



## Ayuda a Ucrania a detener a Rusia

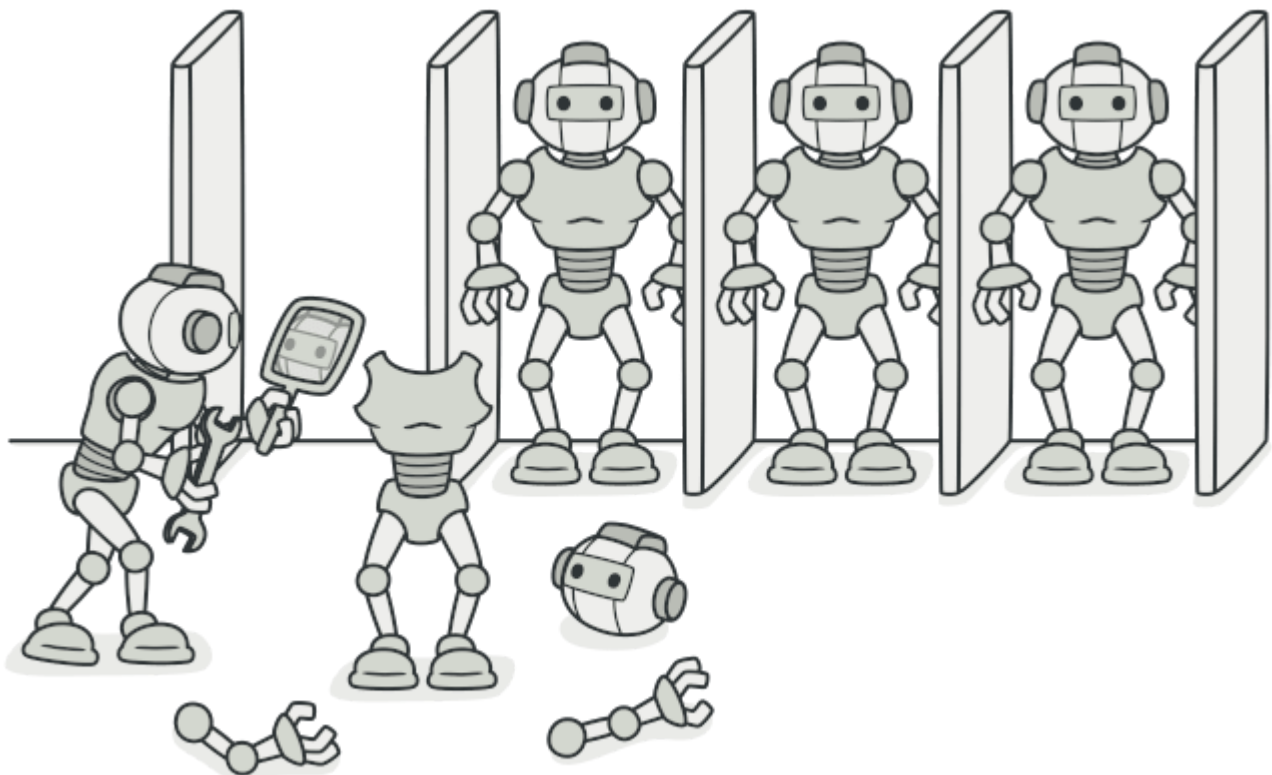
[🏠](#) / [Patrones de diseño](#) / [Patrones creacionales](#)

# Prototype

**También llamado:** Prototipo, Clon, Clone

## Propósito

**Prototype** es un patrón de diseño creacional que nos permite copiar objetos existentes sin que el código dependa de sus clases.



## Problema

Digamos que tienes un objeto y quieres crear una copia exacta de él. ¿Cómo lo harías? En primer lugar, debes crear un nuevo objeto de la misma clase. Después debes recorrer todos los campos

del objeto original y copiar sus valores en el nuevo objeto.

¡Bien! Pero hay una trampa. No todos los objetos se pueden copiar de este modo, porque algunos de los campos del objeto pueden ser privados e invisibles desde fuera del propio objeto.



