



## Implementing Factories

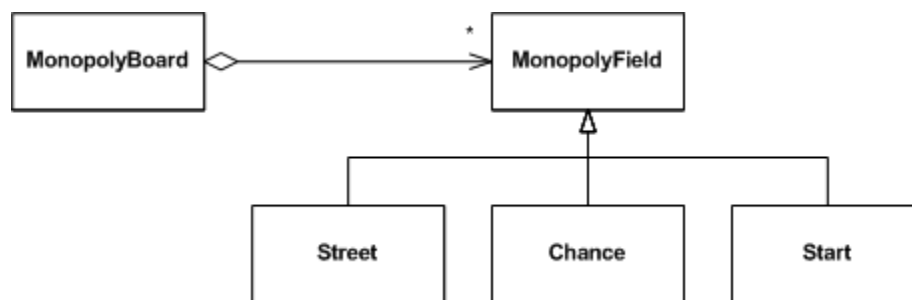
### *The Creational Series 2*

**Serge Beaumont, June 2006**

The article *Factories are about abstraction, not creation* told what factories are for, and when they should be used. This article will show **different ways to implement factories**. The complexity that is needed for a factory implementation depends on the required flexibility and how much effort is needed to construct a complete instance:

- **when should it be possible to configure** the factory repository: can it be coded and remain fixed for a release or should it be configurable at runtime?
- does the client of a factory need to **influence the subclass choice** in some way?
- how complex is the **algorithm to determine the correct subclass**?
- how difficult is it to **construct** a valid instance of the required class?

We'll use an example to illustrate the concepts as we go along. We have a Monopoly board, which has different types of fields.



We don't want the Monopoly board to depend on any subclass, just the `MonopolyField` abstraction. We will create a factory to create `MonopolyField` instances.

Before a factory is used by a client, the factory will have to be configured with all its options. First we'll keep this registration simple by hard-coding the options.

However complicated the factory may be, in the end there will always be a call to a constructor on a concrete class to get an instance. The **strategy for choosing this concrete class** can be implemented in



many ways, from hard-coding to using a full-blown Specification pattern.

The following factors influence the choosing strategy:

- selection algorithm
- options and arguments required to make a choice
- configurability

### **Simplest choice: no choice**

The simplest case is to have **no algorithm, no arguments and no configurability**:

```
package example;

public class MonopolyFieldFactory {
    public MonopolyField createField() {
        return new Street();
    }
}
```

In our case we need something more to be able to create the correct field type, but this simple case already serves a purpose. There are cases when the concrete class is fixed at runtime, but you still want to protect the client code from the creational dependency. Typical examples are supporting utilities like a logger, SAX parser, or a JDBC driver.

This simple case allows you to **switch implementations programmatically**:

```
package example;

public class MonopolyFieldFactory {
    public MonopolyField createField() {
        return new Chance();
    }
}
```

We changed from boards consisting of all Streets to boards consisting of all Chance fields.

### **Configurable choice**

We'll step up the complexity by adding one feature at a time. Let's **add configurability** to our factory. The main difference is calling the constructor through a meta-mechanism like reflection or the `java.lang.Class` class, and using an external configuration mechanism to determine what constructor to call. The solution



presented here is based on the JDK 5.0

`javax.xml.parsers.FactoryFinder`. The choice of class is based on a system property that is passed as a VM startup parameter:

```
> java MyApplication -Dfieldtype=example.Chance
```

The example is kept simple for clarity: you'll need to do some checks and implement fallbacks for a really robust solution. Take a look at the `FactoryFinder` source for a more comprehensive example.

```
package example;

public class MonopolyFieldFactory {
    private String SYSTEM_PROPERTY_FIELDTYPE = "fieldtype";

    public MonopolyField createField() throws ClassNotFoundException {
        return (MonopolyField) createInstance(getClassName());
    }

    private Object createInstance(String className) throws
        ClassNotFoundException {
        // Class.forName() can throw ClassNotFoundException
        Class providerClass = Class.forName(className);
        return providerClass.newInstance();
    }

    private String getClassName() {
        return System.getProperty("fieldtype");
    }
}
```

**There are many ways to configure** the class name. The choice depends on your configuration needs. You could use a properties file with `java.util.Properties`, use the more powerful `java.util.prefs.Preferences`, or set the classname with dependency injection.

