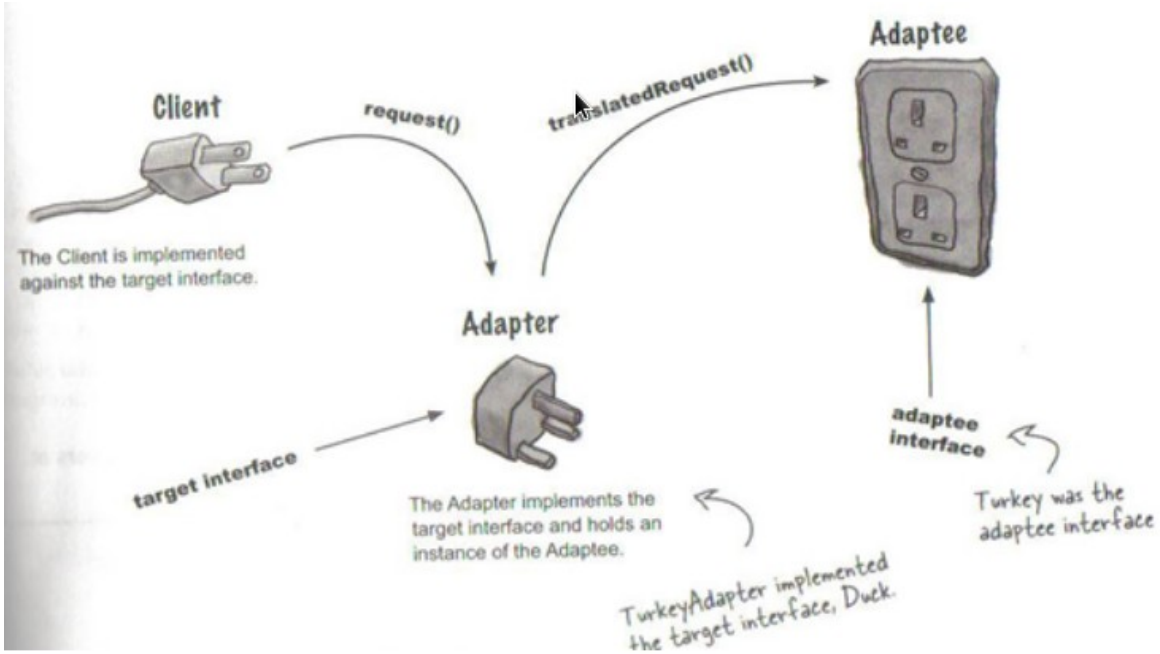
```
public class MotorElectrico {
private boolean conectado = false;

public MotorElectrico() {
    System.out.println("Creando motor electrico");
    this.conectado = false;
}
public void conectar() {
    System.out.println("Conectando motor electrico");
    this.conectado = true;
}
public void activar() {
    if (!this.conectado) {
        System.out.println("No se puede activar porque no " +
          "esta conectado el motor electrico");
    } else {
        System.out.println("Esta conectado, activando motor" +
          " electrico....");
    }
}
public void moverMasRapido() {
    if (!this.conectado) {
        System.out.println("No se puede mover rapido el motor " +
          "electrico porque no esta conectado...");
    } else {
        System.out.println("Moviendo mas rapido");
    }
}
public void detener() {
    if (!this.conectado) {
        System.out.println("No se puede detener motor electrico" +
          " porque no esta conectado");
    } else {
        System.out.println("Deteniendo motor electrico");
    }
}
public void desconectar() {
    System.out.println("Desconectando motor electrico...");
    this.conectado = false;
}
}
```

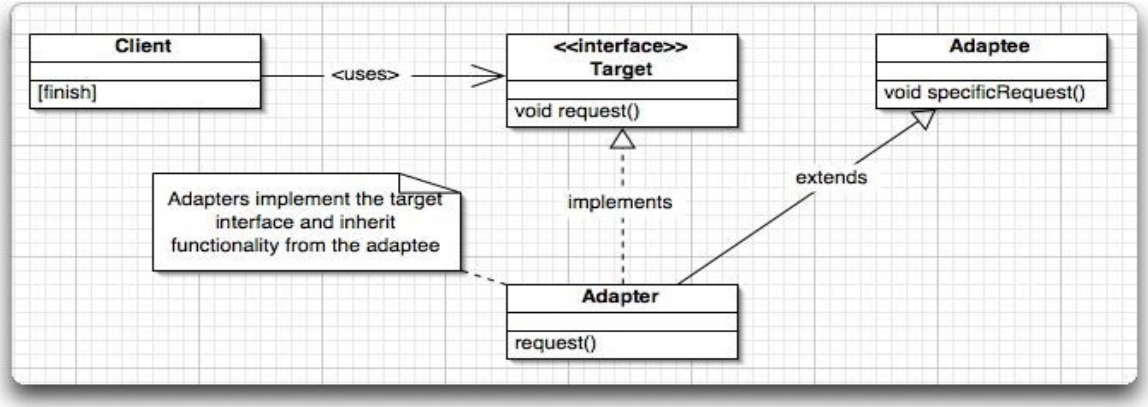
- Adapter-Adaptador
  - Este patrón se puede utilizar con el patrón de Factoria o Singleton.

```
Public static void Main(String []args) {
    Motor motor = new MotorEconomico();
    motor.encender();
    motor.acelerar();
    motor.apagar();
}
```

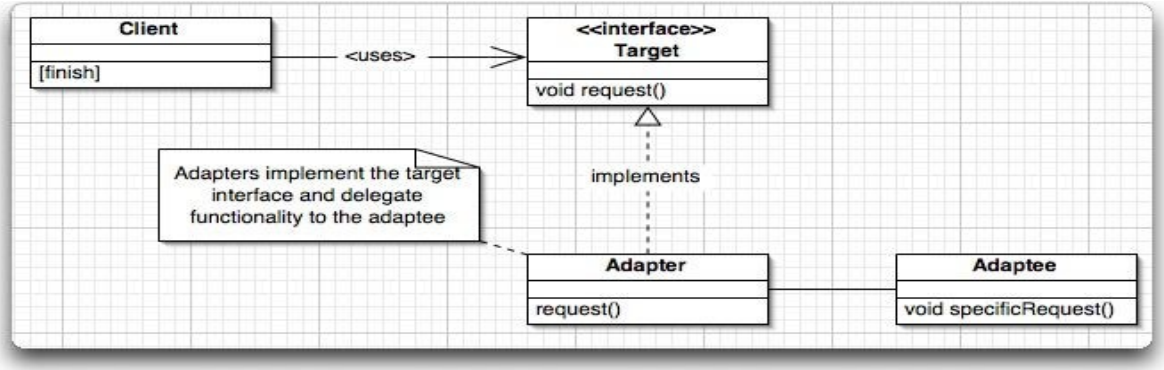
```
Public static void Main(String []args) {
    Motor motor = new MotorElectricoAdapter();
    motor.encender();
    motor.acelerar();
    motor.apagar();
}
```



- Adapter-Adaptador (2/4)
  - Estructura (Libro Gang of Four):
    - ✓ Client colabora con la conformación de objetos para la interfaz Target.
    - ✓ Target define la interfaz que el cliente necesita
    - ✓ Adaptee implementa la interfaz que queremos reutilizar
    - ✓ Adapter adapta la interfaz de Adaptee a Target



Herencia



Composición

Adapter-Adaptador (3/4)

Aplicabilidad:

- ✓ Se desea usar una clase existente y su interfaz no coincide con la que se necesita.
- ✓ Se desea crear una clase reutilizable que debe colaborar con clases no relacionadas o imprevistas.