No siempre es posible copiar un objeto “desde fuera”.

Hay otro problema con el enfoque directo. Dado que debes conocer la clase del objeto para crear un duplicado, el código se vuelve dependiente de esa clase. Si esta dependencia adicional no te da miedo, todavía hay otra trampa. En ocasiones tan solo conocemos la interfaz que sigue el objeto, pero no su clase concreta, cuando, por ejemplo, un parámetro de un método acepta cualquier objeto que siga cierta interfaz.

## 😊 Solución

El patrón Prototype delega el proceso de clonación a los propios objetos que están siendo clonados. El patrón declara una interfaz común para todos los objetos que soportan la clonación. Esta interfaz nos permite clonar un objeto sin acoplar el código a la clase de ese objeto. Normalmente, dicha interfaz contiene un único método `clonar`.

La implementación del método `clonar` es muy parecida en todas las clases. El método crea un objeto a partir de la clase actual y lleva todos los valores de campo del viejo objeto, al nuevo. Se puede incluso copiar campos privados, porque la mayoría de los lenguajes de programación

permite a los objetos acceder a campos privados de otros objetos que pertenecen a la misma clase.

Un objeto que soporta la clonación se denomina *prototipo*. Cuando tus objetos tienen decenas de campos y miles de configuraciones posibles, la clonación puede servir como alternativa a la creación de subclases.

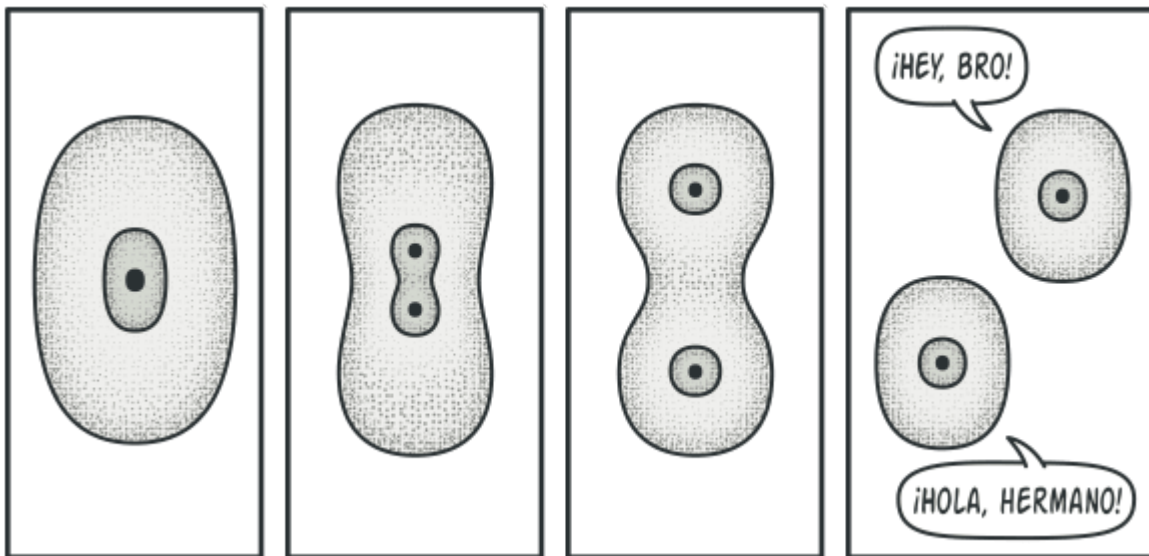


*Los prototipos prefabricados pueden ser una alternativa a las subclases.*

Funciona así: se crea un grupo de objetos configurados de maneras diferentes. Cuando necesites un objeto como el que has configurado, clonas un prototipo en lugar de construir un nuevo objeto desde cero.

