Design patterns with C++

Genesis InSoft Limited

A name you can trust

1-7-1072/A, Opp. Saptagiri Theatre, RTC * Roads, Hyderabad - 500 020

Design Patterns with C++

Genesis Computers (A unit of Genesis InSoft Limited) started its operations on March 16th 1992 in Hyderabad, India primarily as a centre for advanced software education, development and consultancy.
Training is imparted through lectures supplemented with on-line demonstrations using audio visual aids. Seminars, workshops and demonstrations are organized periodically to keep the participants abreast of the latest technologies. Genesis InSoft Ltd. is also involved in software development and consultancy.
We have implemented projects/training in technologies ranging from client server applications to web based. Skilled in PowerBuilder, Windows C SDK, VC++, C++, C, Visual Basic, Java, J2EE, XML, WML, HTML, UML, Java Script, MS.NET (C#, VB.NET and ASP.NET, ADO.NET etc), PHP, Jhoomla, Zend Framework, JQuery, ExtJS, PERL, Python, TCL/TK etc. using Oracle, MySql and SQL Server as backend databases.

Genesis has earned a reputation of being in forefront on Technology and is ranked amongst the top training institutes in Hyderabad city. The highlight of Genesis is that new emerging technologies are absorbed quickly and applied in its areas of operation.
We have on our faculty a team of highly qualified and trained professionals who have worked both in India and abroad. So far, we have trained about 55,000+ students who were mostly engineers and experienced computer professionals themselves.

Tapadia (MS, USA), the founder of Genesis Computers, has about 35+ years' experience in software industry. He worked for OMC computers, Intergraph India Private Ltd., Intergraph Corporation (USA), and was a consultant to D.E. Shaw India Software Private Ltd and iLabs Limited.
He is currently the Finishing School Director of KMIT, NGIT, KMEC, KMCE (Keshav Memorial Group of Institutions).

He has more than 35 years of teaching experience, and has conducted training for the corporates like ADP, APSRTC, ARM, B.H.E.L, B2B Software Technologies, Cambridge Technology Enterprises Private Limited, CellExchange India Private Limited, Citicorp Overseas Software Limited, CMC Centre (Gachibowli, Posnett Bhavan), CommVault Systems (India) Private Limited, Convergys Information Management (India) Private Limited, D.E. Shaw India Software Private Limited, D.R.D.L, Dell EMC, Bangalore (behalf of DevelopIntelligence, USA), Deloitee Consulting India Private Limited, ELICO Limited, eSymbiosis ITES India Private Limited, Everypath Private Limited, Gold Stone Software, HCL Consulting (Chennai), iLabs Limited, Infotech Enterprises, Intelligroup Asia Private Limited, Intergraph India Private Limited, Invensys Development Centre India Private Limited, Ivy Comptech, JP Systems (India) Limited, Juno Online Services Development Private Limited, Malpani Soft Private Limited, Mars Telecom Systems Private Limited, Mentor Graphics India Private Limited, Motorola India Electronics Limited, NCR Corporation India Private Limited, Netrovert Software Private Limited, Nokia India Private Limited, Optima Software Services, Oracle India Private Limited, Polaris Software Lab Limited, Pluralsight (USA), Qualcomm India Private Limited (Chennai, Hyderabad, Bangalore), Qualcomm China, Quantum Softech Limited, R.C.I, Renaissance Infotech, Satyam Computers, Satyam GE, Satyam Learning Centre, SIS Software (India) Private Limited, Sriven Computers, Teradata - NCR, Tanla Solutions Limited, Timmins Training Consulting Sdn. Bhd, Kuala Lumpur, Vazir Sultan Tobacco, Verizon, Virtusa India Private Limited, Wings Business Systems Private Limited, Wipro Systems, Xilinx India Technology Services Private Limited, Xilinx Ireland, Xilinx Inc (San Jose).

Genesis InSoft Limited
1-7-1072/A, RTC * Roads, Hyderabad - 500 020

rtapadia@genesisinsoft.com
www.genesisinsoft.com

Introduction

A design pattern is a general repeatable solution to a commonly occurring problem in software design. A design pattern isn't a finished design that can be transformed directly into code. It is a description or template for how to solve a problem that can be used in many different situations.

Why keep reinventing the wheel? Why not just write down your solution and refer back to it as needed? That's what design patterns are all about.

A design pattern is a tested solution to a standard programming problem. Once you learn design patterns, if you face a programming issue, a solution will come to you more quickly. What you will probably need is the Factory pattern." Or the Observer pattern. Or the Adapter pattern. Someone has already faced the problem you're facing and has come up with a solution that implements all kinds of good design.

Design patterns can speed up the development process by providing tested, proven development paradigms. Effective software design requires considering issues that may not become visible until later in the implementation. Reusing design patterns helps to prevent subtle issues that can cause major problems and improves code readability for coders and architects familiar with the patterns.

The charm of knowing about design patterns is that it makes your solution easily reusable, extendable, and maintainable. When you're working on a programming problem, the tendency is to program to the problem, not in terms of reuse, extensibility, maintainability, or other good design issues.

Patterns describe common ways of doing things. They are collected by people who spot repeating themes in designs. These people take each theme and describe it so that other people can read the pattern and see how to apply it.

The power of shared pattern vocabulary

Shared pattern vocabulary is powerful: When you communicate with your teams using patterns, you are communicating not just a pattern name but a whole set of qualities, characteristic and constraints that the pattern represents.

Patterns allow you to say more with less: When you use a pattern in a description, other developers quickly know precisely the design you have in mind.

Talking at the pattern level allows you to stay "in the design" longer: Talking about software systems using patterns allows you to keep the discussion at the design level, without having to get into the details of implementing objects and classes.

Inheritance is fundamental to object-oriented programming. A programming language may have objects and messages, but without inheritance it is not object-oriented (merely "object-based", but still polymorphic).

Composition is also fundamental to every language (in terms of parts and components). It would be impossible to break down complex problems into modular solutions without composition. Composition is fairly easy to understand - we can see composition in everyday life: a chair has

legs, a wall is composed of bricks and mortar. Computer system has a hard disk, memory. Car has an Engine, chassis, and steering wheels.

The set of 23 standard design patterns was published by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides in their book Design Patterns: Elements of Reusable Object-Oriented Software. They've come to be known in programming circles as the Gang of Four, or, more popularly, GoF. According to these authors design patterns are primarily based on the following principles of object oriented design.

- Program to an interface not an implementation
  - Interface is a contract between an object and its clients. An interface specifies the things that an object can do, and the signatures for accessing those things. Implementations are the actual behaviors. For example you have a method sort(). You can implement QuickSort or MergeSort. That should not matter to the client code calling sort as long as the interface does not change.

    Coding against interface means, the client code always holds an Interface object which is supplied by a factory. Any instance returned by the factory would be of type Interface which any factory candidate class must have implemented. This way the client program is not worried about implementation and the interface signature determines what all operations can be done. This can be used to change the behavior of a program at run-time.

- Favor object composition over inheritance
  - It is a design principle that gives the design higher flexibility. It is more natural to build business-domain classes out of various components than trying to find commonality between them and creating a family tree. For example, a pedal and a wheel share very few common traits, yet are both vital components in a car. What they can do and how they can be used to benefit the car is easily defined. In other words, it is better to compose what an object can do (HAS-A) than extend what it is (IS-A).

    Initial design is simplified by identifying system object behaviors in separate interfaces instead of creating a hierarchical relationship to distribute behaviors among business-domain classes via inheritance. This approach more easily accommodates future requirements changes that would otherwise require a complete restructuring of business-domain classes in the inheritance model. Additionally, it avoids problems often associated with relatively minor changes to an inheritance-based model that includes several generations of classes.

    One common drawback of using composition instead of inheritance is that methods being provided by individual components may have to be implemented in the derived type, even if they are only forwarding methods. In contrast, inheritance does not require all of the base class's methods to be re-implemented within the derived class.

  Inheritance should only be used when:
  - Both classes are in the same logical domain
  - The subclass is a proper subtype of the superclass
  - The superclass's implementation is necessary or appropriate for the subclass
  - The enhancements made by the subclass are primarily additive.

Design patterns are classified by two criteria.

1. The first criterion, called purpose, reflects what a pattern does, which can be Creational, Structural, or Behavioral.

2. The second criterion, called scope, specifies whether the pattern applies primarily to classes or objects. Class patterns deal with relationships between classes and their subclasses. These relationships are established through inheritance, so they are static - fixed at compile time. Object patterns deal with object relationships, which can be changed at run time and are more dynamic.

