Prototype pattern

From Wikipedia, the free encyclopedia

The **prototype pattern** is a creational design pattern in software development. It is used when the type of objects to create is determined by a prototypical instance, which is cloned to produce new objects. This pattern is used to:

- avoid subclasses of an object creator in the client application, like the abstract factory pattern does.
- avoid the inherent cost of creating a new object in the standard way (e.g., using the 'new' keyword) when it is prohibitively expensive for a given application.

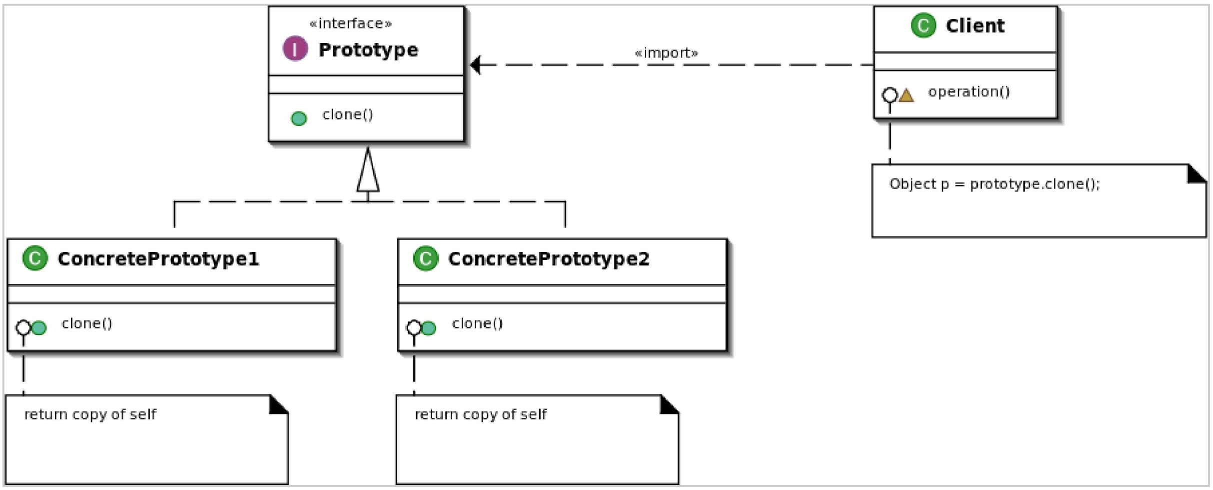
To implement the pattern, declare an abstract base class that specifies a pure virtual *clone()* method. Any class that needs a "polymorphic constructor" capability derives itself from the abstract base class, and implements the *clone()* operation.

The client, instead of writing code that invokes the "new" operator on a hard-coded class name, calls the *clone()* method on the prototype, calls a factory method with a parameter designating the particular concrete derived class desired, or invokes the *clone()* method through some mechanism provided by another design pattern.

The mitotic division of a cell — resulting in two identical cells — is an example of a prototype that plays an active role in copying itself and thus, demonstrates the Prototype pattern. When a cell splits, two cells of identical genotype result. In other words, the cell clones itself.[1]

Contents

Structure



UML class diagram describing the prototype design pattern

Rules of thumb

Sometimes creational patterns overlap — there are cases when either prototype or abstract factory would be appropriate. At other times they complement each other: abstract factory might store a set of prototypes from which to clone and return product objects (GoF, p126). Abstract factory, builder, and prototype can use singleton in their implementations. (GoF, p81, 134). Abstract factory classes are often implemented with factory methods (creation through inheritance), but they can be implemented using prototype (creation through delegation). (GoF, p95)

Often, designs start out using Factory Method (less complicated, more customizable, subclasses proliferate) and evolve toward abstract factory, prototype, or builder (more flexible, more complex) as the designer discovers where more flexibility is needed. (GoF, p136)

Prototype does not require subclassing, but it does require an "initialize" operation. Factory method requires subclassing, but does not require initialization. (GoF, p116)

Designs that make heavy use of the composite and decorator patterns often can benefit from Prototype as well. (GoF, p126)

The rule of thumb could be that you would need to clone() an *Object* when you want to create another Object *at runtime* that is a *true copy* of the Object you are cloning. *True copy* means all the attributes of the newly created Object should be the same as the Object you are cloning. If you could have *instantiated* the class by using *new* instead, you would get an Object with all attributes as their initial values. For example, if you are designing a system for performing bank account transactions, then you would want to make a copy of the Object that holds your account information, perform transactions on it, and then replace the original Object with the modified one. In such cases, you would want to use clone() instead of new.