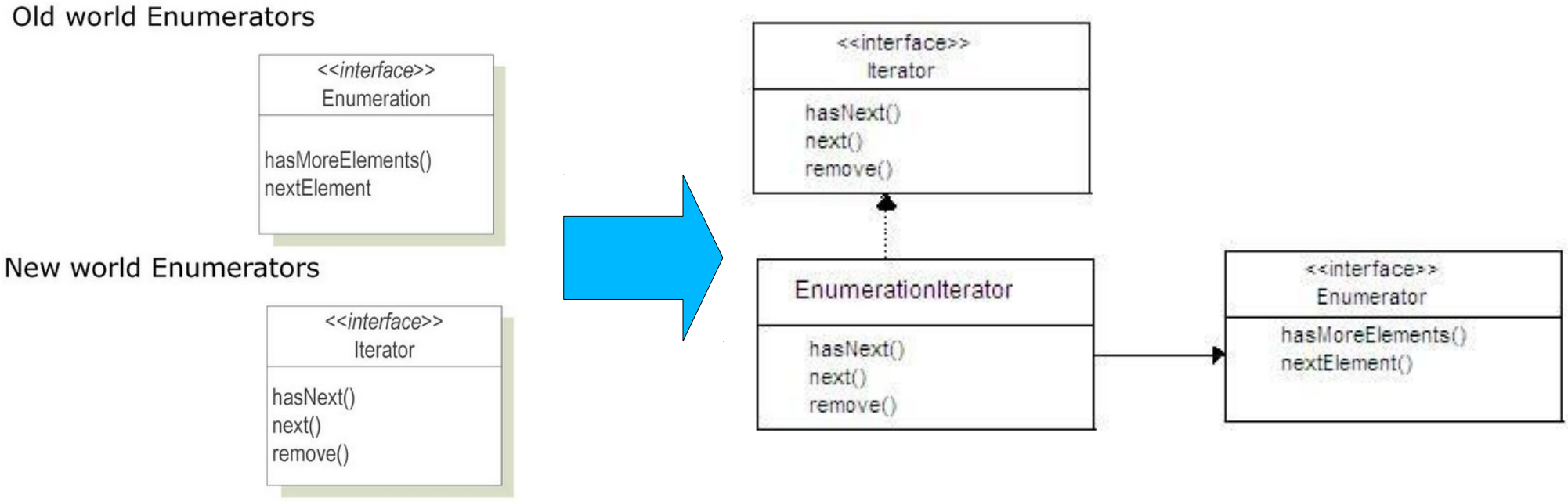
Consecuencias

- ✓ Un adaptador de clase:
  - Una clase adaptadora no funcionará cuando se desea adaptar una clase y todas sus subclases.
  - Permite a los Adapter sobrescribir algo de comportamiento de Adaptee, ya que Adapter es una subclase de Adaptee.
- ✓ Un adaptador de objeto:
  - Permite que un único Adapter trabaje con muchos Adaptees, es decir, el Adapter por sí mismo y las subclases (si es que la tiene).
  - Hace difícil sobrescribir el comportamiento de Adaptee. Esto requerirá derivar Adaptee y hacer que Adapter se refiera a la subclase en lugar que al Adaptee por sí mismo.

Usos en el API de Java

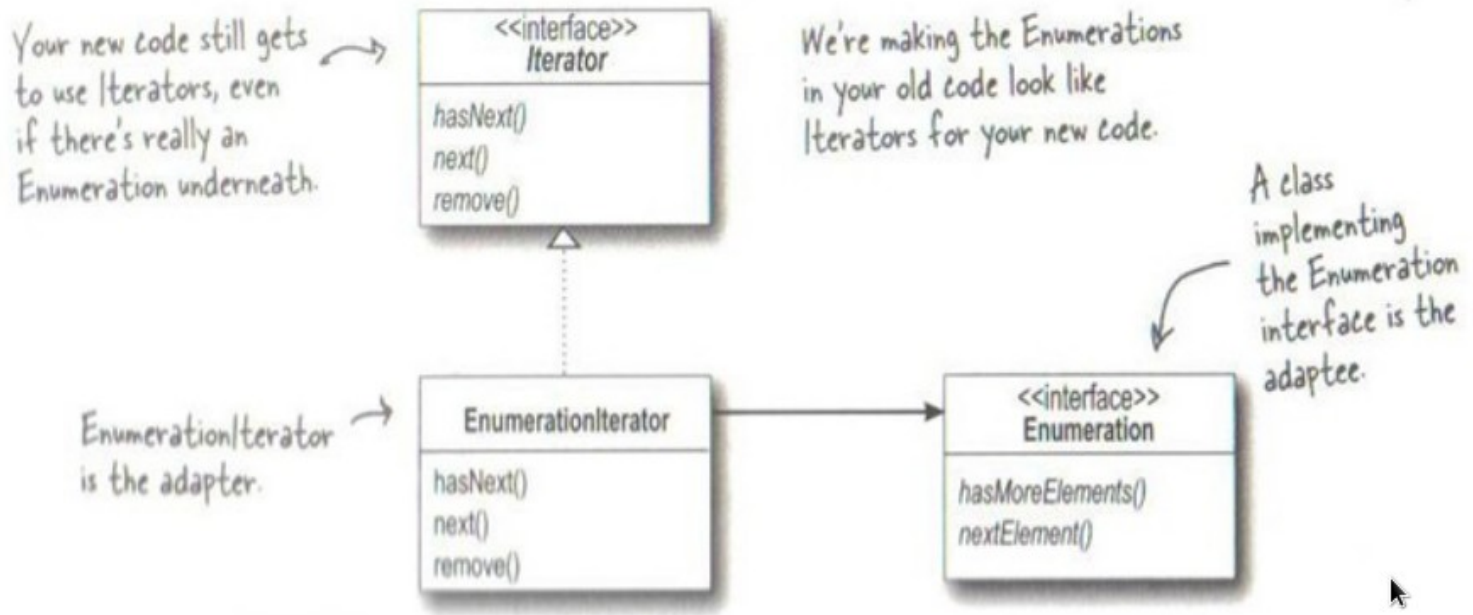
- ✓ Gestión de Eventos: ActionListener, …