Creational patterns

Creational patterns are used to create objects for a suitable class that serves as a solution for a problem. They are particularly useful when you are taking advantage of polymorphism and need to choose between different classes at runtime rather than compile time.

Structural patterns

These patterns are concerned with how classes and objects are composed to form larger structures. These design patterns concern class and object composition. Concept of inheritance is used to compose interfaces and define ways to compose objects to obtain new functionalities.
How do you built a software component?

Behavioral patterns

Behavioral patterns describe interactions between objects and focus on how objects communicate with each other. Behavioral patterns are concerned with algorithms and the assignment of responsibilities between objects.

How do you want to run a behavior in Software?

		Purpose		
		Creational	Structural	Behavioral
Scope	Class	Factory Method	Adapter (class)	Interpreter Template Method
	Object	Abstract factory Builder Prototype Singleton	Adapter (object) Bridge Composite Decorator Façade Flyweight Proxy	Chain of responsibility Command Iterator Mediator Memento Observer State Strategy Visitor

Design pattern space

Purpose	**Design Pattern**	**Aspects that can vary**
Creational	Abstract Factory	Provides an interface for creating families of related or dependent objects without specifying their concrete classes.

	Builder	Separate the construction of a complex object from its representation so that the same construction process can create different representation.
	Factory Method	Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory method lets a class defer instantiation to subclasses.
	Prototype	Specify the kinds of objects to create using a prototypical instance, and create new objects by copying their prototype.
	Singleton	Ensure a class only has one instance, and provide a global point of access to it.
Structural	Adapter	Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.
	Bridge	Decouple an abstraction from its implementation so that the two can vary independently.
	Composite	Compose objects into tree structure to represent part-whole hierarchies. Composite lets clients treat individual objects and composition of objects uniformly.
	Decorator	Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to sub classing for extending functionality.
	Facade	Provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher level interface that makes the subsystem easier to use.
	Flyweight	Use sharing to support large number of fine grained objects efficiently.
	Proxy	Provide a surrogate or placeholder for another object to control access to it.
Behavioral	Chain of responsibility	Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it.
	Command	Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.
	Interpreter	Given a language, define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in its language,
	Iterator	Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.
	Mediator	Define an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently.
	Memento	Without violating encapsulation, capture and externalize an object's internal state so that the object can be restored to this state later.

	Observer	Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.
	State	Allow an object to alter its behavior when its internal state changes. The object will appear to change its class.
	Strategy	Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.
	Template Method	Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.
	Visitor	Represent an operation to be performed on the elements of an object structure. Visitors lets you define a new operation without changing the classes of the elements on which it operates.

Design aspects that design patterns let you vary

Creational patterns: Singleton

It is important for some classes to have exactly one instance. Although there can be many printers in a system, there should be only one printer spooler. There should be only one file system and one window manager.

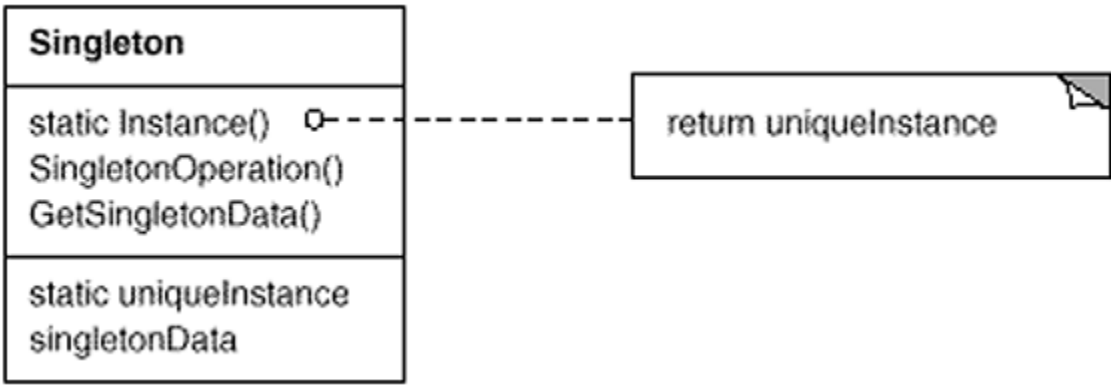
How do we ensure that a class has only one instance and that the instance is easily accessible? A global variable makes an object accessible, but it doesn't keep you from instantiating multiple objects.

A better solution is to make the class itself responsible for keeping track of its sole instance. The class can ensure that no other instance can be created, and it can provide a way to access the instance. This is the singleton pattern.

Use the singleton pattern when

- There must be exactly one instance of a class, and it must be accessible to clients.
- When the sole instance should be extensible by subclassing, and clients should be able to use extended instance without modifying their code.

Structure



Participants

Singleton

- defines an Instance operation that lets clients access its unique instance. Instance is a class operation (that is a static member function in C++).

- may be responsible for creating its own unique instance.

Code listing 4 (Singleton.cpp)

```
#include <iostream>
using namespace std;

class singleton
{
   private:
      int value;
      singleton();
      static singleton* _instance;
   public:
      int getValue();
      void setValue(int val);
      static singleton* getInstance();
      static void DestroyInstance();
   protected:
      // To disallow copy constructor, assignment operator
      // and destructor to be called from outside the class
      singleton(const singleton&) {};
      singleton& operator=(const singleton&){};
      ~singleton(){};
};

singleton* singleton :: _instance = NULL;

singleton :: singleton() {
   value = 20;
}

int singleton :: getValue() {
```

```
   return value;
}

void singleton :: setValue(int val) {
   value = val;
}

singleton * singleton :: getInstance() {
   if (_instance == NULL)  {
      cout << "if block" << endl;
      _instance = new singleton();
   }
   else {
      cout << "else block" << endl;
   }
   return _instance;
}
void singleton::DestroyInstance() {
  delete _instance;
  _instance = NULL;
}

void main() {
   singleton *s1 = singleton::getInstance();
   s1->setValue(40);
   cout << "Value 1 " << s1->getValue() << endl;

   singleton *s2 = singleton::getInstance();
   cout << "Value 2 " << s2->getValue() << endl;

   s2->setValue(50);
   cout << "Value 1 " << s1->getValue() << endl;
   cout << "Value 2 " << s2->getValue() << endl;

  singleton::DestroyInstance();

  singleton *s3 = singleton::getInstance();
  cout << "Value 1 " << s1->getValue() << endl;
  cout << "Value 2 " << s2->getValue() << endl;

  cout << "Value 3 " << s3->getValue() << endl;

  // *s1 = *s2;
  // singleton s4(*s1);
}
```

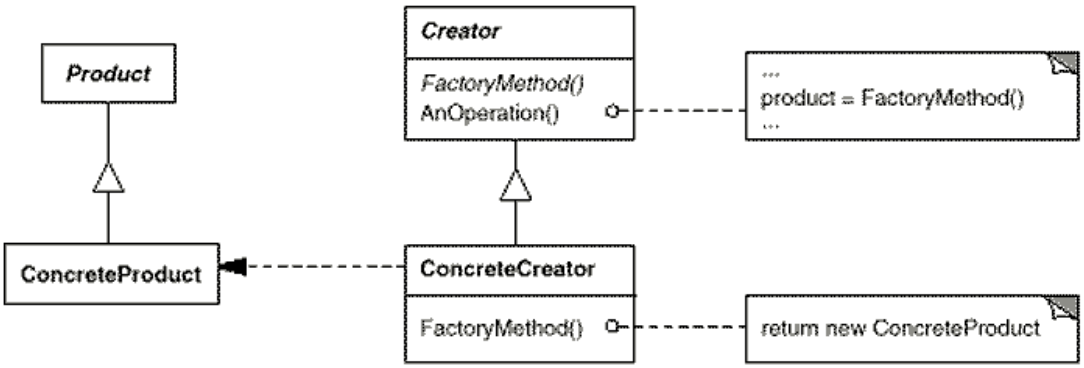
Creational patterns: Factory Method

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

Use the Factory Method pattern when

- A Class can't anticipate the class of objects it must create.
- A Class wants its subclasses to specify the objects it creates.
- Classes delegate responsibility to one of several helper subclasses, and you want to localize the knowledge of which helper subclass is the delegate.

Structure



Participants

Product (Shape)
defines the interface of objects the factory method creates.

ConcreteProduct (Square, Circle, Rectangle)
implements the Product interface.

Creator (Factory)
- declares the factory method (getShape), which returns an object of type Product (Shape). Creator may also define a default implementation of the factory method that returns a default ConcreteProduct object.
- may call the factory method to create a Product object.

ConcreteCreator (ShapeFactory)
overrides the factory method to return an instance of a ConcreteProduct.

