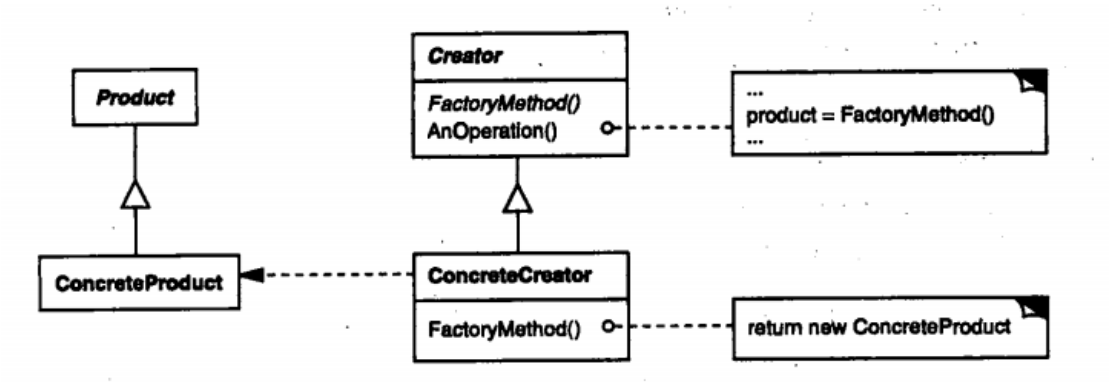
1. Factory 工厂模式:

A) 用途：

基类定义一个创建对象的接口，其调用基类的抽象方法，而实际调用的是根据自己初始化的具体创建者所决定的方法。

B) UML 结构图：



C) 抽象基类：

1)Product:创建出来的对象的抽象基类.

2)Creator:创建对象的工厂方法的抽象基类.

D) 接口函数：

1)Creator::FactoryMethod:纯虚函数,由派生类实现,创建出对应的 Product.

E) 解析：

Product 为产品的抽象基类，Creator 为创建者的抽象基类。

Product 与 Creator 其派生类有着严格的对应关系，多一种 Product，就需要多一种 Creator

F) 实现：

```
/**
    Factory 模式的实现
    File        :factory.cpp
    Author      :singmelody
    Date        :2012.11.12
*/
#include <iostream>
using namespace std;

class Product
{
public:
```

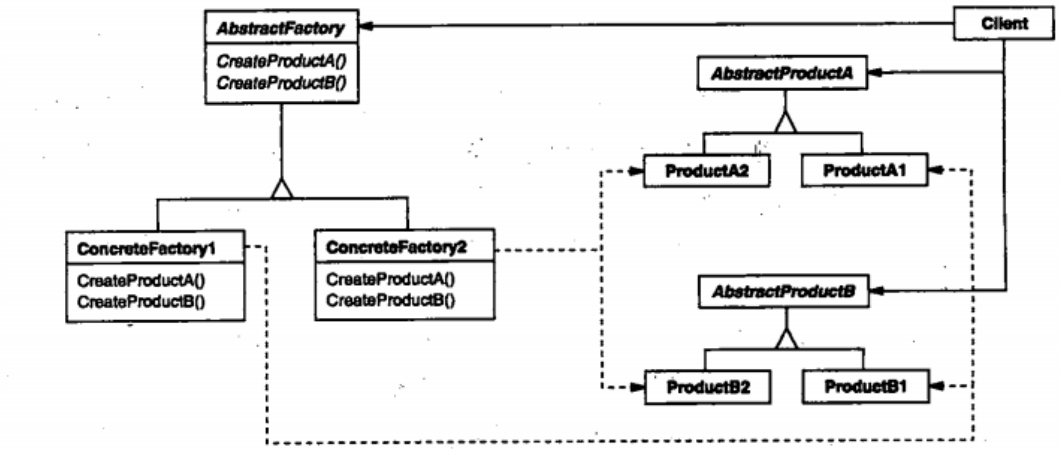
```
    virtual ~Product(){ cout<<__FUNCTION__<<endl; }
};

class ProductA : public Product
{
public:
    ~ProductA() { cout<<__FUNCTION__<<endl; }
};

class Creator
{
public:
    virtual ~Creator(){ cout<<__FUNCTION__<<endl; }
    Product* AnOperation(){    return FactoryMethod();    }

protected:
    virtual Product* FactoryMethod() = 0;

};

class CreatorA : public Creator
{
public:
    ~CreatorA() { cout<<__FUNCTION__<<endl; }

protected:
    Product* FactoryMethod()
    {
        cout<<__FUNCTION__<<endl;
        return new ProductA();
    }
};

int main()
{
    Creator *creator = new CreatorA();
    Product *product = creator->AnOperation();

    delete product;
    delete creator;

    return 0;
}
```

2. Abstract Factory 抽象工厂模式：

A) 用途：

提供一个创建一系列产品的接口。具体的工厂产生一系列具体的产品。

B) UML 图：



C) 抽象基类：

1) ProductA,ProductB ： 分别代表不同类型的产品，而他们的派生类则是这种产品的一个实现。

2) AbstractFactoery ： 生产一系列产品的一个抽象工厂，其派生类是具体产生一系列不同的产品。

D) 接口函数：

1）AbstractoryFactoery::CreateProductA 和 AbstractFactrory::CreateProductB: 分别是生产不同产品的不同的实现，由各个派生出来的抽象工厂实现之。

E) 解析：

正如名字的差别，抽象工厂模式比工厂模式更为抽象了，之前的工厂模式通过继承 Product 和 Creator，使用基类的抽象接口产生一种产品。现在的抽象工厂模式，产生一系列的产品，AbstracFactory 派生出来类也是产生一系列的产品，其中创建的实现其实采用的就是 Factory 模式。

下面是一位前辈举得一个生动的例子：

比如,同样是鸡腿(ProductA)和汉堡(ProductB),它们都可以有商店出售(AbstractFactory),但是有不同的实现,有肯德基(ConcreateFactory1)和麦当劳(ConcreateFactory2)两家生产出来的不同风味的鸡腿和汉堡(比如鸡腿有肯德基的香辣鸡，麦当劳的麦乐鸡；汉堡有肯德基的新奥堡，麦当劳的巨无霸等等). 而负责生产汉堡和鸡腿的就是之前提过的 Factory 模式了.

