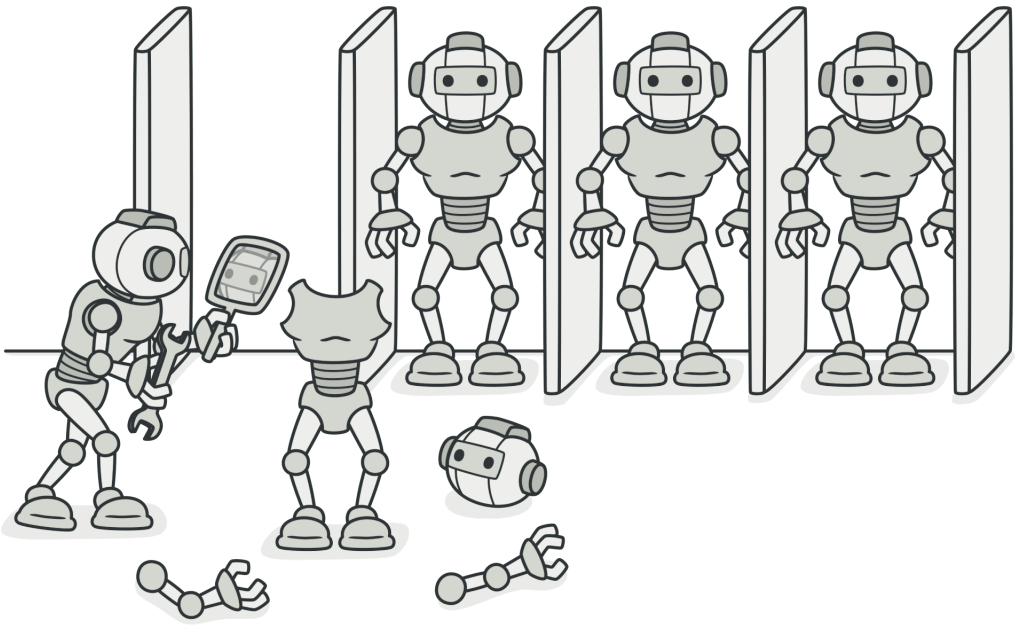
Proyecto 2.
Patrón de Diseño Creacional: Prototype

.



David, Calderón Jiménez

Humberto Ignacio, Hernández Olvera

Iñaky, Ordiales Caballero

06/Diciembre/2020

Introducción

Diseñar software orientado a objetos es difícil, y diseñar software orientado a objetos *reusable* es incluso más difícil. Se deben encontrar objetos pertinentes, factorizarlos en clases, definir interfaces y jerarquías de herencias, para establecer una relación clave entre ellos. El diseño debe ser específico para el problema actual, pero lo suficientemente general para utilizar en problemas futuros. También se debe evitar rediseñarlo, o al menos, minimizarlo. Diseñadores experimentados utilizan soluciones que les han servido en el pasado, lo que los lleva a encontrar patrones recurrentes de clases, que sirven para resolver problemas específicos de diseño y hacen que los diseños orientados a objetos sean mas flexibles, elegantes y principalmente reusables. Al final estos patrones se convierten en lo que se conoce como patrones de diseño.

Un patrón de diseño describe un problema el cual ocurre una y otra vez en nuestro ambiente, y después describe el aspectos central de la solución para ese problema, de una manera tal que se puede usar esta solución un millón de veces, sin hacerlo de la misma manera dos veces.

Los patrones de diseño hacen más fácil la reutilización exitosa de diseños y arquitecturas. Ayudan a los diseñadores a elegir alternativas de diseño que hacen que un sistema sea reutilizable y evitar alternativas que comprometan su reusabilidad, pueden incluso mejorar la documentación y el mantenimiento de sistemas existentes proporcionando una explicación específica de las interacciones entre clases y objetos y su intención subyacente.