Code listing 5 (FactoryMethod.cpp)

```
#include <iostream>
using namespace std;
#define interface struct

interface Shape
{
  virtual void draw() = 0;
};

class Square : public Shape
```

```
{
  public:
    void draw() {
      cout << "Inside Square::draw() method." << endl;
    }
};
class Circle : public Shape
{
  public:
    void draw() {
      cout << "Inside Circle::draw() method." << endl;
    }
};

class Rectangle : public Shape
{
  public:
    void draw() {
      cout << "Inside Rectangle::draw() method." << endl;
    }
};


interface Factory
{
  virtual Shape * getShape(char * shapeType) = 0;
};

class ShapeFactory : public Factory
{
  public:
    Shape * getShape(char * shapeType)
    {
      if(shapeType == NULL) {
          return NULL;
      }
      if(stricmp(shapeType, "CIRCLE") == 0)
      {
        return new Circle();
      }
      else if(stricmp(shapeType, "SQUARE") == 0)
      {
        return new Square();
      }
      else if(stricmp(shapeType, "RECTANGLE") == 0)
      {
        return new Rectangle();
      }
      return NULL;
    }
};

void main()
{
```

```
  //get shape factory
  Factory * shapeFactory = new ShapeFactory();

  //get an object of Shape Circle
  Shape * shape1 = shapeFactory->getShape("CIRCLE");
  if(shape1 != NULL)
    shape1->draw();

  //get an object of Shape Square
  Shape * shape2 = shapeFactory->getShape("SQUARE");
  if(shape2 != NULL)
    shape2->draw();

  //get an object of Shape Rectangle
  Shape * shape3 = shapeFactory->getShape("RECTANGLE");
  if(shape3 != NULL)
    shape3->draw();
}
```

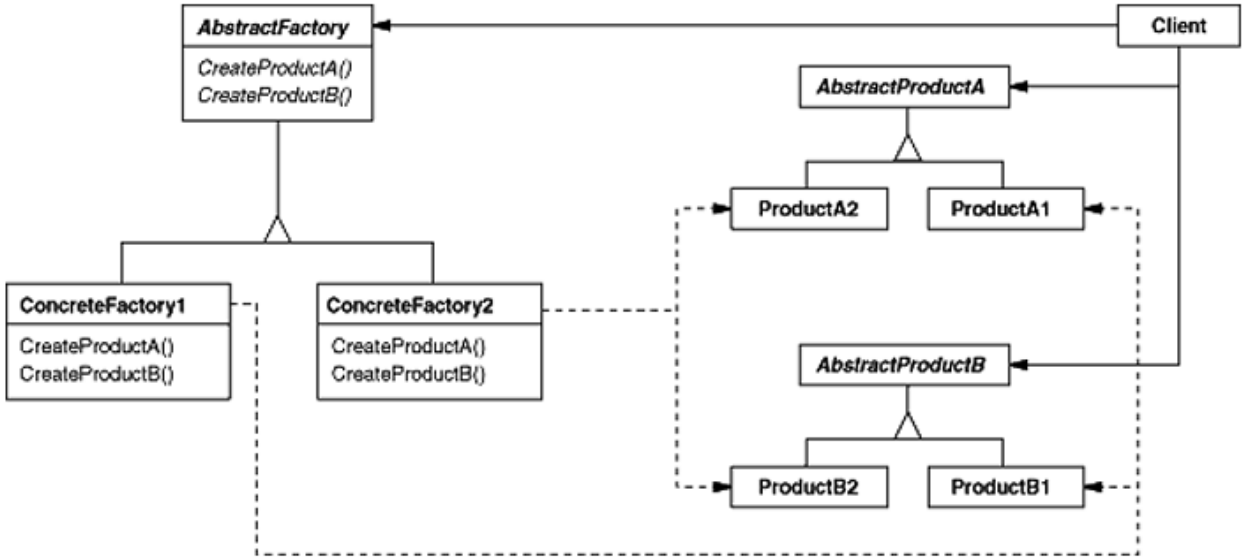
Creational patterns: Abstract factory

Provides an interface for creating families of related or dependent objects without specifying their concrete classes. Abstract Factory patterns work around a super-factory which creates other factories. This factory is also called as factory of factories. In Abstract Factory pattern an interface is responsible for creating a factory of related objects without explicitly specifying their classes. Abstract factory only declares the interface for creating products. It is up to the concrete subclasses to actually create them.

Use the AbstractFactory pattern when

- A system should be configured of how its products are created, composed and represented.
- A system should be configured with one of multiple families of products.
- A family of related product objects is designed to be used together, and you need to enforce this constraint.
- Provide a class library of products, and want to reveal just their interface, not their implementation.

Structure



Participants

AbstractFactory (AbstractFactory)
declares an interface for operations that create abstract product objects.

ConcreteFactory (ShapeFactory)
implements the operations to create concrete product objects.

AbstractProduct (Shape)
declares an interface for a type of product object.

ConcreteProduct (Square, Circle)
defines a product object to be created by the corresponding concrete factory.
implements the AbstractProduct interface.

Client
uses only interfaces declared by AbstractFactory and AbstractProduct classes.

Code listing 6 (AbstractFactory.cpp)

```
#include <iostream>
using namespace std;

#define interface struct

interface Shape
{
   virtual void draw() = 0;
};

class Square : public Shape
{
```

```
  public:
    void draw() {
      cout << "Inside Square::draw() method." << endl;
   }
};

class Circle : public Shape
{
  public:
    void draw() {
      cout << "Inside Circle::draw() method." << endl;
   }
};

interface AbstractFactory
{
   virtual Shape * getShape(char * shape) = 0;
};

class ShapeFactory : public AbstractFactory
{
  public:
    Shape * getShape(char * shapeType)
    {
      if(shapeType == NULL) {
        return NULL;
    }
    if(stricmp(shapeType, "CIRCLE") == 0)
    {
      return new Circle();
    }
    else if(stricmp(shapeType, "SQUARE") == 0)
    {
      return new Square();
    }
    return NULL;
   }
};

class FactoryProducer
{
  public:
    static AbstractFactory * getFactory(char * choice)
    {
      if(stricmp(choice, "SHAPE") == 0)
      {
         return new ShapeFactory();
      }
      return NULL;
    }
};

void main()
{
```

```
  //get shape factory
  AbstractFactory * shapeFactory =
      FactoryProducer::getFactory("SHAPE");

  //get an object of Shape Circle
  Shape * shape1 = shapeFactory->getShape("CIRCLE");

  //call draw method of Shape Circle
  shape1->draw();

  //get an object of Shape Square
  Shape * shape2 = shapeFactory->getShape("SQUARE");

  //call draw method of Shape Square
  shape2->draw();
}
```

