<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)

## 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 behaviors 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”. 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. 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) |

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

***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*

*int size;*

*void ( \*destroy )( void\* );*

*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 Queues*

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 fro 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 a 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.

The idea of set covering is that: given a setand a setof subsetstoof, set, which is composed of one or more sets from, is said to coverif each member inis contained in at least one of the subsets in; in addition, contains as few sets fromas possible. The implementation of this algorithm is that: there are two loops in this algorithm, the outermost loop iterates as long as there are non-covered members inand the algorithm has not run out offor the covering. Inside this loop, during each iteration, it finds the set inthat produces the *largest intersection* with. It then adds this set toand removes its members from. Last in the loop, the selected set is removed from. The runtime complexity of set covering is.

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.

To process arithmetic expressions with a computer is using an *expression tree*. It is a binary tree consisting of nodes containing two types of objects: operators and terminal values. Operators are objects that have operands; terminal values are objects that have no operands. An expression tree allows us to translate an expression into one of three common representations: prefix, infix and postfix. To obtain these representations, we simply traverse the tree using a pre-order, in-order or post-order traversal. Infix expressions are the expressions we are most familiar with from mathematics, but they are not well suited to processing by a computer. Infix expressions do not inherently identify the order operations, whereas prefix and postfix expressions do. Postfix expressions are well suited to processing by a computer. The arithmetic expressions are easy to evaluate with an *abstract stack machine*. First, we move from left to right through the expression, pushing values onto the stack until an operator is encountered. Next, the operands required by the operator are popped, the operator is applied to them, and the result is pushed back on the stack.

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 each 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 because 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 manage this, typically 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 (in other words, 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* (computed by the algorithm) 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.



*Counting sort* is a stable sort and works fundamentally by counting how many times integer elements occur in an unsorted set to determine how the set should be ordered. Additional storage is allocated to store the sorted data temporarily. After allocating storage, we begin by counting the occurrences of each element. There are placed in an array indexed by the integer elements. Finally, exploit the array of counting occurrences and start from the end of the array to sort these data.

*Radix sort* works fundamentally by applying counting sort one position at a time to a set of data. Position by position, we apply counting sort to shuffle and reshuffle the elements, beginning with the least significant position. Once we have shuffled the elements by the digits in the most significant position, sorting is complete.

*Data Compression*

The bitwise XOR () of two binary operands yields 0 in each position where the bits are the same, and 1 where the bits are different.

The following example shows how to set a value to a specified bit

*BYTE mask = 0x2;*

*Bits[ 2 ] = Bits[ 2 ] | mask; // Set the second bit to 1.*

*Bits[ 2 ] = Bits[ 2 ] & ( ~mask ); // Set the second bit to 0.*

The idea of Huffman coding is to encode symbols that occur more frequently with fewer bits than those that occur less frequently. The entropyof a symbolis defined as:, whereis the probability ofbeing found in the data.

In Huffman compression, it begins by scanning the data to determine the frequency of each symbol. Then sort the data according to their frequencies. The symbols are inserted into a priority queue one binary tree. A structure to represent the symbol should at least contain two member variables: a symbol from the data and the frequency of this data. To build the Huffman tree, using the loop to perform mergers of the trees within the priority queue. In each iteration, call the priority queue twice to extract the two binary trees whose nodes have the smallest frequencies. Sum the frequencies, merge the trees into a new one, store the sum of the frequencies in the new tree’s root, and insert the new tree back into the priority queue. Continue this process until the only tree remaining in the priority queue is the final Huffman tree. To build the table, traverse the Huffman tree using a preorder traversal. Once we encounter a leaf node, we store the Huffman codes at the appropriate entry.

In Huffman decompression, we read the compressed data bit by bit. Starting at the root of the Huffman tree, and once we encounter a leaf node, we have obtained the Huffman code for a symbol. The decoded symbol resides in the leaf.

LZ77 is a dictionary-based method, which means that it tries to compress data by encoding long strings of symbols, called phrases, as small tokens that reference entries in a dictionary. LZ77 uses a *look-ahead buffer* and a *sliding window*. LZ77 works by first loading a portion of the data into the look-ahead buffer. Picture the buffer as a sequence of symbols, andas a set of phrases contrasted from the symbols. We form phrases, define as:. Then it moves into the sliding window and becomes part of the dictionary. Consider the window to be a sequence of symbolswe form the set of phrases as follows:. The main idea behind LZ77 is to look continually for the longest phrase in the look-ahead buffer that matches a phrase currently in the dictionary.

