Prototype

Also known as: Clone

Intent

Prototype is a creational design pattern that lets you copy existing objects without making your code dependent on their classes.

Problem

Say you have an object, and you want to create an exact copy of it. How would you do it? First, you have to create a new object of the same class. Then you have to go through all the fields of the original object and copy their values over to the new object.

Nice! But there's a catch. Not all objects can be copied that way because some of the object's fields may be private and not visible from outside of the object itself.

Copying an object from the outside is not always possible.

There's one more problem with the direct approach. Since you have to know the object's class to create a duplicate, your code becomes dependent on that class. It makes dependency doesn't scare you, there's another catch. Sometimes you only know the interface that the object follows, but not its concrete class, when for example, a parameter in a method accepts any objects that follow some interface.

Solution

The Prototype pattern delegates the cloning process to the actual objects that are being cloned. The pattern declares a common interface for all objects that support cloning. This interface lets you stay an object without coupling your code to the class of that object. Usually such an interface contains just a single `clone` method.

The implementation of the `clone` method is very similar in all classes. The method creates an object of the current class and carries over all of the field values of the old object into the new one. You can even copy private fields because most programming languages let objects access private fields of other objects that belong to the same class.

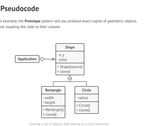
An object that supports cloning is called a *prototype*. When your objects have dozens of fields and hundreds of possible configurations cloning them might serve as an alternative to subclassing.

The Audi prototypes can be an alternative to subclassing

Here's how it works: you create a set of objects, configured in various ways. When you need an object like the one you've configured, you just clone a prototype instead of constructing a new object from scratch.

Real-World Analogy

In real life, prototypes are used for performing various tests before starting mass production of a product. However, in this case, prototypes don't participate in any actual production, playing a passive role instead.



The idea of a pair of pants

Since industrial prototypes don't really copy themselves, a much closer analogy to the pattern is the process of mitotic cell division (biology, remember?). After mitotic divisions a pair of identical cells is formed. The original cell acts as a prototype and takes an active role in creating the copy.

Structure

Basic implementation

Prototype registry implementation

Pseudocode

In this example, the **Prototype** pattern lets you produce exact copies of geometric objects, without coupling the code to their classes.

Cloning is an alternative way to create a class instance

All shape classes follow the same interface, which provides a cloning method. A subclass may call the parent's cloning method before copying its own field values to the resulting object.

```
// Base prototype.
abstract class Shape Is
  field X: int
  field Y: int
  field color: string

  // A regular constructor.
  constructor Shape(s: Shape) Is
    // ...

  // The prototype constructor. A fresh object is initialized
  // with values from the existing object.
  constructor Shape(source: Shape) Is
    this(source.X, source.Y, source.color)
    this.X = source.X
    this.Y = source.Y
    this.color = source.color

  // The clone operation returns one of the Shape subclasses.
  abstract method clone(): Shape

// Concrete prototypes. The cloning method creates a new object
// in one go by calling the constructor of the current class and
// passing every field of this object through its constructor.
// Performing all the actual copying in the constructor helps to
// keep the result consistent: the constructor will not return a
// half-baked clone even if an exception is thrown during object
// creation (a different kind of bug fun). Plus, we object
// can have a different type than the object it clones.
class Rectangle extends Shape Is
  field width: int
  field height: int

  constructor Rectangle(source: Rectangle) Is
    // A parent constructor call is needed to copy private
    // fields defined in the parent class.
    super(source)
    this.width = source.width
    this.height = source.height

  method clone(): Shape Is
    return new Rectangle(this)

class Circle extends Shape Is
  field radius: int

  constructor Circle(source: Circle) Is
    super(source)
    this.radius = source.radius

  method clone(): Shape Is
    return new Circle(this)

// Somewhere in the client code.
class ShapeClient Is
  field shapes: array of Shape

  constructor businessLogic() Is
    // Prototypes can clone any other shape.
    circle1.X = 10
    circle1.Y = 10
    circle1.radius = 20
    shapes.add(circle1)

    Circle anotherCircle = circle1.clone();
    shapes.add(anotherCircle)
    // The variable 'anotherCircle' will contain an exact copy
    // of the 'circle1' object.

    Rectangle rectangle = new Rectangle();
    rectangle.width = 10
    rectangle.height = 20
    shapes.add(rectangle)

  method businessLogic() : Is
    // Prototypes help because it lets you produce a copy of
    // an unknown object without coupling which its type.
    Array shapesCopy = new Array of Shapes.

    // For instance, we don't know the exact classes in the
    // shapes array. We do know that they extend the
    // shapes. But thanks to polymorphism, when we call the
    // 'clone' method on a shape the program checks its real
    // type and runs a proper method declared in that class.
    // So that clone. That's why we get proper clones
    // instead of a set of simple Shape objects.
    foreach (s: shapes.each()) Is
      shapesCopy.add(s.clone())

    // The 'shapesCopy' array contains exact copies of the
    // 'shapes' array's children.
```

Applicability

Use the Prototype pattern when your code shouldn't depend on the concrete classes of objects that you need to copy.

