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

# Adapter versus Proxy Pattern Comparison

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)? The runtime relationships are awfully similar:

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

## 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.

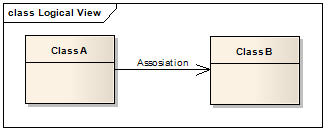
<http://aviadezra.blogspot.tw/2009/05/uml-association-aggregation-composition.html>

### [UML Class Diagram: Association, Aggregation and Composition](http://aviadezra.blogspot.tw/2009/05/uml-association-aggregation-composition.html)

There are three primary inter-object relationships: *association*, *aggregation*, and composition.

#### **Association**

The most abstract way to describe static relationship between classes is using the ‘Association’ link, which simply states that there is some kind of a link or a dependency between two classes or more.

[](http://lh5.ggpht.com/_aUOgqE3fGXc/Sh32OSBGAdI/AAAAAAAAAZ4/HYw8B9VNlMs/s1600-h/image%5B4%5D.png)

**Weak Association**

ClassA may be linked to ClassB to show that one of its methods includes parameter of ClassB instance, or returns instance of ClassB.

[](http://lh3.ggpht.com/_aUOgqE3fGXc/Sh32SJJ1K6I/AAAAAAAAAaA/unIYlYJmUoI/s1600-h/image%5B9%5D.png)

**Strong Association**

ClassA may also be linked to ClassB in order to show that it holds reference to ClassB instance.

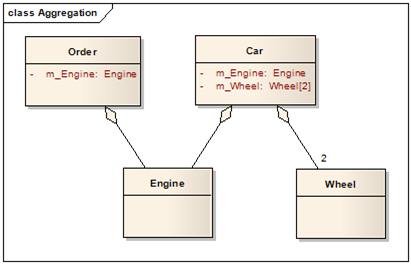
[](http://lh5.ggpht.com/_aUOgqE3fGXc/Sh32Wod9dyI/AAAAAAAAAaI/cWKKrsxaQ-8/s1600-h/image%5B14%5D.png)

#### Aggregation (Shared Association)

In cases where there’s a part-of relationship between ClassA (whole) and ClassB (part), we can be more specific and use the aggregation link instead of the association link, taking special notice that ClassB can also be aggregated by other classes in the application (therefore aggregation is also known as shared association).

[](http://lh6.ggpht.com/_aUOgqE3fGXc/Sh32ZWzvg3I/AAAAAAAAAaQ/mtsE0R03LwA/s1600-h/image%5B34%5D.png)

So basically, the aggregation link **doesn’t state** in any way that ClassA owns ClassB **nor** that there is a parent-child relationship (when parent deleted all its child’s are being deleted as a result) between the two. Actually, quite the opposite! The aggregation link usually used to stress the point that ClassA is not the exclusive container of ClassB, as in fact ClassB has another container.

[](http://lh4.ggpht.com/_aUOgqE3fGXc/Sh32cQ4pjVI/AAAAAAAAAaY/2lq7mZumWM8/s1600-h/image%5B24%5D.png)

##### Aggregation v.s. Association

The association link can replace the aggregation link in every situation, while aggregation cannot replace association in situations were there is only a ‘weak link’ between the classes, i.e. ClassA has method/s that contain parameter of ClassB but ClassA doesn’t hold reference to ClassB instance.

#### **Composition** (Not-Shared Association)

In cases where in addition to the part-of relationship between ClassA and ClassB - there’s a strong life cycle dependency between the two, meaning that when ClassA is deleted then ClassB is also deleted as a result, we should be more specific and use the composition link instead of the aggregation link or the association link.

[](http://lh5.ggpht.com/_aUOgqE3fGXc/Sh35YNKDw9I/AAAAAAAAAao/E4v4uDJcD5w/s1600-h/image%5B5%5D.png)

The composition link shows that a class (container, whole) has exclusive ownership over other class/s (parts), meaning that the container object and its parts constitute a parent-child/s relationship. Unlike association and aggregation, in the composition relationship, the composed class cannot appear as a return type or parameter type of the composite class, thus changes in the composed class cannot be propagated to the rest of the system. Consequently, usage of composition limits complexity growth as the system grows.

**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.

The **delegation pattern** is a [design pattern](http://en.wikipedia.org/wiki/Design_pattern) where an [object](http://en.wikipedia.org/wiki/Object_(computer_science)), instead of performing one of its stated tasks, delegates that task to an associated helper object. In this example, the Printer [class](http://en.wikipedia.org/wiki/Class_(computer_science)) has a print method. This print method, rather than performing the print itself, delegates to class RealPrinter. Here is a simple example:

***class*** *RealPrinter { // the "delegate"*

***void*** *print() { System.out.println("something"); }*

*}*

***class*** *Printer { // the "delegator"*

*RealPrinter p =* ***new*** *RealPrinter(); // create the delegate*

***void*** *print() { p.print(); // delegation }*

*}*

***public******class*** *Main {*

***public******static******void*** *main(String[] args) {*

*Printer printer =* ***new*** *Printer();*

*printer.print();// to the outside world it looks like Printer actually prints.*

*}*

*}*

**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 describing the structure of a system by showing the system's [classes](http://en.wikipedia.org/wiki/Class_%28computer_science%29), their attributes, 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 (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”. 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)

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)

[*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. 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.

##### 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* 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, every instance that it contains is destroyed. 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.

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") can be 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

Realization 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 that connects it to one or more implementors.

### General relationship

#### **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.

[](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).

