**Design pattern**

By definition, Design Patterns are reusable solutions to commonly occurring problems (in the context of software design).

Design patterns represent the best practices used by experienced object-oriented software developers. Design patterns are solutions to general problems that software developers faced during software development. These solutions were obtained by trial and error by numerous software developers over quite a substantial period of time.

**Design Principles:**

Software design principles represent a set of guidelines that helps us to avoid having a bad design.

Software design phase can be divided into three step process.

**1.Describe individual module functionality.**

**2.Determine relation between modules.**

**3.Decompose the system into modules.**

**S.O.L.I.D principle.**

**Single Responsibility Principle(SRP):**

The Single Responsibility Principle states that there should never be more than one reason for a class to change.

This means that every class /structure in your code should have only one job to do. Everything in the class should be related to that single purpose.

It doesn’t mean that your classes should only contain one method or property.

Example  
class CalculateAndSave

{

//calculation code;

// write in XML/PDF format code;

};

This class have two responsibilities, calculate and save, If we change either of calculate or file write logic then this class need to change completely.

Decompose the CalculateAndSave into two class

Class Calculate

{

//calculation code;

};

Class FileWrite

{

//write to file XML/PDF code;

};

**Open/Closed Principle(OCP):**

The Open/Closed Principle states that classes should be open for extension but closed for modification.

“Open to extension” means that you should design your classes so that new functionality can be added as new requirements are generated.

“Closed for Modification” means that once you have developed a class you should never modify it, except to correct bugs.

Example:

Class DrawEditor

{

void drawRectangle(Rectangle ob);

void drawCircle(Circle ob);

};

It violates Open/Closed Principle. Because new shape added then it needs to change DrawEditor class.

Class Shape

{

public:

virtual void draw()=0; //interface

};

Class Rectangle: public Shape

{

public:

void draw()

{

//rectangle draw code;

}

};

Class Circle: public Shape

{

public:

void draw()

{

//circle draw code ;

}

};

Void DrawEditor

{

public:

void draw()

{

s->draw();

}

DrawEditor(shape\* s\_shape);

private:

shape\* s;

};

**Liskov's Substitution Principle(LSP)**

In OOP we create classes then extend some of classes creating new classes. Sometimes while extending a class the new derived class changes the behaviour of the base class.

Liskov substitution principle states that derived classes should not change the expected behaviour of base class.

In simple terms ,if we create a derived class for a base class ,it should properly reflect the base class and extend it without replacing the functionality of old class.

Example of Liskov’s Substitution Principle violation.

class Rectangle

{

protected:

int m\_length;

int m\_width;

public:

virtual void setLength(int w\_length)

{

m\_length=w\_length;

}

virtual void setWidth(int w\_width);

{

m\_width=w\_width;

}

};

Class Square:public Rectangle

{

public:

void setLength(int w\_length)

{

m\_length=m\_width=w\_length;

}

Void setWidth(int w\_width)

{

M\_length=m\_width=w\_width;

}

};

Rectangle and Square. Let's assume that the Rectangle object is used somewhere in the application. We extend the application and add the Square class. The square class is returned by a factory pattern, based on some conditions and we don't know the exact what type of object will be returned. But we know it's a Rectangle. We get the rectangle object, set the width to 5 and height to 10 and get the area. For a rectangle with width 5 and height 10 the area should be 50. Instead the result will be 100

**Interface Segregation Principle(ISP):**

The Interface segregation principle (ISP) states that clients should not be forced to depend upon interface that they don’t use.

Instead of one fat interface many small interfaces are preferred based on groups of methods, each one serving one submodule.

Example:

Below is an example which violates the Interface Segregation Principle. We have a Manager class which represent the person which manages the workers. And we have 2 types of workers some average and some very efficient workers. Both types of workers works and they need a daily lunch break to eat. But now some robots came in the company they work as well , but they don't eat so they don't need a lunch break. One on side the new Robot class need to implement the IWorker interface because robots works. On the other side, the don't have to implement it because they don't eat.

class Iworker

{

public:

virtual void work()=0;

virtual void eat()=0;

};

class worker:public Iworker

{

public:

void work()

{

//work time calculation code

}

void eat()

{

//lunch break time code

}

};

class superWorker:public Iworker

{

Public:

void work()

{

//work time calculation code

}

void eat()

{

//lunch break time code;

}

};

class Manager

{

Iworker\* iw;

public:

void setworker(Iworker m\_worker)

{

iw=&m\_worker;

}

void manage()

{

iw->work();

iw->eat();

}

};

If we keep the present design, the new Robot class is forced to implement the eat method. We can write a dummy class which does nothing(let's say a launch break of 1 second daily), and can have undesired effects in the application(For example the reports seen by managers will report more lunches taken than the number of people).

