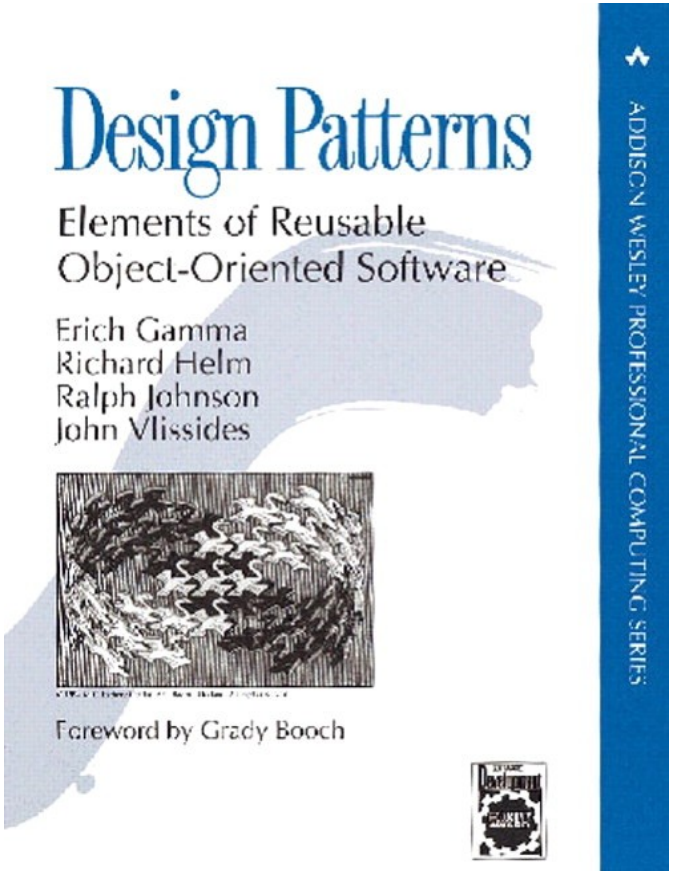
Tema 7. Patrones de Diseño en Java

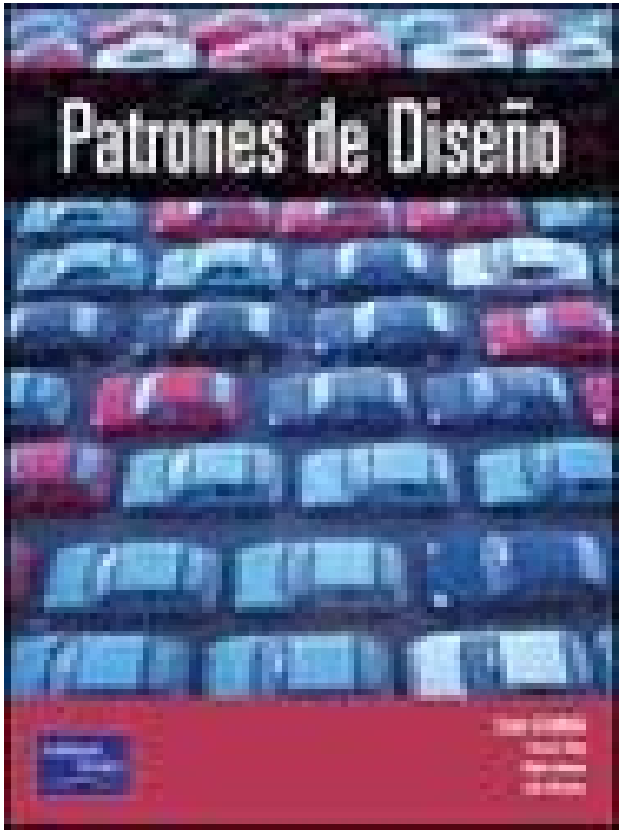
Metodología y Desarrollo de Programas

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)

Introducción

- Diseñar software es una tarea MUY difícil y diseñarlo correctamente para que se reutilizable es aún más difícil.
- Los problemas de diseño suelen repetirse proyecto tras proyecto.
- Para ayudarnos en esta tarea se utilizan patrones.

- **Definición**: Un patrón es una solución probada que se puede aplicar con éxito a un determinado tipo de problemas que aparece con frecuencia
- **Definición del arquitecto Christopher Alexander** "Cada patrón describe un problema que ocurre infinidad de veces en nuestro entorno, así como la solución al mismo, de tal modo que podemos utilizar esta solución un millón de veces más adelante sin tener que volver a pensarla otra vez."

Introducción

- En informática surgió a partir de la publicación del libro Design Patterns escrito por el grupo Gang of Four (GoF) compuesto por Erich Gamma, Richard Helm, Ralph Johnson y John Vlisides, en el que se recogían 23 patrones de diseño comunes.
- **Definición**[Gammaetal.95]:
  - Un patrón de diseño es una descripción de clases y objetos comunicándose entre sí adaptada para resolver un problema de diseño general en un contexto particular.

Introducción

- Los patrones de diseño pretenden:
  - Proporcionar catálogos de elementos reusables en el diseño de sistemas software.
  - Evitar la reiteración en la búsqueda de soluciones a problemas ya conocidos y solucionados anteriormente.
  - Formalizar un vocabulario común entre diseñadores.
  - Estandarizar el modo en que se realiza el diseño.
  - Facilitar el aprendizaje de las nuevas generaciones de diseñadores condensando conocimiento ya existente.
- Asimismo, no pretenden:
  - Imponer ciertas alternativas de diseño frente a otras.
  - Eliminar la creatividad inherente al proceso de diseño.

- Herencia vs Composición
  - Herencia
    - ✓ Herencia es una relación entre clases
    - ✓ Se trata de una relación unidireccional, es decir, un PDI siempre es una persona pero no al revés
    - ✓ Gracias al polimorfismo se consigue que cualquiera de las clases hijas pueden adoptar a la clase padre.
    - ✓ Ventajas
      - Mecanismo más sencillo para reutilizar código
      - Permite utilizar polimorfismo con facilidad que ello supone.
      - Permite sobrescribir métodos, así que si algo no es exactamente como espero siempre puedo cambiarlo si fuera necesario, por lo que es versátil.
      - La herencia usando algo parecido al patrón de diseño Template Method , permite crear clases abstractas consiguiendo un desarrollo guiado por la plantilla creada, ahorrando así tiempo.

Introducción. Antes de empezar...

- Herencia vs Composición
  - Pero ….
    - ✓ La herencia es una **dependencia fuerte** entre clases. La clase hija depende fuertemente de la clase padre y la necesita para funcionar siendo además excluyente. Es más, cualquier cambio en la clase padre puede tener efectos inesperados en cualquiera de las clases hijas
      - ¿Qué sucede si objeto que es de una determinada clase hija pasa a otra? Por ejemplo un estudiante es contratado como PDI.
      - **No se debe usar herencia** para describir una relación IS-A si se prevé que los componentes puedan cambiar en tiempo de ejecución
    - ✓ En ocasiones no se sabe cómo está implementada la clase padre lo que no te permite conocer exactamente lo que hace (luego llegan las sorpresas).

Introducción. Antes de empezar...

- Herencia vs Composición
  - Composición
    - ✓ Composición es una relación entre objetos
    - ✓ Permite la creación de objetos de una clase a partir de objetos de otra clase.
    - ✓ Ventajas
      - Con ello se consigue **delegar** las tareas que nos mandan a hacer a aquella pieza de código que sabe hacerlas
      - Cualquier objeto puede ser reemplazado por otro en tiempo de ejecución siempre que sea del mismo tipo.
    - ✓ Desventajas
      - Una mayor complejidad en el desarrollo del sistema.
  - Muchos de los patrones se basan en la **delegación**, un objeto receptor delega las operaciones en su delegado (strategy, state, visitor, ...)

Introducción. Antes de empezar...

