Prototype Pattern

Introducción y nombre

Prototype. Creacional. Los objetos se crean a partir de un modelo.

Intención

Permite a un objeto crear objetos personalizados sin conocer su clase exacta o los detalles de cómo crearlos. El objetivo de este patrón es especificar prototipos de objetos a crear. Los nuevos objetos que se crearan se clonan de dichos prototipos. Vale decir, tiene como finalidad crear nuevos objetos duplicándolos, clonando una instancia creada previamente.

También conocido como: Patrón prototipo.

Motivación

Este patrón es necesario en ciertos escenarios es preciso abstraer la lógica que decide que tipos de objetos utilizará una aplicación, de la lógica que luego usarán esos objetos en su ejecución. Los motivos de esta separación pueden ser variados, por ejemplo, puede ser que la aplicación deba basarse en alguna configuración o parámetro en tiempo de ejecución para decidir el tipo de objetos que se debe crear.

Por otro lado, también aplica en un escenario donde sea necesario la creación de objetos parametrizados como "recién salidos de fábrica" ya listos para utilizarse, con la gran ventaja de la mejora de la performance: clonar objetos es más rápido que crearlos y luego setear cada valor en particular.

Solución

Situaciones en las que resulta aplicable.

Se debe aplicar este patrón cuando:

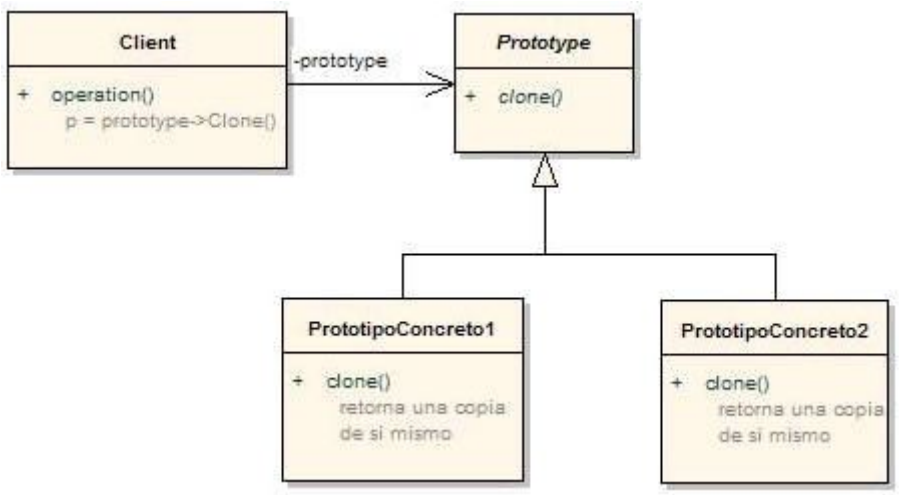
Las clases a instanciar sean especificadas en tiempo de ejecución.

Los objetos a crear tienen características comunes, en cuanto a los valores de sus atributos.

Se quiera evitar la construcción de una jerarquía Factory paralela a la jerarquía de clases de producto.

Diagrama UML

Estructura en un diagramas de clases.



Participantes

Responsabilidad de cada clase participante.

Cliente: solicita nuevos objetos.

Prototype: declara la interface del objeto que se clona. Suele ser una clase abstracta.

PrototypeConcreto: las clases en este papel implementan una operación por medio de la clonación de sí mismo.

Colaboraciones

Cliente: crea nuevos objetos pidiendo al prototipo que se clone.

Los objetos de Prototipo Concreto heredan de Prototype y de esta forma el patrón se asegura de que los objetos prototipo proporcionan un conjunto consistente de métodos para que los objetos clientes los utilicen.

Consecuencias

Un programa puede dinámicamente añadir y borrar objetos prototipo en tiempo de ejecución. Esta es una ventaja que no ofrece ninguno de los otros patrones de creación.

Esconde los nombres de los productos específicos al cliente.

Se pueden especificar nuevos objetos prototipo variando los existentes.

La clase Cliente es independiente de las clases exactas de los objetos prototipo que utiliza. y, además, no necesita conocer los detalles de cómo construir los objetos prototipo.

Clonar un objeto es más rápido que crearlo.

Se desacopla la creación de las clases y se evita repetir la instanciación de objetos con parámetros repetitivos.

Implementación

Dificultades, técnicas y trucos a tener en cuenta al aplicar el PD

Debido a que el patrón Prototye hace uso del método clone(), es necesaria una mínima explicación de su funcionamiento: 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. Una clase da permiso para que su instancia sea clonada si, y solo si, ella implementa el interface Cloneable.

Por otro lado, es importante destacar que si va a variar el número de prototipos se puede utilizar un "administrador de prototipos". Otra opción muy utilizada es un Map como se ve en el ejemplo.

