Pattern Hatching

PLUGGABLE FACTORY, Part II

John Vlissides

C++ Report, February 1999

In this, the final installment on the PLUGGABLE FACTORY pattern, we pick up where we left off last time[1]—at the pattern's Structure diagram, shown in Figure 1. As you look at the structure, please remember that it's merely an example of a typical implementation, not a specification to be beholden to. You're welcome to vary your implementation as you see fit. The Consequences and Implementation sections will guide you in tailoring the pattern to your needs.

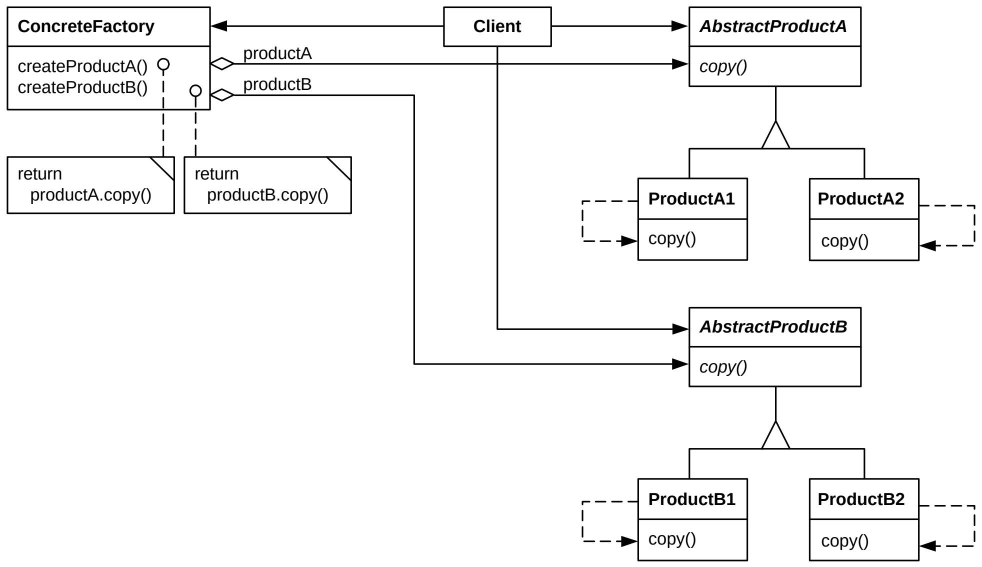

Figure 1: Structure of PLUGGABLE FACTORY

Another thing to keep in mind is PLUGGABLE FACTORY's intent, which is to specify and change product types dynamically without replacing the factory instance. The pattern achieves this intent with a structure similar to ABSTRACT FACTORY's but with notable differences contributed by the PROTOTYPE pattern.[2]

The most glaring of these differences is the absence of an AbstractFactory hierarchy. Figure 2 shows the vanilla ABSTRACT FACTORY structure. Notice that the only way a client can vary the kind of product produced is to use a different kind of ConcreteFactory object. In contrast, PLUGGABLE FACTORY lets clients vary product types by keeping the same ConcreteFactory instance and varying the prototypes it copies. The result is a simpler structure. The run-time relationships between factory and the product classes are a bit more complex, however, as subsequent sections in the pattern reveal.