- Causas comunes de rediseño
  - Crear un objeto indicando la clase.
  - Dependencia de operaciones específicas
  - Dependencia de plataformas hardware o software
  - Dependencia sobre representación de objetos.
  - Dependencias de algoritmos
  - Acoplamiento fuerte entre clases
  - Extender funcionalidad mediante subclases
  - Incapacidad de cambiar clases convenientemente

Introducción

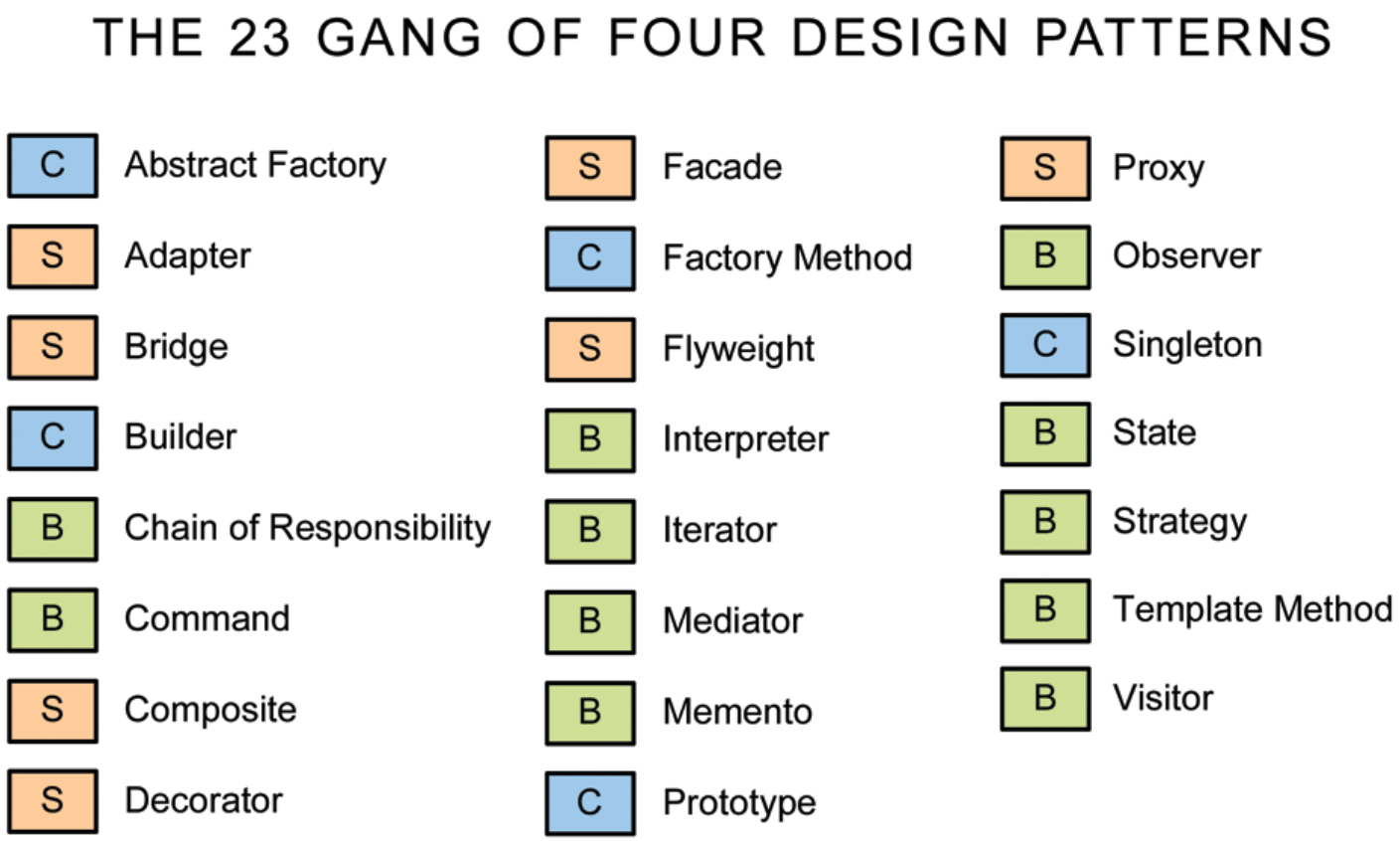
- Elementos esenciales de un patrón
  - **Nombre** se utiliza para describir un problema de diseño.
  - El **problema** describe cuando aplicar el patrón. Descripción de un problema y su contexto.
  - La **solución** describe los elementos que conforman el diseño, sus relaciones responsabilidades y colaboraciones. El patrón proporciona una descripción abstracta de un problema de diseño y como se organizan esos elementos (clases&objetos) para resolverlo.
  - **Consecuencias** son los resultados de aplicar un patrón, a menudo hacen referencia a factores como el espacio de almacenamiento y al tiempo de ejecución. Incluyen el impacto de propiedades del sistema como flexibilidad, portabilidad y extensibilidad.

Clasificación de los patrones de diseño

- Tipos de patrones
  - **Creacionales** relacionados con el proceso de creación de objetos.
  - **Estructurales** relacionado con la composición de clases u objetos.
  - **Comportamiento** caracterizan la forma en que las clases y los objetos colaboran para proporcionar una funcionalidad.
- Ámbito: indica si el patrón se aplica principalmente a clases o a objetos
  - Patrones de clases indica relaciones entre clases y subclases. Estas relaciones se realizan en tiempo de compilación, son estáticas.
  - Patrones de objetos indica relaciones entre objetos. Estas relaciones pueden cambiar en tiempo de ejecución, son dinámicas.

Clasificación de los patrones de diseño

- Catálogo de patrones GoF



Clasificación de los patrones de diseño

- 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 creacionales

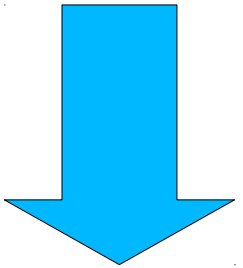
- Los patrones de creación muestran la guía de cómo crear objetos cuando sus creaciones requieren tomar decisiones. Estas decisiones normalmente serán **resueltas dinámicamente** decidiendo que clases instanciar o sobre que objetos un objeto delegará responsabilidades
- A menudo hay **varios patrones de creación que puedes aplicar en una situación**.
- Algunas veces se pueden combinar múltiples patrones. En otros casos se debe elegir entre los patrones.

Patrones creacionales

- Ejemplo de una clase para escribir en un único fichero logs los movimientos de un banco

```
public class Logger {
  private final String logFile = "demo_log.txt";
  private PrintWriter writer;

  public Logger() {
    try {
            FileWriter fw = new FileWriter(logFile);
            writer = new PrintWriter(fw, true);
    } catch (IOException e) {
    }
  }

  public void logWithdraw (String account, double amount) {
        writer.println("WITHDRAW ("
            + account + "): " + amount + "$");
  }

   public void logDeposit (String account, double amount) {
        writer.println("DEPOSIT ("
            + account + "): " + amount + "$");
  }

   public void logTransfer (String fromAccount, String toAccount,
            double amount) {
        writer.println("TRANSFER ("
           + fromAccount + "->" + toAccount + "): " + amount + "$");
    }

}
```

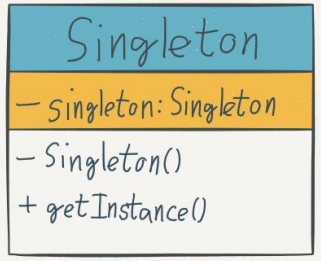
```
public class MainLogger {
  public static void main(String []args) {
    Logger logger1 = new Logger();
    Logger logger2 = new Logger();
    Logger logger3 = new Logger();

    logger1.logDeposit("0001", 80.5);
    logger2.logWithdraw("0002", 100);
    logger1.logTransfer("0001", "0003", 40);
    logger3.logDeposit("0004", 56.74);
    logger2.logWithdraw("0005", 30);
  }
}
```



```
DEPOSIT (0004): 56.74$
RWITHDRAW (0005): 30.0$
.0$
```

Ejemplo obtenido de: https://medium.com/@p.osinaga/using-singleton-in-java-b1b78cf640ed

- Singleton –Instancia Única (1/3)
  - Objetivo: Garantiza que solamente **se crea una instancia de la clase** y provee un punto de acceso global
  - Aplicabilidad:
    - ✓ Debe haber exactamente una instancia de la clase.
    - ✓ La única instancia de la clase debe ser accesible para todos los clientes de esa clase.
  - Solución
    - ✓ Una clase Singleton tiene **una variable static** que se refiere a la única instancia de la clase que quieres usar.
    - ✓ Esta instancia es creada cuando la clase es cargada en memoria, por tanto los **constructores de la clase son privados**.
    - ✓ **Para acceder** a la única instancia de una clase Singleton, la clase proporciona **un método static**, normalmente llamado getInstancia(), el cual retorna una referencia a la única instancia de la clase.