Pseudocode

Let's write an occurrence browser class for a text. This class lists the occurrences of a word in a text. Such an object is expensive to create as the locations of the occurrences need an expensive process to find. So, to duplicate such an object, we use the prototype pattern:

```
class WordOccurrences is
  field occurrences is
    The list of the index of each occurrence of the word in the text.

  constructor WordOccurrences(text, word) is
      input: the text in which the occurrences have to be found
      input: the word that should appear in the text
    Empty the occurrences list
    for each textIndex in text
      isMatching := true
      for each wordIndex in word
        if the current word character does not match the current text character then
          isMatching := false
      if isMatching is true then
        Add the current textIndex into the occurrences list

  method getOneOccurrenceIndex(n) is
      input: a number to point on the nth occurrence.
      output: the index of the nth occurrence.
    Return the nth item of the occurrences field if any.

  method clone() is
      output: a WordOccurrences object containing the same data.
    Call clone() on the super class.
    On the returned object, set the occurrences field with the value of the local occurrences field.
    Return the cloned object.

text := "The prototype pattern is a creational design pattern in software development first described in desi
word := "pattern"d
searchEngine := new WordOccurrences(text, word)
anotherSearchEngine := searchEngine.clone()
```

(the search algorithm is not optimized; it is a basic algorithm to illustrate the pattern implementation)

C# Example

This pattern creates the kind of object using its prototype. In other words, while creating the object of Prototype object, the class actually creates a clone of it and returns it as prototype.You can see here, we have used MemberwiseClone method to clone the prototype when required.

```
public abstract class Prototype
{
    // normal implementation
    public abstract Prototype Clone();
}

public class ConcretePrototype1 : Prototype
{
    public override Prototype Clone()
    {
        return (Prototype)this.MemberwiseClone(); // Clones the concrete class.
    }
}

public class ConcretePrototype2 : Prototype
{
    public override Prototype Clone()
    {
        return (Prototype)this.MemberwiseClone(); // Clones the concrete class.
    }
}
```

Java Example

This pattern creates the kind of object using its prototype. In other words, while creating the object of Prototype object, the class actually creates a clone of it and returns it as prototype.You can see here, we have used Clone method to clone the prototype when required.

```
// Prototype pattern
```

```
    public abstract class Prototype implements Cloneable {
        public abstract Prototype clone();
    }

    public class ConcretePrototype1 extends Prototype {
        @Override
        public Prototype clone() {
            return super.clone();
        }
    }

    public class ConcretePrototype2 extends Prototype {
        @Override
        public Prototype clone() {
            return super.clone();
        }
    }
```

PHP Example

```
// The Prototype pattern in PHP is done with the use of built-in PHP function __clone()

abstract class Prototype
{
    public $a;
    public $b;

    public function displayCONS()
    {
        echo "CONS: {$this->a}\n";
        echo "CONS: {$this->b}\n";
    }

    public function displayCLON()
    {
        echo "CLON: {$this->a}\n";
        echo "CLON: {$this->b}\n";
    }

    abstract function __clone();
}

class ConcretePrototype1 extends Prototype
{
    public function __construct()
    {
        $this->a = "A1";
        $this->b = "B1";

        $this->displayCONS();
    }

    function __clone()
    {
        $this->displayCLON();
    }
}

class ConcretePrototype2 extends Prototype
{
    public function __construct()
    {
        $this->a = "A2";
        $this->b = "B2";

        $this->displayCONS();
    }

    function __clone()
    {
        $this->a = $this->a ."-C";
        $this->b = $this->b ."-C";

        $this->displayCLON();
    }
}

$cP1 = new ConcretePrototype1();
$cP2 = new ConcretePrototype2();
$cP2C = clone $cP2;

// RESULT: #quanton81

// CONS: A1
// CONS: B1
// CONS: A2
// CONS: B2
// CLON: A2-C
// CLON: B2-C
```

See also

- Function prototype

The Wikibook *Computer Science Design Patterns* has a page on the topic of: ***Prototype implementations in various languages***

References

1. Duell, Michael (July 1997). "Non-Software Examples of Design Patterns". *Object Magazine* **7** (5): 54. ISSN 1055-3614.

Sources

- Gamma, Erich; Helm, Richard; Johnson, Ralph; Vlissides, John (1994). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley. ISBN 0-201-63361-2.

Retrieved from "https://en.wikipedia.org/w/index.php?title=Prototype_pattern&oldid=725536482"

Categories: Software design patterns