Design Patterns

Essence of Object Oriented Software

By

Vivek Dutta Mishra

vivek@conceptarchitect.in

http://www.dev.vnc.in

Table of Content

Table of Contents

Design Patterns

In software engineering, a design pattern is a general reusable solution to a commonly occurring problem in software design. These are about design and interaction of objects, as well as providing a communication platform concerning elegant, reusable solutions to commonly encountered programming challenges. These patterns have been successfully implemented again and again in different problem domain and in each of these case they have met with huge success.

The credit of pioneering the ideas of patterns in programming goes to Christopher Alexander and his works on Architectural concepts in the year 1977-79. Further the work was carried forward Kent Beck and Ward Cunningham who began experimenting with the idea of applying patterns to programming . They presented their results at the OOPSLA conference in the year 1987.

However, the true credit of popularizing the idea of Design pattern goes to book Design Patterns: Elements of Reusable Object-Oriented Software was published in 1994. The book was co-authored by four authors - Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides. The team is popularly referred as Gang of Four. The book documented 23 different patterns categorized in three categories. Over a period of time the book gained as cult status and the patterns published in the book are popularly termed as GoF patterns.

During last fourteen years we have documented scores of other patterns and have come up with several more categories, however the patterns originally suggested by the GoF are unanimously considered as the foundation for all other patterns. The Original book is written with C++ as the base language (in fact C++ was the only main stream language worth considering at the time of writing the book). However the patterns can be successfully implemented in any Object Oriented language.

The Design patterns are defined as the proven successful approach to programming and in order to be termed as a pattern they need to have a proven track record. One shot successful code is not referred as pattern.

The book also suggests that patterns are not part of the language API or language construct. While it was true on the date book was published, newer languages such as Java and C# added quite a few pattern as a part of their foundation class and even provided language construct and support for the same. It must be realized that patterns were created as a work around to the limitations of the earlier Object Oriented Programming and the pains in the design limitation acted as motivation. The newer language had a precedence of these patterns which helped them implement cleaner Object Oriented constructs and specifications.

It must be understood that Patterns can’t be mastered by reading a book or with a short class room session. You need to feel the pain and limitation of your software solution to appreciate the solution and offered by the design patterns. Nevertheless, a study of design pattern familiarizes you with the terminology and their intents. And above all, a study of design pattern inculcates better and more effective Object Oriented Programming habit in the programmers.

Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, ithout ever doing it the same way twice

– Christopher Alexander

Rules of Pattern

Rule 1
Prefer Composition ("Has A") relationship over Inheritance ("Is A") relationship

Rules 2

Prefer Inheritance from Interface (Abstraction) over Inheritance from Class (Concrete)

Part 1
Creational Pattern

Motivations:

- Constructors have limitation
  - There is a difference between requesting and Object and Requesting a new Object
  - Constructors do not support polymorphism
  - Constructor creates strong coupling
- Aims at providing an alternativemechanism to create an Object
- Enables Controlled and delegated creation of Objects.

GOF Creational Patterns:

- Factory Patterns
  - Static Factory (Not a GOF Pattern)
  - Factory Method Pattern
  - Abstract Factory
- Prototype Pattern
- Builder Pattern*
- Singleton Pattern

* Not covered in current training program

CP-00: Static Factory Pattern (Not a GOF Pattern)

What?

One of the most popular mechanism (I resist using the word pattern) for abstracting the process of creation. It’s unfortunate; it doesn't appear in the list of Patterns documents by the GOF. Interestingly, quite often developers refer this pattern as Factory Method pattern. While this is not technically correct as per the definition of GOF, for practical purposes it appears to be the real candidate for the title of “Factory Method”. For the purpose of differentiating the two, here we will continue to refer this design as Static Factory Pattern.

A ***Static Factory Pattern*** essentially is:

- A Static Method (generally) of a class that returns the Object of the class (or its sub classes)
- The returned object may be of the same class or its subclass; thus brining polymorphism in the process of creation.
- The returned object may be a newly created object or may be an existing object returned from a pool of Objects.

Design Benefits

A ***Static Factory Pattern*** essentially is a replacement of direct constructor and provides several benefits over a constructor:

- Method can have any name (unlike constructor)
- Method can return object of the sub-classes (unlike constructor)
- It can decouple client from the subclass implementations.
- There is a difference between requesting an Object and requesting a ***new*** object. A constructor almost always fails to differentiate the two.
- Client may not be aware as to which class is being instantiated.

Sample Code

```
class Color
{
        public:
                public static Color * CreateRGB(int r, int g, int b){...}
                public static Color * CreateHSB(int h, int s, int b){...}
}
```

Relation with other Patterns

The ***Static Method Pattern*** is extensively used as one of the essential element in many of the GOF design patterns:

- It generally provides an access to the Singleton Object
- It is used with Facade
- It is generally used for creating Objects of Iterator type.
- It is used with Prototype Pattern.

CP-01: Singleton Pattern

What?

Singleton pattern ensures that only one instance of a given class will ever exist. It also provides a mechanism to obtain this instance that any entity can access within the system. Generally a *Singleton* class is required to satisfy two requirements:

- Single Instance: Only one instance must exist
- Global Access: The only instance must be accessible across the system.

Why?

Singleton Play important role in several different situations including but not limited to:

- When exactly one object is needed to coordinate actions across the system.
- A Preferred replacement of Global resources – methods and constants.
  - Less polluted namespace
  - Lazy initialization
- Allow different parts of the system to share information and also to interact with each other.

Real life Analogy

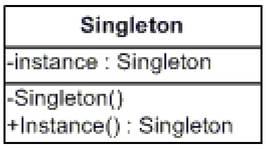
- A central EPBX System which receives all external call and routes it within the Organization.
- A Radio channel, which broadcasts on a particular frequency. In the given context (city or catchment area) no other radio can broadcast at same frequency. Any number of clients can access the same channel at same frequency.

Implementation:

A Singleton class essentially requires two components to implement it:

- A static reference to the only instance that is created or will be created.
- A *Static Factory* to make the object available and optionally create it.

UML Diagram



Sample Code

```
class Singleton
{
      static Singleton * _instance;

      //private constructor prevents object creation from outside
      Singleton()       {     }

public:
      static Singleton * GetInstance()
      {
            if(_instance==NULL)
                  _instance=new Singleton();

            return _instance;
      }

      //Buisness Logic follows here
      void MyBusinessLogic(){...}
};
```

Implementation Concerns

A Singleton Implementation may require addressing one or more optional concerns -

1. Thread Safety

In a multi-threaded environment, Thread safety becomes one of the important concerns. In the code above, if multiple thread attempts to call Singleton.GetInstance() at the same time, we may end of creating several instances of the Singleton object which will ultimately break the motivation of the pattern. Thread safety implementation is generally language and API dependent. Often Thread safety can be achieved by trading off with Lazy Initialization. Preferred way to Thread Safety and Lazy Initialization is to use double checked locks.

2. Lazy Initialization

Often we commit large amount of resource to a Singleton Object. Often being heavy weight is one of the motivations for creating Singleton Class. As such it makes sense to Instantiate Singleton object as late as possible, preferably just before its first requirement. The late Initialization often forces Thread safety concerns.

3. Object Persistence, Deletion and Timeout

In real life scenario, often we need to consider the lifespan of a Singleton object and define optional mechanism to remove the object from the memory if it is no more being referred. Also we can persist the object so that it can once again be instantiated when needed.

Implementation Alternatives

In many a cases, A Singleton Design may simply be replaced by class containing a *Static Class* – that is a class containing all static fields and methods thus eliminating the need to create even a single object. It is left to the readers to find out the limitation of this alternative design.

Known Popular Implementation

CWinApp is a Singleton class in MFC Framework

Derivation

A common derivative of Singleton pattern is ***Multiton Pattern***. This pattern extends the idea of “only one object” to “only n objects”. Thus it allows creation of a limited number of objects in a controlled way. Multiton pattern can be defined as smart Enumeration.

Relation with other Patterns

- Abstract Factory, Builder often use Singleton in their implementation
- Mediator, Facade and State are often defined Singleton or Multiton.

Concerns

Due to overuse, Singleton is often considered as an anti-pattern.

CP02: Factory Method Pattern

Also known as :

Virtual Constructor

What?