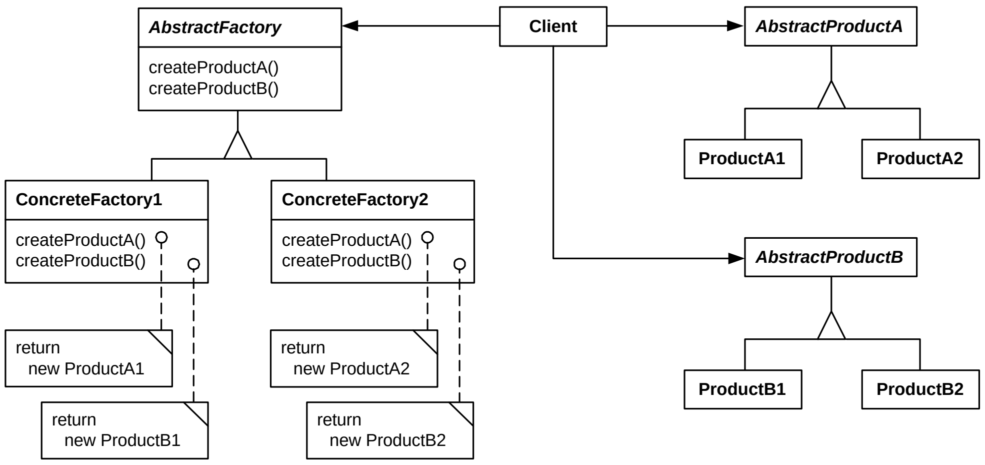

Figure 2: Vanilla ABSTRACT FACTORY *structure*

Participants

ConcreteFactory (WidgetFactory)

- keeps references to prototypical ConcreteProduct instances, each of which conforms to an AbstractProduct interface.
- for each AbstractProduct class, implements an operation that creates a ConcreteProduct object by copying a corresponding prototype.
- may include `get` and `set` operations for each prototype.

AbstractProduct (Button, Scrollbar)

- declares an interface for a type of product object.

ConcreteProduct (MotifButton, MotifScrollbar)

- defines a product object to be created by the concrete factory.
- implements an AbstractProduct interface

Client

- uses only interfaces declared by ConcreteFactory and AbstractProduct classes.

Collaborations

- Normally, a single instance of a ConcreteFactory class is created at run-time. This concrete factory creates product objects of related concrete types.
- To change one of the types of concrete product that will be created, replace the corresponding prototype with another type of product. If a client may do the replacement (as opposed to the factory itself), then the ConcreteFactory class must provide a public operation for setting the prototype.
- If it's ever necessary to change *all* concrete product types at once, it may be easier to replace the ConcreteFactory instance with an entirely new one containing a different set of prototypes.

Consequences

PLUGGABLE FACTORY shares some benefits and liabilities with ABSTRACT FACTORY. Both patterns insulate clients from concrete classes and their instantiation. Client code needn't change to create different types of products, as long as they're *compatible* types. And supporting new AbstractProduct types remains difficult.

Beyond these commonalities, however, the consequences of the two patterns diverge.

1. *PLUGGABLE FACTORY's class structure is simpler than ABSTRACT FACTORY's.* In PLUGGABLE Factory, changing the lone ConcreteFactory's prototypes changes the kinds of products that get created. ABSTRACT FACTORY needs a whole hierarchy of abstract and concrete factory classes to achieve comparable versatility. PLUGGABLE FACTORY gives you the flexibility of ABSTRACT FACTORY with fewer classes.

2. *The factory interface tends to be more complex in PLUGGABLE FACTORY.* ConcreteFactory must let clients specify the prototypes to use. Most simply, it can define constructors that take prototypes as parameters. If clients may update or replace the prototypes after construction (which is likely given this pattern's intent), then ConcreteFactory should also provide getter and setter operations for the prototypes. AbstractFactory classes are generally less flexible by comparison. They don't offer parameterized constructors, getters, or setters; so their interfaces tend to be simpler.

3. *Changing individual product types is easy.* Getter and setter operations provide access to the factory's product prototypes and let clients change them independently. A client can modify the prototype—to update its internal state, for example—or replace it with another instance of any compatible product type.

   With ABSTRACT FACTORY, changing the product type requires a new and different ConcreteFactory instance. That can be a more disruptive and error-prone operation because of the potential for dangling references to the old ConcreteFactory instance. And a conventional AbstractFactory interface provides no way to fine-tune product state.

4. *Consistency among products is hard to enforce statically.* Products are often designed to work as a family, which is a key assumption of ABSTRACT FACTORY. Buttons conforming to one look-and-feel standard, for example, shouldn't be mixed with menus conforming to another. ABSTRACT FACTORY can offer a compile-time guarantee that applications will use products from only one family at a time. The pattern keeps clients from creating products from different families by associating each family with a concrete class.

   PLUGGABLE FACTORY is weaker in its support of the "product family" concept. The pattern doesn't group related products under a static type and can therefore make no static guarantees about product consistency. Instead, a pluggable factory that detects illegal combinations of products must do so at run-time, as discussed in the Implementation and Sample Code sections.

