TwoSigma

In computing, a process is an instance of a computer program. Depending on the operating sy[stem](http://www.1point3acres.com/%E7%BE%8E%E5%9B%BD%E5%9B%BD%E5%9C%9F%E5%AE%89%E5%85%A8%E9%83%A82012%E5%B9%B4%E5%BA%A6%E6%9B%B4%E6%96%B0stem%E4%B8%93%E4%B8%9A%E5%90%8D%E5%8D%95-%E7%9C%8B%E7%9C%8B%E4%BD%A0%E7%9A%84%E4%B8%93%E4%B8%9A/) (OS), a process may be made up of multiple threads of execution that execute instructions concurrently.  
. From 1point 3acres bbs  
In computer science, a thread of execution is the smallest sequence of execution stream within a process.  
  
Each process provides the resources needed to execute a program. A process has a virtual address space, code, priviledges, etc. Each process is started with a single thread, often called the primary thread, but can create additional threads from any of its threads.. 1p

oint 3acres 璁哄潧  
A thread is the entity within a process that can be scheduled for execution. All threads of a process share its virtual address space and system resources.   
  
Typical difference is, processes run in separated memory while threads run in shared memory.

* processes are typically independent, while threads exist as subsets of a process
* processes carry considerably more state information than threads, whereas multiple threads within a process share process state as well as memory and other resources
* processes have separate address spaces, whereas threads share their address space
* processes interact only through system-provided inter-process communication mechanisms
* context switching between threads in the same process is typically faster than context switching between processe

- File. 鐗涗汉浜戦泦,涓€浜╀笁鍒嗗湴  
A record stored on disk, or a record synthesized on demand by a file server, which can be accessed by multiple processes.  
  
  
- Socket  
A data stream sent over a network interface, either to a different process on the same computer or to another computer on the network. Typically byte-oriented, sockets rarely preserve message boundaries. Data written through a socket requires formatting to preserve message boundaries.  
  
  
- Message Queue  
A data stream similar to a socket, but which usually preserves message boundaries. Typically implemented by the operating system, they allow multiple processes to read and write to the message queue without being directly connected to each other.  
  
Publish/Subscribe, Observer  
  
- Pipe     . Waral 鍗氬鏈夋洿澶氭枃绔�,  
A unidirectional data channel. Data written to the write end of the pipe is buffered by the operating system until it is read from the read end of the pipe. Two-way data streams between processes can be achieved by creating two pipes utilizing standard input and output.  
  
Like when we are using arrow symbol in command line..1point3acres缃�  
  
- Shared memory       
Multiple processes are given access to the same block of memory which creates a shared buffer for the processes to communicate with each other.  
-google 1point3acres  
-google 1point3acres  
- Semaphore       
A simple structure that synchronizes multiple processes acting on shared resources.  
  
  
ITC:. from: [1point3acres.com/bbs](http://1point3acres.com/bbs)   
- Synchronization primitives, like locks and semaphores  
- Through Events: wait, notify  
. 鍥磋鎴戜滑@1point 3 acres  
- Shared memory, cause typically they all live in the same process  
  
Each thread has a private stack, which it can quickly add and remove items from. This makes stack based memory fast, but if you use too much stack memory, as occurs in infinite recursion, you will get a stack overflow.  
  
All threads share a common heap. Since all threads share the same heap, access to the allocator/deallocator must be synchronized. There are various methods and libraries for avoiding allocator contention.  
  
Some languages allow you to create private pools of memory, or individual heaps, which you can assign to a single thread.  
. visit [1point3acres.com](http://1point3acres.com/) for more.  
  
every thread would be allocated its own memory space in stack while typically there is only one heap within one process. This means heap space is shared among all threads. Since it is global, it is faster in speed. But also, this causes synchronization issues, which could possibly slow the whole system down.. [1point3acres.com/bbs](http://1point3acres.com/bbs)  
  
Some languages or OS my support allocating heaps for each thread.

Latency vs Throughput

An application consists of one or more processes. A *process*, in the simplest terms, is an executing program. One or more threads run in the context of the process. A *thread* is the basic unit to which the operating system allocates processor time. A thread can execute any part of the process code, including parts currently being executed by another thread.

class WeightedRandom{

private:

    map<int, string> weightMap;

public:

    WeightedRandom(){

    }

    void update(string name, int weight){

        bool found = false;

        int prevWeight = 0;

        vector<pair<int, string>> eraseSet;

        vector<pair<int, string>> addSet;

        int change = 0;

        for(auto it = weightMap.begin(); it != weightMap.end(); it++){

            if(found == false){

                if(it->second != name){

                    prevWeight = it->first;

                }

                else{

                    found = true;

                    change = weight - (it->first - prevWeight);

                    eraseSet.push\_back(\*it);

                    addSet.push\_back(make\_pair(it->first + change, it->second));