### Multiplicity

The association relationship indicates (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) |

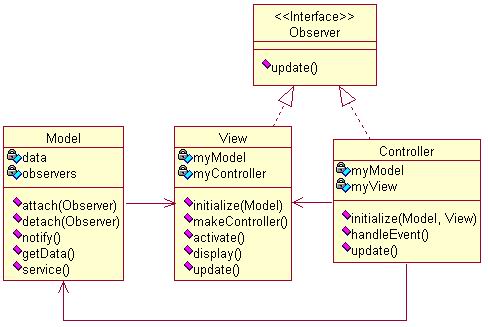
**經典的設計模式MVC**

源於Smalltalk，並在Java中廣泛應用的模型-視圖-控制器(Model-View-Controller，MVC)模式，是非常經典的一個設計模式，通過它你可以更好地理解"模式"這一概念。

MVC模式常用在開發人機交互軟件，這類軟件的最大特點是用戶界面容易改變，系統設計師必須考慮下面兩個因素：1. 用戶界面易於改變，甚至在運行期間也是有可能改變的；2. 用戶界面的修改或移植不會影響軟件的核心功能代碼。

在MVC設計模式中，模型封裝核心數據和功能，它獨立於特定的輸出表示和輸入行為，是執行某些任務的代碼。模型只有純粹的功能性接口，讓系統其它部分可得到模型的內部狀態。視圖用來向用戶顯示信息，它獲得來自模型的數據，決定模型以什麼樣的方式展示給用戶。同一個模型可以對應於多個視圖。控制器是和視圖聯合使用的，它捕捉鼠標移動、點擊和鍵盤輸入等事件，將其轉化成服務請求，然後再傳給模型或者視圖。整個軟件的用戶是通過控制器來與系統交互的，他通過控制器來操縱模型，從而向模型傳遞數據，改變模型的狀態，並最後導致視圖的更新。

應用了MVC設計模式的軟件系統，其基本的實現過程是：1. 控制器創建模型；2. 控制器創建一個或多個視圖，並將它們與模型相關聯；3. 控制器負責改變模型的狀態；4. 當模型的狀態發生改變時，模型會通知與之相關的視圖進行更新。

如果用UML來表示MVC設計模式，則如圖1所示：  


**在Python中應用設計模式**

《設計模式》總結了23個模式，分別從對象的創建，對象和對象間的結構關係以及對象之間如何交互這三個方面入手。

**創建型模式**(creational pattern)**:** 怎樣創建一個對象，以及如何隱藏對象創建的細節，使得程序代碼不依賴於具體的對象，在增加新的對象時，對代碼的改動非常小。**結構型模式**(structural pattern)**:** 類和對象間如何進行有效的組織，形成良好的軟件體系結構，主要的方法是使用繼承關係來組織各個類。**行為型模式**(behavioral pattern)**:** 類或對象間如何交互及如何分配職責，它牽涉的不僅是類或對象的設計模式，還有它們間的通訊模式。

**一、創建型模式**<http://www.ibm.com/developerworks/cn/linux/l-pypt/part2/index.html>

創建型模式(Creational Pattern)對類的實例化過程進行了抽象，能夠使軟件模塊做到與對象創建和組織的無關性。為了使體系結構更加清晰，一些軟件在設計上要求當創建類的具體實例時，能夠根據具體的語境來動態地決定怎樣創建對象，創建哪些對象，以及怎樣組織和表示這些對象，而創建型模式所要描述的就是該如何來解決這些問題。

按照生成目標的不同，創建型模式可以分為類的創建型模式和對象的創建型模式兩種：

**類的創建型模式**   
類的創建型模式通過使用繼承關係，將類的創建交由具體的子類來完成，這樣就向外界隱藏瞭如何得到具體類的實現細節，以及這些類的實例是如何被創建和組織在一起的。

**對象的創建型模式**   
對象的創建型模式通過把對象的創建委託給另一個對象來完成，可以根據語境動態地決定生成哪些具體類的實例，同時還可以向外界隱藏這些實例是如何被創建以及如何被組織在一起的細節。

所有的創建型模式都有兩個永恆的主旋律：第一，它們都將系統使用哪些具體類的信息封裝起來；第二，它們隱藏了這些類的實例是如何被創建和組織的。外界對於這些對像只知道它們共同的接口，而不清楚其具體的實現細節。具體到Python來說，假設有這樣一個類：

|  |
| --- |
| *class Person:*  *def \_\_init\_\_(self, name):*  *self.name = name* |

要創建該類的一個實例，則應該執行下面的語句：

|  |
| --- |
| p = Person("Gary") |

但如果創建對象時完成的工作非常複雜，需要一段很長的代碼​​，你就不能簡單地將其全部寫入\_\_init\_\_方法中，因為這會違背面向對象思想的兩個基本原則：封裝(encapsulation)和委派(delegation)。假如執意要做樣做，結果只會使你的代碼變成一段行為固定的硬編碼（hard coding），而整個軟件的結構都極有可能變得非常糟糕，因為其它某個模塊也許就正依賴於你所創建的這個實例，這樣就在無形之間增加了模塊之間的耦合度。

將Python對象的創建過程封裝到某個類中來單獨完成，可以使你的程序變得更加靈活和通用。實踐證明，使用下面的六種創建型模式可以更好地改善對象的創建過程：

**Simple Factory模式** : 專門定義一個類來負責創建其它類的實例，被創建的實例通常都具有共同的父類。

**Factory Method模式** : 將對象的創建交由父類中定義的一個標準方法來完成，而不是其構造函數，究竟應該創建何種對象由具體的子類負責決定。

**Abstract Factory模式** : 提供一個共同的接口來創建相互關聯的多個對象。

**Singleton模式** : 保證系統只會產生該類的一個實例，同時還負責向外界提供訪問該實例的標準方法。

**Builder模式** : 將復雜對象的創建同它們的具體表現形式(representation)區別開來，這樣可以根據需要得到具有不同表現形式的對象。

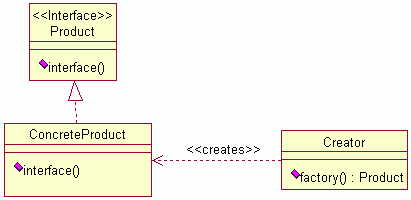
**Prototype模式** : 利用一個能對自身進行複制的類，使得對象的動態創建變得更加容易。