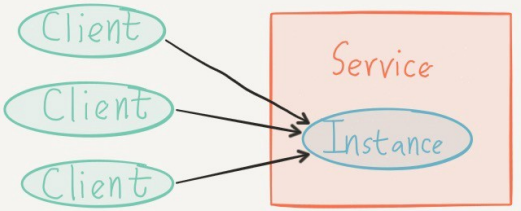


Imagen obtenida de : https://medium.com/@p.osinaga/using-singleton-in-java-b1b78cf640ed

- Singleton –Instancia Única (2/3)
  - Consecuencias
    - ✓ Existe exactamente una instancia de una clase Singleton.
    - ✓ Otras clases que quieran una referencia conseguirán esa instancia llamando al método static getInstancia() de la clase.
    - ✓ Java no permite sobrescribir los métodos static.
  - Usos en el API de Java
    - ✓ En el API de Java la clase java.lang.Runtime es una clase Singleton, tiene exactamente una única instancia de la clase. No tiene constructores públicos. Para conseguir una referencia a su única instancia, otras clases deben llamar a su método static getRuntime.
  - Patrones relacionados
    - ✓ Suele usarse con los patrones Abstract Factory, Builder, y Prototype.

- Singleton –Instancia Única (3/3)
  - Implementación

```
public class Singleton {
    private static Singleton INSTANCE = new Singleton();

    // El constructor privado no permite que se genere un
    // constructor por defecto con mismo modificador de acceso que
    //la definición de la clase)
    private Singleton() {}

    public static Singleton getInstance() {
        return INSTANCE;
    }
}
```

Singleton –Instancia Única (3/3)

- Implementación con hilos (problemas de sincronización)

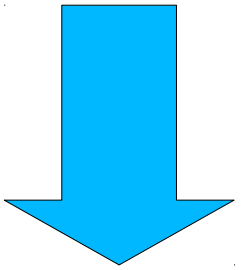
```
public class Singleton {
    private static Singleton INSTANCE = null;

    // Private constructor suppresses
    private Singleton() {}

    // creador sincronizado para protegerse de posibles problemas  multi-
    // hilo
    private synchronized static void createInstance() {
            INSTANCE = new Singleton();
    }

    public static Singleton getInstance() {
        if (INSTANCE == null) createInstance();
        return INSTANCE;
    }
}
```

Singleton –Ejemplo de uso

- **Solución:** El fichero logs es un auténtico caos → Una única instancia

```
public class LoggerSingleton {
  private final String logFile = "demo_log.txt";
  private PrintWriter writer;
  private static LoggerSingleton logger = null;

  private LoggerSingleton() {
    try {
            FileWriter fw = new FileWriter(logFile);
            writer = new PrintWriter(fw, true);
    } catch (IOException e) {
    }
  }
  public static synchronized LoggerSingleton getInstance(){
    if(logger == null)
        logger = new LoggerSingleton();
    return logger;
  }
  public void logWithdraw (String account, double amount) {
        writer.println("WITHDRAW ("
            + account + "): " + amount + "$");
    }
  public void logDeposit (String account, double amount) {
        writer.println("DEPOSIT ("
            + account + "): " + amount + "$");
   }
   public void logTransfer (String fromAccount, String toAccount,
            double amount) {
        writer.println("TRANSFER ("
           + fromAccount + "->" + toAccount + "): " + amount + "$");
    }
}
```

```
public class MainLogger {
public static void main(String []args) {
   LoggerSingleton logger1 =
            LoggerSingleton.getInstance();
   LoggerSingleton logger2 =
            LoggerSingleton.getInstance();
   LoggerSingleton logger3 =
            LoggerSingleton.getInstance();

    logger1.logDeposit("0001", 80.5);
    logger2.logWithdraw("0002", 100);
    logger1.logTransfer("0001", "0003", 40);
    logger3.logDeposit("0004", 56.74);
    logger2.logWithdraw("0005", 30);
  }
}
```

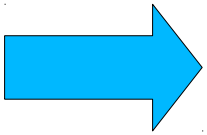


```
DEPOSIT (0001): 80.5$
WITHDRAW (0002): 100.0$
TRANSFER (0001->0003): 40.0$
DEPOSIT (0004): 56.74$
WITHDRAW (0005): 30.0$
```

Ejemplo obtenido de: https://medium.com/@p.osinaga/using-singleton-in-java-b1b78cf640ed

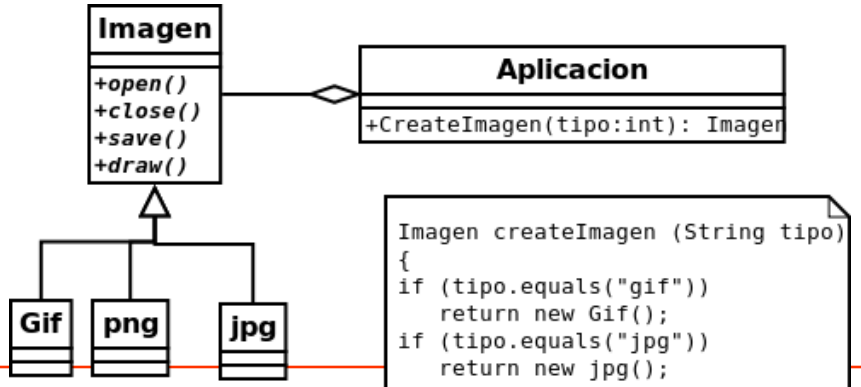
- Factory Method –Método fábrica (1/4)
  - Objetivo: Una clase **que implementa uno o más métodos de creación**, que son los métodos **que se encargan de crear instancias** de objetos (estas instancias pueden ser de esta misma clase o de otras)
  - Motivación
    - ¿Cómo se crean los objetos vehículo en tiempo de ejecución del desguace? ¿Y si se necesitarán dos objetos más y no se puede tocar dicho código?

```
int opcion=t.leerEntero();
Vehiculo v=null;
if (opcion==1)
   v= new Coche();
else if (opcion==2)
   v= new Camion();
else
   v= new Moto();
}
```



```
public class FactoriaVehiculo {

public Vehiculo buildVehiculo(String tipo) {
  if (tipo.equals("Coche")) {
     return new Coche();
  } else if (tipo.equals("Camion")) {
     return new Camion();
  }else return new Moto();
 }

}
```

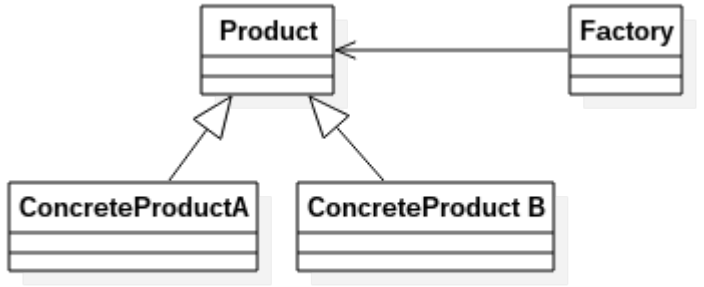
- Factory Method –Método fábrica (2/4)
  - Aplicabilidad:
    - ✓ Una clase no puede anticipar la clase de objetos que debe crear (**creación en tiempo de ejecución**) –> mecanismo para crear objetos sin new (hay métodos para ello)
    - ✓ Una clase necesita que **sean sus subclases quienes especifiquen el tipo de objetos a crear**
    - ✓ Un conjunto de clases delegan una cierta responsabilidad en una serie de clases auxiliares
  - Solución:
    - ✓ La solución para esto es hacer un método (el método de la fábrica) que se define en el creador. Este método abstracto se define para que devuelva un producto. Las subclases del creador pueden sobrescribir este método para devolver subclases apropiadas del producto...