Adapter-Adaptador (4/4)



Obtenido del libro Heard First Design Patterns

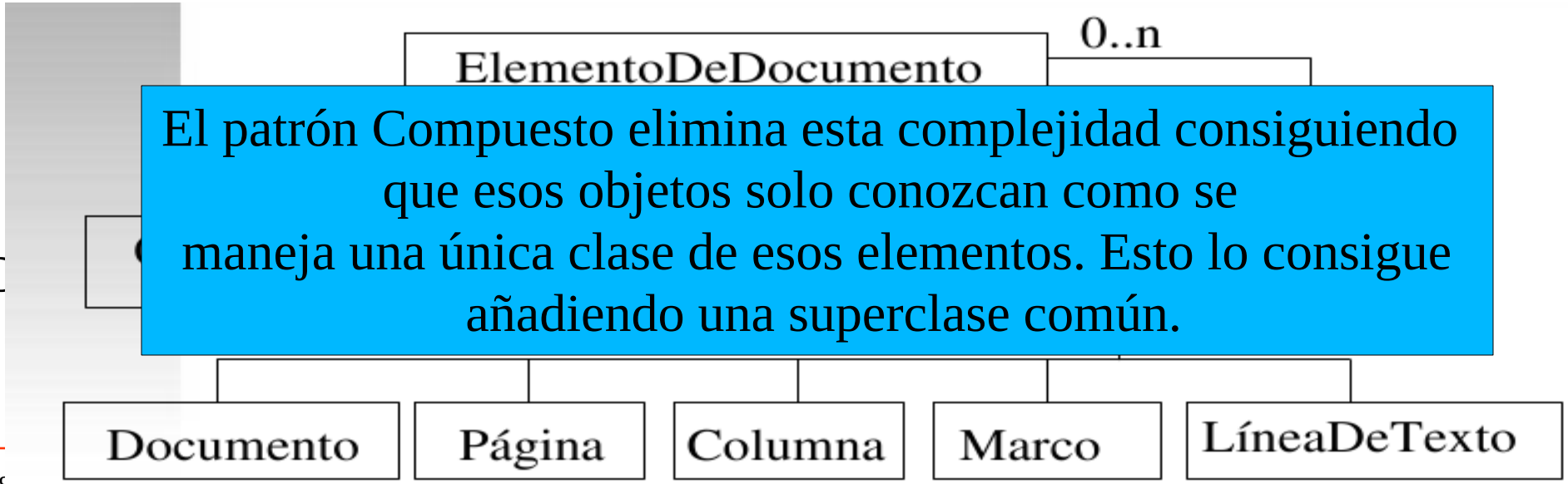
Adapter-Adaptador (4/4)




```
public class EnumerationIterator implements Iterator {
        Enumeration enumeration;
        public EnumerationIterator(Enumeration enumeration) {
                this.enumeration = enumeration;
        }
        public boolean hasNext() {
                return enumeration.hasMoreElements();
        }
        public Object next() {
                return enumeration.nextElement();
        }
        public void remove() {
                throw new UnsupportedOperationException();
```

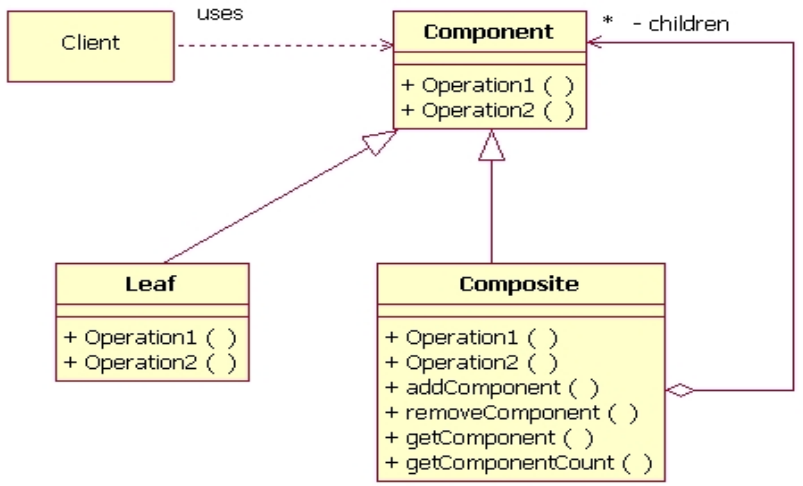
Obtenido del libro Heard First Design Patterns

- Composite-Compositor(1/4)
  - Objetivo:
    - ✓ Permite tratar objetos compuestos como si de uno simple se tratase.
  - Motivación
    - ✓ Un documento está formado por varias páginas, las cuales están formadas por columnas que contienen líneas de texto, formadas por caracteres. Las columnas y páginas pueden contener marcos. Los marcos pueden contener columnas. Las columnas, marcos y líneas de texto pueden contener imágenes.



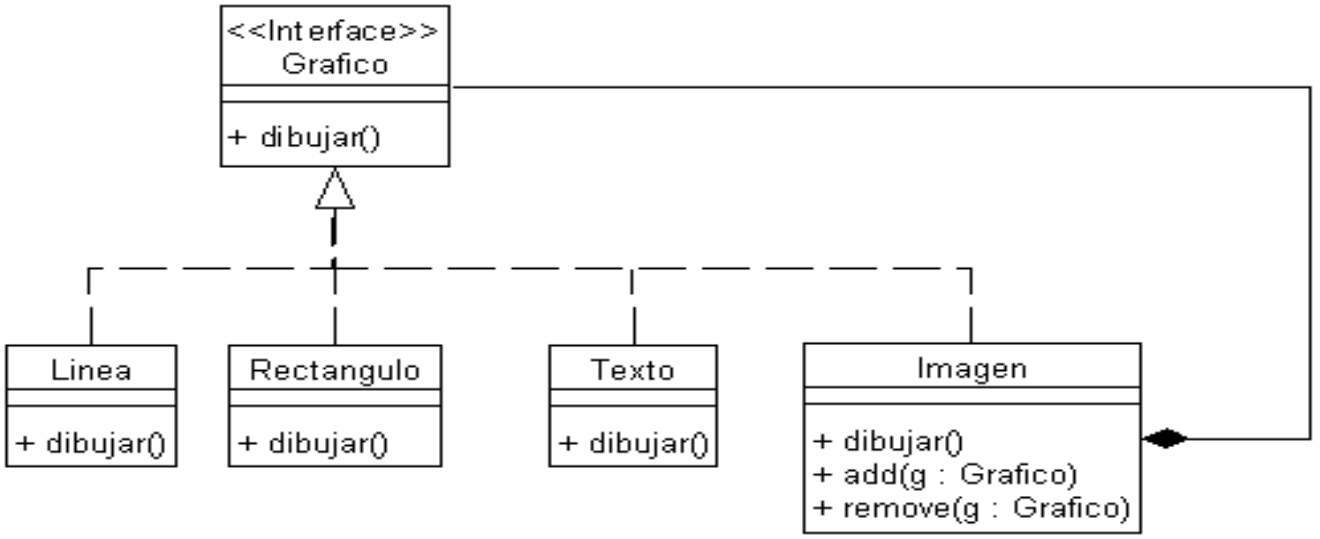
El patrón Compuesto elimina esta complejidad consiguiendo que esos objetos solo conozcan como se maneja una única clase de esos elementos. Esto lo consigue añadiendo una superclase común.