                }

            }

            else{

                eraseSet.push\_back(\*it);

                addSet.push\_back(make\_pair(it->first + change, it->second));

            }

        }

        if(found == false){

            weightMap[weight + prevWeight] = name;

            return;

        }

        for(auto it: eraseSet){

            weightMap.erase(it.first);

        }

        for(auto it: addSet){

            weightMap.insert(it);

        }

    }

    string getRandom(){

        int randNum = rand();

        auto it = weightMap.end();

        it--;

        auto res = weightMap.upper\_bound(randNum % (it->first));

        return res->second;

    }

    void printMap(){

        for(auto it:weightMap){

            cout << it.second << " " << it.first << endl;

        }

    }

};

Why quick sort is better than merge sort:

Quicksort has O(*n*2) worst-case runtime and O(*n*log*n*) average case runtime.

In particular, the often-quoted runtime of sorting algorithms refers to the number of comparisons or the number of swaps necessary to perform to sort the data. This is indeed a good measure of performance, especially since it’s independent of the underlying hardware design. However, other things – such as locality of reference (i.e. do we read lots of elements which are probably in cache?) – also play an important role on current hardware. Quicksort in particular requires little additional space and exhibits good cache locality, and this makes it faster than merge sort in many cases.

In addition, it’s very easy to avoid quicksort’s worst-case run time of O(*n*2) almost entirely by using an appropriate choice of the pivot – such as picking it at random (this is an excellent strategy).

In practice, many modern implementations of quicksort (in particular libstdc++’s std::sort) are actually [introsort](http://en.wikipedia.org/wiki/Introsort), whose theoretical worst-case is O(*n*log*n*), same as merge sort. It achieves this by limiting the recursion depth, and switching to a different algorithm ([heapsort](http://en.wikipedia.org/wiki/Heapsort)) once it exceeds log*n*.

There's one scenario, though, where these advantages disappear. Suppose you want to sort a linked list of elements. The linked list elements are scattered throughout memory, so advantage (1) disappears (there's no locality of reference). Second, linked lists can be merged with only O(1) space overhead instead of O(n) space overhead, so advantage (2) disappears. Consequently, you usually will find that mergesort is a superior algorithm for sorting linked lists, since it makes fewer total comparisons and isn't susceptible to a poor pivot choice.

# Power of Three

bool isPowerOfThree(int n) {

if(n <= 0) return false;

if(n == 1) return true;

int power = log(INT\_MAX) / log(3);

int largestPower = pow(3, power);

if(largestPower % n == 0){

return true;

}

return false;

}

# Power of Four

bool isPowerOfFour(int num) {

if(num == 1) return true;

while(num > 0){

int temp = num % 4;

if(temp != 0){

return false;

}

num = num / 4;

if(num == 1){

return true;

}

}

return false;

}

bool isPowerOfFour(int num) {

if(num <= 0) return false;

if((num & (num-1)) != 0 )return false;

if((num | 0x55555555) != 0x55555555) {

return false;

}

return true;

}

Hash Function:

Open Addressing

Like separate chaining, open addressing is a method for handling collisions. In Open Addressing, all elements are stored in the hash table itself. So at any point, size of table must be greater than or equal to total number of keys (Note that we can increase table size by copying old data if needed).

Insert(k): Keep probing until an empty slot is found. Once an empty slot is found, insert k.

Search(k): Keep probing until slot’s key doesn’t become equal to k or an empty slot is reached.

Delete(k): Delete operation is interesting. If we simply delete a key, then search may fail. So slots of deleted keys are marked specially as “deleted”.

Insert can insert an item in a deleted slot, but search doesn’t stop at a deleted slot.

Open Addressing is done following ways:

a) Linear Probing: In linear probing, we linearly probe for next slot. For example, typical gap between two probes is 1 as taken in below example also.

let hash(x) be the slot index computed using hash function and S be the table size

If slot hash(x) % S is full, then we try (hash(x) + 1) % S

If (hash(x) + 1) % S is also full, then we try (hash(x) + 2) % S

If (hash(x) + 2) % S is also full, then we try (hash(x) + 3) % S

..................................................

..................................................

Let us consider a simple hash function as “key mod 7” and sequence of keys as 50, 700, 76, 85, 92, 73, 101.

Clustering: The main problem with linear probing is clustering, many consecutive elements form groups and it starts taking time to find a free slot or to search an element.

b) Quadratic Probing We look for i2‘th slot in i’th iteration.

let hash(x) be the slot index computed using hash function.

If slot hash(x) % S is full, then we try (hash(x) + 1\*1) % S

If (hash(x) + 1\*1) % S is also full, then we try (hash(x) + 2\*2) % S

If (hash(x) + 2\*2) % S is also full, then we try (hash(x) + 3\*3) % S

..................................................

