AndyPatterns

From Strategy to Bridge

This is a story of how we get from patterns like the Interface pattern to Factory, Strategy, Proxy, Adapter, and finally to the full blown Bridge Design Pattern.

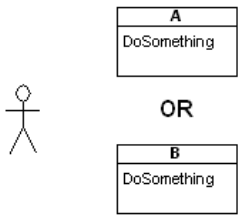
This is an exploration of how to swap implementations of objects within our software architectures.

This article was formerly known as The "Road to the Bridge".

The problem

We want to be flexible in our architecture. We want to be able to swap implementations of objects/classes easily.

- Build to anticipate and celebrate change.
- Program to interfaces, not implementations..



The road we journey on...

Let's examine the range of solutions - showing the story of how to bind to two different implementations of the same interface - simple ways and more complex ways. Specifically how we move from:

1. compile-time binding (one or the other is chosen by compiled code) to
2. factory based binding (one or the other is returned by a factory) to
3. dynamic binding using an intermediary object,

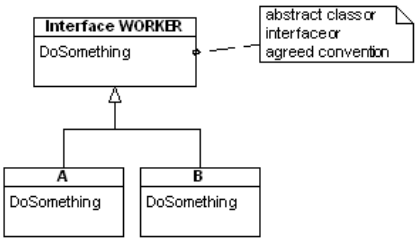
where #3 is achieved using Strategy, Adapter and Proxy, then this ultimately leads us to Bridge.

This journey strikes me as a powerful way of looking at a deep and common problem (building to embrace change), and that also unites multiple patterns under the one theme. Given programmers love the ideal of 'programming to interfaces' and being able to swap in different implementations, this story will show how to do it at many different levels and in fact how many of the design patterns are all about helping us achieve it.

Interface pattern

Interface, compile time choice

Alternative implementations of an interface. Instantiate one or the other implementation of that interface. The code that uses the object is unaware of which object it is using. "Program to an interface"




```
Worker o = new A()
// Worker o = new B()

o.DoSomething()
```

Here the choice is at compile time, by commenting out one or the other instantiation.

Interface, dynamic run time choice

Same solution except choose particular implementation dynamically at runtime using a flag.

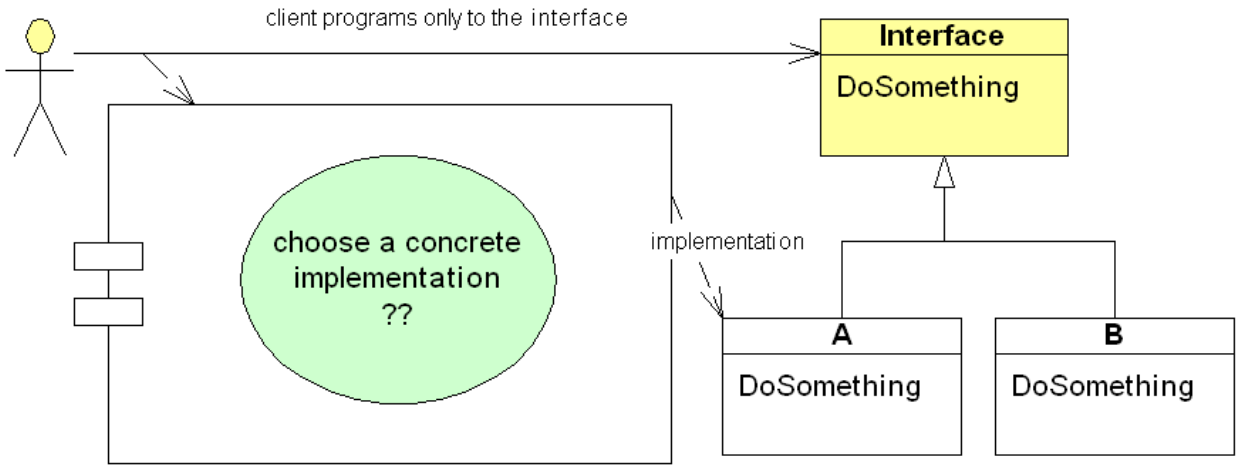
```
if flag
   Worker o = new A()
else
   Worker o = new B()

o.DoSomething()    // we don't know if its an A or a B.  Everything works ok.
```

Factory

Create A or B at runtime by asking another class to create the concrete object for us. Pass in the flag to the factory or let the factory decide for itself which implementation we want.

Factory class is the only class to refer to concrete products. The client refers to the interface/abstract class only.

We are still talking directly to the concrete object (either an A or a B).



