

Objects, Interfaces and Factories

GeoTools loves its interfaces - this drove me **mad** when I first started using the toolkit. One of the things about an interface is that you still need a way to create objects, and that is where a factory comes in. A factory is simply a class with a **create** method.

BEFORE (aka classes):

```
public class Foo {
    public Foo( int magic ){}
    public void doSomething(){
System.out.println( "boo!" ); }
}
```

AFTER (aka interfaces/factory):

```
public interface Foo {
    public void doSomething();
}
public interface FooFactory {
    public Foo createFoo( int magic );
}
```

Why would you do this? Because you want to work with others - including people you have not yet met! The idea is someone can come along and give you their own FooFactory and all of a sudden your code is doing new and wonderful things without evening noticing.

Now the alert will notice there is a problem here. How the heck do you make a FooFactory? A FooFactoryFactory seems a bit silly ... let's work on bootstrapping the process.

BootStrapping Factories

Using a Plugin System (and Defaults)

I have to continue with my above example for a bit ...

```
public class FooFinder(){
    FooFactory[] list();
}
```

... the interesting thing here is that this class really is **magic**. This is really the front end to the GeoTools plug-in system, we change the technology behind it every once in a while (but these FooFinders will always work). Our usual goal for shopping for a plug-in system is this: dropping a jar on the classpath and having more FooFactory instances show up in that above list.

Now we are usually a bit kinder then that and include some utility methods:

```
public class FooFinder(){
    FooFactory[] list();
    FooFactory defaultFactory();
    Foo create( int magic ); //
    create a Foo based on the default factory
}
```

How and where the defaults come from, and even the order of that original list is all kind of a mystery to me. The good news is it rarely matters.

The exception to this rule is with EPSGAuthorityFactories: I end up controlling the process by only including one of the **jars** on the classpath that I am interested in, but that is a lo tech answer.

Let's see this in use:

```
Foo foo = FooFinder.create( 3 );
foo.doSomething();
```

That was not too bad, and we can manage to work with others - just as long as they are setup as the default.

Using a Plugin System (and Metadata)

The other thing to do is use the plugin system and wander through that list and make an intelligent choice. We have to get the factories to tell us something interesting to make that choice, and that something is called (**cough**) metadata.

This time I am going to go for a bit more of a realistic example - Bar (okay not that much more real).

```
interface BarFactory {
    String getName();
    String getDisplayName();
    String getType();
    Bar create( URL url );
}
```

Now when we use **BarFinder** we will have something to talk about.

```
for ( BarFactory factory :
BarFactoryFinder.list() ){
    if( "good".equals( factory.getType() ) )
        return factory.create( URL );
}
throw new FactoryException("Could not find a
good factory");
```

Note we also have `getDisplayNames` to show to users, and `getName` to write in configuration files. Fun.

This is still a little bit backwards, it involves us knowing ahead of time (or at least asking a user) which factory to use. We can also turn the question around and ask the factory if it thinks it is up to the task.

```
interface BarFactory {
    boolean canProcess( URL url );
    Bar create( URL url );
}
```

Now when we use **BarFinder** we will have something to talk about.

```
for ( BarFactory factory :
BarFactoryFinder.list() ){
    if( factory.canProcess( url ) )
        return factory.create( URL );
}
throw new FactoryException("Could not
process "+url );
```

The fun part about all this is that we can use the plugin system to grow the capability of the geotools over time.

Using A Factory as a Parameter

I hope it is mostly clear how to use a Factory yourself, another thing you can do is pass a factory off to other code to get them to use it.

```
Foo parse( File file, FooFactory factory );
```

The factory is used when the parser needs to create a Foo instance to return to you. This is really nice as it allows you the user to use existing parsers to produce your own objects.

AbstractFactory and the Pattern Police

This is especially interesting with Factories that produce a variety of content that designed to work together.

```
interface Math {
    Number number();
}
interface Factory {
    Math literal( Number number );
    Math add( Math a, Math b );
    Math sub( Math a, Math b );
    Math mul( Math a, Math b );
    Math div( Math a, Math b );
}
```

This is often called an **AbstractFactory** by the GOF(aka Patterns Police). You will see the GOF uses these with a **Builder** pattern to make interesting data structures like expressions and documents.

Using Factories together!

So the question becomes what of a Factory that uses another Factory? Annoying, but it is actually very common.

In GeoTools we have the following chain:

- FeatureTypeFactory makes FeatureType
 - AttributeTypeFactory makes AttributeTypes
- FeatureFactory makes Features
 - GeometryFactory makes Geometry
 - CoordinateSequenceFactory makes CoordinateSequences

The only simplification we currently have (by accident) is that FeatureType **is** the FeatureFactory! There has to be a better way!

Using a Container

That still seems like a lot of bother, you have to talk to two classes, a Finder and a Factory, before you ever get anywhere ... there has to be an easier way. And there is 😊

A container is a strange offshoot of the J2EE scene. The formal definition is something along the line of a "blackboard that supports lifecycle" or something like that. **Lifecycle** being alpha and the omega of an Objects existence. A blackboard is a nice architecture that lets code loosely collaborate on stuff.

Justin will probably write us up a container article real soon now !

External link - [Inversion of Control Containers and the Dependency Injection pattern](#)

Mechanics of Setting up a Factory for Container Use

I am just going to jot down some notes on how Containers formalize Factories working together. Oh they call this "dependency injection" and the whole process "Inversion of Control" (IoC) - I am going to let Justin explain that one.

Because of this formalization, containers can do all sorts of magic that make this stuff easy to use and easy to write.

Constructor Injection

This is the easiest to understand:

```
public class GeometryFactory {
    public GeometryFactory(
        CoordinateSequenceFactory coordinateFactory
    ){
        ...
    }
}
```

There is a problem with that - it a **class** not an interface. Easy and straight forward though.

Setter Injection

And this is also straight forward:

```
public interface GeometryFactory {
    public void setCoordinateSequenceFactory(
        CoordinateSequenceFactory coordinateFactory
    );
}
```

Indeed you can do both at once.

Using a Container

Darn - Justin is going to have to rewrite this, but I thought I should at least provide the flavour of what a container

offers you as a user.

```
Parser parser = container.get( Parser.class
);
Feature feature = parser.file( file );
```

Where are all the factories? Well in the container somewhere, I am assuming Justin set it up based on all the things we mentioned before.

Sigh you don't believe me. let's do it by hand:

```
container.add(
MyFeatureTypeFactoryImpl.class );
container.add( MyGeometryFactoryImpl.class
);
container.add(
MyCoordinateSequenceFactoryImpl.class );
container.add( MyFeatureFactoryImpl.class );
```

And after that

```
Parser parser = container.get( Parser.class
);
Feature feature = parser.file( file );
```

Note: we only put in classes, it will construct instances of those factory classes and configure them using constructor or setter injection before constructing that parser for us to use.

Continued in [Supporting Hacks and Versions...](#)