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

Factory Method pattern, like other creational patterns, aims at creating an Object without actually specifying the actual class for which Object will be created. This essentially introduces the idea of polymorphism in the process of creation.

The modern adaption and application of Factory Method pattern blurs the gap between Factory Method Pattern and Static Factory Pattern discussed earlier. This chapter concentrates on the classical design as per GOF.

Design Benefits

- Polymorphism in the process of creation – The client is now totally decoupled from the implementation details of the sub-classes.
- Can often return object from a pool rather than creating a new object.

Where?

Factory Methods are often used in framework and toolkit designs. The Framework developer's often define the abstract method for creating the object and Framework implementers are required to define the creation process in the derived classes.

Implementation

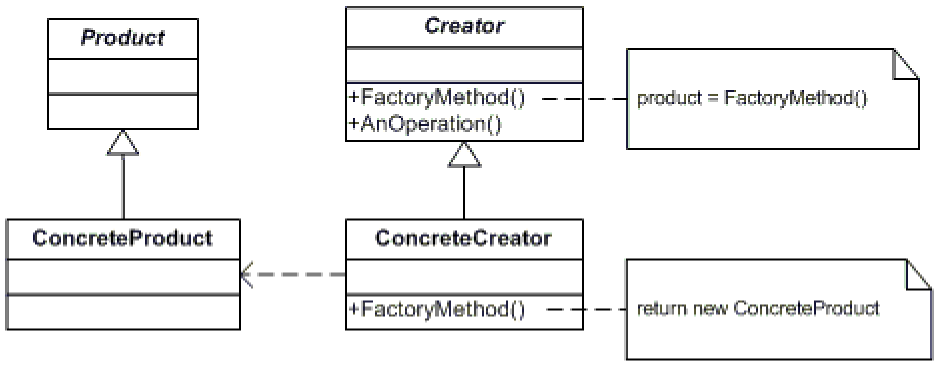
1. Participants:

- Abstract Product
- Concrete Product
- Abstract Creator
- Concrete Creator

Here Creators are the business Objects containing the factory methods which in turn create a Product or its derived product. A concrete creator gets to choose which sub-product they will be creating. In a typical scenario, the Abstract Creator will define the necessary operations and business logic to be used on the line of Products. The sub-classing will typically specify the exact

Products to be used.

2. The UML Diagram



Sample Code

```
class Product
{
            public:
                        virtual double GetPrice()=0;
};

class ProductA  : public Product { };

class ProductB : public Product { };


class Consumer
{

public:
            virtual Product * GetProduct();

            public void PrintProductDetails()
            {
                        Product * p=GetProduct();
                        cout<<endl<<p->getPrice();
                        cout<<endl<<typeinfo(*p).name();
            }
};

class ProductA_Consumer: Consumer
{
public:
            Product * GetProduct() {return new ProductA;}

};

class CreatorB_Consumer: Consumer
```

```
{
public:
        Product * GetProduct() {return new ProductB;}

};

template<class T>
class StandardConsumer:public Consumer
{
public:
        Product * GetProduct(){ return new T;}
};

void main()
{
        Consumer *c=NULL;

        if( some_condition)
                c=new ProductA_Consumer;
        else
                c=new ProductB_Consumer;


        c->PrintProductInfo();
}
```

Sample code provides and structural over-view of how Factory Method works. Here the Consumer class is the Creator for Products. The base class defines the method to print product details, however, it is blissfully unaware of the actual product to be used. The actual product will be decided by sub-classing Consumer who is specifying the actual Product.

Another popular example of Factory Method Pattern is a Maze game in which the base implementation defines Products such as Room, Wall and Doors to create the Maze. Derived Maze creators may use Products such as EnchantedDoors or WallsWithBombs or RotatingRooms. Thus we will have different Maze Creator which will be using Different types of Products to Build it.

Finally need of creating a new Creator for every new Product can be eliminated by creating a template based Creator

Derivation and Extensions

The classical document of this pattern (as per GOF) overlaps in its structure with Abstract Factory and to an extent with the Builder pattern. In an increasingly popular notion, we apply the term Abstract Factory to a more generic scenario which includes non-polymorphic cases. In this more popular notion, the Factory Method essentially takes the form which we earlier discussed as Static Factory.

Relation with other Patterns

- Abstract Factory is often implemented using Factory Method Pattern; however they may also be implemented using Prototype.

- Often confused with Abstract Factory. Important distinction, apart from intent, is
    - Factory Method achieves its goal through inheritance whereas Abstract Factory through delegation.
    - Abstract Factory is defined as Object Based patters; Factory Method as class based.
    - Factory Method emphasizes on creation of one Object; Abstract Factory a family of Objects.
- Often confused with Builder. The two differ in intent.
    - Builder's builds a complicated object in steps; Factory Method subclasses in order to defer decision of creation.
- Often design evolves as Factory Method and then takes the form of Abstract Factory or Builder.

CP03: Abstract Factory Pattern

What?

Provides and interface for creating a family or related or dependent products (objects) without client requiring to the exact concrete class to be Instantiated. It can be achieved by encapsulating a group of factories sharing a common theme.

Why?

If a system need to be portable it needs to:

- Encapsulate Platform dependencies. Platform may include:
    - OS API based GUI or other systems
    - Database API for different databases
    - Themes or Schema
    - Business Logic Specific to a Product or Geographic Domain
- The switching between available options should configurable.
- Client need not be aware or bothered about the exact Set currently in use.

When?

Abstract factory will be the preferred pattern when:

- A System needs to work with a family (group) of related and/or dependent entity (object)
- System should be independent of how the products are created or composed.
- When we want to enforce "*The group of product should be used together in combination"*
- The concrete classes are exposed via an interface alone and implementations are not revealed.

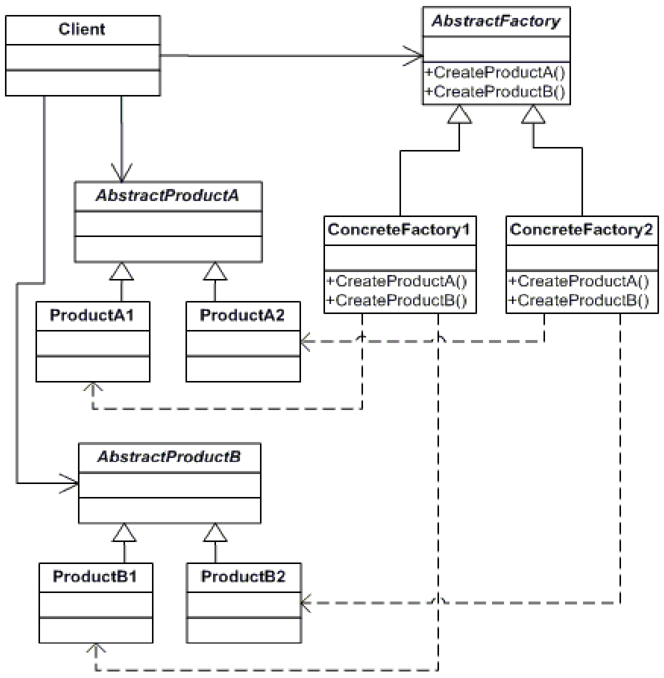
Implementation

1. Participants

- Abstract Products
    - Example *AbstractButton, AbstractForm, AbstractTextBox* etc
- AbstractProductFactory - *AbstractUIFactory*
    - Contains FactoryMethod or Interface to Create the entire range of Abstract Product
- Product Suite One
    - Example: *MetallicUISet*
    - Contains
        - Concrete Products like *MetallicButton, MetallicTextBox* etc

        - Concrete Product Factory - *MetallicUIFactory*
            - returning MetalicButton as Abstract Button, MetallicTextBox as AbstractTextBox
- Product Suite Two
    - Example: *GlassUISet*
    - Contains
        - Concrete Products like *GlassButton, GlassTextBox* etc
        - Concrete Product Factory - *GlassUIFactory*
            - returns GlassButton for AbstractButton, GlassTextBox for AbstractTextBox
- Client
    - Client obtains a reference to one of the ConcreteProductFactories (say *MetallicUIFactory*)
    - Uses interface methods from Abstract factory (*AbstractUIFactory* here*)* to create set of Products.
    - Will not know what actual classes are instantiated.
    - The only concrete class client need to know (in worst case) is the concrete factory class.