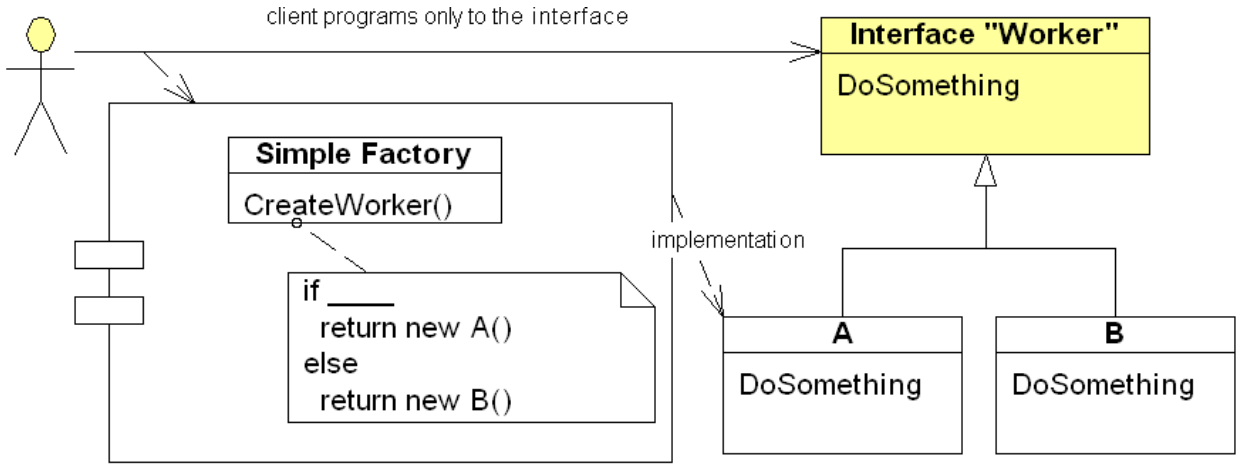
There are a number of factory method variants:

Simple Super Dumb Factory

Encapsulates the "dynamic run time choice" solution discussed in the beginning of this talk. Benefit is that the conditional logic containing the if statement is hidden and possibly centralized in a factory class.

Factory class is the only class that refers directly to concrete products. Client refers only to interface/abstract class.

The choice is made via conditional code.



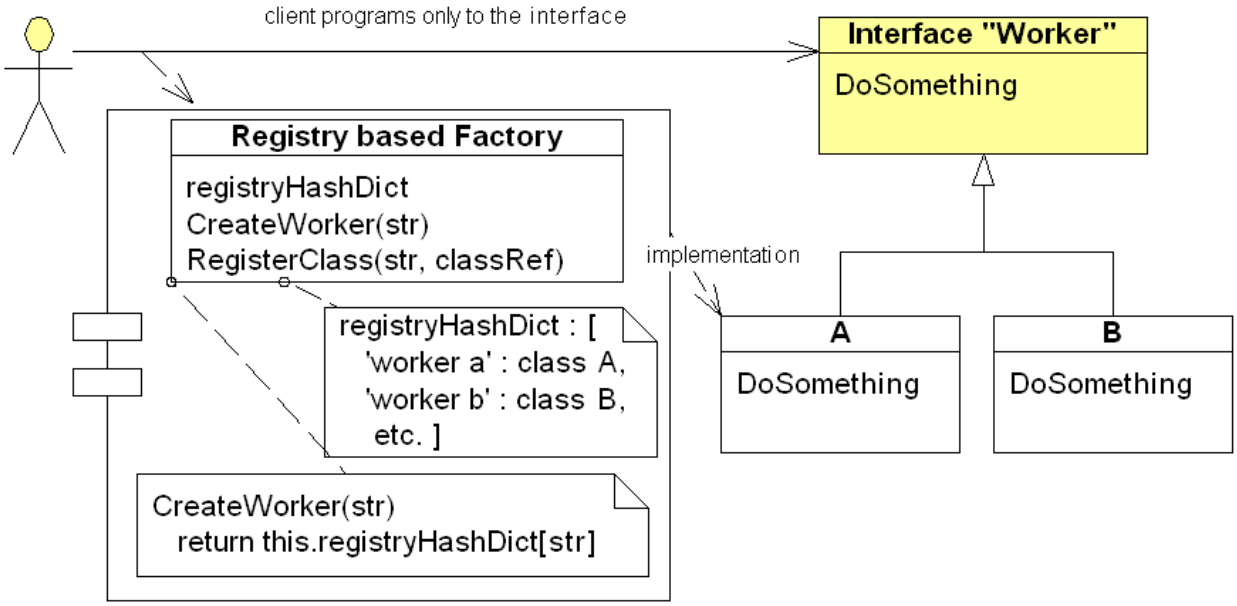

```
Factory f = new SimpleFactory()
Worker o = f.CreateWorker()

o.DoSomething()    // we don't know if its an A or a B.  Everything works ok.
```

Registry Based Factory

Maintains a registry of mappings between strings (or any type of key e.g. objects, class references, numbers etc.) and class references. Benefit: more generalized, no if statements.

