

*Patrones de Diseño:*  
*Patrones de Creación.*

*Tema 3-1:*  
*Introducción*



# Patrones de Creación

- 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 se delegarán responsabilidades.
- La valía de los patrones de creación nos dice como estructurar y encapsular estas decisiones. A menudo hay varios patrones de creación que se pueden aplicar en una situación o incluso combinarlos. En otros casos se debe elegir tan solo uno de ellos.



# Patrones de Creación

- Tienen las siguientes capacidades:
  - **Instanciación genérica:** Permite crear objetos en un sistema sin tener que identificar una clase específica en el código.
  - **Simplicidad:** Algunos de los patrones facilitan la creación de objetos, por lo que no se tendrá que escribir gran cantidad de código para crear un objeto.
  - **Restricciones de creación:** Algunos patrones fuerzan restricciones en el tipo o el número de objetos que pueden ser creados en el sistema.



# Clasificación Patrones de Creación

- **Abstract Factory:** Proporciona un contrato para la creación de familias de objetos relacionados o dependientes sin tener que especificar su clase concreta.
- **Builder:** Simplifica la creación de objetos complejos definiendo una clase cuyo propósito es construir instancias de otra clase.
- **Factory Method:** Permite definir un método estándar para crear un objeto, además del constructor propio de la clase, si bien la decisión del objeto a crear se delega en las subclases.



# Clasificación Patrones de Creación

- **Prototype:** Facilita la creación dinámica al definir clases cuyos objetos pueden crear copias de sí mismos.
- **Singleton:** Permite tener una única instancia de una clase en el sistema, además de permitir que todas las clases tengan acceso a esa instancia.



# Resumen Patrones de Creación

- Hay dos formas de parametrizar un sistema: uno es con la herencia y otro es con la composición. Ejemplo del primer método es el **Factory Method** y ejemplo del segundo son los patrones **Abstract Factory**, **Builder** y **Prototype**.
- **Abstract Factory** produce objetos de varias clases.
- **Builder** construye un producto complejo incrementalmente.
- **Prototype** hace que su objeto fábrica construya un producto copiando un objeto prototípico.
- Muchas veces los diseños comienzan usando el patrón **Factory Method** y luego evolucionan hacia los otros patrones de creación.



*Patrones de Diseño:*  
*Patrones de Creación.*

*Tema 3-2:*  
*Abstract Factory*



# Descripción del patrón

- **Nombre:**
  - Fábrica Abstracta
  - Conocido como: Kit, Toolkit
- **Propiedades:**
  - Tipo: creación
  - Nivel: objeto, componente
- **Objetivo o Propósito:**
  - Proporciona una interfaz para crear familias de objetos relacionados o que dependen entre sí, sin especificar sus clases concretas.

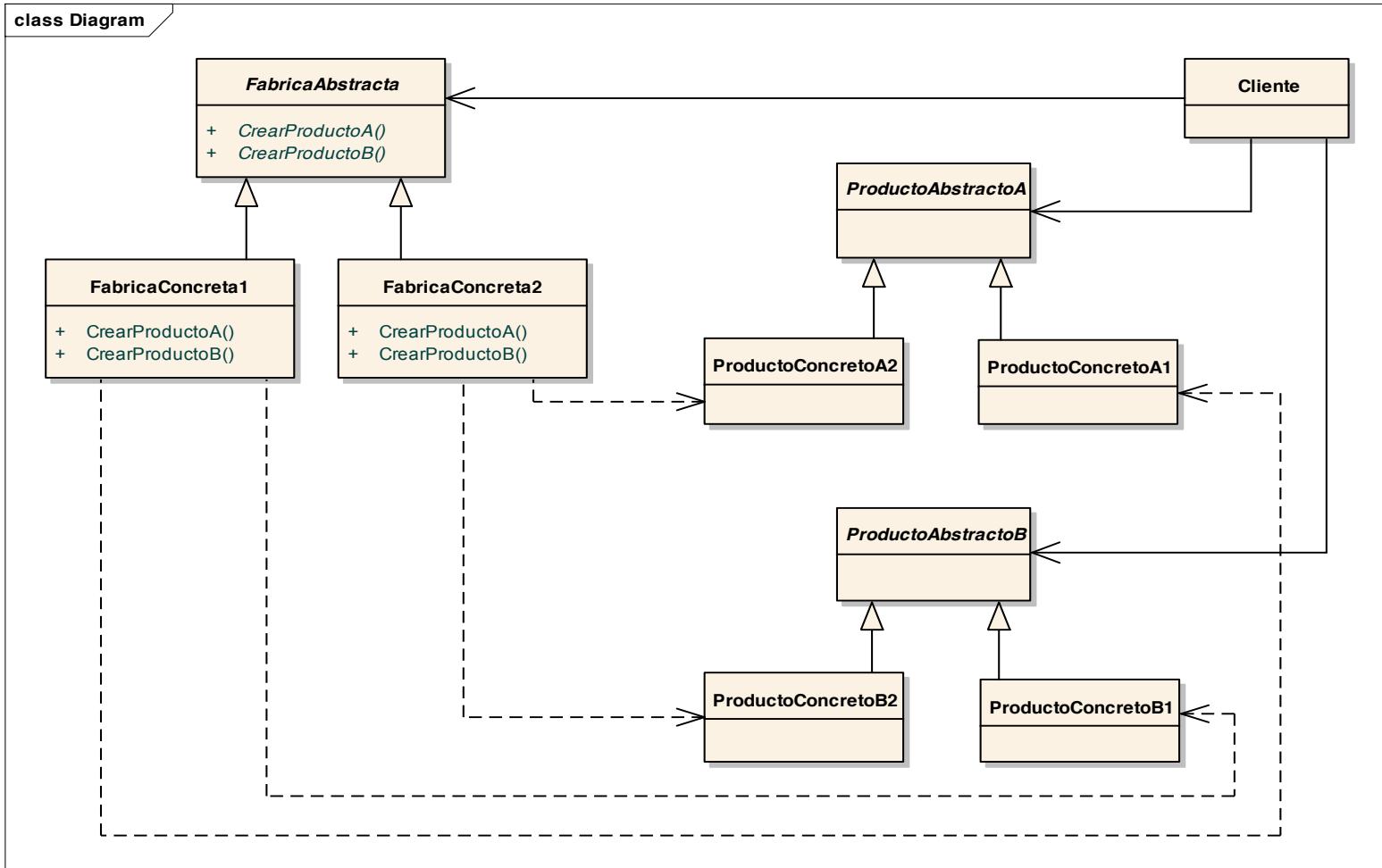


# Aplicabilidad

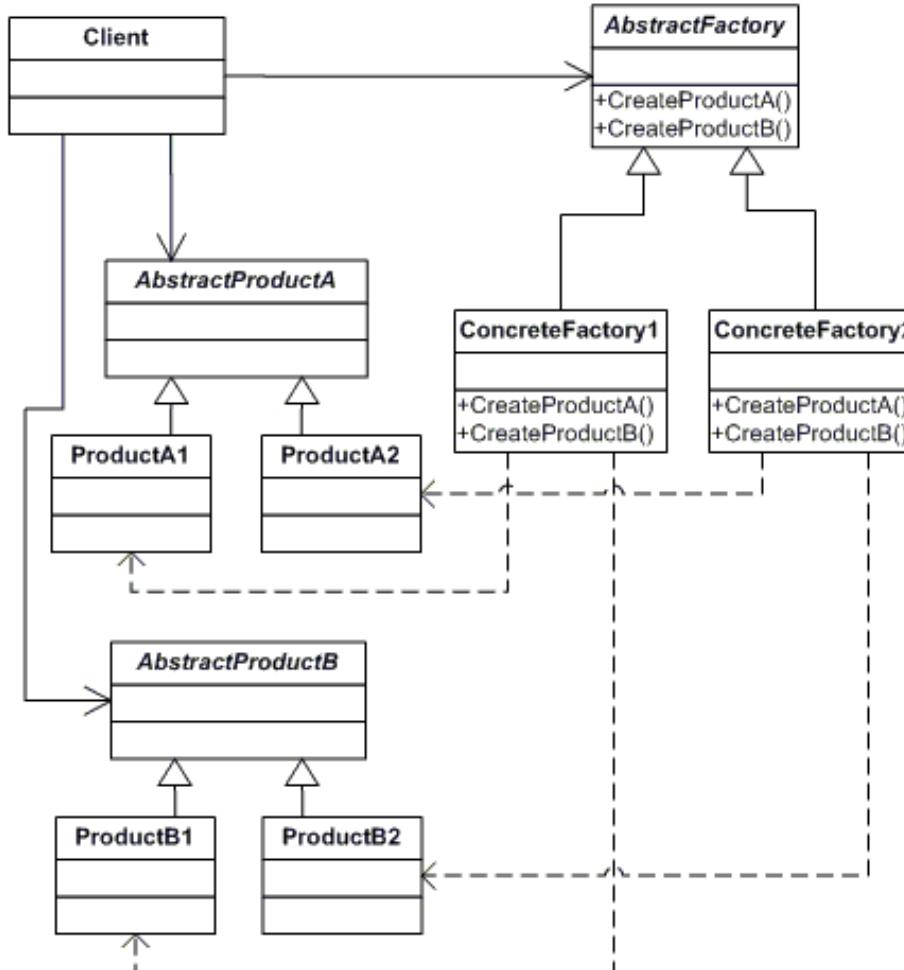
- Use el patrón Abstract Factory cuando:
  - El cliente debe ser independiente del proceso de creación de los productos.
  - La aplicación debe configurarse con una o más familias de productos.
  - Es necesario crear los objetos como un conjunto, de forma que sean compatibles.
  - Se desea proporcionar una colección de clases y únicamente quiere revelar sus interfaces y sus relaciones, no sus implementaciones.
- Ejemplo: Sistema de gestión de ventanas o sistema de ficheros para diversos sistemas operativos.