### Mapped choice

So far our factory was not useful for our purpose: we need to be able to **choose an implementation** with every field we create. We want to avoid a dependency on the subclasses, so we need a way for the client to specify what he wants without a direct coupling to the implementation classes. There are patterns that cover this issue like Product Trader and Specification.

The simplest Specification is a String that is used as a lookup key. For this example I've **taken out configurability and reverted to a hard-coded list**. The main difference is that the **client passes an argument that specifies the required implementation**.



```
package example;

public class MonopolyFieldFactory {
    public String FIELD_TYPE_STREET = "street";
    public String FIELD_TYPE_CHANCE = "chance";
    public String FIELD_TYPE_START = "start";

    public MonopolyField createField(String type) {
        if (FIELD_TYPE_STREET.equals(type)) {
            return new Street();
        } else if (FIELD_TYPE_CHANCE.equals(type)) {
            return new Chance();
        } else if (FIELD_TYPE_START.equals(type)) {
            return new Start();
        }
    }
}
```

Using a `String` as a selector argument works up to a point. It gives a basic decoupling, but it is not very useful for advanced selection algorithms. That's where the Specification pattern comes in.

## Specification

A Specification is an object that you hand a candidate object and that will return a simple boolean to tell you if the **candidate conforms to the specification or not**. The nice thing about this pattern is that the elegant and simple interface can hide arbitrarily complex logic. (See the reference at the end of the article for the full description of the pattern)

```
package example;

public interface Specification {
    public boolean isSatisfiedBy(Object candidate);
}
```

The choice algorithm now uses `Strings` that correspond to the class names to select the implementation.

```
package example;

public class FieldSpecification {
    public String FIELD_TYPE_STREET = "example.Street";
    public String FIELD_TYPE_CHANCE = "example.Chance";
    public String FIELD_TYPE_START = "example.Start";

    private String type;

    FieldSpecification(String type) {
        assert type != null;
    }
}
```



```
    this.type = type;
}

public boolean isSatisfiedBy(Class candidate) {
    return type.equals(candidate.getName());
}
}
```

Now we'll only **change the choosing mechanism**. Later we will deal with more elegant ways to check all the options (no more `if..else if`), but this example is kept as close as possible to the previous example show the difference a Specification makes.

```
package example;

public class MonopolyFieldFactory {
    public MonopolyField createField(FieldSpecification spec) {
        if (spec.isSatisfiedBy(Street.class)) {
            return new Street();
        } else if (spec.isSatisfiedBy(Chance.class)) {
            return new Chance();
        } else if (spec.isSatisfiedBy(Start.class)) {
            return new Start();
        }
    }
}
}
```

The difference is that the **matching logic has been moved out of the `createField()` method**. This allows a client to pass in different Specification objects with different algorithms. The one we implemented checks the name of the class, but we could pass in a `RandomizingSpecification`, a `RoundRobinSpecification`, or a `ClassNameLengthSpecification`...

A client would **use this factory** by creating and configuring the Specification, and passing it to the factory as an argument.

```
package example;

public class FactoryTest1 {
    public static void main(String[] args) {
        MonopolyFieldFactory factory = new MonopolyFieldFactory();
        FieldSpecification spec = new
            FieldSpecification(FieldSpecification.FIELD_TYPE_CHANCE);

        MonopolyField field = factory.createField(spec);

        System.out.println(field.getClass().getName());
    }
}
```



We've seen how you can decouple the specification of a client's wishes from the choosing algorithm. Next we'll see how we handle the other responsibilities.