**二、模式引入**

簡單工廠(Simple Factory)模式又稱為靜態工廠方法(Static Factory Method)模式，屬於類的創建型模式。工廠對象能夠處理的所有類通常都繼承於同一個父類，並且對外界提供基本相同的接口，只不過在具體實現時會有所差別罷了。

**三、一般結構**

簡單工廠模式屬於類的創建型模式，適合用來對大量具有共同接口的類進行實例化，它可以推遲到運行的時候才動態決定要創建哪個類的實例，而不是在編譯時就必須知道要實例化哪個類。簡單工廠模式的一般性結構如圖3所示：

 圖3

簡單工廠模式的實質是由一個工廠類根據傳入的參量，動態決定應該創建出哪一個產品類的實例。從上圖可以看出，簡單工廠模式涉及到工廠角色、抽象產品角色和具體產品角色三個參與者。**工廠（Creator）角色** : 是簡單工廠模式的核心，它負責實現創建所有實例的內部邏輯。工廠類可以被外界直接調用，創建所需的產品對象。**抽象產品（Product）角色** : 是簡單工廠模式所創建的所有對象的父類，它負責描述所有實例所共有的公共接口。**具體產品（Concrete Product）角色** : 是簡單工廠模式的創建目標，所有創建的對像都是充當這個角色的某個具體類的實例。

工廠角色負責創建一個具體產品的實例，並將其返回給調用者，以下是工廠類的示例性Python代碼：

|  |
| --- |
| *代碼清單6：creator.py*  *class Creator:*  *def factory(self): # 創建具體產品類的方法*  *return ConcreteProduct()* |

抽象產品角色的主要目的是為所有的具體產品提供一個共同的接口，通常只需給出相應的聲明就可以了，而不用給出具體的實現。以下是抽象產品類的示例性Python代碼：

|  |
| --- |
| *代碼清單7：product.py*  *class Product:*  *def interface(self): # 所有具體產品類的公共接口*  *pass* |

具體產品角色充當最終的創建目標，一般來講它是抽象產品類的子類，實現了抽象產品類中定義的所有接口方法。以下是具體產品類的示例性Python代碼：

|  |
| --- |
| *代碼清單8：concrete.py*  *class ConcreteProduct(Product):*  *def interface(self): # 公共接口的實現*  *print "Concrete Product Method"* |

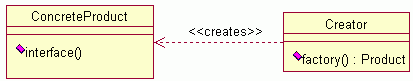
在應用簡單工廠模式時，可以採用下面的示例性Python代碼：

|  |
| --- |
| *fac = Creator()*  *p = fac.factory()*  *if p != None:*  *p.interface()* |

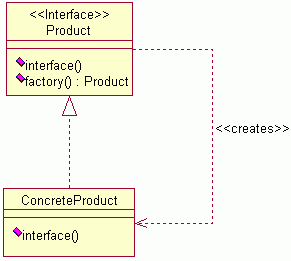
就樣就成功地將類是如何創建的這一實現細節向外界隱藏起來了，這就是簡單工廠模式所採取的基本策略。在這個簡單的示意性實現裡，充當具體產品角色的類只有一個，但在真正的實際應用中，通常遇到的都是同時有多個具體產品類的情況。

**四、實際運用**

如果在實際中運用簡單工廠模式時遇到只有一個具體產品類的情況，為了簡化軟件體系結構，可以考慮對簡單​​工廠模式進行精簡，省略抽象產品這一角色。此時簡單工廠模式的結構將如圖6所示。

 圖6

在簡單廠模式中，充當工廠角色的類通常只會有一個方法，那就是用來創建具體產品對象的factory()。在某些特殊場合下，如果需要的話也可以考慮將工廠角色的功能交由抽象產品角色來替代完成，這時簡單工廠模式的結構將如圖7所示。

 圖7

**六、小結**

創建型模式的目標是做到對象創建和組織的無關性，它可以細分為類的創建型模式和對象的創建型模式。簡單工廠模式屬於類的創建型模式，它可以根據外界所給定的信息，用工廠對象製造出某種特定產品類的實例。完整的簡單工廠模式包含工廠、抽象產品和具體產品三個角色，但在實際運用時可以靈活掌握，對模型進行簡單化。簡單工廠模式的優點是它的工廠類能夠負責所有對象的創建，而缺點則是工廠類的實現邏輯可能過於復雜。

==========================================================================

***Introduction***

*Greedy algorithms* make decisions that look best at the moment. They make decisions that are locally optimal in the hope that they will lead to globally optimal solutions. Therefore, greedy algorithms do not always produce optimal results. Huffman tree is an example of greedy algorithm. To build a Huffman tree, we proceed from its leaf nodes upward, and merge the two trees whose root nodes have the smallest frequencies and store the sum of the frequencies in the new tree’s root. Huffman coding is greedy because it continually seeks out the two trees that appear to be the best to merge at any given time.

***Pointer Manipulation***

*Automatic variables* are those for which storage is allocated and de-allocated automatically when entering and leaving a block or function. For example,is set to the address of the automatic variablein the following function, so is a dangling pointer whenreturns.

*void f( int\*\* iptr ){*

*int a = 10;*

*\*iptr = &a;*

*}*

In C, when we dynamically allocate storage, we get a pointer to some storage on the heap. The storage allocated by *malloc* in the flowing code remains valid until we call free at some later time. The misuse of dynamically allocated storage is a source of memory leaks.