Patrón de diseño: Prototipo

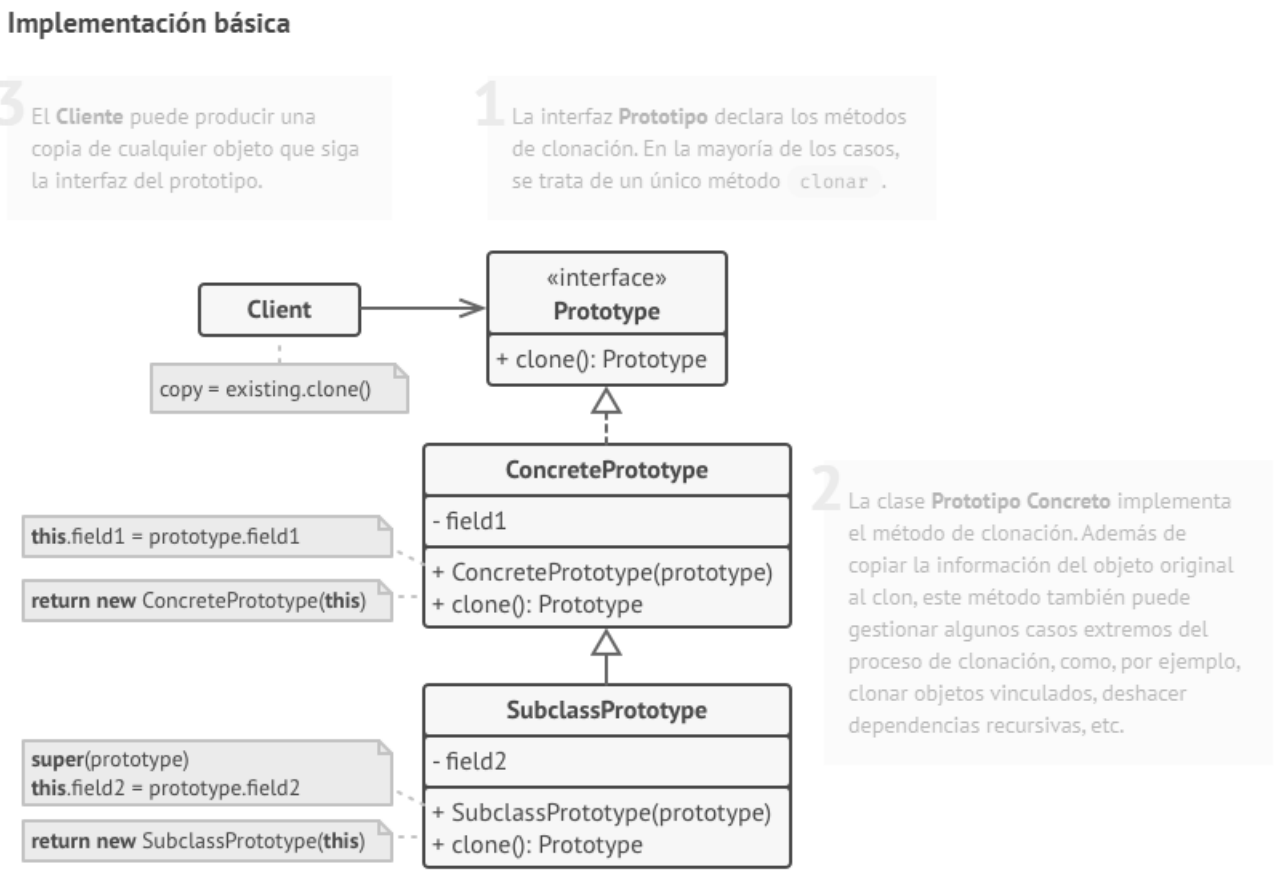
El patrón de diseño **prototipo** es un patrón de diseño creacional, es decir tiene como finalidad crear nuevos objetos. La manera en que realiza esto es mediante un proceso llamado clonación en el cual se estará clonando una instancia concreta creada anteriormente en el programa. En su funcionamiento se específica el objeto de alguna clase que será clonado, a esta instancia se le conoce como el prototipo, ya que a partir de él se realizará la copia (el clon) en otro objeto. De esta denominación prototipo del objeto a clonar es que sale el nombre del patrón. Sin embargo, otros usos como el de crear objetos para realizar pruebas con ellos también nos lleva al nombre de prototipo más cercano al que conocemos fuera de la programación.