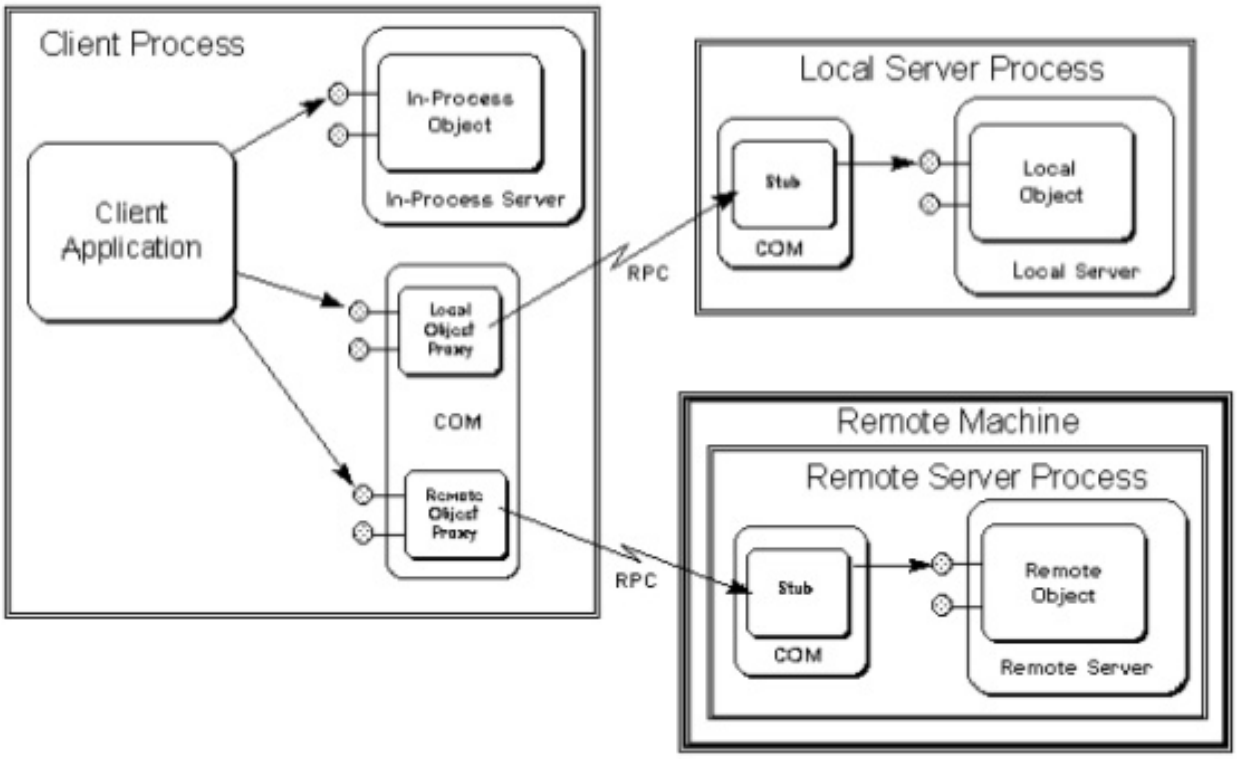
Structural patterns: Proxy

A proxy, in its most general form, is a class functioning as an interface to something else. The proxy could interface to anything: a network connection, a large object in memory, a file, or some other resource that is expensive or impossible to duplicate. In short, a proxy is a wrapper or agent object that is being called by the client to access the real serving object behind the scenes.

Let's look at an example. Say you have some objects running in a process on your desktop, and they need to communicate with other objects running in another process. Perhaps this process is also on your desktop; perhaps it resides elsewhere. You don't want the objects in your system to have to worry about findings other objects on the network or executing remote procedure calls.

What you can do is create a proxy object within your local process for remote object. The proxy has the same interface as the remote object. Your local objects talk to the proxy using the usual in-process message sends. The proxy then is responsible for passing any messages on to the real object, wherever it might reside.

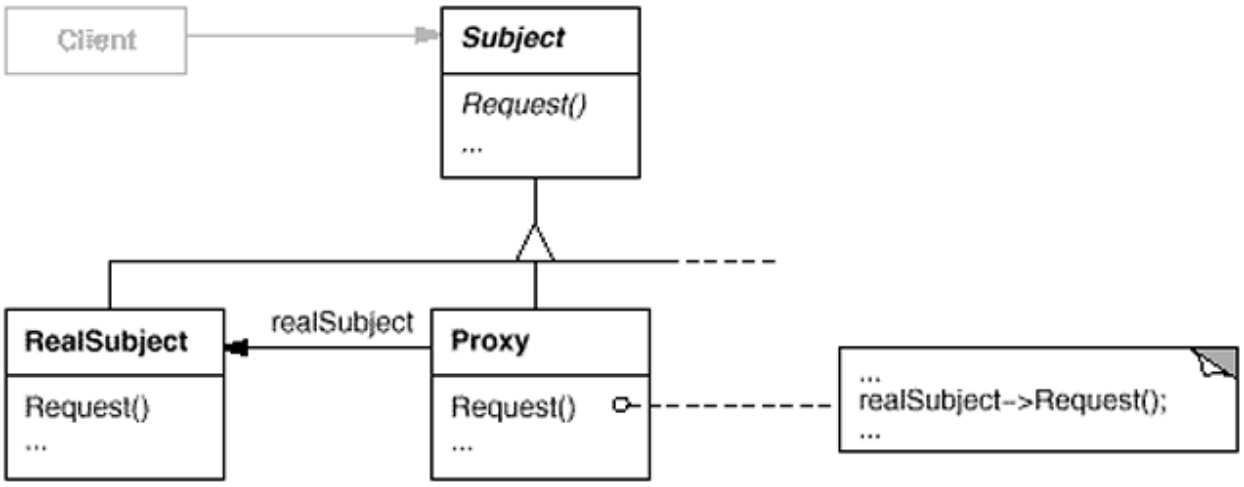
The proxy design pattern allows you to provide an interface to other objects by creating a wrapper class as the proxy. The wrapper class, which is the proxy, can add additional functionality to the object of interest without changing the object's code. A proxy provides a surrogate or placeholder for another object to control access to it.



Below are some of the common examples in which the proxy pattern is used:

- Adding security access to an existing object. The proxy will determine if the client can access the object of interest.
- Simplifying the API of complex objects. The proxy can provide a simple API so that the client code does not have to deal with the complexity of the object of interest.
- Providing interface for remote resources, such as web service or REST resources.
- Add a wrapper and delegation to protect the real component from undue complexity.
- Adding a thread-safe feature to an existing class without changing the existing class's code.

Structure



Participants

Proxy (ProxyCar)

- maintains a reference that lets the proxy access the real subject. Proxy may refer to a Subject if the RealSubject and Subject interfaces are the same.
- provides an interface identical to Subject's so that a proxy can be substituted for the real subject.
- controls access to the real subject and may be responsible for creating and deleting it.
- other responsibilities depend on the kind of proxy:
  - remote proxies are responsible for encoding a request and its arguments and for sending the encoded request to the real subject in a different address space.
  - virtual proxies may cache additional information about the real subject so that they can postpone accessing it.
  - protection proxies check that the caller has the access permissions required to perform a request.

Subject (ICar)
defines the common interface for RealSubject and Proxy so that a Proxy can be used anywhere a RealSubject is expected.

RealSubject (Car)
defines the real object that the proxy represents.

Code listing 7 (Proxy.cpp)

```
#include <iostream>
#include <string>
```

```
using namespace std;
#define interface struct

interface ICar {
  virtual void DriveCar() = 0;
};

// Real Subject
class Car : public ICar
{
  public:
    void DriveCar()
    {
      cout << "Car has been driven!";
    }
};

class Driver
{
  public:
    int Age;

    Driver(int age = 20)
    {
      this->Age = age;
    }
};

//Proxy Subject
class ProxyCar : public ICar
{
  Driver driver;
  ICar *realCar;

  public:
    ProxyCar(Driver driver)
    {
      this->driver = driver;
      realCar = new Car();
    }

    void DriveCar()
    {
      if (driver.Age <= 16)
        cout << "Sorry, the driver is too young to drive." << endl;
      else
        realCar->DriveCar();
    }
};

void main()
{
  ICar *car = new ProxyCar(Driver(16));
  car->DriveCar();
```

```
  car = new ProxyCar(Driver(25));
  car->DriveCar();
}
```

Code listing 8 (Proxy2.cpp)

```
#include <iostream>
#include <string>
using namespace std;

#define interface struct

// A protection proxy controls access to the original object

interface IAccess {
   virtual bool withdraw(int) = 0;
};

class Person
{
    string Role;
    static string roles[];
    static int next;
  public:
    Person()
    {
      if(next == 4)
        next = 0;
      Role = roles[next++];
    }
    string role()
    {
      return Role;
    }
};

string Person::roles[] =
{
  "Manager", "Finance", "Admin", "CFO"
};

int Person::next = 0;
class ActualAccess : public IAccess
{
    int balance;
  public:
    ActualAccess()
    {
      balance = 500;
    }
    bool withdraw(int amount)
    {
      if (amount > balance)
```

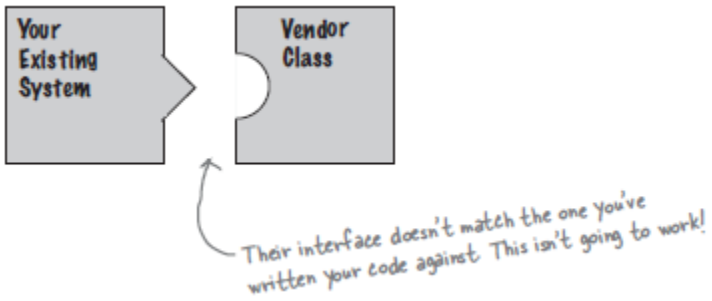
```
        return false;
      balance -= amount;
      return true;
    }
    int getBalance()
    {
      return balance;
    }
};

class ProxyAccess : public IAccess
{
    Person p;
    ActualAccess object;
  public:
    ProxyAccess(Person &obj)
    {
      p = obj;
    }
    bool withdraw(int amount)
    {
      if (p.role() == "Manager" || p.role() == "Director" ||
        p.role() == "CFO")
        return object.withdraw(amount);
      else
        return false;
    }
    int getBalance()
    {
      return object.getBalance();
    }
};

int main()
{
  Person staff[4];
  IAccess *pc;
  for (int i = 0, amount = 100; i < 4; i++, amount += 100)
  {
    pc = new ProxyAccess(staff[i]);
    if (!pc->withdraw(amount))
      cout << "No access for " << staff[i].role() << '\n';
    else
      cout << amount << " dollars for " << staff[i].role() << '\n';
  }
}
```

Structural patterns: Adapter

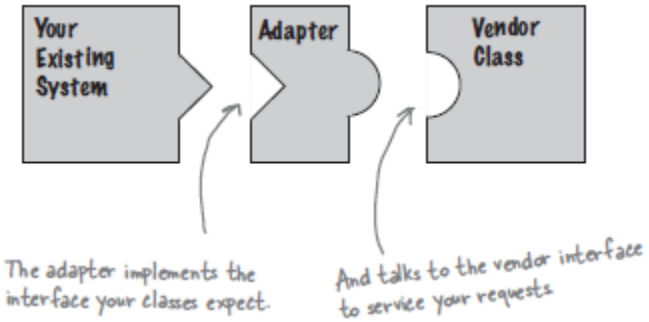
Adapter pattern works as a bridge between two incompatible interfaces. This pattern combines the capability of two independent interfaces.

Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.

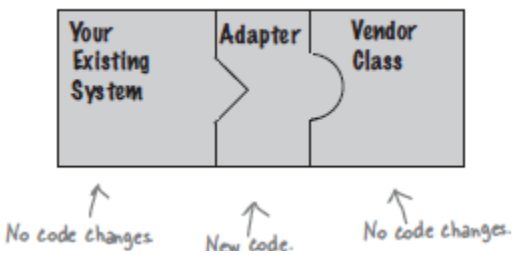
Suppose we have an existing system that you need to work with a new vendor class library, but the new vendor designed their interfaces differently than the last vendor.



We cannot solve the problem by changing the existing code (and we can't change the vendor's code). One solution is to write a class that adapts the new vendor interface into the one you are expecting.



The adapter acts as the middleman by receiving requests from the client and converting them into requests that make sense on the vendor classes.



Can you think of a solution that doesn't require YOU to write ANY additional code to integrate the new vendor classes? How about making the vendor supply the adapter class.

This pattern involves a single class which is responsible to join functionalities of independent or incompatible interfaces. A real life example could be a case of card reader which acts as an adapter

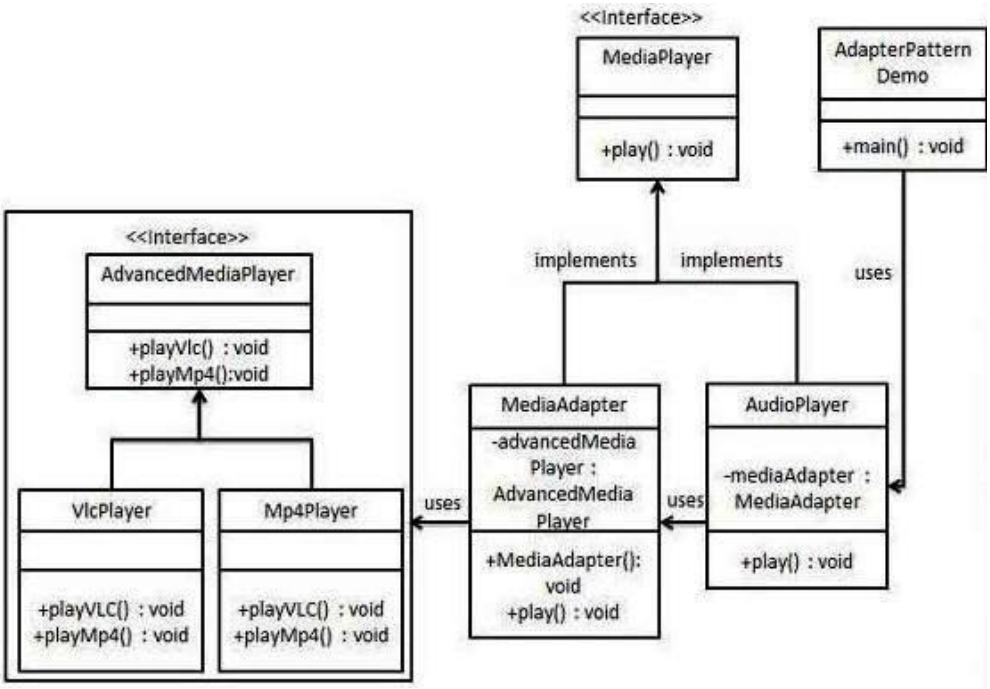
between memory card and a laptop. You plugin the memory card into card reader and card reader into the laptop so that memory card can be read via laptop.

We have a MediaPlayer interface and a concrete class AudioPlayer implementing the MediaPlayer interface. AudioPlayer can play mp3 format audio files by default.

We are having another interface AdvancedMediaPlayer and concrete classes implementing the AdvancedMediaPlayer interface. These classes can play vlc and mp4 format files.

We want to make AudioPlayer to play other formats as well. To attain this, we have created an adapter class MediaAdapter which implements the MediaPlayer interface and uses AdvancedMediaPlayer objects to play the required format.

AudioPlayer uses the adapter class MediaAdapter passing it the desired audio type without knowing the actual class which can play the desired various formats.



Code listing 9 (Adapter.cpp)

```
#include <iostream>
#include <string.h>
using namespace std;

#define interface struct

interface MediaPlayer
{
  virtual void play(char * audioType, char * fileName) = 0;
};
```

```
interface AdvancedMediaPlayer
{
  virtual void playVlc(char * fileName) = 0;
  virtual void playMp4(char * fileName) = 0;
};

class VlcPlayer : public AdvancedMediaPlayer
{
  public:
    void playVlc(char * fileName)
    {
      cout << "Playing vlc file. Name: " << fileName << endl;
    }
    void playMp4(char * fileName)
    {
      //do nothing
    }
};


class Mp4Player : public AdvancedMediaPlayer
{
   public:
    void playVlc(char * fileName)
    {
      //do nothing
    }
    void playMp4(char * fileName)
    {
      cout << "Playing mp4 file. Name: " << fileName << endl;
    }
};

class MediaAdapter : public MediaPlayer
{
  private:
    AdvancedMediaPlayer *advancedMusicPlayer;
  public:
    MediaAdapter(char * audioType)
    {
      if(stricmp(audioType, "vlc") == 0)
      {
        advancedMusicPlayer = new VlcPlayer();
      }
      else if(stricmp(audioType, "mp4") == 0)
      {
        advancedMusicPlayer = new Mp4Player();
      }
    }

    void play(char * audioType, char * fileName)
    {
      if(stricmp(audioType, "vlc") == 0)
      {
```

```
        advancedMusicPlayer->playVlc(fileName);
      }
      else if(stricmp(audioType, "mp4") == 0)
      {
         advancedMusicPlayer->playMp4(fileName);
      }
    }
};

class AudioPlayer : public MediaPlayer
{
   MediaAdapter *mediaAdapter;
   public:
    void play(char * audioType, char * fileName)
    {
      // inbuilt support to play mp3 music files
      if(stricmp(audioType, "mp3") == 0)
      {
         cout << "Playing mp3 file. Name: " << fileName << endl;
      }
      // mediaAdapter is providing support to play other file formats
      else if(stricmp(audioType, "vlc") == 0 ||
        stricmp(audioType, "mp4") == 0)
      {
         mediaAdapter = new MediaAdapter(audioType);
         mediaAdapter->play(audioType, fileName);
      }
      else{
         cout << "Invalid media. " << audioType <<
           "format not supported" << endl;
      }
   }
};



void main() {
  MediaPlayer *audioPlayer = new AudioPlayer();

  audioPlayer->play("mp3", "beyond the horizon.mp3");
  audioPlayer->play("mp4", "homealone.mp4");
  audioPlayer->play("vlc", "far far away.vlc");
  audioPlayer->play("avi", "test.avi");
}
```

Structural patterns: Facade

Provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.