5. *Exchanging families of product types is more difficult compared to ABSTRACT FACTORY.* Both patterns let you change the family of product classes in one fell swoop simply by replacing the factory instance. However, constructing a pluggable factory is often more involved than a conventional factory, because you have to supply prototypes to the pluggable factory's constructor. To compensate, you can provide parameterless constructors that configure the pluggable factory with default prototypes—assuming there are reasonable defaults.

Implementation

Here are three issues to consider when implementing a pluggable factory.

1. *Providing default prototypes.* Suppose that, by default, the WidgetFactory in the Motivation section[1] should create widgets comforming to the Windows look and feel standard. Thus if a client creates a WidgetFactory without prototypes, it will return WindowsButton and WindowsScrollBar instances. In C++ we can define a single constructor to support these defaults, using default parameters:

```
WidgetFactory (Button* btnProto = 0, ScrollBar* sbProto = 0) :
    _button(btnProto ? btnProto : new WindowsButton),
    _scrollbar(sbProto ? sbProto : new WindowsScrollBar) { }
```

Java, lacking default parameters, requires three constructors to achieve the same effect:

```
WidgetFactory () {
    _button = new WindowsButton();
    _scrollbar = new WindowsScrollBar();
}

WidgetFactory (Button btnProto) {
    _button = btnProto;
    _scrollbar = new WindowsScrollBar();
}

WidgetFactory (Button btnProto, ScrollBar sbProto) {
    _button = btnProto;
    _scrollbar = sbProto;
}
```

While more verbose, the Java implementation is marginally more efficient because it lacks run-time tests for null—the appropriate default behavior is determined statically. Of course, a C++ implementation may use overloaded constructors too if run-time overhead is an issue.

2. *Checking for illegal product type combinations.* The `WidgetFactory` constructors in the preceding item do nothing to prevent a client from pairing a button from one look and feel with a scrollbar from another. It can happen easily enough—by supplying a MotifButton prototype as the sole parameter, for example. We can reduce this particular risk by forcing clients to supply either two parameters or none. But mixing look and feels remains a distinct possibility.

   PLUGGABLE FACTORY makes it all but impossible to prevent this problem statically. If mixing is a concern, the factory should implement run-time tests that throw exceptions when mixing occurs. Such tests require a way to identify the look-and-feel family to which a widget belongs. The Sample Code section shows a couple approaches to implementing these tests.

   The factory should also force a client to specify a full set of prototypes, even if it only cares to specify one prototype. But that may be awkward if there are many kinds of products. For example, say WidgetFactory produces only two kinds of widgets, buttons and scrollbars. Then it should define two constructors: a parameterless one that creates default prototypes, and a two-parameter one that takes a button and a scrollbar and checks their compatibility:

```
WidgetFactory();
WidgetFactory(Button* btnProto, ScrollBar* sbProto);
```

   If clients are allowed to change the prototypes, then there should also be a single, two-parameter setter operation:

```
void setPrototypes(Button* btnProto, ScrollBar* sbProto);
```

   Supplying the prototypes in pairs lets the constructor and setter operations check the prototypes for consistency. Specifying the prototypes individually is unsatisfactory because it could leave the factory in an inconsistent state, however briefly.

   To see why, suppose WidgetFactory offered individual setter operations for the button and scrollbar prototypes. Both operations check to make sure the prototypes in the factory are from the same look-and-feel standard. Now suppose a client wants to use these operations to switch from the default Windows look and feel to Motif:

```
WidgetFactory factory;
// ...
```

```
    factory.setButtonPrototype(new MotifButton); // inconsistent with
                                                 // current
                                                 // WindowsScrollBar;
                                                 // throws exception

    factory.setScrollBar(new MotifButton); // consistency would have been
                                           // restored here
```

A combined `setPrototypes` operation can avoid this problem. However, that's a viable option only when there aren't a lot of prototypes to set—and that's probably *not* the case for WidgetFactory. In reality it would also offer pull-down and pop-up menus, text fields, and a host of other common widgets. It would be clumsy and inconvenient to specify prototypes for all widgets in `setPrototypes`.

The bottom line: If your ConcreteFactory produces many kinds of products, and you are concerned about mixing products from different families, then choose ABSTRACT FACTORY over PLUGGABLE FACTORY.