2. UML Diagram



Sample Code

```
//abstract. cpp -- abstractui.dll
```

```
class AbstractButton{};

class AbstractForm{ };

class AbstractTextBox{ };

class AbstractUIFactory
{
        public:
                virtual AbstractButton * CreateButton()=0;
                virtual AbstractTextBox * CreateTextBox()=0;
                virtual AbstractForm * CreateForm()=0;

};

//Metallic.cpp -- metallicui.dll

class MetallicButton: public AbstractButton {};

class MetallicForm: public AbstractForm {};

class MetallicTextBox: public AbstractTextBox {};

class MetallicFactory: public AbstractUIFactory
{
        public:
                AbstractButton * CreateButton() {return new MetallicButton;}
                AbstractTextBox * CreateButton() {return new MetallicTextBox;}
                AbstractForm * CreateButton() {return new MetallicForm;}
};

//Glass.cpp -- Glassui.dll

class GlassButton: public AbstractButton {};

class GlassForm: public AbstractForm {};

class GlassTextBox: public AbstractTextBox {};

class GlassFactory: public AbstractUIFactory
{
        public:
                AbstractButton * CreateButton() {return new GlassButton;}
                AbstractTextBox * CreateButton() {return new GlassTextBox;}
                AbstractForm * CreateButton() {return new GlassForm;}
};



//client code
```

```
void main()
{
	AbstractUIFactory * pFactory=new MetallicUIFactory;

	AbstractButton * button=pFactory->CreateButton();
	AbstractTextBox * textBox=pFactory->CreateTextBox();
	AbstractForm * form=pFactory->CreateForm();



}
```

Relation with Other Patterns

- Often confused with Factory Method Pattern.
    - Highlight is creating a family of product as against a single product in case of Factory Method
- Uses a group of Factory Method to create individual product of the family
- Often uses prototype to create individual products.

CP04: Prototype Pattern

What?

The Pattern creates a new object by cloning an existing object rather than by using its class definition.

Why?

Prototype provides several benefits over other creational patterns and the obvious new operator.

- Clients need not the class to be instantiated.
- No expensive inheritance hierarchy for creators (as in case of Factory Method or Abstract Factory).
- Can reduce class hierarchy in case of Abstract Factory pattern if prototyping is used.

When?

- You need to create a number of similar objects.
- When you know what object you need to clone; but not to which class this object belongs
- When you are working with a large number of objects with common abstraction and you intend to refer the object via abstraction and not directly.

Implementation

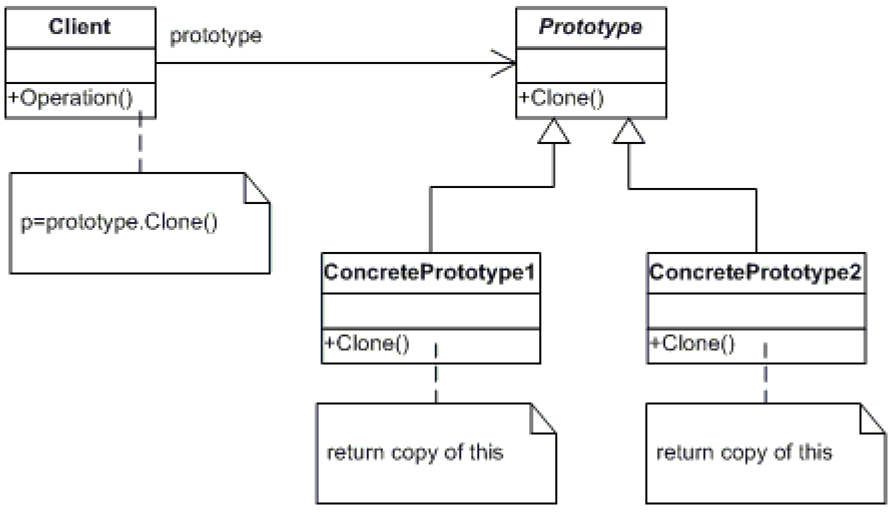
1. Participants

- Abstract Base (base of Hierarchy)
  - defines abstract clone method.
- Concrete Classes
  - each of them defines their own cloning mechanism.
- Client
  - Refers to objects using Abstract Base.
  - Object may be instantiated at different parts of the system using other mechanism.
  - Object may be instantiated using cloning at some part of the system.

2. Structure

- The Base class defines method to clone
- Each Subclass defines its own way to create its duplicate
- Client need to call clone method to create a new instance of the object.

UML Diagram



Sample code

```
class Length
{
public:
        virtual Length * Clone()=0;
        virtual Length * SchemaClone()=0;

        virtual double ToDouble()=0;
        virtusl LPCTSTR ToString()=0;
}

class Meter: Length
{
        int m;
        double cm;
public:
        Meter(int m, double cm)
        {
                this.m=m;
                this.cm=cm;
        }

        double ToDouble(){return m+cm/100;}
        LPCTSTR ToString(){...}
        int getMeter(){ return m;}
        int getCM{return cm;}

        Length * Clone(){ return new Meter(m,cm);}
        Length * SchemaClone(){reutrn new Meter(0,0);}

};

class Feet: Length
{
        int feet;
        double inch;
public:
        Meter(int feet, double inch)
        {
                this.feet=feet;
                this.inch=inch;
        }

        double ToDouble(){return feet+inch/12;}
        LPCTSTR ToString(){...}
        int getMeter(){ return feet;}
        int getCM{return inch;}

        Length * Clone(){ return new Feet(feet,inch);}
        Length * SchemaClone(){reutrn new Feet(,0);}

};


void main()
{
        Length * l=new Meter(10,10);
        Length * l2=new Feet(1,3);

        Length * l3= l->Clone();

        Length * l4= l->SchemaClone();

        cout<<endl<<l3->ToString();
        cout<<endl<<l4->ToString();
}
```

Common Implementations

- Cloning mechanism is available in C++ using
  - copy constructor
  - operator =
- Java
  - Object class defines clone method
- C#.Net
  - Object class defines MemeberwiseClone method.

Implementation Concerns

- Cloning, if not properly executed leads to undesirable consequences including
    - Identity duplication where you fail to distinguish between the original and duplicate objects.
    - Sallow copy where you end up sharing some information between the original and duplicate object.
- Cloning in Java and C# is restricted and we need to take certain steps to activate it.
- In C++ we are often required to override default copy constructor and operator = to ensure deep copy.
- It is recommended to define own language independent cloning mechanism.
- Many a times we just need to copy schema (object type) but not the actual data associated with the protypical instance.

Relation with other patterns

- Prototyping is often used in place of Factory Method in Abstract Factory Pattern
- Cloning can be used to clone an Iterator, Memento or a Composite Structure.

Part 2
Structural Pattern

Motivations

Structural patterns are concerned with how classes and objects are composed to form larger structures.

- Structural *class* patterns use inheritance to compose interfaces or implementations. These patterns are particularly useful for making independently developed class libraries work together.
- structural *object* patterns describe ways to compose objects to realize new functionality rather than composing interfaces or implementations.

GOF Structural Pattern

- Adapter Pattern
- Facade Pattern
- Proxy Pattern
- Decorator Pattern
- Composite Pattern
- Bridge*
- Flyweight*

* Not covered

S01: Adapter Pattern

Also known as:

Wrapper

What?

Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces. An adapter typically wraps an existing class code and does little other than changing the interface and making incompatible pieces work together.

How?

- Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.
- Wrap an existing class with a new interface.
- Impedance match an old component to a new system

When?

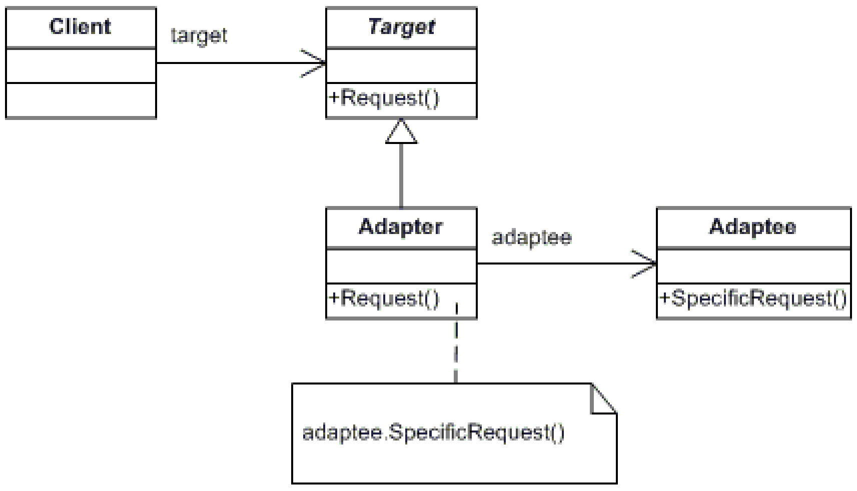
- An "off the shelf" component offers compelling functionality that you would like to reuse
- Its interface is not compatible with the philosophy and architecture of the system currently being developed.