- Composite-Compositor(2/4)
  - Estructura (Libro Gang of Four):
    - ✓ Component define la interfaz común e implementa el comportamiento por defecto, junto con la interfaz necesaria para acceder a los hijos y opcionalmente al padre
    - ✓ Leaf representa a los objetos sin hijos y define el comportamiento de los objetos primitivos de la composición
    - ✓ Composite define el comportamiento de los objetos con hijos, almacena los hijos e implementa las operaciones de acceso a los hijos
    - ✓ Client manipula los objetos de la composición a través de la interfaz Component

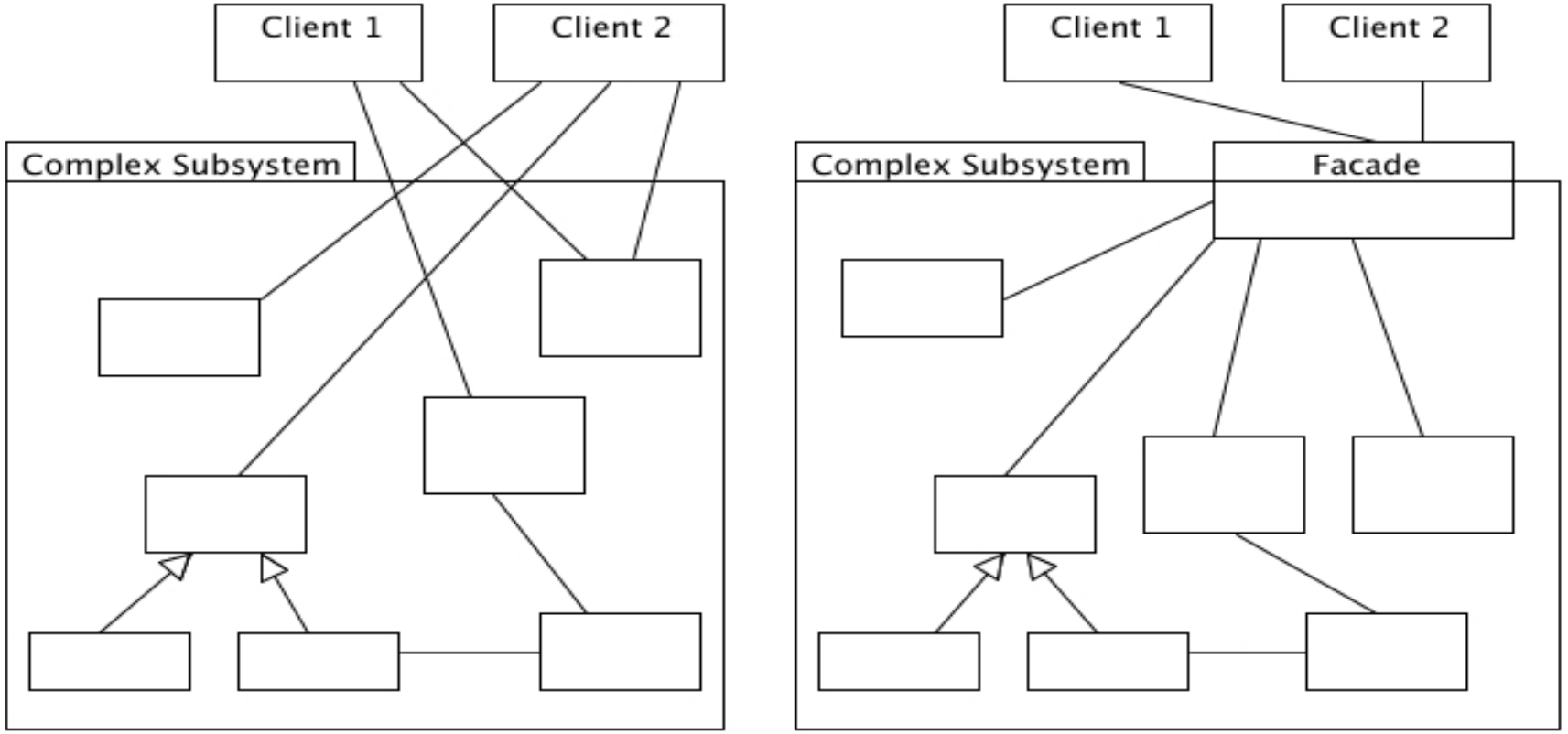


- Composite-Compositor(3/4)
  - Aplicabilidad:
    - ✓ Sirve para representar jerarquías continente-contenido
    - ✓ Se quiere encapsular la diferencia entre objetos simples y composiciones de objetos
  - Solución.
    - ✓ Referencias de componentes hijos a su padre puede ayudar al recorrido y manejo de la estructura compuesta.
    - ✓ **¿Dónde colocamos las operaciones de añadir y eliminar hijos, y obtener hijos?**
      - Compromiso entre seguridad y transparencia.
      - **Transparencia**: en clase Componente
      - **Seguridad**: en clase Composite

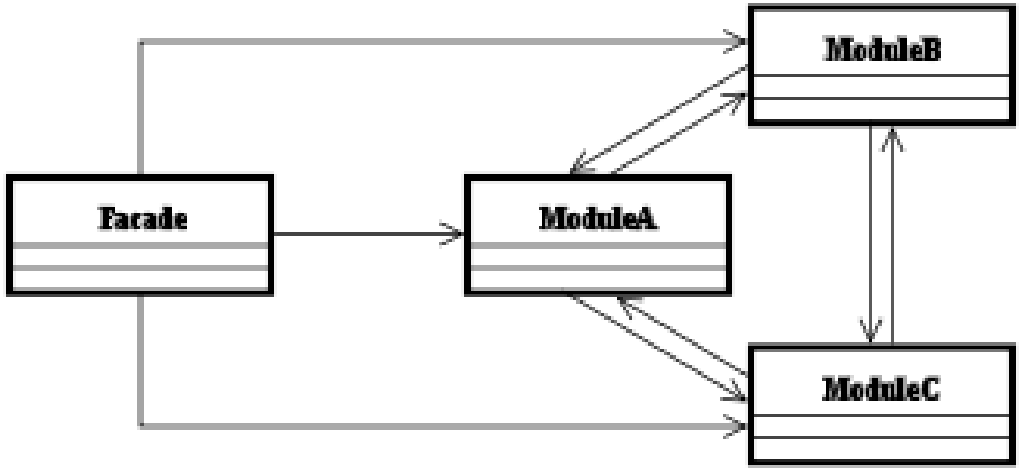
- Composite-Compositor(4/4)
  - Consecuencias
    - ✓ Permite tratar de manera homogénea a objetos simples y compuestos
    - ✓ Simplifica el código de los clientes, evitando distinciones de casos
    - ✓ Facilita la definición de nuevos tipos de componentes sin afectar a los clientes
  - Usos en el API de Java
    - ✓ En Java: las clases java.awt.Component (Componente), java.awt.Container (Contenedor), java.awt.Panel (Contenedor concreto), java.awt.Button (Boton)