Factory class is the only class that refers directly to concrete products. Client refers only to interface/abstract class.

The choice is made via registry key.




```
key = 'worker a'   // in setup code somewhere

Factory f = new RegistryFactory()
Worker o = f.CreateWorker(key)

o.DoSomething()    // we don't know if its an A or a B.  Everything works ok.
```

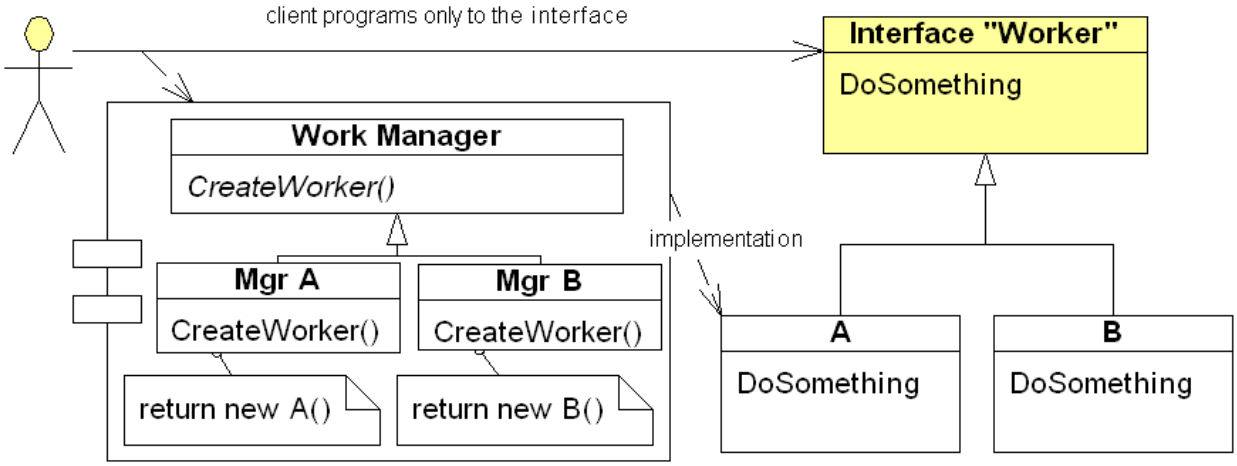
GOF Factory Method

Assumes the client *already has* an instance of some class which needs either a A or B version of a worker class.

Each alternative instance of the existing class overrides a create method differently, each instantiating a different concrete product - typically one matching their own functionality. Benefit: no class reference language facilities required.

Factory class is the only class that refers directly to concrete products. Client refers only to interface/abstract class.

The choice is made via polymorphic override.



Note that the choice as to which Work Manager (MgrA or MgrB) to instantiate in the first place is going to be an issue, but is not the point of this example. The point is that once you have a particular brand of work manager, then you will get a related brand of of worker via the suitably overriden CreateWorker factory method.

```
WorkManager f = new MgrA()      // done somewhere in setup

Worker o = f.CreateWorker(key)

o.DoSomething()    // we don't know if its an A or a B.  Everything works ok.
```

There will be parallel hierarchies, e.g. the WorkManager and the Worker hierarchies closely match, with A and B versions of their subclasses. Start to think of a *family* of classes.

My further thoughts, including a more detailed example of Factory Method here.