When there is at least one match, we encode the longest match as a *phrase token*. Phrase tokens contain three pieces of information: the offset in the sliding window where the match begins the number of symbols in the match, the number of symbols in the match and the first symbol in the look-ahead buffer after the match. When there is no match, we encode the unmatched symbol as a *symbol token*. Typical sizes for sliding windows are around 4K bytes. Look-ahead buffers are generally less than 100 bytes. In most cases, LZ77 results in better compression ratios than Huffman coding, but compression times are considerably slower. Compressing data with LZ77 is time consuming because we spend a lot of time searching the sliding window for matching phrases. However, in general, uncompressing data with LZ77 is even faster than uncompressing data with Huffman coding.

Effective compression with Huffman coding depends on symbols occurring in the data at varying frequencies. If all possible symbols occur at nearly the same frequency, poor compression results. Huffman coding also performs poorly when used to compress small amounts of data. Fortunately, most data are not uniformly distributed.

Effective compression with LZ77 depends on being able to encode many sequences of symbols using phrase tokens. If we generate a large number of symbol tokens and only a few phrase tokens representing predominantly short phrases, poor compression results. This occurs when the sliding window is made too small to use recurring phrases effectively.

LZ77 looks for matching phrases by comparing portions of the sliding window to portions of the look-ahead buffer essentially symbol by symbol. A more effective approach is to replace the sliding window with some type of data structure for efficient searching. For example, we might use a hash table or a binary search tree to store phrases encountered earlier.

*Data Encryption*

Data encryption, or cryptography, is the science of secrecy. Like data compression, it is another product of information theory. Data encryption entails two processes: in one process we encipher recognizable data, called plaintext, into an unrecognizable form, called ciphertext; in a second process we decipher the ciphertext back into the original plaintext.

Ciphers use a special piece of information, called a key. A cipher is either symmetric or asymmetric. In symmetric ciphers, the same key is used both to encipher and decipher data. In asymmetric ciphers, usually called public-key ciphers, the key used to encipher data is different from the key used to decipher it. The key used to encipher data is different from the key used to decipher it. The key used to encipher data is called the *public key*. The key used to decipher data is called the *private key*.

DES (Data Encryption Standard) is one of the most popular symmetric ciphers. DES is symmetric because it uses a single key both to encipher and decipher data. DES is a block cipher, which means that it processes data in fixed-size sections called blocks.

*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.

*Geometric Algorithms*

The formula to calculate the outer product:. It is a vector and the direction depends on the right hand rule. We can change the formula to a determinant:. Two steps to test for intersecting between line segments: (1) Perform a quick rejection test. Construct the bounding box for each line segment. The bounding boxes of two line segments intersect if all of the following tests are true:. (2) Perform a straddle test. The equations just given come from representing the line segments fromto, to, andtoas vectors, , andand using the signs of the z-components of the cross productsandas gauges of orientation.

Convex Hull: 葛立恆掃描法。由最底的一點A\_1開始，計算它跟其他各點的連線和x軸的角度，按小至大將這些角度排序，稱它們的對應點為*A*2,*A*3,...,*An*。這裡的時間複雜度可達*O*(*nlogn*)。考慮最小的角度對應的點*A*3。若由*A*2到*A*3的路徑相對*A*1到*A*2的路徑是向右轉的（可以想象一個人沿*A*1走到*A*2，他站在*A*2時，是向哪邊改變方向），表示*A*3不可能是凸包上的一點，考慮下一點由*A*2到*A*4的路徑；否則就考慮*A*3到*A*4的路徑是否向右轉……直到回到*A*1。這個演算法的整體時間複雜度是*O*(*nlogn*)，注意每點只會被考慮一次，而不像Jarvis步進法中會考慮多次。它的缺點是不能推廣到二維以上的情況。

The smallest polygon surrounding a set of points is called a convex hull. Recall that a polygon is convex if any line segment connecting two points inside the polygon lies completely inside the polygon itself.

*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;*

*}*