Como se sabe cualquier patrón cuenta con 4 elementos: nombre del patrón, problema, solución, consecuencias. Y además se le podría agregar otro elemento que sea su importancia. A continuación se mencionan los cuatro elementos para el patrón de diseño Prototype.

- **Nombre:** Prototype (prototipo)
- **Problema:** El patrón de diseño prototype resulta útil para aquellos casos donde se necesite crear un objeto de una clase previamente ya instanciada, pero la creación desde su clase resulta muy costosa, o inclusive no se conoce la clase del objeto que se quiere replicar. Prototype te permite clonar un objeto dentro de una jerarquía de clases sin saber en específico qué clase es. También es usado cuando el objeto que se desea duplicar tiene modificadores de acceso restrictivos, debido a lo cual no se puede acceder a los valores de sus atributos, sin embargo, clonarlo mediante el patrón no es problema, ya que la clonación se realiza desde dentro. A demás en caso de que el objeto se encuentre en memoria secundaria o en algún lugar de almacenamiento al cual sea costoso acceder, Prototype realiza la duplicación en un sólo acceso. A diferencia de cuando se iguala atributo por atributo. El objeto que se desee ser clonado debe tener la capacidad inherente de clonación. Por ejemplo en lenguaje Java, ningún objeto de la clase Object puede ser clonado.
- **Solución:** Para implementar el patrón prototype se debe de crear una interfaz de clonación con la declaración de un método clonar, la cuál sea implementada en todas las clases que se desee posean la capacidad de ser clonadas. Este método deberá desarrollar la capacidad de devolver una copia del objeto.
- **Consecuencias:**
  - Aplicar el patrón prototipo permite ocultar las clases producto (prototipos concretos) del cliente y permite que el cliente trabaje con estas clases dependientes de la aplicación sin cambios.
  - Además, hace posible añadir y eliminar productos en tiempo de ejecución al invocar a la operación clonar, lo que supone un método que proporciona una configuración dinámica de la aplicación.
  - Este patrón permite la especificación de nuevos objetos generando un objeto con valores por defecto sobre el que posteriormente se podrán aplicar cambios. La especificación de nuevos objetos también puede realizarse mediante la variación de su estructura. Reducción del número de subclases.

Los cuatro elementos del patrón serán mencionados y explicados más a fondo en el resto del documento, así como algunos ejemplos para ilustrar mejor su funcionamiento.

Este patrón resulta útil en escenarios donde es impreciso abstraer la lógica que decide qué 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. En ese caso, la aplicación necesitará crear nuevos objetos a partir de modelos. Estos modelos, o prototipos, son clonados y el nuevo objeto será una copia exacta de los mismos, con el mismo estado. Esto resulta interesante para crear, en tiempo de ejecución, copias de objetos concretos inicialmente fijados, o también cuando sólo existe un número pequeño de combinaciones diferentes de estado para las instancias de una clase.



Dicho de otro modo, este patrón propone la creación de distintas variantes de objetos que la aplicación necesite, en el momento y contexto adecuado. Toda la lógica necesaria para la decisión sobre el tipo de objetos que usará la aplicación en su ejecución se hace independiente, de manera que el código que utiliza estos objetos solicitará una copia del objeto que necesite. En este contexto, una copia significa otra instancia del objeto. El único requisito que debe cumplir este objeto es suministrar la funcionalidad de clonarse.