- Facade-Fachada (1/4)
  - Objetivo:
    - ✓ Provee de una interfaz unificada simple para acceder a una interfaz o grupo de interfaces de un subsistema.
  - Motivación

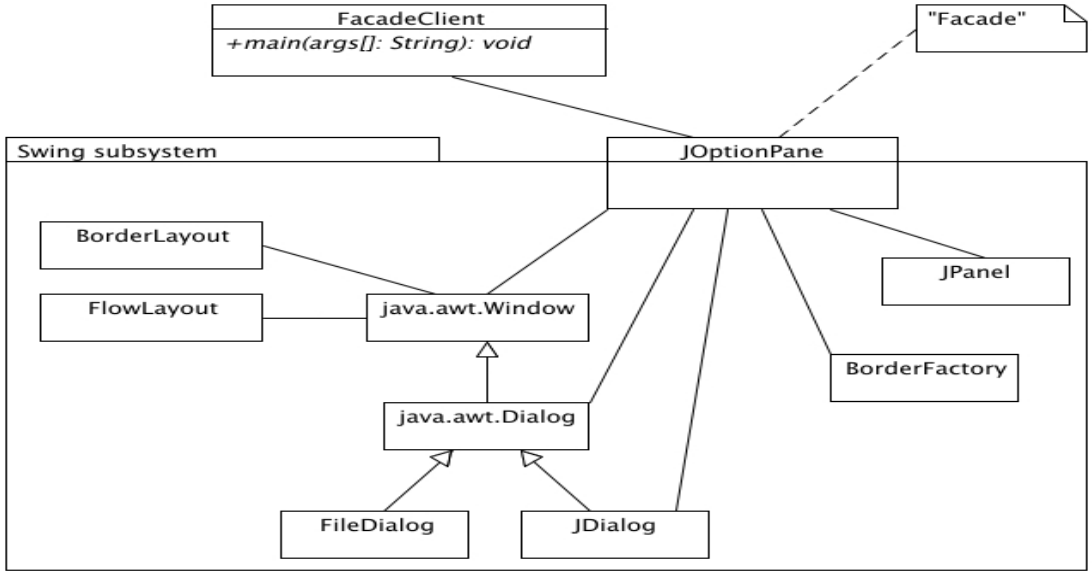


- Facade-Fachada (2/4)
  - Estructura (Libro Gang of Four):
    - ✓ Fachada (Facade): conoce qué clases del subsistema son responsables de una determinada petición, y delega esas peticiones de los clientes a los objetos apropiados del subsistema.
    - ✓ Subclases (ModuleA, ModuleB, ModuleC...): implementan la funcionalidad del subsistema. Realizan el trabajo solicitado por la fachada. No conocen la existencia de la fachada.

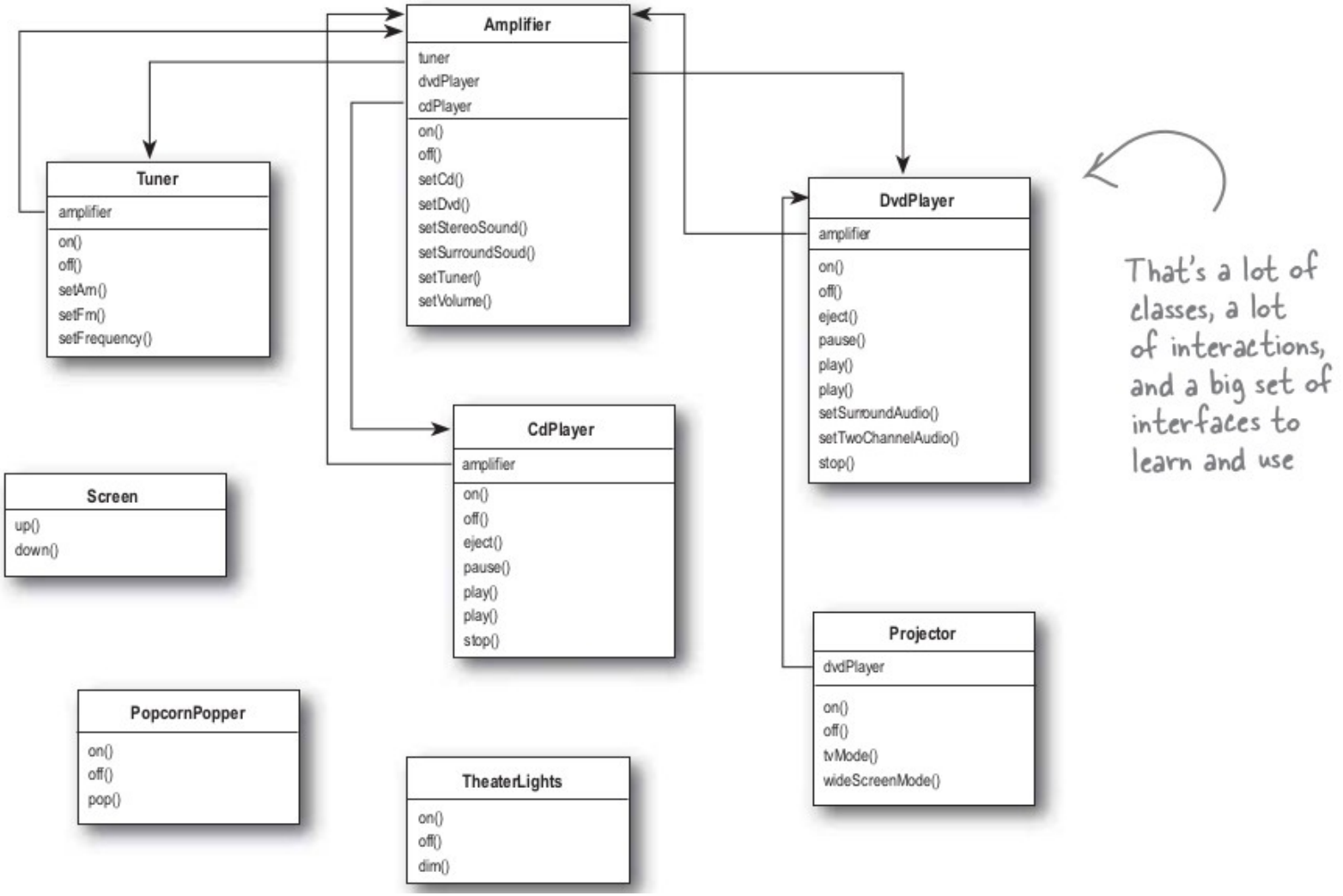


- Facade-Fachada (3/4)
  - Aplicabilidad:
    - ✓ Proporciona una interfaz simple a un subsistema complejo. El uso de patrones tiende a complicar el diseño y a la mayoría de los clientes les basta con una interfaz simple
    - ✓ Permite abstraer a los clientes de las implementaciones concretas incluidas en un subsistema, haciéndolo más portable
    - ✓ Simplifica las dependencias entre susbsistemas y permite organizarlos en capas
  - Solución.
    - ✓