Following it's the code supporting the Interface Segregation Principle. By splitting the IWorker interface in 2 different interfaces the new Robot class is no longer forced to implement the eat method. Also if we need another functionality for the robot like recharging we create another interface IRechargeble with a method recharge.

class Iworkable

{

public:

virtual void work()=0;

};

class Ifeedable

{

public:

virtual void eat()=0;

};

class worker: public Iworkable,public Ifeedable

{

void work()

{

//work duration code

}

void eat()

{

//lunch break time code

}

};

class robot: public Iworkable

{

void work()

{

//work duration code;

}

};

**Dependency Inversion Principle:**

The Dependency Inversion Principle states that high level modules should not depend upon low level modules.

Abstraction should not depend upon details.

Details should depend upon Abstraction.

Example:

Class Manager

{

Developer dev;

Designer desg;

Tester test;

public:

void assignDevWork();

void assignDesgWork();

void assignTestWork()

{

Test.work();

}

void addDeveloper();

void addDesigner();

void addTester();

};

Here we have exposed everything about the lower layer to the upper layer ,thus abstraction is not mentioned. That means Manager must already know about the type of the workers that he can supervise.

Now if another type of worker comes under the manager lets say, QA (quality assurance), then the whole class needs to be change. This is where dependency inversion principle violated.

Let us design again.

class Employee

{

public:

virtual void work();

};

class Developer: public Employee

{

public:

void work();

};

class Designer: public Employee

{

public:

void work();

};

class Tester: public Employee

{

public:

void work();

};

class Manager

{

Employee list[10];

public:

void addEmployee(Developer );

void assignWork();

}

if any other kind of the employee is added it can be simply be added to Manager without making the manager explicitly aware of it. Now to add another class of employee we can simply call.

The creation of the abstraction between different employees and Manager has resulted in very good looking design code which is easily maintainable and extendable.

**Creational design pattern:**

These Design patterns are all about class instantiation.

Below are the patterns that comes under creational design patterns.

1.Singleton Design Pattern

2. Factory Method Pattern

3.Object Pool Pattern.

**Singleton Design pattern:**

Singleton Design pattern is one of simplest design pattern widely used in many OOP applications.

Since there is only one instance of resource, if multiple applications require access to the only one instance, one application will have to wait, until the other application release the instance .

Sometimes it's important to have only one instance for a class. For example, in a system there should be only one window manager (or only a file system or only a print spooler). Usually singletons are used for centralized management of internal or external resources and they provide a global point of access to themselves.

#include <iostream>

using namespace std;

class Singleton

{

public:

static Singleton \*getInstance();

private:

Singleton(){}

static Singleton\* instance;

};

Singleton\* Singleton::instance = 0;

Singleton\* Singleton::getInstance()

{

if(!instance) {

instance = new Singleton();

cout << "getInstance(): First instance\n";

return instance;

}

else {

cout << "getInstance(): previous instance\n";

return instance;

}

}

int main()

{

Singleton \*s1 = Singleton::getInstance();

Singleton \*s2 = Singleton::getInstance();

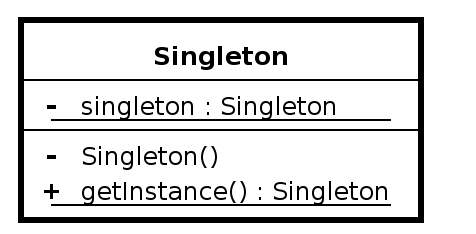
return 0;

}

Output:

getInstance(): First instance

getInstance(): previous instance



**Fig: UML Class diagram**

Real situations where the singleton is used:

Example 1: **Logger Classes**

The Singleton pattern is used in the design of logger classes. This classes are usually implemented as a singletons, and provides a global logging access point in all the application components without being necessary to create an object each time a logging operations is performed.

Example 2: **Configuration Classes**

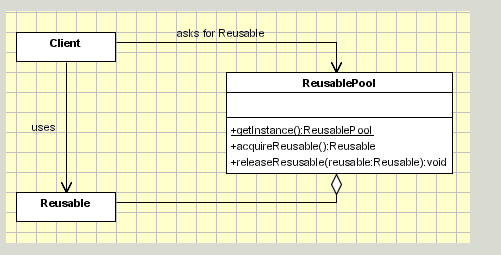
The Singleton pattern is used to design the classes which provides the configuration settings for an application. By implementing configuration classes as Singleton not only that we provide a global access point, but we also keep the instance we use as a cache object. When the class is instantiated ( or when a value is read ) the singleton will keep the values in its internal structure. If the values are read from the database or from files this avoids the reloading the values each time the configuration parameters are used.

