Patrones de Diseño

Una introducción a su aplicación al diseño de software

Introducción

- Diseñar software orientado a objetos es difícil
- Diseñar software reusable orientado a objetos es todavía más difícil
- Es difícil que un diseñador inexperto sea capaz de hacer un buen diseño
  - Conoce los principios básicos de la orientación a objetos (herencia, polimorfismo, etc.) y un lenguaje de programación orientado a objetos
  - Pero no sabe cómo usar esos conocimientos de manera eficaz

Introducción

- Un diseñador experto es capaz de hacer buenos diseños
  - Se basan en su experiencia
    - Reutilizan soluciones de diseño que les han funcionado bien en el pasado para resolver problemas similares
    - Rechazan soluciones y enfoques que en el pasado no les han permitido resolver problemas similares
- Un patrón es una solución a un problema de diseño no trivial que es
  - Efectiva: ha valido para resolver el problema en diseños pasados
  - Reusable: la solución es la misma en problemas similares

Ventajas e Inconvenientes

- Ventajas del diseño con patrones
  - Permiten reutilizar soluciones probadas
  - Proporcionan soluciones flexibles ante cambios en el futuro
  - Facilitan la comunicación entre diseñadores
    - Los patrones tienen nombres estándar
  - Facilitan el aprendizaje al diseñador inexperto
- Inconvenientes del diseño con patrones
  - Introducen mayor complejidad que los diseños específicos
- Qué no son los patrones de diseño
  - Código directamente aplicable a cada problema
  - La solución más eficiente (complejidad computacional o espacial)
  - Error: "Cuantos más patrones se utilicen mejor software"

Tipos de Patrones en el Software

- Patrones de Arquitectura
  - Soluciones probadas para estructurar los componentes (Modelo-Vista-Controlador, arquitectura tres capas, peer to peer, Maestro-esclavo, arquitectura orientada a servicios,…)
- Patrones Web
  - Soluciones probadas para la creación de sitios Web (diseño 3 columnas, rollovers, migas de pan, …)
- **Patrones de Diseño**
  - Soluciones eficaces para el diseño de software orientado a objetos
- Patrones de Programación
  - Soluciones específicas para algoritmos y estructuras de control (algoritmos de ordenación, recursión, iteración, funciones objeto, FOS,…)
- Patrones de Refactorización
  - Soluciones para simplificar el código sin modificar su comportamiento

Tipos de Patrones de Diseño
Clasificación según [GoF]

- Patrones Creacionales
  - **Abstract Factory** (Fábrica abstracta), Builder (Constructor virtual), **Factory Method** (Método de fabricación), Prototype (Prototipo), **Singleton** (Instancia única)
- Patrones Estructurales
  - **Adapter** (Adaptador), Bridge (Puente), **Composite** (Objeto compuesto), Decorator (Envoltorio), **Facade**(Fachada), Flyweight (Peso ligero), Proxy
- Patrones de Comportamiento
  - Chain of Responsibility (Cadena de responsabilidad), **Command** (Orden), Interpreter (Intérprete), Iterator (Iterador), Mediator (Mediador), Memento (Recuerdo), **Observer** (Observador), **State** (Estado), Template Method (Método Plantilla), Visitor (Visitante)

Descripción de Patrones

- Plantilla para patrones de diseño [GoF]
  - Nombre: nombre estándar reconocido por la comunidad (generalmente en inglés)
  - Clasificación: creacional , estructural o de comportamiento
  - Intención: ¿Qué problema pretende resolver el patrón?
  - También conocido como: Otros nombres de uso común para el patrón
  - Motivación: Escenario de ejemplo para la aplicación del patrón
  - Aplicabilidad: Usos comunes y criterios de aplicabilidad del patrón
  - Estructura: Diagramas de clases oportunos para describir las clases que intervienen en el patrón
  - Participantes: Enumeración y descripción de las entidades abstractas (y sus roles) que participan en el patrón
  - Colaboraciones: Explicación de las interrelaciones que se dan entre los participantes
  - Consecuencias: Consecuencias positivas y negativas en el diseño derivadas de la aplicación del patrón
  - Implementación: Técnicas o comentarios oportunos de cara a la implementación del patrón
  - Código de ejemplo
  - Usos conocidos: Ejemplos de sistemas reales que usan el patrón
  - Patrones relacionados

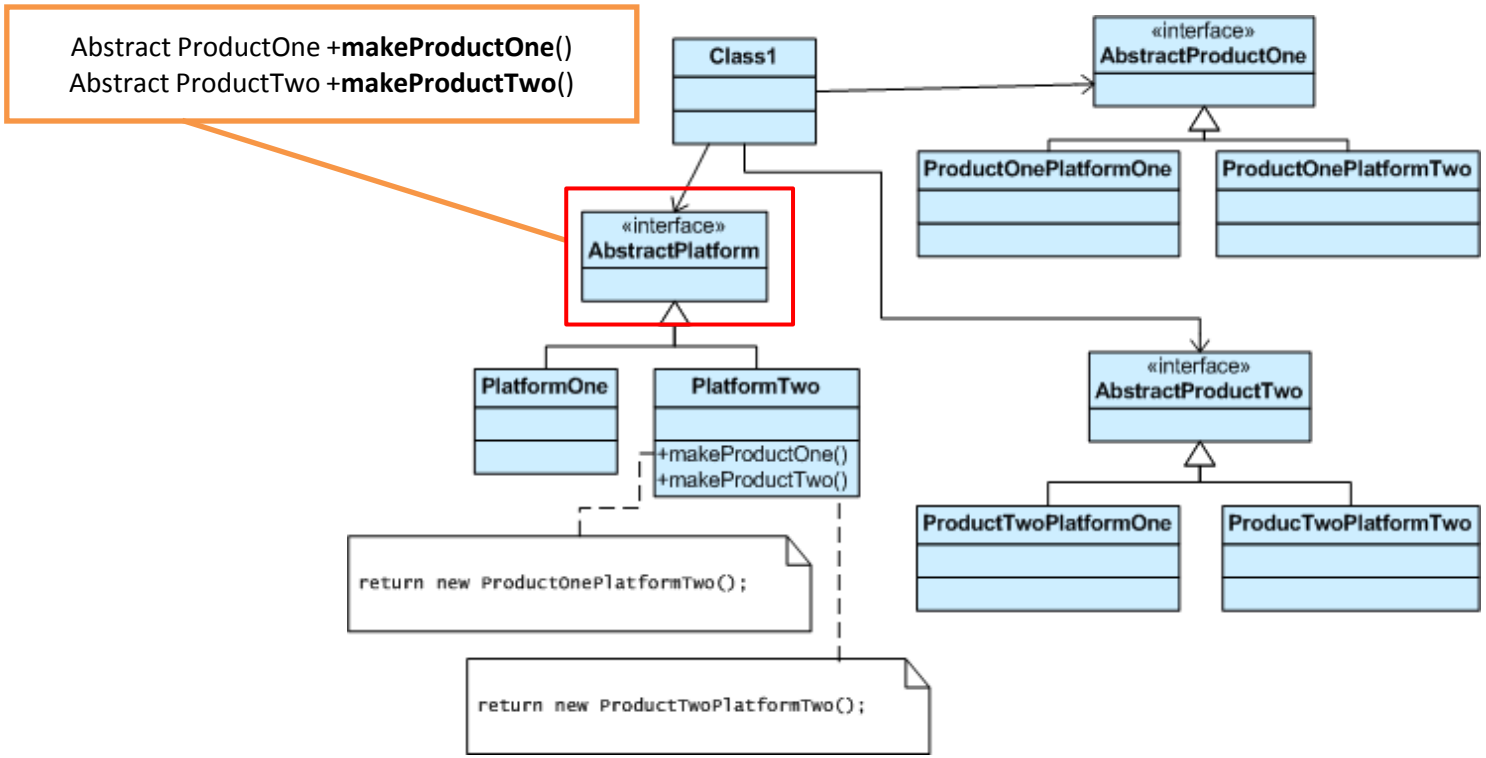
Patrones de Creación

- Propósito:
  - Crear un objeto es una toma de decisión.
  - Separar los procesos de creación de objeto y de uso de un objeto.
- ¿Cuáles veremos?
  - Abstract Factory
  - Factory Method
  - Singleton
  - Builder
  - Prototype

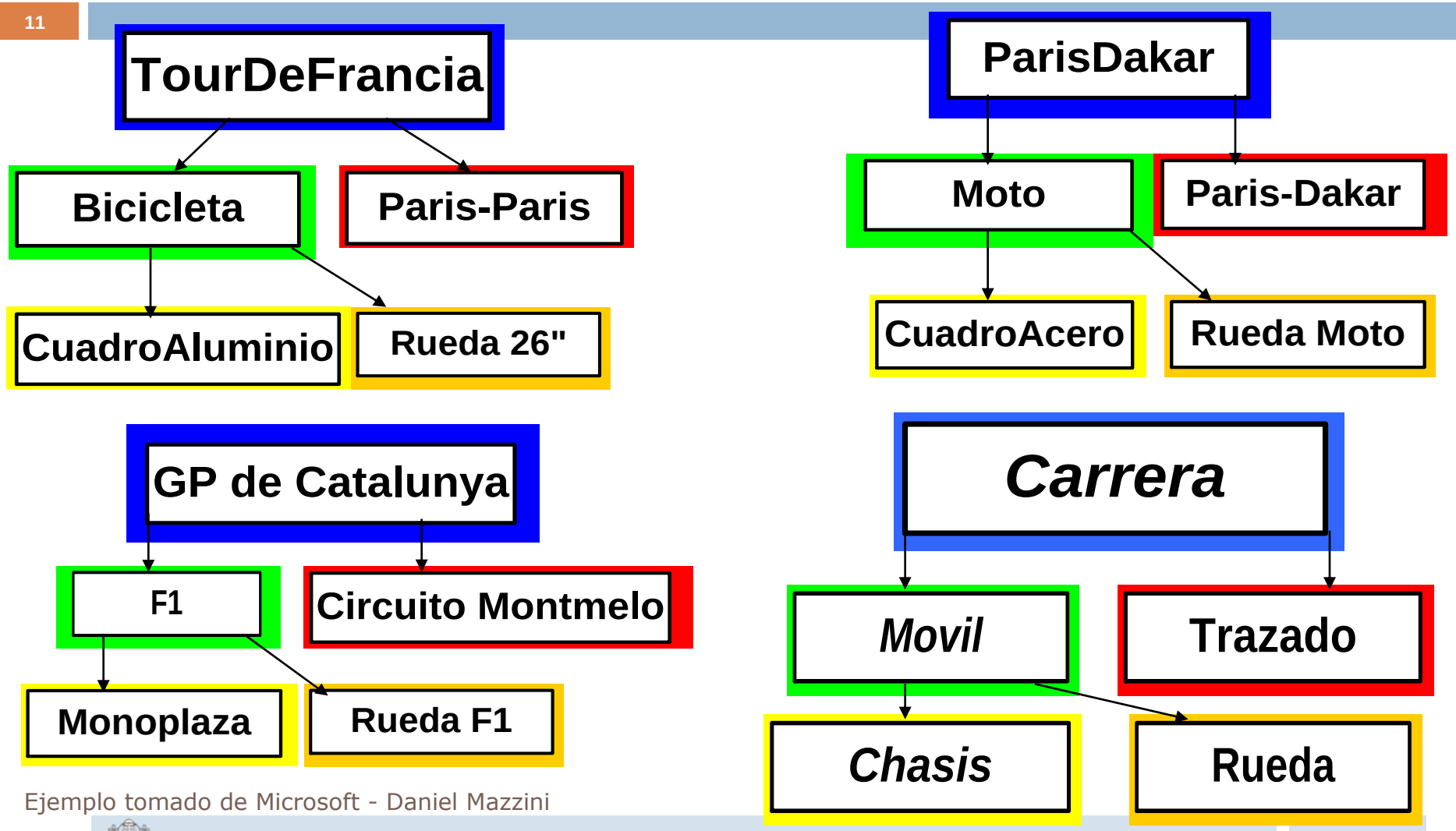
Abstract Factory– Patrón de Creación (Fábrica Abstracta, Kit)