如果有 n 种产品同时有 m 中不同的实现,那么根据乘法原理可知有 n*m 个 Factory 模式的使用.

F) 代码实现：

```
/**
    Abstract Factory 模式的实现
    File        :abstract factory.cpp
    Author      :singmelody
    Date        :2012.11.18
*/
#include <iostream>
using namespace std;

class AbstractProductA
{
public:
    virtual ~AbstractProductA(){}
};

class ProductA1 : public AbstractProductA
{
public:
    ProductA1() { cout<<"ProductA1()"<<endl; }
    ~ProductA1() { cout<<"~ProductA1()"<<endl; }
};

class ProductA2 : public AbstractProductA
{
public:
    ProductA2() { cout<<"ProductA2()"<<endl; }
    ~ProductA2() { cout<<"~ProductA2()"<<endl; }
};

class AbstractProductB
{
public:
    virtual ~AbstractProductB(){}
};

class ProductB1 : public AbstractProductB
{
public:
    ProductB1() { cout<<"ProductB1()"<<endl; }
    ~ProductB1() { cout<<"~ProductB1()"<<endl; }
};

class ProductB2 : public AbstractProductB
{
```

```
public:
    ProductB2() { cout<<"ProductB2()"<<endl; }
    ~ProductB2() { cout<<"~ProductB2()"<<endl; }
};


class AbstractFactory
{
public:
    virtual AbstractProductA* CreateProductA() = 0;
    virtual AbstractProductB* CreateProductB() = 0;
};

class ConcreateFactory1 : public AbstractFactory
{
public:
    AbstractProductA* CreateProductA(){  return new ProductA1;}
    AbstractProductB* CreateProductB(){  return new ProductB1;}
};

class ConcreateFactory2 : public AbstractFactory
{
public:
    AbstractProductA* CreateProductA(){  return new ProductA2;}
    AbstractProductB* CreateProductB(){  return new ProductB2;}
};


int main()
{
    AbstractFactory * factory = new ConcreateFactory1;
    AbstractProductA* productA = factory->CreateProductA();
    AbstractProductB* productB = factory->CreateProductB();
    delete productA;
    delete productB;
    delete factory;

    factory = new ConcreateFactory2;
    productA = factory->CreateProductA();
    productB = factory->CreateProductB();
    delete productA;
    delete productB;
    delete factory;
```

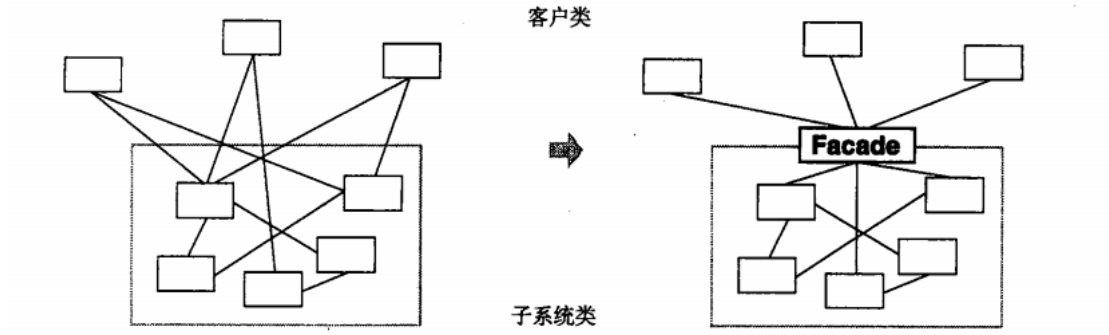
```
    return 0;
}
```

3. Facade 外观模式：

A) 用途：

Facade 可以提供一个简单的缺省视图，方便一般用户直接使用。这一视图对一般用户来说已经足够。而更多的需要的可定制的用户可以越过 facade 层。

B) UML 图：



C) 抽象基类：

1) Facade:包含各个子系统的引用，实现系统的默认行为，以方便使用。

D) 接口函数：

1） Facade::startCompute(): 基类的调用接口，实现默认的方法。

E) 解析：

我有一个专业的 Nikon 相机，我就喜欢自己手动调光圈、快门，这样照出来的照片才专业，但 MM 可不懂这些，教了半天也不会。幸好相机有 Facade 设计模式，把相机调整到自动档，只要对准目标按快门就行了，一切由相机自动调整，这样 MM 也可以用这个相机给我拍张照片了。 通常来说仅需要一个 Facade 对象，因此它通常属于 Singleton

F) 代码实现：

```
/**
    Facade 模式的实现
    File      :Facade.cpp
    Author    :singmelody
    Date      :2012.11.18
*/
#include <iostream>
using namespace std;

class CPU
```

```
{
public:
    void freeze() { cout<<__FUNCTION__<<endl; }
    void jump(long position) {  cout<<__FUNCTION__<<endl; }
    void execute() { cout<<__FUNCTION__<<endl; }
};

class Memory {
public :
    void load(long position, char  data[]) { cout<<__FUNCTION__<<endl; }
};

class HardDrive {
public:
    void read(long lba, int size) { cout<<__FUNCTION__<<endl; }
};

class Facade
{
private:
    CPU*     cpu;
    Memory*      memory;
    HardDrive*   hardDrive;

public:
    void Compute()
    {
         cpu = new CPU();
        memory = new Memory();
        hardDrive = new HardDrive();
    }

    void startComputer() {
        cpu->freeze();
        memory->load(0,"hello");
        cpu->jump(0);
        cpu->execute();
    }
};

int main()
{
    Facade* facade = new Facade;
    facade->startComputer();
```