## **Registration**

In the previous examples I've stuck to the hard coded `if...else if` to show that a factory's choosing responsibility is separate from the registration responsibility. Registration is the responsibility of a factory to **know all implementations** that it needs to be able to create.

We will create a more advanced factory by removing the hard coded `if...else if` with a **dynamic registration mechanism**. A dynamic registration mechanism has two sides: a collection-based storage of options, allowing us to extend the list of options, and a registration mechanism to populate the collection.

### **First step: introduce a Collection**

First we replace the hard-coded `if...else if` with a hard-coded collection. I'm reverting back to a simple `String` based specification for this example.

```
public class MonopolyFieldFactory {
    public String FIELD_TYPE_STREET = "street";
    public String FIELD_TYPE_CHANCE = "chance";
    public String FIELD_TYPE_START = "start";

    private Map<String, Class> options = new HashMap<String, Class>();

    public MonopolyFieldFactory() {
        options.put(FIELD_TYPE_STREET, Street.class);
        options.put(FIELD_TYPE_CHANCE, Chance.class);
        options.put(FIELD_TYPE_START, Start.class);
    }

    public MonopolyField createField(String type) {
        assert options.get(type) != null;

        Class clazz = options.get(type);
        return clazz.newInstance();
    }
}
```

The main difference is that we've hidden the knowledge of all options from the choosing algorithm. The `createField()` method does not know the options, which it did in the `if...else if` version. With this separation we can **evolve the registration and choosing algorithm separately**. We will illustrate this by changing the choosing algorithm to one based on a `Specification`. Note that there still remains a



dependency: with a Specification-based solution the interface to the registration part changes as well. I've also refactored the Map to a Set. I could have kept the Map and just used `options.keys()`. What does *not* change is our registration mechanism: hardcoded options.

```
package example;

public class MonopolyFieldFactory {
    private Set<Class> options = new HashSet<Class>();

    public MonopolyFieldFactory() {
        options.add(Street.class);
        options.add(Chance.class);
        options.add(Start.class);
    }

    private Class chooseConcreteClass(Specification spec) {
        for (Class clazz : options) {
            if (spec.isSatisfiedBy(clazz)) {
                return clazz;
            }
        }
    }

    public MonopolyField createField(Specification spec) {
        Class clazz = chooseConcreteClass(spec);
        if (clazz != null) {
            return clazz.newInstance();
        } else {
            return null;
        }
    }
}
```

## Configured registration

Advanced registration mechanisms allow you to **defer the decision** which implementation classes will be used. Instead of hard-coding the options they can be made **configurable or even dynamically loaded** at runtime. The configuration of a factory can be done in many ways: a properties file, `java.util.prefs.Preferences`, an XML file, maybe even by loading configuration information from a database, LDAP or JNDI... Whatever the method, in our example the goal is always to load the `options` collection with the necessary items. For this example I'll use the Spring framework to configure our factory.

A Spring configuration snippet for our factory looks like this:

```
<bean id="fieldFactory" class="example.MonopolyFieldFactory">
    <property name="options">
        <set>
```



```
<value type="java.lang.Class">example.Street</value>
<value type="java.lang.Class">example.Chance</value>
<value type="java.lang.Class">example.Start</value>
</set>
</property>
</bean>
```

Since this is not an article about Spring I will not go into the details. The effect is that when a `MonopolyFieldFactory` is instantiated with a Spring `BeanFactory`, Spring looks for the `options` property (getter/setter pair), and loads the `Set` with the given classes. To change the options we only have to change the list of `value` tags.

Our factory class now looks like this:

```
public class MonopolyFieldFactory {
    private Set<Class> options = new HashSet<Class>();

    public Set<Class> getOptions() {
        return options;
    }

    public void setOptions(Set<Class> value) {
        this.options = value;
    }

    private Class chooseConcreteClass(Specification spec) {
        for (Class clazz : options) {
            if (spec.isSatisfiedBy(clazz)) {
                return clazz;
            }
        }
    }

    public MonopolyField createField(Specification spec) {
        Class clazz = chooseConcreteClass(spec);
        return clazz.newInstance();
    }
}
```

Behold the power of configuration through dependency injection: all hard-coded knowledge of the options is completely gone from the factory.