*int f( int\*\* iptr ){*

*if ( ( \*iptr = ( int\* ) malloc( sizeof( int ) ) ) == NULL ) return -1;*

*return 0;*

*}*

To understand the relationship between arrays and pointers in C, recall that to access the *i*th in an array *a*, we use the expression: *a[ i ]* which is equivalent to *\*(a+i)*. It is evaluated using the rules of *pointer arithmetic*. When we add an integer *i* to a pointer, *the result is the address plus i times the number of byes in the data type the pointer references*. This explains why arrays are zero-indexed in C; that is, the first element in an array is at position 0. Multiple dimensional arrays are stored in row-major order. This means that subscripts to the right vary more rapidly than those to the left. In two dimension, the expression *a[ i ][ j ] = \*(\*(a+i)+j)*.

Function pointers have a type that is described in terms of a return value and parameters that the function accepts. Declarations for function pointers look much like declarations for functions except that an asterisk appears before the function name and the asterisk and name are surrounded by parentheses for reasons of associativity.

***Recursion***

Recursion delineates two basic phases of process: *winding and unwinding*. In the winding phase, each recursive call perpetuates the recursion by making an additional recursive call itself. The winding phase terminates when one of the calls reaches a terminating condition. A *terminating condition* defines the state at which a recursive function should return instead of making another recursive call. Once the winding phase is complete, the process enters the unwinding phase, in which previous instances of the function are revisited in reverse order. For example:



A recursive tail is *tail recursive* when it is the last statement that will be executed within the body of a function and its return value is not a part of an expression. Tail-recursive functions are characterized as having nothing to do during unwinding phase and lead to better performance in practice.



*int facttail( int n, int a ){*

*if ( n < 0 ) return 0;*

*else if ( n == 0 ) return 1;*

*else if ( n == 1 ) return a; // Caution*

*else return facttail( n – 1, n \* a);*

*}*

***Analysis of Algorithms***

O-notation expresses the upper bound of a function within a constant factor. Primarily, we are interested only in the *growth rate (order of growth)*, which describes how quickly the algorithm’s performance will degrade as the size of the data it processes *becomes arbitrarily large*.

Simple rules for O-notation are shown below:

1. Constant terms are expressed as O(1). .
2. Multiplicative constants are omitted. .
3. Addition is performed by taking the maximum. .
4. Multiplication is not changed but often is re-written more compactly. .

***Linked Lists***

Linked list are more efficient in performing insertions and deletions and also make use of dynamically allocated storage.

*struct ListElmt { // Define a structure for linked list elements*

*LPVOID data;*

*ListElmt\* next;*

*};*

*struct List { // Define a structure for linked lists*

*ListElmt\* head;*

*ListElmt\* tail;  
};*

A circular list may be singly-linked or doubly-linked, but its distinguishing feature is that it has no tail. In circular list, the *next* pointer of the last element points back to its first element rather than to *NULL*. In the case of a doubly-linked circular list, the *prev* pointer of the first element is set to point to the last element as well.

***Stacks and Queue***

A *stack* stores and retrieves data in last-in, first-out, or *LIFO*, manner. To place an element on the top of a stack, we *push* it; to remove an element from the top, we *pop* it. However, a queue stores and retrieves data in a first-in, first-out, or *FIFO*, manner. To place an element at the tail of a queue, we *en-queue* it; to remove an element from the head, we *de-queue* it. Generally, a stack or queue can be implemented in a list structure.

One popular application of queues is handling events in event-driven applications. In a general user interface developed in Windows, the behavior of an application depends a great deal on key presses, mouse movements, and other events triggered by the user.

***Sets***

*Sets* are collections of distinguishable objects, called *members*, grouped together because they are in some way related. Two important characteristics of sets are that their members are *unordered* and *no members occur more than once*. Formally, sets are written with braces around them. Thus, ifis a set containing the members 1, 2, and 3, then. Because a set is unordered, this is same as writing. If a member,, is in a set,, then membership is indicated by writing; otherwise,.

Some definitions of sets are: (1) A set containing no members is the *empty set*. The set of all possible members is the *universe*.is the universe.is an empty set. (2) Two sets are *equal* if they contain exactly the same members. (3) One set,, is a subset of another set,, ifcontains all of the members of. In set notation,meansis a subset of;meansis not a subset of.

Operations of sets are: (1) The *union* of two sets,and, is a set,, that contains all of the members ofin addition to all of the members of.represents the union ofand. (2) The *intersection* of two sets,and, is a set,, that contains only the members existing in bothand.represents the intersection ofand. (3) The *difference* of two sets,and, is a set,, that contains all of the members ofexcept those in. represents the difference ofand. The way to implement a set is as a linked list.

A multi-set is a type of set that allows members to occur more than once. In a multi-set, inserting a member is considerably more efficient because we do not have to traverse the members looking for duplicates. Therefore, we can insert the new member intime. In a multi-set, removing a member remains anprocess because we still must search for the member we want to remove.

***Hash Tables***

A *chained hash table* fundamentally consists of an array of linked lists. When two keys hash to the same position in a hash table, they collide. Ideally, we would like all buckets to grow at the same rate so that they remain nearly the same size and as small as possible. This theoretically perfect situation is known as *uniform hashing*. The *load factor* of a hash table is defined as:whereis the number of elements in the table andis the number of positions into which elements may be hashed.

The goal of a good hash function is to approximate uniform hashing, that is, to spread elements about a hash table in as uniform and random a manner as possible. A hash functionis a function we define to map a keyto some position in a hash table.is called the *hash coding* of. Formally stated:. Mostly, the keyis an integer.

Division method is to map it into one ofpositions in a table by taking the remainder ofdivided by. Formally stated: . Typically, we should avoid values forthat are powers of 2. This is because if,becomes just thelowest-order bits of. Usually we chooseto be a prime number not too close to a power of 2, while considering storage constraints and load factor.

The multiplication method is to multiply the integer keyby a constantin the range; extract the fractional part; multiply this value by the number of positions in the table,. An advantage to this method is that, the number of positions in the table, is not as critical as in the division method. Formally stated:, where. An advantage to this method is that, the number of positions in the table, is not as critical as in the division method.