- Propósito
  - Proporciona una interfaz para crear familias de objetos, sin especificar clases concretas
- Aplicabilidad (¿Cuándo?)
  - Responde a la necesidad de diferentes tipos de objetos que deben colaborar (familia de productos)
  - Configura un sistema seleccionando una familia de productos entre varios
  - Separa (creación/composición/representación) de los productos, de cómo se utilizan
  - Cuando queremos ofrecer una biblioteca de productos, y sólo queremos revelar sus interfaces no sus implementaciones

Abstract Factory - Estructura



Abstract Factory – Ejemplo



Ejemplo tomado de Microsoft - Daniel Mazzini

Abstract Fa...ión

Carrera
+CrearMovil(Chasis,Rueda[]):Movil
+CrearRueda():Rueda
+CrearChasis():Chasis
+CrearTrazado():Trazado

TourDeFrancia
+CrearMovil(Chasis,Rueda[]):Movil
+CrearRueda():Rueda
+CrearChasis():Chasis
+CrearTrazado():Trazado

```
Ruedas CrearRueda() {
    return new Rueda26();
}
```

ParisDakar
+CrearMovil(Chasis,Rueda[]):Movil
+CrearRueda():Rueda
+CrearChasis():Chasis
+CrearTrazado():Trazado

```
Ruedas CrearRueda() {
    return new RuedaMoto();
}
```

GP de Catalunya
+CrearMovil(Chasis,Rueda[]):Movil
+CrearRueda():Rueda
+CrearChasis():Chasis
+CrearTrazado():Trazado

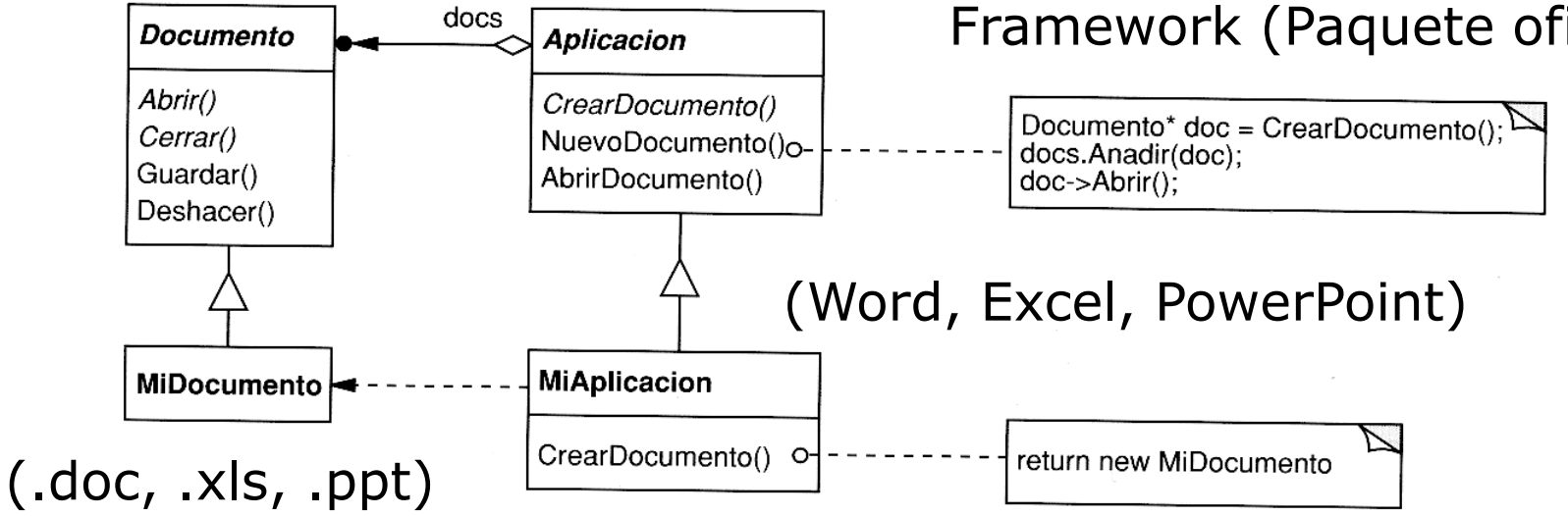
```
Ruedas CrearRueda() {
    return new RuedaF1();
}
```

Abstract Factory - Consecuencias

- Aísla las clases concretas del cliente que las usa
- Facilita el intercambio de familias de productos
- Promueve la consistencia entre productos: familias
- Es difícil dar cabida a nuevos tipos de productos
  - Implica crear nuevos métodos en toda la jerarquía de fábricas

Factory Method – Patrón de Creación
(Método de Fabricación, Virtual Constructor, Constructor Virtual)

- Propósito y Aplicabilidad
  - Define una interfaz para crear un objeto, pero permite a sus subclases decidir qué clase instanciar.
  - Permite a una clase delegar la instanciación en sus subclases.
  - Define un Constructor Virtual

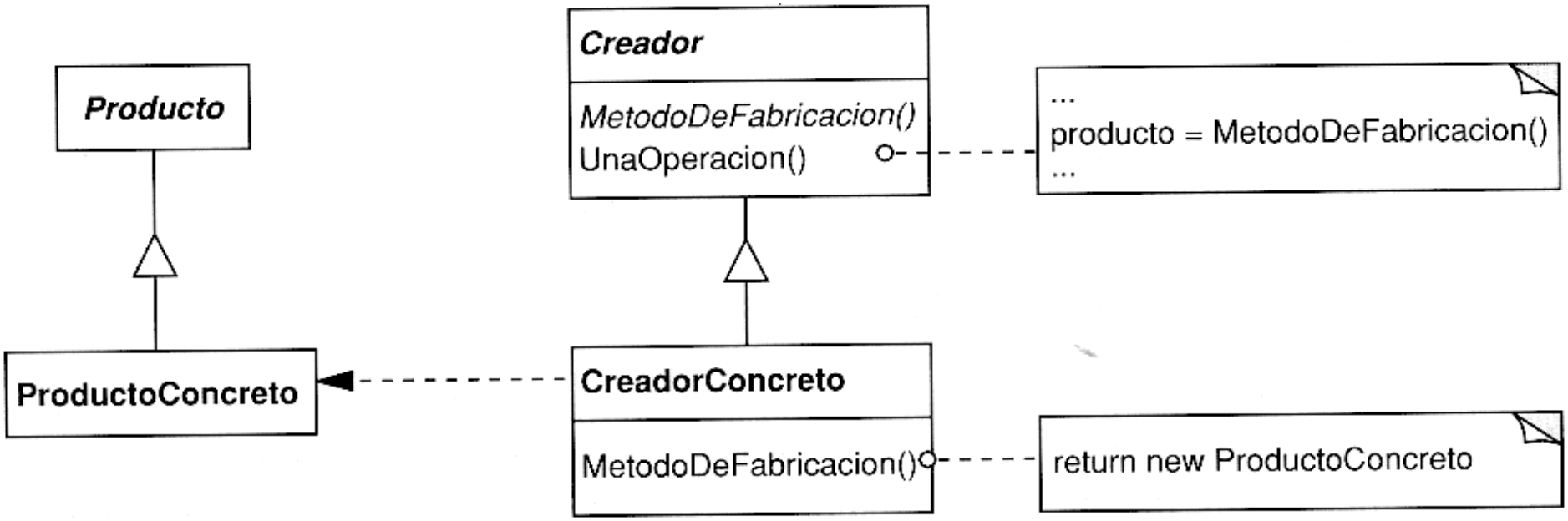


Framework (Paquete ofimático)

(Word, Excel, PowerPoint)

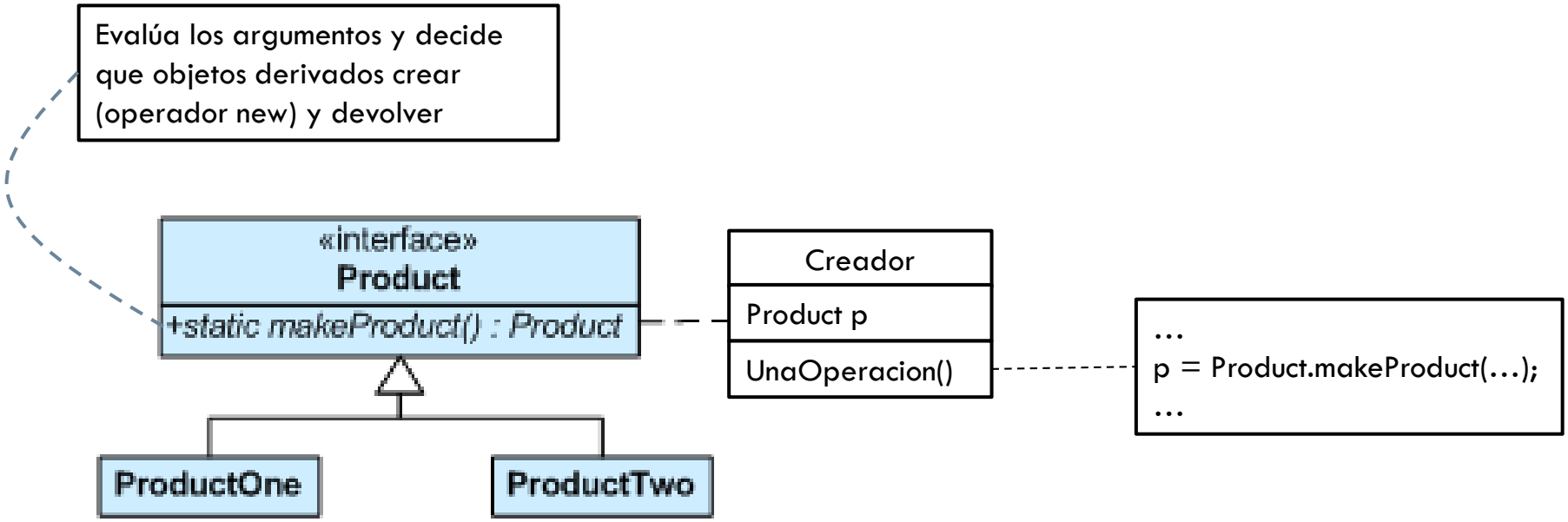
(.doc, .xls, .ppt)

Factory Method- Estructura



- Creador al crear objetos Producto no utiliza el operador **new**, sino que delega en el método *MetodoDeFabricacion()*

Factory Method (Parametrizado) – Estructura alternativa



- Creador delega en un método estático *makeProduct()* de Product para crear su producto concreto.