..................................................

c) Double Hashing We use another hash function hash2(x) and look for i\*hash2(x) slot in i’th rotation.

let hash(x) be the slot index computed using hash function.

If slot hash(x) % S is full, then we try (hash(x) + 1\*hash2(x)) % S

If (hash(x) + 1\*hash2(x)) % S is also full, then we try (hash(x) + 2\*hash2(x)) % S

If (hash(x) + 2\*hash2(x)) % S is also full, then we try (hash(x) + 3\*hash2(x)) % S

..................................................

..................................................

See this for step by step diagrams.

Comparison of above three:

Linear probing has the best cache performance, but suffers from clustering. One more advantage of Linear probing is easy to compute.

Quadratic probing lies between the two in terms of cache performance and clustering.

Double hashing has poor cache performance but no clustering. Double hashing requires more computation time as two hash functions need to be computed.

Open Addressing vs. Separate Chaining

Advantages of Chaining:

1) Chaining is Simpler to implement.

2) In chaining, Hash table never fills up, we can always add more elements to chain. In open addressing, table may become full.

3) Chaining is Less sensitive to the hash function or load factors.

4) Chaining is mostly used when it is unknown how many and how frequently keys may be inserted or deleted.

5) Open addressing requires extra care for to avoid clustering and load factor.

Advantages of Open Addressing

1) Cache performance of chaining is not good as keys are stored using linked list. Open addressing provides better cache performance as everything is stored in same table.

2) Wastage of Space (Some Parts of hash table in chaining are never used). In Open addressing, a slot can be used even if an input doesn’t map to it.

3) Chaining uses extra space for links.

Performance of Open Addressing:

Like Chaining, performance of hashing can be evaluated under the assumption that each key is equally likely to be hashed to any slot of table (simple uniform hashing)

m = Number of slots in hash table

n = Number of keys to be inserted in has table

Load factor α = n/m ( < 1 )

Expected time to search/insert/delete < 1/(1 - α)

So Search, Insert and Delete take (1/(1 - α)) time

# Advantage of BST over hash table

Hash Table supports following operations in Θ(1) time.

1) Search

2) Insert

3) Delete

The time complexity of above operations in a self-balancing Binary Search Tree (BST) (like Red-Black Tree, AVL Tree, Splay Tree, etc) is O(Logn).

So Hash Table seems to beating BST in all common operations. When should we prefer BST over Hash Tables, what are advantages. Following are some important points in favor of BSTs.

We can get all keys in sorted order by just doing Inorder Traversal of BST. This is not a natural operation in Hash Tables and requires extra efforts.

Doing order statistics, finding closest lower and greater elements, doing range queries are easy to do with BSTs. Like sorting, these operations are not a natural operation with Hash Tables.

BSTs are easy to implement compared to hashing, we can easily implement our own customized BST. To implement Hashing, we generally rely on libraries provided by programming languages.

With BSTs, all operations are guaranteed to work in O(Logn) time. But with Hashing, Θ(1) is average time and some particular operations may be costly, especially when table resizing happens.

# Find Median from data stream

struct compare{

bool operator()(int num1, int num2){

return num1 > num2;

}

};

class MedianFinder {

private:

priority\_queue<int> left;

priority\_queue<int, vector<int>, compare> right;

public:

/\*\* initialize your data structure here. \*/

MedianFinder() {

}

void addNum(int num) {

right.push(num);

if( left.size() != 0 && right.top() < left.top()){

left.push(right.top());

right.pop();

right.push(left.top());

left.pop();

}

if(right.size() - left.size() > 1){

left.push(right.top());

right.pop();

}

}

double findMedian() {

if(right.size() == left.size()){

return ((double)left.top() + (double)right.top())/2.0;

}

else{

return (double)right.top();

}

}

};

上来是个比较nice的美国小哥，是休斯顿那边的office打来的。  
1. 介绍自己比较有挑战的项目  
2. 什么是Hash table，怎么实现的，如何解决碰撞  
3. 求data stream的中数  
4. Process vs Thread  
5. Coding: Power of Four  
6. 问他问题  
7. 心血来潮又追问了我latency vs throughput

# What is a Thread?

A thread is a path of execution within a process. Also, a process can contain multiple threads.

Process vs Thread?

The typical difference is that threads within the same process run in a shared memory space, while processes run in separate memory spaces.

Threads are not independent of one other like processes as a result threads shares with other threads their code section, data section and OS resources like open files and signals. But, like process, a thread has its own program counter (PC), a register set, and a stack space.

Advantages of Thread over Process

1. Responsiveness: If the process is divided into multiple threads, if one thread completed its execution, then its output can be immediately responded.

2. Faster context switch: Context switch time between threads is less compared to process context switch. Process context switch is more overhead for CPU.

3. Effective Utilization of Multiprocessor system: If we have multiple threads in a single process, then we can schedule multiple threads on multiple processor. This will make process execution faster.