En el caso de un editor gráfico, se pueden crear rectángulos, círculos u otros, como copias de prototipos. Estos objetos gráficos pertenecerán a una jerarquía cuyas clases derivadas implementarán el mecanismo de clonación.

Participantes

- Cliente: es el encargado de solicitar la creación de los nuevos objetos a partir de los prototipos.
- Prototipo Concreto: posee características concretas que serán reproducidas para nuevos objetos e implementa una operación para clonarse.
- Prototipo: declara una interfaz para clonarse, a la que accede el cliente.

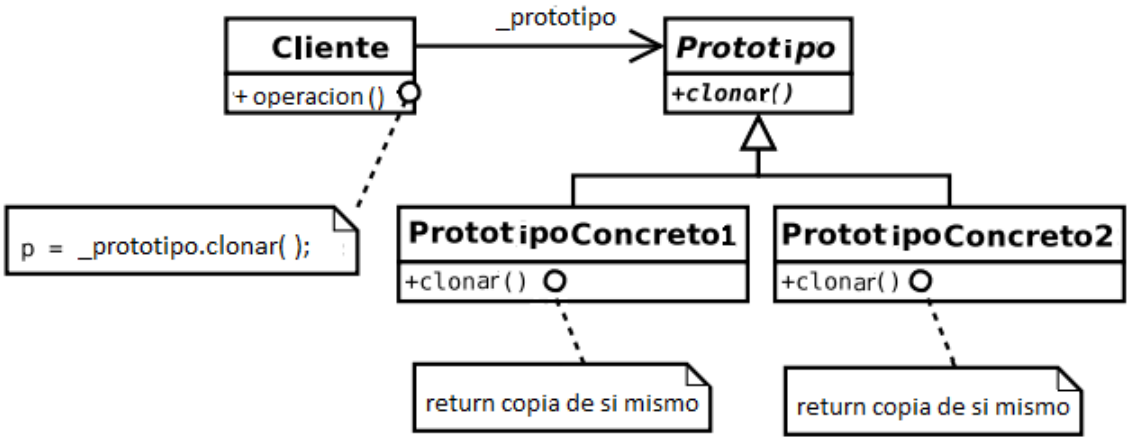


Figura 1: Diagrama de clases para el patrón de diseño Prototipo

Clonación profunda frente a clonación superficial

Como se ha mencionado anteriormente, el patrón de diseño prototipo tiene fundamentalmente dos variaciones. Éstas son la clonación superficial y la clonación profunda, las cuales nos permiten decidir qué tipos de atributos queremos clonar según las necesidades de nuestra solución.

Clonación Superficial (Shallow copy)

La clonación superficial de un objeto implica que todos los atributos de datos primitivos que no son del tipo referencial, es decir almacenan directamente el valor y no una referencia, junto con las Strings que según el lenguaje de programación pueden o no ser una referencia, serán copiados y establecidos como los valores del nuevo objeto. Pero a pesar de que se modifiquen en uno, el otro seguirá sin ser modificado. Por el otro lado para los atributos de tipo de dato abstracto que guardan una referencia hacia los objetos, se copiará la referencia tal cual y se establecerá al atributo del nuevo objeto. Al hacer esto, se está usando la misma referencia y por ende el mismo dato abstracto en ambos objetos, y si uno lo modifica, el cambio se reflejará en el segundo objeto ya que finalmente los dos tienen el mismo valor para su atributo. El caso de los Strings es interesante, ya que a pesar de ser una referencia, éste tipo de dato es inmutable, entonces al modificarlo en un objeto la referencia cambia, no se estará modificando en si el atributo del prototipo.

El lenguaje orientado a objetos Java, ya nos provee con una interfaz para la clonación superficial de objetos. Esta interfaz se llama ***cloneable***, además se cuenta con una excepción de la clase Object, ***Object Clone() throws CloneNotSupportedException*** para llevar a cabo la implementación del prototipo de manera compatible con los prototipos ya existentes en las librerías Java.