# Estructura



# Estructura

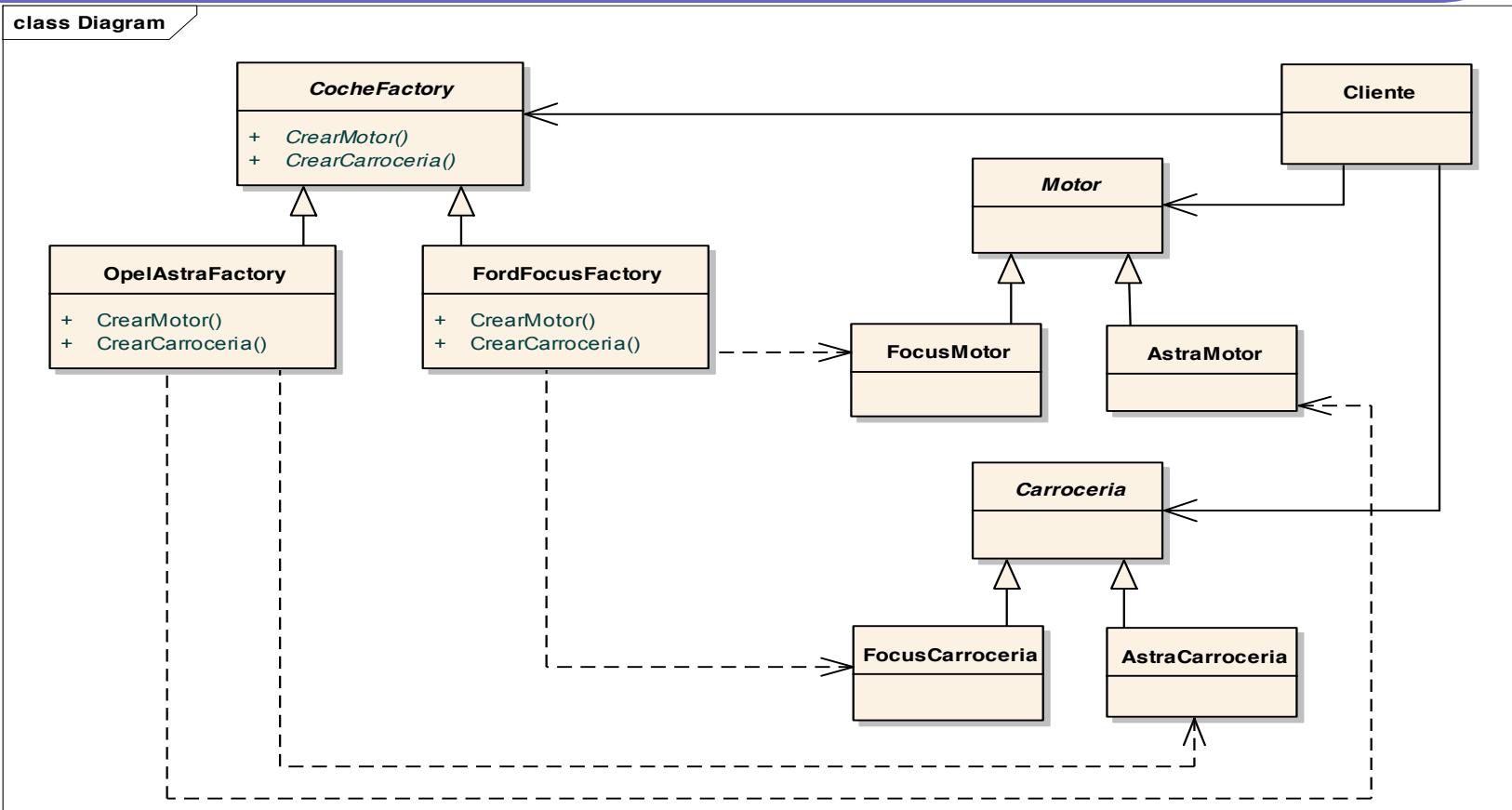


# Estructura. Participantes

- **FabricaAbstracta:** Declara una interfaz para operaciones que crean objetos de tipo producto abstracto.
- **FabricaConcreta:** Implementa las operaciones para crear objetos de tipo producto concreto.
- **ProductoAbstracto:** Declara una interfaz para un tipo de objeto producto.
- **ProductoConcreto:** Define un objeto producto para que sea creado por la fábrica correspondiente. Implementa la interfaz ProductoAbstracto.
- **Cliente:** Solo usa interfaces declaradas por las clases FabricaAbstracta y ProductoAbstracto.



# Estructura Ejemplo



# Estructura Ejemplo

- Identificamos a continuación los elementos del patrón Abstract Factory:
  - FabricaAbstracta: CocheFactory.
  - FabricaConcreta: OpelAstraFactory, FordFocusFactory.
  - ProductoAbstracto: Motor, Carrocería.
  - ProductoConcreto: FocusMotor, AstraMotor, FocusCarroceria, AstraCarroceria.
  - Cliente: Cliente.



# Consecuencias

1. **Aísla las clases concretas.** Ayuda a controlar las clases de objetos que crea una aplicación. Aísla a los clientes de las clases de implementación. Los clientes manipulan las instancias a través de sus interfaces abstractas.
2. **Facilita el intercambio de familias de productos.** Como se crea una familia completa de productos, toda la familia de productos cambia de una vez.
3. **Promueve la consistencia entre productos.** Facilita que se trabaje con una sola familia de productos.
4. **Dificulta la incorporación de nuevos tipos de productos.** Debido a que la interfaz FabricaAbstracta fija el conjunto de productos que se pueden crear.



# Patrones relacionados

- **Factory Method y Prototype:** Las clases FabricaAbstracta suelen implementarse con métodos de fabricación pero también se pueden implementar usando prototipos.
- **Singleton:** Una fábrica concreta suele ser un Singleton.
- **Builder.** El patrón Abstract Factory se preocupa de lo que se va a hacer, y el Builder, de cómo se va a hacer. Abstract Factory es similar al patrón Builder en que ambos pueden construir objetos complejos. La principal diferencia es que el patrón Builder se centra en construir un objeto complejo paso a paso. Abstract Factory hace énfasis en familias de objetos Producto, tanto simples como complejas.



# Conclusiones

- **Ocultación de las clases de implementación.** Esto también permite ofrecer clases sin mostrar su implementación, pudiéndonos reservar la posibilidad de cambiar la implementación en el futuro.
- Facilidad de **sustitución** de conjuntos de clases por otras equivalentes.
- **Consistencia** entre productos, al desarrollarse las aplicaciones con un mismo conjunto homogéneo de clases, dichas aplicaciones son más consistentes entre ellas.
- **Soportar nuevos tipos de productos es difícil.** Extender los Abstract Factory para producir nuevos tipos de Productos no es fácil. Soportar nuevos tipos de productos requiere extender la interface factory, que implica cambiar la clase Abstract Factory y todas sus subclases.

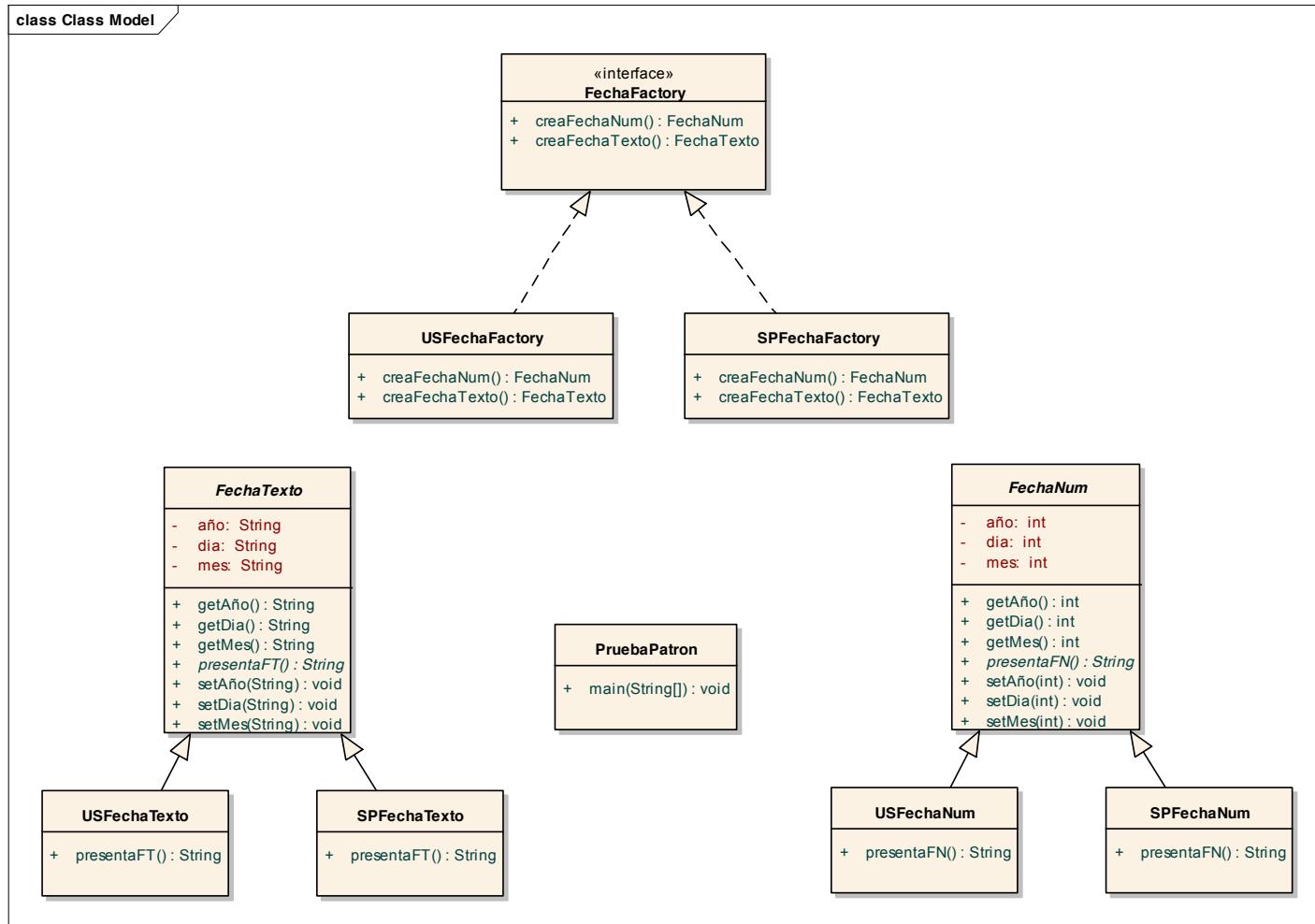


# Código de ejemplo

## Factoría de fechas



# Código de ejemplo



# Código de ejemplo

- Identificamos a continuación los elementos del patrón Abstract Factory:
  - FabricaAbstracta: FechaFactory.
  - FabricaConcreta: USFechaFactory, SPFechaFactory.
  - ProductoAbstracto: FechaTexto, FechaNum.
  - ProductoConcreto: USFechaTexto , SPFechaTexto, USFechaNum, SPFechaNum.
  - Cliente: PruebaPatron.



*Patrones de Diseño:*  
*Patrones de Creación.*

*Tema 3-3:*  
*Builder*



# Descripción del patrón

- **Nombre:**
  - Constructor
- **Propiedades:**
  - Tipo: creación
  - Nivel: objeto, componente
- **Objetivo o Propósito:**
  - Separar la construcción de un objeto complejo de su representación, de forma que el mismo proceso de construcción pueda crear diferentes representaciones.

