A few important patterns

Perdita Stevens

School of Informatics
University of Edinburgh

Plan

- Singleton
- Factory method
- Facade
- ...more if time, but only those patterns mentioned in lecture slides or tutorial sheets are examinable...

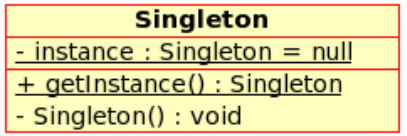
Singleton

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

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

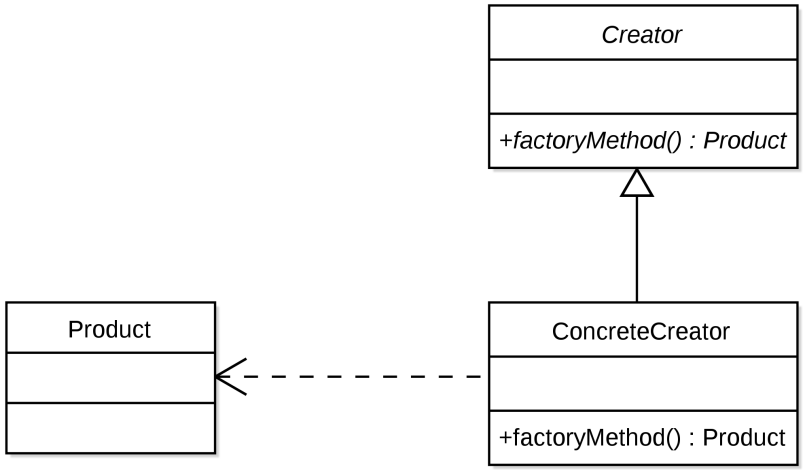


Image: Wikipedia

Notes on Factory Method

- How does your class contact the object that provides the factory method? Various ways: it may be a Singleton; or there may be some coordinator object that you can ask for a reference to the creator object. NB the creator you are given may be a `VerySpecialCreator` object, creating `VerySpecialProduct` objects – your code doesn't care. (Provided LSP is satisfied!)
- Common problem of unit testing, esp. when the code wasn't designed for testability: object needs to create something expensive or dangerous before it works. How do you test it without creating the expensive or dangerous thing? Make the creation into a factory method *in this object*. Then create subclass that overrides the factory method with dummy creation, and test that.
  (If you were designing for testability, you'd probably use Dependency Injection, see later.)
- The `Product` is often given a private constructor so that it can only be created by the factory method.

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

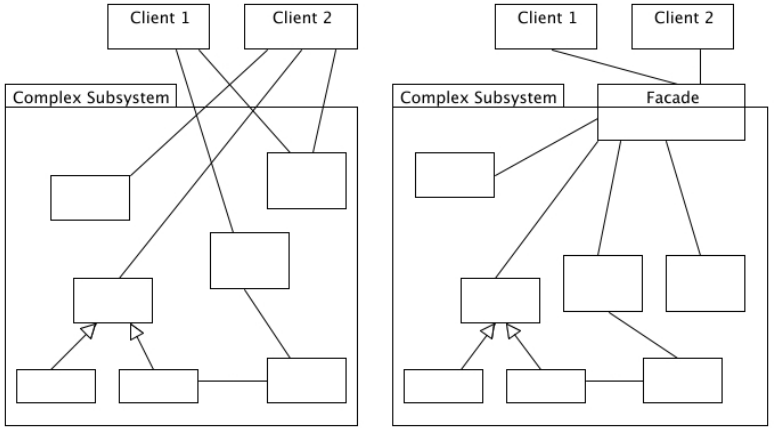


Image: `http://best-practice-software-engineering.ifs.tuwien.ac.at/patterns/facade.html`

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

Structural patterns

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

Behavioral patterns

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