- Facade-Fachada (4/4)
  - Consecuencias
    - ✓ la reducción del acoplamiento entre clientes y subsistemas (consiguiendo que los cambios de las clases del sistema sean transparentes a los clientes) y el aislamiento de cambios en la implementación.
  - Usos en el API de Java
    - ✓ javax.swing.JOptionPane, es un Facade, provee una interfaz para diferentes diálogos (Confirm, Input, Message and OptionDialogs)

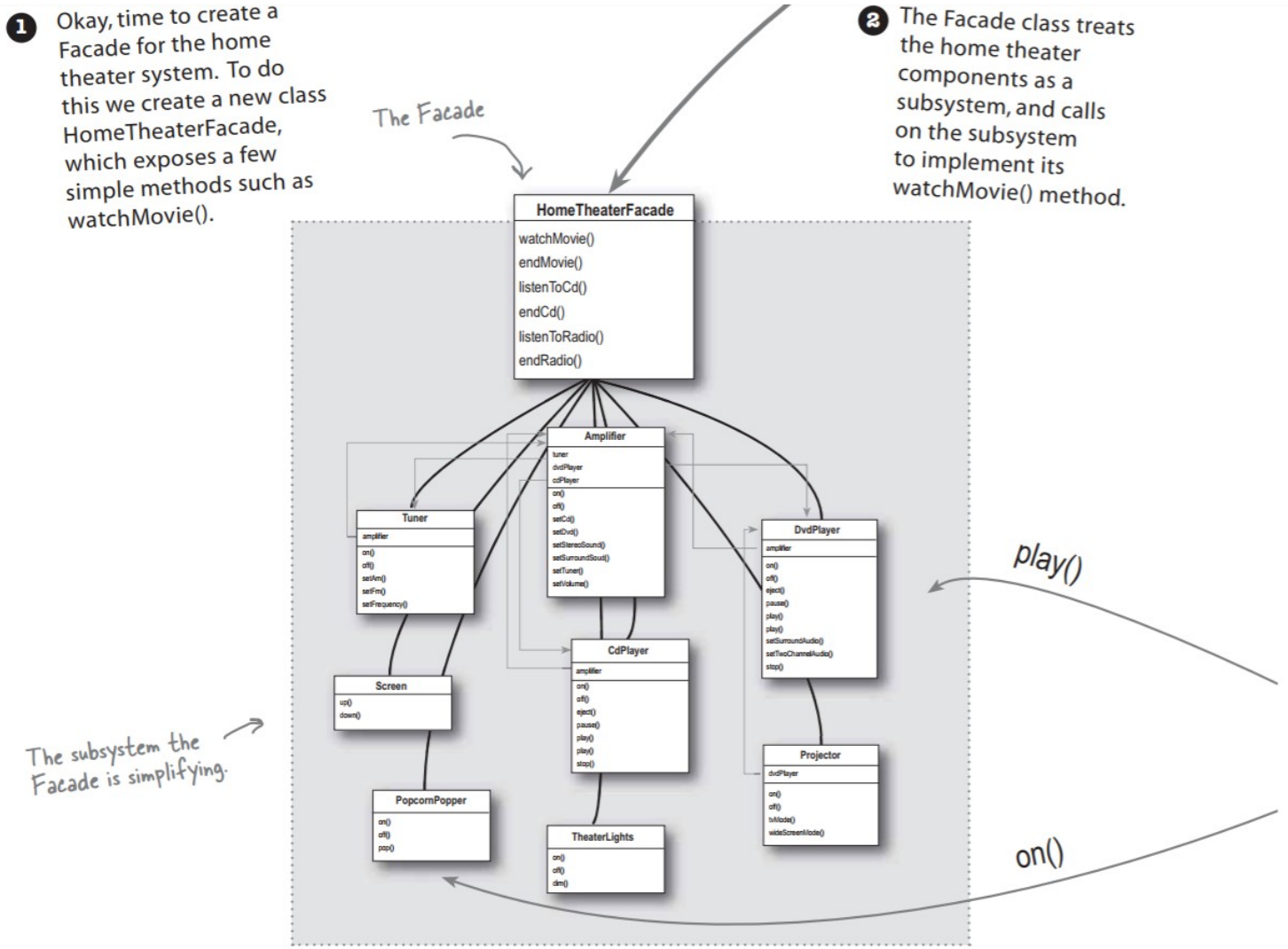


Facade-Fachada (4/4)



Obtenido del libro Heard First Design Patterns

Facade-Fachada (4/4)



Obtenido del libro Heard First Design Patterns

- Catálogo de patrones GoF

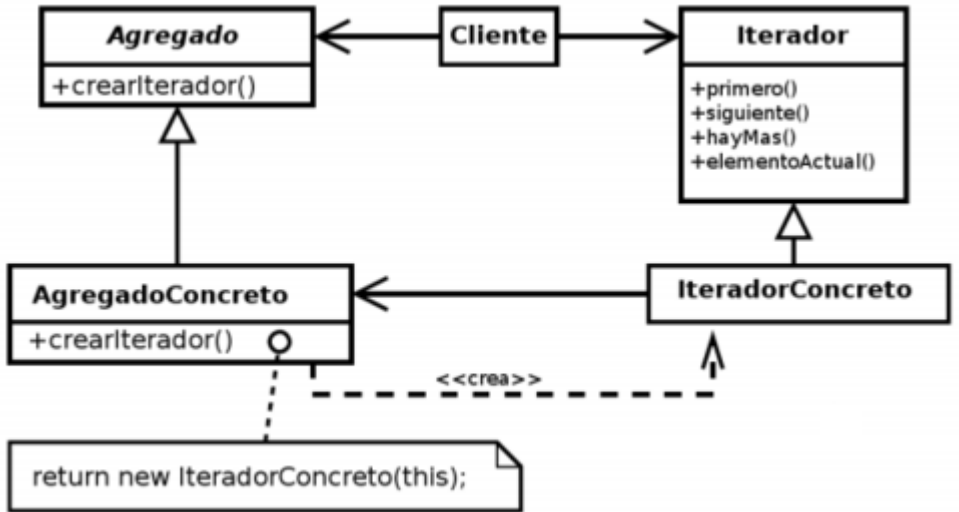
		Propósito		
		Creación	*Estructural*	*Comportamiento*
Ámbito	*Herencia*	Factory Method	**Adapter**	Interpreter Template Method
	Compo-sición	Abstract Factory Builder Prototype Singleton	**Adapter** Bridge **Composite** Decorator **Facade** Flyweight Proxy	Chain of Responsability Command **Iterator** Mediator Memento Observer State Strategy Visitor