Abstract Factory

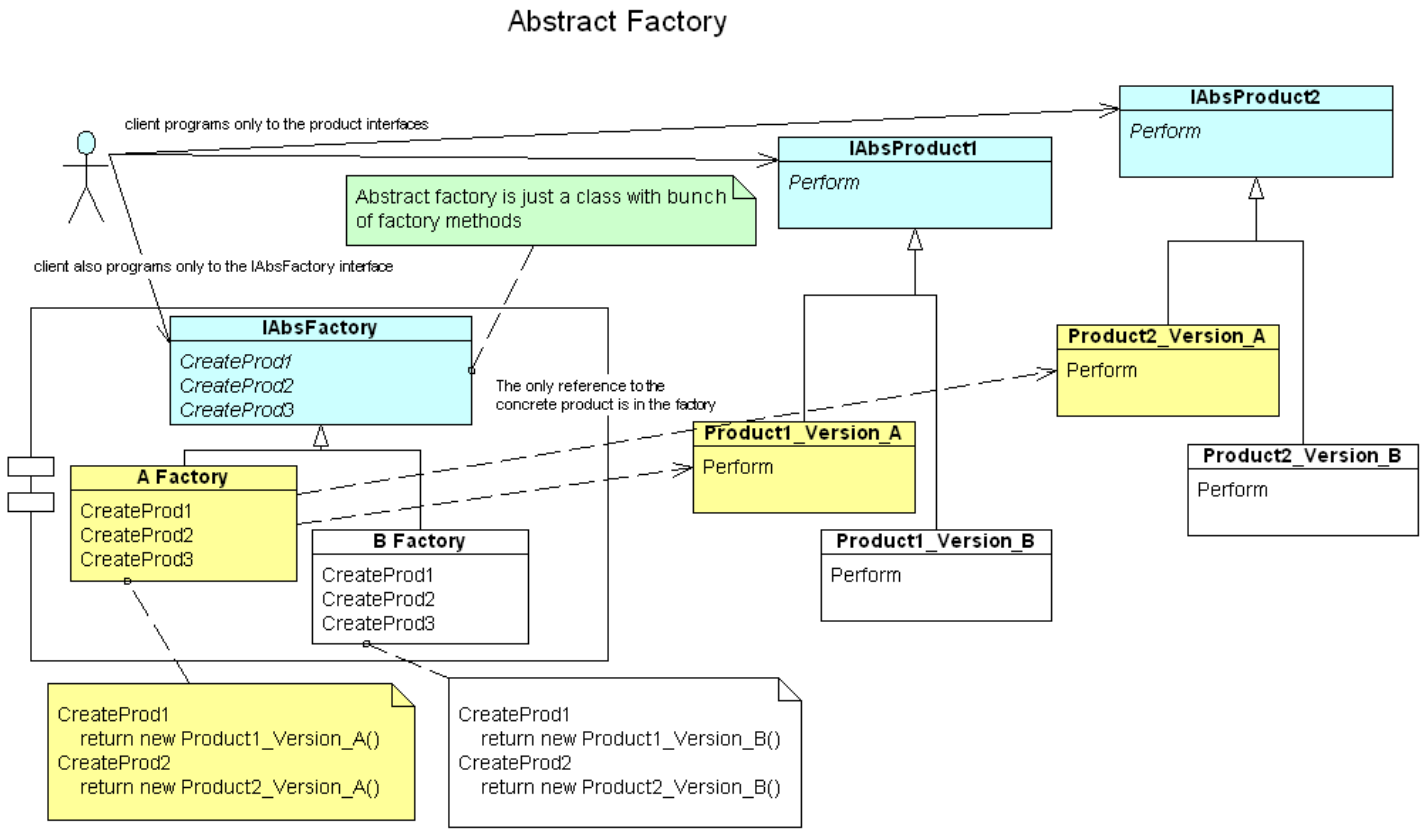
Abstract factory similar to factory method, in that there is something being overridden.

Abstract factory is the same as factory method, except there is more than one Creation method. E.g. CreateWorker, CreateAdministrator, CreatePoliceman - such that the class containing the factory methods might as well become a sole purpose class for dispensing these related classes.

The abstract factory is a mere mechanism for delivering A versions of B versions. E.g. Client wants A version of products

Client programs against interfaces thus can switch between A or B. Specifically, the client only talks to

- IAbstractProductFactory
- IProduct1
- IProduct2
- IProduct3



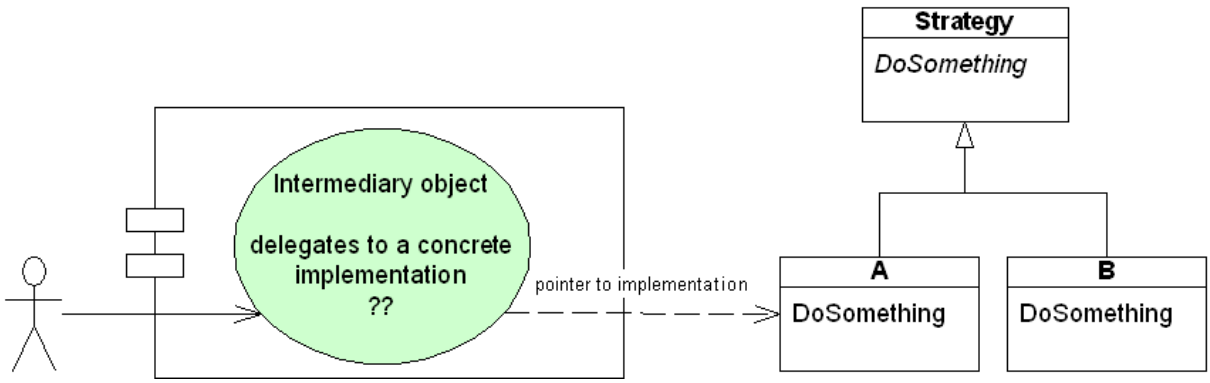

```
IAbstractProductFactory f = new ProductFactoryVersionA()
// choice is made at compile time, via factory method (run time) via strategy (runtime)
IProduct1 p1 = f.CreateProduct1()
IProduct2 p2 = f.CreateProduct2()
IProduct3 p3 = f.CreateProduct3()
```

All products p1, p2, p3 are in the above example A versions, and compatible with each other.

