<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)? 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.

**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**  
   Inheritance should be used only when a subclass *is-a* superclass. For example, 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.

<http://en.wikipedia.org/wiki/Class_diagram>

**Class diagram**

A **class diagram** in the [Unified Modeling Language](http://en.wikipedia.org/wiki/Unified_Modeling_Language) (UML) is a type of static structure diagram that describes the structure of a system by showing the system's [classes](http://en.wikipedia.org/wiki/Class_%28computer_science%29), their attributes, operations (or methods), and the relationships among the classes.

## Overview

The class diagram is the main building block of [object oriented](http://en.wikipedia.org/wiki/Object_oriented) modeling. It is used both for general [conceptual modeling](http://en.wikipedia.org/wiki/Conceptual_model) of the systematics of the application, and for detailed modeling translating the models into [programming code](http://en.wikipedia.org/wiki/Programming_code). The classes in a class diagram represent both the main objects, interactions in the application and the classes to be programmed.

[](http://en.wikipedia.org/wiki/File:BankAccount1.svg)

A class with three sections.

In the diagram, classes are represented with boxes which contain three parts:

* The upper part holds the name of the class.
* The middle part contains the attributes of the class.
* The bottom part gives the methods or operations the class can take or undertake.

In the design of a system, a number of classes are identified and grouped together in a class diagram which helps to determine the static relations between those objects.

## Members

UML provides mechanisms to represent class members, such as attributes and methods, and additional information about them.

### Visibility

To specify the visibility of a class member (i.e., any attribute or method) these are the following notations that must be placed before the member's name:

"+" Public

"-" Private

"#" Protected

"~" Package

"/" Derived

"\_" Static

### Scope

The UML specifies two types of scope for members: *instance* and *classifier*.

* **Classifier members** are commonly recognized as “static” in many programming languages. The scope is the class itself.
  + Attribute values are equal for all instances.
  + Method invocation does not affect the instance’s state.
* **Instance members** are scoped to a specific instance.
  + Attribute values may vary between instances.
  + Method invocation may affect the instance’s state (i.e., change instance’s attributes).

To indicate a classifier scope for a member, its name must be underlined. Otherwise, instance scope is assumed by default.

## Relationships

A relationship is a general term covering the specific types of logical connections found on class and object diagrams. UML shows the following relationships:

### Instance level relationships

### External links

A *Link* is the basic relationship among objects.

#### Association

[http://upload.wikimedia.org/wikipedia/commons/thumb/4/4d/UML_role_example.gif/400px-UML_role_example.gif](http://en.wikipedia.org/wiki/File:UML_role_example.gif)

Class diagram example of association between two classes

An [*association*](http://en.wikipedia.org/wiki/Association_%28object-oriented_programming%29) represents a family of links. Binary associations (with two ends) are normally represented as a line. An association can be named, and the ends of an association can be adorned with role names, ownership indicators, multiplicity, visibility, and other properties.  
There are four types of association: bi-directional, uni-directional, Aggregation (includes composition aggregation) and Reflexive. Bi-directional and uni-directional associations are the most common ones. For instance, a flight class is associated with a plane class bi-directionally. Association represents the static relationship shared among the objects of two classes. Example: "department offers courses", is an association relation.

##### Aggregation

[http://upload.wikimedia.org/wikipedia/commons/thumb/2/2a/KP-UML-Aggregation-20060420.svg/300px-KP-UML-Aggregation-20060420.svg.png](http://en.wikipedia.org/wiki/File:KP-UML-Aggregation-20060420.svg)

Class diagram showing Aggregation between two classes

[*Aggregation*](http://en.wikipedia.org/wiki/Aggregation_%28object-oriented_programming%29) is a variant of the "has a" or association relationship; aggregation is more specific than association. It is an association that represents a part-whole or part-of relationship. As a type of association, an aggregation can be named and have the same adornments that an association can. However, an aggregation may not involve more than two classes. *Aggregation* can occur when a class is a collection or container of other classes, but where the contained classes do not have a strong *life cycle dependency* on the container—essentially, if the container is destroyed, its contents are not.

In [UML](http://en.wikipedia.org/wiki/Unified_Modeling_Language), it is graphically represented as a *hollow* [diamond shape](http://en.wikipedia.org/wiki/Rhombus) on the containing class end of the tree with a single line that connects the contained class to the containing class. The aggregate is semantically an extended object that is treated as a unit in many operations, although physically it is made of several lesser objects.