Clonación Profunda (Deep copy)

La clonación Profunda se puede considerar como la clonación más completa e independiente que se puede llegar a hacer. Sin embargo, también es la que más problemas puede generar. La clonación profunda funciona de manera similar a la clonación superficial, de hecho para poder realizar una clonación profunda normalmente primero se realiza la clonación superficial y ya después se adecua a profunda. A diferencia de la superficial, al realizar una clonación profunda los dos objetos no compartirán referencias para sus valores de los tipos de datos abstractos. Se deberán crear clones de los tipos de datos abstractos y poner esas nuevas referencias al objeto clonados. Es hasta cierto punto recursiva la clonación profunda, porque si tiene una referencia y esta referencia tiene otra referencia como atributo, se tendrá que clonar cada uno desde lo más profundo hacia lo más superficial. Y para lograr esto todas las clases relacionadas deberán contar con la opción de ser clonadas. Por esto dijimos que es la opción que más problemas genera.

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.

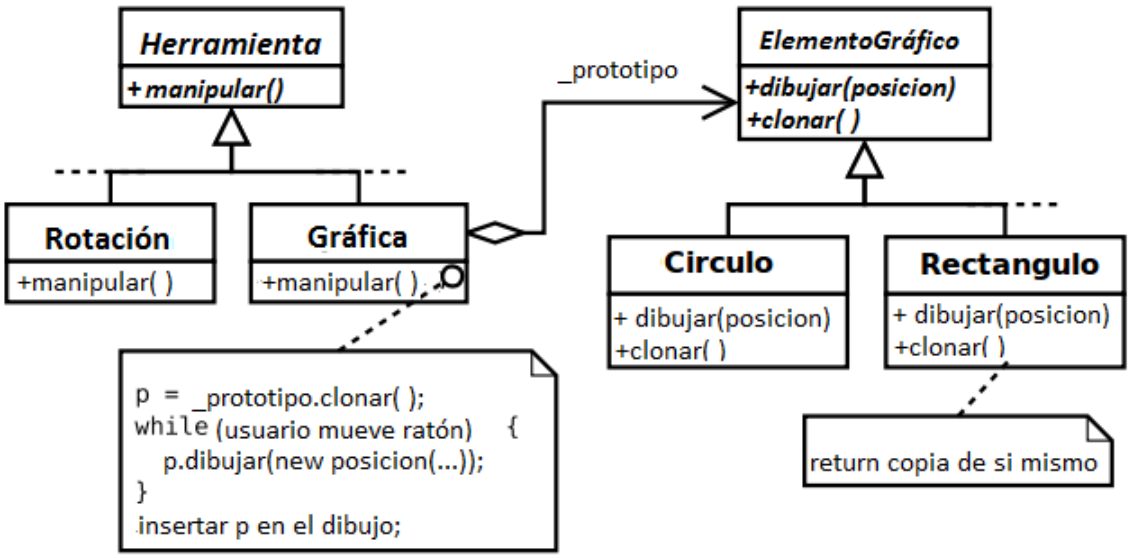
   Si el lenguaje de programación que utilizas no soporta la sobrecarga de métodos, puedes definir un método especial para copiar la información del objeto. El constructor es el lugar más adecuado para hacerlo, porque entrega el objeto resultante justo después de invocar el operador new.

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 sobreescribir 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.