In a chained hash table, elements reside in buckets extending from each position. In an *open-addressed hash table*, all elements reside in the table itself. This table is a fixed size, and needs another way to resolve collisions.

***Trees***

A tree consists of elements called *nodes* organized in a hierarchical arrangement. The node at the top of the hierarchy is called the *root*. The nodes directly below the root are its children. A tree’s *branching factor* dictates how fast the tree will branch out as nodes are inserted. The binary tree is a tree with a branching factor of 2. Traversing a binary tree means visiting its nodes one at a time in a specific order. There are four types: (1) Pre-order traversal: we first traverse its root, then to the left then to the right. It is a depth-first exploration. (2) *In-order traversal*: we first traverse to the left, then to the root, and then to the right. (3) *Post-order traversal*: we fist traverse to the left, then to the right, and then to the root. (4) *Level-order traversal*: we visit its nodes beginning at the root and proceed downward, visiting the nodes at each level from left to right. It is a breadth-first exploration.

To remove the sub-tree rooted at the left/right child of a specified node, we can implement a function which removes nodes by performing post-order traversal.

We can create a binary search tree to search the data. To insert/find a node, we start at the root of the tree. If we encounter a node that is less, we follow its right pointer otherwise we follow its left pointer. When we reach the end of a branch, we make the inserting. Duplicate keys are not allowed. Searching becomes anoperation, provided the tree is kept balanced. Keeping a tree balanced means that it will be as short as possible for a given number of nodes.

The AVL tree, also called *height balanced binary search tree*, is a balanced tree which stores an extra piece of information with each node: *balance factor*. A binary treeis called an AVL tree if it fits two principals: (1) andare also an AVL tree, whichandare the sub-trees of. (2), whichandare the heights of these two sub-trees. In the other words, it means the balance factor of each node is -1, 0 or 1, or.

A sub-tree whose root node has a balance factor of +1 is said to be *left-heavy*. A sub-tree whose root node has a balance factor of -1 is said to be *right-heavy*. A sub-tree whose root node has a balance factor of 0 is considered *balanced*. If any balance factor becomes 2 or -2, we must re-balance the tree from that point down, which is done by performing an operation called a *rotation*. There are four different conditions to keep an AVL tree balanced when insert/delete a tree into/from this tree: (1) LL type: the child node exists in left node of the left sub-tree. (2) RR type: the child node exists in right node of the right sub-tree. (3) LR type: the child node exists in left node of the right sub-tree. (1) RL type: the child node exists in right node of the left sub-tree.

The smallest node in a binary search tree is the node that is the furthest to the left and the largest one is the node to the right. If we are interested only in determining the smallest element in a set of data repeatedly, we use a priority queue.

***Heaps and Priority Queues***

To quickly determine only the largest or smallest element, we need only keep this element where we can find it. Heaps and priority queues let us do this in an efficient way. It is not the goal to keep every element in order.

A heap is a tree, usually a binary tree. A heap in which each child is smaller than its parent is *top-heavy*. This is because the largest node is on top. A heap in which each child is larger than its parent is *bottom-heavy*. Heaps are left-balanced, so as nodes are added, the tree grows level by level from left to right. Therefore, heaps are stored nodes contiguously in an array in the order we would encounter them in a level traversal. Assuming a zero-indexed array, this means that the parent of each node at some positionin the array is located at position, wheremeans to ignore the fractional part of. The left and right children of a node are located at the positionsand.

To insert a node into heap, the new node initially places it into the last position in the array. When this causes the heap property to be violated, we must re-heapify the tree. Starting at the new node, we move up the tree level by level, comparing each child with its parent. At each level, if a parent and child are in the wrong order, we swap their contents. This process continues until we reach a level at which no swap is required, or we reach the top of the tree. The runtime complexity is. On the other hand, extracting a node from a heap may cause the heap property to be violated, we must re-heapify the tree. To re-heapify a tree after extracting a node, we start at the root node and move down the tree level by level, comparing each node with its two children. At each level, if a parent and its children are in the wrong order, we swap their contents and move to the child that was the most out of order. This process continues until we reach a level at which no swap is required, or we each a leaf node. The runtime complexity is.

A priority queue consists of elements organized so that the highest priority element can be ascertained efficiently. The most intuitive approach is simply to maintain a sorted set of data. In this approach, the element at the beginning of the sorted set is the one with the highest priority. However, inserting and extracting elements require re-sorting the set, which is anprocess in the worst case. Therefore, a better solution is to keep the set *partially ordered* using a heap.

Left-balanced binary trees are particularly well-suited to arrays since no nodes go un-used between 0 and n-1, but not true for all binary trees. In a task scheduler, the lower-priority tasks are said to be experiencing starvation. To solve this, a system employs some mechanism to increase a task’ priority gradually as its time in the queue grows. Thus, even in a busy system flooded by high-priority tasks, a low priority task eventually will obtain a high enough priority to rise to the top.

***The Bresenham line algorithm***

The common conventions that pixel coordinates increase in the down and right directions and that pixel centers have integer coordinates will be used. The endpoints of the line are the pixels at (*x*0, *y*0) and (*x*1, *y*1), where the first coordinate of the pair is the column and the second is the row.

The algorithm will be initially presented only for the [octant](http://en.wikipedia.org/wiki/Octant) in which the segment goes down and to the right (*x*0≤*x*1 and *y*0≤*y*1), and its horizontal projection *x*1 − *x*0 is longer than the vertical projection *y*1 − *y*0 (the line has a [slope](http://en.wikipedia.org/wiki/Slope) less than 1 and greater than 0.) In this octant, for each column *x* between *x*0 and *x*1, there is exactly one row *y* containing a pixel of the line, while each row between *y*0 and *y*1 may contain multiple rasterized pixels.