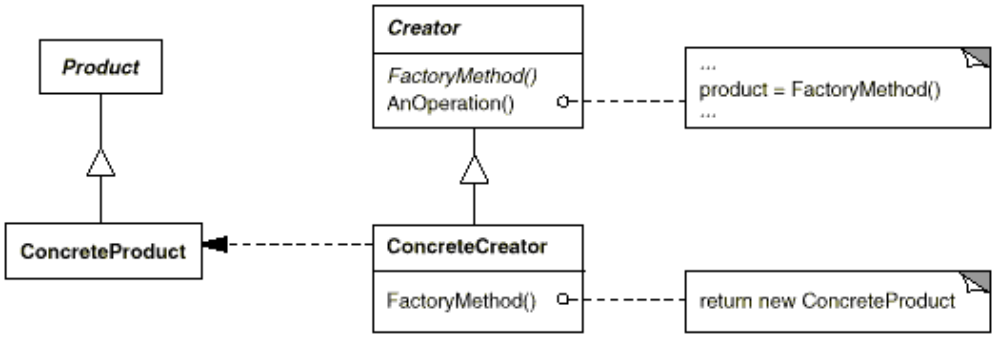
Factory Method –Método fábrica (3/4)

- Consecuencias
  - ✓ Permite **escribir algoritmos genéricos que funcionan con distintos Objetos**
  - ✓ Un inconveniente es que puede obligar a crear subclases artificiales
  - ✓ Permite separar una jerarquía compleja en dos jerarquías paralelas
- Usos en el API de Java
  - ✓ URLConnection para leer los bytes de una URL.
  - ✓ Procesadores SAX XML.
- Patrones relacionados
  - ✓ Suele usarse con los patrones Abstract Factory, Template Method, y Prototype.

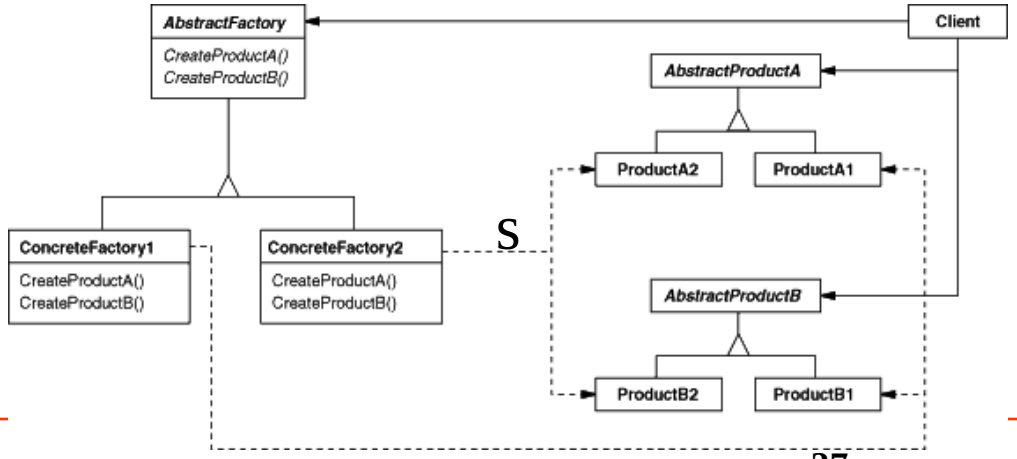
- Factory Method –Método fábrica (4/4). Tres tipos:
  - **Simple Factory**: Clase utilizada para crear nuevas instancias de objetos.



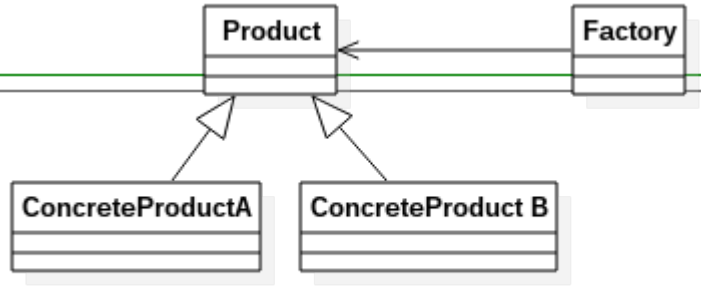
  - **Factory Method**: Define una interfaz para crear objetos pero deja que sean las subclases las que deciden qué clases instanciar



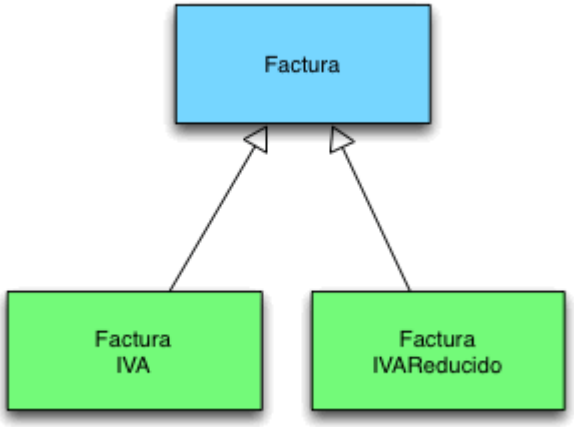
  - **Abstract Factory**: Proporciona una interfaz para crear familias de objetos relacionados o que dependen entre sí, sin especificar sus clases concretas



Patrones creacionales.

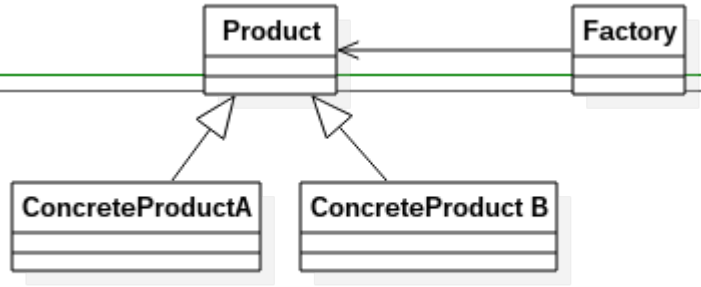


- Ejemplo de **Simple Factory**: Diagrama de clases y se desea implementar una factoría de Facturas

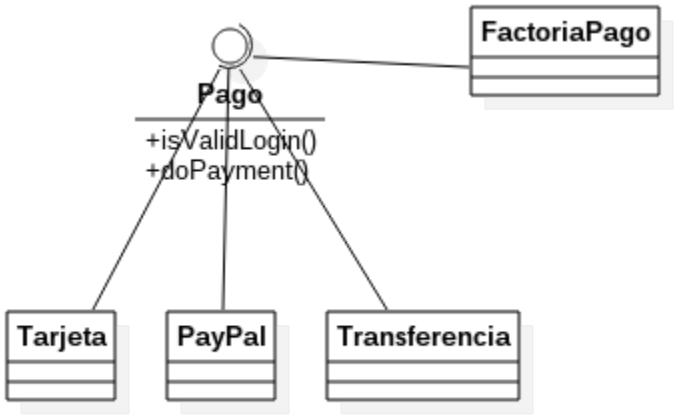



```
public class FactoriaFacturas {
 public Factura getFactura(String tipo) {
  if (tipo.equals("iva")) {
          return new FacturaIva();
  } else {
          return new FacturaIvaReducido();
  }
 }
}

public static void main(String[] args) {
 FactoriaFacturar fact= new FactoriaFacturas();
 Factura f=fact.getFactura("iva");
 f.setId(1);
 f.setImporte(100);
 System.out.println(f.getImporteIva());
 }
```

Patrones creacionales.