## Analogía del mundo real

En la vida real, los prototipos se utilizan para realizar pruebas de todo tipo antes de comenzar con la producción en masa de un producto. Sin embargo, en este caso, los prototipos no forman parte de una producción real, sino que juegan un papel pasivo.

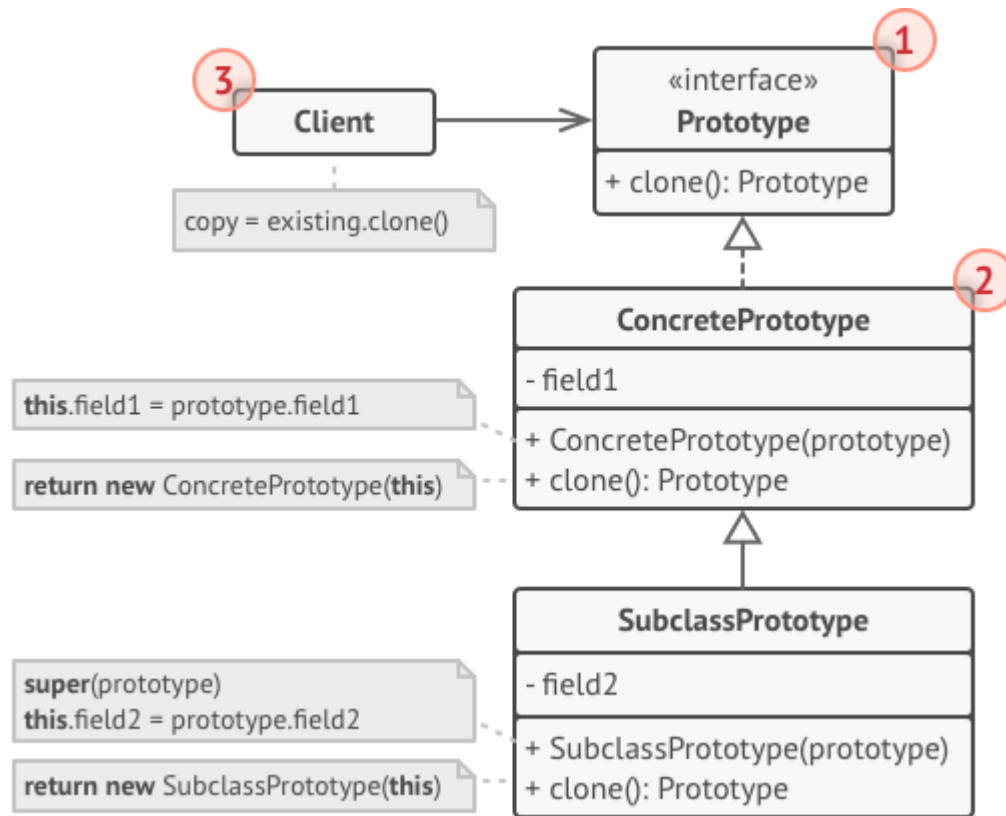


*La división de una célula.*

Ya que los prototipos industriales en realidad no se copian a sí mismos, una analogía más precisa del patrón es el proceso de la división mitótica de una célula (biología, ¿recuerdas?). Tras la división mitótica, se forma un par de células idénticas. La célula original actúa como prototipo y asume un papel activo en la creación de la copia.