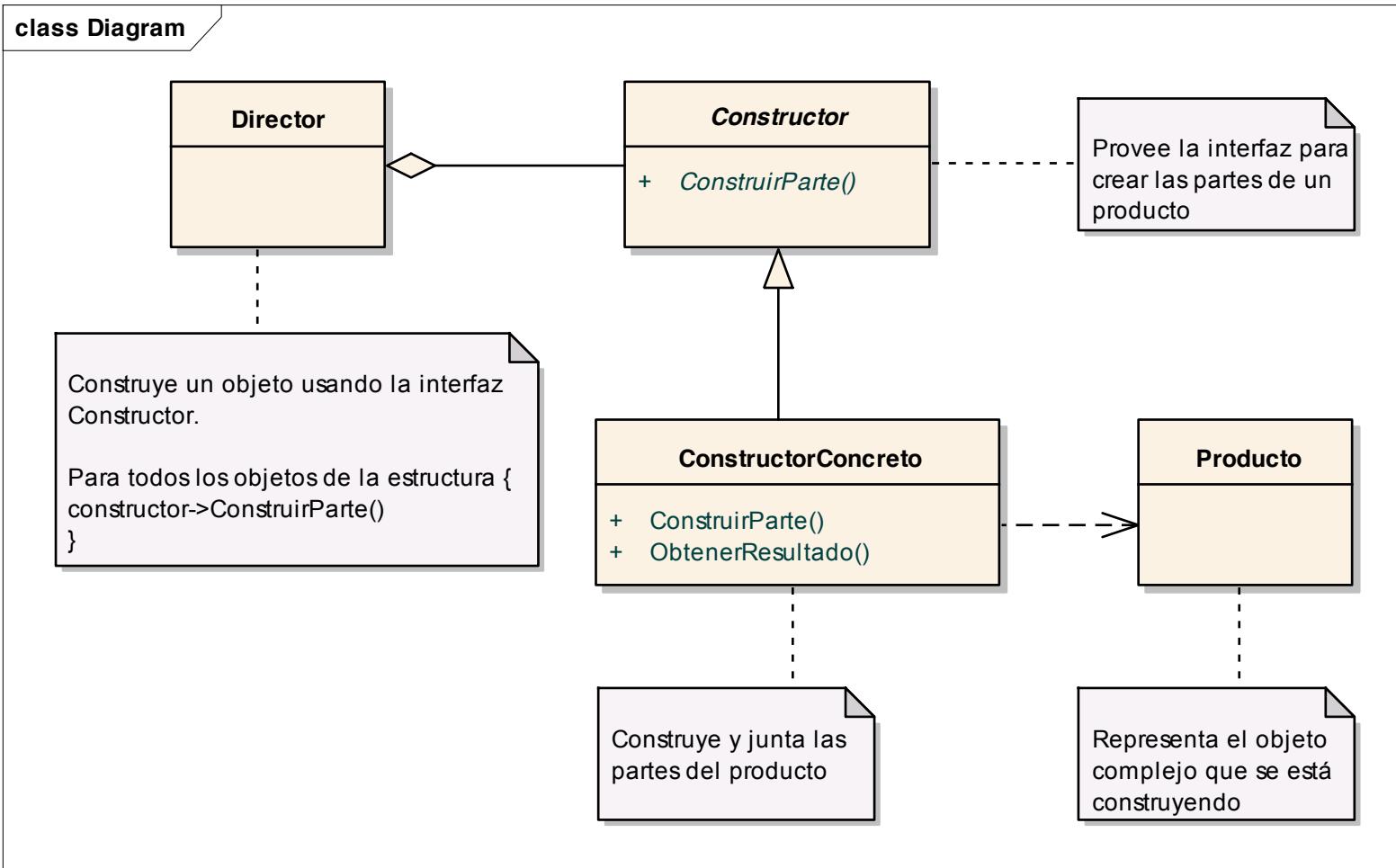


# Aplicabilidad

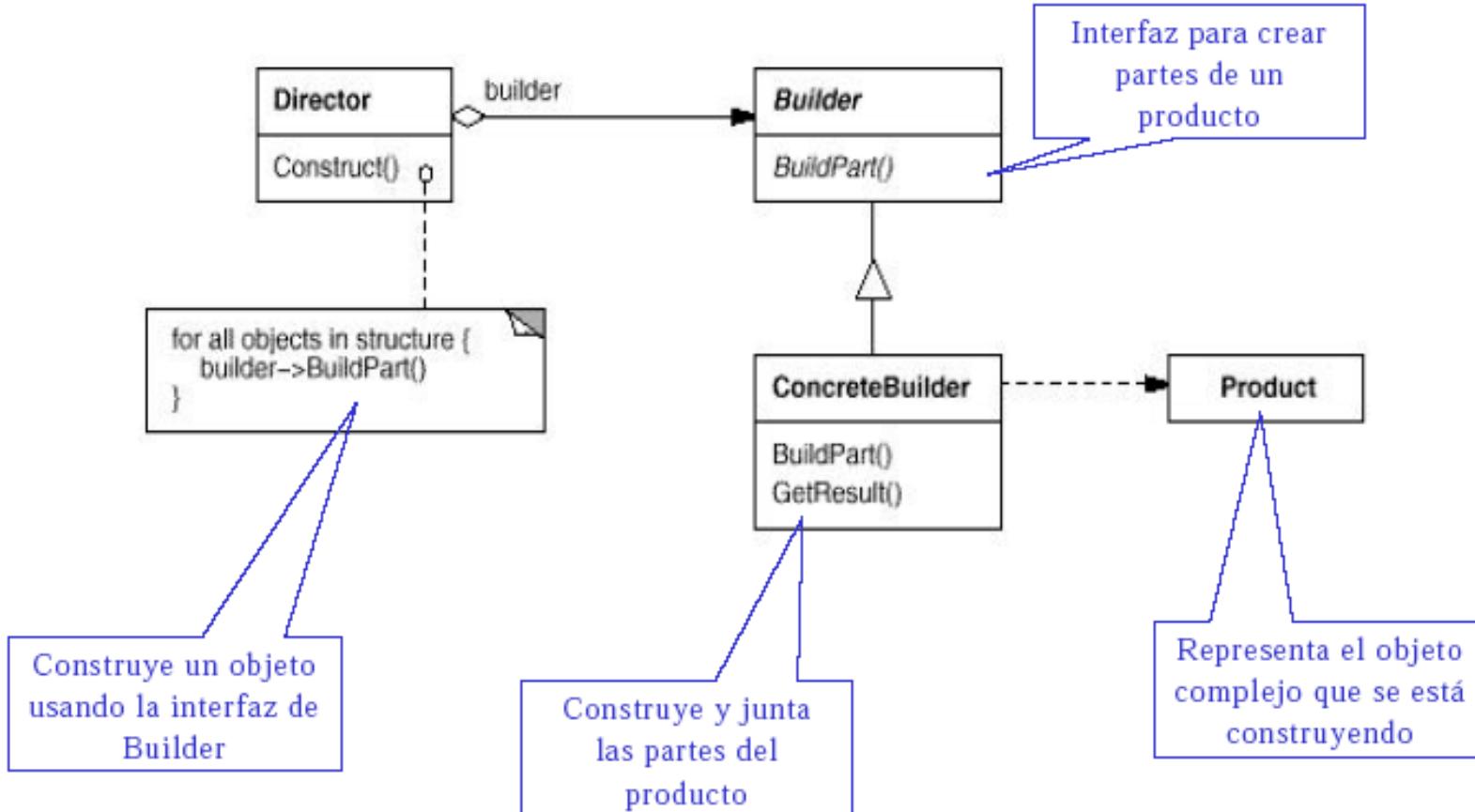
- Use el patrón Builder cuando:
  - Una clase tiene una estructura compleja (si tiene un conjunto variable de objetos relacionados).
  - La clase tiene atributos dependientes entre si. Construcción por etapas.
  - El algoritmo para crear un objeto complejo debiera ser independiente de las partes de que se compone dicho objeto y de cómo se ensamblan.
  - El proceso de construcción debe permitir diferentes representaciones del objeto que está siendo construido.



# Estructura



# Estructura

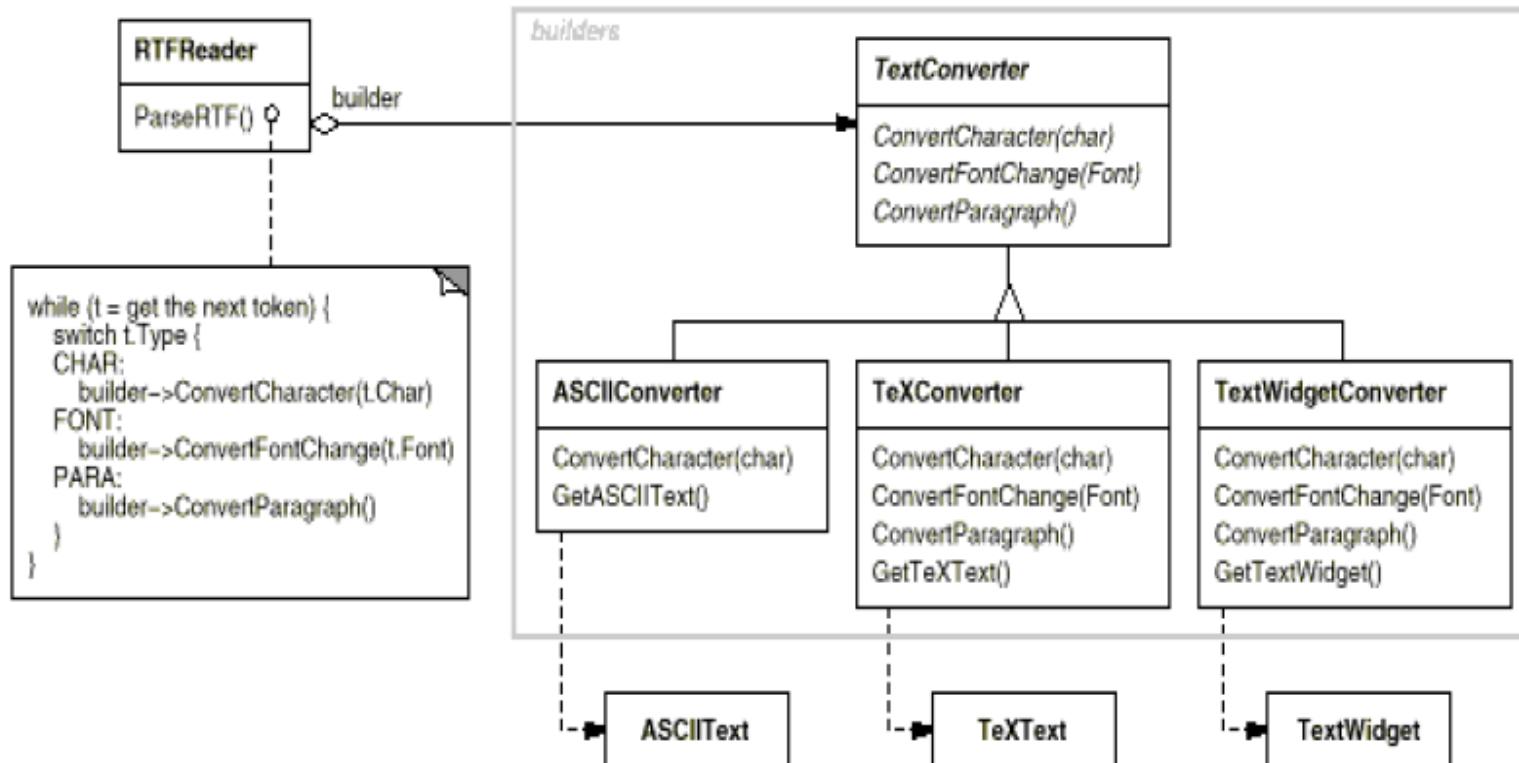


# Estructura. Participantes

- **Constructor:** Especifica una interfaz para crear las partes de un objeto.
- **ConstructorConcreto:** Implementa la interfaz Constructor para construir y ensamblar las partes del producto. Define la representación a crear. Proporciona un método que devuelve una instancia del producto.
- **Producto:** Representa el objeto complejo en construcción.
- **Director:** Construye un objeto usando la interfaz Constructor.
- **Cliente:** Crea el objeto Director y lo configura con el objeto Constructor deseado.



# Estructura. Ejemplo



# Estructura. Ejemplo

- Supongamos un editor que quiere convertir un tipo de texto (p.ej. RTF) a varios formatos de representación diferentes, y que puede ser necesario en el futuro definir nuevos tipos de representación.
- El lector de RTF (RTFReader) puede configurarse con una clase de Conversor de texto (TextConverter) que convierta de RTF a otra representación.
- A medida que el RTFReader lee y analiza el documento, usa el conversor de texto para realizar la conversión: cada vez que reconoce un token RTF llama al conversor de texto para convertirlo.
- Hay subclases de TextConverter para cada tipo de representación.
- El conversor (*builder*) está separado del lector (*director*): se separa el algoritmo para interpretar un formato textual de cómo se convierte y se representa.



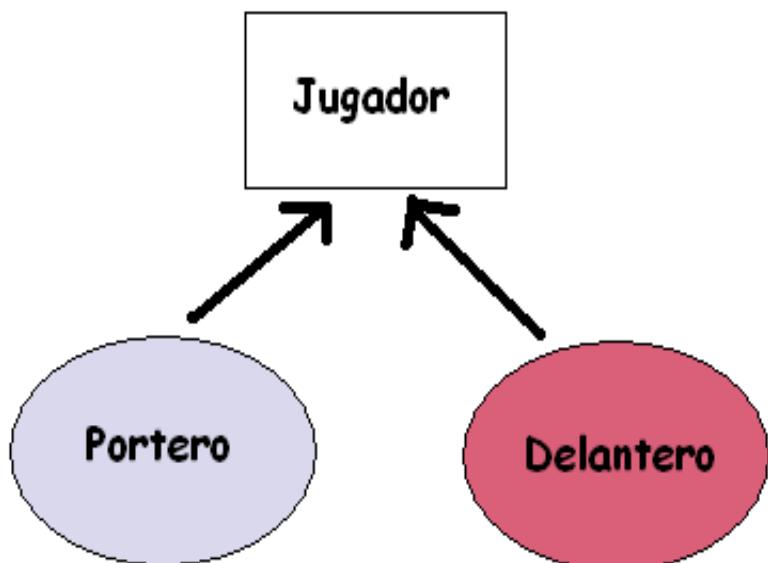
# Estructura. Ejemplo

- Identificamos a continuación los elementos del patrón Builder:
  - Constructor: TextConverter.
  - Constructor concreto: ASCIIConverter, TeXConverter, TextWidgetConverter.
  - Producto: ASCIIText, TeXText, TextWidget.
  - Director: RTFReader.

