Introduction

Design patterns are similar to algorithms for developers. A design pattern provides a reusable structure or approach to solve a common design problem. This article will introduce you to a creational pattern called the "Prototype Pattern." A creational pattern attempts to solve the general issue of how to create class instances in your design. In particular, the prototype pattern attempts to solve the specific issue on how to create class instances for unknown class types.   
  
Description

"Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype."[GoF] Wow! That was a very wordy description for "cloning an object." This creational pattern relies on being able to create new objects by cloning an existing object — the object to be cloned is commonly known as the 'prototype' or 'template' object. Typically this means a deep clone but the definition given by GoF certainly doesn't preclude a shallow clone if applicable (see the considerations section for more on this).   
  
When and when not

The prototype pattern and the factory pattern are very similar in both intent and functionality. Both are creational patterns that will create objects of some interface without needing to specifically know the underlying class types.   
  
The main difference between the two patterns involve how objects are constructed. The factory pattern will, generally, construct an object using the same construction parameters each time. Each object will be initialized with the same state information and be roughly equivalent to each other. The prototype pattern, on the other hand, can use any cloneable object that is given to it — even if those objects are of the same class type but with different state information assigned to them. Each object then becomes the prototype or template for any objects cloned from them. An example can demonstrate this:

public void factoryDoSomething(Factory factory) {

Point pt = factory.createPoint();

... do something with the point ...

}

public void prototypeDoSomething(Point prototype) {

Point pt = (Point) prototype.clone();

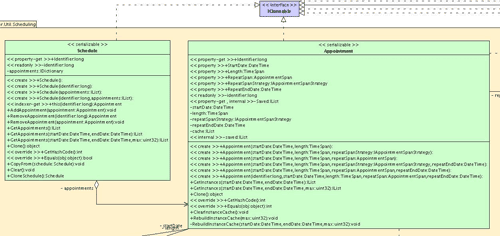
... do something with the point ...

}

You will notice that in the factoryDoSomething method, the point that is created is initialized in the same way and cannot be customized1. The prototypeDoSomething method can create a point from any other point with any type of state assigned to it. We could have called it with a "new Point(23,85)" or a "new Point(2929,59483)" and the cloned object would have similar state to those prototypes. The, ahem, point of the matter is that we can customize the state of the objects that will be created by the prototype pattern.   
  
Another major difference between the two patterns is the type of classes that can be created. A factory pattern will know (either through a registry or through subclassing) the various class types that can be created. The prototype pattern is not restricted to this as long as the object can be cloned. This allows the cloning of objects that are loaded dynamically from a class loader or ddl library or some other source where the class of the object is unknown either to the application or the component itself.   
  
Lastly, a factory pattern can handle a limited set of class types efficiently but gets bogged down as the number of types increase. The more types a factory pattern can create, the more cumbersome the factory pattern becomes from the overhead of managing those classes (both in the factory itself and in creating the necessary supporting classes). If the classes that are managed by the factory pattern are a hierarchy of classes for any given implementation, that hierarchy is generally repeated for each type regardless of how similar each class is to any other implementation. If the factory pattern uses a registry of some sort, the registry becomes harder to manage and certainly consumes more resources. On the other hand, the prototype pattern does not suffer from these issues since the pattern is focused on an existing object. The pattern is very scalable as the types increase because it has no management or overhead associated to it. If a class hierarchy is involved for each implementation, the pattern becomes very efficient because it can reduce the number of redundant classes involved since any class can be simply cloned (in other words, you can mix and match classes from different types where applicable).   
  
To sum up, a prototype pattern provides benefits over a factory pattern when the state of the objects should can be customized by the calling application, the class types are dynamically loaded or otherwise unknown or when there is a large number of class type implementations that potentially deal with a lot of similar classes.   
  
Considerations

The first consideration is how difficult it is to clone the object versus constructing the object. Some items are very easy to construct and clone — such as the Point object we demonstrated above. The state is well know and the constructor is very straight-forward in that it sets the state information to the constructor argument information. This type of object is well suited for either pattern. While this is probably true in about 75% of the objects (a pure guess by the way), the other 25% are more complicated. They may involve database interactions or have private state information that is difficult to clone. Likewise, they may have embedded objects that are difficult to clone themselves. The more complicated the object, the less applicable the object is to the prototype pattern unless each piece of the object, in turn, also implements the prototype pattern. This may not always be possible.   
  
The second consideration is whether a deep or shallow clone is necessary. A deep clone is when the cloned object and the original object contain no references to the same object. In other words, every object within the cloned object was in turn, cloned also. A shallow clone is when the cloned object does contain references to objects that are also shared in the original object. Consider cloning of a java List. Each list will contain the same object references. Changing the state of an element within either list will affect the other list since they share object references. This describes a shallow clone. The prototype pattern itself is silent on whether a cloned object should be a deep or shallow clone and in truth, it doesn't really matter from the point of the pattern itself. This is however, an important consideration on how the pattern is utilized within the component and certainly affects how the cloned items are treated in your component.   
  
Implementation details.

The implementation of the prototype pattern is very simple — define a clone method that classes must obey. For C# and Java, there are already existing interfaces that you can use (ICloneable and Cloneable respectively). The design can simply specify using on or the other interface and should state whether a deep, shallow or either clone is acceptable.   
  
Let's take a look at an example — the .NET Schedule component. This component provides a schedule of appointments and the span of time that appointment lasts. Figure 1 shows some of the relevant parts of the design (although it was reformatted from the original class diagram for our purposes). One of the requirements for this component was the ability to copy schedules. The copied schedule should contain all the appointments and their times from the original — and changes to one or the other will not affect other. In addition, the schedule will hold a list of the fairly complex Appointment type — all with their own state information.   
  
Because of this, the designer chose to utilize the prototype pattern to perform a deep clone of the schedule and all it's appointments within the schedule. The schedule and each appointment within it become the prototype for the new schedule.   
  
As shown in Figure 1, both the schedule and the appointment implement the ICloneable interface. When the Clone() method (or the strongly typed version CloneSchedule) is called on the Schedule, the following will be performed: 1. For each appointment in the current schedule, clone it and add it to a temporary list 2. Create and return a new Schedule object using the cloned appointments in the temporary list.   
  
Cloning the appointment will in turn, create a new Appointment based on the clone of all it's internal variables (of which, they all implement the ICloneable interface also [see the component for details on this]).

**Figure 1**  
  
[](http://www.topcoder.com/i/education/prototype01_lg.gif)

As you can see, an Factory pattern would be fully inadequate here because of the complexity of constructing the Appointment objects. Especially given that the Appointment is immutable once created and it depends on other complex objects that would also have to be constructed with specific state information. The prototype pattern was applied to simplify the whole design.