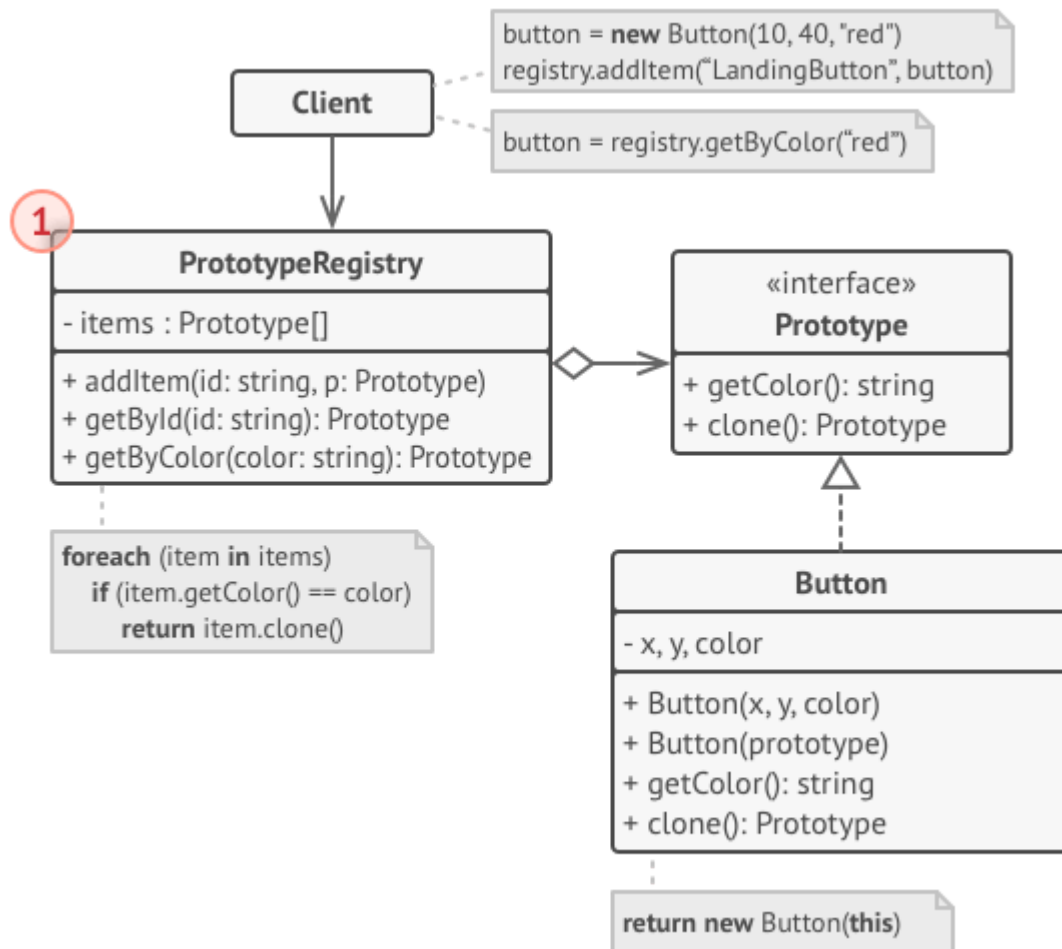
## Estructura

### Implementación básica



1. La interfaz **Prototipo** declara los métodos de clonación. En la mayoría de los casos, se trata de un único método `clonar`.
2. La clase **Prototipo Concreto** implementa el método de clonación. Además de copiar la información del objeto original al clon, este método también puede gestionar algunos casos extremos del proceso de clonación, como, por ejemplo, clonar objetos vinculados, deshacer dependencias recursivas, etc.
3. El **Ciente** puede producir una copia de cualquier objeto que siga la interfaz del prototipo.

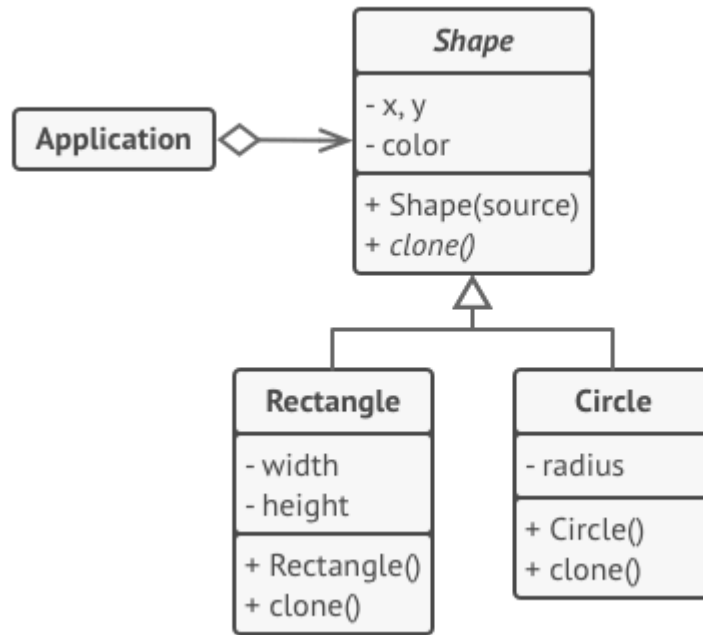
## Implementación del registro de prototipos