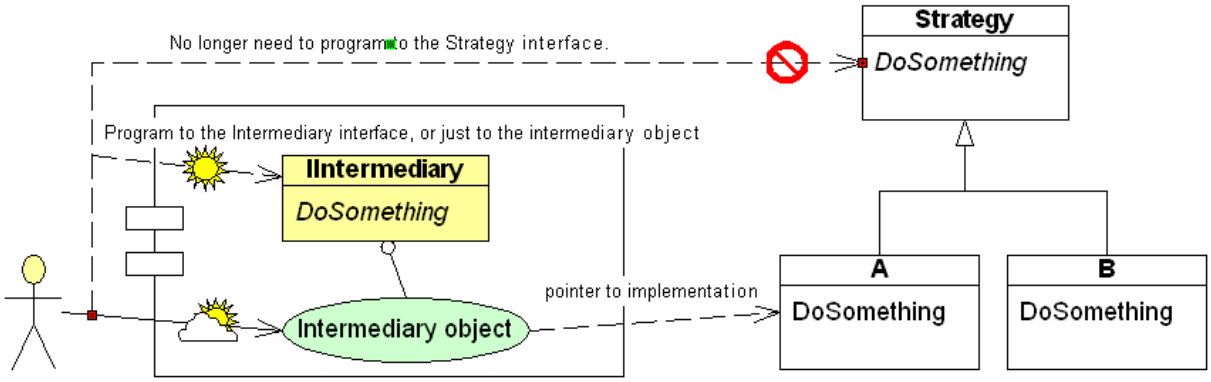
My further thoughts on Abstract Factory here.

Indirection - get to implementation A or B via intermediary

Rather than instantiate A or B and refer to them directly (albiet via a flexible interface variable), another approach is to refer to the same object all the time and hide the switching *behind* that object.



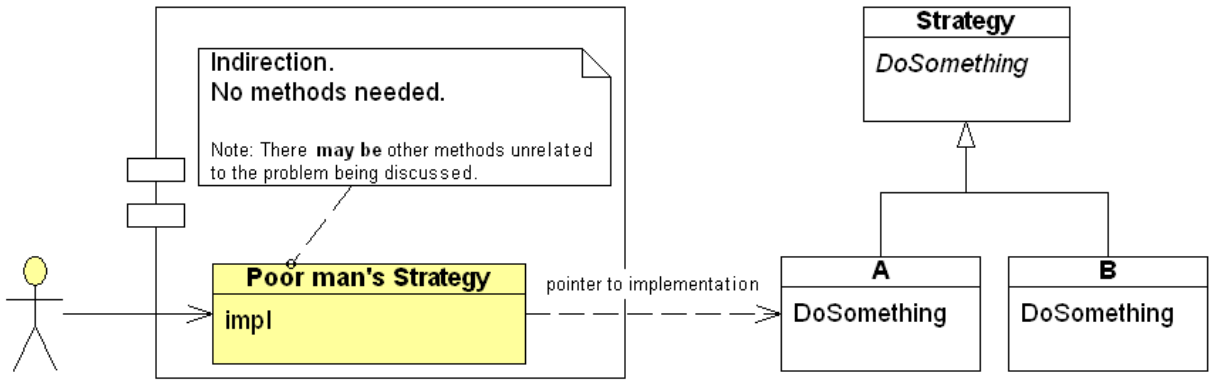
Now, because what is behind the intermediate object is hidden (and rightly so), you no longer need to program to the Strategy interface.



If you want to still program to an interface (good idea) then program to the Intermediary interface. If you want to run free and wild, program to the intermediary object api.