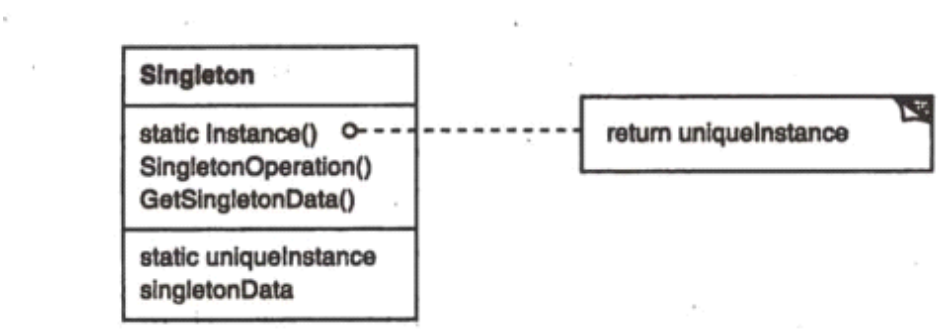
```
    return 0;
}
```

4. Singleton 单例模式：

A) 用途：

保证一个类仅有一个实例，并提供一个访问它的全局访问点。

B) UML 图：



C) 抽象基类：

Singleton：可以使用模板使 Singleton 作为单独基类，也可以在具体类的实现中实现 Singleton 的功能，可以通过 Instance()函数来获取类的全局唯一实例。

D) 接口函数：

Singleton::Instance()：可以通过 Instance()函数来获取类的全局唯一实例。

E) 解析：

Singleton 模式其实是对全局静态变量的一个取代策略，不直接使用静态变量的原因是因为，静态变量只有文件作用域，而且可以出现重复拷贝的情况，这样就失去了 Singleton 的最重要的特性。访问的时候可以直接通过类的函数作用符，这样就可以随时方便获取到类的实例，而不需要一直通过传参的方式，来增加类的耦合度。

一般的,如果一个项目中需要使用到 Singleton 模式比较多的话,那么一般会实现一个 Singleton 的模板类,模板类的模板参数是需要采用 Singleton 模式的类,如 boost 就有专门的 Singleton 类。

F) 代码实现：

普通实现：

```
/**
    Singleton 模式的实现
    File        :singleton.cpp
    Author      :singmelody
    Date        :2012.11.19
*/
```

```cpp
#pragma once

#include <iostream>
using namespace std;

class Singleton
{
public:
    static Singleton& Instance();
    static Singleton* InstancePtr();

    void Test();
private:
    Singleton(){}
    ~Singleton(){}
    Singleton(Singleton &);
    Singleton & operator=(Singleton &);
};

Singleton& Singleton::Instance()
{
    static Singleton* local = NULL;
    if(local == NULL)
        local = new Singleton();

    return *local;
}

Singleton* Singleton::InstancePtr()
{
    return &Singleton::Instance();
}

void Singleton::Test(){    cout<<__FUNCTION__<<endl; }


int main()
{
    Singleton::InstancePtr()->Test();
    return 0;
}
```

Singleton 模板类：

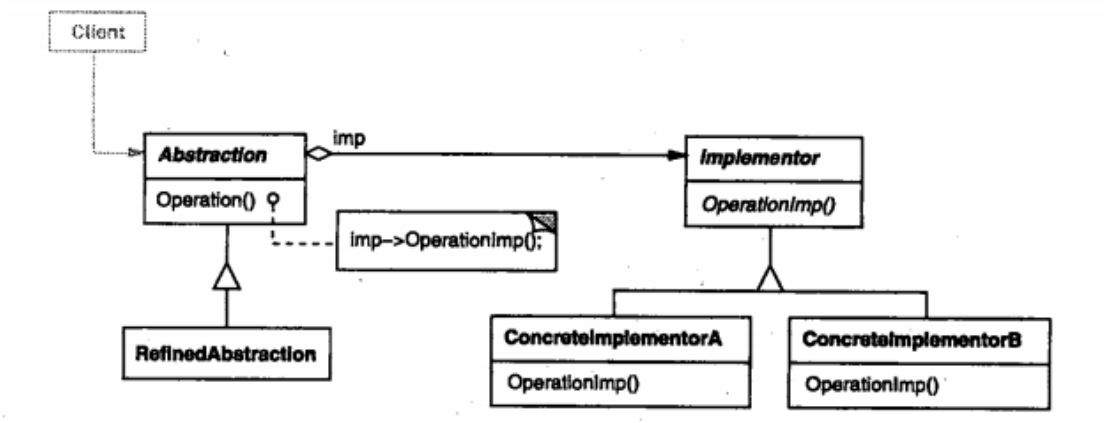
```
/**
    Singleton模式的实现
    File      :singleton.cpp
    Author        :singmelody
    Date      :2012.11.19
*/
#include <iostream>
using namespace std;

template <class T>
class Singleton
{
public:
    static inline T& getInstance()
    {
        static T _instance;
        return _instance;
    }
protected:
    Singleton() {}
    ~Singleton() {}
    Singleton( const Singleton< T >& );
    Singleton< T >& operator= ( const Singleton< T > & );
};


class Test : public Singleton<Test>
{
public:
     void echo()
     {
          cout<<__FUNCTION__<<endl;
     }
};

int main()
{
     Test::getInstance().echo();
     return 0;
}
```

5. Bridge 桥接模式：

A) 用途：

将抽象部分与它的实现部分分离，使它们都可以独立地变化。

B) UML 图：



C) 抽象基类：

1) Abstraction :某个抽象类,它的实现方式由具体的 Implementor 完成.