1. El **Registro de Prototipos** ofrece una forma sencilla de acceder a prototipos de uso frecuente. Almacena un grupo de objetos prefabricados listos para ser copiados. El registro de prototipos más sencillo es una tabla *hash* con los pares `name → prototype`. No obstante, si necesitas un criterio de búsqueda más preciso que un simple nombre, puedes crear una versión mucho más robusta del registro.

## # Pseudocódigo

En este ejemplo, el patrón **Prototype** nos permite producir copias exactas de objetos geométricos sin acoplar el código a sus clases.



*Clonación de un grupo de objetos que pertenece a una jerarquía de clase.*

Todas las clases de forma siguen la misma interfaz, que proporciona un método de clonación. Una subclase puede invocar el método de clonación padre antes de copiar sus propios valores de campo al objeto resultante.

```

// Prototipo base.
abstract class Shape is
    field X: int
    field Y: int
    field color: string

    // Un constructor normal.
    constructor Shape() is
        // ...

    // El constructor prototipo. Un nuevo objeto se inicializa
    // con valores del objeto existente.
    constructor Shape(source: Shape) is
        this()
        this.X = source.X
        this.Y = source.Y
        this.color = source.color

    // La operación clonar devuelve una de las subclases de
    // Shape (Forma).
    abstract method clone():Shape
  
```

```
// Prototipo concreto. El método de clonación crea un nuevo
// objeto y lo pasa al constructor. Hasta que el constructor
// termina, tiene una referencia a un nuevo clon. De este modo
// nadie tiene acceso a un clon a medio terminar. Esto garantiza
// la consistencia del resultado de la clonación.
```

```
class Rectangle extends Shape is
```

```
    field width: int
```

```
    field height: int
```

```
constructor Rectangle(source: Rectangle) is
```

```
    // Para copiar campos privados definidos en la clase
```

```
    // padre es necesaria una llamada a un constructor
```

```
    // padre.
```

```
    super(source)
```

```
    this.width = source.width
```

```
    this.height = source.height
```

```
method clone():Shape is
```

```
    return new Rectangle(this)
```

```
class Circle extends Shape is
```

```
    field radius: int
```

```
constructor Circle(source: Circle) is
```

```
    super(source)
```

```
    this.radius = source.radius
```

```
method clone():Shape is
```

```
    return new Circle(this)
```

```
// En alguna parte del código cliente.
```

```
class Application is
```

```
    field shapes: array of Shape
```

```
constructor Application() is
```

```
    Circle circle = new Circle()
```

```
    circle.X = 10
```

```
    circle.Y = 10
```

```
    circle.radius = 20
```

```
    shapes.add(circle)
```

```
    Circle anotherCircle = circle.clone()
```

```
    shapes.add(anotherCircle)
```

```
    // La variable `anotherCircle` (otroCírculo) contiene
```

```
    // una copia exacta del objeto `circle`.
```

```
    Rectangle rectangle = new Rectangle()
```



```
rectangle.width = 10
rectangle.height = 20
shapes.add(rectangle)
```