3. *Class names or objects as alternatives to prototypes.* Instead of specifying the types to instantiate with prototypes, Smalltalk and Java implementations can use class objects or strings identifying class names, as suggested in item 2 of ABSTRACT FACTORY's Implementation section.[2] Strings and class names are good alternatives when you're coding in these languages *and* either of the following are true:

   - Products do not maintain internal state that can vary across initializations (for example, the font of a widget changing after docking a notebook computer, as described in the Motivation section[1]).
   - The prototypes are too large to keep instantiated continuously, or copying them is unacceptably expensive compared to conventional instantiation.

Sample Code

The simplest implementation of the WidgetFactory example[1] looks like this in C++:

```
class WidgetFactory {
public:
    WidgetFactory(Button* btnProto, ScrollBar* sbProto);

    Button* createButton ()       { return _button->copy(); }
    ScrollBar* createScrollBar () { return _scrollbar->copy(); }

private:
    Button* _button;
    ScrollBar* _scrollbar;

};
```

where the constructor is implemented as shown in Implementation item 1. The Button and ScrollBar classes implement `copy` according to the PROTOTYPE pattern, normally as a deep copy. If clients are allowed to change the prototypes after the widget factory's construction, then you'll need to provide getter and setter operations. Here's a getter/setter pair for the button prototype:

```
Button* WidgetFactory::getButtonPrototype () { return _button; }

void WidgetFactory::setButtonPrototype (Button* newProto) {
    if (newProto != _button) delete _button;
    _button = newProto;
}
```

Note the `delete` operation in `setButtonPrototype`. We're assuming that the factory owns its prototypes, as the aggregation diamonds in the Structure diagram suggest (Figure 1). Therefore the setter operation is responsible for deleting the old prototype before adopting the new one—assuming they're not one and the same object. You can ignore this issue in Java, which will reclaim the old prototype automatically.

If you want to avoid mixing prototypes from different product families, then define a single setter operation that checks for type consistency at run-time (it's still okay to provide a getter operation for every product type):

```
void WidgetFactory::setPrototypes (Button* btnProto, ScrollBar* sbProto) {
    if (incompatible(btnProto, sbProto)) {
        throw IncompatiblePrototypes(this, btnProto, sbProto);
    }

    if (btnProto != _button) delete _button;
    if (sbProto != _scrollbar) delete _scrollbar;

    _button = btnProto;
    _scrollbar = sbProto;
}
```

`incompatible` is a boolean-returning operation that abstracts the process of checking for compatibility among the prototypes. Before we look at implementations of this operation, consider a couple of other things about `setPrototype`'s implementation. First, it throws an exception when `incompatible` detects an incompatibility. The type of exception is entirely up to you; here it is a simple class that bundles the factory with the prototypes deemed incompatible:

```
class IncompatiblePrototypes {
public:
    IncompatiblePrototypes (
        WidgetFactory* factory, Button* button, ScrollBar* sb
    );

    WidgetFactory* getFactory () { return _factory; }
    Button* getButton ()         { return _button; }
    ScrollBar* getScrollBar ()   { return _scrollbar; }

private:
    WidgetFactory* _factory;
    Button* _button;
    ScrollBar* _scrollbar;
};
```

Clients that catch the exception may access the prototypes, fix the incompatibility, and retry the `setPrototypes` operation on the factory.

The second thing to note about `setPrototypes` is that it also deletes old prototypes, still under the assumption that the factory owns them. The checks for identity are crucial, because there will be times when a client wants to change just one of the prototypes. The current interface forces the client to supply a prototype even if the client doesn't want it changed. If the client specifies an existing prototype, the identity check will avert a misguided deletion.

If it's common for clients to change just one prototype at a time, then supplying the unchanged prototypes will be inconvenient. You can make things easier by letting a null value signify no change:

```
void WidgetFactory::setPrototypes (Button* btnProto, ScrollBar* sbProto) {
    btnProto = btnProto ? btnProto : _button;
    sbProto = sbProto ? sbProto : _scrollbar;

    // remaining implementation as before
}
```

Now let's look at how we might implement `incompatible`. The main challenge lies in determining which widget classes belong to which look-and-feel family. It's challenging because the mapping of classes to look-and-feel isn't explicit in the code. ABSTRACT FACTORY defines such a mapping explicitly in the form of a ConcreteFactory subclass for each family. PLUGGABLE FACTORY defines just one ConcreteFactory, leaving no other place to define the mapping.