2) Implementor :实现类的抽象基类,定义了实现 Abstraction 的基本操作,而它的派生类实现这些接口.

D) 接口函数：

1) Implementor::OperationImpl: 定义了为实现 Abstraction 需要的基本操作,由 Implementor 的派生类实现之,而在 Abstraction::Operation 函数中根据不同的指针多态调用这个函数.。

2) Abstraction::Operation():如果需要重写 Operation()，但是还是需要保持基类特性，可以调用基类的方法，这样既可以多态的调用具体的 X::OperationImpl()，而且还可以对基类的 Abstraction::Opertion() 方法做出改变。

E) 解析：

Bridge 用于将表示和实现解耦,两者可以独立的变化.在 Abstraction 类中维护一个 Implementor 类指针,需要采用不同的实现方式的时候只需要传入不同的 Implementor 派生类就可以了.

Bridge 的实现方式其实和 Builder 十分的相近,可以这么说:本质上是一样的,只是封装的东西不一样罢了。.两者的实现都有如下的共同点:抽象出来一个基类,这个基类里面定义了共有的一些行为,形成接口函数(对接口编程而不是对实现编程),这个接口函数在 Builder 中是 BuildePart 函数在 Bridge 中是 OperationImpl 函数;其次,聚合一个基类的指针,如 Builder 模式中 Director 类聚合了一个 Builder 基类的指针,而 Bridge 模式中 Abstraction 类聚合了一个 Implementor 基类的指针(优先采用聚合而不是继承);而在使用的时候,都把对这个类的使用封装在一个函数中,在 Bridge 中是封装在 Director::Construct 函数中,因为装配不同部分的过程是一致的,而在 Bridge 模式中则是封装在 Abstraction::Operation 函数中,在这个函数中调用对应的 Implementor::OperationImpl 函数.就两个模式而言,Builder 封装了不同的生成组成部分的方式,而 Bridge 封装了不同的实现方式.

这也反映了 Bridge 模式作为一种结构型的模式，负责解耦结构本身，而不负责创建的本身。而 Builder 模式作为一种创建型模式，主要负责创建的解耦。当遇到相似的模式，可以用相似的方式要去理解，但更重要的是要找出其不同之处。

F) 代码实现：

```
/**
    Bridge模式的实现
    File      :Brige.cpp
    Author        :singmelody
    Date      :2012.11.19
*/
#include <iostream>
using namespace std;

class Implementor
{
public:
    virtual void OperationImp() = 0;
    virtual ~Implementor(){ cout<<__FUNCTION__<<endl;}
};

class ConcreteImplementorA : public Implementor
{
public:
    void OperationImp()   {   cout<<__FUNCTION__<<endl; }
    virtual ~ConcreteImplementorA() { cout<<__FUNCTION__<<endl; }

};

class ConcreteImplementorB : public Implementor
{
public:
    void OperationImp()   {   cout<<__FUNCTION__<<endl; }
    virtual ~ConcreteImplementorB() { cout<<__FUNCTION__<<endl; }
};
class Abstraction
{
public:
    Abstraction(Implementor                    *implementor):m_implementor(implementor){
    cout<<__FUNCTION__<<endl; }
    virtual ~Abstraction(){ cout<<__FUNCTION__<<endl;delete m_implementor; }

    virtual void OperationImpl()
    {
```

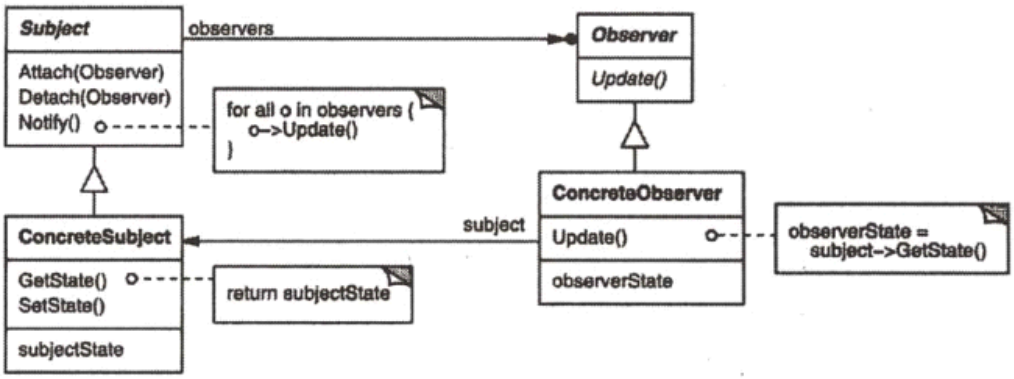
```
        cout<<__FUNCTION__<<endl;
        m_implementor->OperationImp();
    }
protected:
    Implementor* m_implementor;
};

class RefinedAbstraction : public Abstraction
{
public:
    RefinedAbstraction(Implementor              *implementor):Abstraction(implementor)
{ cout<<__FUNCTION__<<endl; }
    ~RefinedAbstraction(){ cout<<__FUNCTION__<<endl; }

    void OperationImpl(){ Abstraction::OperationImpl(); doSomething(); }
    void doSomething(){ cout<<"Hello World"<<endl;}
};

int main()
{
    Implementor* implentor = new ConcreteImplementorA();
    Abstraction* abstraction = new RefinedAbstraction(implentor);
    abstraction->OperationImpl();

    delete abstraction;
    return 0;
}
```