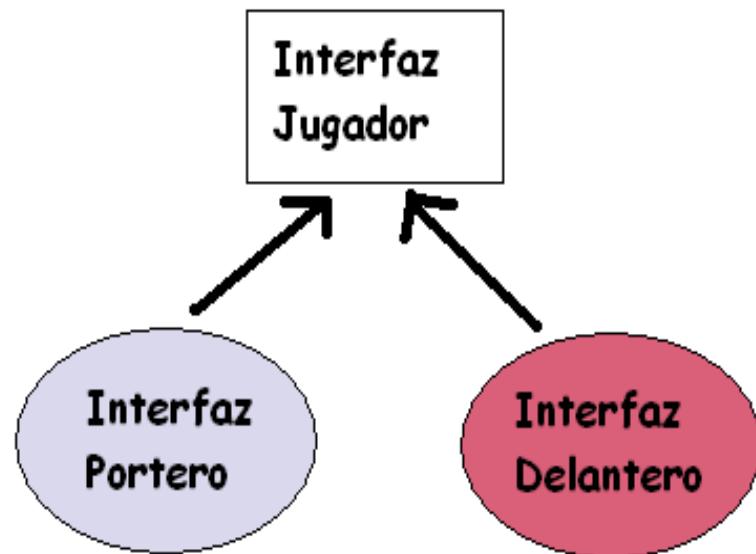


# Ejemplo

- **Datos**



- **Representación**



# Consecuencias

1. **Permite variar la representación interna de un producto.** El Constructor proporciona al director una interfaz para la construcción del producto, permitiendo la ocultación de su representación, su estructura interna y su ensamblaje. Para cambiar la representación interna del producto se define un nuevo constructor.
2. **Aísla el código de construcción y representación.** Se aumenta la modularidad al encapsular cómo se construyen y se representan los objetos complejos. Los clientes no saben nada de las clases que definen la estructura interna del producto.
3. **Proporciona un control más fino sobre el proceso de construcción.** Se construye el producto paso a paso (los demás patrones lo hacen de una vez) bajo el control del director.



# Consecuencias

- Reduce el acoplamiento.
- Permite variar la representación interna de estructuras complejas, respetando la interfaz común de la clase Builder.
- Se independiza el código de construcción de la representación. Las clases concretas que tratan las representaciones internas no forman parte de la interfaz del Builder.
- Cada ConcreteBuilder tiene el código específico para crear y modificar una estructura interna concreta.
- Distintos Directores con distintas utilidades pueden utilizar el mismo ConcreteBuilder.
- Permite un control mayor en el proceso de creación del objeto. El Director controla la creación paso a paso, solo cuando el Builder ha terminado de construir el objeto lo recupera el Director.



# Patrones relacionados

- **Factory Method.** El patrón Builder usa el patrón Factory Method para decidir qué clase concreta instanciar para construir el tipo de objeto deseado.
- **Abstract Factory.** El patrón Abstract Factory se preocupa de lo que se va a hacer, y el Builder, de cómo se va a hacer. Abstract Factory es similar al patrón Builder en que ambos pueden construir objetos complejos. La principal diferencia es que el patrón Builder se centra en construir un objeto complejo paso a paso. Abstract Factory hace énfasis en familias de objetos producto, tanto simples como complejas.
- **Prototype.** El builder se centra en crear paso a paso objetos complejos. Una clase builder es creada por cada clase producto que encapsula la complejidad de crear el objeto. La ventaja del Prototipo es, junto con la necesidad de solo una clase “factory”, el hecho de que la lógica de creación está unida directamente con cada clase producto.
- **Composite:** El patrón Builder a menudo se utiliza para producir objetos Composite, porque suelen tener una estructura compleja.

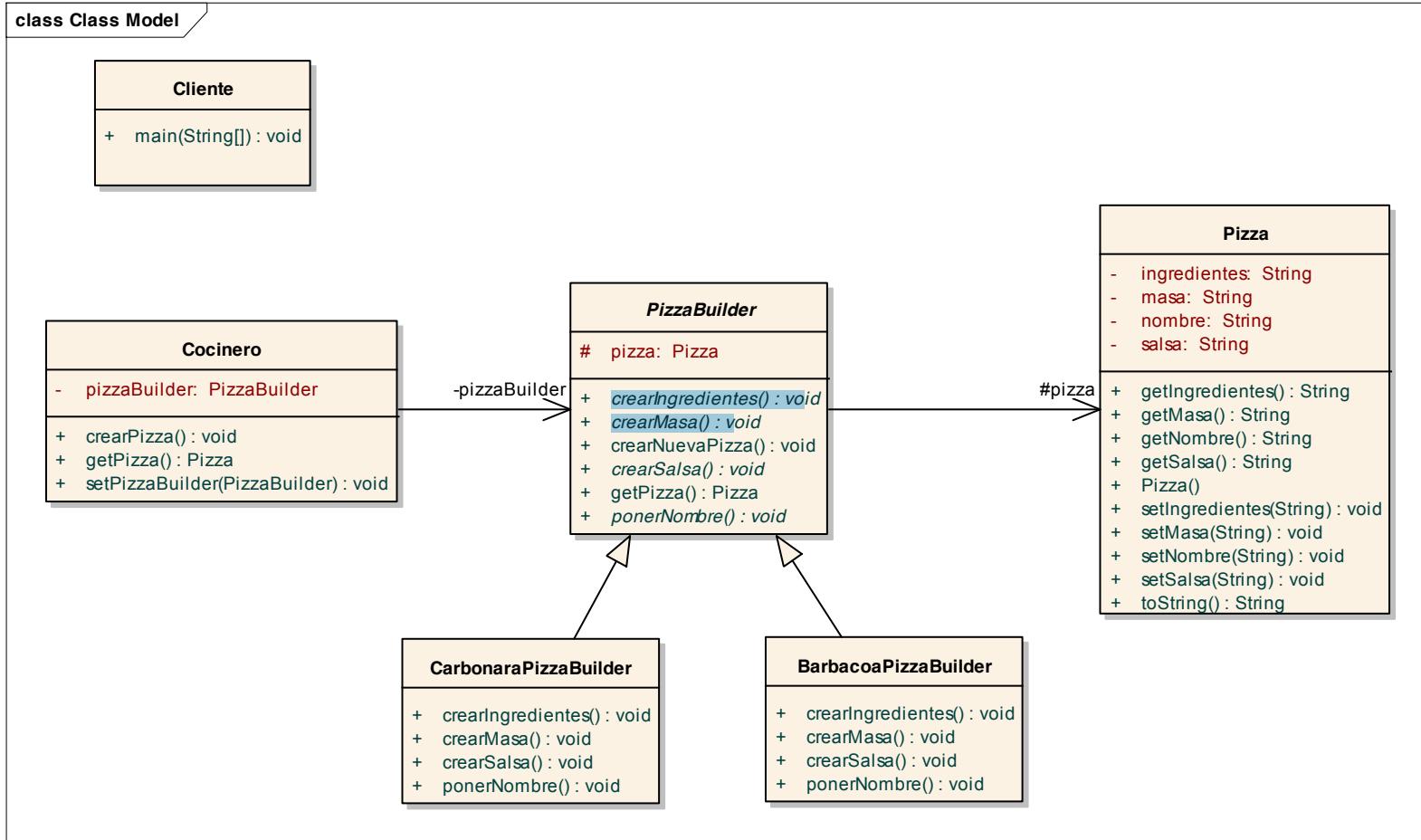


# Código de ejemplo

## Creación de Pizzas



# Código de ejemplo



# Estructura. Ejemplo

- Identificamos a continuación los elementos del patrón Builder:
  - Constructor: PizzaBuilder.
  - Constructor concreto: CarbonaraPizzaBuilder, BarbacoaPizzaBuilder.
  - Producto: Pizza.
  - Director: Cocinero.
  - Cliente: Cliente.



*Patrones de Diseño:*  
*Patrones de Creación.*

*Tema 3-4:*  
*Factory Method*



# Descripción del patrón

- **Nombre:**
  - Método de Fabricación
  - También conocido como Virtual Builder o Virtual Constructor (Constructor virtual)
- **Propiedades:**
  - Tipo: creación
  - Nivel: clase
- **Objetivo o Propósito:**
  - 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 del objeto.