Implementation

1. Participants

- Legacy class (Adaptee)
- Adapter class
- Client

2. UML Diagrams



Sample Code

```
//Legacy Code
class Box
{
public:
        void SetStart(Point pt){...};
        void SetEnd(Point pt){...};
        Point GetStart(){...}
        Point GetEnd(){...}
        void Draw(){...};
}

//New System Interface
class IRectangle
{
        public:
                virtual void SetStart(Point pt)=0;
                virtual void SetSize(Size sz)=0;
                void Draw()=0;
}

//Adapter
class Rectangle
{
        Box b;
        public:
                void SetStart(Point pt)
                {
                        b.SetStart(pt);
                }
```

```
                    void SetSize(Size sz)
                    {
                              b.SetEnd( b.GetStart().GetX()+sz.GetWidth(),
                                                    b.GetStart().GetY()+sz.GetHeight());
                    }

                    void Draw()
                    {
                              b.Draw();
                    }
}


//client
void main()
{
          IRectangle *r=new Rectangle( Point(4,5), Size(10,20));
          r->Draw(); //Draws Box (4,5) -  (14, 25)
}
```

Relation with Other patterns

- Often confused with other structural patterns like
    - Facade
        - Facade simplifies a complex system; Adapter makes system work
    - Bridge
        - Bridge is an early design consideration to ensure interface and implementation can change independent of each other
    - Proxy
        - Aims at providing the same interface; Adapter changes the interface
    - Decorator
        - Adds and configures new set of functionality; Adapter doesn't add a new functionality.

S02: Facade Pattern

What?

Provide a simplified and uniform interface to a set of interfaces in a subsystem. It defines higher level interface that makes the subsystem that makes the subsystem easier to use.

It wraps a complicated subsystem within a simpler interface.

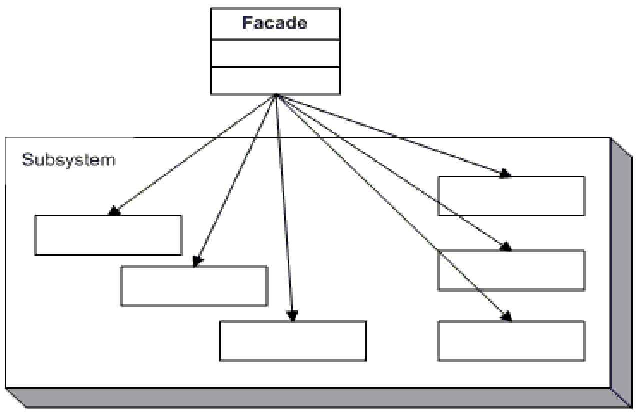
When?

- Provide a simplified, improved, or more object-oriented interface to a sub-system that is
  - overly complex (or may simply be more complex than is needed for the current use)
  - poorly designed
  - a decayed legacy system
  - or just inappropriate for the system consuming it.
- Multiple Different clients require different interface to access the same subsystem
  - Power User Interface
  - Wizard Interface
- Need for the re-use of a valuable sub-system without coupling to the specifics of its nature.

Real life Analogy

- A Customer care center or Reception acts as a Facade to the whole organization from a clients Perspective.

UML Diagram



Implementation

- Identify a simpler, unified interface for the subsystem or component.
- Design a “wrapper” class that encapsulates the subsystem.
- The facade/wrapper captures the complexity and collaborations of the component, and delegates to the appropriate methods.
- The client uses (is coupled to) the Façade only.
- Consider whether additional Facades would add value.

Sample Code

```
class SubSystemOne
{
  public:
  bool MethodOne()
  {
    printf("SubSystemOne Method");
            return rnd()%100>20;
  }
}

class SubSystemTwo
{
  public:
  void MethodTwo()
  {
    printf("SubSystemTwo Method");
  }
}

class SubSystemThree
{
  public:
  bool MethodThree()
  {
    printf("SubSystemThree Method");
            return rnd()%100>40;
  }
}

class SubSystemFour
{
  public:
  void MethodFourIf()
  {
    printf("SubSystemFour Method");

  }

  void MethodFourElse()
  {
    printf("SubSystemFour Method");
  }
}

// "Facade"
class Facade
{
  // Fields
  SubSystemOne* one;
  SubSystemTwo* two;
  SubSystemThree* three;
  SubSystemFour* four;

  // Constructors
  public: Facade()
  {
    one = new SubSystemOne();
    two = new SubSystemTwo();
    three = new SubSystemThree();
    four = new SubSystemFour();
  }

  // Methods
  public: void MethodA()
  {
    printf( "\nMethodA() ---- " );
    if(one->MethodOne())
                    two->MethodTwo();
            else
                    four->MethodFourIf();
  }

  public: void MethodB()
  {
    printf( "\nMethodB() ---- " );
    two->MethodTwo();
            if(three->MethodThree())
                    four->MethodFourIf();
            else
                    four->MethodFourElse();
  }
};

  void main()
  {

    Facade f = new Facade();
    f->MethodA();
    f->MethodB();

  }
```

Relationship with Other Patterns

- Not to be confused with
  - Adapter
    - Adapter is a patch solution to make incompatible element work; facade makes a working system work in a simpler manner.
  - Decorator
    - Adds or configures new set of functionality; Facade simply reorganizes current set of functionality.

S03: Proxy Pattern

Also Known as

Surrogate

What?

- Provide a surrogate or placeholder for another object to control access to it.
- A class functioning as an interface to:
    - Other class
    - Network
    - Expensive resource
- A Proxy is preferred to be transparent; sharing the same interface as the object it is impersonating.

When?

- You want to avoid undue and extra complexity added to original object
- Use extra level of indirection to support features such as:
    - Distributed access
    - Controlled access
    - Intelligent access
- Provide access to expensive resource or resources that can’t be duplicated.

Implementation

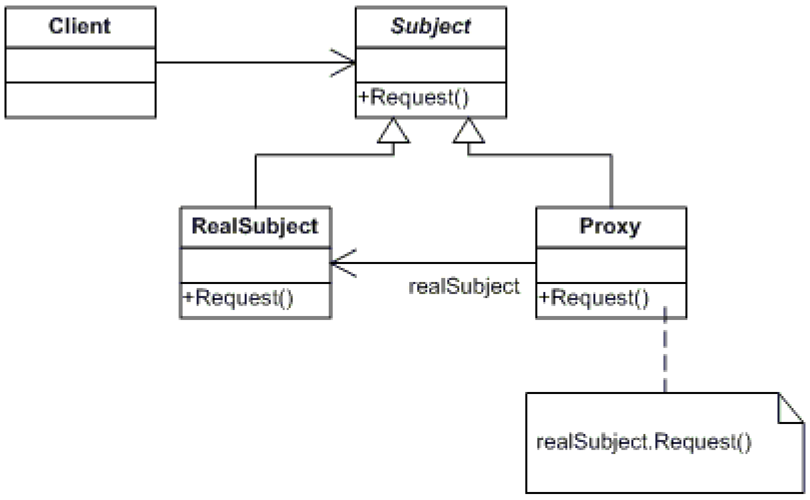
1. Participants

- Common Interface
- Source Object
- Proxy Object
- Client

2. Structure

- Identify Interface of the Source Object
- Define Proxy class with same Interface
- Wrap Source class Object in Proxy Class Object
- Let Client use Proxy Object instead of Real Object
- Proxy Object may filter request and optionally user Source Object to process it.

