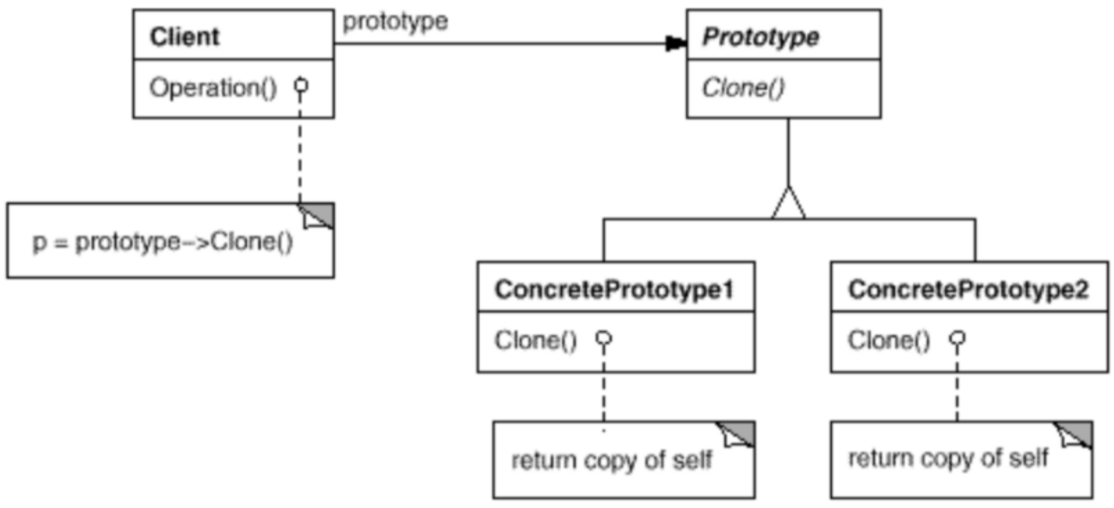
Design Patterns - Prototype Pattern

Prototype pattern refers to creating duplicate objects while keeping performance in mind. This type of design pattern comes under creational pattern as this pattern provides one of the best ways to create an object.

This pattern involves implementing a prototype interface that tells to create a clone of the current object. This pattern is used when the creation of an object directly is costly. For example, an object is to be created after a costly database operation. We can cache the object, returns its clone on the next request, and update the database as and when needed thus reducing database calls.



This pattern is really useful in cases when it’s cheaper to clone some existing instance than instantiate it from a class. This approach really helps when classes to instantiate are specified at run-time.

Building a hierarchy of factories that represents a hierarchy of products might have a huge cost, especially in cases when only a few combinations of state are possible for some concrete instance.

So the Prototype pattern should be considered a lightweight solution to get a copy of the existing class instance.

This pattern includes only two main roles:

Prototype — interface, and implementation of a class that can clone itself.

Client — creates a new instance by asking a Prototype to clone itself.

So, whenever the system needs to get a new class instance, it asks an internal Prototype method to clone itself and return a result.

Advantages of Prototype Design Pattern:

- Adding and removing products at run-time – Prototypes let you incorporate a new concrete product class into a system simply by registering a prototypical instance with the client. That's a bit more flexible than other creational patterns because a client can install and remove prototypes at run-time.
- Specifying new objects by varying values – Highly dynamic systems let you define new behavior through object composition by specifying values for an object's variables and not by defining new classes.
- Specifying new objects by varying structure – Many applications build objects from parts and subparts. For convenience, such applications often let you instantiate complex, user-defined structures to use a specific subcircuit again and again.
- Reduced subclassing – The factory Method often produces a hierarchy of Creator classes that parallels the product class hierarchy. The Prototype pattern lets you clone a prototype instead of asking for a factory method to make a new object. Hence you don't need a Creator class hierarchy at all.

Disadvantages of Prototype Design Pattern:

- Overkill for a project that uses very few objects and/or does not have an underlying emphasis on the extension of prototype chains.
- It also hides concrete product classes from the client
- Each subclass of Prototype must implement the clone() operation which may be difficult when the classes under consideration already exist. Also implementing clone() can be difficult when their internals includes objects that don't support copying or have circular references.

Prototype Example code:

Suppose we have to read data from our monster database before we can create a zombie. Then we have to modify this data in our game multiple times for each zombie. It won't a good idea to create zombies using the new keyword each time, and having to load the same or similar data again from the database. This would make the creation of a zombie horde very time-consuming.

```java
public class Zombie implements java.lang.Cloneable {

   // other code

   @Override
   public Object clone() {
      // appropriate code
   }
}
```

The clone() method has protected access in the Object class. We would override this method with a public method. The customary first step in any clone() implementation is to invoke the superclass's implementation of clone():

```
public Object clone() throws CloneNotSupportedException {
   «type» cloned = («type»)super.clone();  // type cast to correct class type
   cloned.mutableField   = (MutableField) mutableField.clone();
   cloned.immutableField = immutableField;
   cloned.primitiveField = primitiveField;
   ...
   return cloned;
}
```

If we are writing a clone() method for a direct subclass of Object, we will need to either catch CloneNotSupportedException or declare it in the throws clause. The Object class does a shallow field-by-field copy in low-level native code. We could alternatively create a copy constructor, and call it from the overridden clone() method. The requirements and mechanics of a copy constructor are possibly easier to understand than that of the clone() method. Sample code follows:

```
public class Zombie implements java.lang.Cloneable {

   // other code

   public Zombie (Zombie zombie) {
      // appropriate code - easy to understand
   }

   @Override
   public Object clone() {
      return new Zombie(this);
   }
}
```

Asmaa Raafat Ahmed Zohiry

Section: 2