Factory Method – Parametrizado

- El Propio Producto abstracto dispone de una **operación estática** para crear distintos Productos
- Los argumentos de la operación definen el Producto Concreto a crear en cada caso
- Se devuelve una subclase de Producto
- Los constructores pueden ser protegidos/privados

```
/* Producto Abstracto */
abstract class Producto{
 static Producto creaProducto(Tipo);
}
/* Productos concretos */
class Producto-A extends Producto {…}
class Producto-B extends Producto {…}
/* uso: creando un producto de tipo A */
Producto prA = Producto.CreaProducto(1);
```

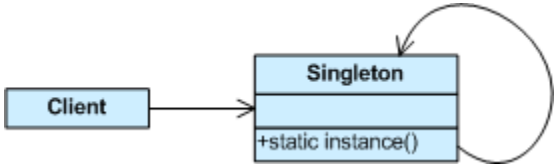
```
Switch(Tipo){
case 1:
  return new Producto-A();
case 2:
  return new Producto-B();
}
```

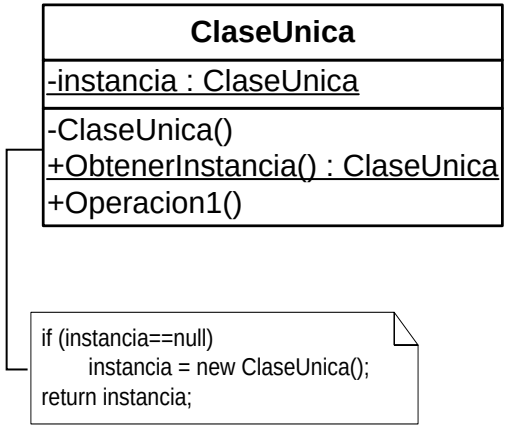
Factory Method – Consecuencias

- Consecuencias
  - Permite variar la representación interna de un producto
  - Aísla el código de construcción y representación
    - Los Clientes no necesitan conocer la estructura interna del Producto, tampoco aparecen en Constructor
    - ConstructorConcreto define todas las componentes y como se ensamblan
  - Proporciona un control más fino sobre el proceso de construcción
    - El **Constructor** construye el producto paso a paso
    - El **Cliente** sólo obtiene el producto cuando está terminado
  - Pueden devolver la misma instancia varias veces (Prototype)
  - (Parametrizado) el método del Producto abstracto puede devolver objetos de diferentes subclases

Singleton – Patrón de Creación (Único, Instancia única)

- Propósito
  - Garantiza que una clase tenga una sola instancia, y proporciona un punto global de acceso a ella.
- Aplicabilidad
  - Se necesita exactamente una sola instancia de una clase
  - Es deseable una inicialización perezosa de la instancia
  - Necesario un acceso global a la instancia
- Estructura





Singleton – Implementación


```
public class Singleton {

// Constructor privado – evita que se creen más instancias
 private Singleton() {}

 // instancia única
 private static Singleton instancia = null;

 // acceso (y creación) a la instancia Singleton
 public static Singleton getInstancia(){
  if (instancia == null)
     instancia = new Singleton();
  return instancia;
 }
 // resto de propiedades y métodos del objeto
}
```

Singleton - Consecuencias

- Acceso controlado a la única instancia por la clase
- Evita inundar el código con variables globales
  - (En C++, en Java no hay variables globales)
  - No deberían ser un "sustituto" de toda variable global
- Permite el refinamiento mediante herencia
  - El observador puede devolver un objeto Singleton o de una clase derivada especializada
- Permite un número de variable de instancias
  - Ej: Acceso a un objeto de comunicaciones, para no saturar un canal, el Singleton puede disponer internamente de 2 instancias que va alternando en cada solicitud.
- Son más flexibles que usar directamente la clase con métodos estáticos
  - Los métodos estáticos no se pueden reescribir en métodos heredados

Patrones Estructurales

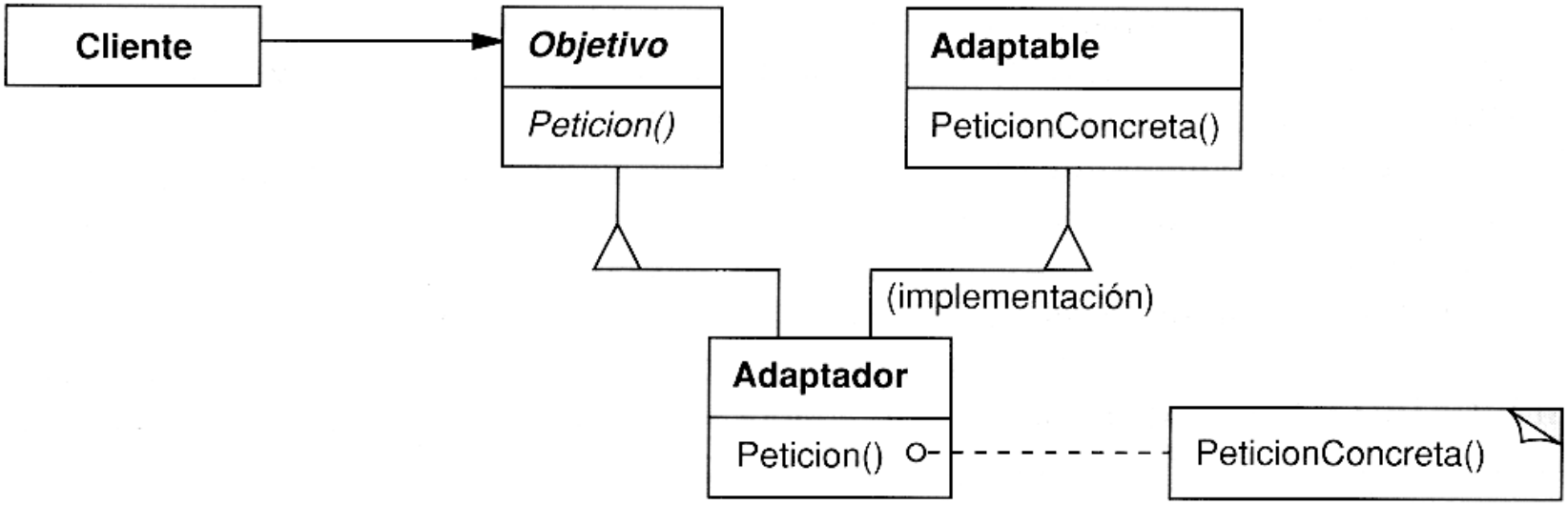
- Propósito:
  - Desacoplar el sistema
  - Obtener una estructura flexible
  - Organizar
- ¿Cuáles veremos?
  - Adapter
  - Facade
  - Composite

Adapter – Patrón Estructural (Adaptador, Wrapper, Envoltorio)

- Propósito
- Aplicabilidad
  - Se quiere usar una clase existente y su interfaz no concuerda con la que necesita
  - Se quiere crear una clase que coopere con clases con las que no comparte interfaces compatibles
    - Clase no relacionadas o que no han sido previstas
  - (adaptador de objetos) es necesario usar varias subclases existentes pero no resulta práctico adaptar su intefaz heredando de cada una de ellas. Un adaptador de objetos puede adaptar la interfaz de su clase padre

Adapter – Estructura

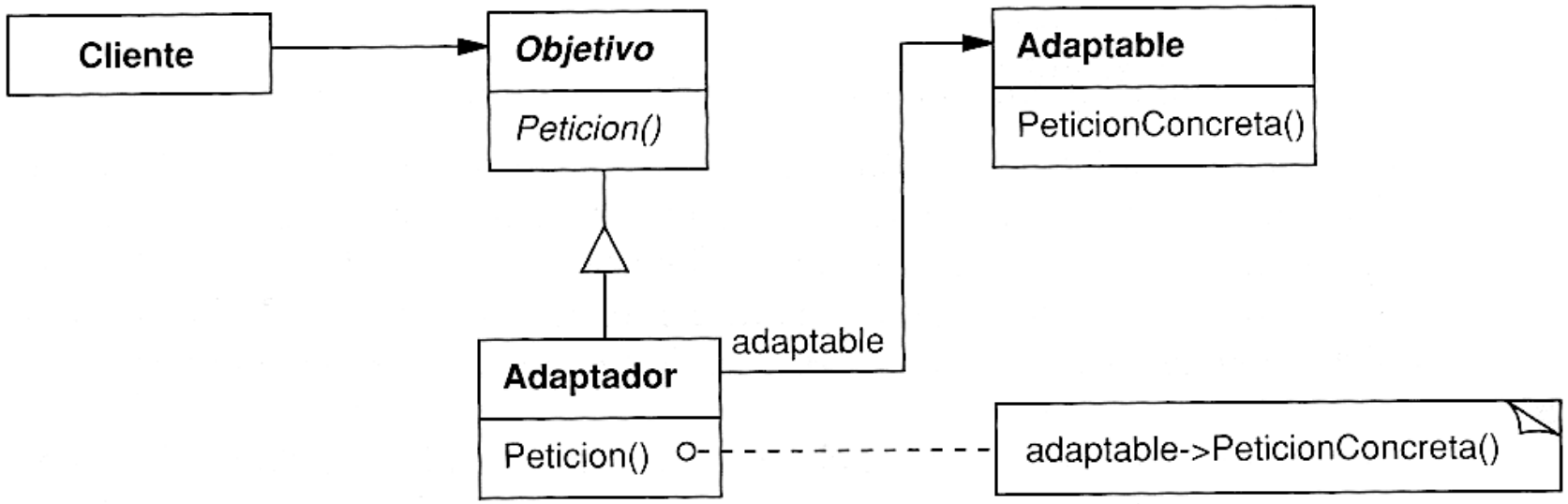
- Adaptador de clases



- Adaptación por **herencia** de la clase Adaptable y herencia de la clase Objetivo