6. Observer 观察者模式：

A) 用途：

定义对象间的一种一对多的依赖关系，当一个对象状态发生改变的时候，所有依赖于它的对象都得到通知并被自动更新。也被称为出版者-订阅者模式

B) UML 图：



C) 抽象基类：

1) Subject ：提供添加和删除监听者的接口，当被监听者(Subject)状态改变的时候，可以通过 Notify()来调用 Object 的抽象方法 Update()，来更新监听者状态

2) Observer：提供 Update()接口，来接收 Subject 发生的状态改变。

D) 接口函数：

1) Observer::Update()：用来接收 Subject 或其派生类状态的改变。

2) Subject::Notify() ：有需要通知监听者的状态改变，需要调用此方法。

E) 解析：

Observer 模式定义的是一种一对多的关系，一为 Subject，多为 Observer。,当 Subject 类的状态发生变化的时候通知与之对应的 Obesrver 类们也去相应的更新状态,同时支持动态的添加和删除 Observer 对象的功能.因为可能需要频繁的添加和删除监听者，所以一般保存监听者指针的容器常用 list。

一般在实际应用中，Notify()常常别名为 Update()，Subject 往往为 Singleton。在游戏中当需要每帧更新时，调用 Subject 的 Update()方法(也就是 Notify())。来更新对应 Subject 下的所有监听者，比如所有需要渲染物体的的位置更新。

多个设计模式的使用是在实际应用中经常要面对的问题，要熟练的掌握设计模式，既要熟练应用，不要拘泥于模式本身。

F) 代码实现：

```
/**
    Observer模式的实现
    File      :observer.cpp
    Author    :singmelody
    Date      :2012.11.19
*/
#include <iostream>
#include <list>
#include <algorithm>
using namespace std;

#define FUNCINFO cout<<__FUNCTION__<<endl;
```

```
class Observer;

class Subject
{
public:
    enum SUBJECT_STATE
    {
        SUBJECT_STATE_WAIT,
        SUBJECT_STATE_WALK,
        SUBJECT_STATE_RUN
    };

    Subject(){ FUNCINFO }
    virtual ~Subject();

    void          Attach(Observer *observer);
    void          Detach(Observer *observer);
    void          Notify();

    SUBJECT_STATEGetState() { FUNCINFO return m_subjectState; }
    void          SetState(SUBJECT_STATEstate){  FUNCINFO m_subjectState = state; }
protected:
    list<Observer* > m_observers;
    SUBJECT_STATE    m_subjectState;
};

class ConcreteSubject : public Subject
{
public:
    ConcreteSubject(){ FUNCINFO }
    ~ConcreteSubject() { FUNCINFO }
};

class Observer
{
public:
    virtual void Update(Subject* subject) = 0;
    virtual ~Observer(){ FUNCINFO }
    Observer(){ FUNCINFO }
};

class ConcreteObserver : public Observer
{
public:
```

```
    ConcreteObserver() : m_observerState(-1){ FUNCINFO }
    ~ConcreteObserver() { FUNCINFO }
    void Update(Subject* subject);

private:
    int      m_observerState;
};

void Subject::Notify()
{
    FUNCINFO
    for(list<Observer *>::iterator itr = m_observers.begin();
        itr != m_observers.end(); ++itr)
    {
        (*itr)->Update(this);
    }
}

Subject::~Subject()
{
    FUNCINFO
    for(list<Observer *>::iterator itr = m_observers.begin();
    itr != m_observers.end();)
    {
        delete *itr;
        itr = m_observers.erase(itr);
    }
}

void Subject::Attach(Observer *observer)
{
    FUNCINFO
    m_observers.push_back(observer);
}

void Subject::Detach(Observer *observer)
{
        FUNCINFO

    list<Observer *>::iterator itr = std::find(m_observers.begin(),
        m_observers.end(),observer);

    if(itr != m_observers.end()){
        m_observers.erase(itr);
```

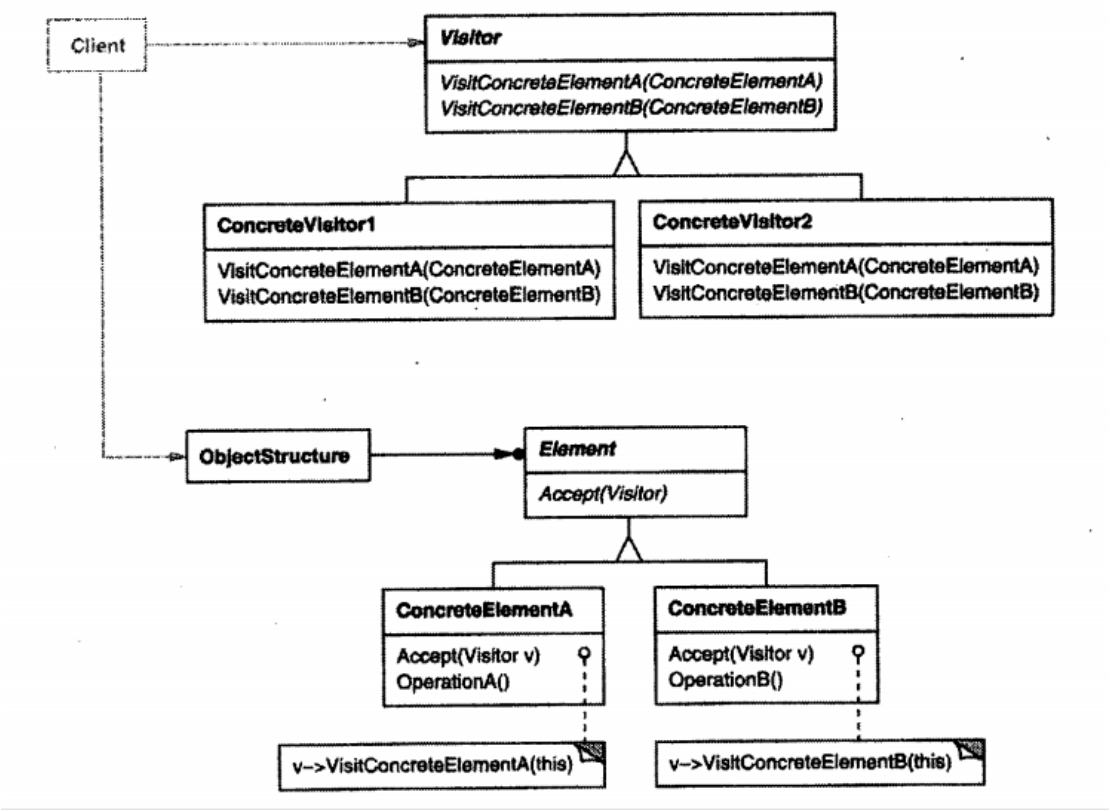
```
        cout<<"Detach Success"<<endl;
    }
    else{
        cout<<"Observer Not Found!"<<endl;
    }
}

void ConcreteObserver::Update(Subject *subject)
{
    FUNCINFO

    if(subject == NULL)
        return;

    m_observerState = subject->GetState();
}

int main()
{
    Observer *p1 = new ConcreteObserver;
    Observer *p2 = new ConcreteObserver;

    Subject* subject = new ConcreteSubject;
    subject->Attach(p1);
    subject->Attach(p2);
    subject->SetState(Subject::SUBJECT_STATE_WAIT);
    subject->Notify();

    subject->Detach(p1);
    subject->SetState(Subject::SUBJECT_STATE_RUN);
    subject->Notify();

    delete subject;
    return 0;
}
```