Bresenham's algorithm chooses the integer *y* corresponding to the pixel center that is closest to the ideal (fractional) *y* for the same *x*; on successive columns y can remain the same or increase by 1. The general equation of the line through the endpoints is given by:

y - y_0 = \frac{y_1-y_0}{x_1-x_0} (x-x_0).[](http://en.wikipedia.org/wiki/File:Bresenham.svg)

Since we know the column, *x*, the pixel's row, *y*, is given by rounding this quantity to the nearest integer:

\frac{y_1-y_0}{x_1-x_0} (x-x_0) + y_0.

The slope (*y*1 − *y*0) / (*x*1 − *x*0) depends on the endpoint coordinates only and can be pre-computed, and the ideal *y* for successive integer values of *x* can be computed starting from *y*0 and repeatedly adding the slope. Definitions of SWAP(x, y) (x ^= y ^= x ^= y) are provided for efficiency.

***Graphs***

Graphs are generally used to model problems defined in terms of relationships or connections between objects, and composed of two types of elements: *vertices* and *edges*. Vertices represent objects, and edges establish relationships or connections between the objects. In many problems, weights are associated with a graph’s edges. Graphs may be either *directed* or *undirected*. Formally, a graph is a pair, whereis a set of vertices andis a binary relation on. A *path* is a sequence of vertices traversed by following the edges between them. An undirected graph is connected if every vertex is reachable from each other by following some path. If it contains certain sections that are connected, it is called *connected components*.

The most common way to represent a graph is using an *adjacency-list* representation. Each structure in the list contains two members: a vertex and a list of vertices adjacent to the vertex. In a graph, if two verticesandin form an edgein, vertexis included in the adjacency list of vertex. Thus, in a directed graph, the total number of vertices in all adjacency lists is the same as the total number of edges. In an undirected graph, since edgeimplies an edge, the total number of vertices in all adjacency lists is twice the total number of edges.

Typically, *adjacency lists* are used for graphs that are spares, that is, graphs in which the number of edges is less than the number of vertices squared. However, if a graph is dense, we may choose to represent it using an *adjacency-matrix* representation.

*Breadth-first search* explores a graph by visiting all vertices adjacent to a vertex before exploring the graph further. The start vertex is placed alone in a queue. For each vertex in the queue, we peek at the vertex at the front of the queue and explored each vertex adjacent to it. Once all adjacent vertices have been explored, we de-queue the vertex at the front of the queue. The BFS determines the shortest path.

*Depth-first search* explores a graph by first visiting undiscovered vertices adjacent to the vertex most recently discovered. Thus, the search continually tries to explore as deep as it can. Exploit the recursion to implement the DFS.

An adjacency-list representation of a graph consists of a linked list of adjacency-list structures. Each structure in the list contains two members: a vertex and a list of vertices adjacent to the vertex. Each adjacency list is implemented as a set.

A binary tree is a directed acyclic graph with the following characteristics. Each node has up to two edges incident from it and one edge incident to it, except for the root node, which has only the two edges incident from it. Thus, the adjacency list of each vertex contains its children.

***Sorting and Searching***

*Insertion sort* is inefficient for large sets of data since determining each element belongs in the sorted set potentially requires comparing it with every other element in the sorted set thus far. An important virtue of insertion sort is that inserting a single element into a set that is already sorted requires only one scan of the sorted elements.

Insertion sort works fundamentally by inserting elements from an unsorted set one at a time into a sorted set. It can easily be adapted to work with linked lists efficiently. Its complexity is.

Quick sort is divide-and-conquer sorting algorithm and it efficiency makes it a better choice for medium to large sets of data. The algorithm begins with an unsorted pile that we partition in two. In one pile we place all checks numbered less than or equal to what we think may be the median value (pivot), and in the other pile we place the checks greater than this. It is a divide-and-conquer algorithm. It performs badly when we choose partition values that continually force the majority of the elements into one partition. Instead, we need to partitions the elements in as balanced a manner as possible. The implementation is as follows:

1. Define the most left data as a pivot.
2. Letbe an index searching from left to right until finding a number larger than this pivot.
3. Letbe an index searching from right to left until finding a number smaller than this pivot.
4. If, break the loop.
5. If, swap the values in indexand.
6. Swap the value of pivot and index.
7. Recursive the left and right loops.

Merge sort is divide-and-conquer sorting algorithm and is exploited to sort two sorted data. The spirit of merge sort takes advantage of the portions of the sorted data to increase the sorting efficiency. If two smaller data are sorted, then it will be more efficient to use the merge sort to sort these two sorted data.



***Graph Algorithms***

Formally states, given an undirected, weighted graph, a minimum spanning tree is the setof edges inthat connect all vertices inat a minimum cost. Prim’s algorithm grows a minimum spanning tree by adding edges one at a time based on which looks best at the moment. The fact that Prim’s algorithm adds edges using this approach makes it greedy. Although greedy algorithms often yield approximations rather than optimal solutions, Prim’s algorithm actually provides an optimal result. This algorithm works by repeatedly selecting a vertex and exploring the edges incident on it to determine. The algorithm resembles breadth-first search because it explores all edges incident on a vertex before moving deeper in the graph. Depending on the vertex we select, we end up exploring a different set of edges incident from the vertex. Thus, we can get different edges in the minimum spanning tree. Although the edges in the minimum spanning tree may vary, the total weight of the tree is always the same, which is the minimum for the graph. We can improve this part of the algorithm dramatically by using a *priority queue*.

Formally states, given a directed, weighted graph, the shortest path from vertextoinis the setof edges inthat connecttoinis the setof edges inthat connecttoat a minimum cost. Dijkstra’s algorithm grows a shortest path tree. Like Prim’s algorithm, Dijkstra’s algorithm is another example of a greedy algorithm that happens to produce an optimal result. The algorithm is greedy because it adds edges to the shortest-path tree based on which looks best at the moment. The algorithm resembles a breadth-first search because it explores all edges incident from a vertex before moving deeper in the graph.