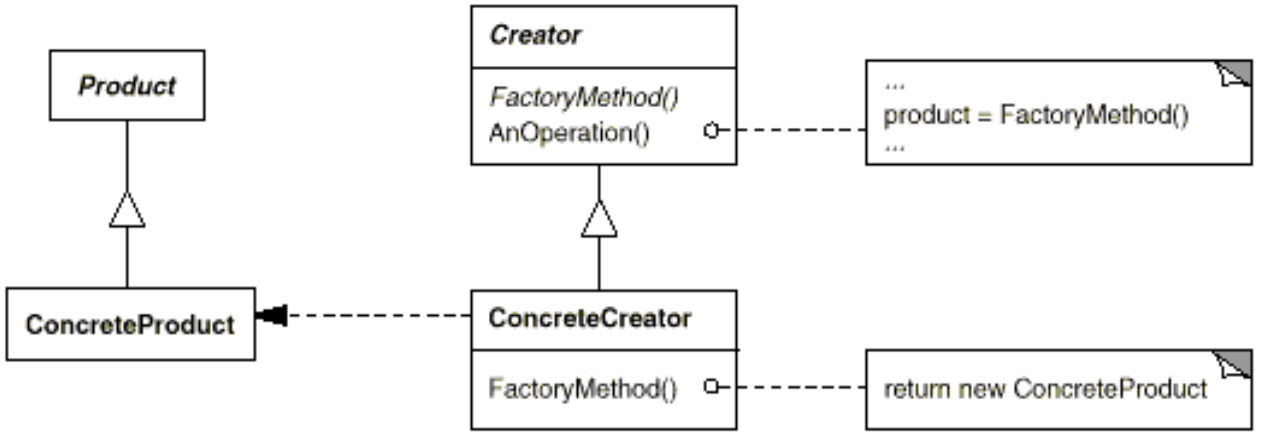
- Ejemplo de **Simple Factory**: Diagrama de clases y se desea implementar una factoría de Pago




```
public class PaymentFactory {

  public enum TypePayment {CARD, PAYPAL, WIRE_TRANSFER}

   public static Payment getPayment(TypePayment typePayment) {
     switch (typePayment) {
          case CARD: return new CardPayment();
          case PAYPAL: return new PaypalPayment();
          case WIRE_TRANSFER: return new WireTransferPayment();
          default: return new CardPayment();
      }
   }
}
public class MainPayment {

    private Payment payment;

    public MainPayment(PaymentFactory.TypePayment typePayment){
        this.payment = PaymentFactory.getPayment(typePayment);
    }
    public static void main(String []args) {
          MainPayment m= new MainPayment(TypePayment.CARD);
          Payment.?????
          //TODO
    }
```

Patrones creacionales.

- Ejemplo de **Factory Method**: delega la creación de la instancia a las subclases, pero sabe que hacer con el objeto creado.




```
abstract class Creator{
    // Definimos método abstracto
    public abstract Product factoryMethod();
}
```

Ahora definimos el creador concreto.

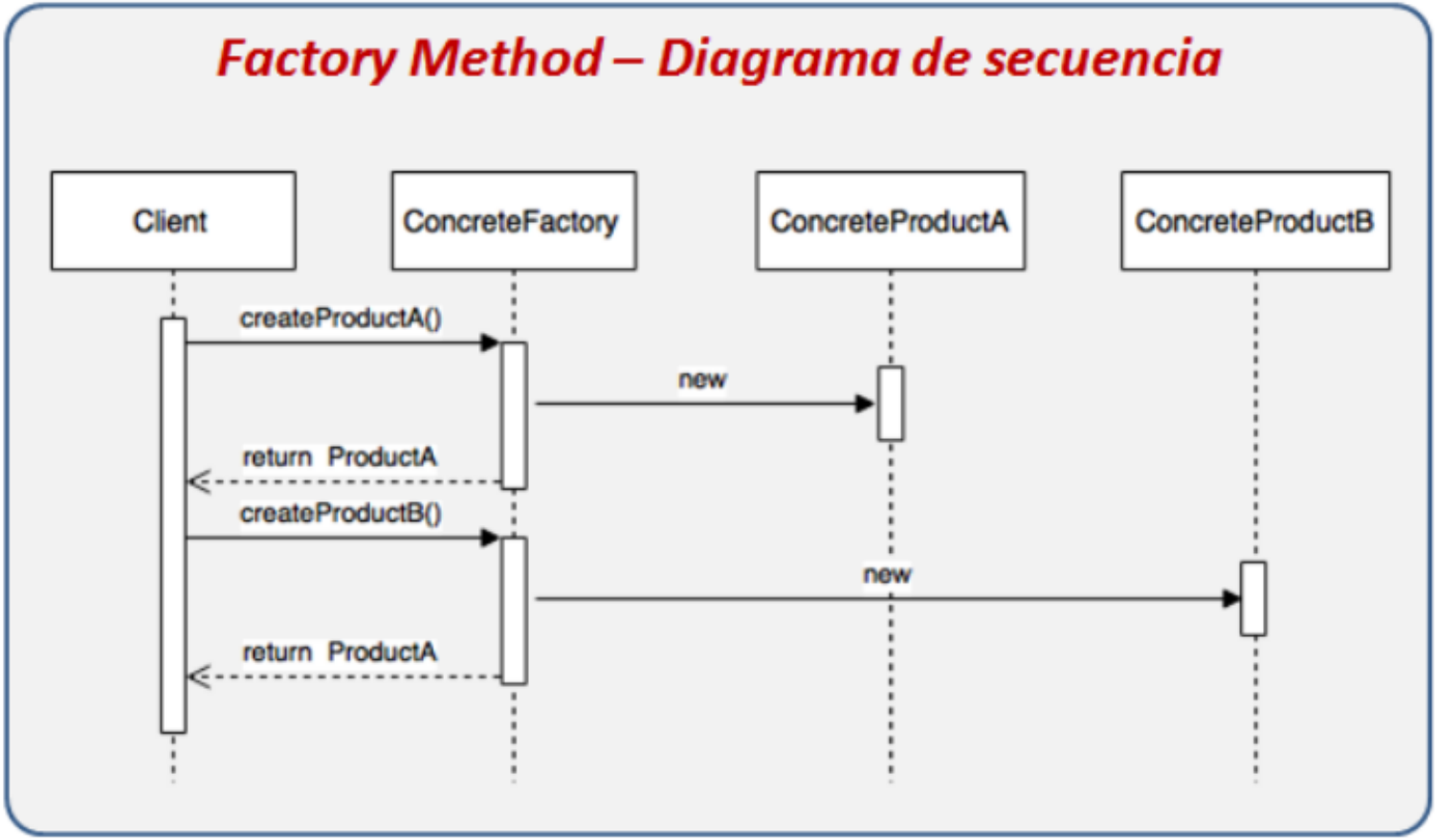
```
public class ConcreteCreator extends Creator{
    public Product factoryMethod() {
        return new ConcreteProduct();
    }
}
```

```
public interface Product{
    public void operacion();
}

public class ConcreteProduct implements Product{
    public void operacion(){
        System.out.println("Una operación de este producto");
    }
}
```

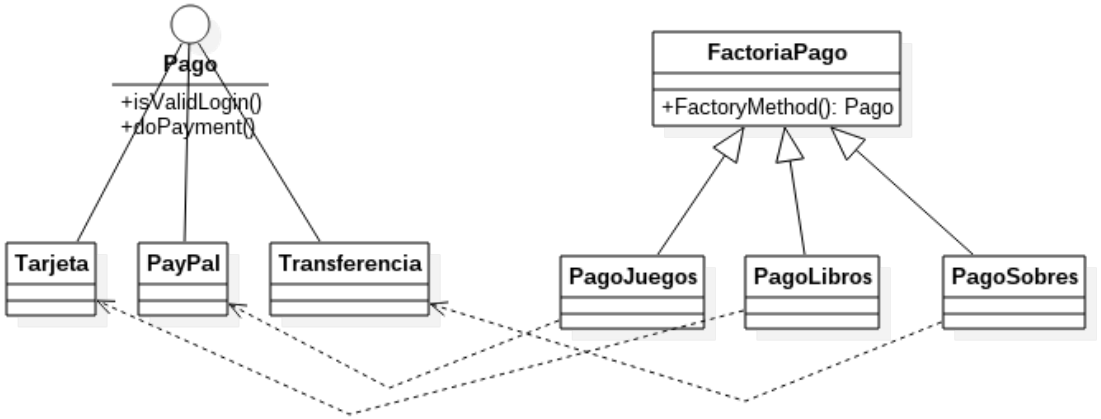
Y un ejemplo de uso :

```
public static void main(String args[]){
    Creator aCreator;
    aCreator = new ConcreteCreator();
    Product producto = aCreator.factoryMethod();
    producto.operacion();
}
```

- Factory Method: delega la creación de la instancia a las subclases, pero sabe que hacer con el objeto creado.