How do you define the mapping? One way is with a map of class names to look-and-feels:

```
class WidgetFactory {
public:
    // ...

    static void mapping(const string& className, const string& lookAndFeel);
    static void unmap(const string& className);

    static const string& getLookAndFeel(const string& className);

private:
    map<string, string> _mapping;
};
```

`mapping` puts the given (`className`, `lookAndFeel`) pair into the map, and `unmap` erases the mapping for the given class name. `getLookAndFeel` returns the look and feel mapped to the given class name, if it has been mapped; otherwise it returns an empty string.

`incompatible` uses the `typeid` operator to get the class name of each prototype. Then it compares the look and feels mapped to these names:

```
bool WidgetFactory::incompatible (Button* button, Scrollbar* scrollbar) {
    string btnName = typeid(button).name();
    string sbName = typeid(scrollbar).name();

    return getLookAndFeel(btnName) != getLookAndFeel(sbName);
}
```

`getLookAndFeel` simply looks the class name up in the map:

```
const string& WidgetFactory::getLookAndFeel (const String& className) {
    map<string, string>::const_iterator i = _mapping.find(className);

    return (i == _mapping.end()) ? "" : *i;
}
```

Of course, this assumes someone has mapped widget class names to their appropriate look and feels. In other words, someone, somewhere must have said:

```
WidgetFactory::mapping("MotifButton", "Motif");
WidgetFactory::mapping("MotifScrollBar", "Motif");
WidgetFactory::mapping("PMButton", "PM");
WidgetFactory::mapping("PMScrollBar", "PM");
WidgetFactory::mapping("WindowsButton", "Windows");
WidgetFactory::mapping("WindowsScrollBar", "Windows");
// etc.
```

That begs the question of where this code gets implemented. There are at least three choices:

1. *In the WidgetFactory*, perhaps in its constructor. However, you'll have to change that constructor if and when new widget types get defined, or when you target a new look and feel.
2. *In a client of WidgetFactory*, say, in a global initialization routine. That may be more extensible than putting the code in WidgetFactory, but it's still less automatic than we might like.
3. *In the widgets themselves.* Widgets can do the registration in their constructor. The trick is doing it just once; repeated registration of the same mapping is unnecessary and promotes inefficiency. A simple approach to one-time registration tests a static variable:

```
MotifButton::MotifButton () {
    static bool unmapped = true;
```

```
        if (unmapped) {
            WidgetFactory::mapping("MotifButton", "Motif");
            unmapped = false;
        }

        // ...
    }
```

There is an entirely different approach to mapping widget types to look-and-feel standards that doesn’t require explicit mapping code. It relies on a simple widget naming convention wherein a widget’s class name concatenates the look-and-feel’s name with the generic name for the widget. For example, if we use “Windows” to denote the Windows look and feel, then the conventional name for the Windows button class is “WindowsButton.”

This naming convention lets us eschew explicit mapping code by implementing `incompatible` like this:

```
bool WidgetFactory::incompatible (Button* button, Scrollbar* scrollbar) {
    string btnName = typeid(button).name();
    string sbName = typeid(scrollbar).name();

    string btnLookAndFeel = btnName.substr(0, btnName.find("Button")-1);
    string sbLookAndFeel = sbName.substr(0, sbName.find("ScrollBar")-1);
        // strip off generic widget name to obtain look-and-feel name

    return btnLookAndFeel != sbLookAndFeel;
}
```

Notice there’s no longer a need for a `getLookAndFeel` operation.