Adapter – Estructura

- Adaptador de objetos

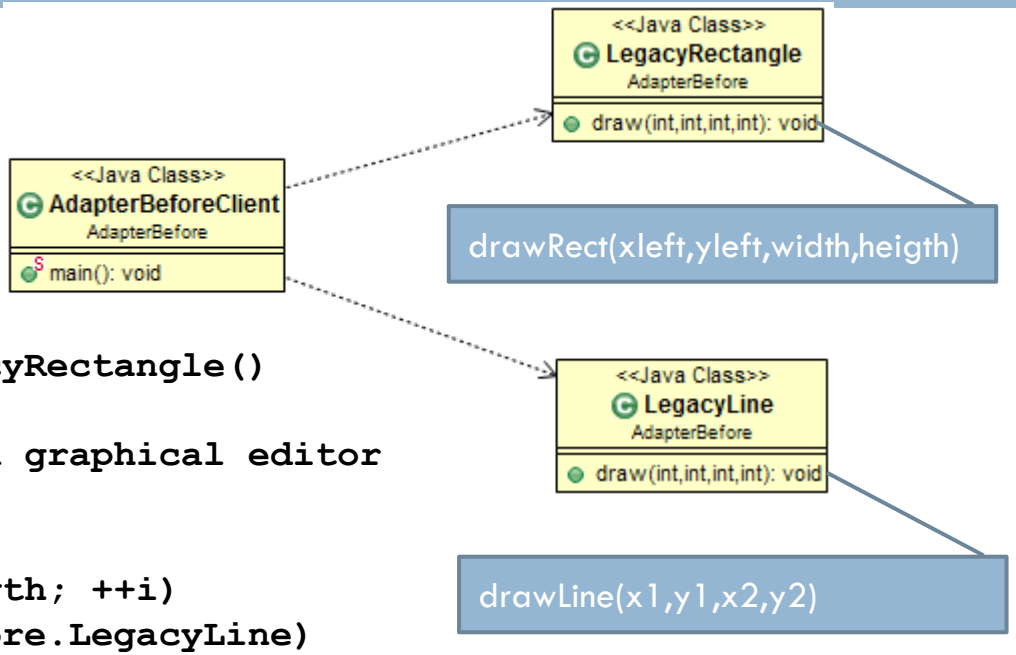


- Adaptación por **composición** de la clase Adaptable y herencia de la clase Objetivo

Adapter - Participantes

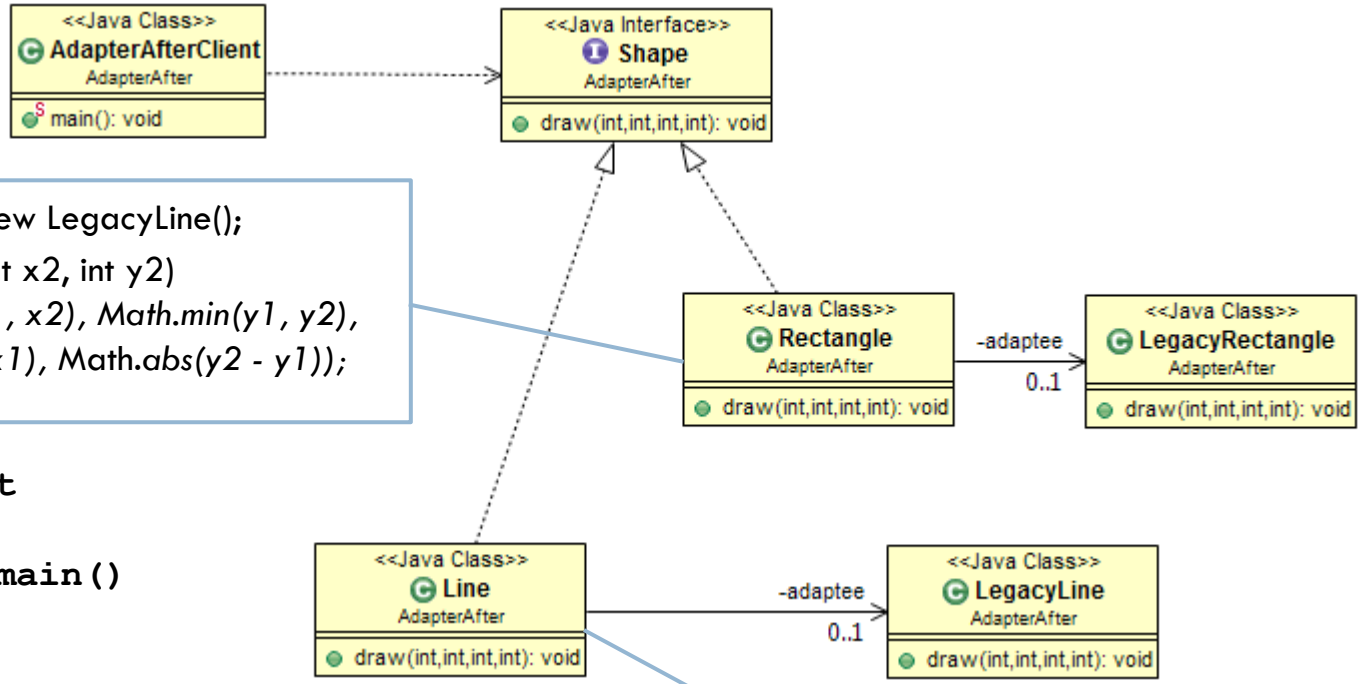
- Objetivo
  - Define la interfaz específica del dominio que usa el Cliente
- Cliente
  - Colabora con objetos que se ajustan con la interfaz Objetivo
- Adaptable
  - Define una clase existente con una inferfaz que necesita ser adaptada
- Adaptador
  - Adapta la interfaz de Adaptable a la interfaz Objetivo

Adapter – Ejemplo (sin Adapter)




```
Class AdapterBeforeClient{
  public static void main()
    {
        Object[] shapes = {
            new LegacyLine(), new LegacyRectangle()
        };
        // A begin and end point from a graphical editor
        int x1 = 10, y1 = 20;
        int x2 = 30, y2 = 60;
        for (int i = 0; i < shapes.length; ++i)
          if (shapes[i] instanceof before.LegacyLine)
            ((LegacyLine)shapes[i]).draw(x1, y1, x2, y2);
          else if (shapes[i] instanceof before.LegacyRectangle)
            ((LegacyRectangle)shapes[i]).draw(Math.min(x1, x2), Math.min(y1, y2)
              , Math.abs(x2 - x1), Math.abs(y2 - y1));
...
```

Adapter – Ejemplo (con Adapter de objetos)

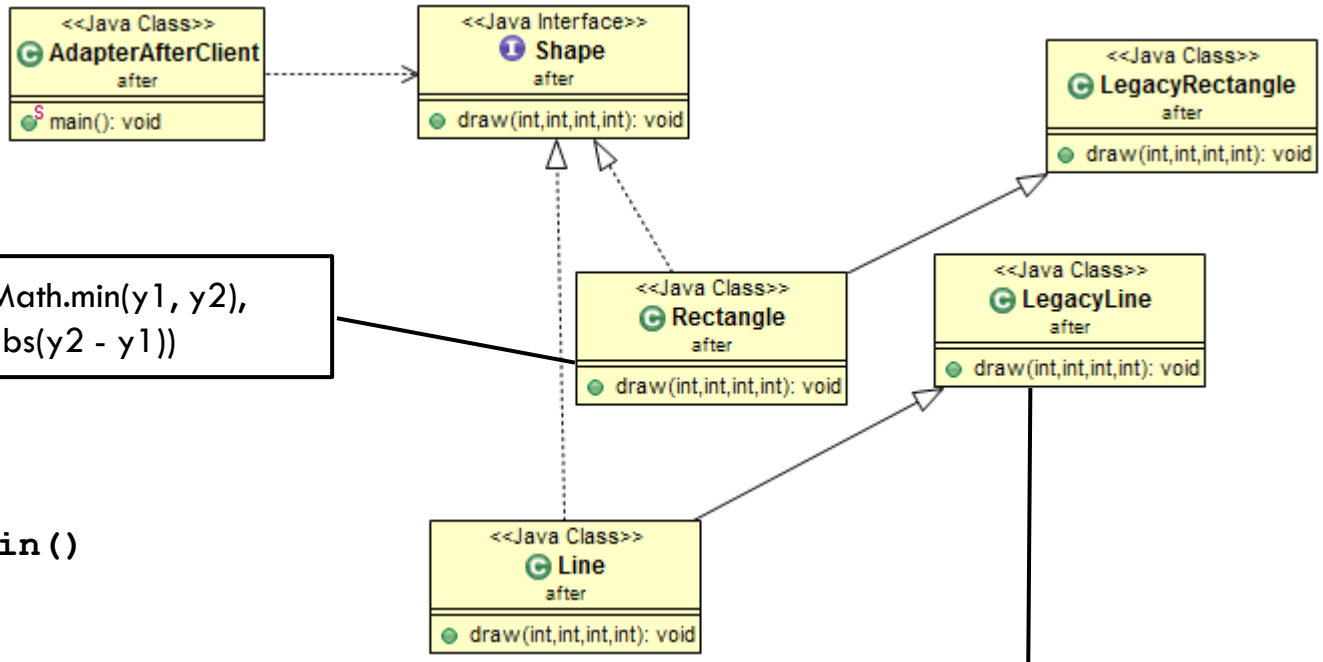



```
private LegacyLine adaptee = new LegacyLine();
public void draw(int x1, int y1, int x2, int y2)
{ adaptee.drawRect(Math.min(x1, x2), Math.min(y1, y2),
                   Math.abs(x2 - x1), Math.abs(y2 - y1));
}
```

```
private LegacyLine adaptee = new LegacyLine();
public void draw(int x1, int y1, int x2, int y2)
{ adaptee.drawLine(x1,y1,x2,y2); }
```

```
class AdapterAfterClient
{
    public static void main()
    {
     Shape[] shapes = {
            new Line(), new Rectangle()};
 // A begin and end point from a graphical editor
     int x1 = 10, y1 = 20;
     int x2 = 30, y2 = 60;
     for (int i = 0; i < shapes.length; ++i)
       shapes[i].draw(x1, y1, x2, y2);
...
```

Adapter – Ejemplo (con Adapter de clases)




```
class AdapterAfterClient
{
    public static void main()
    {
     Shape[] shapes = {
            new Line(), new Rectangle()};
 // A begin and end point from a graphical editor
     int x1 = 10, y1 = 20;
     int x2 = 30, y2 = 60;
     for (int i = 0; i < shapes.length; ++i)
       shapes[i].draw(x1, y1, x2, y2);
```