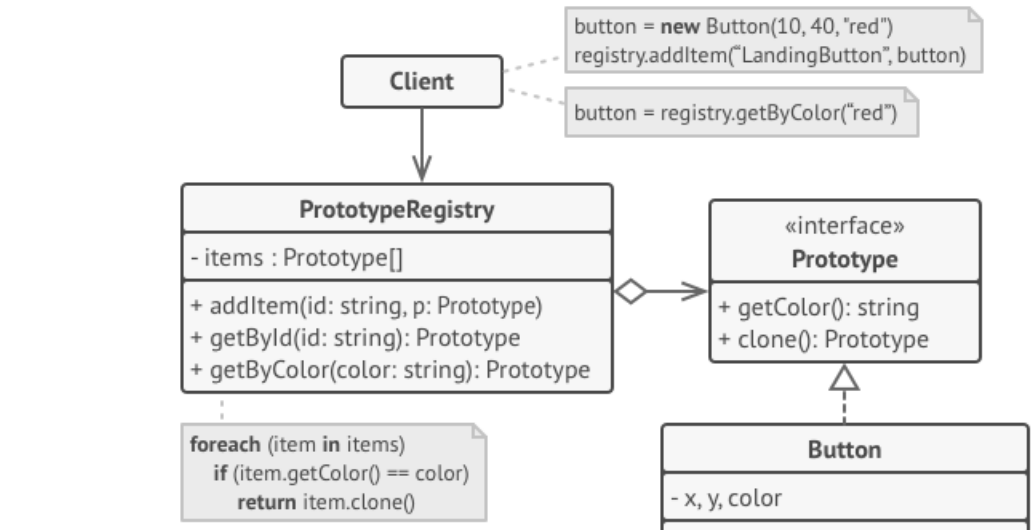
4. Opcionalmente, puedes crear un registro de prototipos centralizado para almacenar un catálogo de prototipos de uso frecuente.

   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.



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.

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.

Usos conocidos

Quizás el primer ejemplo del patrón Prototype fue en el sistema Sketchpad de Ivan Sutherland. La primera aplicación ampliamente conocida del patrón en un lenguaje orientado a objetos fue en ThingLab, en donde los usuarios podían formar un objeto compuesto y luego promoverlo a un prototipo instalándolo en una librería de objetos reusables. Goldberg y Robson mencionan prototipos como un patrón, pero Coplien da una descripción mucho más completa. El describe lenguajes relacionados al patrón Prototype para C++ y da muchos ejemplos y variaciones.

Ventajas y Desventajas

Se pueden clonar objetos sin acoplarlos a sus clases concretas.	Clonar objetos complejos con referencias circulares puede resultar complicado.
Puedes evitar un código de inicialización repetido clonando prototipos prefabricados.	La jerarquía de prototipos debe ofrecer la posibilidad de clonar un elemento y esta operación puede no ser sencilla de implementar.
Puedes crear objetos complejos con más facilidad.	Si se maneja una colección ordenada mediante claves únicas, en una clonación la clave única del prototipo tendría el mismo valor que la clave del clon. Por lo que puede causar problemas al recuperar información.
Obtienes una alternativa a la herencia al tratar con preajustes de configuración para objetos complejos.	Si la clonación se produce frecuentemente, el coste puede ser importante.

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.

Ejemplo de uso

En el caso de una tarjeta bancaria, para poder realizar los cortes de mes, se puede utilizar este patrón, de tal forma, que la clonación se hace al momento de hacer el corte, clonando todos los datos de las compras hechas en ese mes, y guardando esta información en algún lugar, como podría ser, un arreglo que almacene los estados de cuenta previos.

ICuenta.java

```java
public interface ICuenta extends Cloneable {
  ICuenta clonar();
}
```

Figura 2: Interfaz que hereda de *Cloneable* para poder realizar la clonación.

Se crea la interfaz que nos permitirá utilizar el método de clonación de objetos, ésta será nuestra clase **prototipo**, con un método *clonar()*, que es el que permitirá utilizar el método proveído en la interfaz *Cloneable*.

```java
@Override
public ICuenta clonar() {
  CuentaAH cuenta = null;

  try {
    cuenta = (CuentaAH) clone();
    LinkedList<String> listaNueva1 = new LinkedList();
    LinkedList<Float> listaNueva2 = new LinkedList();
    cuenta.compras = listaNueva1;
    cuenta.precios = listaNueva2;
    cuenta.mes+=1;
    int auxiliar = 0;
    cuenta.mesPalabra = meses[cuenta.mes];
    for(Float elemento : precios)
      auxiliar += elemento;
    cuenta.saldo -= auxiliar;
  } catch(CloneNotSupportedException e) {
    e.printStackTrace();
  }

  return cuenta;
}
```

Figura 3: Sobre escritura del método *clonar*.