Debe realizarse un gestor de prototipos, para que realice el manejo de los distintos modelos a clonar.

Por último, se debe inicializar los prototipos concretos.

Código de muestra

Imaginemos que estamos haciendo el software para una empresa que vende televisores plasma y LCD. La gran mayoría de los TVs plasma comparten ciertas características: marca, color, precio, etc. Lo mismo ocurre con los LCD. Esto es normal en ciertos rubros ya que compran por mayor y se obtienen datos muy repetitivos en ciertos productos. También es muy normal en las fábricas: salvo algún serial, los productos son todos iguales.
Volviendo a nuestro ejemplo, se decidió realizar una clase genérica TV con los atributos básicos que debe tener un televisor:

[code]
```
public abstract class TV implements Cloneable {
    private String marca;
    private int pulgadas;
    private String color;
    private double precio;

    public TV(String marca, int pulgadas, String color, double precio) {
        this.marca = marca;
        this.pulgadas = pulgadas;
        this.precio = precio;
        this.color = color;
    }

    public String getMarca() {
        return marca;
    }

    public void setMarca(String marca) {
        this.marca = marca;
    }

    public int getPulgadas() {
        return pulgadas;
    }

    public void setPulgadas(int pulgadas) {
        this.pulgadas = pulgadas;
    }

    public String getColor() {
        return color;
    }

    public void setColor(String color) {
        this.color = color;
    }

    public double getPrecio() {
        return precio;
    }

    public void setPrecio(double precio) {
        this.precio = precio;
    }

    @Override
    public Object clone() throws CloneNotSupportedException {
        return super.clone();
    }
}
```

También tenemos los televisores específicos:

```
public class Plasma extends TV {
   private double anguloVision;
   private double tiempoRespuesta;

   public Plasma(String marca, int pulgadas, String color, double precio, double anguloVision, double
tiempoRespuesta) {
      super(marca, pulgadas, color, precio);
      this.anguloVision = anguloVision;
      this.tiempoRespuesta = tiempoRespuesta;
   }

   public double getAnguloVision() {
      return anguloVision;
   }

   public void setAnguloVision(double anguloVision) {
      this.anguloVision = anguloVision;
   }

   public double getTiempoRespuesta() {
      return tiempoRespuesta;
   }

   public void setTiempoRespuesta(double tiempoRespuesta) {
      this.tiempoRespuesta = tiempoRespuesta;
   }
}

public class LCD extends TV {
   private double costoFabricacion;

   public LCD(String marca, int pulgadas, String color, double precio, double costoFabricacion) {

      super(marca, pulgadas, color, precio);
      this.costoFabricacion = costoFabricacion;
   }

   public double getCostoFabricacion() {
      return costoFabricacion;
   }

   public void setCostoFabricacion(double costoFabricacion) {
      this.costoFabricacion = costoFabricacion;
   }
}
```

Debido a que se decidió realizar ciertos prototipos, estos se ven plasmados en la clase TvPrototype:

```
public class TvPrototype {
   private Map<String, TV> prototipos = new HashMap<String, TV>();

   public TvPrototype() {
      Plasma plasma = new Plasma("Sony", 21, "Plateado", 399.99, 90, 0.05);
      LCD lcd = new LCD("Panasonic", 42, "Plateado", 599.99, 290);

      prototipos.put("Plasma", plasma);
      prototipos.put("LCD", lcd);
   }

   public Object prototipo(String tipo) throws CloneNotSupportedException {
      return prototipos.get(tipo).clone();
   }
}
```

La invocación al método prototipo() sería:

```
public class Main {
   public static void main(String[] args) throws Exception {
      TvPrototype tvp = new TvPrototype();
      TV tv = (TV) tvp.prototipo("Plasma");

      System.out.println(tv.getPrecio());
   }
}
```

El resultado de la consola es: 399.99
[/code]

Cuándo utilizarlo

Este patrón debe ser utilizado cuando un sistema posea objetos con datos repetitivos: por ejemplo, si una biblioteca posee una gran cantidad de libros de una misma editorial, mismo idioma, etc.

Por otro lado, el hecho de poder agregar o eliminar prototipos en tiempo de ejecución es una gran ventaja que lo hace muy flexible.

Patrones relacionados

Abstract Factory: el patrón Abstract Factory puede ser una buena alternativa al Prototype donde los cambios dinámicos que Prototype permite para los objetos prototipo no son necesarios. Pueden competir en su objetivo, pero también pueden colaborar entre sí.

Facade: la clase cliente normalmente actúa comúnmente como un facade que separa las otras clases que participan en el patrón Prototype del resto del programa.

Factory Method: puede ser una alternativa al Prototype cuando los objetos prototipo nunca contiene más de un objeto.

Singleton: una clase Prototype suele ser Singleton.