7. Visitor 访问者模式：

A) 用途：

表示一个作用于某对象结构中的各元素的操作。它使你可以在不改变各元素的类的前提下定义作用于这些元素的新操作。

B) UML 图：



C) 抽象基类：

1) Visitor ： 提供访问者的抽象基类，其中方法的数量决定于 Element 的多少。当新添加一个 Element 的时候，Visitor 需要提供相应的 VisitConcreteElementX() 方法。

2) Element ： 访问者需要访问的节点元素，提供抽象的 Accept()方法，不同的 Visitor 派生类访问不同的 Element 派生类。

D) 接口函数：

1） Element::Accept(): 提供抽象接口，根据具体 Element 派生类，来多态到不同的 ConcreteElement 类。

2） Visitor::VisitConcreteElementX() : 提供所有派生元素的访问方法。可以有朋友会疑问为什么不把所有的 VisitConcreteElementX() 合并为一个 VisitConcreteElement()方法。这样之后添加 Element 直接调用 Visitor::VisitConcreteElement() 就好，不用再为每次添加不同的元素再添加对应的的方法。这样理解是好的，但是要注意适用的前提，就是 VisitConcreteElement()中的操作是抽象的，可以合并在一起的。而对于不同的 Element 访问的方法肯定不一致，比如,你去老师家肯定要带点礼物吧，但是下班回家给女朋友一个吻就可以。这也反映了 Visitor 是根据不同的 Element 来设计的。

E) 解析：

Visitor 模式把对结点的访问封装成一个抽象基类,通过派生出不同的类生成新的访问方式.在实现的时候,在 visitor 抽象基类中声明了对所有不同结点进行访问的接口函数,如图中的 VisitConcreateElementA 函数等,这样也造成了 Visitor 模式的一个缺陷--新加入一个结点的时候都要添加 Visitor 中的对其进行访问接口函数,这样使得所有的 Visitor 及其派生类都要重新编译了。.另外,还需要指出的是 Visitor 模式采用了所谓的"双重分派"的技术,拿上图来作为

例子,要对某一个结点进行访问,首先需要产生一个 Element 的派生类对象,其次要传入一个 Visitor 类派生类对象来调用对应的 Accept 函数,也就是说,到底对哪种 Element 采用哪种 Visitor 访问,需要两次动态绑定才可以确定下来,具体的实现可以参考下面实现代码。

Visitor 适用的范围是 Element 相对稳定的情况下，产生不同的 Visitor，不改变 Element 接口的情况下来进行扩展。这也反映出了 OOP 中的开放封闭原则。