- Ejemplo de **Simple Factory**: Diagrama de clases y se desea implementar una factoría de Pago



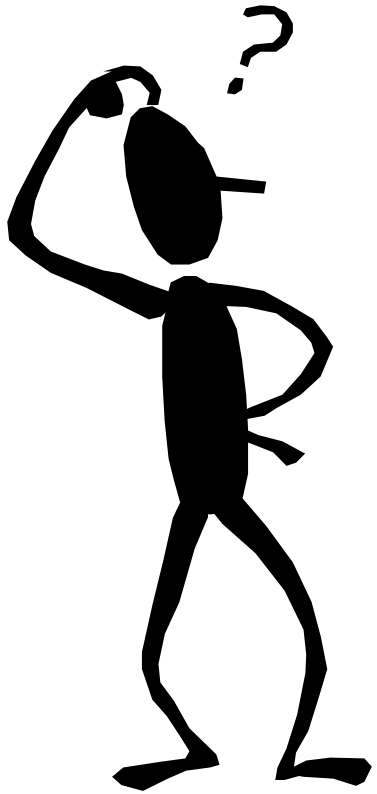

```
public abstract class PaymentFactoryMethod {
  public abstract Payment FactoryMethod();
}
```

```
public class PagoJuegos extends
PaymentFactoryMethod {
@Override
  public Payment FactoryMethod() {
          return new PaypalPayment();
  }
}
```

```
public class MainPaymentFactory {
    public static void main(String []args) {
          PaymentFactoryMethod p= new PagoJuegos();
          Payment payment=p.FactoryMethod();
    }
}
```

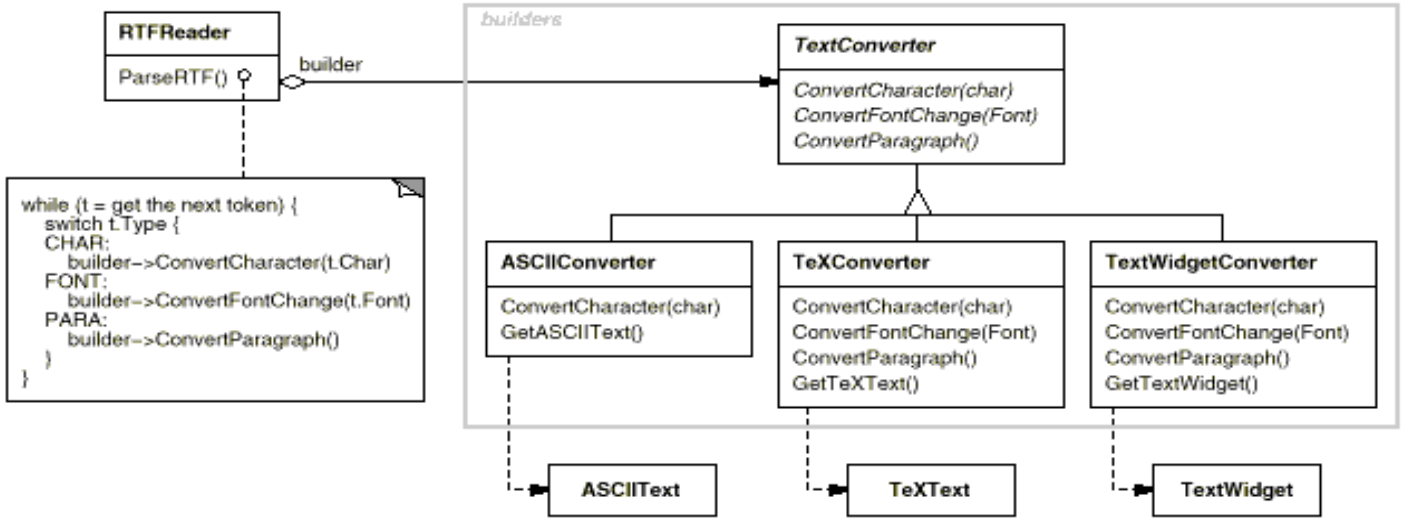
```
public class PagoLibros extends
PaymentFactoryMethod {
@Override
  public Payment FactoryMethod() {
          return new CardPayment();
  }
}
```

- Ejemplo de Desguace:
  - ¿Se podría utilizar SimpleFactory? ¿En dónde?
  - ¿Se podría usar abstract factory? ¿En dónde?



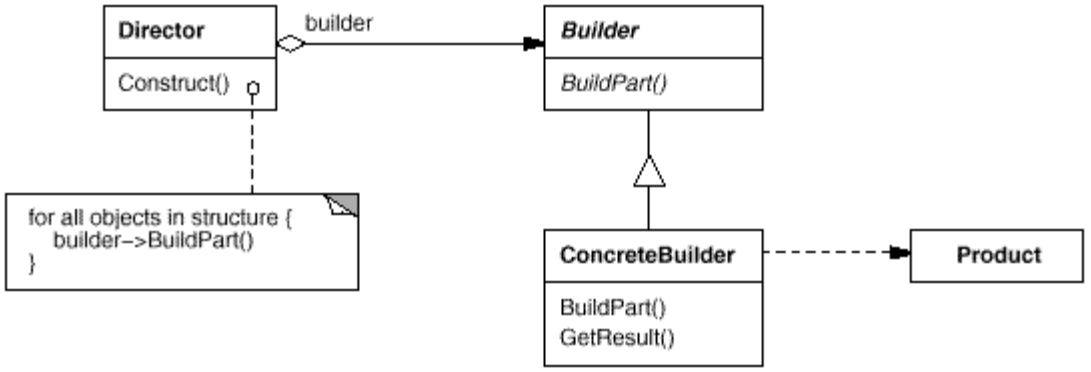
Builder – Constructor (1/4)

- Objetivo: Abstrae el proceso de creación de un objeto complejo, **centralizando dicho proceso en un único punto**.
- Motivación (Libro Gang of Four):
  - ✓ Un traductor de documentos RTF a otros formatos. ¿Es posible añadir una nueva conversión sin modificar el traductor?




```
TeXConverter ctex = new TeXConverter()
RTFReader trad = new RTFReader  (ctex, doc) //En la creación se indica el objeto TextConverter
trad.parseRTF(doc);                  //Se convierte RTF a TeX
TextTex texto =TeXConverter.GetTeXText();, //Se obtiene el texto traducido (TextTex)
```

- Builder – Constructor (2/4)
  - Estructura (Libro Gang of Four):
    - ✓ **Builder especifica la interfaz** para la creación de las partes de Product
    - ✓ **ConcreteBuilder construye y ensambla las partes del producto**, mantiene el estado intermedio del proceso de construcción y ofrece una operación para obtener el resultado
    - ✓ **Director construye un objeto mediante la interfaz de Builder**
    - ✓ **Product representa al objeto complejo que se está construyendo,** incluyendo las clases que definen las partes y las interfaces necesarias para ensamblarlas
  - Ejemplo: Constructor de Mensaje que recibe un objeto Cifrador ya creado (luego se usará los métodos Cifrar o Descifrar)



Builder – Constructor (3/4)

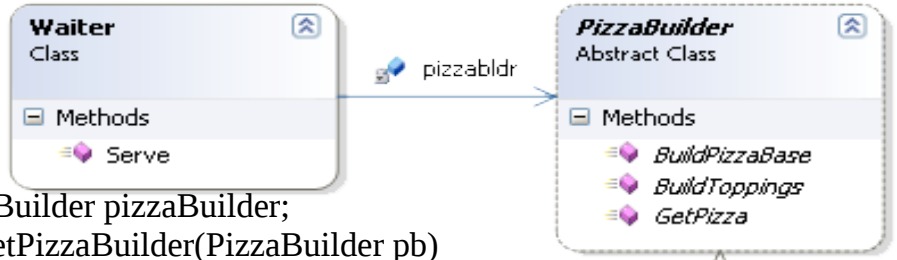
- Aplicabilidad:
  - Cuando el algoritmo para crear un objeto complejo debe ser independiente de las piezas que conforman el objeto y de cómo se ensamblan.
  - El proceso de construcción debe permitir diferentes representaciones para el objeto que se construye.
- Solución.
  - La interfaz de *Builder* debe ser lo suficientemente general para permitir la construcción de productos para cualquier *Builder* concreto.
  - La construcción puede ser más complicada de añadir el nuevo *token* al producto en construcción.
  - Los métodos de *Builder* no son abstractos sino vacíos.
  - Las clases de los productos no siempre tienen una clase abstracta común.

