Factory: Encapsulating Object Creation

When you discover that you need to add new types to a system, the most sensible first step is to use polymorphism to create a common interface to those new types. This separates the rest of the code in your system from the knowledge of the specific types that you are adding. New types may be added without disturbing existing code ... or so it seems. At first it would appear that the only place you need to change the code in such a design is the place where you inherit a new type, but this is not quite true. You must still create an object of your new type, and at the point of creation you must specify the exact constructor to use. Thus, if the code that creates objects is distributed throughout your application, you have the same problem when adding new types-you must still chase down all the points of your code where type matters. It happens to be the *creation* of the type that matters in this case rather than the *use* of the type (which is taken care of by polymorphism), but the effect is the same: adding a new type can cause problems.

The solution is to force the creation of objects to occur through a common *factory* rather than to allow the creational code to be spread throughout your system. If all the code in your program must go through this factory whenever it needs to create one of your objects, then all you must do when you add a new object is to modify the factory.

Since every object-oriented program creates objects, and since it's very likely you will extend your program by adding new types, I suspect that factories may be the most universally useful kinds of design patterns.

Simple Factory Method

As an example, let's revisit the **Shape** system.

One approach is to make the factory a **static** method of the base class:

The **factory( )** takes an argument that allows it to determine what type of **Shape** to create; it happens to be a **String** in this case but it could be any set of data. The **factory( )** is now the only other code in the system that needs to be changed when a new type of **Shape** is added (the initialization data for the objects will presumably come from somewhere outside the system, and not be a hard-coded array as in the above example).

Note that this example also shows the new Python 2.2 **staticmethod( )** technique for creating static methods in a class.

I have also used a tool which is new in Python 2.2 called a *generator*. A generator is a special case of a factory: it's a factory that takes no arguments in order to create a new object. Normally you hand some information to a factory in order to tell it what kind of object to create and how to create it, but a generator has some kind of internal algorithm that tells it what and how to build. It "generates out of thin air" rather than being told what to create.

Now, this may not look consistent with the code you see above:

looks like there's an initialization taking place. This is where a generator is a bit strange - when you call a function that contains a **yield** statement (**yield** is a new keyword that determines that a function is a generator), that function actually returns a generator object that has an iterator. This iterator is implicitly used in the **for** statement above, so it appears that you are iterating through the generator function, not what it returns. This was done for convenience of use.

Thus, the code that you write is actually a kind of factory, that creates the generator objects that do the actual generation. You can use the generator explicitly if you want, for example:

So **next( )** is the iterator method that's actually called to generate the next object, and it takes no arguments. **shapeNameGen( )** is the factory, and **gen** is the generator.

Inside the generator-factory, you can see the call to **__subclasses__( )**, which produces a list of references to each of the subclasses of **Shape** (which must be inherited from **object** for this to work). You should be aware, however, that this only works for the first level of inheritance from **Item**, so if you were to inherit a new class from **Circle**, it wouldn't show up in the list generated by **__subclasses__( )**. If you need to create a deeper hierarchy this way, you must recurse the **__subclasses__( )** list.

Also note that in **shapeNameGen( )** the statement:

Is only executed when the generator object is produced; each time the **next( )** method of this generator object is called (which, as noted above, may happen implicitly), only the code in the **for** loop will be executed, so you don't have wasteful execution (as you would if this were an ordinary function).

Preventing direct creation

To disallow direct access to the classes, you can nest the classes within the factory method, like this:

Polymorphic Factories

The static **factory( )** method in the previous example forces all the creation operations to be focused in one spot, so that's the only place you need to change the code. This is certainly a reasonable solution, as it throws a box around the process of creating objects. However, the *Design Patterns* book emphasizes that the reason for the *Factory Method* pattern is so that different types of factories can be subclassed from the basic factory (the above design is mentioned as a special case). However, the book does not provide an example, but instead just repeats the example used for the *Abstract Factory* (you'll see an example of this in the next section). Here is **ShapeFactory1.py** modified so the factory methods are in a separate class as virtual functions. Notice also that the specific **Shape** classes are dynamically loaded on demand:

Now the factory method appears in its own class, **ShapeFactory**, as the **create( )** method. The different types of shapes must each create their own factory with a **create( )** method to create an object of their own type. The actual creation of shapes is performed by calling **ShapeFactory.createShape( )**, which is a static method that uses the dictionary in **ShapeFactory** to find the appropriate factory object based on an identifier that you pass it. The factory is immediately used to create the shape object, but you could imagine a more complex problem where the appropriate factory object is returned and then used by the caller to create an object in a more sophisticated way. However, it seems that much of the time you don't need the intricacies of the polymorphic factory method, and a single static method in the base class (as shown in **ShapeFactory1.py**) will work fine.

Notice that the **ShapeFactory** must be initialized by loading its dictionary with factory objects, which takes place in the static initialization clause of each of the shape implementations.

Abstract Factories

The *Abstract Factory* pattern looks like the factory objects we've seen previously, with not one but several factory methods. Each of the factory methods creates a different kind of object. The idea is that at the point of creation of the factory object, you decide how all the objects created by that factory will be used. The example given in *Design Patterns* implements portability across various graphical user interfaces (GUIs): you create a factory object appropriate to the GUI that you're working with, and from then on when you ask it for a menu, button, slider, etc. it will automatically create the appropriate version of that item for the GUI. Thus you're able to isolate, in one place, the effect of changing from one GUI to another.

As another example suppose you are creating a general-purpose gaming environment and you want to be able to support different types of games. Here's how it might look using an abstract factory:

In this environment, **Character** objects interact with **Obstacle** objects, but there are different types of Characters and obstacles depending on what kind of game you're playing. You determine the kind of game by choosing a particular **GameElementFactory**, and then the **GameEnvironment** controls the setup and play of the game. In this example, the setup and play is very simple, but those activities (the *initial conditions* and the *state change*) can determine much of the game's outcome. Here, **GameEnvironment** is not designed to be inherited, although it could very possibly make sense to do that.

This also contains examples of *Double Dispatching* and the *Factory Method*, both of which will be explained later.

Of course, the above scaffolding of **Obstacle**, **Character** and **GameElementFactory** (which was translated from the Java version of this example) is unnecessary - it's only required for languages that have static type checking. As long as the concrete Python classes follow the form of the required classes, we don't need any base classes:

Another way to put this is that all inheritance in Python is implementation inheritance; since Python does its type-checking at runtime, there's no need to use interface inheritance so that you can upcast to the base type.

You might want to study the two examples for comparison, however. Does the first one add enough useful information about the pattern that it's worth keeping some aspect of it? Perhaps all you need is "tagging classes" like this:

Then the inheritance serves only to indicate the type of the derived classes.

Exercises

1. Add a class **Triangle** to **ShapeFactory1.py**
2. Add a class **Triangle** to **ShapeFactory2.py**
3. Add a new type of **GameEnvironment** called **GnomesAndFairies** to **GameEnvironment.py**
4. Modify **ShapeFactory2.py** so that it uses an *Abstract Factory* to create different sets of shapes (for example, one particular type of factory object creates "thick shapes," another creates "thin shapes," but each factory object can create all the shapes: circles, squares, triangles etc.).