This happens a lot when your code works with objects passed to you from 3rd-party code via some interface. The concrete classes of these objects are unknown, and you couldn't depend on them even if you wanted to.

The Prototype pattern provides the client code with a general interface for working with other objects, that makes the client code more flexible and makes the client code independent from the concrete classes of objects that it clones.

Use the pattern when you want to reduce the number of subclasses that only differ in the way they initialize their respective objects.

Suppose you have a complex class that requires a laborious configuration before it can be used. There are several common ways to configure this class, and this code is scattered through your app. To reduce the duplication, you create several subclasses and put every common configuration code into their constructors. This solution has its own drawbacks, but now you have lots of dummy subclasses.

The Prototype pattern lets you use a set of pre-built objects configured in various ways as prototypes. Instead of instantiating a subclass that matches some configuration, the client can simply look for an appropriate prototype and clone it.

How to Implement

1. Create the prototype interface and declare the `clone` method in it. Or just add the method to all classes of an existing class hierarchy, if you have one.
2. A prototype class must define the alternative constructor that accepts an object of that class as an argument. The constructor must copy the values of all fields defined in the class from the passed object into the newly created instance. If you're changing a subclass you must call the parent constructor to let the superclass handle the cloning of its private fields.

   If your programming language doesn't support method overloading you won't be able to create a separate "prototype" constructor. Thus copying the objects data into the newly created clone will have to be performed within the `clone` method. Stick having this code in a regular constructor is safe because the resulting object returned fully configured right after you call the `new` operator.
3. The cloning method usually consists of just one line running a `new` operator with the prototype version of the current class. Remember, that every class should override the cloning method and use its own class name along with the `new` operator. Otherwise, the cloning method may produce an object of a parent class.
4. Optionally, create a centralized prototype registry to store a catalog of frequently used prototypes.

   You can implement the registry as a new factory class or put it in the base prototype class with a static method for fetching the prototype. This method should search for a prototype based on search criteria that the client code passes to the method. The criteria might either be put into a complex set of parameters or just be a simple string. Once appropriate prototype is found, the registry should clone it and return the copy to the client.

   Finally, replace the direct calls to the subclasses' construction with calls to the factory method of the prototype registry.

Pros and Cons

- Pros: You can clone objects without coupling to their concrete classes.
- Pros: You can get rid of repeated initialization code in favor of cloning pre-built prototypes.
- Pros: You can produce complex objects more conveniently.
- Pros: You get an alternative to inheritance when dealing with configuration presets for complex objects.
- Cons: Cloning complex objects that have circular references might be very tricky.

Relations with Other Patterns

- Many designs start by using **Factory Method** (less complicated and more customizable via subclasses) and evolve toward **Abstract Factory**, **Prototype** or **Builder** (more flexible, but more complicated).
- **Abstract Factory** classes are often based on a set of **Factory Methods**, but you can also use **Prototype** to compose the methods on these classes.
- **Prototype** can help when you need to save copies of **Commands** into history.
- Designs that make heavy use of **Composite** and **Decorator** can often benefit from using **Prototype**. Applying the pattern lets you clone complex structures instead of re-constructing them from scratch.
- **Prototype** isn't based on inheritance, so it doesn't have its drawbacks. On the other hand, Prototype requires a complicated initialization of the cloned object. **Factory Method** is based on inheritance but doesn't require an initialization step.
- Sometimes **Prototype** can be a simpler alternative to **Memento**. This works if the object, the state of which you want to store in the history, is fairly straightforward and doesn't have links to external resources, or the links are easy to re-establish.
- **Abstract Factories**, **Builders** and **Prototypes** can all be implemented as **Singletons**.

Code Examples

Why should you take this eBook on commerce?

- You'll learn something useful on the way to work.
- No need for the Internet, always ready and handy.
- Easy on your eyes with a choice of reading modes.
- One-time thing to carry and never-ever-a-fee!
- All devices supported: PDF,EPUB,MOBI,KF8 formats.

Learn more...

RETURN
← Builder
READ NEXT
Singleton →

© 2014-2024 Refactoring.Guru, All rights reserved.
Illustrations by Dmitry Zhart

Ukrainian office:
☎ PSP-Plus Sole&Ltd
Dmytrivska Street, 15
Kyiv, Ukraine, 01601
✉ Email: support@refactoring.guru

Spanish office:
☎ Dreamsoft Street
Calle Ribera del Loira, 16
Madrid, Spain, 28042
✉ Email: support@refactoring.guru