Builder – Constructor (4/4)

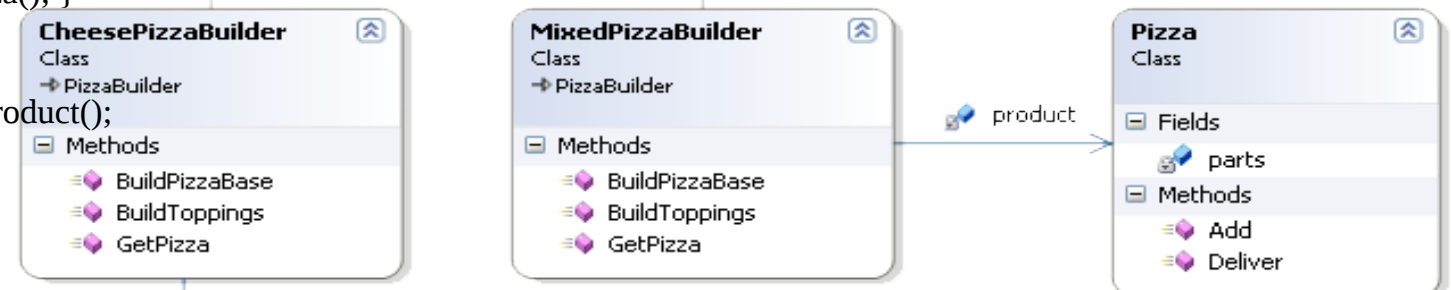
- Consecuencias
  - ✓ Permite cambiar la representación interna del producto.
  - ✓ Separa el código para la representación y para la construcción.
  - ✓ Los clientes no necesitan conocer nada sobre la estructura interna.
  - ✓ Diferentes “directores” pueden reutilizar un mismo “builder”
  - ✓ Proporciona un control fino del proceso de construcción.
- Usos en el API de Java
  - ✓
- Patrones relacionados
  - ✓

Ejemplo de Builder

```
abstract class PizzaBuilder {
  protected Pizza pizza;

  public Pizza getPizza() { return pizza; }
  public void createNewPizzaProduct() { pizza = new Pizza(); }

  public abstract void buildDough();
  public abstract void buildSauce();
  public abstract void buildTopping();
}
```




```
class Waiter {
  private PizzaBuilder pizzaBuilder;
  public void setPizzaBuilder(PizzaBuilder pb)
         { pizzaBuilder = pb; }
  public Pizza getPizza()
       { return pizzaBuilder.getPizza(); }

  public void constructPizza() {
    pizzaBuilder.createNewPizzaProduct();
    pizzaBuilder.buildDough();
    pizzaBuilder.buildSauce();
    pizzaBuilder.buildTopping();
  }
}
```



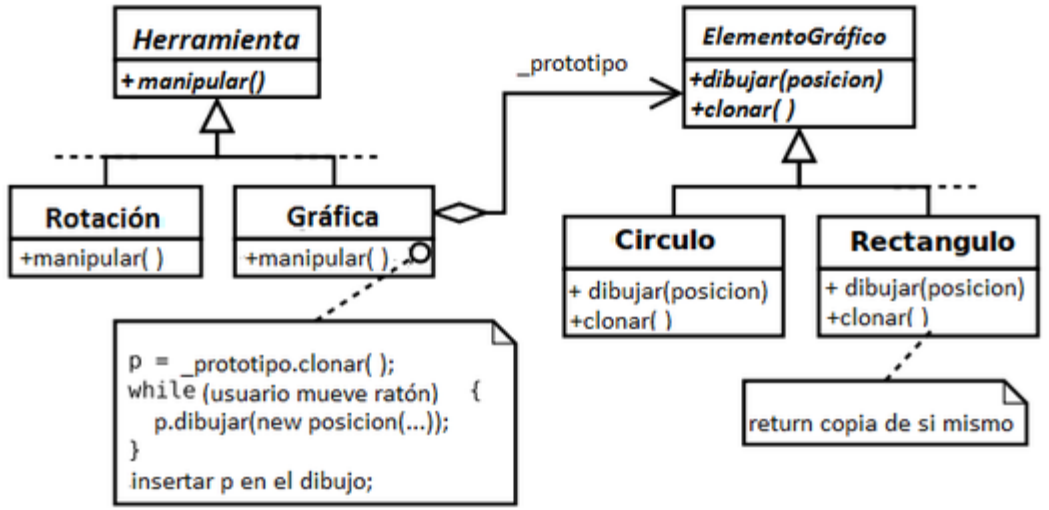

```
class CheesePizzaBuilder extends PizzaBuilder {
  public void buildDough()   { ….. }
  public void buildSauce()   { ….}
  public void buildTopping() { ….. }
}
```

```
class Pizza {
private String dough = "";
private String sauce = "";
private String topping = "";
```

```
class BuilderExample {
  public static void main(String[] args) {
    Waiter waiter = new Waiter();
    PizzaBuilder cheese_pizzabuilder = new CheesePizzaBuilder();
    PizzaBuilder mixed_pizzabuilder = new MixedBuilder();
    waiter.setPizzaBuilder( cheese_pizzabuilder );
    waiter.constructPizza();
    Pizza pizza = waiter.getPizza();
  }
```

Prototype -Prototipo (1/4)

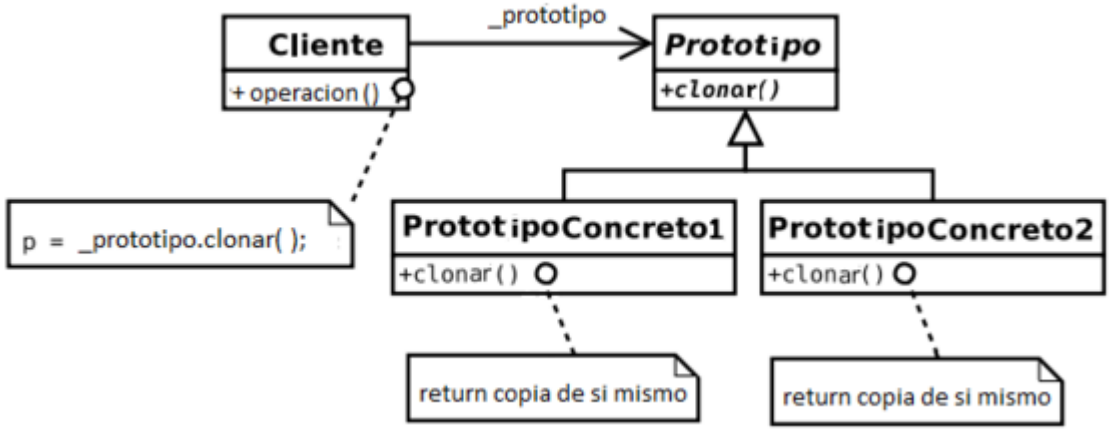
- Objetivo:
  - ✓ Crea **nuevos objetos clonándolos de una instancia ya existente**. De este modo los nuevos objetos que se crearán de los prototipos, en realidad, son clonados.
- Motivación (Libro Gang of Four):
  - ✓ Un editor gráfico, podemos crear rectángulos, círculos, etc... como copias de prototipos. Estos objetos gráficos pertenecerán a una jerarquía cuyas clases derivadas implementarán el mecanismo de clonación.