En la clase **prototipo concreto**, se debe sobre escribir el método *clonar()*, que heredamos desde la clase prototipo, con el propósito de utilizar el método *clone()*, éste método se encuentra en la interfaz Cloneable y es la que nos permite realizar una clonación superficial. Para clonar la demás información, se tiene que hacer manualmente, para nuestro ejemplo, asignar una nueva lista a las compras y precios con el propósito de simular que para el siguiente mes no

se tienen gastos pendiente, incrementar en uno el mes correspondiente al siguiente estado de cuenta, y descontar del saldo todos los gastos del mes en que se hizo el corte.

```
32      do{
33        opcion = menu.mostrarMenu();
34        switch(opcion){
35          case 1:
36            System.out.println("\n\n Registrar compra.");
37            cuentaAhorro.registrarCompra();
38            break;
39          case 2:
40            System.out.println("\n\n Corte de mes.");
41            estados.add(cuentaAhorro);
42            cuentaAhorro = (CuentaAH) cuentaAhorro.clonar();
43            String carga = "";
44            for (int i = 0; i < 11; i++) {
45                int porcent = i*10;
46                System.out.println("Corte Progreso "+carga+" "+porcent+" %");
47                carga = carga + "--";
48                Thread.sleep(250);
49            }
50            System.out.println("\n\n Se realizo el corte exitosamente...\n\n");
51            break;
52          case 3:
53            System.out.println("\n\n Mostrar estados de cuenta anteriores.");
54            if(estados.isEmpty())
55              System.out.println("\n No hay estados, realice el corte de mes.");
56            else{
57              for(CuentaAH elemento : estados){
58                elemento.mostrarCorte();
59                System.out.println();
60              }
61            }
62            break;
63        }
64        if(opcion!=4)
65          pressEnter();
66      }while(opcion!=4);
67
68    }
```

Figura 4: El cliente hace la clonación del objeto en la línea 42.

El otro elemento que falta por intervenir en nuestro ejemplo, es el **cliente**, el cual lo ubicamos en el método principal, cuando se hace el corte del mes, es entonces cuando queremos guardar el estado de cuenta actual al arreglo que almacena los estados de cuenta, posteriormente realizamos la clonación del objeto (nuestra cuenta de ahorro) en el mismo objeto.

La excepción *CloneNotSupportedException* (Excepción que señala que no todos los objetos se pueden clonar, como ejemplo esta el objeto *Object*) se especifica en el cliente que se va a lanzar, y la manejamos en el prototipo concreto.

Conclusiones

David Calderón Jiménez

Desde mi punto de vista los patrones de diseño me resulta un tema muy profesional pues nos muestra soluciones a muchos problemas que se nos podrían presentar de manera frecuente al momento ce diseñar un programa, y pues estos patrones ya han sido muy estudiados por lo cual son eficientes y requieren poca memoria.

Ahora me doy cuenta de que programar ya no solo significa saber usar arreglos ciclos o listas sino que ya empezamos a ver temas de diseño que para mi son mas difíciles que los temas de programación, pues el diseño tienen un grado de dificultad alto debido a que no es fácil planificar un problema desde 0 y saber si lo estas haciendo de la mejor manera.A partir de todo esto puedo decir que con forme mas avanzan los temas mas amor le tomo a la carrera pues empiezo a notar que aunque todos puedan aprender a programar nosotros llevamos esto a un nivel mas alto y mas profesional.Al final del día aunque este proyecto no fue un programa como tal para mi me enseño hasta mas que el primer proyecto pues me llevo a conocer cosas nuevas que nunca había trabajado en ningún lenguaje de programación.