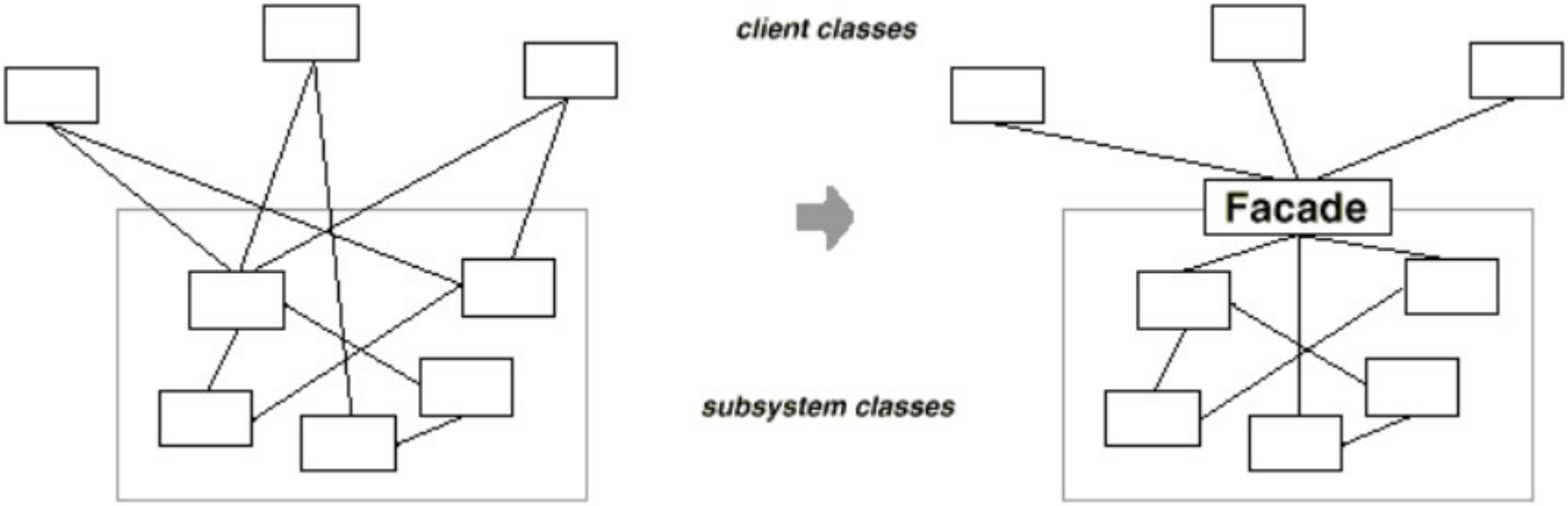
...

Adapter – Consecuencias

- Adaptador de Clases
  - Un Adapter sólo nos sirve para adaptar una clase Adaptable concreta, no toda las subclases de Adaptable
  - Adaptador, como subclase de Adaptable, puede redefinir parte de la clase heredada
  - Introduce un solo objeto, no necesita ninguna referencia a un objeto por composición
- Adaptador de Objetos
  - Un mismo Adaptador puede funcionar como múltiples Adaptables (sus subclases)
  - Adaptador puede añadir funcionalidad a todos los adaptables
  - Introduce un objeto aparte del Adaptable

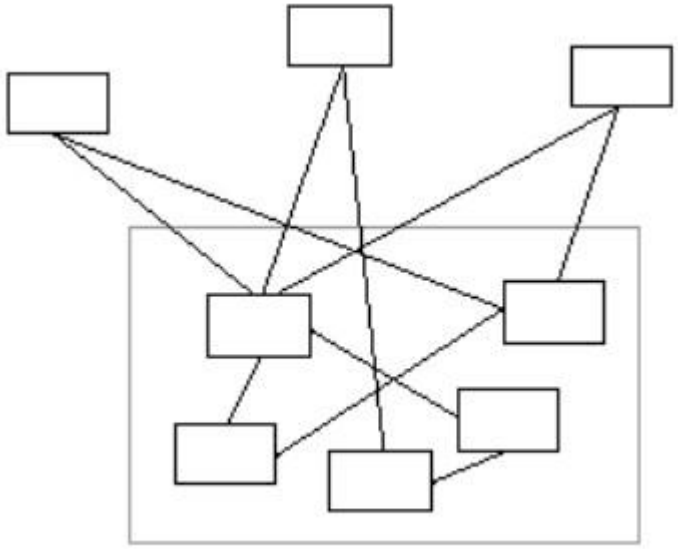
Facade – Patrón Estructural (Fachada)

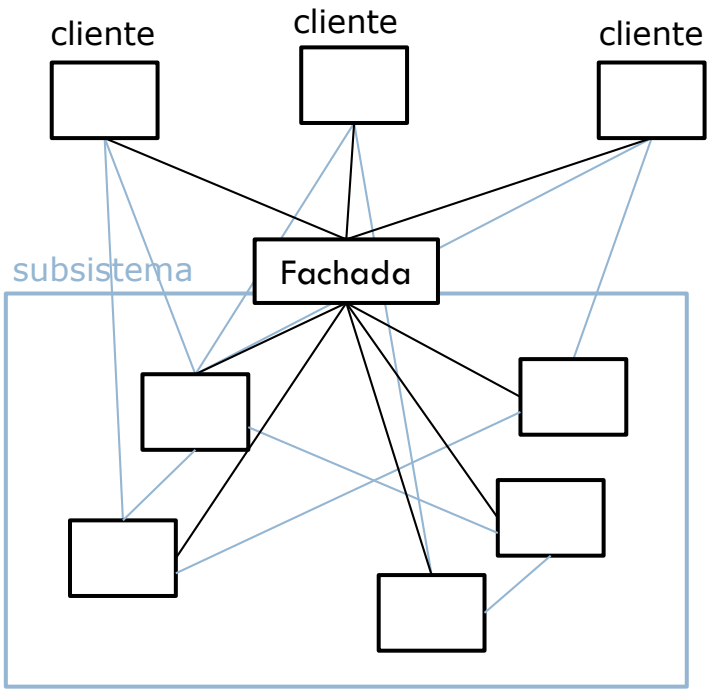
- Propósito
  - Proporciona una interfaz unificada a un conjunto de interfaces en un subsitema
  - Define una interfaz de alto nivel que facilita el uso del subsistema



Facade – Patrón Estructural (Fachada)

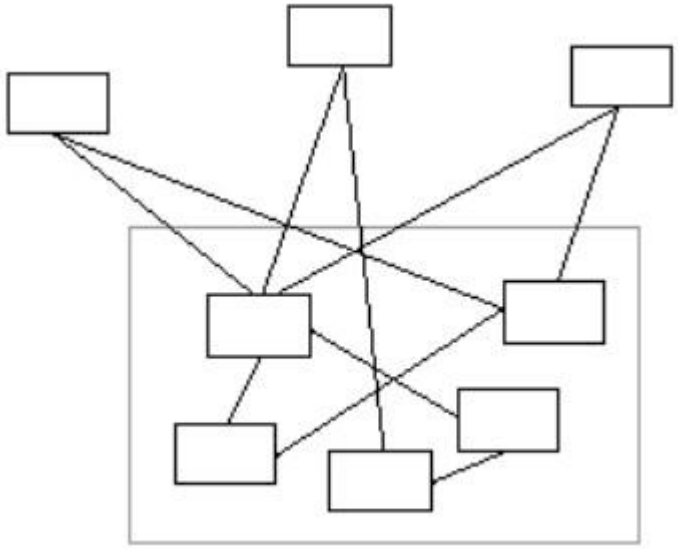
- Aplicabilidad
  - Se quiere proveer de una simple interfaz a un subsistema complejo
  - Hay muchas dependencias entre los clientes y las clases del subsistema

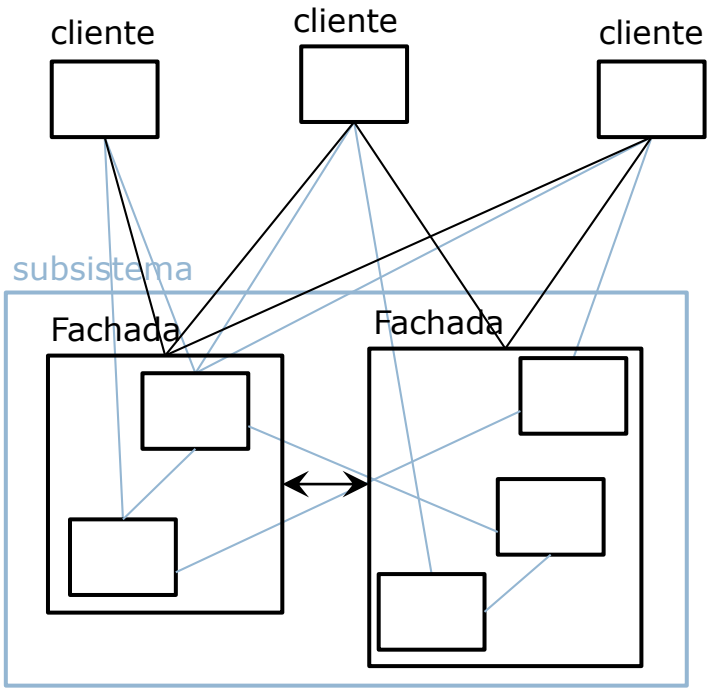




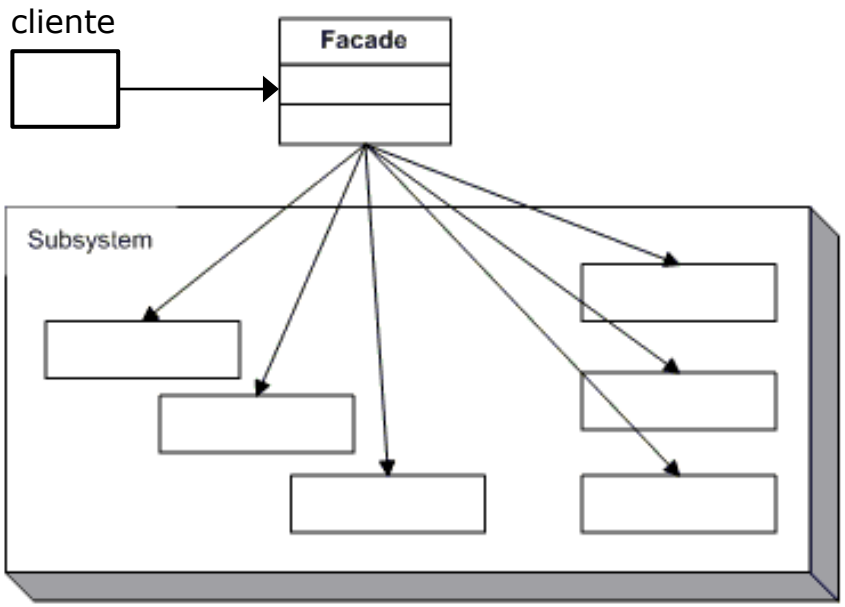
Facade – Patrón Estructural (Fachada)

