**Prototype**

## **Intención**

**Prototype** es un patrón de diseño creativo que le permite producir nuevos objetos copiando los existentes sin comprometer sus elementos internos.

## **Problema**

Tienes un objeto y quieres crear un duplicado. ¿Como lo harias? Primero, necesitas crear un objeto nuevo de la misma clase. Luego, debe pasar por todos los campos del objeto original y copiar sus valores en el nuevo objeto.

¡Bonito! Pero hay una trampa. No todos los objetos se pueden copiar de esta manera. Algunos de los objetos pueden tener campos privados que no son accesibles desde el exterior.

Hay un problema más con este enfoque. Dado que debe conocer la clase de objetos para poder recorrer todos sus campos, su código dependerá de las clases de objetos que copie. No podrá copiar objetos si solo conoce sus interfaces, pero no las clases reales.

## **Solución**

El patrón de prototipo delega el proceso de clonación a los propios objetos.

Declara la interfaz común para todos los objetos que permiten la clonación. Permite la clonación de objetos sin acoplamiento estrecho a sus clases concretas. Por lo general, la interfaz prototipo contiene un solo clone método.

La implementación del clone método es similar para todas las clases. El método crea un objeto de una clase actual y copia todos sus valores de campo en el nuevo objeto. La mayoría de los lenguajes de programación permiten acceder a campos privados en un objeto de la misma clase, por lo que el proceso de copia es sencillo.

Los objetos que se pueden clonar se llaman prototipos. A veces, especialmente cuando sus objetos tienen una tonelada de campos y cientos de configuraciones posibles, los prototipos pueden servir como una alternativa a la sub-clasificación. En este caso, el programa crea un montón de prototipos de antemano y luego los clona en lugar de reconstruir los objetos desde cero.

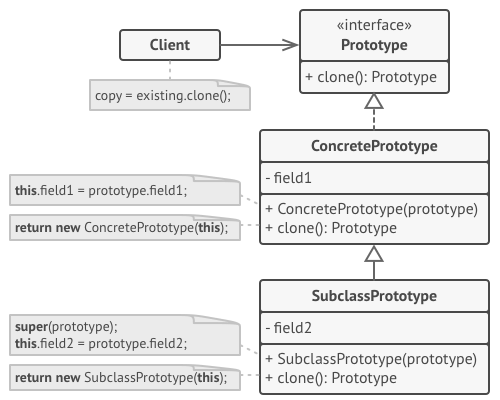
## **Analogía del mundo real**

**División de industria vs celular**

En la vida real, los prototipos se utilizan para realizar varias pruebas antes de comenzar la producción en masa de un producto. Sin embargo, en este caso, los prototipos no participan en la producción real, desempeñando un papel pasivo.

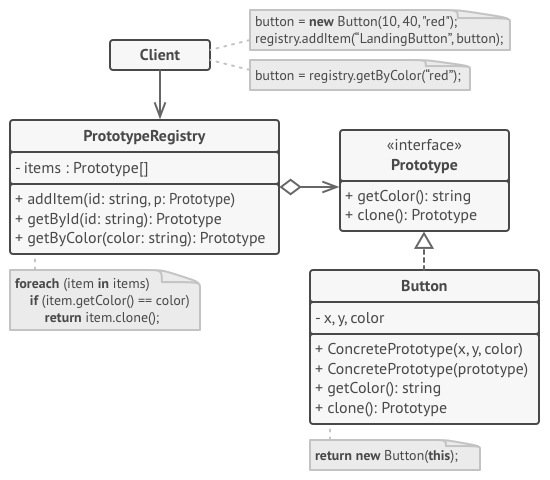
Dado que los prototipos industriales no se copian a sí mismos, la analogía mucho más estrecha del patrón es la división celular mitótica (biología, ¿recuerdas?). Después de tal división, se forman dos células completamente idénticas. La celda original actúa como un prototipo y desempeña un papel activo en la creación de una copia.

## **Estructura**



1. **Prototipo** declara la interfaz de clonación. En la mayoría de los casos, un solo clone método será suficiente.
2. **Prototipo concreto** implementa el método de clonación. Además de copiar directamente los valores de campo a un clon nuevo, este método puede resolver algunas advertencias, que deberían permanecer ocultas para los Clientes. Por ejemplo, clonando objetos vinculados, desenredando dependencias recursivas, etc.
3. **El cliente** utiliza la interfaz de Prototipo para recuperar el clon de un objeto.