Humberto Ignacio Hernández Olvera

El tema de patrones de diseño nunca lo había escuchado, y dentro de los otros cursos que había tomado de programación no los mencionaron. Me parece excelente que en el temario se agregue el ver este tema. Ya habiendo investigado algo acerca de ellos, y para nuestro caso, conocer el patrón *prototype*, creo que le podemos sacar provecho y utilizarlo más a menudo en el futuro, especialmente para proyectos grandes con múltiples colaboradores, en donde se necesiten hacer pruebas con los objetos creados por otras personas y no se deba cambiar lo que se tiene originalmente, por ejemplo, para no moverle a algo que está en producción, pero que tiene algún bug y se deba arreglar. Pues no sería adecuado trabajar con el producto que está en vivo, aquí es mejor clonar todo el producto, hacer las pruebas y modificaciones en un *sandbox*, para que no se caiga el servicio, y ya que se tenga arreglado el problema, ahora sí mandarlo a producción.

En resumen, me gustó que hayamos conocido que este tipo de cosas existen, ya que uno por cuenta propia en otro lugar más informal, sería muy difícil poderlo encontrar, o siquiera, llegar a escuchar de ellos.

Iñaky Ordiales Caballero

A través de este proyecto nos pudimos familiarizar profundamente con un patrón de diseño en específico. En nuestro caso fue con el patrón de diseño para la creación de objetos llamado Prototype. Pero no sólo aprendimos un patrón de diseño, sino que tuvimos la oportunidad de probar y realmente entender la posible utilidad de los patrones de diseño a través de él. Si bien antes de ver este patrón, nunca había pensado en la necesidad de clonar objetos, una vez lo vi hace mucho sentido tanto en la utilidad como en la necesidad de implementar el patrón. Y pienso que será igual para muchos otros patrones de diseño, donde bien si en un principio no sabemos que los necesitamos, al ir aprendiendo sobre ellos iremos entendiendo sus beneficios y los usaremos cada vez más seguidos en futuros proyectos.

La elaboración del proyecto contando con una parte escrita y una parte audiovisual también fue un reto nuevo de afrontar ya que en semestres anteriores lo equivalente era hacer presentaciones, pero al tener que hacer un video la dinámica cambia bastante. Tuvimos que aprender a planear las diferentes tomas y el usar tanto un software para grabar, como uno para editar el video. Creo que al final el resultado de ambas partes salió bastante bien, quedando un video que te deja la idea clara acerca del patrón y del cual se puede extender la compresión a través del trabajo de investigación escrito. Finalmente tomando en cuenta que el objetivo del proyecto era trabajar con un patrón de diseño para entender la importancia de estos, creo que se puede concluir que el objetivo del proyecto fue cumplido con éxito.

Fuentes de información

[1] Erich Gamma y col. *Design Patterns*. Addison-Wesley, 1995. ISBN: 0-201-63361-2.

[2] Refactoring Guru. *Prototype*. URL: https://refactoring.guru/es/design-patterns/prototype (visitado 27-11-2020).

[3] Naresh Joshi. *Shallow and Deep Java Cloning*. URL: https://dzone.com/articles/shallow-and-deep-java-cloning (visitado 02-12-2020).

[4] Joydip Kanjilal. «Implementing the prototype design pattern.» En: *InfoWorld.com* (2017). URL: http://pbidi.unam.mx:8080/login?url=http://search.ebscohost.com/login.aspx?direct=true&db=edsgit&AN=edsgit.A477816695&lang=es&site=eds-live.

[5] A.A. Nykonenko. «Creational design patterns in computational linguistics: Factory Method, Prototype, Singleton». en. En: *Cybern Syst Anal* 48 (2012), págs. 138-145. URL: https://doi-org.pbidi.unam.mx:2443/10.1007/s10559-012-9383-1.