4. Resource sharing: Resources like code, data and file can be shared among all threads within a process.

Note : stack and registers can’t be shared among the threads. Each thread have its own stack and registers.

5. Communication: Communication between multiple thread is easier as thread shares common address space. while in process we have to follow some specific communication technique for communication between two process.

6. Enhanced Throughput of the system: If process is divided into multiple threads and each thread function is considered as one job, then the number of jobs completed per unit time is increased. Thus, increasing the throughput of the system.

# IPC vs ITC

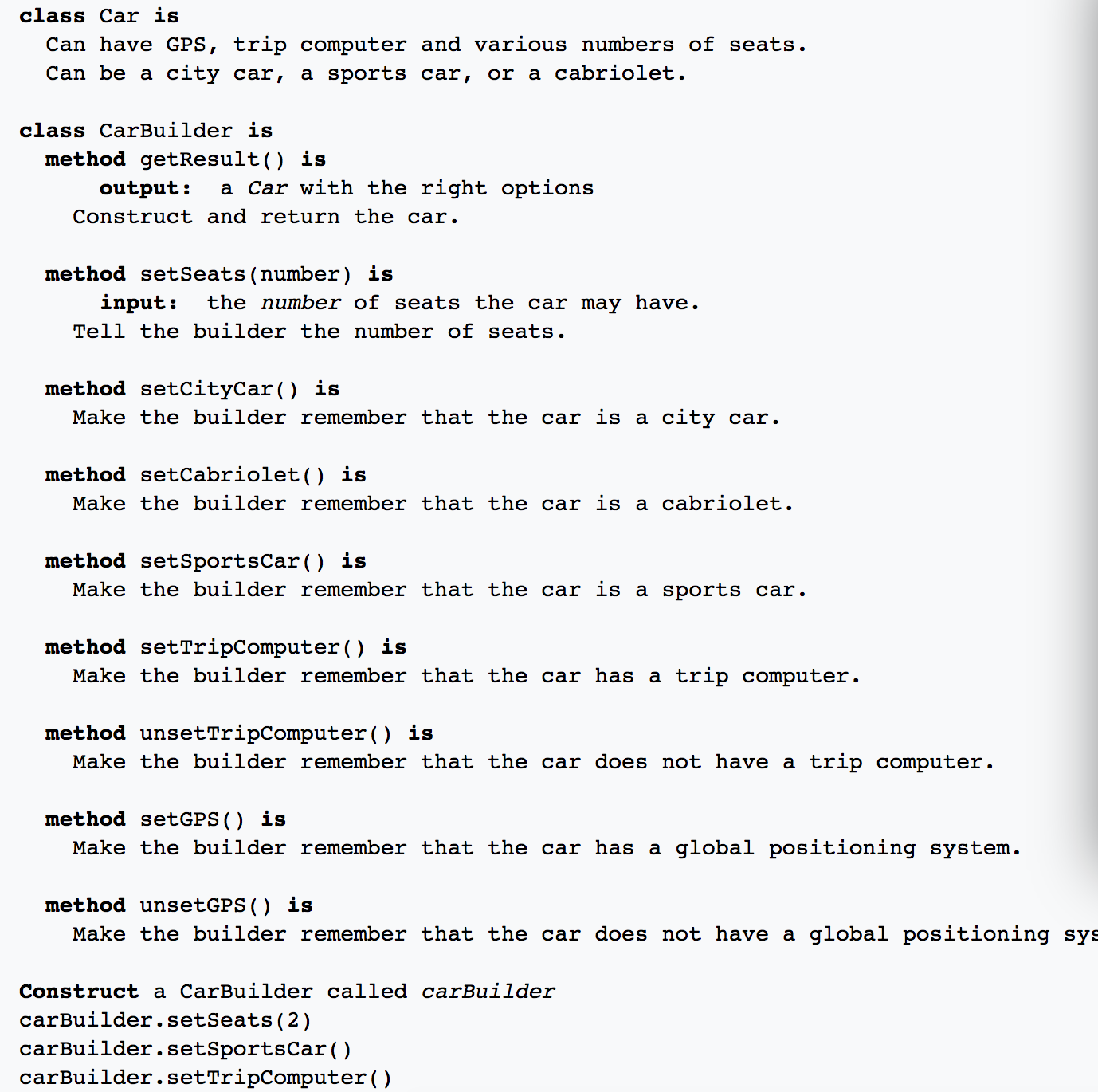
In practice, Java interthread communication can be implemented as plain Java method calls on shared object with appropriate synchronization thrown in. Alternatively, you can use the new concurrency classes to hide some of the nitty-gritty (and error prone) synchronization issues.