所谓的开放封闭原则主要是指对之前的代码结构下不进行改变(封闭)，但是可以通过增加类和方法来扩展功能(开放)。

F) 代码实现：

```
/**
    Visitor模式的实现
    File      :visitor.cpp
    Author        :singmelody
    Date      :2012.11.20
 */
#include <iostream>
#include <list>
#include <algorithm>
using namespace std;

#define FUNCINFO cout<<__FUNCTION__<<endl;

class ConcreteElementA;
class ConcreteElementB;

class Visitor
{
public:
    Visitor(){ FUNCINFO; }
    virtual ~Visitor(){ FUNCINFO; }

    virtual void VisitConcreteElementA(ConcreteElementA* elment) = 0;
    virtual void VisitConcreteElementB(ConcreteElementB* elment) = 0;
};

class ConcreteVisitor1 : public Visitor
{
public:
    ConcreteVisitor1() { FUNCINFO; }
    ~ConcreteVisitor1() { FUNCINFO; }
    void VisitConcreteElementA(ConcreteElementA* elment);
    void VisitConcreteElementB(ConcreteElementB* elment);
};
```

```
class ConcreteVisitor2 : public Visitor
{
public:
    ConcreteVisitor2() { FUNCINFO; }
    ~ConcreteVisitor2() { FUNCINFO; }
    void VisitConcreteElementA(ConcreteElementA* element);
    void VisitConcreteElementB(ConcreteElementB* element);
};

class Element
{
public:
    Element() { FUNCINFO; }
    virtual ~Element() { FUNCINFO; }

    virtual void Accept(Visitor & visitor) = 0;
};

class ConcreteElementA : public Element
{
public:
    ConcreteElementA() { FUNCINFO; }
    ~ConcreteElementA() { FUNCINFO; }

    void Accept(Visitor & visitor){ FUNCINFO; visitor.VisitConcreteElementA(this); }
    void OperationA() { }
};

class ConcreteElementB : public Element
{
public:
    ConcreteElementB() { FUNCINFO; }
    ~ConcreteElementB() { FUNCINFO; }
    void Accept(Visitor & visitor){ FUNCINFO; visitor.VisitConcreteElementB(this); }
    void OperationB() { }

};

class CompositeElement
{
public:
    CompositeElement() { FUNCINFO; }
    ~CompositeElement()
    {
```

```
        for(list<Element*>::iterator itr = m_children.begin();
            itr != m_children.end(); ++itr)
        {
            delete *itr;
        }

        m_children.clear();
    }
    virtual void Accept(Visitor * visitor)
    {
        for(list<Element*>::iterator itr = m_children.begin();
            itr != m_children.end(); ++itr)
        {
            (*itr)->Accept(*visitor);
        }
    }

    void Attach(Element * element)
    {
        m_children.push_back(element);
    }

    void Detach(Element * element)
    {
        list<Element *>::iterator itr =
std::find(m_children.begin(),m_children.end(),element);
    }
private:
    list<Element *> m_children;
};

void ConcreteVisitor1::VisitConcreteElementA(ConcreteElementA* element){ FUNCINFO;}
void ConcreteVisitor1::VisitConcreteElementB(ConcreteElementB* element){ FUNCINFO;}

void ConcreteVisitor2::VisitConcreteElementA(ConcreteElementA* elment){ FUNCINFO;}
void ConcreteVisitor2::VisitConcreteElementB(ConcreteElementB* elment){ FUNCINFO;}

int main()
{
    CompositeElement *collection = new CompositeElement();
    Element * elementA = new ConcreteElementA();
    Element * elementB = new ConcreteElementB();
    collection->Attach(elementA);
    collection->Attach(elementB);
```

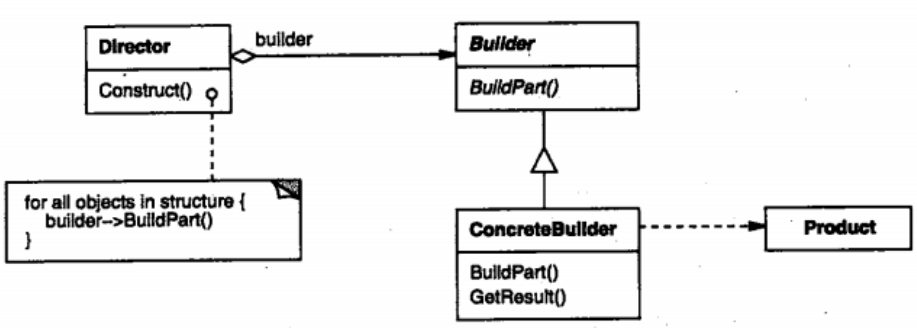
```
    Visitor * visitor1 = new ConcreteVisitor1();
    Visitor * visitor2 = new ConcreteVisitor2();
    collection->Accept(visitor1);
    collection->Accept(visitor2);

    delete visitor1;
    delete visitor2;
    delete collection;

    return 0;
}
```

8. Builder 生成器模式：

A) 用途：

将一个负责对象的构建与它的表示分离，使得同样的构建过程可以创建不同的表示。

B) UML 图：



C) 抽象基类：

1) Director ：这个基类是全部创建对象过程的抽象，提供创建不同组成部分的接口函数

2) Builder ：这个基类是创建不同组成部分的抽象，派生类通过继承它来创建不同的表示

D) 接口函数：

1) Builder::BuilderPartX() : 实现创建复杂对象的算法。

2) Director::Construct() : 调用抽象方式 BuilderPartX()，根据初始化的 Builder 成员，创建不同的对象，这里主要实现装配的方式。

E) 解析：

要使用这个模式，首先创建对象的算法和实现装配的过程于方式要相互独立。这样实现了装备算法和创建算法的解耦。

Builder 模式是基于这样的一个情况:一个对象可能有不同的组成部分,这几个部分的不同的创建对象会有不同的表示,但是各个部分之间装配的方式是一致的.比方说一辆单车,都是由车轮车座等等的构成的(一个对象不同的组成部分),不同的品牌生产出来的也不一样(不同的构建方式).虽然不同的品牌构建出来的单车不同,但是构建的过程还是一样的。

F) 代码实现：

```
/**
    Builder模式的实现
    File      :buidler.cpp
    Author        :singmelody
    Date      :2012.11.20
*/
#include <iostream>
using namespace std;

#define FUNCINFO cout<<__FUNCTION__<<endl;

class Builder;
class Director
{
public:
    Director(Builder *builder):m_builder(builder) { FUNCINFO;}
    ~Director() { FUNCINFO; delete m_builder; m_builder = NULL; }

    void Construct();
private:
    Builder* m_builder;
};

class Builder
{
public:
    Builder() { FUNCINFO; }
    virtual ~Builder() { FUNCINFO; }

    virtual void BuilderPartA() = 0;
    virtual void BuilderPartB() = 0;
};

class ConcreteBuilder1 : public Builder
{
public:
    ConcreteBuilder1() { FUNCINFO; }
    ~ConcreteBuilder1() { FUNCINFO; }
```