Here’s the same approach implemented in its entirety in Java:

```
class WidgetFactory {
    public WidgetFactory (Button btnProto, ScrollBar sbProto) {
        _button = btnProto;
        _scrollbar = sbProto;
    }

    public Button createButton ()             { return _button.copy(); }
    public ScrollBar createScrollBar ()       { return _scrollbar.copy(); }

    public Button getButtonPrototype ()       { return _button; }
    public ScrollBar getScrollBarPrototype () { return _scrollbar; }

    public void setPrototypes (Button btnProto, ScrollBar sbProto) {
        if (incompatible(btnProto, sbProto)) {
            throw new IncompatiblePrototypes(this, btnProto, sbProto);
        }

        _button = btnProto;
        _scrollbar = sbProto;
    }

    protected boolean incompatible (Button button, Scrollbar scrollbar) {
        String btnName = button.getClass().getName();
        String sbName = scrollbar.getClass().getName();

        String btnLookAndFeel =
            btnName.substr(0, btnName.indexOf("Button")-1);
        String sbLookAndFeel =
            sbName.substr(0, sbName.indexOf("ScrollBar")-1);

        return btnLookAndFeel.compareTo(sbLookAndFeel);
    }
```

```
        private Button _button;
        private ScrollBar _scrollbar;
    }
```

Known Uses

I have good news and bad news about this section. First the good news: Lacking citable examples of PLUGGABLE FACTORY, I posted a request for some to the Gang of Four mailing list* and received many responses. Here are two representative examples, both refreshingly non-GUI-oriented. The first is from Anthony Lauder:[3]

> *I used a prototype-based abstract factory extensively. It was a central component in a large securities clearance system developed for an international bank in Luxembourg. The basic idea was that new classes were loaded at run-time via a dynamic linker and registered a prototype of themselves along with a symbolic name to a common abstract factory. New classes could then be developed without modifying the common framework. Applications would read in from a configuration file the symbolic name of classes they should be using and pass the symbolic names of classes to the abstract factory when object were to be created. If a named class was not known to the abstract factory, it would use the dynamic linker to look it up, link it in, and generate a prototype, which was then stored against the symbolic name for subsequent cloning.*

Peter Shillan's examples were on a more systems-y (not to say "lower") level:[4]

> *As for [PLUGGABLE FACTORY], I'm using it now in a Client/Server System to provide plug-in components.*
>
> *One use I make is a network system. The abstraction of an EndPoint is chosen and various kinds of EndPoint (client, server, serverclient) can be used. The actual network mechanism (UDP, TCP/IP, MSMQ) is hidden in the lower levels. A Comms "service" gives the user the correct type of EndPoint.*
>
> *I have found this useful in other areas too, such as implementing application services on a server differently depending on the underlying OS.*

These and other examples I received left no doubt in my mind that PLUGGABLE FACTORY is indeed a real pattern, not something synthetic. That's the good news. The bad news is that even though I specifically asked for citable examples, *not one* of those I received is published in archival form. And yet, I can hardly point the finger here—my own examples have the very same problem.

Why is this? Is it because PLUGGABLE FACTORY is too new, too specialized, or too … too … *embarrassing* to write up? Surely not. People arrived at it independently, didn't they? It has proved useful time and again, hasn't it? Maybe I haven't been looking hard enough.

Or maybe people just haven't been good about writing up their experiences. If so, let me encourage you to amend your New Year's resolutions with a promise to disclose one use of PLUGGABLE FACTORY—or any other combination of existing patterns—by year's end. I bet someone ends up thanking you for it. In the meantime, if you have a published known use of PLUGGABLE FACTORY, I'd love to hear about it. I'll even throw in a little gift (*very* little—don't get excited).

* To subscribe, send a note to gang-of-4-patterns-request@cs.uiuc.edu with the subject "subscribe".

Related Patterns

Dirks Bäumer and Riehle have written extensively on creational patterns that discriminate product types using abstract specifications—prototypes being the simplest example. In particular, their PRODUCT TRADER pattern[5] offers several variants of and alternatives to PLUGGABLE FACTORY.

Acknowledgments

Erich Gamma, Richard Helm, and Dirk Riehle gave me lots of good feedback.

References

[1] Vlissides, J. Pluggable factory, part I. *C++ Report*, November/December 1998, pp. 52–56, 68.
[2] Gamma, E., et al. *Design Patterns*, Addison–Wesley, Reading, MA, 1995.
[3] Lauder, A. E-mail communication, Nov. 2, 1998.
[4] Shillan, P. E-mail communication, Nov. 2, 1998.
[5] Bäumer, D. and D. Riehle. Product Trader, in *Pattern Languages of Program Design 3*, Addison–Wesley, Reading, MA, 1998.