By contrast, Java interprocess communication is based at the lowest level on turning state, requests, etc into sequences of bytes that can be sent as messages or as a stream to another Java process. You can do this work yourself, or you can use a variety of "middleware" technologies of various levels of complexity to abstract away the implementation details. Technologies that may be used include, Java object serialization, XML, JSON, RMI, CORBA, SOAP / "web services", message queing, and so on.

At a practical level, interthread communication is many orders of magnitude faster than interprocess communication, and allows you to do many things a lot more simply. But the downside is that everything has to live in the same JVM, so there are potential scalability issues, security issues, robustness issues and so on.

# Builder Design Pattern

The builder pattern is an object creation software design pattern. Unlike the abstract factory pattern and the factory method pattern whose intention is to enable polymorphism, the intention of the builder pattern is to find a solution to the telescoping constructor anti-pattern[citation needed] that occurs when the increase of object constructor parameter combination leads to an exponential list of constructors. Instead of using numerous constructors, the builder pattern uses another object, a builder, that receives each initialization parameter step by step and then returns the resulting constructed object at once.



# Factory method:

Without factory:

Client(int type)  {

        // Client explicitly creates classes according to type

        if (type == 1)

            pVehicle = new TwoWheeler();

        else if (type == 2)

            pVehicle = new FourWheeler();

        else

            pVehicle = NULL;

    }

// Factory method to create objects of different types.

// Change is required only in this function to create a new object type

Vehicle\* Vehicle::Create(VehicleType type) {

    if (type == VT\_TwoWheeler)

        return new TwoWheeler();

    else if (type == VT\_ThreeWheeler)

        return new ThreeWheeler();

    else if (type == VT\_FourWheeler)

        return new FourWheeler();

    else return NULL;

}

Create objects in the library instead of the client so that the client does not need to recompile each time a new case is added.

<http://www.geeksforgeeks.org/design-patterns-set-2-factory-method/>

# Abstract Factory Pattern

The abstract factory pattern provides a way to encapsulate a group of individual factories that have a common theme without specifying their concrete classes.[1] In normal usage, the client software creates a concrete implementation of the abstract factory and then uses the generic interface of the factory to create the concrete objects that are part of the theme. The client doesn't know (or care) which concrete objects it gets from each of these internal factories, since it uses only the generic interfaces of their products.[1] This pattern separates the details of implementation of a set of objects from their general usage and relies on object composition, as object creation is implemented in methods exposed in the factory interface



# Design Pattern

<http://www.geeksforgeeks.org/software-design-patterns/>

# Singleton

// Thread Synchronized Java implementation of

// singleton design pattern

class Singleton

{

    private static Singleton obj;

    private Singleton() {}

    // Only one thread can execute this at a time

    public static synchronized Singleton getInstance()

    {

        if (obj==null)

            obj = new Singleton();

        return obj;

    }

}

# Decorator pattern:

Different pizza need toppings: choice 1:

// Sample getCost() in super class

public int getCost()

{

int totalToppingsCost = 0;

if (hasJalapeno() )

totalToppingsCost += jalapenoCost;

if (hasCapsicum() )

totalToppingsCost += capsicumCost;

// similarly for other toppings

return totalToppingsCost;

}

// Sample getCost() in subclass

public int getCost()

{

// 100 for Margherita and super.getCost()

// for toppings.

return super.getCost() + 100;

}

Make topping to be variable of the super class, and add calculation method in the super class. Problems:

1. New topping will force us to change the getCost() function in the super class.
2. Price changes will force us to change the getCost() function in the super class
3. Some toppings may not be appropriate for some pizza but they inherit them anyways.
4. Double topping will be hard to calculate

Decorator pattern:

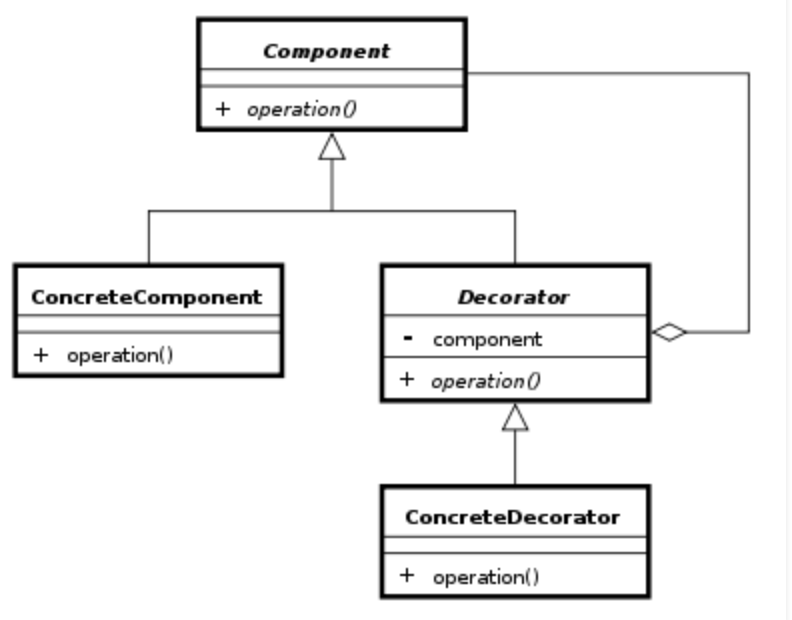
Call getCost() and use delegation instead of inheritance to calculate the toppings cost.