- Aplicabilidad
  - Se quiere simplificar el subsistema
    - Dividiéndolo en una o más capas que definan un punto de entrada único para cada nivel del subsistema





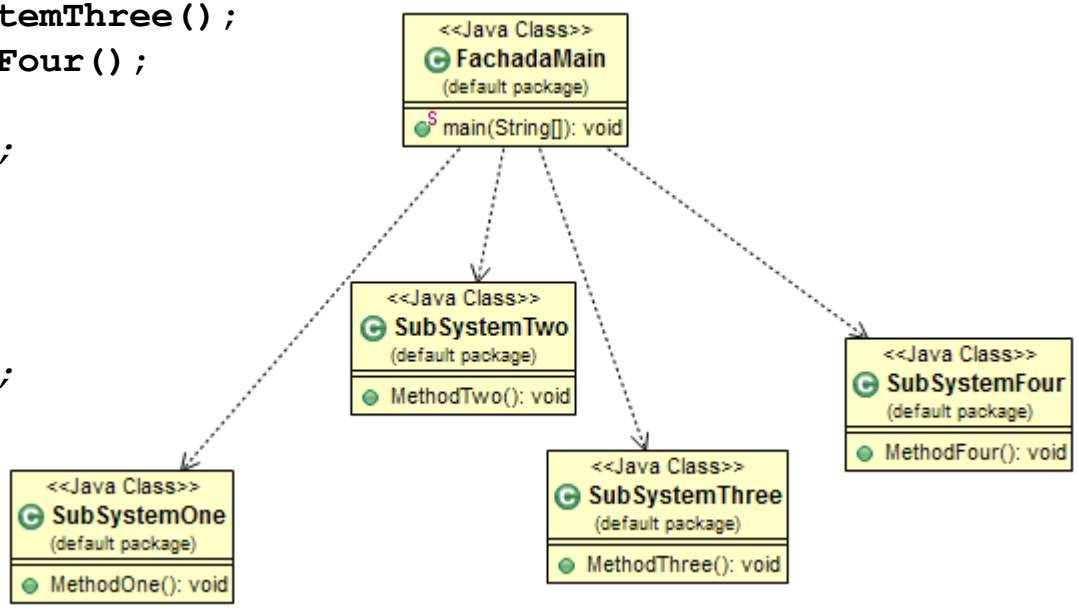
Facade – Estructura



- Participantes
  - Clientes
    - Hacen uso del subsistema
  - Facade
    - Redirige las peticiones del cliente a las clases correspondientes
    - Trabajo adicional de adaptación de interfaces
  - Clases del subsistema
    - Implementan la funcionalidad del sistema
    - No distinguen a Facade de cualquier otro cliente

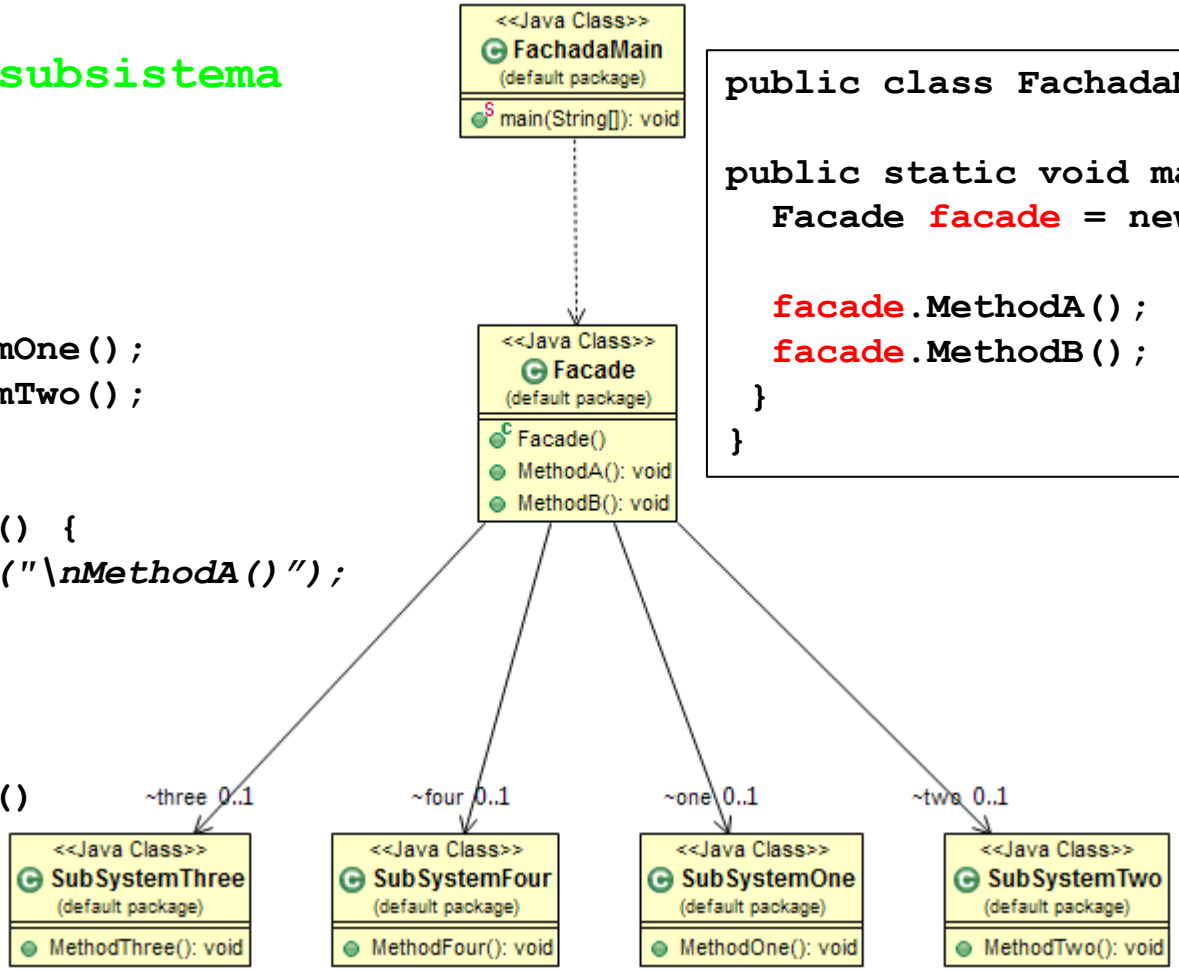
Facade – Ejemplo (sin Fachada)


```
// Clase Cliente
public class FachadaMain {

public static void main(String[] args) {
  // objetos del subsistema
  SubSystemOne one = new SubSystemOne();
  SubSystemTwo two = new SubSystemTwo();
  SubSystemThree three = new SubSystemThree();
  SubSystemFour four= new SubSystemFour();

  System.out.println("\nMethodA()");
  one.MethodOne();
  two.MethodTwo();
  four.MethodFour();

  System.out.println("\nMethodB()");
  two.MethodTwo();
  three.MethodThree();
 }
}
```



Facade – Ejemplo (con Fachada)


```
class Facade{
  // objetos del subsistema
  SubSystemOne one;
  SubSystemTwo two;
  …
  public Facade()
  {
   one = new SubSystemOne();
   two = new SubSystemTwo();
   …
  }
  public void MethodA() {
   System.out.println("\nMethodA()");
   one.MethodOne();
   two.MethodTwo();
   four.MethodFour();
  }
  public void MethodB()
  { … }
```




```
public class FachadaMain {

public static void main(…) {
   Facade facade = new Facade();

   facade.MethodA();
   facade.MethodB();
  }
}
```

Facade - Consecuencias

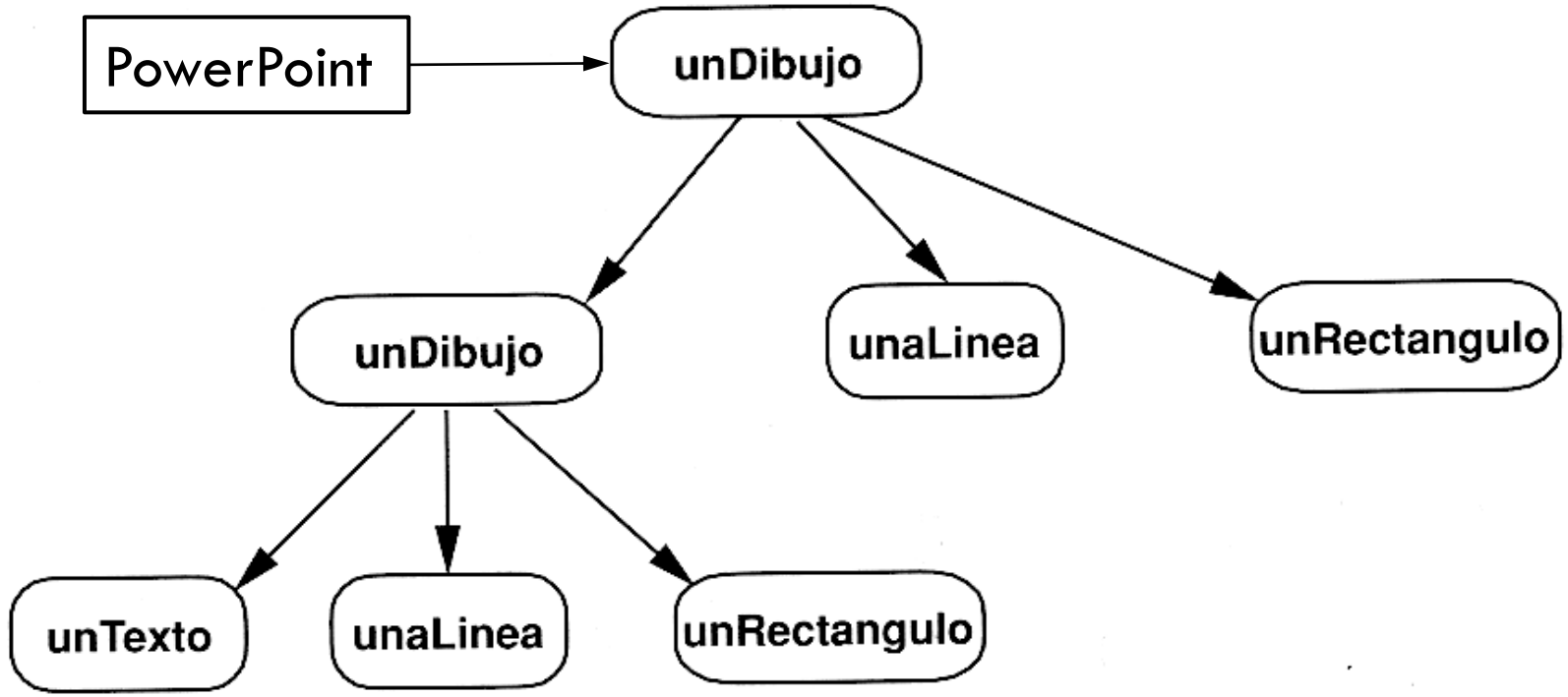
- Separa al cliente de los componentes del subsistema
  - El subsistema es más fácil de usar
- Favorece el bajo acoplamiento entre el subsistema y el cliente
  - Ayuda a estructurar el software en capas y sus dependencias
  - Facilita la portabilidad entre plataformas (se reimplementa sólo las capas necesarias)
    - Ej: Gestor Música: Colecciones, Ficheros, Acceso a disco
  - Permite identificar dependencias circulares o complejas
- No impide el acceso directo de los clientes a los objetos del subsistema

