A few important patterns and their connections

Perdita Stevens

School of Informatics
University of Edinburgh

Plan

- Singleton
- Factory method
- Facade

and how they are connected.

You should understand how to use the patterns individually, but more importantly, how they are related and how they contribute to good OO design.

See the end of this slide set for which patterns are examinable. NB not spending a lot of lecture time on patterns this year: that is because there are so many good sources of information on them elsewhere, *not* because they aren't important. See schedule page.

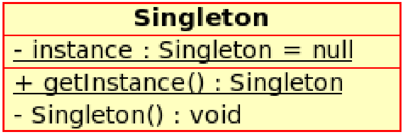
Singleton

Common problem: In OO systems a class often has only one object. Sometimes, it's important to ensure that it only has one object.

E.g., it's maintaining an important datum that needs to be held consistently, in only one version; or it's connecting to an external system with which your system should be having only one "conversation".

Solution: Singleton, one of the simplest patterns, ensures this. Key element: make the constructor private so that you can control how objects are created.

Singleton: class diagram



Note the notation (underlining) for the class-level (static, in Java) attribute and operation. These are essential, since the constructor is private!

Image: Wikipedia

Notes on Singleton

- Advantage: lazy instantiation possible – the instance need not actually be created unless or until it is needed.
- Drawback: introduces global state.
- Drawback: great care is needed in multi-threaded applications.
- Advantage: often useful in conjunction with other patterns, e.g. Factory Method, Facade (coming up).
- Use sparingly

Exercise

Read the Wikipedia article on Singleton, which includes many implementations and discussions of their pros and cons, as well as discussion of the use and abuse of Singleton. For extra education, read the Talk page too.

new considered harmful?

In typical OO programming, the clients of `Classname` use `...new Classname...` code to control object instantiation directly.

This has two effects:

1. responsibility for deciding whether, when, and how objects are created rests with the clients; it may be widely distributed through the system, especially if there are many clients;
2. `new` is not polymorphic. It takes a single specific class name and creates an object of exactly that class. So a client's code must be modified, if it is to be made to work with a subclass of `Classname`.

These effects may or may not be problems – it depends on the context.

new considered sometimes harmful, if misplaced

Alternatives to (uncontrolled use of) new

The Singleton pattern can be seen as a way to give an alternative to point 1. A client of a Singleton class can still get an instance, but the responsibility for actually creating the instance belongs to the Singleton itself.

Note that, despite the name, there could even be more than one object created – the point is that clients don't control how many there are, the Singleton class itself does.

With a private constructor, we can't subclass a Singleton, so we don't address 2. With care, we could make the constructor protected and do so... But would we ever want to? This takes us to Factory Method.

Factory Method

Common problem: your class needs to own an object and use its services. The services you need are described by a fairly abstract class/interface (`Product`, say) which has various subclasses and/or a complicated creation process, that shouldn't be your class's business.

Only one line of your code depends on which kind of `Product` you're going to have and how it's going to be built:

```
Product p = new ConcreteProduct(...);
```

Solution: Instead, get something else to build and give you a `Product` so that *all* your code is written purely in terms of `Product`.

In effect, your class says "Give me an appropriate `Product`" and declines to concern itself with exactly what is returned or how it is created.

Factory Method: class diagram

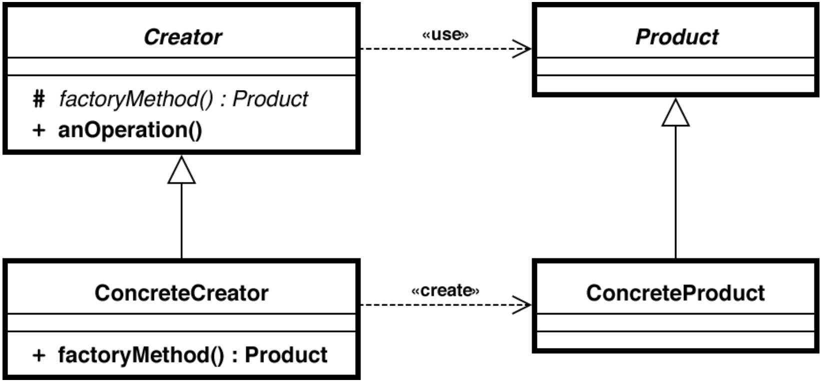


Image: `http://stackoverflow.com/questions/5739611/differences-between-abstract-factory-pattern-and-factory-method`

Where does a factory method live?

In a factory, of course :-)

In most versions of the Factory Method pattern, and in the original Gamma description, it's important that the factory that creates a product is in a different class – and class hierarchy – from the product itself.

Without that, we don't get advantage 2. above (making object creation polymorphic). We want depending on a factory class not to imply depending on the concrete product class, and if they're the same class, of course it does!

But some other advantages of Factory Method apply when a product can be its own factory, and these days this is sometimes also included in Factory Method. Other sources give this a different name, e.g. static factory method, and get irate when people lump them together. (The sourcemaking page is a little ambivalent, I think: watch out.)