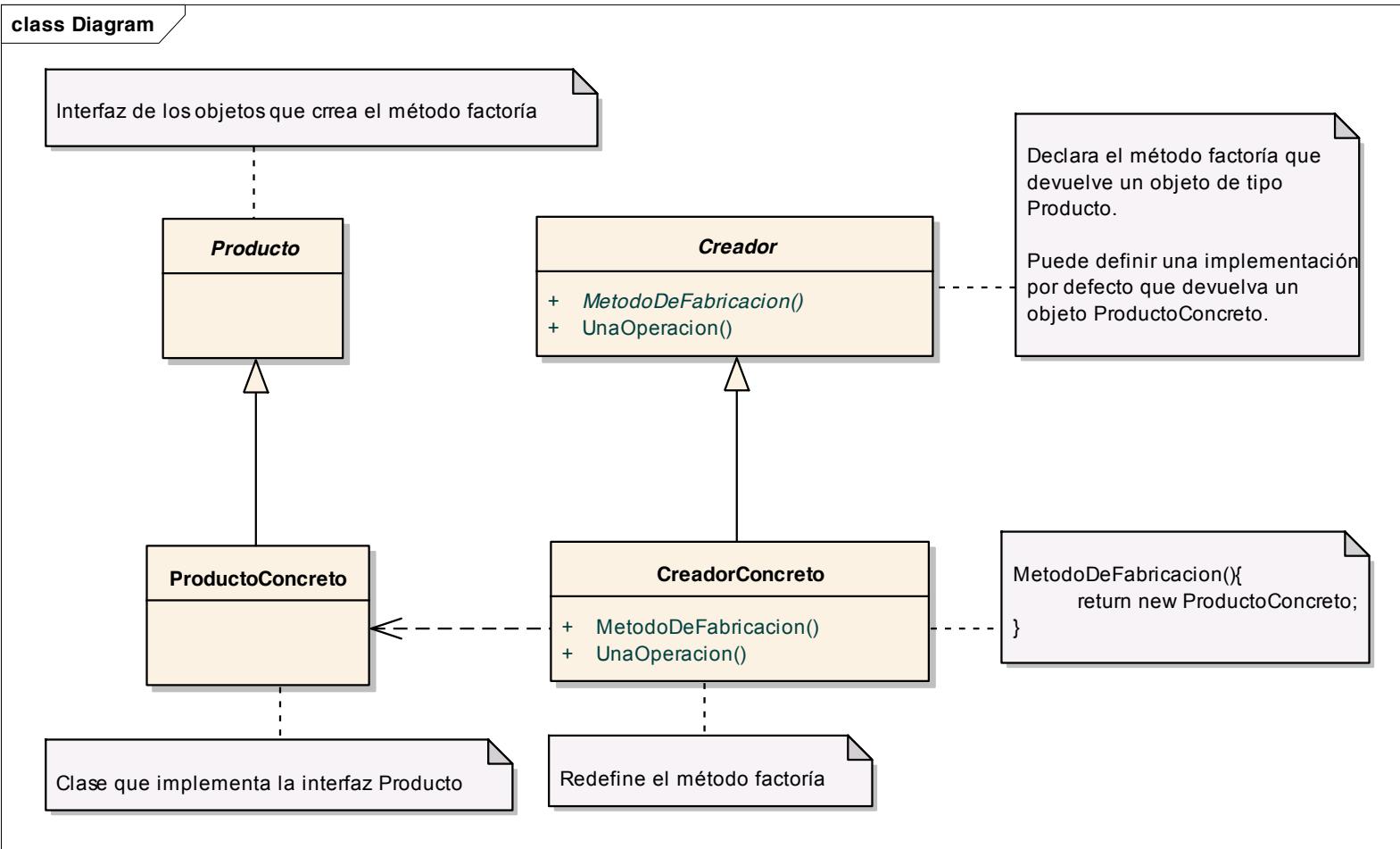


# Aplicabilidad

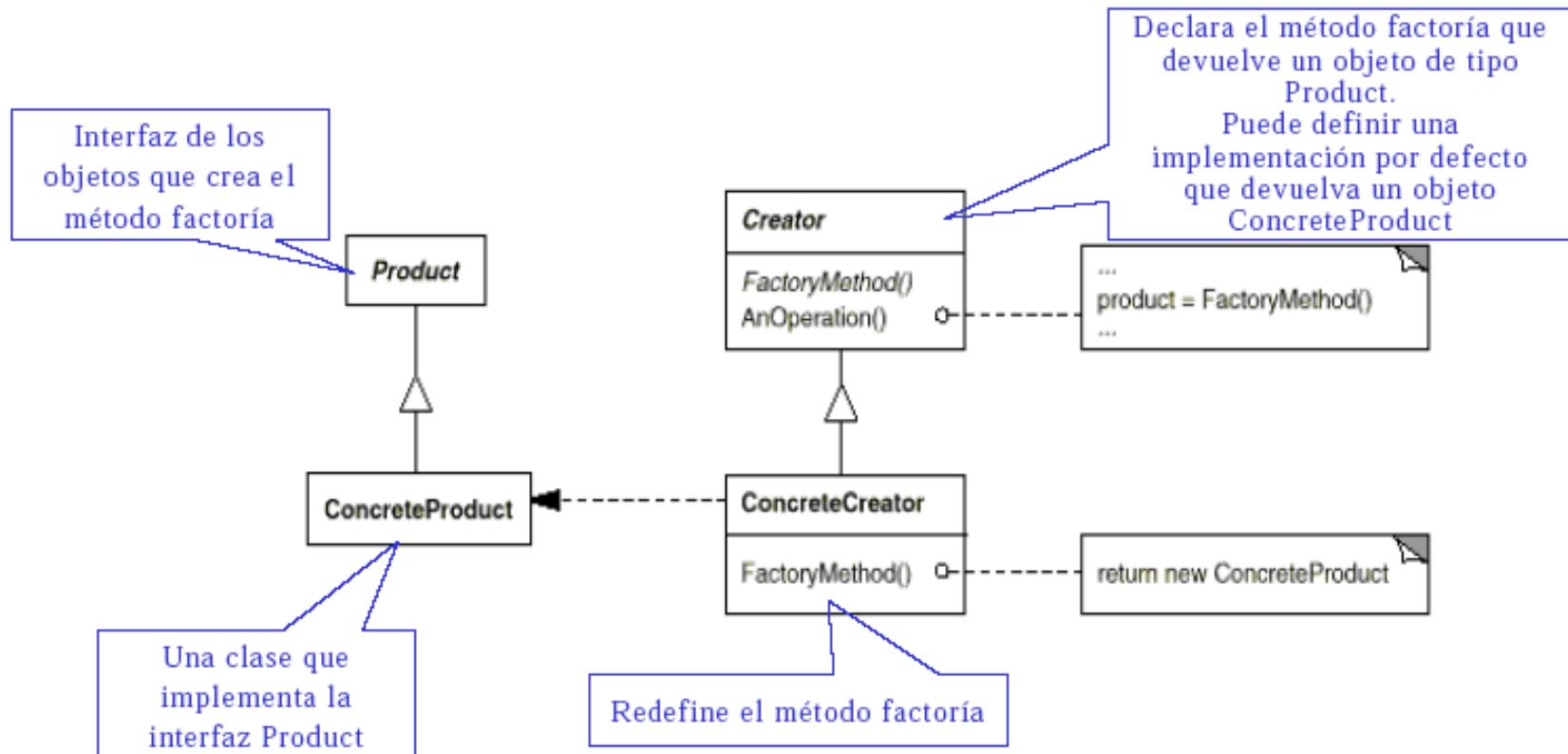
- Use el patrón Factory Method cuando:
  - Una clase no puede prever la clase de objetos que debe crear.
  - Una clase quiere que sean sus subclases quienes especifiquen los objetos que ésta crea.
  - Se sabe cuándo crear un objeto pero no conoce su tipo.
  - Se desea crear un framework extensible.
  - Necesita algunos constructores sobrecargados con la misma lista de parámetros (no permitido en Java). En vez de eso, utilizar varios métodos de fabricación con distinto nombre.



# Estructura



# Estructura



# Estructura. Participantes

- **Producto:** Define la interfaz de los objetos que crea el método de fabricación.
- **ProductoConcreto:** Implementa la interfaz Producto.
- **Creador:** Declara el método de fabricación, el cual devuelve un objeto de tipo Producto. También puede definir una implementación predeterminada del método de fabricación que devuelva un objeto ProductoConcreto.
- **CreadorConcreto:** Redefine el método de fabricación para devolver una instancia de un ProductoConcreto.



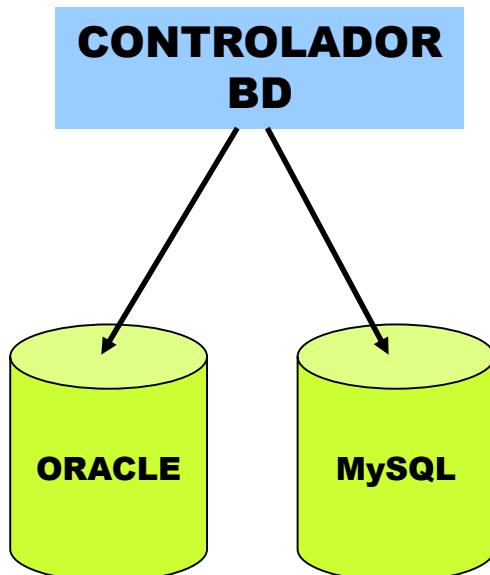
# Estructura. Variaciones

- **Variaciones del patrón:**
- **Creador:** Puede proporcionar una implementación estándar para los métodos de fabricación. Por ello, Creador no tiene que ser una clase abstracta o una interfaz, sino una clase completamente definida.
- **Producto:** Puede ser implementado como una clase abstracta. Debido a que Producto ya es una clase, puede incluir implementaciones para los otros métodos.
- El método de fabricación puede tomar algún parámetro para poder crear varios tipos de productos. Esto reduce el número de métodos de fabricación necesarios.

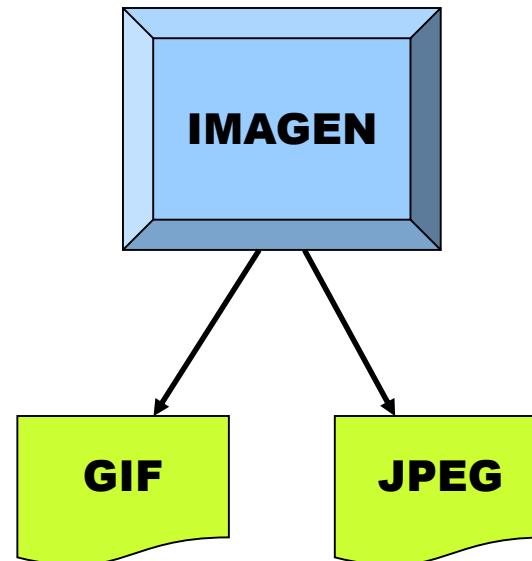


# Ejemplos

- **JDBC**



- **Imágenes**



# Consecuencias

1. **Proporciona enlaces para las subclases.** Crear objetos dentro de una clase con un método de fabricación es siempre más flexible que hacerlo directamente. El Factory Method da a las subclases un punto de enlace para proveer una versión extendida de un objeto.
2. **Conectar jerarquías de clases paralelas.** Las jerarquías de clases paralelas se producen cuando una clase delega alguna de sus responsabilidades a una clase separada. Los métodos factoría pueden ser llamados por los clientes, no sólo por los Creadores.
  - Ejemplo: Manipuladores de figuras: un tipo de manipulador con varios métodos factoría para cada tipo de figura



# Patrones relacionados

- **Prototipo.** Factory Method es un patrón de creación de clase, esto quiere decir que utiliza la herencia para decidir qué objeto instanciar. Mientras que el Prototipo es un patrón de creación de objeto porque delega la instanciación a otro objeto. En contraste, Prototipo usa, solo una clase “factory” de creación ( encapsulada en el prototipo) y permite la creación dinámica de objetos.
- **Abstract Factory.** El patrón Factory Method proporciona una interfaz para producir objetos, pero el tipo exacto de objeto que produce depende de qué subclase exacta es usada por el cliente. Es similar a Abstract Factory en que los métodos de Abstract Factory pueden ser implementados como métodos de factoría. La principal diferencia es que mientras Abstract Factory trata con una familia de productos, el Factory Method solo trata con un único producto. **El Factory Method ofrece una interfaz para crear un objeto, no una familia de objetos relacionados, como en Abstract Factory.**

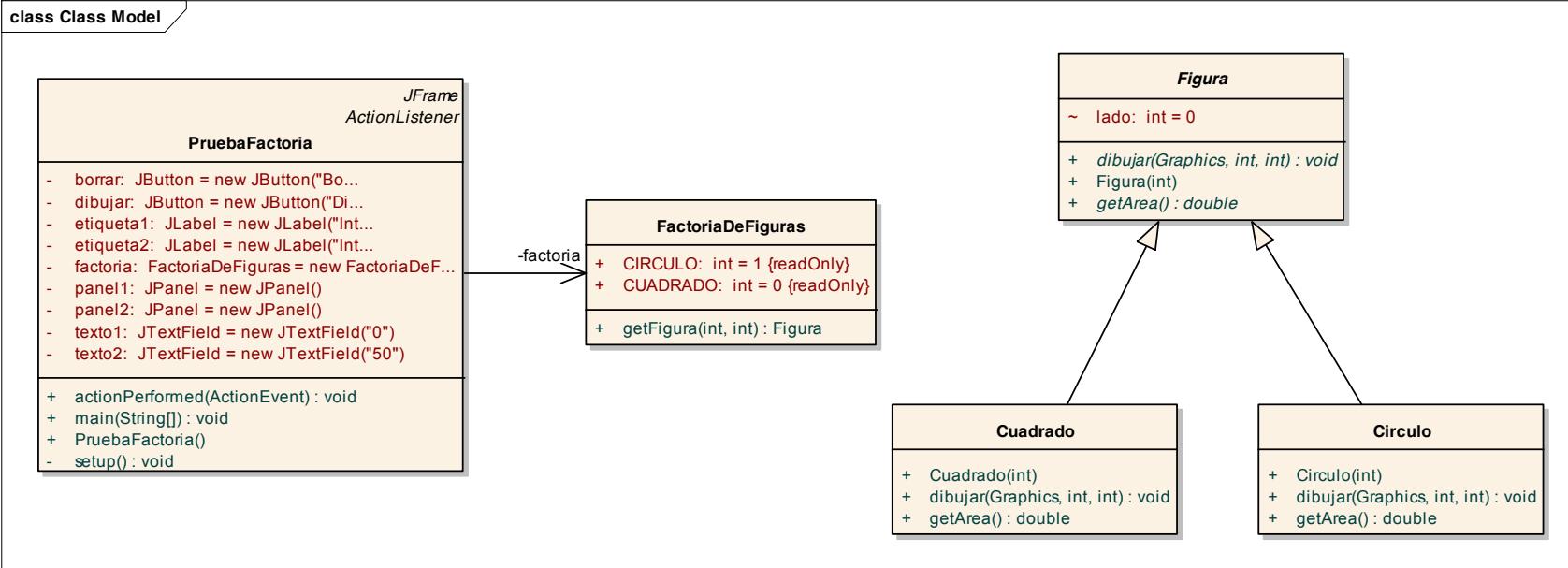


# Código de ejemplo

## Dibujo de Figuras



# Código de ejemplo



# Código de ejemplo

- Identificamos a continuación los elementos del patrón:
  - Producto: Figura.
  - ProductoConcreto: Cuadrado, Circulo.
  - Creador: FactoriaDeFiguras.
- Nos encontramos en este ejemplo con la variación del patrón en la que el Creador proporciona una implementación estándar para los métodos de fabricación. Creador no es una clase abstracta o una interfaz, sino una clase completamente definida (ver transparencia 7: Estructura. Variaciones).



*Patrones de Diseño:*  
*Patrones de Creación.*

*Tema 3-5:*  
*Prototype*



# Descripción del patrón

- **Nombre:**
  - Prototipo
- **Propiedades:**
  - Tipo: creación
  - Nivel: objeto, clase única
- **Objetivo o Propósito:**
  - Especifica los tipos de objetos a crear por medio de una instancia prototípica, y crea nuevos objetos copiando dicho prototipo.
  - Este patrón se usa en los casos en los que crear una instancia de una clase sea un proceso muy complejo y requiera mucho tiempo. Lo que hace es crear una instancia original, y, cuando necesitemos otra, en lugar de volver a crearla, copiar esa original y modificarla.

