<http://www.netobjectivestest.com/PatternRepository/index.php?title=AdapterVersusProxyVersusFacadePatternComparison>

# Adapter versus Proxy Pattern Comparison

One of the more frequent questions I get in class is "what's the difference between [Adapter](http://www.netobjectivestest.com/PatternRepository/index.php?title=TheAdapterPattern) and [Proxy](http://www.netobjectivestest.com/PatternRepository/index.php?title=TheProxyPattern)? They really seem the same to me". This is mostly because the runtime relationships are awfully similar:

[](http://www.netobjectivestest.com/PatternRepository/index.php?title=Image:AdapterProxyFacade.jpg)

This is a good example of the critical notion that patterns are not diagrams, or code snippets.

## Proxy vs. Adapter

* The [Proxy](http://www.netobjectivestest.com/PatternRepository/index.php?title=TheProxyPattern) changes the behavior of the Service, but preserves its interface.
* The [Adapter](http://www.netobjectivestest.com/PatternRepository/index.php?title=TheAdapterPattern) changes the interface of the Service, but preserves it behavior.
* A Client can use the [Proxy](http://www.netobjectivestest.com/PatternRepository/index.php?title=TheProxyPattern) or the Service Entity in the same way.
* A Client designed to use the [Adapter](http://www.netobjectivestest.com/PatternRepository/index.php?title=TheAdapterPattern) would not be able to use the Service Entity without it.
* The [Proxy](http://www.netobjectivestest.com/PatternRepository/index.php?title=TheProxyPattern) can be cast to the interface of the Service. The [Adapter](http://www.netobjectivestest.com/PatternRepository/index.php?title=TheAdapterPattern) can be cast to the interface the Client expects.

**Composition**

public class A {

private B b = new B();

public A() {

}

}

Once there are no more references to a particular instance of class A, its instance of class B is destroyed.

Rationale. Allows classes to define behaviors and attributes in a modular fashion.

Further Study. <http://www.artima.com/designtechniques/compoinh.html>

**Delegation**

public class A {

private B b = new B();

public void method() {

b.method();

}

}

When clients of A call method, class A delegates the method call to B.

Rationale. Class A can inherit from one class, but expose behaviours that belong elsewhere.

Further Study. <http://beust.com/java-delegation.html>

**Aggregation**

public class A {

private B b;

public A( B b ) {

this.b = b;

}

}

public class C {

private B b = new B();

public C() {

A a = new A( this.b );

}

}

Once there are no more references to a particular instance of class A, its instance of class B will not be destroyed. In this example, both A and C must be garbage collected before B will be destroyed.

Rationale. Allows instances to reuse objects.

Further Study. <http://faq.javaranch.com/java/AssociationVsAggregationVsComposition>

**Choosing between composition and inheritance**

1. **Make sure inheritance models the *is-a* relationship**  
   My main guiding philosophy is that inheritance should be used only when a subclass *is-a* superclass. In the example above, an Apple likely is-a Fruit, so I would be inclined to use inheritance.
2. **Don't use inheritance just to get code reuse**  
   If all you want is to reuse code and there is no is-a relationship in sight, use composition.
3. **Don't use inheritance just to get at polymorphism**  
   If all you want is polymorphism, but there is no natural is-a relationship, use composition with interfaces.