Static factory method example

```
class Point {
private float x, y; // secretly, we use Cartesian coordinates inside
private Point (float xin, float yin) { x = xin; y = yin; }
public static Point createCartesianPoint(float x, float y) {
   return new Point(x, y);
}
public static Point createPolarPoint(float r, float theta) {
   // calculate x and y using trigonometry...
   return new Point(x, y);
}
```

NB because we can't override static methods, it's not straightforward to get all the benefits of Factory Method this way.

Relation between Factory Method and Singleton

How does your class contact the object that provides the factory method? One possibility: it may be a Singleton.

In fact, the Singleton's `getInstance()` method is a static factory method, so if you count that as Factory Method, you can see Singleton as a special case of Factory Method.

A simple example of Factory Method

is the single method

`ContentHandler createContentHandler(String mimetype)`

of the interface `ContentHandlerFactory` in the JDK.

A class that implements `ContentHandlerFactory` must implement that method, i.e. provide a way for clients to supply a string describing a mime type and get back a(n appropriate, we hope!) `ContentHandler`.

This makes it possible to write clients that work uniformly on all `ContentHandlers`, and do not have to concern themselves with what subclasses `ContentHandler` may have, or indeed, with how mime type strings map to those classes.

Naming: a further advantage of Factory Method

In the `Point` example, we used meaningful names to distinguish different ways of creating an object.

More generally: could be different in argument list, or meaning of arguments (as in `Point`) or in other characteristics e.g. efficiency.

Constructor overloading doesn't get you all the flexibility you want, and leads to unreadable code anyway.

NB this is an advantage we get both from Factory Method and from static factory methods. E.g.

```
Product p = factory.createSpaceOptimisedProduct(17);
```

might be creating an object of a subclass of `Product`, or not: we don't need to know.

Visibility of constructor

If your factory is not the same class as the product, the product class still needs a public (or at least, package visible) constructor – or its own static factory method! – as the factory will need to use it.

Inside the product class, you have a choice. Constructor could be:

- public: maximum flexibility for clients. If you've added the factory method to old code, leaving the public constructor means old clients don't have to be rewritten.
- private: now your objects can *only* be created via the factory method(s). The class cannot be subclassed.
- protected: now the class can be subclassed, and subclasses might add new factory methods. Risk and benefit.

Abstract Factory

An Abstract Factory is an object containing a collection of related Factory Methods.

Useful where which family of Product subclasses is appropriate is determined by one condition, e.g., which operating system we're running on.

Classic example: GUI toolkits. See for example `java.awt.toolkit`.

Facade

Common problem: you have roughly separated your classes into two packages, but there are several dependencies of classes in package A on classes in package B. It becomes hard to work out what the effect of a change in package B will be, and developers who want to use the services of package B have to understand the detail of what's inside it.

Solution: you create a Facade class whose job it is to present a single, simple interface to package B. All requests for services from anything in package B are sent to an object of the Facade class. This object may just forward the request to the right object, or it may do more complex things.

Facade: class diagram

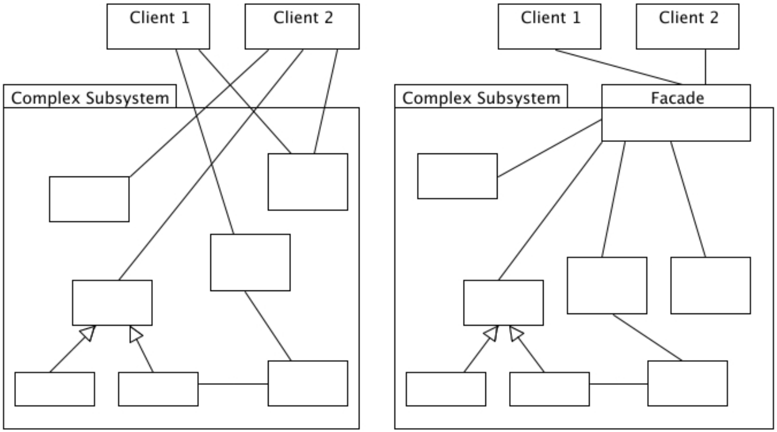


Image: `http://best-practice-software-engineering.ifs.tuwien.ac.at/patterns/facade.html`

Notes on Facade

- Advantage: very useful way to control dependencies
- Advantage: hides a multitude of sins. Lets you redesign a subsystem behind a facade without impacting what is outside.
- Disadvantage: incurs the cost of extra method calls (usually not a problem).
- The Facade may be a Singleton, but this isn't always necessary.

Creational patterns

- * Abstract Factory
- Builder
- * Factory Method
- Prototype
- * Singleton

(Learn those with *)

Structural patterns

- Adapter
- Bridge
- * Composite
- * Decorator
- * Facade
- Flyweight
- Proxy

(Learn those with *)

Behavioral patterns

- Chain of responsibility
- Command
- Interpreter
- Iterator
- Mediator
- Memento
- * Observer
- * State
- * Strategy
- Template method
- * Visitor

(Learn those with *)