POLITÉCNICA

"Ingeniamos el futuro"

CAMPUS
DE EXCELENCIA
INTERNACIONAL

MiW

Patrones de Diseño

21. *Prototype*

Luis Fernández Muñoz

https://www.linkedin.com/in/luisfernandezmunyoz

setillofm@gmail.com

INDICE

1. Problema

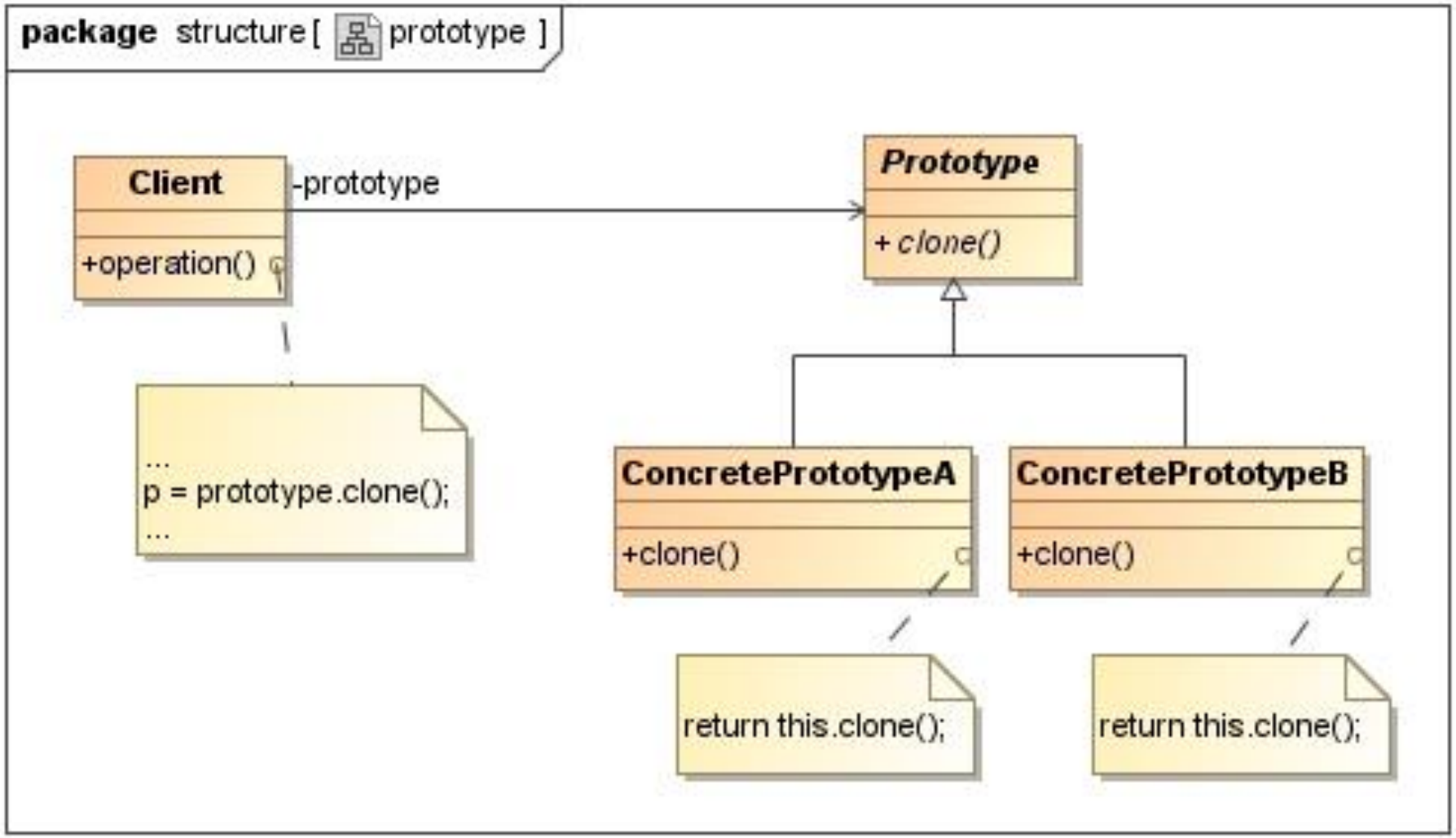
- *Un poeta en un parque vende poemas famosos que escribe en el momento en que se lo piden porque se los sabe de memoria. Si cambian los gustos, tendrá que adaptarse y aprender nuevos poemas. Lo cual es complejo y tiende a crecer*
- *Otro poeta dispone de copias de poemas famosos y realiza fotocopias en el momento en que se lo piden sin sabérselos de memoria. Si cambian los gustos, tendrá que conseguir una copia del nuevo poema, incluso en otro idioma. Lo cual es muy sencillo y no tiende a crecer.*