Structuring a system into subsystems helps reduce complexity. A common design goal is to minimize the communication and dependencies between subsystems. One way to achieve this goal

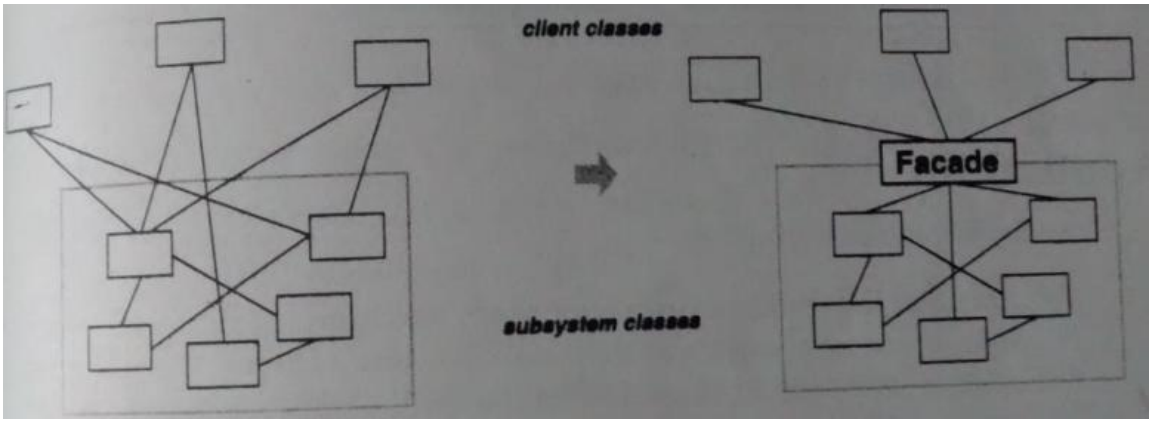
is to introduce a facade object that provides a single, simplified interface to the more general facilities of a subsystem.

Use the facade pattern when

- You want to provide a simple interface to a complex subsystem. Subsystems often get more complex as they evolve. Most patterns, when applied, result in more and smaller classes. This makes the subsystem more reusable and easier to customize, but it also becomes harder to use for clients that don't need to customize it. A facade provides a default view of the subsystem that is good enough for most clients. Only clients needing more customizability will need to look beyond the facade.

- There are many dependencies between clients and the implementation classes of an abstraction. Introduce a facade to decouple the subsystem from clients and other subsystems, thereby promoting subsystem independence and portability.

- You want to layer your subsystems. Use a façade to define an entry point to each subsystem level. If subsystems are dependent, then you can simplify the dependencies between them by making them communicate with each other solely through facades.



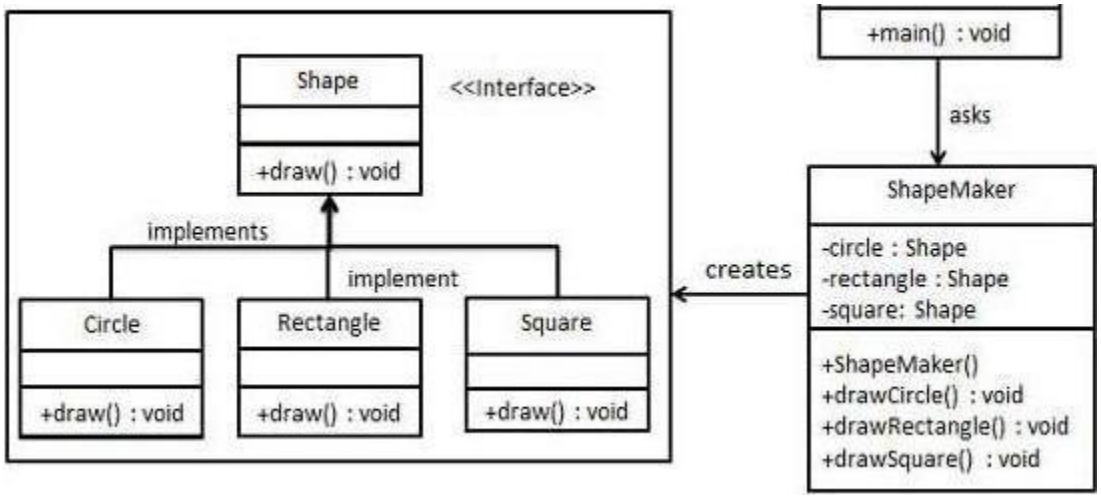
The facade pattern offers the following benefits:

1. It shields clients from subsystem components, thereby reducing the number of objects that clients deal with and making the subsystem easier to use.

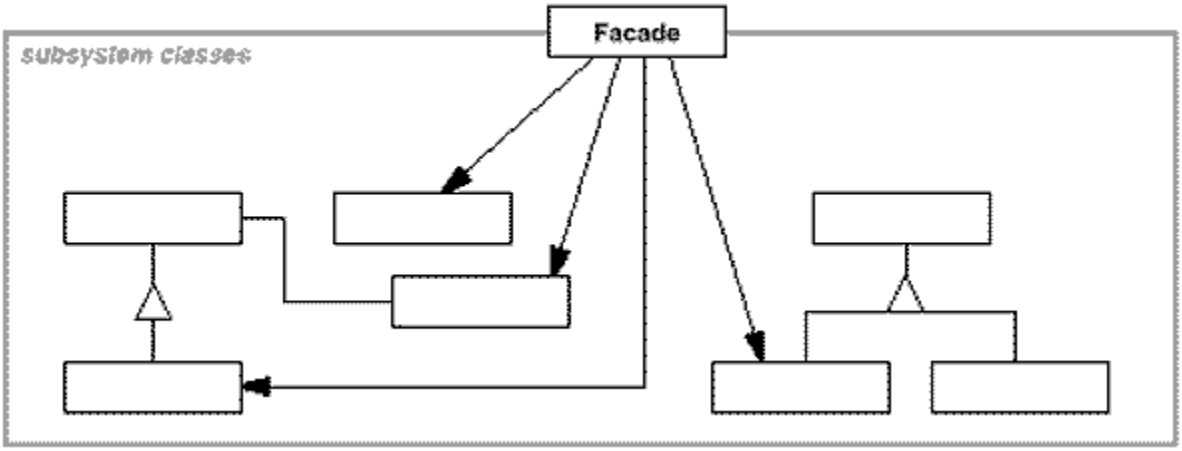
2. It promotes weak coupling between the subsystems and its clients. Often the components in a subsystem are strongly coupled. Weak coupling lets you vary the components of a subsystem without affecting its clients. Facades help layer a system and the dependencies between objects.

3. It doesn't prevent applications from using subsystem classes if they need to. Thus you can choose between ease of use and generality.

We create a Shape interface and concrete classes implementing the Shape interface. A facade class ShapeMaker uses the concrete classes to delegate user calls to these classes.



Structure



Participants

Facade (ShapeMaker)

- knows which subsystem classes are responsible for a request.
- delegates client requests to appropriate subsystem objects.

subsystem classes (Square, Circle, Rectangle)

- implement subsystem functionality.
- handle work assigned by the Facade object.
- have no knowledge of the facade; that is, they keep no references to it.

Code listing 10 (facade.cpp)

```
#include <iostream>
using namespace std;
#define interface struct

interface Shape
```

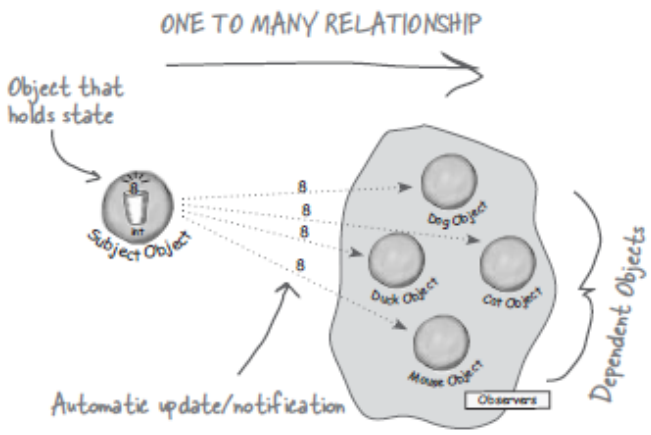
```
{
   virtual void draw() = 0;
};

class Square : public Shape
{
  public:
    void draw() {
      cout << "Inside Square::draw() method." << endl;
  }
};

class Circle : public Shape
{
  public:
    void draw() {
      cout << "Inside Circle::draw() method." << endl;
  }
};


class Rectangle : public Shape
{
  public:
    void draw() {
      cout << "Inside Rectangle::draw() method." << endl;
  }
};

class ShapeMaker {
  private:
    Shape *circle;
    Shape *rectangle;
    Shape *square;

  public:
    ShapeMaker() {
      circle = new Circle();
      rectangle = new Rectangle();
      square = new Square();
    }

    void drawCircle(){
      circle->draw();
    }

    void drawRectangle(){
      rectangle->draw();
    }

    void drawSquare(){
      square->draw();
    }
};
```

```
void main()
{
  ShapeMaker shapeMaker;

  shapeMaker.drawCircle();
  shapeMaker.drawRectangle();
  shapeMaker.drawSquare();
}
```