3. UML Diagram



Sample Code (Structure)

```
class IActionInterface
{
        public:
                virtual void Process()=0;
};

class RealClass : public IActionInterface
{
        public:
                void Process(){...}
}

class ProxyClass: public IActionInterface
{
                RealClass realObj;
        public:
                void Process()
                {
                        if(some_condition)
                                DoItYourSelf();
                        else
                                realObj->Process();
                }

                void DoItYourSelf(){...}
}

//Client
void main()
{
        IAction *pAction=new ProxyClass;
        pAction->Process(); //Provides controlled access to the real object

}
```

Common Scenario

- Distributed Proxy
    - Provides Access to a Remote Server as if it were a local resource
- Firewall Proxy
    - Provides Security Layer to an Unsecured Resource
- Cache Proxy
    - Provides cache support to a Resource that is expensive to access.
- Lazy Initialization Proxy
    - Delays Initialization of an expensive source object.

Relation with other Patterns

- Often Confused with
    - Adapter
        - Proxy publishes the same interface as real object; Adapter necessarily changes the interface.
    - Decorator
        - Proxy is static solution; Decorator is dynamic solution
            - Proxy is defined as class relationship.
            - Decorator is defined as instance relationship.
        - Proxy uses same interface; decorator may change interface.

S04: Decorator Pattern

Also Known as:

Onion Layer Architecture

What?

- Adds additional capabilities to an Object at Runtime using wrapper Objects
- Flexible alternative to sub classing

When?

- Sub classing is either impossible or undesirable.
- When we need to configure different set of capability to an object runtime depending or requirement.
- Need to apply different set of capabilities.
- You want to add state or behavior to a particular object rather than whole class.

Implementation

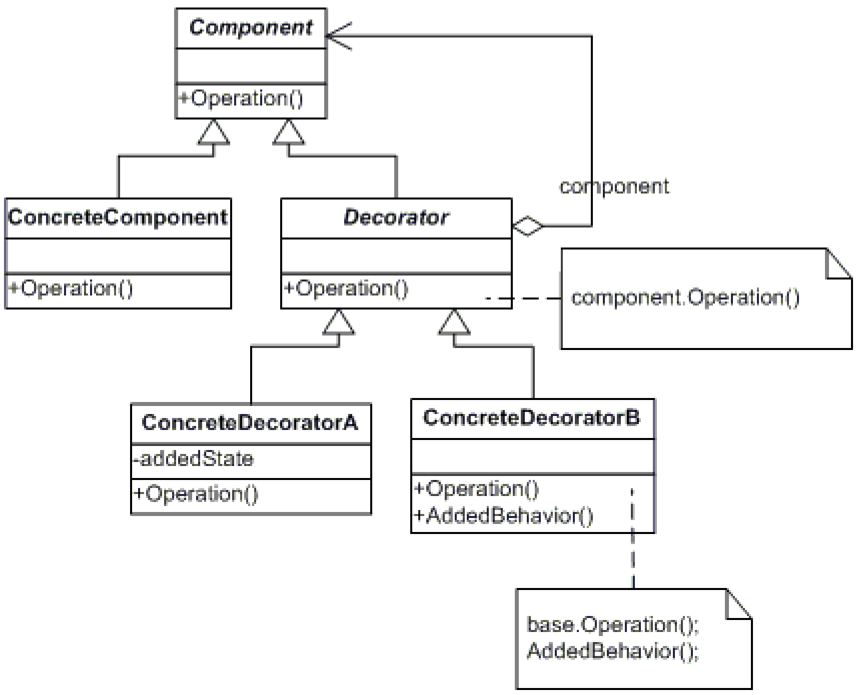
1. Participants

- Core Class (The object to be decorated)
- Decorator Class
- Client

2. Design Structure

- Create the Decorator class
    - which wraps a core object
    - Adds new set of state or behavior related to the core object
    - Uses the new capability in combination with the core object
    - Exposes the core object if required.

3. UML Diagram



Sample Code (Structure)

```
class Core
{
        public:
                void Action1(){...}
                void Action2(){...}
};

class IDecorator
{
        public:
                virtual Core * GetSource()=0;
};

class Decorator1:IDecorator
{
        Core *pSource;
public:
        Decorator1( Core *pSource ){this->pSource=pSource;}

        Core * getSource(){return pSource;}
        void NewAction1()    {...}
        void NewAction2()   {...}
};

class Decorator2:IDecorator
{
        Core *pSource;
public:
	Decorator2( Core *pSource ){this->pSource=pSource;}

	Core * getSource(){return pSource;}
	void AltAction1()	{...}
	void AltAction2()   {...}
};


//client
void main()
{
	Core * c1=new Core;
	Core * c2=new Core;
	Core * c3= new Core;

	Decorator1 d1(c1);
	Decorator2 d2(c1);

	d2.AltAction2(); //perform AltAction2()  with respect to c1
	d1.NewAction1(); //performs NewAction1()  with respect to c1
	d1.getSource()->Action1(); // performs Action1()  assoicated with c1

	Decorator2 d3(c2);
	d3.AltAction1(); // performs AltAction1() in context to c2
	Decorator1 d3( d3.getSource());	//configures a second decorator to the
source of first decorator

}
```

Well known Implementation

- JscrollPane class in Java swing package
- BufferedReader class in Java.IO package
- StreamWriter class in C# System.IO namespace

Relation with other patterns

- Often confused with
  - Proxy
    - Proxy is a static solution; Decorator a dynamic solution
  - Builder
    - Builder is a creational pattern and a static solution; Decorator is Structural pattern and dynamic solution.

S05: Composite Pattern

What?

- Compose objects into tree structures to represent part-whole hierarchies
- Whole and Part implements same interface.
- Composite lets clients treat individual objects and compositions of objects uniformly.
- Recursive Structure.
    - Leaf element (non-recursive end point)
    - Branch Element (Composite recursive end point)
- 1-to-many “has a” up the “is a” Hierarchy.

When?

- There are structures which naturally conform to Composite designs.

Where?

- Directory Hierarchy
    - A Directory can contain Files (Leaf)
    - Sub Directories (Composite structure)
- UI Design
    - Container (A Component that can contain more components)
    - Other Components (such as Button, TextBox acts as Leaf)

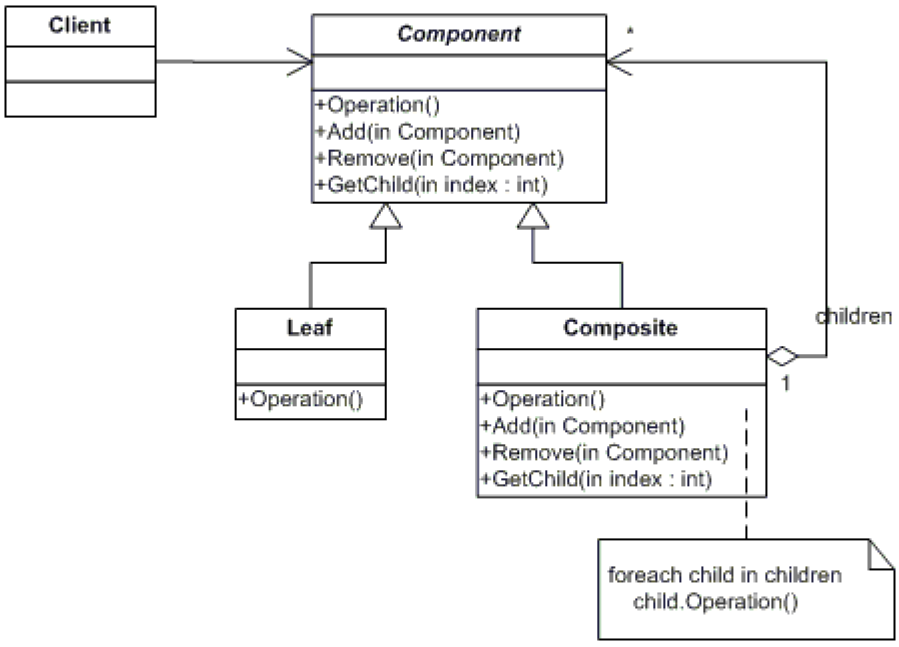
Implementation

1. Participants

- ILeaf Interface
    - Implemented by both Branch and Leaf
- IComposite Interface
    - extends ILeaf interface
    - Adds functionality to AddChild elements
    - Implemented by Branch elements
- NonComposite Elements
    - Implements ILeaf
- Composite elements