## Pluggable Factories

Configured registration is sufficient in many cases, but we can go one step further if we need it. We can introduce **run-time pluggability**.

Let's say we want to load different board types at run-time. We extend our example by deriving `MonopolyField` from `Field` and add new hierarchies like `ChessField` and `SnakesAndLaddersField`.



We will make the `FieldFactory` and `Field` abstractions a plug-in, where we can dynamically plug in `Field` implementations.

We need an initialisation mechanism that works simply by putting something on the classpath. The first chance you have at initialisation is when a class gets loaded, which takes an active call from your application. Such an initialisation mechanism would be ideal for plugins, because you could write a plugin jar, put it on the classpath, and hey presto, it gets registered. And there is a way to do this...

The JDK has a nice trick up its sleeve, through the `sun.misc.Service` class, also called the **Service Provider Interface (SPI)**. Officially you're not supposed to use it since Sun holds the right to change all the `sun.*` code without notice, but it's easy to set up (and change if need be), and it's been part of quite a release now, up to the latest 5.0, so I'm not too worried about it.

Take a look at your JDK: open up `rt.jar`, and look in `META-INF/services`. There you will find several files that have fully qualified classnames as filenames, and within those files you see a list of fully qualified classnames. That is the SPI registration mechanism. Each file refers to a class or interface, and the entries within the file are the classes implementing it.

In our example, we can put all the implementors of `Field` in a separate jar with a `FieldFactory` implementation that can create them. Let's call this jar `fieldplugin.jar`. The jar has the following files:

```
fieldplugin.jar
  META-INF/
    services/
      example.FieldFactory
  example/
    MonopolyFieldFactory.class
  chessexample/
    ChessFieldFactory.class
```

...and a bunch of `Field` implementations that the factories know how to create. We put a `META-INF/services` directory in this jar, and in this directory we put a file that has the fully qualified name of the interface we're implementing or base class we're subclassing:

```
file: example.MonopolyFieldFactory

# You can put comments in as well
example.MonopolyFieldFactory
chessexample.ChessFieldFactory
```



The conclusion from this file is that there is an `example.MonopolyFieldFactory` and a `chessexample.ChessFieldFactory` implementation of `example.FieldFactory` which we can use to create our fields. The code you need to get at that factory is very simple:

```
import sun.misc.Service;
import example.FieldFactory;

public class Board {

    FieldFactory factory;

    ...other code...

    private void initializeFieldFactory() {
        Iterator providers = Service.providers(FieldFactory.class);
        if (providers.hasNext()) {
            factory = (FieldFactory) providers.next();
        }
    }
}
```

The `Service.providers()` call will look in all jars on the classpath for a file in `META-INF/services` called `example.FieldFactory`, and returns an instance of every fully qualified classname it can find in those files. In our example we would get two instances: an `example.MonopolyFieldFactory` and a `chessexample.ChessFieldFactory`. In the example above we've simply taken the first implementation we could find, but you could query each factory instance to see if it is what you need (with a `Specification`, for instance), which would mean putting some extra code in the factory hierarchy for this purpose.

You can even switch implementations while your application is running (*hot pluggability!*). In JDK 5.0 there is a sixty second cache window, so you might need to wait up to a minute to see an effect.

## **Conclusion**

In this article we've seen different aspects that make up a Factory, and how we can implement more advanced versions that allow for more and more flexibility.

A final word: choose the right implementation for every occasion. A full-blown pluggable-factory-with-Specifications-all-over-the-place is not necessary all the time. Even the humble `if..else if` is an option for simple cases. This article should help you upgrade your Factory if you need to.



## References

<http://www.martinfowler.com/apSUPP/spec.pdf> – article about the Specification pattern.