Behavioral patterns: Observer

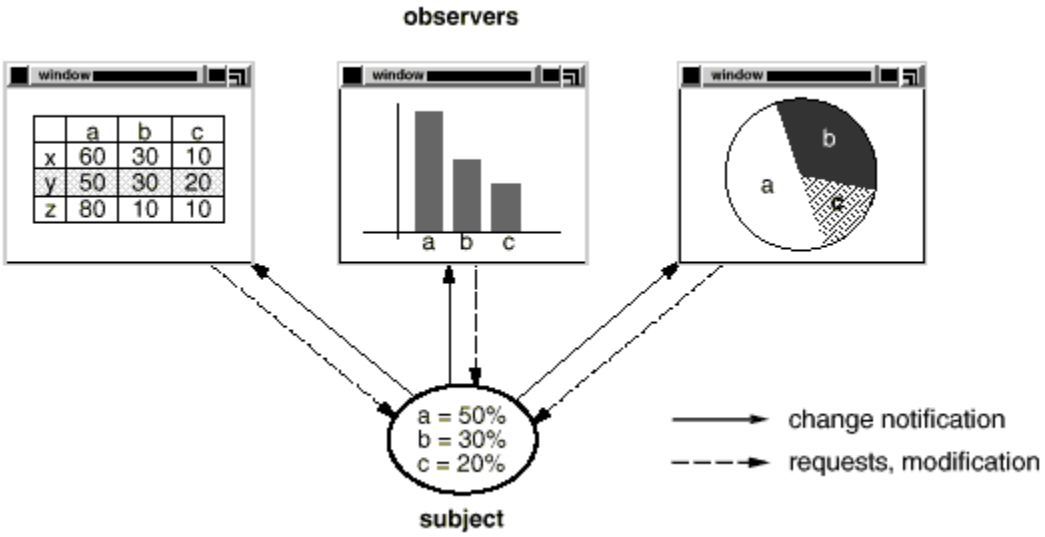
The Observer Pattern defines a one-to-many dependency between objects so that when one object changes state, all of its dependents are notified and updated automatically. When two objects are loosely coupled, they can interact, but have very little knowledge of each other.

The Observer Pattern provides an object design where subjects and observers are loosely coupled. The only thing the subject knows about an observer is that it implements a certain interface. Observer pattern uses three actor classes - Subject, Observer and Client. Subject is an object having methods to attach and detach observers to a client object.



When the state of one object changes, all of its dependents are notified.

Many GUIs toolkits separate the presentation aspects of the UI from the underlying application data. Classes defining application data and presentation can be reused independently. For example, a spreadsheet object and chart object can depict information in the same application data object using different presentations. The spreadsheet and chart object don't know about each other, but they behave as though they do. When the user changes the information in the spreadsheet, the chart object reflects the changes immediately, and vice versa.



This behavior implies that the spreadsheet and the charts objects are dependent on the data object and therefore should be notified of any changes in its state.

The observer pattern describes how to establish these relationships. The key objects in this pattern are subject and observer. A subject can have any number of dependent observers. All the observers are notified when the subject undergoes a change in state. In response, each observer will query the subject to synchronize its state with the subject's state.

Use the observer pattern in any of the following situations:

We can add a new observer at any time.

We never need to modify the subject to add a new type of observers.

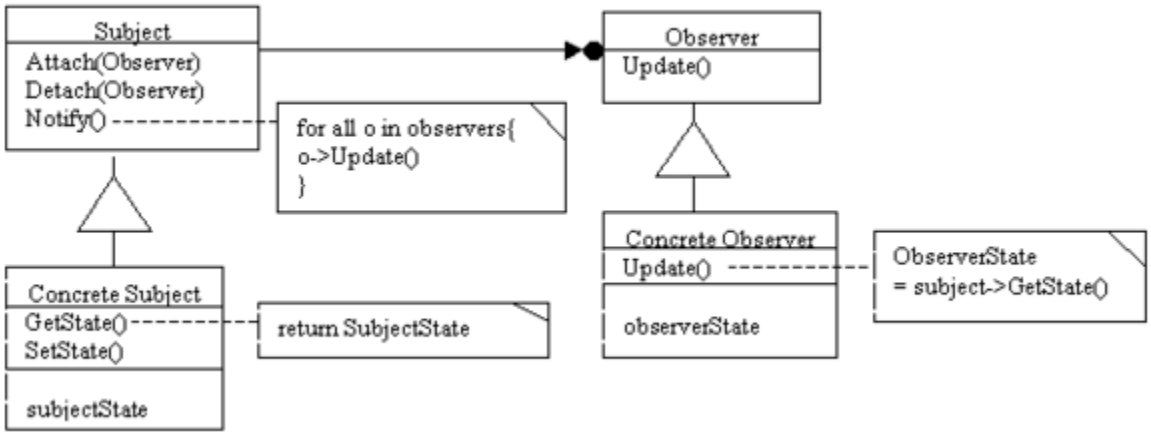
We can reuse subjects or observers independently of each other.

Changes to either the subject or an observer will not affect each other.

UML class diagrams and participants

Subject maintains a list of Observers and provides an interface for attaching and detaching them. Observer defines an abstract Update interface, which is called by the Subject when its state changes. Concrete Subject maintains state of interest for Concrete Observers and notifies them when the state changes. Concrete Observer implements the update interface provided by the Observer. It also maintains a reference to Concrete Subject and updates its state to be in sync with the Concrete Subject.

Structure



Participants

Subject (Subject)
- knows its observers. Any number of Observer objects may observe a subject.
- provides an interface for attaching and detaching Observer objects.

Observer (Observer)
defines an updating interface for objects that should be notified of changes in a subject.

ConcreteSubject (ConcreteSubject)
- stores state of interest to ConcreteObserver objects.
- sends a notification to its observers when its state changes.

ConcreteObserver (HexadecimalObserver, OctalObserver)
- maintains a reference to a ConcreteSubject object.
- stores state that should stay consistent with the subject's.
- implements the Observer updating interface to keep its state consistent with the subject's.

Code listing 12 (observer.cpp)

```
# include <iostream>
# include <list>

using namespace std;

class Subject;

class Observer {
   protected :
     Subject *subject;
   public:
     virtual void update() = 0;
};

class Subject {
  private:
    list <Observer *> observers;
```

```
  public:
    void attach(Observer *o){
      observers.push_back(o);
    }

    void detach (Observer* o) {
      observers.remove(o);
    }

    void notifyAllObservers()
    {
      list <Observer *>::iterator o_Iter;

      for (o_Iter = observers.begin(); o_Iter != observers.end();
        o_Iter++)
      {
        if(*o_Iter != 0)
        {
           (*o_Iter)->update();
        }
      }
    }
};

class ConcreteSubject: public Subject {
  private:
    int state;

  public:
    int getState() {
      return state;
    }

    void setState(int changedData) {
      state = changedData;
      notifyAllObservers();
    }
};

class HexadecimalObserver : public Observer
{
  private:
    ConcreteSubject* _subject;
  public:
    HexadecimalObserver(ConcreteSubject *sub)
    {
      _subject = sub;
      _subject->attach(this);
    }

    HexadecimalObserver()
    {
      this->_subject = NULL;
```

```
    }

    void update() {
      cout << "Hexadecimal String: " <<  hex <<  _subject->getState()
        << endl;
    }
};

class OctalObserver  : public Observer
{
  private:
    ConcreteSubject* _subject;

  public:
    OctalObserver (ConcreteSubject *sub){
      _subject = sub;
      _subject->attach(this);
    }

    OctalObserver()
    {
      this->_subject = NULL;
    }

    void update() {
      cout << "Octal String: " <<  oct <<  _subject->getState() <<
        endl;
    }
};

void main()
{
  ConcreteSubject *subject = new ConcreteSubject();
  OctalObserver *Octobj = new OctalObserver(subject);
  HexadecimalObserver *hexaObj = new HexadecimalObserver(subject);
  subject->setState(25);
  subject->setState(50);
  subject->detach(Octobj);
  subject->setState(100);
  subject->detach(hexaObj);
  subject->setState(200);
}
```

Behavioral patterns: Mediator

Usually a program is made up of a large number of classes. So the logic and computation is distributed among these classes. However, as more classes are developed in a program, especially during maintenance and/or refactoring, the problem of communication between these classes may become more complex. This makes the program harder to read and maintain. Furthermore, it can become difficult to change the program, since any change may affect code in several other classes.

With the mediator pattern, communication between objects is encapsulated with a mediator object. Objects no longer communicate directly with each other, but instead communicate through the

mediator. This reduces the dependencies between communicating objects, thereby lowering the coupling.