    - Implements IComposite Interface
- Composite Object
    - A root element implementing IComposite structure
- ILeaf operation performed on IComposite element is executed for each Child components

2. UML Diagrams



Sample Codes [Structure]

```
class ILeaf
{
public:
        virtual void Action()=0;
};

class IComposite:public ILeaf
{
public:
        virtual void AddChilds(ILeaf *element)=0;
};

class Leaf: ILeaf
{
public:
        void Action(){...}
};

public Composite:IComposite
{
        CPtrArray childs;
public:
        void AddChilds(ILeaf * element)
        {
                childs.Add(element);
        }

        void Action()
        {
                for(int i=0; i<childs.Count(); i++)
                {
                        ILeaf * e=(ILeaf *) childs[i];
                        e->Action();
                }
        }
}


void main()
{
        ILeaf *l1=new Leaf;
        ILeaf *l2=new Leaf;
        ILeaf *l3=new Leaf;

        IComposite *c1=new Composite;
        c1.AddChilds(l1);
        c1.AddChilds(l2);

        IComposite *c2=new Composite;
        c2.AddChilds(c1);
        c2.AddChilds(l3);

        c2->Action(); // Recursive calls Action for each components.
}
```

Relation with other patterns

- Often works with Chain Of Responsibility.
- Often constructed using Builder or Interpreter Pattern.
- A composite pattern often works with Iterator and Visitor pattern.

Part 03
Behavioral Patterns

Motivations:

- Behavioral patterns are concerned with algorithms and the assignment of responsibilities between objects.
- Behavioral patterns describe the patterns of communication between objects.
- They shift your focus away from flow of control to let you concentrate just on the way objects are interconnected.

GOF Behavioral Patterns:

- Strategy
- Command
- State
- Observer
- Iterator
- Visitor
- Interpreter*
- Template*
- Chain of Responsibility*
- Memento*
- Mediator*

* Not Covered

B01: Strategy Pattern

Also Known as

Policy Pattern

what?

- Encapsulates and Algorithm in a class.
- Allows a family of algorithm to be used interchangeably.
- Lets algorithm vary independently of client and API used.
- Customized Generic component to specific scenario.

When?

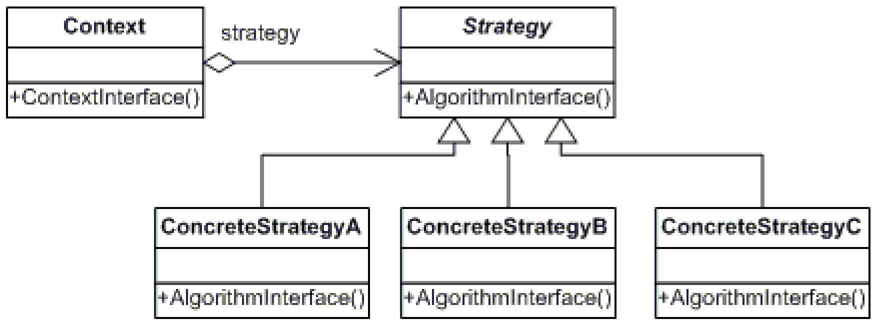
- You want to customize a generic API to meet a specific client requirement.
- When there are several possible outcomes of an application in the same context.
- When possibilities of various requirement cannot be predicted while designing the generic component.

Implementation

1. Participants

- Strategy Interface
- Concrete Strategy
- Context (Generic Component)
- Client

2. UML Diagram



Sample Code [Real life]

```
class IDateFormatter
{
        public:
                virtual CString Format(int dd, int mm, int yy, int hrs, int min, int sec)=0;
};

class Date
{
        int d,m,y,h,m,s;
        public:
                void PrintDate(IDateFormatter *fmt)
                {
                        cout<<endl<< fmt->Format(d,m,y,h,m,s);
                }
};

class DDMMYYFormatter :IDateFormatter
{
        public:
                CString Format(int dd,int mm,int yy, int hrs, int min,int sec)
                {
                        CString str;
                        str.Format("%02d/%02d/%4d",dd,mm,yy);
                        return str;
                }
};

class TimeFormatter: IDateFormatter
{
        public:
                CString Format(int dd, int mm, int yy, int hrs, int min, int sec)
                {
                        CString str;
                        str.Format("%02d:%02d %s", (hrs%12==0?12:hrs%12), min,
(hrs>=12?"pm":"am"));
                }
};
void main()
{
        Date dt=Date::Today();

        cout<<"todays date is ";
        dt.PrintDate( new DDMMYYformater );

        cout<<"todays time is ";
        dt.PrintDate(new TimeFormatter);

}
```

Relation with Other Patterns

- Often confused with
    - Command Pattern
        - Command Represents a Parameterized Action; Strategy A policy to specialize a Components

- We may often Specify a Strategy for a Command or a Visitor.
- We may apply a Strategy to select an Abstract Factory
- Strategy and Template have similar intent but different approach
    - Strategy uses composition; template uses inheritance
- Factory method often visualized as Strategy pattern in creation.

B02: Command Pattern

Also Known as

Action or Transaction

What?

- Represents a Request with all its parameter.
- Object Oriented Representation of Parameterized Method.
- Represents a single operation and all its parameters.
- Object oriented equivalent of a Call back function

When?

- You want to represent an Action with its parameters
- Parameters may require to be collected
    - Over a period of time
    - Via a series of complex interaction
- Object may need to exist for a long time
- Need to support operations such as
    - Queuing
    - Pooling
    - Undo Capability
    - Transactions
    - Serialization and Passing across network

Scenario

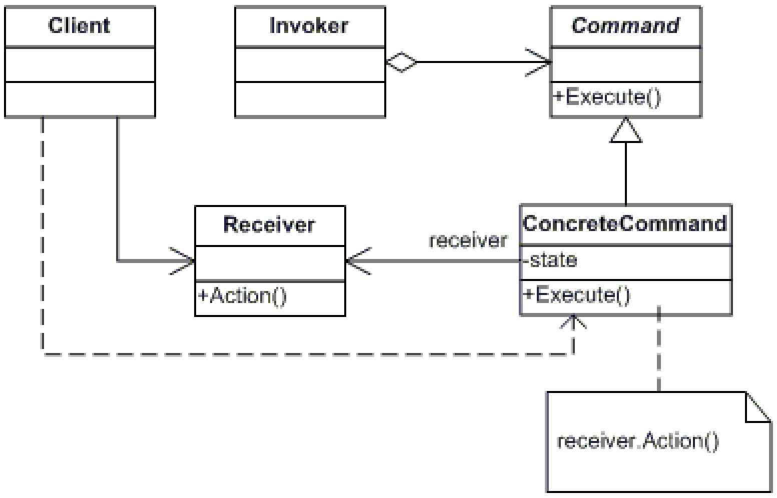
- A Print Command
- A Thread Pool Command
- A PlayerAction in multi-player game
- A Wizard
- A Progress bar
- Macro Recording

Implementation

1. Participants

- Command
- Creator
  - Creates the command
- Invoker
  - Requests the command to execute
- Receiver
  - Command fired in the Receiver Context

2. UML Diagram



Sample Code

```
class PrintCommand
{
                PrintDocument * pDoc;
        public:
                ExecutionStatus GetStatus(){...}
                void SetStatus(ExecutionStatus s){...}
                void SetDocument(PrintDocument *pDoc){...}
                void SetPriority(PrintPriority priority){...}
                PrintPriority GetPriority(){....}

                //the main command method
                void Print()
                {
			}
}

//Context or Receiver
class PrintManager
{
	PriorityQueue printQueue;
	public:
		void AddToQueue(PrintCommand *cmd){...}
		void UpdateQueue(){...}
		void RemoveFromQueue(PrintCommand *cmd){...}
		void Pause(){...}
		void Flush(){...}
		void Start(){...}
}

//Creator and Sender

void main()
{
	PrintManager m;

	PrintCommand *cmd1=new PrintCommand;
	cmd1->SetDocument( new WordDocument("d:/hello.doc"));
	cmd1->SetPriority(PrintPriority.Normal);
	m.AddToQueue(cmd1);

	PrintCommand *cmd2=new PrintCommand;
	cmd2->SetDocument(new TextDocument("d:/prog.cpp"));
	cmd2->SetPriority(PrintPriority.AboveNormal);
	m.AddToQueue(cmd2);

	m.Start();

	cmd1->SetPriority(PrintPriority.Highest);
	m.Pause();
	m.UpdateQueue();
	m.Start();
}
```

Relation with Other Patterns

- Chain of Responsibility often builds on Command Object
- Command may use Memento to provide undo operation
- Command may act as prototype to maintain a History List
- Often confused with strategy
    - Strategy is often short lived; Command is a long lasting object

B03: State Pattern

What?