Patrones de comportamiento

- Chain of Responsibility (Cadena de responsabilidad): Permite establecer la línea que deben llevar los mensajes para que los objetos realicen la tarea indicada.
- Command (Orden): Encapsula una operación en un objeto, permitiendo ejecutar dicha operación sin necesidad de conocer el contenido de la misma.
- Interpreter (Intérprete): Dado un lenguaje, define una gramática para dicho lenguaje, así como las herramientas necesarias para interpretarlo.
- Iterator (Iterador): Permite realizar recorridos sobre objetos compuestos independientemente de la implementación de estos.
- Mediator (Mediador): Define un objeto que coordine la comunicación entre objetos de distintas clases, pero que funcionan como un conjunto.
- Memento (Recuerdo): Permite volver a estados anteriores del sistema.
- Observer: Define una dependencia de uno-a-muchos entre objetos, de forma que cuando un objeto cambie de estado se notifique y actualicen automáticamente todos los objetos que dependen de él.
- State (Estado): Permite que un objeto modifique su comportamiento cada vez que cambie su estado interno.
- Strategy (Estrategia): Permite disponer de varios métodos para resolver un problema y elegir cuál utilizar en tiempo de ejecución.
- Template Method (Método plantilla): Define en una operación el esqueleto de un algoritmo, delegando en las subclases algunos de sus pasos, esto permite que las subclases redefinan ciertos pasos de un algoritmo sin cambiar su estructura.
- Visitor (Visitante): Permite definir nuevas operaciones sobre una jerarquía de clases sin modificar las clases sobre las que opera.

- Iterator (1/4)
  - Objetivo:
    - ✓ Define una interfaz que declara los métodos necesarios para acceder secuencialmente a un grupo de objetos de una colección. Algunos métodos: Primero(), Siguiente(), HayMas() y ElementoActual().
  - Motivación
    - ✓ El patrón surge del deseo de acceder a los elementos de un contenedor de objetos (por ejemplo, una lista) sin exponer su representación interna.
    - ✓ La solución que propone el patrón es añadir métodos que permitan recorrer la estructura sin referenciar explícitamente su representación.
    - ✓ La responsabilidad del recorrido se traslada a un objeto iterador.

- Iterator (2/4)
  - Estructura (Libro Gang of Four):
    - ✓ Iterador (Iterator) define la interfaz para recorrer el agregado de elementos y acceder a ellos, de manera que el cliente no tenga que conocer los detalles y sea capaz de manejarlos de todos modos.
    - ✓ Iterador Concreto (ConcreteIterator) implementa la interfaz propuesta por el Iterador y se encarga de mantener la posición actual en el recorrido.
    - ✓ Agregado (Aggregate) define la interfaz para el método de fabricación de iteradores.
    - ✓ Agregado Concreto (ConcreteAggregate) implementa la estructura de datos y el método de fabricación de iteradores que crea un iterador específico para su estructura.



- Iterator
  - Aplicabilidad:
    - ✓ Permite el acceso al contenido de una estructura sin exponer su representación interna.
    - ✓ No tienen por qué limitarse a recorrer la estructura, sino que podrían incorporar otro tipo de lógica
  - Consecuencias
    - ✓ El patrón Iterador permite por tanto diferentes tipos de recorrido de un agregado y varios recorridos simultáneos, simplificando la interfaz del agregado.
  - Usos en el API de Java
    - ✓ Gestión de Eventos: ActionListener, …

Patrones de comportamiento

```
public class Vector2 {
    public int[] _datos;
    public Vector2(int valores){
        _datos = new int[valores];
        for (int i = 0; i < _datos.length; i++)
            _datos[i] = 0;
    }
    public int getValor(int pos){
        return _datos[pos];
    }
    public void setValor(int pos, int valor){
        _datos[pos] = valor;
    }
    public int dimension(){
        return _datos.length;
    }
    public IteradorVector iterador(){
        return new IteradorVector(this);
    }
}
```

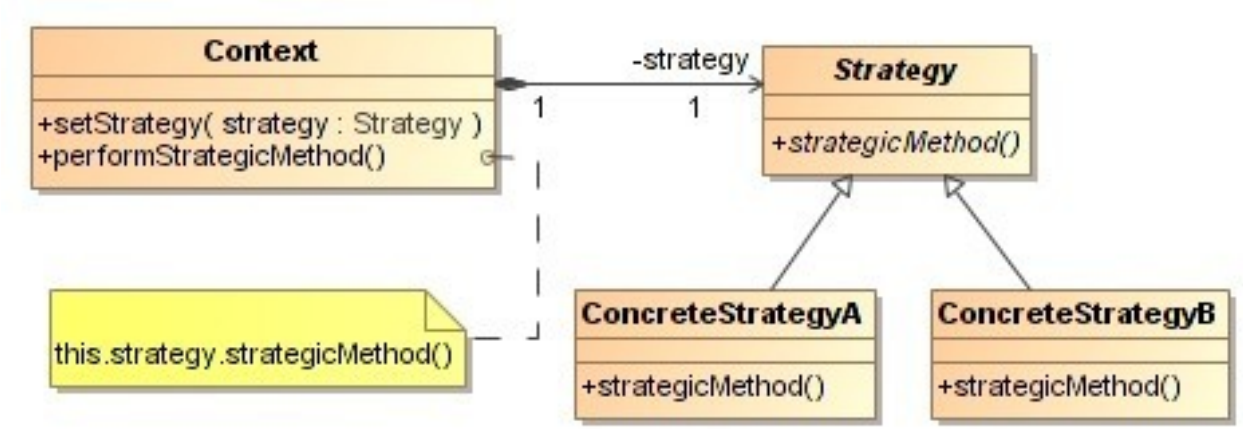
```
public class IteradorVector{
    private int[] _vector;
    private int _posicion;
    public IteradorVector(Vector2 vector) {
        _vector = vector._datos;
        _posicion = 0;
    }

    public boolean hasNext(){
        if (_posicion < _vector.length)
            return true;
        else
            return false;
    }
    public Object next(){
            int valor = _vector[_posicion];
            _posicion++;
            return valor;
    }
}
```

```
public static void main(String argv[]) {
    Vector2 vector = new Vector2(5);
    IteradorVector iterador = vector.iterador();
    while (iterador.hasNext())        //Recorrido con el iterador
       System.out.println(iterador.next());
}
```