**Implementación de prototipos de registro**



1. **El registro de prototipos** proporciona un fácil acceso a los prototipos de uso frecuente, almacenando un conjunto de objetos creados previamente, listos para ser copiados. Por lo general, se puede implementar con un simple name → prototype mapa hash. Pero por conveniencia, cualquier otro criterio de búsqueda se puede agregar al registro.

## **Pseudocódigo**

En este ejemplo, el prototipo se utiliza para clonar objetos que representan formas geométricas sin acoplamiento a sus clases.

Todas las clases de formas implementan la interfaz de clonación común con un solo método de clonación. Las subclases llaman al método de clonación principal y luego copian sus propios campos en el objeto resultante.

Por lo tanto, el patrón de diseño de prototipo permite que el código del cliente clone objetos, incluso sin saberlo e independientemente de sus clases específicas.

// Base prototype.

**abstract class** **Shape** **is**

**field** X: int

**field** Y: int

**field** color: string

// A regular constructor.

**constructor** Shape(X, Y, color) **is**

**this**.X = X

**this**.Y = Y

**this**.color = color

// The prototype constructor. A fresh object is initialized

// with values from the old object in the constructor.

**constructor** Shape(source: Shape) **is**

**if** (source != **null**) **then**

**this**.X = source.X

**this**.Y = source.Y

**this**.color = source.color

// Clone operation always returns one of the Shape

// subclasses.

**abstract** **method** clone(): Shape

// Concrete prototype. Cloning method creates a new object and

// passes itself to the constructor. Until constructor is

// finished, has a reference to a fresh clone. Therefore, nobody

// has access to a partly built clone. This helps to make the

// cloning result consistent.

**class** **Rectangle** **extends** Shape **is**

**field** width: int

**field** height: int

**constructor** Rectangle(source: Rectangle) **is**

// Parent constructor call is mandatory in order to copy

// private fields defined in parent class.

**super**(source)

**if** (source != **null**) **then**

**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)

**if** (source != **null**) **then**

**this**.radius = source.radius

**method** clone(): Shape **is**

**return** **new** Circle(**this**)

// Somewhere in client code.

**class** **Application** **is**

**field** shapes: array of Shape

**constructor** Application() **is**

Circle circle = **new** Circle(10, 20, 15)

shapes.add(circle)

Circle anotherCircle = circle.clone()

shapes.add(anotherCircle)

// anotherCircle is the exact copy of circle.

Rectangle rectangle = **new** Rectangle()

rectangle.width = 10

rectangle.height = 20

shapes.add(rectangle)

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

// Prototype rocks because it allows producing a copy of

// an object without knowing anything about its type.

Array shapesCopy = **new** Array of Shapes.

// For instance, we do not know exact types of elements

// in shapes array. All we know is that all of them are

// Shapes. But thanks to the polymorphism, when we call

// the `clone` method on a shape, the program checks its

// real class and runs the appropriate clone method,

// defined in that class. That is why we get proper

// clones instead of a set of simple Shape objects.

**foreach** (s in shapes) do

shapesCopy.add(s.clone())

// The variable `shapesCopy` will contain exact copies

// of the `shape` array's children.

## **Applicability**

**Cuando su código no debe depender de las clases concretas de objetos que necesita copiar. Por ejemplo, cuando las clases de objetos son desconocidas, ya que trabaja con ellos a través de una interfaz.**

El patrón de prototipo proporciona a un cliente una interfaz para trabajar con todos los prototipos. Esta interfaz es común para todos los objetos que admiten la clonación. Hace que el código del cliente sea independiente de las clases concretas de productos que clona.

**Cuando desea reducir el tamaño de una jerarquía de clases que consiste en objetos similares, configurados de diferentes maneras (en otras palabras, cada clase tendría valores de campo únicos).**

 El patrón de prototipo permite crear un conjunto de objetos prototipo que representan todas las configuraciones posibles de un objeto.

Luego, en lugar de crear una instancia de una subclase que coincida con alguna configuración, el código del cliente busca el prototipo adecuado y lo clona.

## **Cómo implementar**

1. Crea la interfaz prototipo y declara el clone método en ella. Simplemente puede agregar el método a todas las clases de jerarquía de clases existentes, si tiene una.
2. Agregue un constructor alternativo a todas las clases de prototipos, que acepte un objeto de una clase actual. El constructor debe copiar los valores de todos los campos definidos en la clase del objeto pasado a la instancia actual. Luego debe llamar al constructor principal para que se ocupe de los campos de superclase.