- Allow an object to alter its behavior when it's internal state changes.
- The object appears to change its class.
- A state machine mechanism

Procedural Analog

- A scalar value typically with names like state, status or flag
    - typically integer or boolean
- The values used with nested ifs or switch case to take actions.

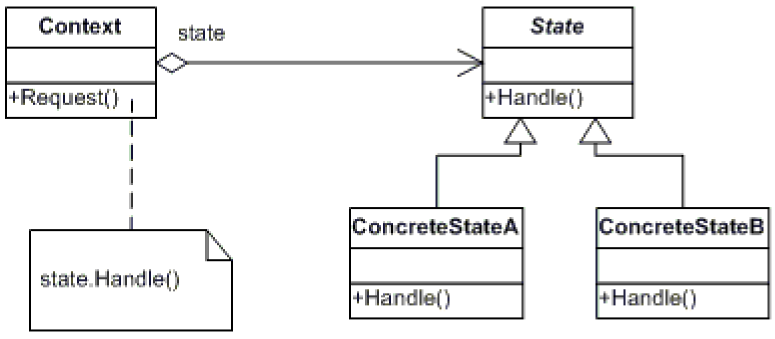
Popular Scenario

- A bank account may provide different features and facility depending on Account State
- A Player in game appears to take different kind of Action depending on Situation.
- A Network connection State
- A Command Execution Status

Implementation

- Encapsulate State and Facilities as a single entity
- Apply the state to Object
- Let Object call the behavior from its State object

UML Diagram



Sample Code [Real life example]

```
class IState
{
        public:
                virtual void Action()=0;

}

class Attack: IState
{
        public:
                void Action(){ ...}
}

class Defend:IState
{
        public:
                void Action(){...}
};

class Run: IState
{
        public:
                void Action(){...}
};

class Tank
{
        IState state;
public:
        void SetState(IState state){this.state=state;}

        void NextStep()
        {
                state.Action();
        }
};

void main()
{
        Tank *t1=new Tank;
        t1->SetState(Defend.GetInstance());

        Tank *t2=new Tank;
        t2->SetState(Attak.GetInstance());

        t1.NextStep();

        t2.NextStep();

}
```

Relation with other Patterns

- State are often represented as Singleton or Flyweight
- Similar in structure with Command, Strategy
    - differ in Intent
    - Strategy is bind once; state is more dynamic

B04: Observer Pattern

Also known as:

publisher/subscriber pattern

What?

- Define a one-to-many dependency between objects such 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
- Encapsulates the variable (or optional or user interface) components in an Observer hierarchy

Where?

- Event Delegation Framework
- MVC Architecture
- Publisher/Subscriber Patterns
- Implementing full duplex or Asynchronous communication.
- Avoiding Polling cost.

How? (Implementation)

1. Participants

- Subject (Interface)
    - AddObserver()
    - RemoveObserver()
    - NotifyChange()
- Concrete Subject
    - Implements subject in context specific scenario
- Observer Interface
    - GetUpdate()
- Concrete Observers
    - Implement behaviors depending on scenario.

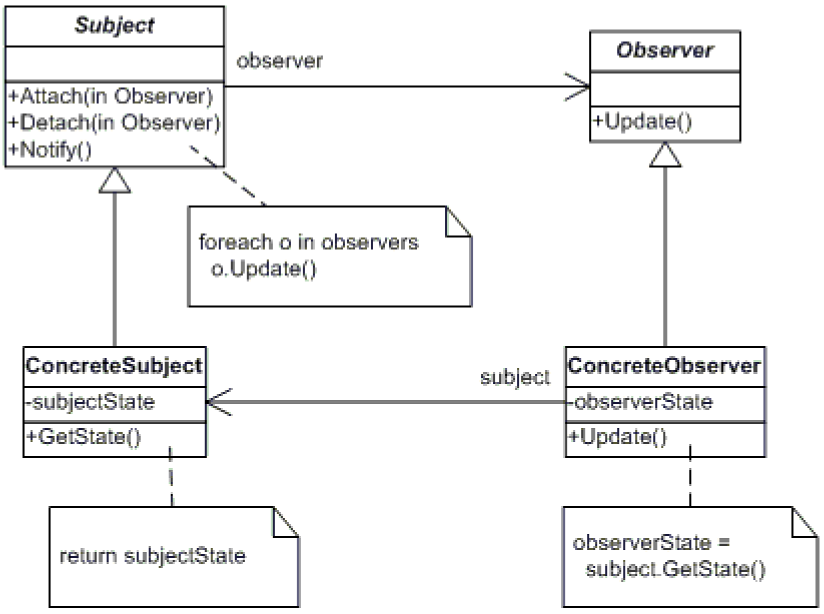
2. Design Structure

- Observers Subscribe to the Subject using AddObserver method
  - They can unsubscribe notification using RemoveObserver
- When the subjects state change the fire *GetUpdate*() for every subscriber Observers

Language Specific Implementation

- Java provides standard Observer-Observable API to implement the same.
- C# provides language construct of form events and delegates.

UML Diagram



Sample Code [Structure]

```
class IObserver
{
public:
        virtual void GetUpdate(VPTR state)=0;
};

class ISubject
{
public:
        virtual void Add(IObserver * o)=0;
        virtual void Remove(IObserver * o)=0;
        virtual void Update()=0;
};
```

```
class AbsSubject:ISubject
{
        MyState state;
        List<IObserver *> lst;
public:
        void Add(IObserver * o)
        {
                lst.Add(o);
        }

        void Remove(IObserver * o)
        {
                lst.Remove(o);
        }

        void Update()
        {
                for(int i=0;i<lst.Count();i++)
                        lst[i]->GetUpdate(state->Clone());
        }

};


class Observer1:IObserver
{
        public:
                void GetUpdate(VPTR state){...}
};

class Observer2: IObserver
{
        public:
                void GetUpdate(VPTR state){...}
};

//main
void main()
{
        Subject s;

        s.Add(new Observer1);
        s.Add(new Observer2);

        s.Update();


}
```

Relation with Other Patterns

- Observer and Mediator and competing patterns
    - Observers distribute communication; Mediator encapsulates communication
- Observer is easier to implement in a standard reusable ways.

B05: Iterator Pattern

Also known as:

Cursor

What?

- Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation
- Polymorphic traversal
- encapsulates the internal structure of How Iteration occurs.

When?

- We want to access a collection or a composite structure
- We want a sequential access.
- Client need not be concerned about the internal structure of the object being iterated.

Implementations

1. Participants

- Iterator Interface
- Concrete Iterator
- Aggregate Class
    - Defines and exposes its own Iterator Concrete Class
- Client

2. Design Structure

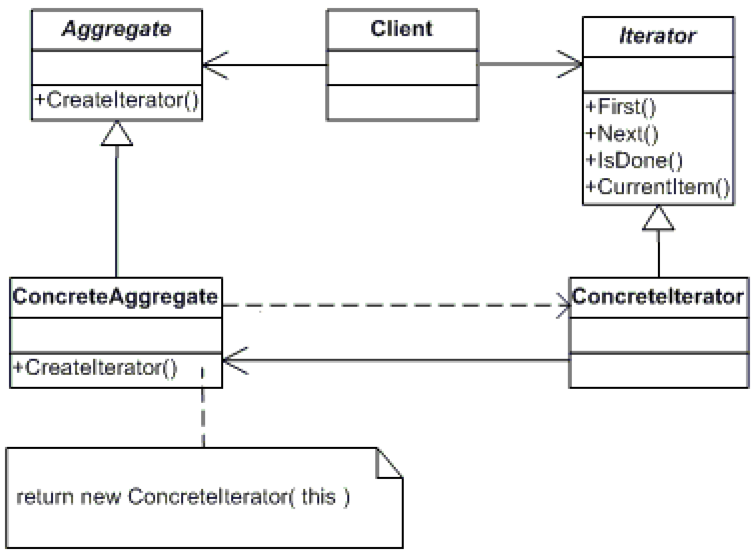
- Client Requests Aggregate Class to Provide with an Iterator Object
- Aggregate Class creates an Iterator from the concrete class and returns
- Client accesses the Iterator using Interface and never needs to know actual concrete class

3. Language Specific Implementation

- C++ Provides Iterator idea in its STL and as Smart Pointer
- Java Provider Iterator interface
- C# provide IEnumerator interface

- C# and Java provides for-each loop to support Iteration.