Decorators have the same super type as the object they decorate.

You can use multiple decorators to wrap an the object.

Since decorators have same type as object, we can pass around decorated object instead of original.

We can decorate objects at runtime.



Each component can be used on its own or may be wrapped by a decorator.

Each decorator has an instance variable that holds the reference to component it decorates(HAS-A relationship).

The ConcreteComponent is the object we are going to dynamically decorate.

Advantages:

The decorator pattern can be used to make it possible to extend (decorate) the functionality of a certain object at runtime.

The decorator pattern is an alternative to subclassing. Subclassing adds behavior at compile time, and the change affects all instances of the original class; decorating can provide new behavior at runtime for individual objects.

Decorator offers a pay-as-you-go approach to adding responsibilities. Instead of trying to support all foreseeable features in a complex, customizable class, you can define a simple class and add functionality incrementally with Decorator objects.

Disadvantages:

Decorators can complicate the process of instantiating the component because you not only have to instantiate the component, but wrap it in a number of decorators.

It can be complicated to have decorators keep track of other decorators, because to look back into multiple layers of the decorator chain starts to push the decorator pattern beyond its true intent.

# Strategy Design Pattern:

Some streetFighters will have jump and roll while all fighters will be able to kick and punch. We can move the kick and punch outside of the fighter base class and make it an interface. But how do we avoid repeating code? We can introduce interface for certain common methods.

**Advantages:**

1. A family of algorithms can be defined as a class hierarchy and can be used interchangeably to alter application behavior without changing its architecture.
2. By encapsulating the algorithm separately, new algorithms complying with the same interface can be easily introduced.
3. The application can switch strategies at run-time.
4. Strategy enables the clients to choose the required algorithm, without using a “switch” statement or a series of “if-else” statements.
5. Data structures used for implementing the algorithm are completely encapsulated in Strategy classes. Therefore, the implementation of an algorithm can be changed without affecting the Context class.

**Disadvantages:**

1. The application must be aware of all the strategies to select the right one for the right situation.
2. Context and the Strategy classes normally communicate through the interface specified by the abstract Strategy base class. Strategy base class must expose interface for all the required behaviours, which some concrete Strategy classes might not implement.
3. In most cases, the application configures the Context with the required Strategy object. Therefore, the application needs to create and maintain two objects in place of one.

// Java program to demonstrate implementation of

// Strategy Pattern

// Abstract as you must have a specific fighter

abstract class Fighter

{

    KickBehavior kickBehavior;

    JumpBehavior jumpBehavior;

    public Fighter(KickBehavior kickBehavior,

                   JumpBehavior jumpBehavior)

    {

        this.jumpBehavior = jumpBehavior;

        this.kickBehavior = kickBehavior;

    }

    public void punch()

    {

        System.out.println("Default Punch");

    }

    public void kick()

    {

        // delegate to kick behavior

        kickBehavior.kick();

    }

    public void jump()

    {

        // delegate to jump behavior

        jumpBehavior.jump();

    }

    public void roll()

    {

        System.out.println("Default Roll");

    }

    public void setKickBehavior(KickBehavior kickBehavior)

    {

        this.kickBehavior = kickBehavior;

    }

    public void setJumpBehavior(JumpBehavior jumpBehavior)

    {

        this.jumpBehavior = jumpBehavior;

    }

    public abstract void display();

}

// Encapsulated kick behaviors

interface KickBehavior

{

    public void kick();

}

class LightningKick implements KickBehavior

{

    public void kick()

    {

        System.out.println("Lightning Kick");

    }

}

class TornadoKick implements KickBehavior

{

    public void kick()

    {

        System.out.println("Tornado Kick");

    }

}

// Encapsulated jump behaviors

interface JumpBehavior

{

    public void jump();

}

class ShortJump implements JumpBehavior

{

    public void jump()

    {

        System.out.println("Short Jump");

    }

}

class LongJump implements JumpBehavior

{

    public void jump()

    {

        System.out.println("Long Jump");

    }

}

// Characters

class Ryu extends Fighter

{

    public Ryu(KickBehavior kickBehavior,

               JumpBehavior jumpBehavior)

    {

        super(kickBehavior,jumpBehavior);

    }

    public void display()

    {

        System.out.println("Ryu");

    }

}

class Ken extends Fighter

{

    public Ken(KickBehavior kickBehavior,

               JumpBehavior jumpBehavior)