The problem with monolithic code is that the large number of cross connections between components makes it very hard to change or maintain. Using the mediator design pattern can help reduce class interdependencies, aid componentization, and ultimately help make classes service-oriented.

Define an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently.

Use the mediator pattern when

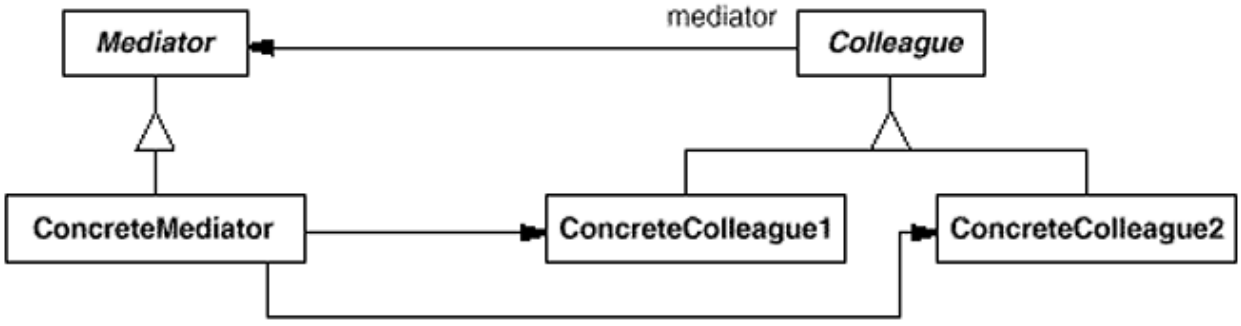
- A set of objects communicate in well defined but complex ways. The resulting interdependencies are unstructured and difficult to understand.
- Reusing an object is difficult because it refers to and communicates with many other objects.
- A behavior that's distributed between several classes should be customizable without a lot of subclassing.

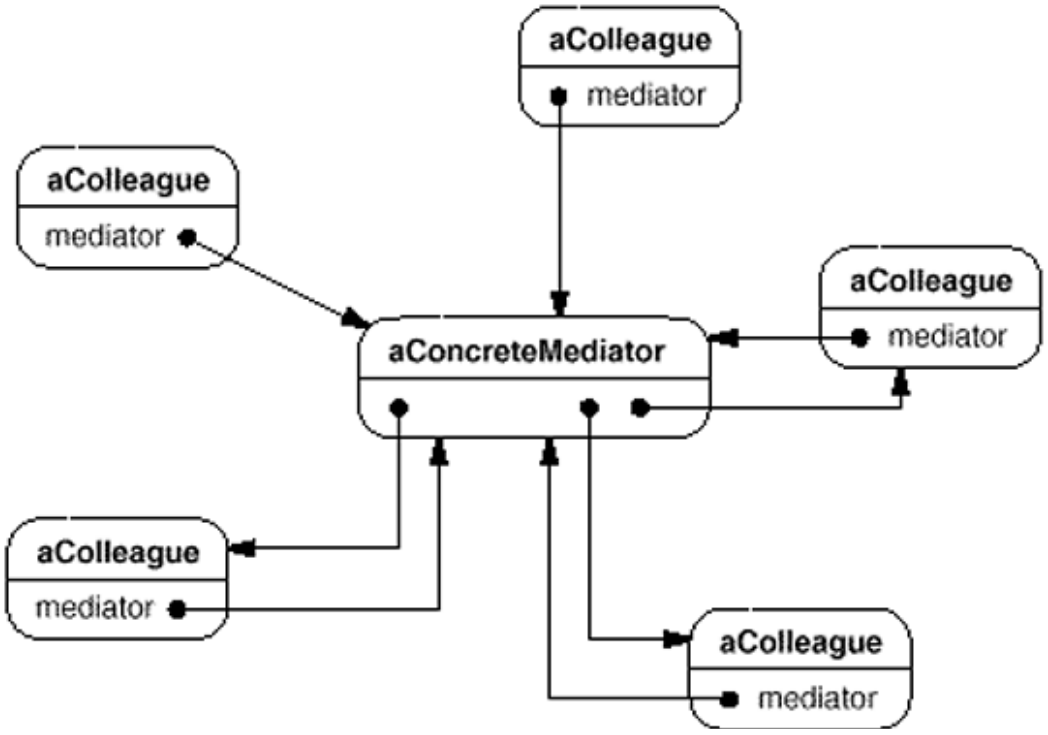
Participants

Mediator - defines an interface for communication with Colleague objects

ConcreteMediator - implements the Mediator interface and coordinates communication between Colleague objects. It is aware of all the Colleagues and their purpose with regards to inter communication.

ConcreteColleague classes - communicates with other Colleagues through its Mediator. Each colleague class knows its mediator object. Each Colleague communicates with its mediator whenever it would have otherwise communicates with another colleague.





Colleagues send and receive requests from a Mediator object. The mediator implements the cooperative behavior by routing requests between the appropriate colleague(s).

Code listing 13 (mediator.cpp)

```
#include <iostream>
#include <string>
# include <list>
using namespace std;

class Mediator;

class Colleague
{
  public:
    virtual ~Colleague() {};
    virtual void SendMsg(string) = 0;
    virtual void GetMsg(string) = 0;
  protected:
    Colleague(Mediator*);
    Mediator* _mediator;
};

class ConcreteColleague1 : public Colleague
{
  public:
    virtual ~ConcreteColleague1() {};
    ConcreteColleague1(Mediator*);
    virtual void SendMsg(string msg);
    virtual void GetMsg(string);
};
```

```
class ConcreteColleague2 : public Colleague
{
  public:
    virtual ~ConcreteColleague2() {};
    ConcreteColleague2(Mediator*);
    virtual void SendMsg(string msg);
    virtual void GetMsg(string);
};

class Mediator
{
  public:
    virtual ~Mediator() {};
    virtual void SendMsg(string, Colleague*) = 0;
  protected:
    Mediator() {};
};

class ConcreteMediator : public Mediator
{
  public:
    ConcreteMediator(Colleague* = NULL);
    virtual ~ConcreteMediator() {};
    void SetColleague(Colleague*);
    virtual void SendMsg(string msg, Colleague*);
  private:
    list <Colleague *> Colleagues;
};

ConcreteMediator::ConcreteMediator(Colleague* o) {
  Colleagues.push_back(o);
}

Colleague::Colleague(Mediator* pMediator)
{
  this->_mediator = pMediator;
}

ConcreteColleague1::ConcreteColleague1(Mediator* pMediator) :
  Colleague(pMediator) {}

void ConcreteColleague1::SendMsg(string msg)
{
  this->_mediator->SendMsg(msg, this);
}

void ConcreteColleague1::GetMsg(string msg)
{
   cout <<"ConcreteColleague1 Received:"<<msg << endl;
}

ConcreteColleague2::ConcreteColleague2(Mediator* pMediator) :
  Colleague(pMediator) {}
```

```
void ConcreteColleague2::SendMsg(string msg)
{
  this->_mediator->SendMsg(msg, this);
}
void ConcreteColleague2::GetMsg(string msg)
{
  cout <<"ConcreteColleague2 Received:" <<msg << endl;
}

ConcreteColleague3::ConcreteColleague3(Mediator* pMediator) :
  Colleague(pMediator) {}

void ConcreteColleague3::SendMsg(string msg)
{
  this->_mediator->SendMsg(msg, this);
}
void ConcreteColleague3::GetMsg(string msg)
{
  cout <<"ConcreteColleague3 Received:" <<msg << endl;
}

void ConcreteMediator::SetColleague(Colleague* p)
{
  Colleagues.push_back(p);
}

void ConcreteMediator::SendMsg(string msg, Colleague* p)
{
  list <Colleague *>::iterator o_Iter;

  for (o_Iter = Colleagues.begin(); o_Iter != Colleagues.end();
    o_Iter++)
  {
    if(*o_Iter != 0 && p != *o_Iter)
    {
      (*o_Iter)->GetMsg(msg);
    }
  }
}

int main()
{
  Mediator* pMediator = new ConcreteMediator();

  Colleague* c1 = new ConcreteColleague1(pMediator);
  Colleague* c2 = new ConcreteColleague2(pMediator);
  Colleague* c3 = new ConcreteColleague3(pMediator);

  ConcreteMediator *cm = NULL;
  if(cm = dynamic_cast<ConcreteMediator *>(pMediator)) {
    cm->SetColleague(c1);
    cm->SetColleague(c2);
    cm->SetColleague(c3);
  }
```

```
  c1->SendMsg("Colleague1 sends message");
  c2->SendMsg("Colleague2 sends message");
  c3->SendMsg("Colleague3 sends message");
  return 0;
}
```