**method** `businessLogic()` **is**


```
// Prototype es genial porque te permite producir una
// copia de un objeto sin conocer nada de su tipo.
Array shapesCopy = new Array of Shapes.

// Por ejemplo, no conocemos los elementos exactos de la
// matriz de formas. Lo único que sabemos es que son
// todas formas. Pero, gracias al polimorfismo, cuando
// invocamos el método `clonar` en una forma, el
// programa comprueba su clase real y ejecuta el método
// de clonación adecuado definido en dicha clase. Por
// eso obtenemos los clones adecuados en lugar de un
// grupo de simples objetos Shape.
foreach (s in shapes) do
    shapesCopy.add(s.clone())


// La matriz `shapesCopy` contiene copias exactas del
// hijo de la matriz `shape`.
```

## Aplicabilidad

 Utiliza el patrón Prototype cuando tu código no deba depender de las clases concretas de objetos que necesites copiar.

 Esto sucede a menudo cuando tu código funciona con objetos pasados por código de terceras personas a través de una interfaz. Las clases concretas de estos objetos son desconocidas y no podrías depender de ellas aunque quisieras.

El patrón Prototype proporciona al código cliente una interfaz general para trabajar con todos los objetos que soportan la clonación. Esta interfaz hace que el código cliente sea independiente de las clases concretas de los objetos que clona.

 Utiliza el patrón cuando quieras reducir la cantidad de subclases que solo se diferencian en la forma en que inicializan sus respectivos objetos. Puede ser que alguien haya creado estas subclases para poder crear objetos con una configuración específica.

⚡ El patrón Prototype te permite utilizar como prototipos un grupo de objetos prefabricados, configurados de maneras diferentes.

En lugar de instanciar una subclase que coincida con una configuración, el cliente puede, sencillamente, buscar el prototipo adecuado y clonarlo.

## 📋 Cómo implementarlo

1. Crea la interfaz del prototipo y declara el método `clonar` en ella, o, simplemente, añade el método a todas las clases de una jerarquía de clase existente, si la tienes.
2. Una clase de prototipo debe definir el constructor alternativo que acepta un objeto de dicha clase como argumento. El constructor debe copiar los valores de todos los campos definidos en la clase del objeto que se le pasa a la instancia recién creada. Si deseas cambiar una subclase, debes invocar al constructor padre para permitir que la superclase gestione la clonación de sus campos privados.

Si el lenguaje de programación que utilizas no soporta la sobrecarga de métodos, puedes definir un método especial para copiar la información del objeto. El constructor es el lugar más adecuado para hacerlo, porque entrega el objeto resultante justo después de invocar el operador `new`.

3. Normalmente, el método de clonación consiste en una sola línea que ejecuta un operador `new` con la versión prototípica del constructor. Observa que todas las clases deben sobrescribir explícitamente el método de clonación y utilizar su propio nombre de clase junto al operador `new`. De lo contrario, el método de clonación puede producir un objeto a partir de una clase madre.
4. Opcionalmente, puedes crear un registro de prototipos centralizado para almacenar un catálogo de prototipos de uso frecuente.

Puedes implementar el registro como una nueva clase de fábrica o colocarlo en la clase base de prototipo con un método estático para buscar el prototipo. Este método debe buscar un prototipo con base en el criterio de búsqueda que el código cliente pase al método. El criterio puede ser una etiqueta tipo *string* o un grupo complejo de parámetros de búsqueda. Una vez encontrado el prototipo adecuado, el registro deberá clonarlo y devolver la copia al cliente.

Por último, sustituye las llamadas directas a los constructores de las subclases por llamadas al método de fábrica del registro de prototipos.