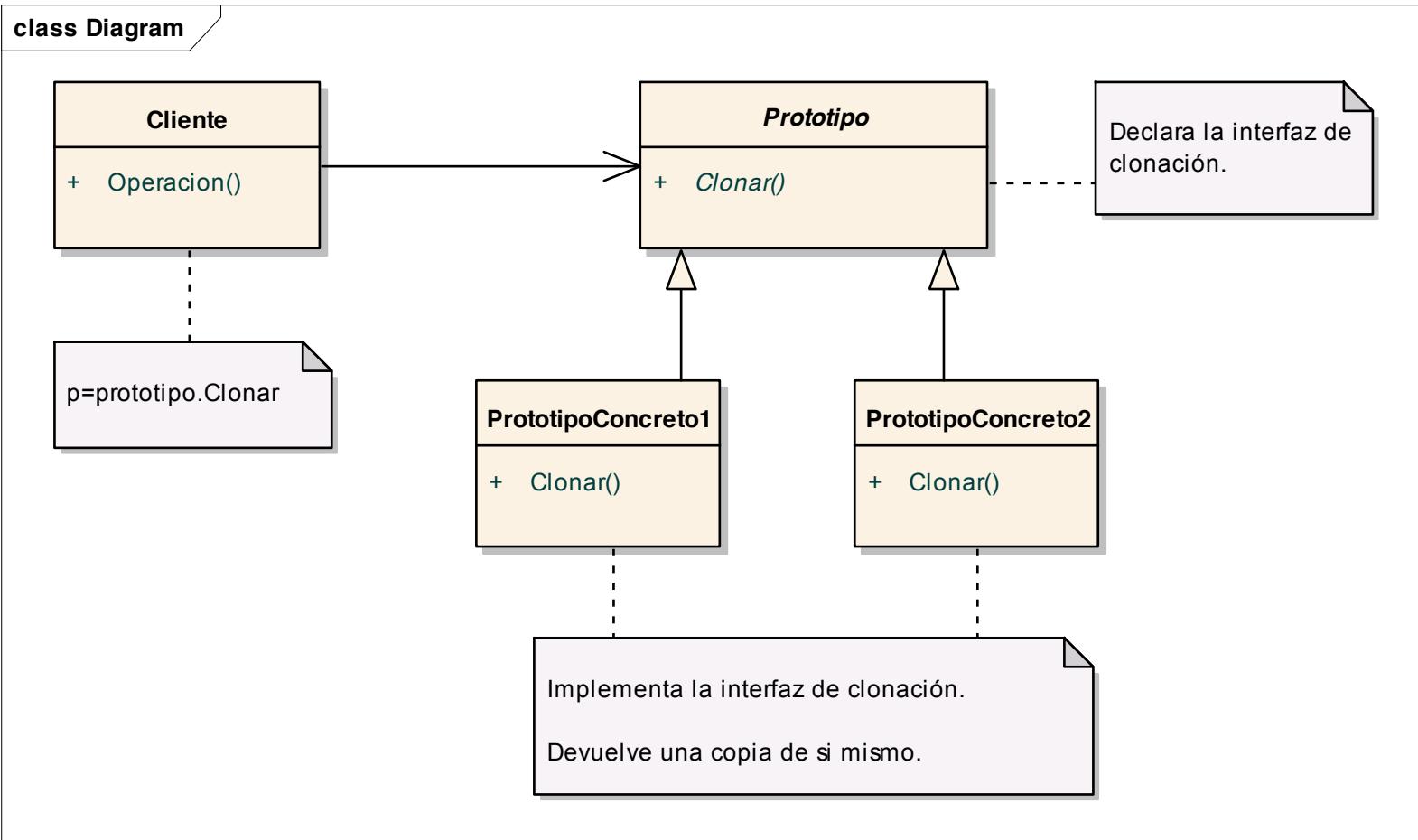


# Aplicabilidad

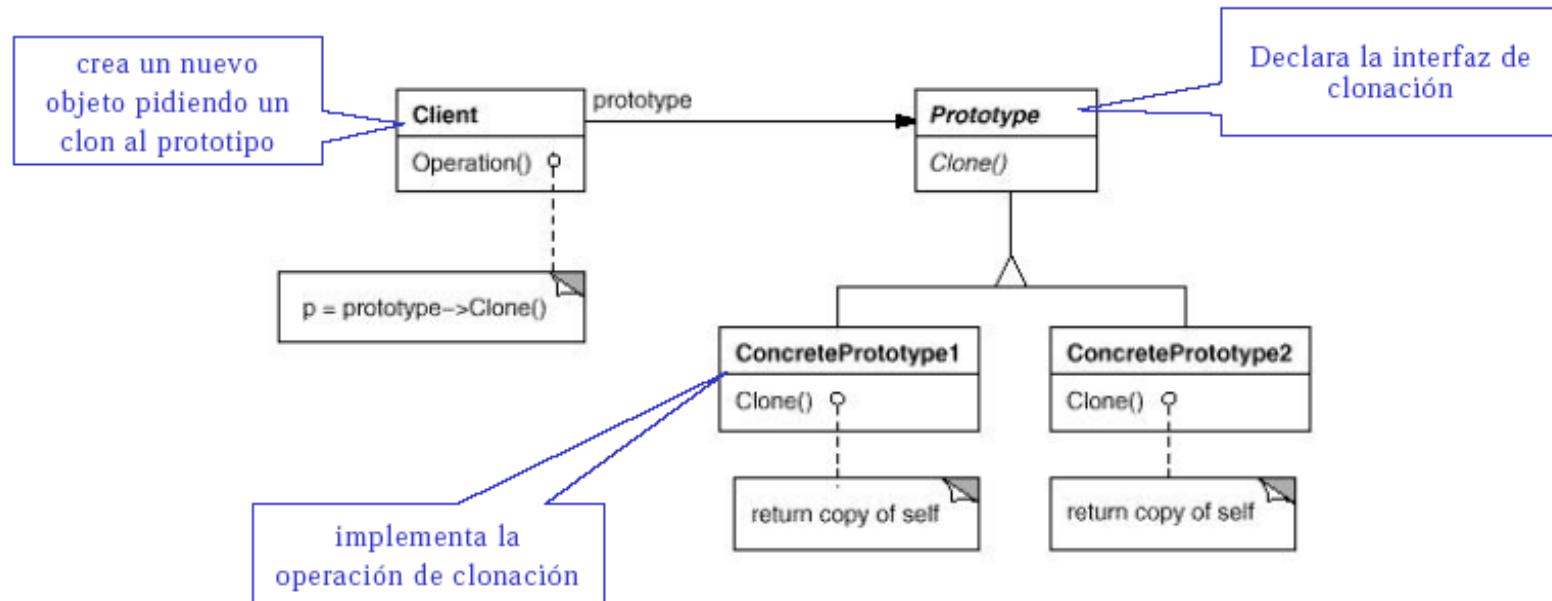
- Use el patrón Prototype cuando:
  - Un sistema deba ser independiente de cómo se crean, se componen y se representan sus productos.
  - Cuando las clases a instanciar sean especificadas en tiempo de ejecución.
  - Para evitar construir una jerarquía de clases de fábrica paralela a la jerarquía de clases de los productos.
  - Cuando las instancias de una clase puedan tener uno de entre sólo unos pocos estados diferentes. Puede ser más adecuado tener un número equivalente de prototipos y clonarlos, en vez de crear manualmente instancias de la clase cada vez con el estado apropiado.
- Ejemplo: Un listado muy largo de una base de datos ordenado por distintos criterios ¿Qué es mejor?, ¿leer de nuevo la base de datos entera o hacer una copia de la primera lectura y ordenarla? Obviamente lo más adecuado es la segunda.



# Estructura



# Estructura



# Estructura. Participantes

- **Prototipo:** Declara una interfaz para clonarse. Esta clase define la interfaz para crear clones de sí mismo. Típicamente esto involucra definir una función `clone()` que devuelve una copia del objeto original.
- **PrototipoConcreto:** Un Prototipo Concreto es una clase que implementa la operación de clonar definida en la clase Prototipo. Esto involucra copiar los valores y el estado del objeto original.
- **Cliente:** Crea un nuevo objeto pidiéndole a un prototipo que se clone.



# Estructura. Variaciones

- **Variaciones del patrón:**
- **Constructor de copia:** Este tipo de constructores toma una instancia de la misma clase como parámetro y devuelve una nueva copia con los mismos valores que el parámetro. Es posible tener un constructor al que se le pasan dos parámetros, el objeto a ser copiado y un valor booleano para indicar si debe aplicarse copia superficial o profunda.
- **Método Clone:** Java define un método en la clase Object llamado clone para hacer copias de objetos.



# Método clone

- Todas las clases en Java heredan un método de la clase *Object* llamado *clone*. Un método *clone* de un objeto retorna una copia de ese objeto. Esto solamente se hace para instancias de clases que dan permiso para ser clonadas.
- El método *clone()* es declarado “protected” (por defecto sólo puede ser llamado por el objeto que lo contenga, un objeto en el mismo paquete, u otro objeto del mismo tipo o subtipo). Si necesitamos hacer que un objeto pueda ser clonado por cualquiera, tenemos que redefinir su método *clone()* y hacerlo público.
- Una clase da permiso para que su instancia sea clonada si, y solo si, ella implementa el interface *Cloneable*.



# Método clone

- Hay dos estrategias básicas para implementar la operación *clone*:
  1. **Copia superficial** significa que las variables de los objetos clonados contienen los mismos valores que las variables del objeto original y que todas las referencias al objeto son a los mismos objetos. Es decir, la copia superficial copia solamente el objeto que será clonado, no los objetos a los que se refiere.
  2. **Copia profunda** significa que las variables de los objetos clonados contienen los mismos valores que las variables del objeto original, excepto que estas variables que se refieren a objetos realizan copias de los objetos referenciados por el objeto original. Es decir, la copia profunda copia el objeto que será clonado y los objetos a los que se referencia.
- Implementar la copia profunda puede ser delicado. Se necesita decidir si se quiere hacer copia profunda o superficial de los objetos copiados indirectamente. También hay que tener cuidado con el manejo de las referencias circulares.



# Consecuencias

- Como los otros patrones de creación de objetos (Abstract Factory y Builder), el Prototipo **aísla las clases de productos concretos del cliente**. Esto reduce el número de clases conocidas por el cliente y le permite trabajar con diferentes clases concretas sin ser modificado.
- Haciendo uso del patrón Prototipo, **podemos añadir y quitar productos en tiempo de ejecución**, según se vayan necesitando, mediante la clonación. Podemos ajustar la representación interna de los valores de una clase en tiempo de ejecución basándonos en las condiciones del programa. Además, podemos especificar nuevos objetos en tiempo de ejecución, sin crear una proliferación de clases y estructuras de herencia.



# Consecuencias

- Otra manera de crear nuevos objetos es **variando sus estructuras**. Si implementamos `clone()` como copia profunda, se pueden añadir diferentes estructuras al conjunto de prototipos disponibles.
- Si tenemos clases de prototipo que copiar implica que tenemos el acceso suficiente a sus valores o métodos para cambiarlas después de ser clonadas. Esto puede requerir añadir métodos de acceso a los valores de las clases de estos prototipos para que podamos modificar los valores una vez que hayamos clonado la clase.
- Podemos crear un **registro de clases prototípicas** que pueden ser clonadas y pedir al objeto registro una lista de los posibles prototipos. A lo mejor podemos clonar una clase existente mejor que escribirla desde cero.



# Consecuencias

- **Reduce la herencia.** El patrón Factory Method suele producir una jerarquía de clases Creador que es paralela a la jerarquía de clases de productos. El patrón Prototype permite clonar un prototipo en vez de decirle a un método de fabricación que cree un nuevo objeto. Por tanto no es necesaria una jerarquía de clases creador.
- El principal **inconveniente** del patrón Prototype es que cada subclase de Prototipo debe implementar la operación clonar, lo cual puede ser difícil. Por ejemplo, si las clases ya existen, es difícil añadir el método clonar, sobre todo si incluyen objetos que no pueden clonarse o que tienen referencias cíclicas.