Si su lenguaje de programación no admite la sobrecarga de métodos, puede definir un método especial para copiar los datos. El constructor es más conveniente ya que entrega la copia inmediatamente después de usar el new operador.

1. El clone método generalmente consiste en una sola línea: ejecutar un new operador usando el constructor de prototipo. Tenga en cuenta que cada clase que admita la clonación debe anular explícitamente el clone método para usar su nombre de clase junto con el new operador. De lo contrario, la clonación producirá un objeto de una clase padre.
2. Opcionalmente, cree un registro de prototipos centralizado para almacenar el catálogo de prototipos de uso frecuente. Incluso puede almacenar objetos de la misma clase, configurados de diferentes maneras.

Puede implementar el registro como una nueva clase de fábrica o un método de fábrica dentro de la clase de prototipo basen. El método de fábrica debe buscar el prototipo apropiado en función de la descripción que el código del cliente pasa como argumento. Podría ser solo una etiqueta de cadena o un conjunto complejo de criterios de búsqueda. Después de que se haya encontrado el prototipo apropiado, debe clonarlo y devolver la copia al cliente.

El último paso sería eliminar las llamadas directas a los constructores de objetos en favor de las llamadas al método de fábrica del registro de prototipos.

## **Pros y contras**

**Pros**

* Permite clonar objetos sin acoplar a sus clases concretas.
* Reduce la repetición del código de inicialización.
* Crea objetos complejos más rápido.
* Proporciona una alternativa para crear subclases cuando se trata de objetos complejos que tienen muchas opciones de configuración.

**Contras**

Los objetos complejos con muchas referencias a otros objetos pueden ser difíciles de clonar.

# Ejemplo en Java

## **shapes:** Shape list

#### **shapes/Shape.java:** Common shape interface

**package** refactoring\_guru.prototype.example.shapes;

**import** java.util.Objects;

**public** **abstract** **class** **Shape** {

**public** **int** x;

**public** **int** y;

**public** **String** color;

**public** Shape() {

}

**public** Shape(Shape target) {

**if** (target != **null**) {

**this**.x = target.x;

**this**.y = target.y;

**this**.color = target.color;

}

}

**public** **abstract** Shape clone();

**@Override**

**public** **boolean** equals(**Object** object2) {

**if** (!(object2 **instanceof** Shape)) **return** **false**;

Shape shape2 = (Shape) object2;

**return** shape2.x == x && shape2.y == y && Objects.equals(shape2.color, color);

}

}

#### **shapes/Circle.java:** Simple shape

**package** refactoring\_guru.prototype.example.shapes;

**public** **class** **Circle** **extends** Shape {

**public** **int** radius;

**public** Circle() {

}

**public** Circle(Circle target) {

**super**(target);

**if** (target != **null**) {

**this**.radius = target.radius;

}

}

**@Override**

**public** Shape clone() {

**return** **new** Circle(**this**);

}

**@Override**

**public** **boolean** equals(**Object** object2) {

**if** (!(object2 **instanceof** Circle) || !**super**.equals(object2)) **return** **false**;

Circle shape2 = (Circle) object2;

**return** shape2.radius == radius;

}

}

#### **shapes/Rectangle.java:** Another shape

**package** refactoring\_guru.prototype.example.shapes;

**public** **class** **Rectangle** **extends** Shape {

**public** **int** width;

**public** **int** height;

**public** Rectangle() {

}

**public** Rectangle(Rectangle target) {

**super**(target);

**if** (target != **null**) {

**this**.width = target.width;

**this**.height = target.height;

}

}

**@Override**

**public** Shape clone() {

**return** **new** Rectangle(**this**);

}

**@Override**

**public** **boolean** equals(**Object** object2) {

**if** (!(object2 **instanceof** Rectangle) || !**super**.equals(object2)) **return** **false**;

Rectangle shape2 = (Rectangle) object2;

**return** shape2.width == width && shape2.height == height;

}

}

#### **Demo.java:** Cloning example

**package** refactoring\_guru.prototype.example;

**import** refactoring\_guru.prototype.example.shapes.Circle;

**import** refactoring\_guru.prototype.example.shapes.Rectangle;

**import** refactoring\_guru.prototype.example.shapes.Shape;

**import** java.util.ArrayList;

**import** java.util.List;

**public** **class** **Demo** {

**public** **static** **void** main(**String**[] args) {

List<Shape> shapes = **new** ArrayList<>();