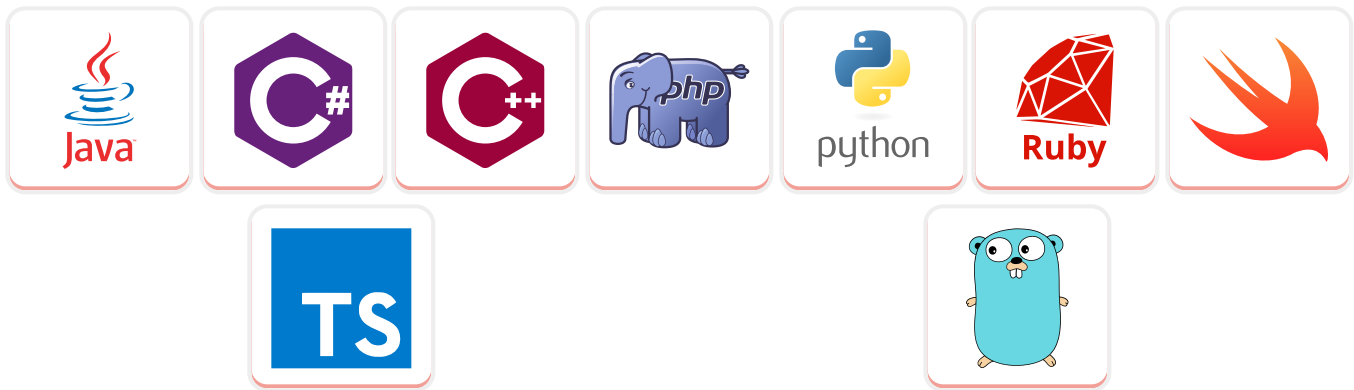
## Pros y contras

- ✓ Puedes clonar objetos sin acoplarlos a sus clases concretas.
- ✓ Puedes evitar un código de inicialización repetido clonando prototipos prefabricados.
- ✓ Puedes crear objetos complejos con más facilidad.
- ✓ Obtienes una alternativa a la herencia al tratar con preajustes de configuración para objetos complejos.
- ✗ Clonar objetos complejos con referencias circulares puede resultar complicado.

## ↔ Relaciones con otros patrones

- Muchos diseños empiezan utilizando el **Factory Method** (menos complicado y más personalizable mediante las subclases) y evolucionan hacia **Abstract Factory**, **Prototype**, o **Builder** (más flexibles, pero más complicados).
- Las clases del **Abstract Factory** a menudo se basan en un grupo de **métodos de fábrica**, pero también puedes utilizar **Prototype** para escribir los métodos de estas clases.
- **Prototype** puede ayudar a cuando necesitas guardar copias de **Comandos** en un historial.
- Los diseños que hacen un uso amplio de **Composite** y **Decorator** a menudo pueden beneficiarse del uso del **Prototype**. Aplicar el patrón te permite clonar estructuras complejas en lugar de reconstruirlas desde cero.
- **Prototype** no se basa en la herencia, por lo que no presenta sus inconvenientes. No obstante, *Prototype* requiere de una inicialización complicada del objeto clonado. **Factory Method** se basa en la herencia, pero no requiere de un paso de inicialización.
- En ocasiones, **Prototype** puede ser una alternativa más simple al patrón **Memento**. Esto funciona si el objeto cuyo estado quieres almacenar en el historial es suficientemente sencillo y no tiene enlaces a recursos externos, o estos son fáciles de restablecer.
- Los patrones **Abstract Factory**, **Builder** y **Prototype** pueden todos ellos implementarse como **Singletons**.

## </> Ejemplos de código



### ¡Apoya nuestro sitio web gratuito y compra el libro!

- 22 patrones de diseño y 8 principios explicados en profundidad
- 436 páginas bien estructuradas, fáciles de leer y libres de tecnicismos
- 225 ilustraciones y diagramas claros y útiles
- Un archivo con ejemplos de código en 9 lenguajes
- Todos los dispositivos soportados: Formatos PDF/EPUB/MOBI/KFX

 Saber más...

Inicio

Refactorización

Patrones de diseño



[Contenido Premium](#)

[Foro](#)

[Contáctanos](#)

© 2014-2022 Refactoring.Guru. Todos los derechos reservados

 Ilustraciones por Dmitry Zhart

 Khmelnytske shosse 19 / 27, Kamianets-Podilskyi, Ucrania, 32305

 Email: [support@refactoring.guru](mailto:support@refactoring.guru)

[Términos y condiciones](#)

[Política de privacidad](#)

[Política de uso de contenido](#)