Composite – Patrón Estructural (Compuesto)

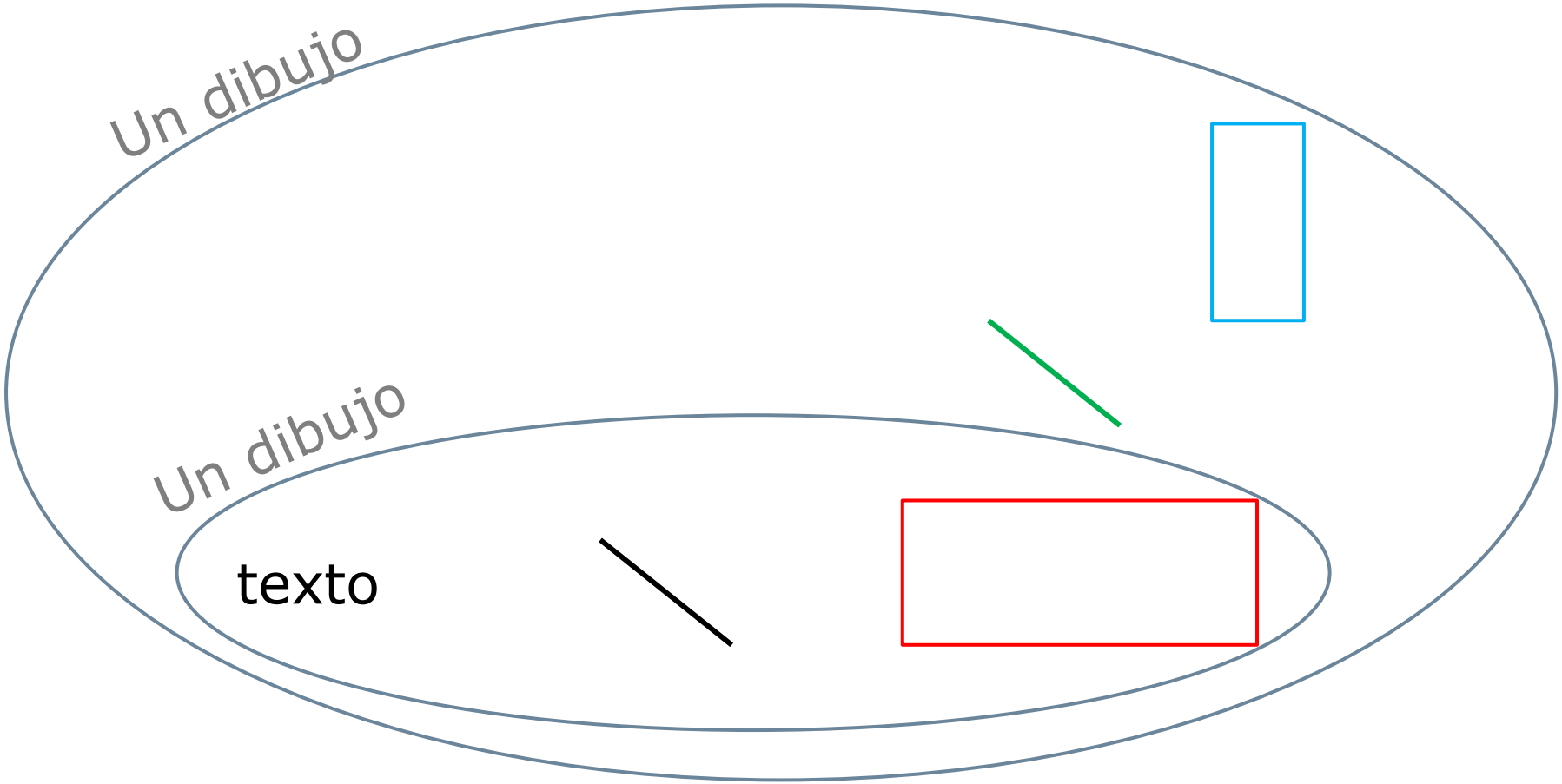
- Propósito
  - Compone objetos en estructuras de árbol para representar jerarquías parte-todo
    - Composición recursiva
  - Permite que los clientes traten de manera uniforme a los objetos individuales y a los compuestos
- Aplicabilidad
  - Representar jerarquías parte-todo
  - Los clientes no ven diferencias entre composiciones de objetos y objetos individuales: todos los objetos se tratan de igual forma

Composite - Motivación

- Gestión de objetos gráficos (Ej: Dibujos en PowerPoint)
  - Representación de objetos Gráficos compuestos
  - Estructura de objetos compuestos recursivamente