List<Shape> shapesCopy = **new** ArrayList<>();

Circle circle = **new** Circle();

circle.x = 10;

circle.y = 20;

circle.radius = 15;

shapes.add(circle);

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

shapes.add(anotherCircle);

Rectangle rectangle = **new** Rectangle();

rectangle.width = 10;

rectangle.height = 20;

shapes.add(rectangle);

cloneAndCompare(shapes, shapesCopy);

}

**private** **static** **void** cloneAndCompare(List<Shape> shapes, List<Shape> shapesCopy) {

**for** (Shape shape : shapes) {

shapesCopy.add(shape.clone());

}

**for** (**int** i = 0; i < shapes.size(); i++) {

**if** (shapes.get(i) != shapesCopy.get(i)) {

System.out.println(i + ": Shapes are different objects (yay!)");

**if** (shapes.get(i).equals(shapesCopy.get(i))) {

System.out.println(i + ": And they are identical (yay!)");

} **else** {

System.out.println(i + ": But they are not identical (booo!)");

}

} **else** {

System.out.println(i + ": Shape objects are the same (booo!)");

}

}

}

}

#### **OutputDemo.txt:** Execution results

0: Shapes are different objects (yay!)

0: And they are identical (yay!)

1: Shapes are different objects (yay!)

1: And they are identical (yay!)

2: Shapes are different objects (yay!)

2: And they are identical (yay!)

#### Prototype registry

You cold implement a centralized prototype registry (or factory), which would contain a set of pre-defined prototype objects. This way you could retrieve new objects from the factory by passing its name or other parameters. The factory would search for an appropriate prototype, clone it and return you a copy.

## **cache**

#### **cache/BundledShapeCache.java:** Prototype factory

**package** refactoring\_guru.prototype.caching.cache;

**import** refactoring\_guru.prototype.example.shapes.Circle;

**import** refactoring\_guru.prototype.example.shapes.Rectangle;

**import** refactoring\_guru.prototype.example.shapes.Shape;

**import** java.util.HashMap;

**import** java.util.Map;

**public** **class** **BundledShapeCache** {

**private** Map<**String**, Shape> cache = **new** HashMap<>();

**public** BundledShapeCache() {

Circle circle = **new** Circle();

circle.x = 5;

circle.y = 7;

circle.radius = 45;

circle.color = "Green";

Rectangle rectangle = **new** Rectangle();

rectangle.x = 6;

rectangle.y = 9;

rectangle.width = 8;

rectangle.height = 10;

rectangle.color = "Blue";

cache.put("Big green circle", circle);

cache.put("Medium blue rectangle", rectangle);

}

**public** Shape put(**String** key, Shape shape) {

cache.put(key, shape);

**return** shape;

}

**public** Shape get(**String** key) {

**return** cache.get(key).clone();

}

}

#### **Demo.java:** Cloning example

**package** refactoring\_guru.prototype.caching;

**import** refactoring\_guru.prototype.caching.cache.BundledShapeCache;

**import** refactoring\_guru.prototype.example.shapes.Shape;

**public** **class** **Demo** {

**public** **static** **void** main(**String**[] args) {

BundledShapeCache cache = **new** BundledShapeCache();

Shape shape1 = cache.get("Big green circle");

Shape shape2 = cache.get("Medium blue rectangle");

Shape shape3 = cache.get("Medium blue rectangle");

**if** (shape1 != shape2 && !shape1.equals(shape2)) {

System.out.println("Big green circle != Medium blue rectangle (yay!)");

} **else** {

System.out.println("Big green circle == Medium blue rectangle (booo!)");

}

**if** (shape2 != shape3) {

System.out.println("Medium blue rectangles are two different objects (yay!)");

**if** (shape2.equals(shape3)) {

System.out.println("And they are identical (yay!)");

} **else** {

System.out.println("But they are not identical (booo!)");

}

} **else** {

System.out.println("Rectangle objects are the same (booo!)");

}

}

}

#### **OutputDemo.txt:** Execution results

Big green circle != Medium blue rectangle (yay!)

Medium blue rectangles are two different objects (yay!)

And they are identical (yay!)

# Referencia

# https://refactoring.guru/design-patterns/prototype