Variants are as follows:

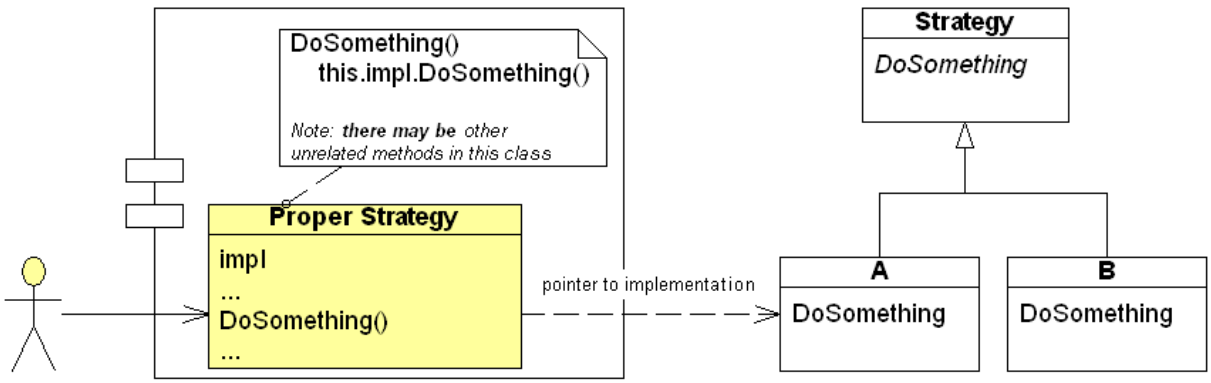
Proxy - methodless indirection using "demeter" referencing



Responsibility of the client to know the API of the strategy. So still programming to the strategy interface. You have to since the intermediary has no methods, or rather, has no methods specifically related to accessing the A & B classes.

```
o = new Intermediary()
o.SetStrategy(new A())   // done in setup somewhere, or via a factory or via dependency injection framework

o.impl.DoSomething()
```

Proper Strategy



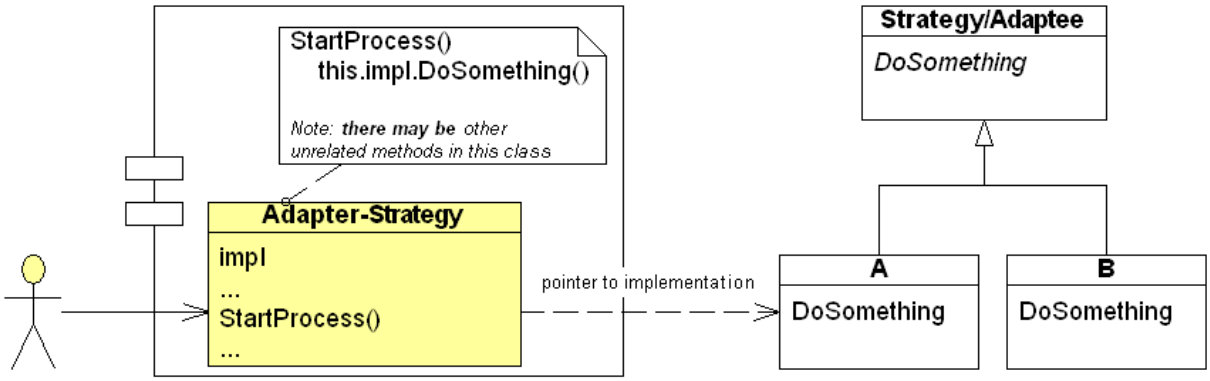

```
o = new Intermediary()
o.SetStrategy(new A())   // done in setup somewhere, or via a factory or via dependency injection framework

o.DoSomething()
```

later you can switch the strategy without the client code caring.

```
o.SetStrategy(new B())
o.DoSomething()        // different behaviour or different implementation occurs
```

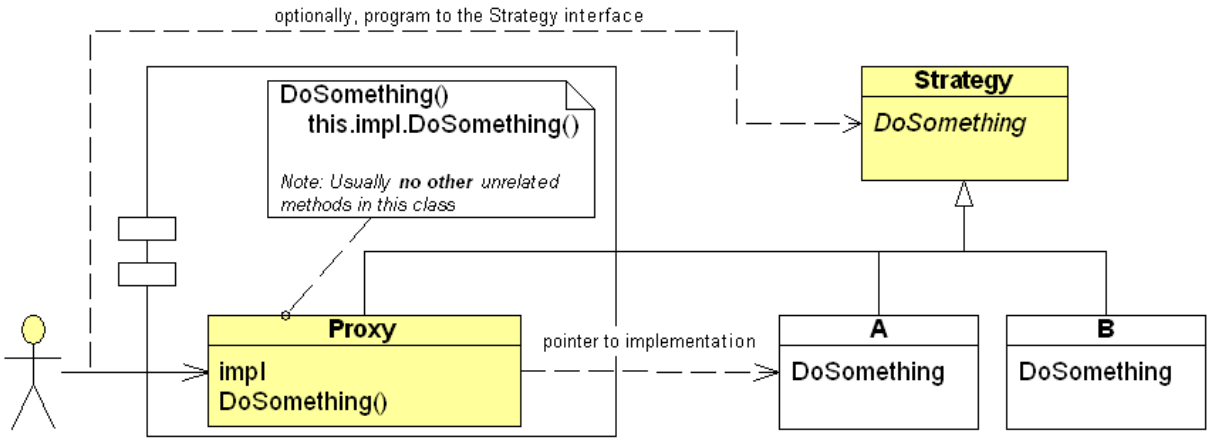
Strategy with a touch of the Adapter pattern

If your implementation has a slightly different API than the one your client code wants to use, then you can adapt it at the same time as you are strategizing...