    {

        super(kickBehavior,jumpBehavior);

    }

    public void display()

    {

        System.out.println("Ken");

    }

}

class ChunLi extends Fighter

{

    public ChunLi(KickBehavior kickBehavior,

                  JumpBehavior jumpBehavior)

    {

        super(kickBehavior,jumpBehavior);

    }

    public void display()

    {

        System.out.println("ChunLi");

    }

}

// Driver class

class StreetFighter

{

    public static void main(String args[])

    {

        // let us make some behaviors first

        JumpBehavior shortJump = new ShortJump();

        JumpBehavior LongJump = new LongJump();

        KickBehavior tornadoKick = new TornadoKick();

        // Make a fighter with desired behaviors

        Fighter ken = new Ken(tornadoKick,shortJump);

        ken.display();

        // Test behaviors

        ken.punch();

        ken.kick();

        ken.jump();

        // Change behavior dynamically (algorithms are

        // interchangeable)

        ken.setJumpBehavior(LongJump);

        ken.jump();

    }

}

# Command Design Pattern

Encapsulates a request as an object.

interface Command

{

    public void execute();

}

// Light class and its corresponding command

// classes

class Light

{

    public void on()

    {

        System.out.println("Light is on");

    }

    public void off()

    {

        System.out.println("Light is off");

    }

}

class LightOnCommand implements Command

{

    Light light;

    // The constructor is passed the light it

    // is going to control.

    public LightOnCommand(Light light)

    {

       this.light = light;

    }

    public void execute()

    {

       light.on();

    }

}

class LightOffCommand implements Command

{

    Light light;

    public LightOffCommand(Light light)

    {

        this.light = light;

    }

    public void execute()

    {

         light.off();

    }

}

class Stereo

{

    public void on()

    {

        System.out.println("Stereo is on");

    }

    public void off()

    {

        System.out.println("Stereo is off");

    }

    public void setCD()

    {

        System.out.println("Stereo is set " +

                           "for CD input");

    }

    public void setDVD()

    {

        System.out.println("Stereo is set"+

                         " for DVD input");

    }

    public void setRadio()

    {

        System.out.println("Stereo is set" +

                           " for Radio");

    }

    public void setVolume(int volume)

    {

       // code to set the volume

       System.out.println("Stereo volume set"

                          + " to " + volume);

    }

}

class StereoOffCommand implements Command

{

    Stereo stereo;

    public StereoOffCommand(Stereo stereo)

    {

        this.stereo = stereo;

    }

    public void execute()

    {

       stereo.off();

    }

}

class StereoOnWithCDCommand implements Command

{

     Stereo stereo;

     public StereoOnWithCDCommand(Stereo stereo)

     {

         this.stereo = stereo;

     }

     public void execute()

     {

         stereo.on();

         stereo.setCD();

         stereo.setVolume(11);

     }

}

// A Simple remote control with one button

class SimpleRemoteControl

{

    Command slot;  // only one button

    public SimpleRemoteControl()

    {

    }

    public void setCommand(Command command)

    {

        // set the command the remote will

        // execute

        slot = command;

    }

    public void buttonWasPressed()

    {

        slot.execute();

    }

}

// Driver class

class RemoteControlTest

{

    public static void main(String[] args)

    {

        SimpleRemoteControl remote =

                  new SimpleRemoteControl();

        Light light = new Light();

        Stereo stereo = new Stereo();

        // we can change command dynamically

        remote.setCommand(new

                    LightOnCommand(light));

        remote.buttonWasPressed();

        remote.setCommand(new

                StereoOnWithCDCommand(stereo));

        remote.buttonWasPressed();

        remote.setCommand(new

                   StereoOffCommand(stereo));

        remote.buttonWasPressed();

     }

  }

Advantages:

Makes our code extensible as we can add new commands without changing existing code.

Reduces coupling the invoker and receiver of a command.

Disadvantages:

Increase in the number of classes for each individual command

Definition: The command pattern encapsulates a request as an object, thereby letting us parameterize other objects with different requests, queue or log requests, and support undoable operations.

Remote control is the client and stereo, lights are the receivers. In command pattern there is a Command object that encapsulates a request by binding together a set of actions on a specific receiver. It does so by exposing just one method execute() that causes some actions to be invoked on the receiver.