##### Composition

[](http://en.wikipedia.org/wiki/File:AggregationAndComposition.svg)

Class diagram showing Composition between two classes at top and Aggregation between two classes at bottom.

[*Composition*](http://en.wikipedia.org/wiki/Object_composition) is a stronger variant of the "owns a" or association relationship; composition is more specific than aggregation. *Composition* usually has a strong *life cycle dependency* between instances of the container class and instances of the contained class(es): If the container is destroyed, normally every instance that it contains is destroyed as well. (Note that, where allowed, a part can be removed from a composite before the composite is deleted, and thus not be deleted as part of the composite.) The UML graphical representation of a composition relationship is a *filled* diamond shape on the containing class end of the tree of lines that connect contained class(es) to the containing class.

##### Differences between composition and aggregation

When attempting to represent real-world whole-part relationships, e.g., an engine is a part of a car, the composition relationship is most appropriate. However, when representing a software or database relationship, e.g., car model engine ENG01 is part of a car model CM01, an aggregation relationship is best, as the engine, ENG01 may be also part of a different car model. Thus the aggregation relationship is often called "catalog" containment to distinguish it from composition's "physical" containment.

The whole of a composition must have a multiplicity of 0..1 or 1, indicating that a part must belong to only one whole; the part may have any multiplicity. For example, consider University and Department classes. A department belongs to only one university, so University has multiplicity 1 in the relationship. A university can (and will likely) have multiple departments, so Department has multiplicity 1..\*.

### Class level relationships

#### Generalization

[](http://en.wikipedia.org/wiki/File:KP-UML-Generalization-20060325.svg)

The Generalization relationship ("is a") indicates that one of the two related classes (the *subclass*) is considered to be a specialized form of the other (the *super type*) and superclass is considered as '***Generalization'*** of subclass. In practice, this means that any instance of the subtype is also an instance of the superclass. The relationship is easily understood by the phrase 'an A is a B'.

The UML graphical representation of a Generalization is a hollow [triangle](http://en.wikipedia.org/wiki/Triangle) shape on the superclass end of the line (or tree of lines) that connects it to one or more subtypes. The generalization relationship is also known as the [*inheritance*](http://en.wikipedia.org/wiki/Inheritance_%28computer_science%29) or *"is a"* relationship. The [*superclass*](http://en.wikipedia.org/wiki/Superclass) (base class) in the generalization relationship is also known as the *"parent"*, *superclass*, *base class*, or *base type*. The [*subtype*](http://en.wikipedia.org/wiki/Subtype) in the specialization relationship is also known as the *"child"*, *subclass*, *derived class*, *derived type*, *inheriting class*, or *inheriting type*.

#### Realization

In UML modeling, a realization relationship is a relationship between two model elements, in which one model element (the client) realizes (implements or executes) the behavior that the other model element (the supplier) specifies. The UML graphical representation of a Realization is a hollow triangle shape on the interface end of the *dashed* line (or tree of lines) that connects it to one or more implementors. A plain arrow head is used on the interface end of the dashed line that connects it to its users. In component diagrams, the ball-and-socket graphic convention is used (implementors expose a ball or lollipop, while users show a socket). Realizations can only be shown on class or component diagrams. A realization is a relationship between classes, interfaces, components, and packages that connects a client element with a supplier element. A realization relationship between classes and interfaces and between components and interfaces shows that the class realizes the operations offered by the interface.

### General relationship

[](http://en.wikipedia.org/wiki/File:Class_Dependency.png)

Class diagram showing dependency between "Car" class and "Wheel" class (An even clearer example would be "Car depends on Wheel", because Car already *aggregates* (and not just *uses*) Wheel).

#### Dependency

[Dependency](http://en.wikipedia.org/wiki/Dependency_%28UML%29) is a weaker form of relationship which indicates that one class depends on another because it uses it at some point in time. One class depends on another if the independent class is a parameter variable or local variable of a method of the dependent class. This is different from an association, where an attribute of the dependent class is an instance of the independent class.

### Multiplicity

The association relationship indicates that (at least) one of the two related classes makes reference to the other. In contrast with the generalization relationship, this is most easily understood through the phrase 'A has a B'.

The UML representation of an association is a line with an optional arrowhead indicating the *role* of the object(s) in the relationship, and an optional notation at each end indicating the *multiplicity* of instances of that entity (the number of objects that participate in the association).

|  |  |
| --- | --- |
| **0..1** | No instances, or one instance (optional, may) |
| **1** | Exactly one instance |
| **0..\*** or **\*** | Zero or more instances |
| **1..\*** | One or more instances (at least one) |