- ***Especifica las clases de objetos a crear usando una instancia prototípica y crea nuevos objetos por copias de este prototipo***

1. Problema

- Úsese
  - Cuando un sistema deba ser independiente de cómo se crean, se componen y se representan sus productos; y
  - Cuando las clases a instanciar sean especificadas en tiempo de ejecución (por ejemplo, mediante carga dinámica); o
  - Para evitar construir una jerarquía de clases fábrica paralela a la jerarquía de clases de los productos; o
  - 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

2. Solución


package structure [prototype]
Client
-prototype
+operation()
Prototype
+ clone()
...
p = prototype.clone();
...
ConcretePrototypeA
+clone()
ConcretePrototypeB
+clone()
return this.clone();
return this.clone();

2. Solución

- Cuando el número de prototipos de un sistema no es fijo, mantiene un registro de los prototipos disponibles. Los clientes no gestionarían ellos mismos los prototipos, sino que los guardarán y recuperarán del registro. Un cliente le pedirá al registro un prototipo antes de clonarlo.
  - Este gestor de prototipos es un almacén asociativo que devuelve el prototipo que concuerda con una determinada clave. Tiene operaciones para registrar un prototipo con una clave y para des-registrarlo. Los clientes pueden cambiar el registro o incluso navegar por él en tiempo de ejecución. Esto permite que los clientes extiendan el sistema y lleven un inventario del mismo sin necesidad de escribir código

2. Solución

- La parte más complicada del patrón *Prototype* es implementar correctamente la operación Clonar. Es particularmente delicado cuando las estructuras de objetos contienen referencias circulares
  - La mayoría de los lenguajes proporcionan algún tipo de ayuda para clonar objetos (*clone()*, constructor de copia, …). Pero estas facilidades no resuelven el problema de la "copia superficial frente a la copia profunda"
  - Una copia superficial es simple y a menudo suficiente. Pero clonar prototipos con estructuras complejas normalmente requiere una copia profunda, porque el clon y el original deben ser independientes. Por tanto debemos garantizar que los componentes del clon son clones de los componentes del prototipo. La clonación nos obliga a decidir qué será compartido, si es que lo será algo.

2. Solución

- Mientras que a algunos clientes les sirve el clon tal cual, otros querrán inicializar parte de su estado interno, o todo, con valores de su elección. Generalmente no pasamos dichos valores en la operación Clonar, porque su número variará de unas clases de prototipos a otras. Algunos prototipos pueden necesitar múltiples parámetros de inicialización; otros no necesitarán ninguno. Pasar parámetros en la operación Clonar impide tener una interfaz de clonación uniforme
  - Puede darse el caso de que nuestras clases prototipo ya definan operaciones para establecer las partes principales de su estado. Si es así, los clientes pueden usar estas operaciones inmediatamente después de la clonación.

3. Consecuencias

- Oculta al cliente las clases producto concretas, reduciendo así el número de nombre que conocen los clientes. Además, estos patrones permiten que un cliente use las clases específicas de la aplicación sin cambios
- Permite incorporar a un sistema una nueva clase concreta de producto simplemente registrando una instancia prototípica con el cliente. Esto es algo más flexible que otros patrones de creación, ya que un cliente puede instalar y eliminar prototipos en tiempo de ejecución

3. Consecuencias

- Los sistemas altamente dinámicos permiten definir comportamiento nuevo mediante la composición de objetos – por ejemplo, especificando valores para las variables de un objeto – y no definiendo nuevas clases. Podemos definir nuevos tipos de objetos creando instancias de clases existentes y registrando esas instancias como prototipos de los objetos cliente. Un cliente puede exhibir comportamiento nuevo delegando responsabilidad en su prototipo.
  - Este tipo de diseño permite que los usuarios definan nuevas "clases" sin programación. De hecho, clonar un prototipo es parecido a crear una instancia de una clase. El patrón prototipo puede reducir en gran medida el número de clases necesarias en un sistema.

3. Consecuencias

- El patrón *Factory Method* suele producir una jerarquía de clases Creador que es paralela a la jerarquía de clases de productos. El patrón *Prototype* permite clonar un prototipo en vez de decirle a un método de fabricación que cree un nuevo objeto. Por tanto, no es en absoluto necesaria una jerarquía de clases Creador. Este beneficio es aplicable principalmente a lenguajes como C++ que no tratan a las clases como objetos en toda regla.
- Algunos entornos de tiempo de ejecución permiten cargar clases en una aplicación dinámicamente. El patrón *Prototype* es la clase para explotar dichas facilidades en un lenguaje como C++.
  - Una aplicación que quiere crear instancias de una clase cargada dinámicamente no podrá hacer referencia al constructor de ésta estáticamente.

4. Relaciones

- ***Singleton*** para controlar un único *Prototype* global