4. UML Diagram



Sample Code [Real World]

```
template<class T>
class IIterator
{
public:
	virtual void First()=0;
	virtual void Next()=0;
	virtual bool IsEnd()=0;
	virtual T GetCurrent()=0;
};

class EvenIterator:IIterator<int>
{
	int *arr;
	int count;
	int ndx;

	void Check()
	{
		if(ndx<0 || ndx>count)
		{
			ndx=-1;
			return;
		}

		if(arr[ndx]%2!=0)
		{
```

```
                ndx++;
                Check();
            }

        }
public:
        EventIterator(int *arr, int count)
        {
            this->count=count;
            this->arr=arr;
            ndx=-1;
        }

        void First()
        {
            ndx=0;
            Check();
        }

        void Next()
        {
            ndx++;
            Check();
        }

        bool IsEnd()
        {
            return ndx>=0 && ndx<count;
        }

        int GetCurrent()
        {
            if(IsEnd)
                throw new Exception;
            else
                return arr[ndx];
        }
}

class Aggregate
{
        int * items;
        int count;
public:
        Aggregate(int *item,int count)
        {
            this->item=item;
            this->count=count;
        }
        IIterator<int> GetEvenIterator(int type){ return new EventIterator(items,count);}
```

```
};

void main()
{
	int []x={5,3,2,1,8,6,4,2};

	Aggregate a(x,8);

	IIterator<int> *it=a.GetEvenIterator();

	for(it->First(); !it->IsEnd(); it->Next())
		cout<<endl<<it->GetCurrent();


}
```

Relation with Other Patterns

- Often a competing interface with Visitor interface
- Need Factory method to create polymorphic Iterator
- Can Traverse a composite
- A memento can be used to keep Iteration State

B06: Visitor Pattern

What?

- Defines new operation to an existing class
- Separates behavioral (or operation) from the structure of a class

When?

- When you want to provide extensible operation on a Aggregate
- When the behavior defined need to be independent of the structure of Aggregate
- When we need to apply same behavior on different structures

Implementation

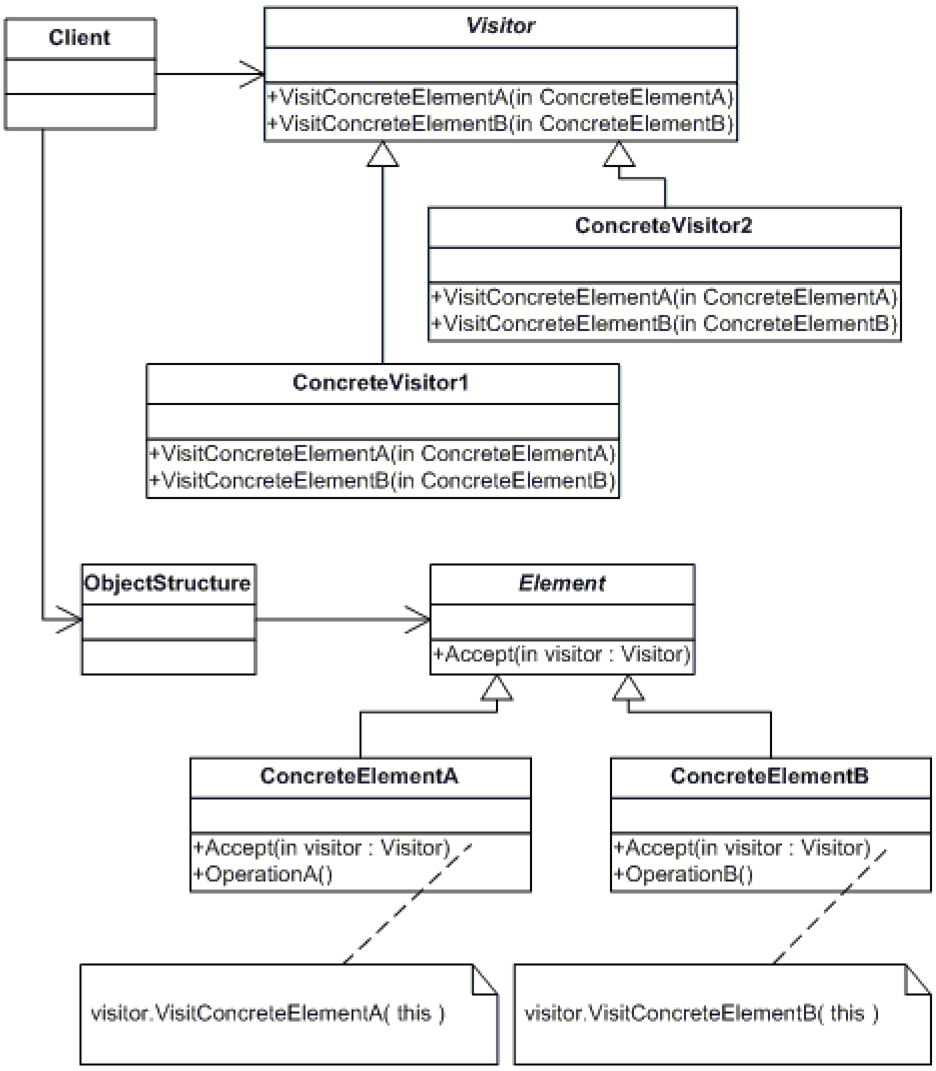
1. Participants

- Visitor Interface
    - Defines Visit(LPVOID state) method
- Concrete Visitors
    - Allowed to be selected polymorphically
- IVisitable Interface
    - Defines Accept(Visitor v) method
- Aggregate Class
    - Implements IVisitable interface

2. Design Structure

- Create concrete visitor
- Pass visitor to Accept() method of Aggregate

2. UML Diagram


Client
Visitor
+VisitConcreteElementA(in ConcreteElementA)
+VisitConcreteElementB(in ConcreteElementB)
ConcreteVisitor2
+VisitConcreteElementA(in ConcreteElementA)
+VisitConcreteElementB(in ConcreteElementB)
ConcreteVisitor1
+VisitConcreteElementA(in ConcreteElementA)
+VisitConcreteElementB(in ConcreteElementB)
ObjectStructure
Element
+Accept(in visitor : Visitor)
ConcreteElementA
+Accept(in visitor : Visitor)
+OperationA()
ConcreteElementB
+Accept(in visitor : Visitor)
+OperationB()
visitor.VisitConcreteElementA(this)
visitor.VisitConcreteElementB(this)

Sample Code [Real World]

```
template<class T>
class IVisitor
{
public:
        virtual void Visit(T state)=0;
};

template<class T>
class IVisitable
{
public:
        virtual void Accept(IVisitor<T> v)=0;
};

class Visitor1: IVisitor<int>
{
	int count;
public:
	Visitor(){count=0;}
	int GetCount(){reutrn count;}

	void Visit(int state)
	{
		count++;
	}
};

class Visitor2:IVisitor<int>
{
	public:
		void Visit(int state)
		{
			cout<<endl<<state;
		}
}

class Aggregate:IVistable<int>
{
	int *arr;
	int count;
public:
	Aggregate(int *arr, int count)
	{
		this->arr=arr;
		this->count=count;
	}

	void Accept(IVisitor<int> v)
	{
		for(int i=0;i<count;i++)
			v->Visit(arr[i]);
	}
};

void main()
{
	int arr[]={5,2,9,3,1};
	Aggregate x(arr,5);

	Visitor v1;
	x.Accept(v1);
	cout<<endl<<"Total elements "<<v1.GetCount();

	cout<<endl<<"elements are :"
	x.Accept(Visitor2);
}
```

Relation with Other Patterns

- Often a competing pattern with Iterator
    - Iterator is more tightly coupled with the Aggregate than Visitor
- Visitor is more powerful than Command; but works for a specific scenario.