# Patrones relacionados

- **Abstract Factory.** Una factoría abstracta podría almacenar un conjunto de prototipos a partir de los cuales clonar y devolver objetos de productos.
- **Factory Method.** El patrón Factory Method podría ser una alternativa al patrón Prototipo cuando la paleta de objetos prototípicos no contenga más de un objeto.
- **Builder.** El Builder se centra en la creación paso a paso de objetos complejos. Se crea una clase Builder por cada clase de producto que encapsula la complejidad de crear el objeto. La ventaja del Prototipo es, a parte de la necesidad de sólo una clase “factory”, el hecho de que la lógica de la creación está unida directamente a cada clase de producto.
- **Singleton.** Puede ser usado en la implementación del patrón Prototipo.



# Conclusiones

- Como otros patrones de creación, el patrón Prototipo como mejor se aplica es en los casos en los que el sistema debe ser independiente de cómo sus objetos son creados, compuestos y representados.
- El patrón Prototipo es útil en los casos en los que la construcción e inicialización de un objeto resulta más compleja o consume más tiempo que el que supone clonar un objeto existente.
- Permite añadir y quitar clases de productos en tiempo de ejecución clonándolos según se vayan necesitando. Además podemos especificar nuevos objetos en tiempo de ejecución sin crear una proliferación de clases y de estructuras de herencia.
- La utilidad de este patrón, puede aumentar si lo combinamos con el patrón Abstract Factory, o si creamos un registro en el que almacenar y gestionar todos los prototipos de un sistema.



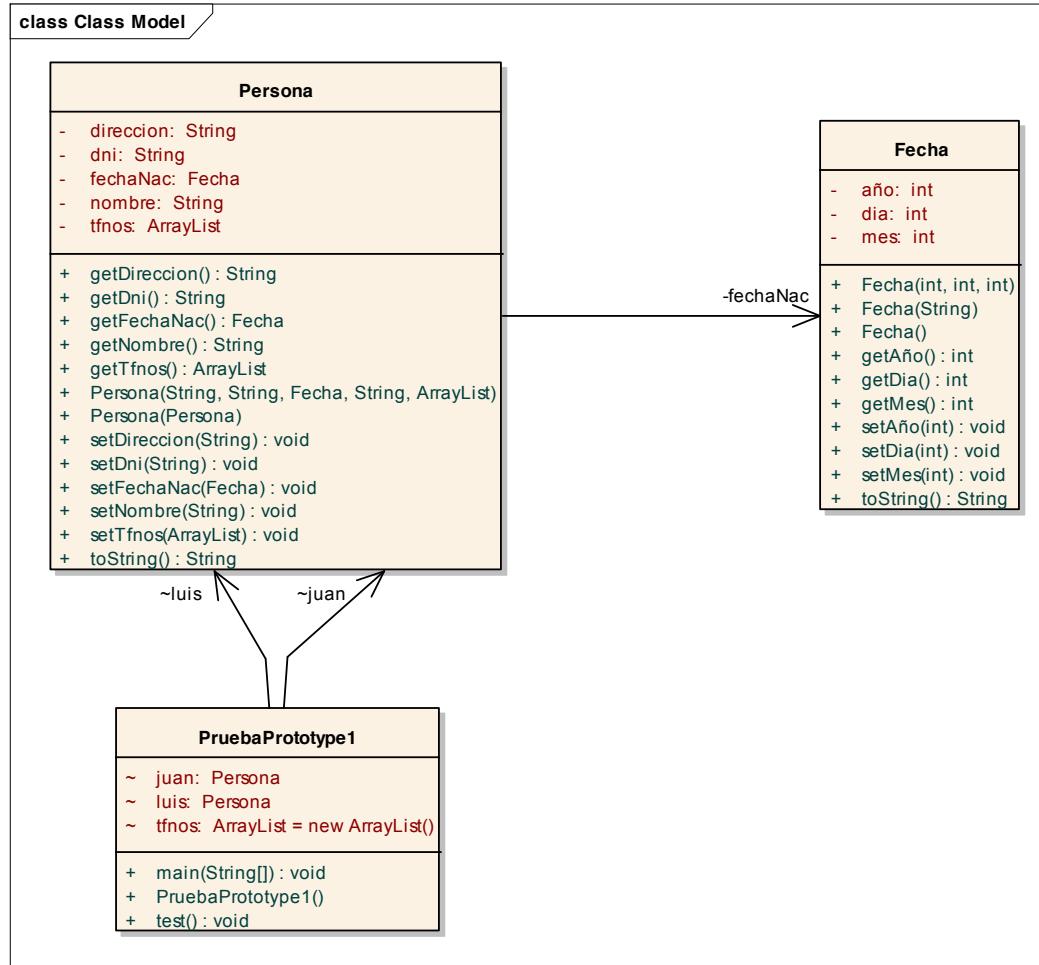
# Código de ejemplo

CLONACIÓN  
PERSONAS



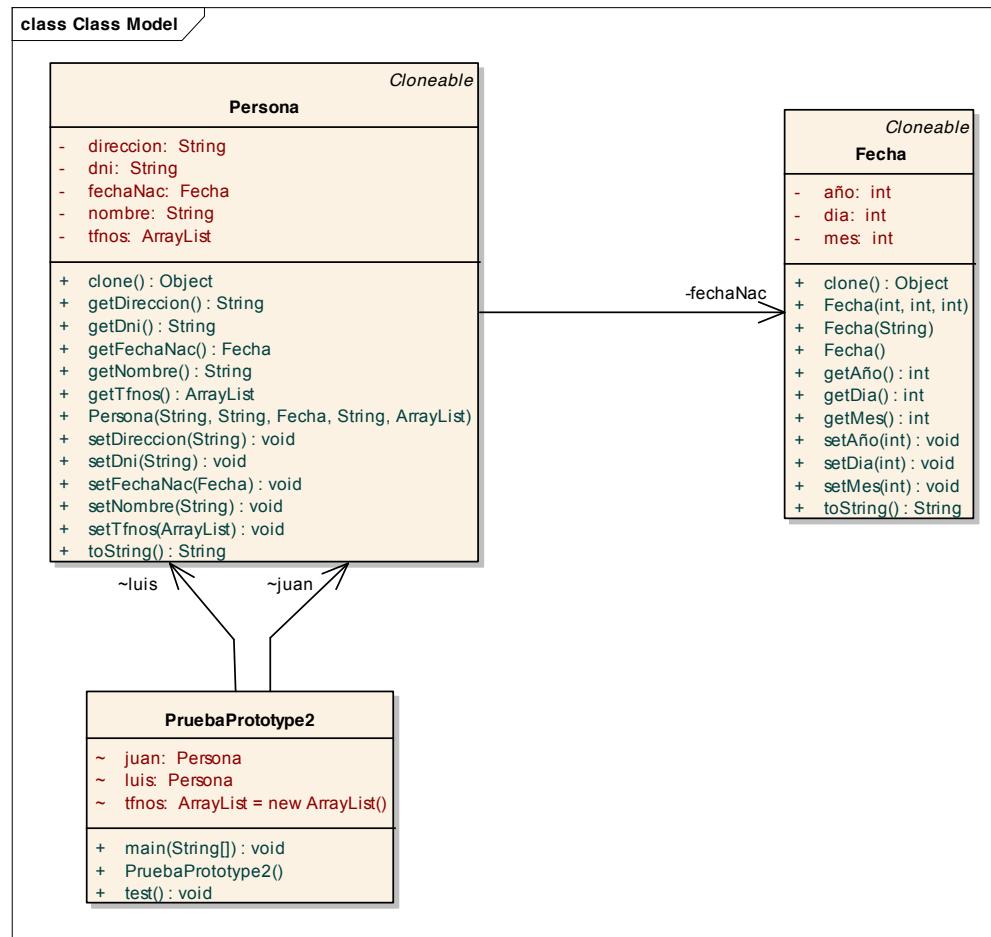
# Código de ejemplo. V1

## CONSTRUCTOR DE COPIA:



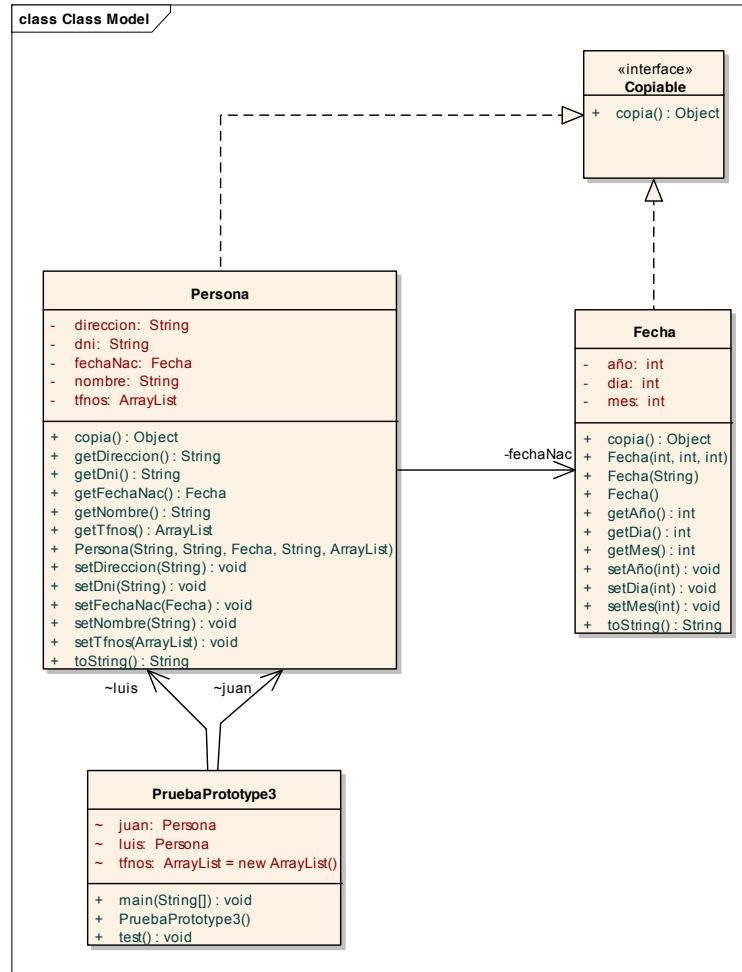
# Código de ejemplo. V2

## MÉTODO CLONE:



# Código de ejemplo. V3

## MÉTODO COPIA:



# Código de ejemplo. V3

- Identificamos a continuación los elementos del patrón:
  - Prototipo: interfaz Copiable.
  - PrototipoConcreto: Persona, Fecha.
  - Cliente: PruebaPrototipo3.





*Patrones de Diseño:  
Patrones de Creación.  
Tema 3-6:  
Singleton*

# Descripción del patrón

- **Nombre:**
  - Único
- **Propiedades:**
  - Tipo: creación
  - Nivel: objeto
- **Objetivo o Propósito:**
  - Garantiza que una clase solo tenga una instancia, y proporciona un punto de acceso global a ella. Todos los objetos que utilizan una instancia de esa clase usan la misma instancia.

