Daza Corredor Alejandro
Paolo

Patrones Creacionales

Prototype

"Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice" [Christopher Alexander].

1. Introducción

El Patrón de diseño Prototype especifica los tipos de objetos a crear por medio de una instancia prototípica, y crea nuevos objetos copiando dicho prototipo. Este patrón se usa cuando la creación de una instancia de clase consume mucho tiempo o es compleja de alguna manera. Entonces, en vez de crear más instancias se hacen copias de la instancia original modificándolas de manera apropiada. Veamos la formalidad de la plantilla GoF de este patrón.

2. Prototype

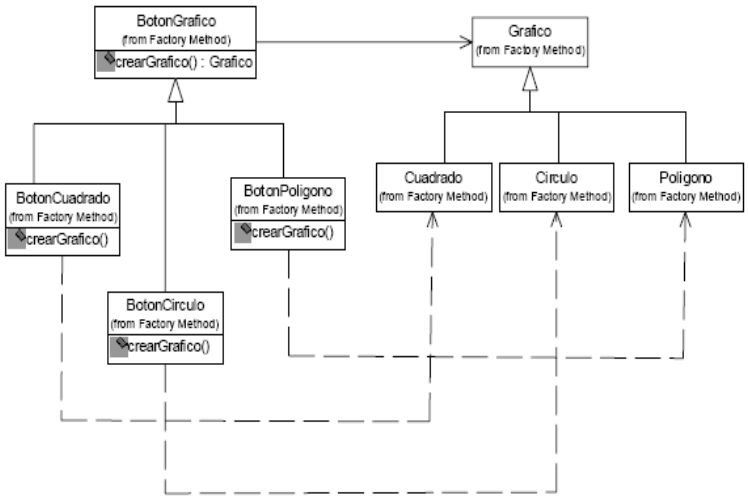
Nombre del patrón:
Prototype

Clasificación del patrón:
Creacional.

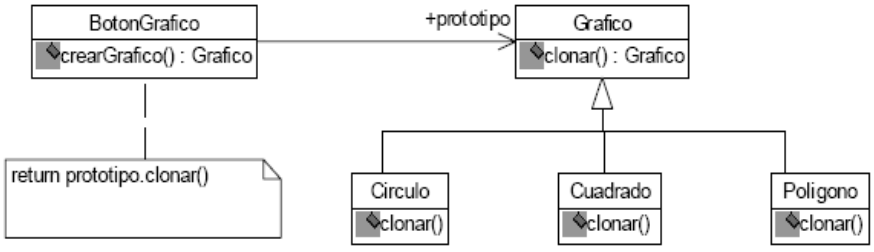
Intención:
Especifica el tipo de objeto a crear usando una instancia prototípica y crea nuevos objetos copiando este prototipo.

También conocido como:
Prototype.

Motivación:
Imaginemos en el siguiente escenario, se desea adaptar un framework de editores gráficos con parejas botón – objeto, donde cada botón debe crear un tipo especifico de objeto, hasta este punto lo mas lógico es pensar en usar el patrón de diseño Factory Method, en donde tendríamos que crear una jerarquía de clases paralelas en las que solo se diferenciaría su factoryMethod().



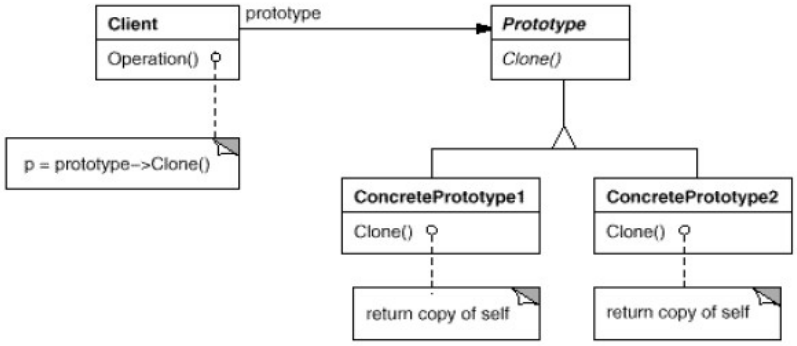
Usando prototipos no seria necesaria esa jerarquía paralela de clases.



Aplicabilidad:
El patrón Prototype debe ser usado cuando:

- Se puede usar este patrón 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 (por ejemplo, mediante carga dinámica).
- Cuando se desea evitar construir una jerarquía de clases de fábricas 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.

