Power Objects: The Object Factory

Application development would not be complete without some mention of the Factory pattern. It provides an encapsulated means of manufacturing simple and complex objects on demand. Often though, Factory discussions concentrate on ad hoc implementations of the pattern, fulfilling the specific needs of an application versus providing a reusable solution. The Factory pattern defines a reusable idea, but this is not to say that some implementations of the Factory pattern could not be reusable objects. I'd like to take this opportunity to discuss a simple implementation of the Factory pattern that eliminates some of the need for having to write application specific code.

We begin with a simple interface that specifies the generic functionality of object creation: IFactory

```
public interface IFactory {
	public Object manufacture() throws ManufacturingException;
}
```

Note that the interface specifies no arguments to the manufacture method and returns only a generic Object reference. This tells us that an implementation of IFactory will most likely provide creation logic for a single type of object that is part of a larger set of objects. Next we have the simple exception class mentioned in the IFactory interface that provides for run time notification of creation issues using the Exception chaining available in jdk1.4: ManufacturingException:

```
public class ManufacturingException extends RuntimeException {

	public ManufacturingException() {
		super();
	}

	public ManufacturingException(Throwable arg0) {
		super(arg0);
	}
}
```

Our last object is an implementation of the IFactory interface. The constructor takes an argument of type Class, and uses reflection to manufacture objects of that class in the implementation of the IFactory interface.

```
public class ObjectFactory implements IFactory {

	private Class clazz;

	public ObjectFactory(Class clazz) {
		this.clazz = clazz;
	}
```

```
        /*
         * Implementation of manufacture.
         */
        public Object manufacture() throws ManufacturingException {
              try {
                    return clazz.newInstance();
              }
              catch (Exception exception) {
                    throw new ManufacturingException(exception);
              }
        }
}
```

This simple object is an effective means of dynamically creating application objects that can be reused. Of course no claim to usefulness is valid without some concrete examples so here they are:

Pooling:

Many applications use the tactic of object pooling in order to maintain a set of "ready-to-go" objects that can be increased or decreased depending on application needs. Consider a pool with the following constructor and initialization:

```
Pool pool = new Pool (new ObjectFactory(MyClass.class) , 10);
pool.initialize();
```

Here the pool can use the ObjectFactory to create as many objects as it needs (10 in this case) in order to prepare itself for use. The pool need not know the nature of the object it will store nor any of the logic used in object creation.

Dynamic Handling:

Once an object-oriented application has procured some data the next obvious question is: What object will be used to process or act upon that data? The solution often comes in the form of a series of "if statements" that use some piece of the data or a series of instanceof comparisons to determine the path the data will follow. This is often neither a dynamic or scalable solution (in the case of multiple string comparisons). By mapping a set of ObjectFactories to some deterministic piece of the data or the data's class we can easily achieve a dynamic and scalable means of handling as illustrated below:

```
processData(Data data) {

        // Use the data's class type to procure a handler factory
        ObjectFactory factory = factorymap.getDataHandler(data.class);

        // Alternatively use a deterministic piece of the data
        //      ObjectFactory factory = factorymap.getDataHandler(data.getType());
        DataHandler handler = (DataHandler) factory.manufacture();
        handler.handlData(data);
        …
        …
}
```

Additionally another object, perhaps a DataHandlerFactory could act as an intermediary for the proceessData method, by looking up the appropriate ObjectFactory, manufacturing an object and casting it to DataHandler.

Dynamic Initialization:

Often objects are configured using information from some sort of external source like a properties file. It is sometimes useful to be able to create and configure objects dynamically in order to increase application flexibility and reduce the amount of necessary code. An application could use a property (being a class name) to construct an ObjectFactory for use in composing one or more application objects like so:

```
configureApplication() {

        Properties properties = readPropertiesfromSomewhere();
        ApplicationObject  object = new ApplicationObject();

        ObjectFactory factory = new ObjectFactory(properties.getObjectAClassName);

        ObjectA  objectA = (ObjectA) factory.manufacture();

        factory = new ObjectFactory(properties.getObjectBClassName);

        ObjectB  objectB = (ObjectB) factory.manufacture();

        …
        …
}
```

Conclusion:

ObjectFactories have many uses beyond the three shown here. They provide a simple, effective means for creating or architecting the creation of “on demand” objects within an application. As such they can be an asset to any developer’s toolbox.