Proxy - going all the way

If your only have the same methods in your intermediary object as you have in your implementation, then you can have the intermediary inherit from the abstract implementation interface. This turns the pattern into proxy, and lets you optionally, program to the Strategy interface again.



The proxy, whilst *inheriting* from Strategy, can also implement extra methods, though this is diverging a little from the intent of Proxy.
An alternative to inheritance, the proxy can *implement the interface* of the Strategy class, and get some similar polymorphic substitutability benefits.

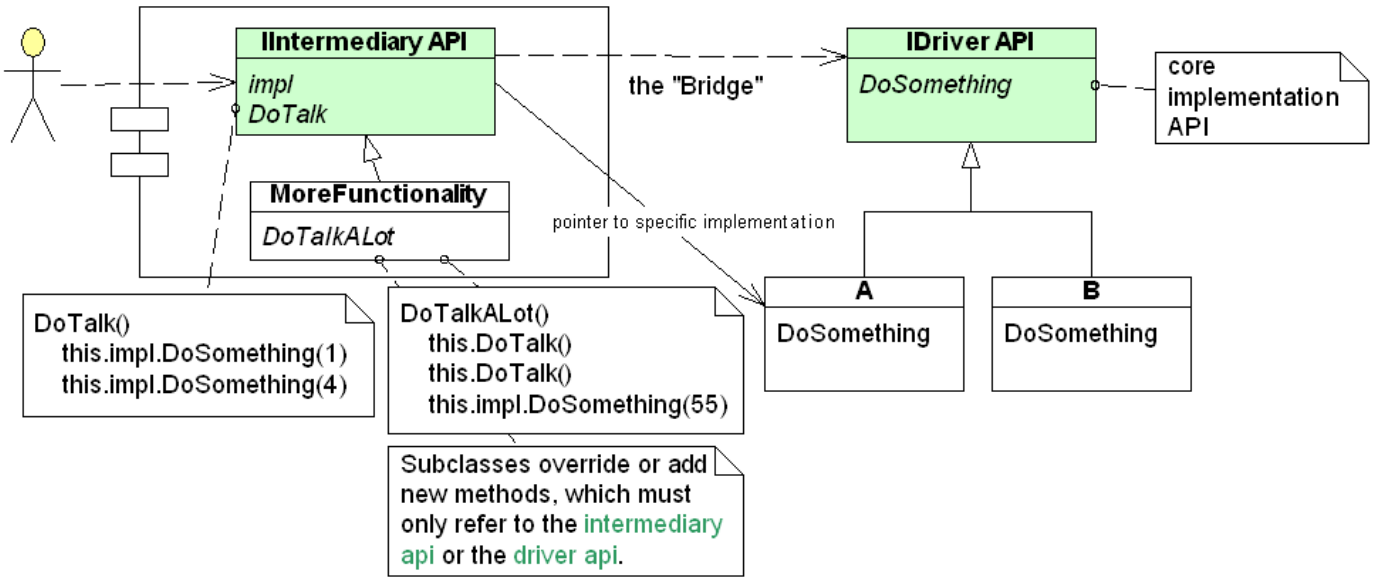
Bridge

This is still a variant on accessing different behaviour via an intermediary.

Bridge is just strategy with a oversized lhs context.

Same as strategy except there is

- Massive subclassing going on on the lhs (the 'context' side).
- The nature of the lhs methods are more compositional, adaptive and far reaching (not just a simply strategy delegation)



Massive subclassing going on in the lhs. context

The reason is that you are wanting lots of methods and lots of functionality, lots of classes. E.g. you want to have a GUI or DB subsystem, not just a single strategy.

The nature of the lhs and rhs methods

Typically rhs (implementation/driver) calls are more primitive, and one lhs method will call the rhs. many times. e.g. see the DoTalk() method, above.

The lhs methods can be diverse, comprising

- lhs method simply calls rhs method. Method names can change or be the same. Simple delegation with no extra work.
- lhs methods more complex and adapt and do extra lines of code as needed
- Lots of logic in the lhs methods and may have associated helper classes. But in the end they call stuff on the intermediary api.