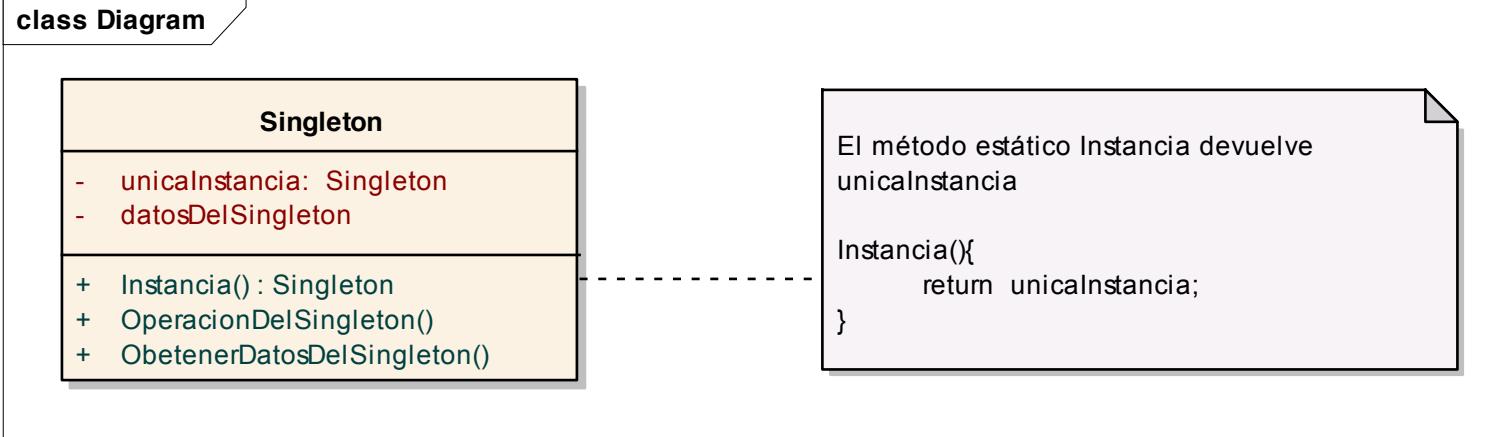


# Aplicabilidad

- Use el patrón Singleton cuando:
  - Sólo puede haber una instancia de una clase, y debe ser accesible a los clientes desde un punto de acceso bien conocido.
  - La única instancia debería ser extensible por herencia, y los clientes deberían poder usar una instancia extendida sin modificar su código.



# Estructura



# Estructura



# Estructura. Participantes

- **Singleton:** Define una operación `Instancia` que permite a los clientes acceder a su única instancia. `Instancia` es una operación estática (de clase).
- Puede ser responsable de crear su única instancia.



# Estructura. Variaciones

- **Variaciones del patrón:**
- Se puede tener más de una instancia dentro de la clase. Aquellos objetos que conocen la existencia de múltiples instancias pueden utilizar algún método para obtener una de ellas. Ventaja: el resto de la aplicación permanece inalterada.
- El método de acceso al Singleton puede ser el punto de acceso al conjunto total de instancias, aunque todas tengan un tipo distinto. El método de acceso puede determinar, en tiempo de ejecución, qué tipo de instancia devolver.



# Estructura. Código

```
public class Singleton
{
    private static Singleton instancia;

    private Singleton() {}

    public static Singleton getInstancia()
    {
        if (instancia == null) {
            instancia = new Singleton();
        }
        return instancia;
    }
}
```



# Estructura. Código

```
public class SingletonOpt
{
    private static SingletonOpt instancia = new SingletonOpt();

    private SingletonOpt() {}

    public static SingletonOpt getInstancia()
    {
        return instancia;
    }
}
```



# Consecuencias

- Existe exactamente una instancia de una clase Singleton.
- **Acceso controlado a la única instancia.** Otras clases que quieran una referencia a la única instancia de la clase Singleton conseguirán esa instancia llamando al método estático *getInstancia* de la clase.
- **Espacio de nombres reducido.** Este patrón es una mejora sobre las variables globales. Evita contaminar el espacio de nombres con variables globales que almacenen las instancias.



# Consecuencias

- **Permite un número variable de instancias.** El patrón permite la creación de más de una instancia de la clase Singleton, facilitando el control del número de instancias usadas en la aplicación. Solo hay que cambiar la operación de acceso a la instancia del Singleton.
- Tener **subclases** de una clase Singleton es complicado y resultan clases imperfectamente encapsuladas. Una clase derivada de un Singleton no es un Singleton. Para hacer subclases de una clase Singleton, se debería tener un constructor que no sea privado.



# Patrones relacionados

- **Abstract Factory.** Las clases Factorías Concretas (Concrete Factory) son implementadas a menudo como clases Singleton.
- **Factory Method.** Una factoría por aplicación, es un ejemplo perfecto de un Singleton.
- **Builder**, que es usado para construir objetos complejos, mientras que el Singleton es usado para crear un objeto accesible globalmente.
- **Prototipo**, que es usado para copiar un objeto, o crear un objeto a partir de un prototipo, mientras que un Singleton es usado para asegurar que solo se garantiza un prototipo.



# Código de ejemplo

Control de Entrada



# Código de ejemplo

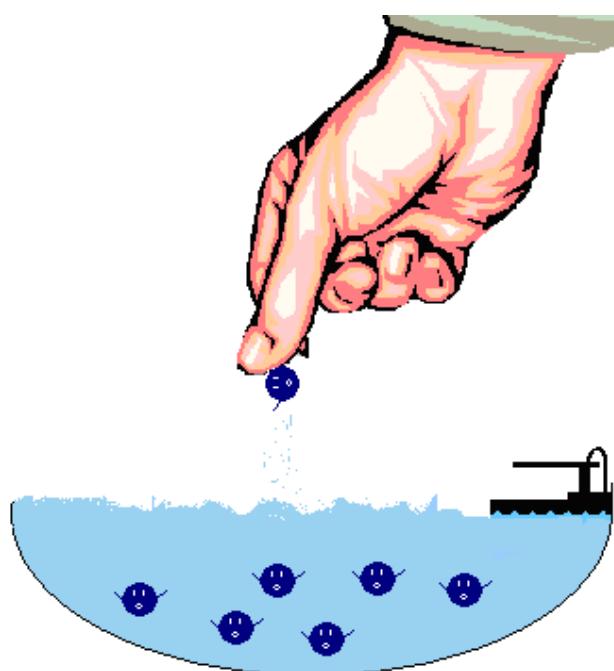
```
class Class Model

    ContadorLogin
        - instancia: ContadorLogin
        - usuarios: ArrayList<String> = new ArrayList<S...
        + borrarLogin(String) : void
        - ContadorLogin()
        + devolverEstadoCuenta(String) : boolean
        + getInstancia() : ContadorLogin

    ControlEntrada
        - entrada: BufferedReader = new BufferedReader...
        - res: String
        - usuario: String
        + main(String[]) : void
```



# *Anexo: Object Pool*



# Descripción del patrón

- El Patrón Object Pool gestiona la reutilización de objetos para una clase cuyas instancias son caras de crear o de la que solo podemos crear un número limitado de objetos.
- Una vez visto el patrón Singleton y sabiendo lo que es una clase Singleton nos será muy sencillo entender este patrón. El patrón Object Pool lleva como base el patrón Singleton.



# Aplicabilidad

- Tengamos que programar una aplicación para proporcionar acceso a una base de datos dada. Los clientes mandarán peticiones a la base de datos a través de una conexión de red. El servidor de la base de datos recibirá las peticiones a través de la conexión de red y devolverá los resultados a través de la misma conexión.
- Crear conexiones a bases de datos que no son necesarias es malo por las siguientes razones:
  - Crear cada conexión puede llevar unos cuantos segundos.
  - Cuantas más conexiones tengamos establecidas, más tiempo lleva el crear nuevas conexiones.
  - Cada conexión usa una conexión de red. Algunas plataformas limitan el número de conexiones de red que permiten.

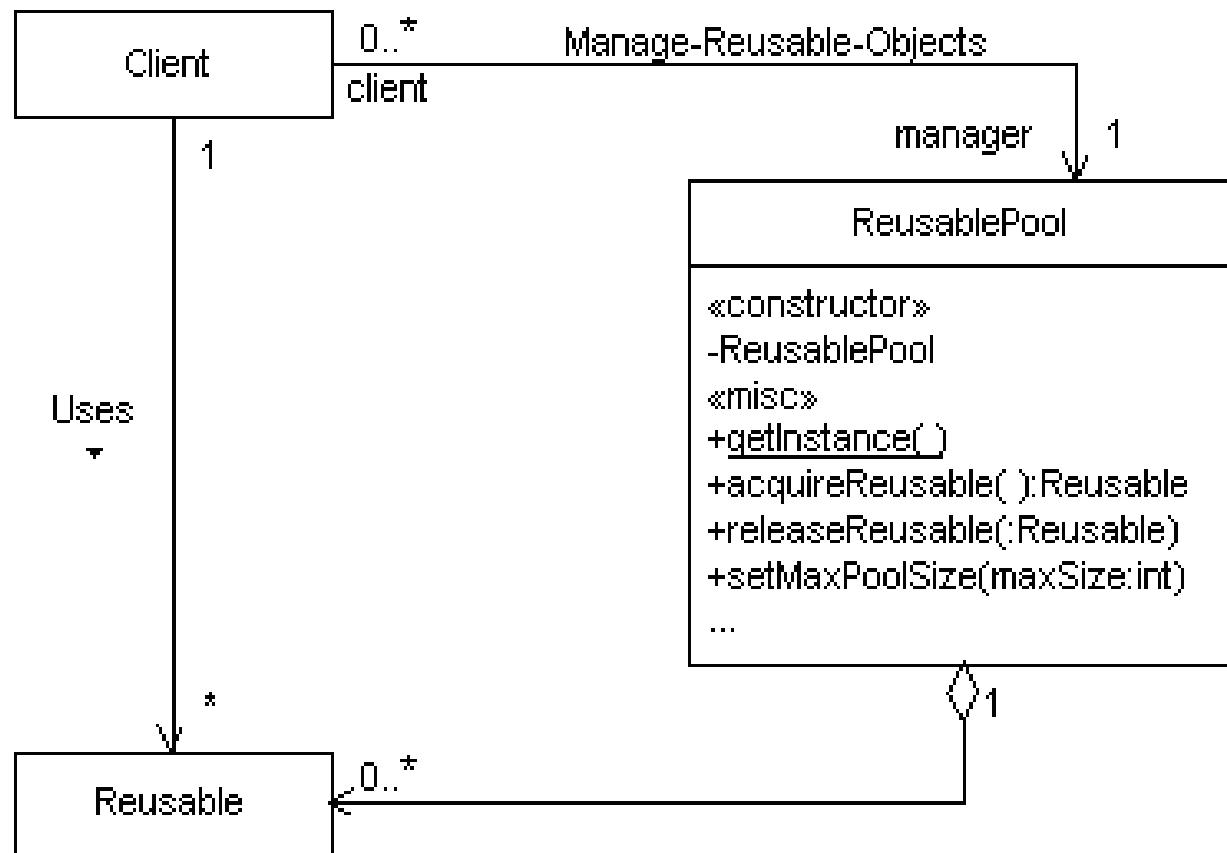


# Aplicabilidad

- Por lo tanto el patrón evita el gasto de crear objetos de conexión y pone un límite en su número.
- La estrategia que usa el patrón para manejar las conexiones a la base de datos se basará en la premisa de que las conexiones son intercambiables. Tan pronto como necesitemos hacer una petición a una base de datos, no importa cuál de las conexiones de un programa se use.



# Estructura



# Estructura. Participantes

- **Reusable.** Sus instancias colaboran con otros objetos durante un tiempo limitado, después ya no van a ser necesitados para esa colaboración.
- **Client.** Sus instancias utilizan los objetos Reusables.
- **ReusablePool.** Sus instancias manejan los objetos Reusables para ser usados por objetos Client. Se mantienen objetos Reusable que no están siendo utilizados en la misma piscina de objetos de forma que puedan ser utilizados. Está diseñada como una clase Singleton. Su constructor(es) es privado, lo que fuerza a que las otras clases tengan que llamar a su método getInstance para conseguir una instancia de la misma.



# Estructura. Participantes

- Un objeto ReusablePool mantiene una colección de objetos Reusable.
- Un Cliente llama al método acquireReusable cuando necesita un objeto Reusable. Situaciones:
  - Si hay algún objeto Reusable en la piscina, lo saca y lo devuelve.
  - Si la piscina está vacía, se crea un objeto Reusable si se puede. Si no puede crear un nuevo objeto Reusable, entonces espera hasta que alguno sea devuelto.
- Un Cliente llama al método releaseReusable cuando ya ha terminado con el objeto, devolviéndolo a la piscina de objetos.



# Consecuencias

- Los Clientes no crean objetos. La creación y destrucción de objetos es manejada por una piscina centralizada.
- Los Clientes piden objetos de la piscina y devuelven objetos a la piscina en lugar de destruirlos. Se evita la sobrecarga de crear y destruir objetos frecuentemente.
- Algunas veces, el sistema solo puede dar soporte a un número limitado de objetos. La piscina de objetos se encarga de este límite limitando el número de objetos que hay dentro de la piscina



# Código de ejemplo