**Singleton class is not thread safe**.

**Object pool pattern.**

Object pool pattern is a software creational design pattern which is used in situations where the cost of initializing a class instance is very high or limited.

Basically, an Object pool is a container which contains some amount of objects. So, when an object is taken from the pool, it is not available in the pool until it is put back.



Implementation involves the following objects:

Client - uses an instance of type Reusable.  
Reusable - Wraps the limited resource, will be shared by several clients for a limited amount of time.  
ReusablePool - manage the reusable objects for use by Clients, creating and managing a pool of objects.

When a client asks for a Reusable object, the pool performs the following actions:

-    Search for an available Reusable object and if it was found it will be returned to the client.  
-    If no Reusable object was found then it tries to create a new one. If this actions succeeds the new Reusable object will be returned to the client.  
-    If the pool was unable to create a new Reusable, the pool will wait until a reusable object will be released.  
  
The Client is responsible to request the Reusable object as well to release it to the pool. If this action will not be performed the Reusable object will be lost, being considered unavailable by the ResourcePool.

Example:

#include<iostream>

#include<vector>

using namespace std;

class resource

{

int i;

public:

resource(int data=0):i(data)

{}

void setdata(int data)

{

i=data;

}

int getdata()

{

return i;

}

friend ostream& operator<<(ostream& os,const resource& ob)

{

return os<<ob.i;

}

};

void display(vector<resource\*>&pvc)

{

for(auto it:pvc)

{

cout<<\*it<<" ";

}

cout<<endl;

}

class objectpool

{

vector<resource\*>&vc;

static objectpool\* instance;

objectpool(vector<resource\*>&rvec):vc(rvec)

{}

public:

static objectpool\* getintsance(vector<resource\*>&ovec)

{

instance=new objectpool(ovec);

return instance;

}

resource\* getresource() ///geting resource from pool

{

if(vc.size())

{

resource\* ptr=vc[0];

vc.erase(vc.begin());

return ptr;

}

else{

resource\* ptr=new resource(1);

vc.push\_back(ptr);

return ptr;

}

}

void releaseresource(resource\* ptr)

{

vc.push\_back(ptr);

}

};

objectpool\* objectpool::instance=NULL;

int main()

{

vector<resource\*>lvc;

for(int i=0;i<5;i++)

{

resource\* ptr=new resource(15); ////////////initialize the pool

lvc.push\_back(ptr);

}

cout<<"original reusable resource : "<<"\n";

display(lvc);////displaying original pool

cout<<"------------------------"<<endl;

objectpool \*mypool=objectpool::getintsance(lvc);

resource\* rptr;

rptr=mypool->getresource();

cout<<"geting resource from reusable: "<<"\n";

cout<<\*rptr<<endl; //displaying the resource

rptr->setdata(90); //modifying the resource

cout<<"reusable resource after aquire by reusable pool: "<<"\n";

display(lvc); //displaying pool after geting resource

cout<<"---------------------------------"<<endl;

mypool->releaseresource(rptr);

cout<<"returning resource to reusable :"<<"\n";

display(lvc); // displaying the pool after release the resource to pool

}

Output:

original reusable resource :

15 15 15 15 15

------------------------

geting resource from reusable:

15

reusable resource after aquire by reusable pool:

15 15 15 15

---------------------------------

returning resource to reusable :

15 15 15 15 90

Example:

**Connection pool**

**Factory Method Pattern:**

Factory Method is to creating objects as Template Method is to implementing an algorithm. A superclass specifies all standard and generic behaviour (using pure virtual “placeholders” for creation steps), and then delegates the creation details to subclasses that are supplied by the client.

Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses**.**

**Example**

#include<iostream>

#include<vector>

using namespace std;

class shape

{

public:

static shape\* makeShape(int choice); //factory method

virtual void display()=0;

};

class rectangle:public shape

{

public:

void display()

{

cout<<"rectangle here "<<endl;

}

};

class circle:public shape

{

public:

void display()

{

cout<<"circle here "<<endl;

}

};

class sqaure:public shape

{

public:

void display()

{

cout<<"square here "<<endl;

}

};

shape\* shape::makeShape(int choice) ///factory method

{

switch(choice)

{

case 1:

return new sqaure;

break;

case 2:

return new rectangle;

break;

case 3:

return new circle;

break;

default:

{

cout<<"invalid option "<<endl;

return NULL;

}

}

}

int main()

{

vector<shape\*> vc;

shape\* ptr;

for(int i=1;i<=3;i++)

{

ptr=shape::makeShape(i);

vc.push\_back(ptr);

}

for(int i=0;i<vc.size();i++)

{

vc[i]->display();

}

}

Also known as Virtual Constructor, the Factory Method is related to the idea on which libraries work: a library uses abstract classes for defining and maintaining relations between objects. One type of responsibility is creating such objects. The library knows when an object needs to be created, but not what kind of object it should create, this being specific to the application using the library.

**Structural Design pattern**

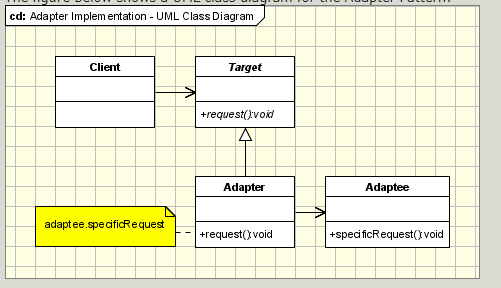
Structural pattern deal with how classes and objects are composed to form large structure.

This pattern are particularly useful for making independently developed class libraries work together.

**Adapter Pattern:**

Adapter pattern Convert the interface of a class into another interface clients expect.

Adapter pattern let two incompatible interfaces work together.



The classes/objects participating in adapter pattern:

**Target** - defines the domain-specific interface that Client uses.

**Adapter** - adapts the interface Adaptee to the Target interface.

**Adaptee** - defines an existing interface that needs adapting.

**Client** - collaborates with objects conforming to the Target interface.

**Example:**

#include <iostream>

using namespace std;

typedef int Coordinate;

typedef int Dimension;

// Desired interface

class Rectangle

{

public:

virtual void draw() = 0;

};

// Legacy component

class LegacyRectangle

{

public:

LegacyRectangle(Coordinate x1, Coordinate y1, Coordinate x2, Coordinate y2)

{

x1\_ = x1;

y1\_ = y1;

x2\_ = x2;

y2\_ = y2;

cout << "LegacyRectangle: create. (" << x1\_ << "," << y1\_ << ") => ("

<< x2\_ << "," << y2\_ << ")" << endl;

}

void oldDraw()

{

cout << "LegacyRectangle: oldDraw. (" << x1\_ << "," << y1\_ <<

") => (" << x2\_ << "," << y2\_ << ")" << endl;

}

private:

Coordinate x1\_;

Coordinate y1\_;

Coordinate x2\_;

Coordinate y2\_;

};

// Adapter wrapper

class RectangleAdapter: public Rectangle, private LegacyRectangle

{

public:

RectangleAdapter(Coordinate x, Coordinate y, Dimension w, Dimension h):

LegacyRectangle(x, y, x + w, y + h)

{

cout << "RectangleAdapter: create. (" << x << "," << y <<

"), width = " << w << ", height = " << h << endl;

}

virtual void draw()

{

cout << "RectangleAdapter: draw." << endl;

oldDraw();

}

};

int main()

{

Rectangle \*r = new RectangleAdapter(120, 200, 60, 40);

r->draw();

}

Output:

-----------

LegacyRectangle: create. (120,200) => (180,240)

RectangleAdapter: create. (120,200), width = 60, height = 40

RectangleAdapter: draw.

LegacyRectangle: oldDraw. (120,200) => (180,240)

**Another Example:**

#include <iostream>

using namespace std;

class ExecuteInterface {

public:

// Specify the new interface

virtual ~ExecuteInterface(){}

virtual void execute() = 0;

};

// Design a "wrapper" or "adapter" class

template <class TYPE>

class ExecuteAdapter: public ExecuteInterface {

public:

ExecuteAdapter(TYPE \*o, void(TYPE:: \*m)()) {

object = o;

method = m;

}

~ExecuteAdapter() {

delete object;

}

// The adapter/wrapper "maps" the new to the legacy implementation

void execute() { /\* the new \*/

(object->\*method)();

}

private:

TYPE \*object; // ptr-to-object attribute

void(TYPE:: \*method)(); /\* the old \*/ // ptr-to-member-function attribute

};

// The old: three totally incompatible classes

// no common base class,

class Fea {

public:

// no hope of polymorphism

~Fea() {

cout << "Fea::dtor" << endl;

}

void doThis() {

cout << "Fea::doThis()" << endl;