Parameterizing other objects with different requests in our analogy means that the button used to turn on the lights can later be used to turn on stereo or maybe open the garage door.

queue or log requests, and support undoable operations means that Command’s Execute operation can store state for reversing its effects. The Command may have an added unExecute operation that reverses the effects of a previous call to execute.It may also support logging changes so that they can be reapplied in case of a system crash。

# Flyweight Pattern:

Want to decrease object count when we need a large number of similar objects. Use HashMap to store reference to the object created. Intrinsic vs extrinsic states:

<http://www.geeksforgeeks.org/flyweight-design-pattern/>

# Prototype Design Pattern:

Use existing objects to create new objects using clone methods

abstract class Color implements Cloneable

{

    protected String colorName;

    abstract void addColor();

    public Object clone()

    {

        Object clone = null;

        try

        {

            clone = super.clone();

        }

        catch (CloneNotSupportedException e)

        {

            e.printStackTrace();

        }

        return clone;

    }

}

class blueColor extends Color

{

    public blueColor()

    {

        this.colorName = "blue";

    }

    @Override

    void addColor()

    {

        System.out.println("Blue color added");

    }

}

class blackColor extends Color{

    public blackColor()

    {

        this.colorName = "black";

    }

    @Override

    void addColor()

    {

        System.out.println("Black color added");

    }

}

class ColorStore {

    private static Map<String, Color> colorMap = new HashMap<String, Color>();

    static

    {

        colorMap.put("blue", new blueColor());

        colorMap.put("black", new blackColor());

    }

    public static Color getColor(String colorName)

    {

        return (Color) colorMap.get(colorName).clone();

    }

}

// Driver class

class Prototype

{

    public static void main (String[] args)

    {

        ColorStore.getColor("blue").addColor();

        ColorStore.getColor("black").addColor();

        ColorStore.getColor("black").addColor();

        ColorStore.getColor("blue").addColor();

    }

}

# Façade

Shield implementation from the outside. You have a complex system and you want to expose to clients in a simplified way.

package structural.facade;

public interface Hotel

{

    public Menus getMenus();

}

Interface of hotel:

package structural.facade;

public interface Hotel

{

    public Menus getMenus();

}

Non veg restaurant

package structural.facade;

public class NonVegRestaurant implements Hotel

{

    public Menus getMenus()

    {

        NonVegMenu nv = new NonVegMenu();

        return nv;

    }

}

Hotel Keeper:

package structural.facade;

public class HotelKeeper

{

    public VegMenu getVegMenu()

    {

        VegRestaurant v = new VegRestaurant();

        VegMenu vegMenu = (VegMenu)v.getMenus();

        return vegMenu;

    }

    public NonVegMenu getNonVegMenu()

    {

        NonVegRestaurant v = new NonVegRestaurant();

        NonVegMenu NonvegMenu = (NonVegMenu)v.getMenus();

        return NonvegMenu;

    }

    public Both getVegNonMenu()

    {

        VegNonBothRestaurant v = new VegNonBothRestaurant();

        Both bothMenu = (Both)v.getMenus();

        return bothMenu;

    }

}

Client:

package structural.facade;

public class Client

{

    public static void main (String[] args)

    {

        HotelKeeper keeper = new HotelKeeper();

        VegMenu v = keeper.getVegMenu();

        NonVegMenu nv = keeper.getNonVegMenu();

        Both = keeper.getVegNonMenu();

    }

}

# Bridge Design



Without bridge pattern



With bridge pattern: decouple abstraction from implementation

abstract class Vehicle {

protected Workshop workShop1;

protected Workshop workShop2;

protected Vehicle(Workshop workShop1, Workshop workShop2)

    {

        this.workShop1 = workShop1;

        this.workShop2 = workShop2;

    }

    abstract public void manufacture();

}

// Refine abstraction 1 in bridge pattern

class Car extends Vehicle {

public Car(Workshop workShop1, Workshop workShop2)

    {

        super(workShop1, workShop2);

    }

    @Override public void manufacture()

    {

        System.out.print("Car ");

        workShop1.work();

        workShop2.work();

    }

}

// Refine abstraction 2 in bridge pattern

class Bike extends Vehicle {

public Bike(Workshop workShop1, Workshop workShop2)

    {

        super(workShop1, workShop2);

    }

    @Override public void manufacture()

    {

        System.out.print("Bike ");

        workShop1.work();

        workShop2.work();

    }

}

// Implementor for bridge pattern

interface Workshop

{

    abstract public void work();

}

// Concrete implementation 1 for bridge pattern

class Produce implements Workshop {

    @Override public void work()

    {

        System.out.print("Produced");

    }

}

// Concrete implementation 2 for bridge pattern

class Assemble implements Workshop {

    @Override public void work()

    {

        System.out.print(" And");

        System.out.println(" Assembled.");

    }

}

// Demonstration of bridge design pattern

class BridgePattern {

public static void main(String[] args)

    {

        Vehicle vehicle1 = new Car(new Produce(), new Assemble());

        vehicle1.manufacture();

        Vehicle vehicle2 = new Bike(new Produce(), new Assemble());

        vehicle2.manufacture();

    }

}

# Composite:

“Compose objects into tree structure to represent **part-whole hierarchies**. Composite lets client treat individual objects and compositions of objects uniformly”.

Interface: employee. Developer and Manager implement the interface. Company directory implement employee and has list<Employee>