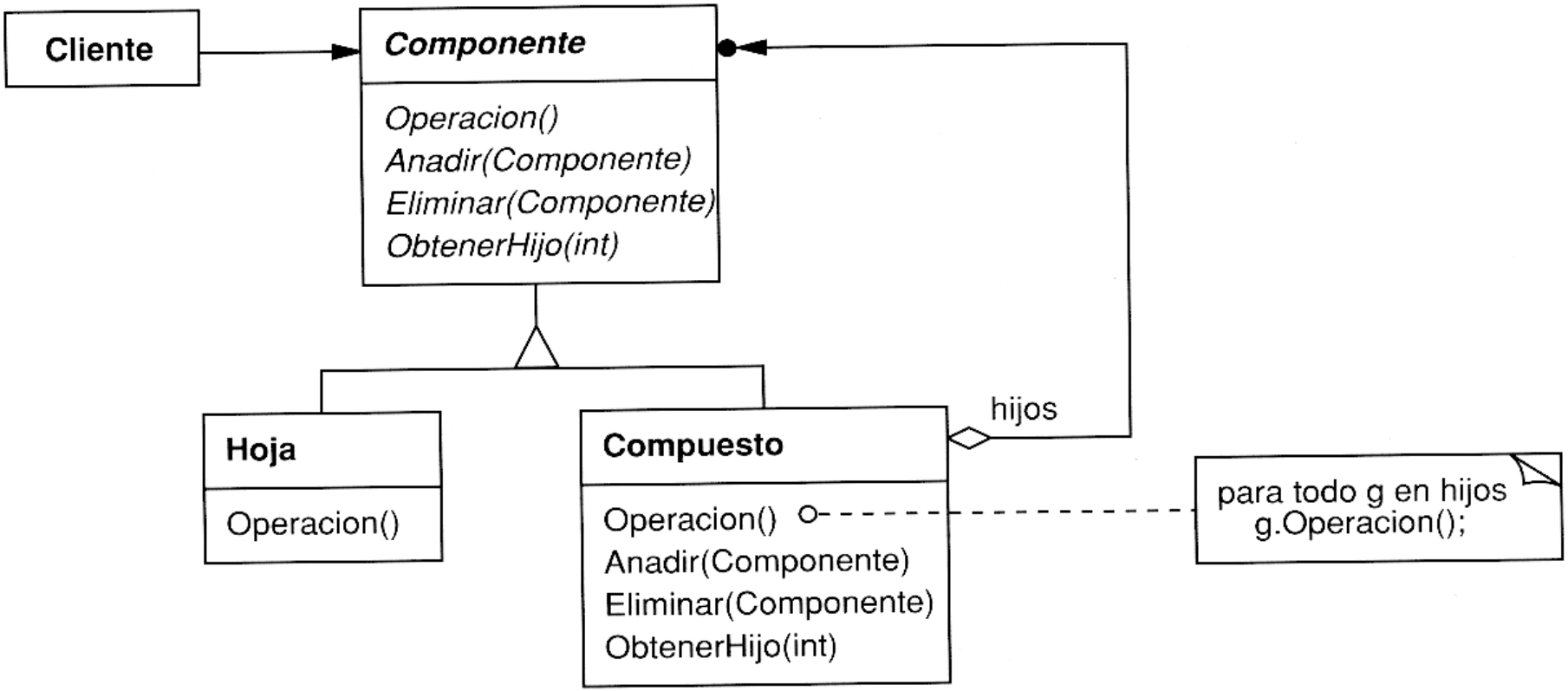


Ejemplo: Dibujo en PowerPoint



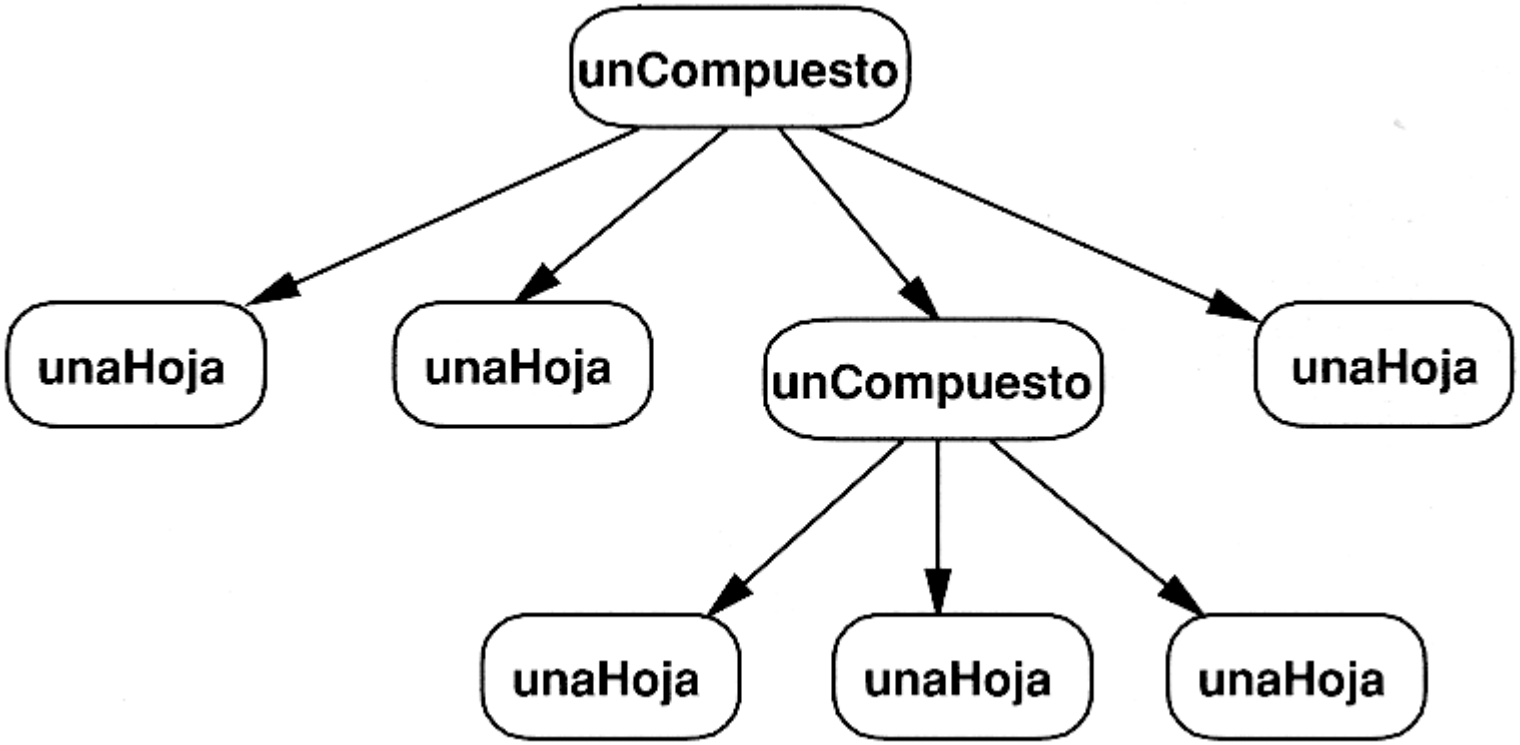
Composite - Estructura

- Estructura de clases y participantes



Composite – Estructura

- Ejemplo de estructura de objetos compuestos típica

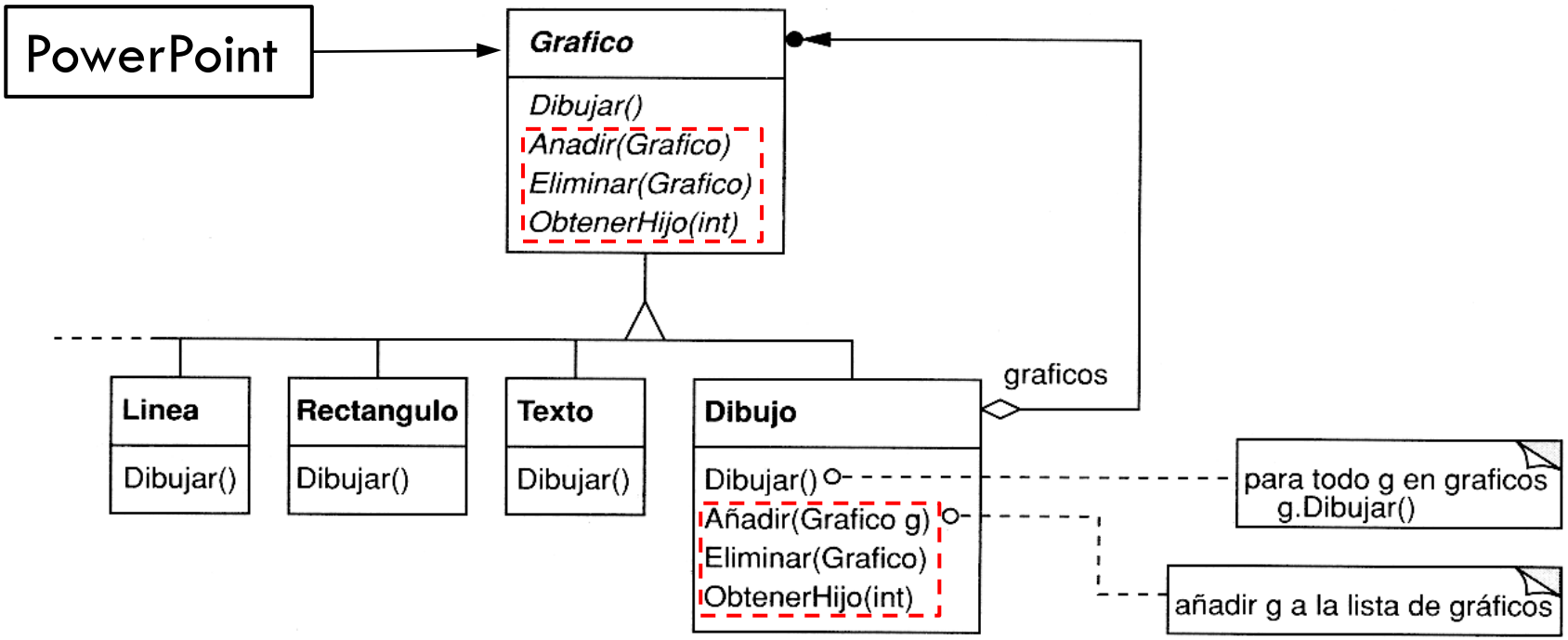


Composite – Participantes

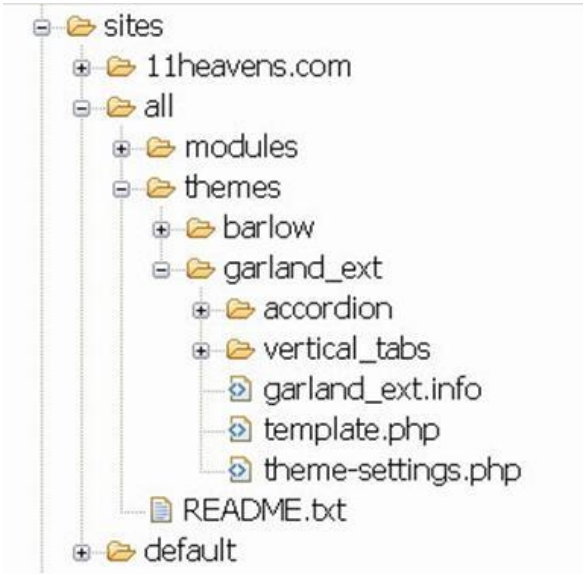
- Componente (Gráfico)
  - Declara la Interfaz de los objetos de la composición
  - Implementa comportamiento predeterminado de la interfaz común a todos los objetos
  - Define interfaz para acceder a nodos hijos y gestionarlos
  - (opcional) interfaz para acceder al nodo padre
- Hoja (Rectángulo, Línea, Texto, etc.)
  - Representa objetos finales, sin hijos
- Compuesto (Dibujo)
  - Define interfaz para acceder a nodos hijos y gestionarlos
  - Implementa operaciones interfaz gestión de nodos hijos
  - Almacena nodos hijos
- Cliente
  - Manipula objetos compuestos a través de la interfaz **Componente**

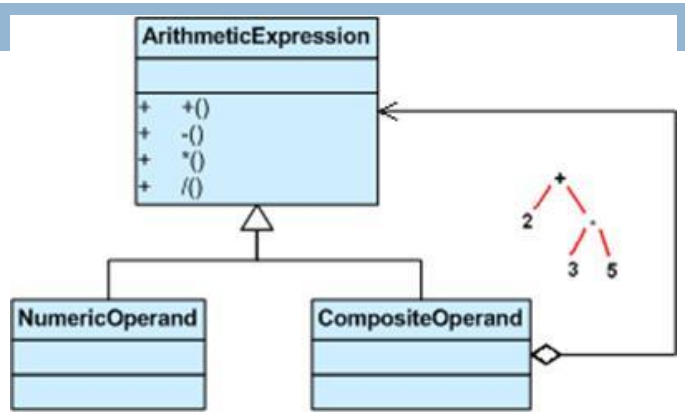
Composite – Aplicado a Gráficos

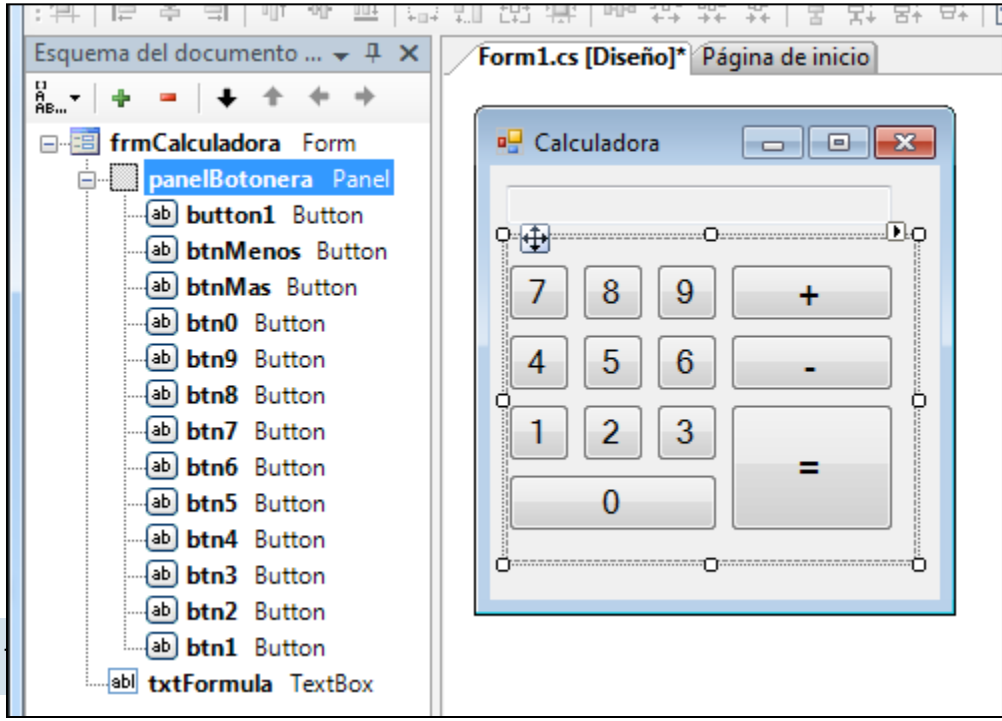
- Representación de objetos Gráficos compuestos
  - Estructura de objetos compuestos recursivamente



Composite – ¿Otros Ejemplos?







Composite – Consecuencias

- Define jerarquías de clases formadas por objetos primitivos y compuestos.
  - Los objetos primitivos pueden ser sustituidos por objetos compuestos por otros objetos primitivos o compuestos.
- Simplifica el cliente
  - Tratan de forma uniforme a todos los componentes de la composición por igual (la misma interfaz)
- Facilita añadir nuevos tipos de componentes
  - Basta con heredar de la clase componente Compuesto u Hoja
- Riesgo de excesiva generalidad en el diseño
  - No es posible restringir el tipo de cierto componentes de un determinado compuesto