==========================================================================

***Adapter***

If the interface of a class can’ t fulfill the new interface, and you can’t modify the class, in this situation, you can define an adapter. There are two ways to implement an adapter: (1) class adapter, (2) object adapter.

(1) Class adapter: An adapter publically derived from a target and privately derived from an adaptee.

(2) Object adapter: An adapter publically derived from a target and defines a pointer to an adaptee.

For example:

*class Shape{ // Target*

*public:*

*virtual void BoundingBox(POINT& ptBL, POINT& ptTR);*

*};*

*Class TextView{ // Adaptee*

*public:*

*void GetOrigin(Coord& x, Coord& y)const;*

*void GetExtent(Coord& width, Coord& height)const;*

*virtual bool IsEmpty()const;*

*};*

*Class TextShape : public Shape, private TextView{ // Class Adapter*

*public:*

*TextShape();*

*virtual void BoundingBox(POINT& ptBL, POINT& ptTR)*

*{*

*Coord bottom, left, width, height;*

*GetOrigin(bottom, left);*

*GetExtent(width, height);*

*ptBL = POINT(bottom, left);*

*ptTR = POINT(bottom + height, left + width);*

*}*

*bool IsEmpty()const{ return TextView::IsEmpty(); }*

*}*

*Class TextShape : public Shape{ // Object Adapter*

*private:*

*TextView\* m\_text;*

*public:*

*TextShape(TextView\* t);{ m\_text = t; }*

*virtual void BoundingBox(POINT& ptBL, POINT& ptTR)*

*{*

*Coord bottom, left, width, height;*

*m\_text->GetOrigin(bottom, left);*

*m\_text->GetExtent(width, height);*

*ptBL = POINT(bottom, left);*

*ptTR = POINT(bottom + height, left + width);*

*}*

*bool IsEmpty()const{ return m\_text->IsEmpty(); }*

*}*

The object adapter can own more flexibility than the class one.

***Singleton***

If you want to make sure a specified class can be allocated only one object in an application, you should use singleton.

*class CSingleton{*

*public:*

*static CSingleton\* GetInstance();*

*DWORD AddRef();*

*DWORD Release();*

*protected:*

*CSingleton(){m\_dwRes = 0; }*

*static CSingleton\* m\_pInstance;*

*DWORD m\_dwRes;*

*};*

*CSingleton\* CSingleton::m\_pInstance = NULL;*

*CSingleton\* CSingleton::GetInstance(){*

*if ( m\_pInstance == NULL ) m\_pInstance = new CSingleton;*

*m\_pInstance->AddRef();*

*}*

*DWORD CSingleton::AddRef(){*

*::InterlockedIncrement( ( LPLONG )&m\_dwRes );*

*return m\_dwRes*

*}*

*DWORD CSingleton::Release(){*

*if ( m\_dwRes != 0 ) ::InterlockedDecrement( ( LPLONG )&m\_dwRes );*

*if ( m\_dwRes == 0 ){*

*delete this;*

*return 0;*

*}*

*return m\_dwRes;*

*}*

***Decorator***

Decorator uses Delegation, but in a very specific way. That’s something like this:

*class Delegator {*

*private final ClassA a = new ClassA();*

*private final ClassB b = new ClassB/();*

*public void doWork() {*

*a.setup();*

*final ResFromA resa = a.getRes();*

*b.setup();*

*b.consume(resa);*

*}*

*}*

Delegator does not share any type or interface with either *ClassA* or *ClassB*, and knows the exact type of both a and b.

Decorator is a dynamic way to use delegation to add behavior to a logical entity at runtime. In Decorator all entities share a common interface, and use delegation to concatenate their work.

*public interface Item {*

*public voic drawAt(final int x, final int y);*

*}*

*public class CircleAround implements Item {*

*private final Item wrapped;*

*private final int radius;*

*public CircleAround(public final Item wrapped, public final int radius) {*

*this.wrapped = wrapped;*

*this.radius = radius;*

*}*

*public void drawAt(final int x, final int y) {*

*wrapped.drawAt(x,y); // First handle whatever we are wrapping*

*Graphics.drawCircle(x, y, radius); // Then add our circle*

*}*

*}*

Note that unlike the first example, *CircleAround* does not know the exact type of the item that it wraps, and shares a common interface with it.

Introduction

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

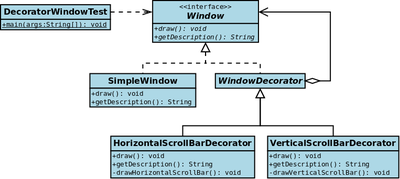
Designing a new *decorator* class can be used to [wraps](http://en.wikipedia.org/wiki/Wrapper_pattern) the original class. This wrapping could be achieved by the following sequence of steps:

1. Subclass the original *Component* class into a *Decorator* class (see UML diagram);
2. In the *Decorator* class, add a *Component* pointer as a field;
3. Pass a *Component* to the *Decorator* constructor to initialize the *Component* pointer;
4. In the Decorator class, redirect all *Component* methods to the *Component* pointer;
5. In the *ConcreteDecorator* class, override any *Component* method(s) whose behavior needs to be modified.

Note that decorators and the original class object share a common set of features. In the previous diagram, the *operation* method was available in both the decorated and undecorated versions.

The decoration features (e.g., methods, members) are usually defined by an interface or class inheritance which is shared by the decorators and the decorated object. In the previous example the class "*Component*" is inherited by both the "*ConcreteComponent*" and the subclasses that descend from "*Decorator*".

The decorator pattern is an alternative to [subclassing](http://en.wikipedia.org/wiki/Subclass_(computer_science)). Subclassing adds behavior at [compile time](http://en.wikipedia.org/wiki/Compile_time), and the change affects all instances of the original class; decorating can provide new behavior at [run-time](http://en.wikipedia.org/wiki/Run_time_(program_lifecycle_phase)) for individual objects.