}

};

class Feye {

public:~Feye() {

cout << "Feye::dtor" << endl;

}

void doThat() {

cout << "Feye::doThat()" << endl;

}

};

class Pheau {

public:

~Pheau() {

cout << "Pheau::dtor" << endl;

}

void doTheOther() {

cout << "Pheau::doTheOther()" << endl;

}

};

/\* the new is returned \*/

ExecuteInterface \*\*initialize() {

ExecuteInterface \*\*array = new ExecuteInterface \*[3];

/\* the old is below \*/

array[0] = new ExecuteAdapter < Fea > (new Fea(), &Fea::doThis);

array[1] = new ExecuteAdapter < Feye > (new Feye(), &Feye::doThat);

array[2] = new ExecuteAdapter < Pheau > (new Pheau(), &Pheau::doTheOther);

return array;

}

int main() {

ExecuteInterface \*\*objects = initialize();

for (int i = 0; i < 3; i++) {

objects[i]->execute();

}

// Client uses the new (polymporphism)

for (int i = 0; i < 3; i++) {

delete objects[i];

}

delete objects;

return 0;

}

Output:

Fea::doThis()

Feye::doThat()

Pheau::doTheOther()

Fea::dtor

Feye::dtor

Pheau::dtor

**Behavioural patterns**

In software engineering, behavioural design patterns are design patterns that identify common communication patterns between objects and realize these patterns. By doing so, these patterns increase flexibility in carrying out this communication.

**Observer Design Pattern**

**Intent**

* Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.
* Encapsulate the core (or common or engine) components in a Subject abstraction, and the variable (or optional or user interface) components in an Observer hierarchy.

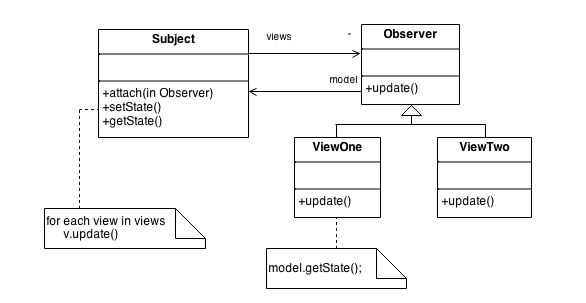
Define an object that is the "keeper" of the data model and/or business logic (the Subject). Delegate all "view" functionality to decoupled and distinct Observer objects. Observers register themselves with the Subject as they are created. Whenever the Subject changes, it broadcasts to all registered Observers that it has changed, and each Observer queries the Subject for that subset of the Subject's state that it is responsible for monitoring.

This allows the number and "type" of "view" objects to be configured dynamically, instead of being statically specified at compile-time.

The protocol described above specifies a "pull" interaction model. Instead of the Subject "pushing" what has changed to all Observers, each Observer is responsible for "pulling" its particular "window of interest" from the Subject. The "push" model compromises reuse, while the "pull" model is less efficient.

Issues that are discussed, but left to the discretion of the designer, include: implementing event compression (only sending a single change broadcast after a series of consecutive changes has occurred), having a single Observer monitoring multiple Subjects, and ensuring that a Subject notify its Observers when it is about to go away.

The Observer pattern captures the lion's share of the Model-View-Controller architecture that has been a part of the Smalltalk community for years.



UML class diagram

**Example:**

#include <iostream>

#include<vector>

using namespace std;

class Observer

{

public:

virtual void update(int value) = 0;

};

class Subject

{

int m\_value;

vector<Observer\*> m\_views;

public:

void attach(Observer \*obs)

{

m\_views.push\_back(obs);

}

void set\_val(int value)

{

m\_value = value;

notify();

}

void notify()

{

for (int i = 0; i < m\_views.size(); ++i)

m\_views[i]->update(m\_value);

}

};

class DivObserver: public Observer

{

int m\_div;

public:

DivObserver(Subject \*model, int div)

{

model->attach(this);

m\_div = div;

}

/\* virtual \*/void update(int v)

{

cout << v << " div " << m\_div << " is " << v / m\_div << '\n';

}

};

class ModObserver: public Observer

{

int m\_mod;

public:

ModObserver(Subject \*model, int mod)

{

model->attach(this);

m\_mod = mod;

}

/\* virtual \*/void update(int v)

{

cout << v << " mod " << m\_mod << " is " << v % m\_mod << '\n';

}

};

int main()

{

Subject subj;

DivObserver divObs1(&subj, 4);

DivObserver divObs2(&subj, 3);

ModObserver modObs3(&subj, 3);

subj.set\_val(14);

}