Estructura:



Participantes:

- **Prototype**: declara una interfaz para clonarse él mismo.
- **ConcretePrototype**: implementa la operación para clonarse él mismo.
- **Client**: crea un nuevo objeto pidiéndole a un Prototipo que se clone.

Colaboraciones:

- Un cliente pide al prototipo clonarse a si mismo.

Consecuencias:

- Tiene la mismas consecuencias que Abstract Factory y Builder, ocultando los productos concretos al cliente y reduciendo de esta forma el número de nombres que el cliente debe conocer.
- Flexibilidad a la hora de añadir objetos en tiempo de ejecución.
- Especifica nuevos objetos variando su valor.
- Especifica nuevos objetos variando su estructura.

- Reduce las jerarquías de clases, no hace falta una jerarquía de creadores.

Implementación:

- Usar un gestor de prototipos si el número de prototipos no es fijo, para mantener un registro de los prototipos disponibles. El cliente pide al gestor de prototipos un prototipo antes de clonarle.
- La operación Clonar es compleja, hay que decidir entre:
  - shallow copy: Solo se copia el objeto inicial no los objetos que contiene.
  - deep copy: Se copian tanto el objeto inicial como los objetos que contenga.
- Puede ser necesario inicializar los clones: es mejor hacerlo con una operación específica como Inicializar que incluir parámetros en la signatura de Clonar.

Código de ejemplo:

```
classDepthReadingimplementsCloneable{
   private double depth;
   publicDepthReading(double depth) {
          this.depth = depth;
   }
   public Object clone(){
          Object o = null;
          try {
               o = super.clone();
          } catch(CloneNotSupportedExceptione) {
               e.printStackTrace(System.err);
          }
          return o;
   }
   public StringtoString(){
          return "Depth: " + depth + "\n";
   }
}

classTemperatureReadingimplementsCloneable{
   private long time;
   private double temperature;
   publicTemperatureReading(double temperature) {
          time = System.currentTimeMillis();
          this.temperature = temperature;
   }
   public Objectclone(){
          Object o = null;
          try {
               o = super.clone();
          } catch(CloneNotSupportedExceptione) {
               e.printStackTrace(System.err);
          }
          return o;
   }
   public StringtoString(){
          return "Time: " + time + " Temperature: " + temperature + "\n";
   }
}


classOceanReadingimplementsCloneable{
   privateDepthReadingdepth;
   privateTemperatureReadingtemperature;
   publicOceanReading(doubletdata, doubleddata){
          temperature = newTemperatureReading(tdata);
          depth = newDepthReading(ddata);
   }
   public Object clone(){
          OceanReadingo = null;
          try {
               o = (OceanReading)super.clone();
          } catch(CloneNotSupportedExceptione) {
               e.printStackTrace(System.err);
          }
          // Sedeben clonar las referencias
          o.depth = (DepthReading)o.depth.clone();
          o.temperature = (TemperatureReading)o.temperature.clone();
          return o;
   }
   public StringtoString(){
          return depth.toString() + temperature.toString();
   }
}
```

Usos conocidos:

- Java API:
  - Este patrón es la esencia de JavaBeans.
  - Los JavaBeans son instancias de clases que satisfacen una convención de nombres.
  - El programa encargado de la creación de beans utiliza esta convención para personalizarlos.
  - Después de personalizar el objeto para su uso en una aplicación,el objeto es salvado a un fichero para poder ser cargado por una aplicación en ejecución.

Patrones relacionados:

- Abstract Factory, Composite, Decorator.

Referencias bibliográficas

[1] Design Patterns. Elements of Reusable Object-Oriented Software - Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides - Addison Wesley (GoF- Gang of Four)

[2] Desing Patterns Java. James W Cooper. ebook – Addison Wesley.

[3] Patrones de Diseño, Diseño de Software Orientado a Objetos - Joaquin Garcia. http://www.ingenierosoftware.com/analisisydiseno/patrones-diseno.php.

[4] Patrones Creacionales, Área de Lenguajes y Sistemas Informáticos. Universidad de Burgos

Daza Corredor, Alejandro Paolo
Ingeniero de Sistemas, egresado de la Universidad Distrital Francisco José de Caldas, Docente TCO del proyecto curricular de Ingeniería de Sistemas en el área de programación y estudiante de la especialización de Ingeniería de Software de la Universidad Distrital Francisco José de Caldas.

e-mail: apdaza@gmail.com