Insulated from change. Allow lhs and rhs to vary independently.

Client is insulated from changes. Should not talk talk to implementation, even if it is the abstract implementation interface because the abs impl. may change. If the abstract implementation interface does change then this affects only the Intermediary but not the client code. Client code should thus only talk to intermediary.

Similarly, if you change the Intermediary API, then only the client is affected - the r.h.s. (the abstract implementation interface and concrete implementations) are not affected.

In this sense the lhs and rhs can vary independently. Ok - so there are repercussions when things vary - but they are limited, as discussed above.

Final thought on Bridge

You could simplify Bridge and have the client code talk directly to the rhs. abstract implementation interface. You would be reverting to where we started on this "road to Bridge". Nothing wrong with that - but you would lose the 'insulation against change' that Bridge gets you. And with Bridge the lhs can have lots of complex logic and the rhs implementations need only implement the more primitive operations. That is a big win.

Solutions overview

Summary of the ways of coupling your components

Technique	Meta-Pattern	Pattern	Description	
To implementation A or B directly	Program to Interface	Interface	Interface, compile time choice	Alternative implementations of an interface. Instantiate one or the other implementation of that interface. The code that uses the object is unaware of which object it is using. "Program to an interface"
			Interface, conditional code	Same solution as above, except choose particular implementation dynamically at runtime using a flag.
	Factory	Factory Method	conditional code	
			registry	
			polymorphic factory method	
		Abstract Factory	abstract factory - polymorphic	
			conditional code	
			class registry	
To implementation A or B via intermediary	Indirection	Dot notation drilling	methodless proxy using demeter referencing	
		Strategy	strategy - may be extra methods not related to the strategising	
		Proxy	proxy, all methods mapped (demeter is happy). inherit	
		Adapter	adapted proxy-like strategy. different method names sometimes	

		Bridge	rhs - methods usually more primitive. Only talk to abs. impl. lhs - all adapted & thus changeable. can build hierarchies	

Final thoughts

The presentation of the patterns form a story of simple to complex.

And its a story of two broadly different techniques,

- Getting to the implementation A or B directly
- Getting to the implementation A or B via an intermediary object

Adapter vs. Bridge

Adapter is closer to Bridge in that the adaptation on the lhs. (the context) can be not just a renaming and mapping of methods, but extra logic and whatever it takes to make the adapation work. So the lhs. is closer to the free wheeling compositional lhs of Bridge Pattern. By compositional I mean that a single lhs. method can comprise of complex code and multiple calls to the rhs. methods. In Bridge the lhs methods can even call on other methods in the same lhs, whereas in Adapter this is not really the intent.

IOC (inverison of control) also fits in here somewhere.

Dependency injection. Inject a context object or wire up dependent objects. Allows you to program normally. Allows different implementations to be injected in.

Microkernels also fit in here.

Amongst other things, a Microkernel style architecture allows alternative plugins (services) to fulfil the implementation.

- Maybe think of it as *service* A or B.
- Or *plugin* A or B.

There seem to be three types of MicroKernel:

1. Service location, like COM where you either ask for a service and get an interface which you use, or you call a service and the late binding binds to an appropriate service at the last minute. **Style of programming:** slightly different - must ask for an interface before using it.
2. Message broadcasting kernel, where messages are broadcast to all plugins and the chain of responsibility pattern is used, and a plugin/service which can make sense of the message acts on it (either consuming it or passing it on for someone else to have a go at). **Style of programming:** different - you must create messages send them into the kernel, either synchronously or asynchronously.
3. Dependency Injection Microkernel, where all object attributes refering to other objects (dependencies) are injected for you by a framework. Rather than setting up these references yourself manually, as normal programming style dictates, you leave it to magic - which allows other implementations to be swapped in. You must of course program to interfaces not to concrete classes, in order for this trick to work. **Style of programming:** normal, you just call methods on objects that you have references to. The fact that the references have been wired up by a framework (which consults a plugin directory & setup file telling us which plugins are active) is hidden from us.

Maybe one or more of the above three descriptions of a Microkernel is not actually a microkernel - I am just learning about this stuff. But I have seen references that suggest my analysis is correct. E.g. The Castle IOC framework for .NET calls itself a microkernel.

A variable of type interface is really a another 'secret' form of indirection

I have made a broad distinction between accessing implementations A or B either directly or via an intermediary. Thinking about it some more, when you do access A or B directly, you do so via an an intermediary variable declared of type abstract/interface. This is when you are being good and 'programming to interfaces'.

Thus you could argue that even even when you are accessing an object (implementation A or B) directly, you are in fact still acting through an intermediary - the interface variable. !

Comments? Please email me.