[](http://en.wikipedia.org/wiki/File:UML2_Decorator_Pattern.png)

As an example, consider a window in a [windowing system](http://en.wikipedia.org/wiki/Windowing_system). To allow [scrolling](http://en.wikipedia.org/wiki/Scrolling) of the window's contents, we may wish to add horizontal or vertical [scrollbars](http://en.wikipedia.org/wiki/Scrollbar) to it, as appropriate. Assume windows are represented by instances of the *Window* class, and assume this class has no functionality for adding scrollbars. We could create a subclass *ScrollingWindow* that provides them, or we could create a *ScrollingWindowDecorator* that adds this functionality to existing *Window* objects.

The following Java example illustrates the use of decorators using the window/scrolling scenario.

***public******interface*** *Window { // the Window interface class*

***public******void*** *draw(); // draws the Window*

***public*** *String getDescription(); // returns a description of the Window*

*}*

***class*** *SimpleWindow* ***implements*** *Window { // extension of a simple Window without any scrollbars*

***public******void*** *draw() { // draw window }*

***public*** *String getDescription() {* ***return*** *"simple window"; }*

*}*

The following classes contain the decorators for all Window classes, including the decorator classes themselves.

***abstract******class*** *WindowDecorator* ***implements*** *Window { // abstract decorator class - note that it implements Window*

***protected*** *Window decoratedWindow; // the Window being decorated*

***public*** *WindowDecorator (Window decoratedWindow) {* ***this****.decoratedWindow = decoratedWindow; }*

***public******void*** *draw() { decoratedWindow.draw(); //delegation }*

***public*** *String getDescription() {* ***return*** *decoratedWindow.getDescription(); //delegation }*

*}*

***class*** *VerticalScrollBarDecorator* ***extends*** *WindowDecorator {// concrete decorator - adds vertical scrollbar functionality*

***public*** *VerticalScrollBarDecorator (Window decoratedWindow) {* ***super****(decoratedWindow); }*

*@Override*

***public******void*** *draw() {*

***super****.draw();*

*drawVerticalScrollBar();*

*}*

***private******void*** *drawVerticalScrollBar() { // draw the vertical scrollbar }*

*@Override*

***public*** *String getDescription() {* ***return******super****.getDescription() + ", including vertical scrollbars"; }*

*}*

Here's a test program that creates a Window instance which is fully decorated (i.e., with vertical and horizontal scrollbars), and prints its description:

***public******class*** *DecoratedWindowTest {*

***public******static******void*** *main(String[] args) {*

*// create a decorated Window with horizontal and vertical scrollbars*

*Window decoratedWindow =* ***new*** *HorizontalScrollBarDecorator (*

***new*** *VerticalScrollBarDecorator (****new*** *SimpleWindow()));*

*System.out.println(decoratedWindow.getDescription()); // print the Window's description*

*}*

*}*

The output is "simple window, including vertical scrollbars, including horizontal scrollbars". Notice how *getDescription()* of the two decorators first retrieve the decorated Window's description and *decorates* it with a suffix.

The next example illustrates the use of decorators using coffee making scenario:

*#include <iostream>*

*#include <string>*

*/\* Abstract base class \*/*

*class Coffee { // The abstract Coffee class defines the functionality of Coffee implemented by decorator*

*virtual double getCost() = 0; // returns the cost of the coffee*

*virtual std::string getIngredients() = 0; // returns the ingredients of the coffee*

*virtual ~Coffee() = 0;*

*};*

*inline Coffee::~Coffee(){}*

*/\* SimpleCoffee class. \*/*

*class SimpleCoffee : public Coffee { // extension of a simple coffee without any extra ingredients*

*double getCost() { return 1.0; }*

*std::string getIngredients() { return "Coffee"; }*

*};*

*/\* Decorators \*/*

*class MilkDecorator : public Coffee { // Decorator Milk that adds milk to coffee*

*public:*

*MilkDecorator (Coffee\* basicCoffee) :basicCoffee\_(basicCoffee) {}*

*double getCost() { return basicCoffee\_->getCost() + 0.5; // providing methods defined in the abstract superclass }*

*std::string getIngredients() { return basicCoffee\_->getIngredients() + ", " + "Milk"; }*

*private:*

*Coffee\* basicCoffee\_;*

*};*

*class WhipDecorator : public Coffee { // Decorator Whip that adds whip to coffee*

*public:*

*WhipDecorator (Coffee\* basicCoffee) :basicCoffee\_(basicCoffee) {}*

*double getCost() {return basicCoffee\_->getCost() + 0.7; }*

*std::string getIngredients() { return basicCoffee\_->getIngredients() + ", " + "Whip"; }*

*private:*

*Coffee\* basicCoffee\_;*

*};*

*/\* Test program \*/*

*int main() {*

*SimpleCoffee s;*

*std::cout << "Cost: " << s.getCost() << "; Ingredients: " << s.getIngredients() << std::endl;*

*MilkDecorator m(&s);*

*std::cout << "Cost: " << m.getCost() << "; Ingredients: " << m.getIngredients() << std::endl;*

*WhipDecorator w(&s);*

*std::cout << "Cost: " << w.getCost() << "; Ingredients: " << w.getIngredients() << std::endl;*

*MilkDecorator m2(&w); // Note that you can stack decorators:*

*std::cout << "Cost: " << m2.getCost() << "; Ingredients: " << m2.getIngredients() << std::endl;*

*}*

The output of this program is given below:

Cost: 1.0; Ingredients: Coffee

Cost: 1.5; Ingredients: Coffee, Milk

Cost: 1.7; Ingredients: Coffee, Whip

Cost: 2.2; Ingredients: Coffee, Whip, Milk

**Dynamic languages**

The decorator pattern can also be implemented in dynamic languages either with interfaces or with traditional OOP inheritance.

<http://stackoverflow.com/questions/13389544/difference-between-decorator-pattern-and-delegation-pattern>