Prototype -Prototipo (2/4)

Estructura (Libro Gang of Four):

- ✓ Cliente: Es el encargado de solicitar la creación de los nuevos objetos a partir de los prototipos.
- ✓ Prototipo Concreto: Posee una características concretas que serán reproducidas para nuevos objetos e implementa una operación para clonarse.
- ✓ Prototipo: Declara una interfaz para clonarse, a la que accede el cliente.



- Prototype -Prototipo (3/4)
  - Aplicabilidad:
    - ✓ Cuando un sistema deba ser independiente de cómo sus productos son creados, compuestos y representados y:
      - las clases a instanciar son especificadas en tiempo de ejecución, **o**
      - para evitar crear una jerarquía de factorías paralela a la de productos, **o**
      - cuando las instancias de una clase sólo pueden encontrarse en uno de un pequeño grupo de estados, **o**
      - inicializar un objeto es costoso
  - Solución (Ver estructura).

Prototype -Prototipo (4/4)

Consecuencias

- Aplicar el patrón prototipo permite ocultar las clases producto (prototipos concretos) del cliente y permite que el cliente trabaje con estas clases dependientes de la aplicación sin cambios.
- Permite añadir y eliminar productos en tiempo de ejecución al invocar a la operación clonar, lo que supone un método que proporciona una configuración dinámica de la aplicación.
- **Desventaja**: La jerarquía de prototipos debe ofrecer la posibilidad de clonar un elemento y esta operación puede no ser sencilla de implementar. Por otro lado, si la clonación se produce frecuentemente, el coste puede ser importante.

Usos en el API de Java

- En Java se dispone de la interfaz cloneable y del Object Clone() throws CloneNotSupportedException para llevar a cabo la implementación nuestro prototipo de manera compatible con los prototipos ya existentes en las librerías Java.

- Prototype -Prototipo. Ejemplo
  - En el desguace se crean 4 tipos de productos/piezas:
    - ✓ frenos, neumáticos, correas y limpiaparabrisas.
  - Las nuevas piezas se crearán como copias de estos 4 prototipos.

```
public abstract class Pieza implements Cloneable{
  private String id;
  private String nombre;
  private int stock;

  public Pieza(String id, String nombre, int contador) {
            this.id = id;
            this.nombre = nombre;
            this.stock = contador;
  }
  public Object clone() throws CloneNotSupportedException{
            return super.clone();
  }
  ….
}
```

```
public class Freno extends Pieza{
  private String id;
  private String nombre;
  private int stock;
  private String tipo;
  public Frenos(String id, String nombre, int contador, String t) {
      super(id,nombre, contador);
      setTipo(t);
  }
  public Object clone() throws CloneNotSupportedException{
     return new Freno (this.id, this.nombre, this.stock,this.tipo);
  }
….
}
```

Prototype -Prototipo. Ejemplo

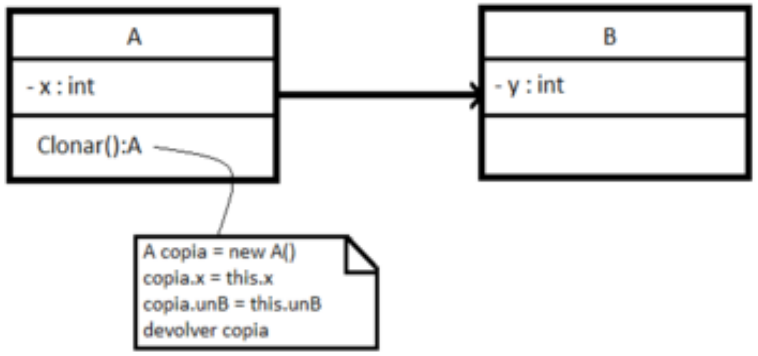
```
public class PiezaPrototype {
  private HashMap<String,Pieza> prototipos= new HashMap<String,Pieza>();

  public PiezaPrototype(){
          Freno f= new Freno("0","Freno",1,"");
          Neumatico n= new Neumatico("0","Neumatico",1,"");
          Correa c= new Correa("0","Correa",1,"");
          //Se añaden los distintos tipos que pueden existir
          prototipos.put("frenos", f);
          prototipos.put("neumaticos", n);
          prototipos.put("correas", c);
  }
  public Object prototipo(String tipo) throws CloneNotSupportedException{
          return prototipos.get(tipo).clone();
  }
}
```

```
public class MainPrototype {
  public static void main(String[]args) throws CloneNotSupportedException{
    //Instancia la fábrica de crear objetos prototipados
    PiezaPrototype pt= new PiezaPrototype();
    Freno f= (Freno) pt.prototipo("frenos");
    f.setId("5");    //Se modificas los atributos diferentes
    Neumatico n= (Neumatico)pt.prototipo("neumaticos");
    n.setId("10");
    System.out.println(f.toString());
  }
}
```

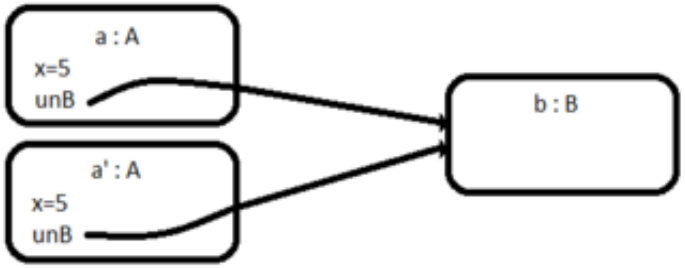
Prototype -Prototipo (4/4)

Clonacion profunda vs Clonacion superficial

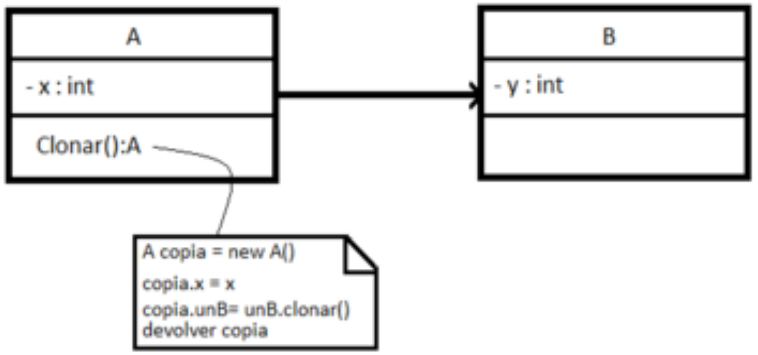
Superficial:



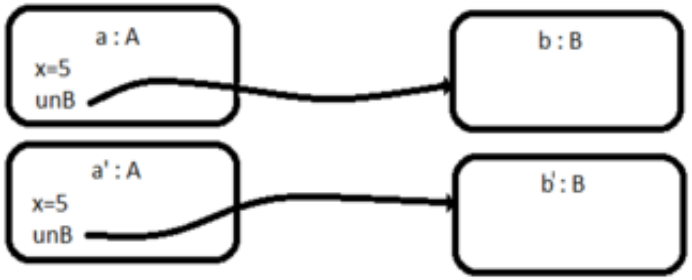
Con esto obtendriamos:



Profunda:



Con esto obtendriamos:



Resumiendo

- Patrones de creación
  - Singleton. Garantiza que una clase sólo tenga una instancia, y proporciona un punto de acceso global a ella.
  - Factory Method. Define una interfaz para crear un objeto, pero deja que sean las subclases quienes decidan qué clase instanciar. Permite que una clase delegue en sus subclases la creación de objetos.
  - Builder. Separa la construcción de un objeto complejo de su representación, de forma que el mismo proceso de construcción pueda crear diferentes representaciones.
  - Prototype. Especifica los tipos de objetos a crear por medio de una instancia prototípica, y crear nuevos objetos copiando este prototipo.

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