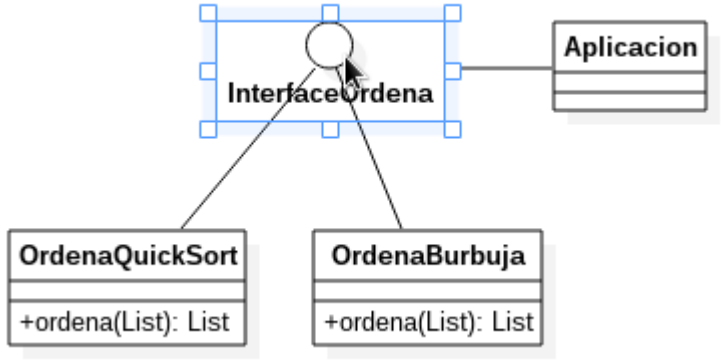
- Estrategia (1/4):
  - Objetivo:
    - ✓ Permite mantener un conjunto de algoritmos de entre los cuales el objeto cliente puede elegir aquel que le conviene e intercambiarlo dinámicamente según sus necesidades.
  - Motivación
    - ✓ Incluir el código de los algoritmos en los clientes hace que estos sean demasiado grandes y complicados de mantener y/o extender.
    - ✓ El cliente no va a necesitar todos los algoritmos en todos los casos, de modo que no queremos que dicho cliente los almacene si no los va a usar.
    - ✓ Si existiesen clientes distintos que usasen los mismos algoritmos, habría que duplicar el código, por tanto, esta situación no favorece la reutilización.
  - Solución
    - ✓ Encapsular los distintos algoritmos en una jerarquía y que el cliente trabaje contra un objeto intermediario contexto

- Estrategia (2/4)
  - Estructura (Libro Gang of Four):
    - ✓ Contexto (Context) : Es el elemento que usa los algoritmos, por tanto, delega en la jerarquía de estrategias. Configura una estrategia concreta mediante una referencia a la estrategia necesaria.
    - ✓ Estrategia (Strategy): Declara una interfaz común para todos los algoritmos soportados. Esta interfaz será usada por el contexto para invocar a la estrategia concreta.
    - ✓ EstrategiaConcreta (ConcreteStrategy): Implementa el algoritmo utilizando la interfaz definida por la estrategia.



- Estrategia (3/4)
  - Aplicabilidad:
    - ✓ Cualquier programa que ofrezca un servicio o función determinada, que pueda ser realizada de varias maneras, es candidato a utilizar el patrón estrategia.
    - ✓ Si un algoritmo utiliza información que no deberían conocer los clientes, la utilización del patrón estrategia evita la exposición de dichas estructuras.
    - ✓ Permite añadir nuevos algoritmos sin necesidad de cambiar mucho.
  - Consecuencias
    - ✓ El uso del patrón proporciona una alternativa a la extensión de contextos, ya que puede realizarse un cambio dinámico de estrategia.
    - ✓ Los clientes deben conocer las diferentes estrategias y debe comprender las posibilidades que ofrecen.
  - Usos en el API de Java
    - ✓ Gestión de Eventos: ActionListener, …

Estrategia (4/4)




```
public interface InterfaceOrdena {
   public Object[] ordena (Object [] unArray);
}

public class OrdenaQuickSort implements InterfaceOrdena{
  public Object[] ordena (Object [] unArray) {
  /* Aquí iría el código de ordenación*/
     return unArray;
  }
}

public class OrdenaBurbuja implements InterfaceOrdena{
  public Object[] ordena (Object [] unArray) {
    /* Aquí iría el código de ordenación*/
    return unArray;
  }
}
```

```
public class ContextoEstrategia {
  private InterfaceOrdena algortimoOrdenar;
  /** Se le pasa el algoritmo para ordenar */
  public void setAlgoritmoOrdenar(InterfaceOrdena algoritmoOrdenar) {
          this.algortimoOrdenar = algoritmoOrdenar;
  }
  /** Método que utiliza el algoritmo de ordenar */
  public void usarEstrategia() {
     Object[] array; // Aquí código que crea un array
     Object[] arrayOrdenado = algortimoOrdenar.ordena(array);
  }
}
```


Ejemplo obtenido de: http://www.chuidiang.org/ood/patrones/patron_estrategia.php

Antipatrones

Definen patrones que no se deben seguir.

Antipatrones

- Nos ofrecen una forma de resolver un problema típico, documentando lo que no se debe hace.
- Nos hablan de actitudes o formas de enfrentarse a los problemas que ya sabemos suelen tener consecuencias negativas.
- También reflejan experiencia, pero de este caso, de los errores cometidos.

- Objetivos
  - Dar a conocer y clasificar los errores mas comunes a la hora de desarrollar software.
  - Proponer un proceso para atacar esos problemas.
  - Evitar que se repitan constantemente.
  - Difundir la experiencia de otros desarrolladores.

Antipatrones de Desarrollo de Software

- The Blob (Clases Gigantes)
- Lava Flow (Código Muerto).
- Functional Decomposition (Diseño no Orientado a Objetos)
- Poltergeists (No se sabe lo que hace alguna clase)
- Golden Hammer (Para un martillo todos son clavos)
- Spaghetti Code (Sobre anidamiento de IF's o SWITCH's)
- Cut-and-Paste Programming (Copiar y pega código)

Antipatrones de Arquitectura de software.

- Stovepipe Enterprise: Aislamiento en la empresa o desarrollo de sistmas aislados y parchados
- Stovepipe Systems: Sistemas que por su mala arquitectura se comportan como si sus componentes hubieran sido desarrollados por separado.
- Vendor Lock-In: Arquitectura dependiente del Producto.
- Architecture by implication: Arquitectura implícita.
- Design by Commite: El proyecto es diseñado por un comité demasiado numerosos e inexperto.
- Reinvent the Whell: Desarrollo desde cero.

Bibliografía Recomendada

- Libros:
  - Design Patterns: Elements of Reusable Object-Oriented Software, Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides (the"Gang of Four"or Gof),1994.
  - "Patterns in Java, A Catalog of Reusable Design Patterns Illustrated with UML"
  - UML y patrones. Introducción al análisis y diseño orientado a objetos. Larman.
  - Head First. Design patterns. EricFreeman & Elisabeth Freeman. O'reilly
  - Piensa en Java. 4ª Edición. Bruce Eckel. Pearson Prentice Hall.
- Web:
  - PFC. Guía de construcción de software en java con patrones de diseño. Director: Juan Manuel Cueva Lovelle. Autor: Francisco Javier Martínez Juan.
    http://www.di.uniovi.es/~cueva/asignaturas/PFCOviedo/PFCpatronesJava.pdf