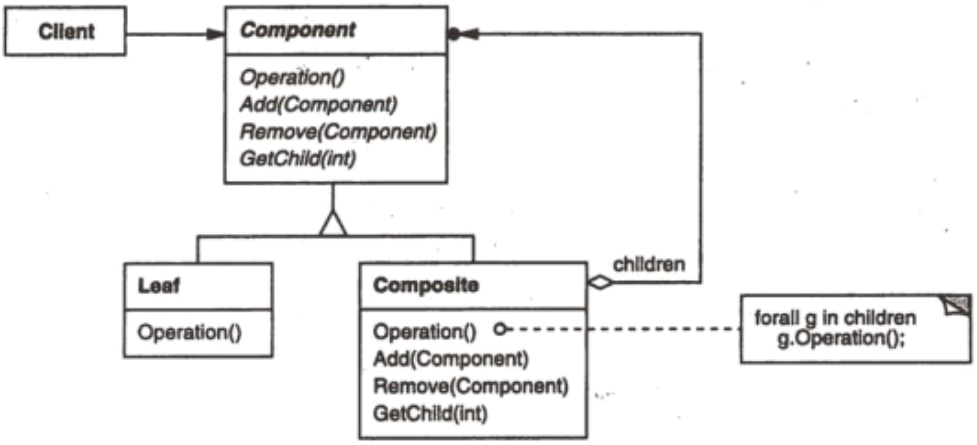
```
    void BuilderPartA() { FUNCINFO; }
    void BuilderPartB() { FUNCINFO; }
};

class ConcreteBuilder2 : public Builder
{
public:
    ConcreteBuilder2() { FUNCINFO; }
    ~ConcreteBuilder2() { FUNCINFO; }

    void BuilderPartA() { FUNCINFO; }
    void BuilderPartB() { FUNCINFO; }
};

void Director::Construct()
{
    FUNCINFO;
    m_builder->BuilderPartA();
    m_builder->BuilderPartB();
}

int main()
{
    Builder * builder1 = new ConcreteBuilder1();
    Director* director1 = new Director(builder1);
    director1->Construct();

    Builder * builder2 = new ConcreteBuilder2();
    Director* director2 = new Director(builder2);
    director2->Construct();

    delete director1;
    delete director2;
    return 0;
}
```

9. Composite 组成模式：

A) 用途：

将对象组合成树形结构以表示“部分-整体”的层次结构。Composite 使得用户对单个对象和组合对象的使用具有一致性。

B) UML 图：



C) 抽象基类：

1） Component：为组合中的对象声明接口，声明了类共有接口的缺省行为(如Add,Remove,GetChild 等)，声明接口可以多态的访问 Component 的子组件。

D) 接口函数：

1） Component::Operation(): 定义各个组件的接口行为，由各个组件具体实现。

2） Component::Add(),Component::Remove(),Component::GetChild(): 有缺省的实现方式，如果需要根据不同的行为，重写次方法。

E) 解析：

Component 模式是为解决组件之间的递归组合提供了解决的办法。抽象基类为树形体系中提供了抽象的方法，因为所有的子节点都是继承于抽象的基类 Component，当递归整个树形体系的时候，每个不同的结点使用其特有的方法。如在 UI 系统中，当你移动一个窗口，其中的所有子窗口(可以想象为子结点)，都会用他特有的方法来变化，举个例子当你在 Mac 系统下点击全屏按钮，子窗口菜单栏就会消失，而子窗口主界面会放大，其中的字体也会放大等等。这样就在一个继承体系中响应了不同的行为，这正是我们想要的。

F) 代码实现：

```
/**
    Composite模式的实现
    File      :composite.cpp
    Author    :singmelody
    Date      :2012.11.21
*/
#include <iostream>
#include <list>
#include <algorithm>
#include <assert.h>
using namespace std;
#define FUNCINFO cout<<__FUNCTION__<<endl;

class Component
```

```
{
public:
    Component()  {   FUNCINFO }
    virtual ~Component() {FUNCINFO }

    virtual void Operation() = 0;
    virtual void Add(Component *component){}
    virtual bool Remove(Component *component){ return false; }
    virtual Component * GetChild(int index) { return NULL; }
};

class Leaf : public Component
{
public:
    Leaf() { FUNCINFO; }
    ~Leaf() { FUNCINFO; }
    void Operation(){ FUNCINFO; }
};

class Composite : public Component
{
public:
    Composite() { FUNCINFO }
    ~Composite()
    {
        FUNCINFO;
        for(ComponentIterator itr = m_list.begin();
            itr != m_list.end(); ++itr)
        {
            delete *itr;
        }
        m_list.clear();
    }

    void Operation()
    {
        FUNCINFO;
        for(ComponentIterator itr = m_list.begin(); itr != m_list.end(); ++itr)
        {
            (*itr)->Operation();
        }
    }

    Component* GetChild(int index)
```

```
    {
        FUNCINFO;
        if(index < 0 && index >= m_list.size())
            return NULL;

        ComponentIterator itr;
        int count = 0;
        for(itr = m_list.begin();count < index;count++,itr++);

        return *itr;
    }

    void Add(Component *component) { FUNCINFO; m_list.push_back(component); }
    bool Remove(Component *component)
    {
        FUNCINFO;
        ComponentIterator itr = find(m_list.begin(),m_list.end(),component);
        if(itr != m_list.end())
        {
            delete *itr;
            m_list.erase(itr);
            return true;
        }
        else
        {
            return false;
        }
    }
private:
    list<Component *> m_list;
    typedef list<Component *>::iterator ComponentIterator;
};

int main()
{
    Component * composite = new Composite();
    Component * leaf1   = new Leaf();
    Component * leaf2   = new Leaf();

    composite->Add(leaf1);
    composite->Add(leaf2);
    composite->Operation();

    Component * leafX = composite->GetChild(1);
```

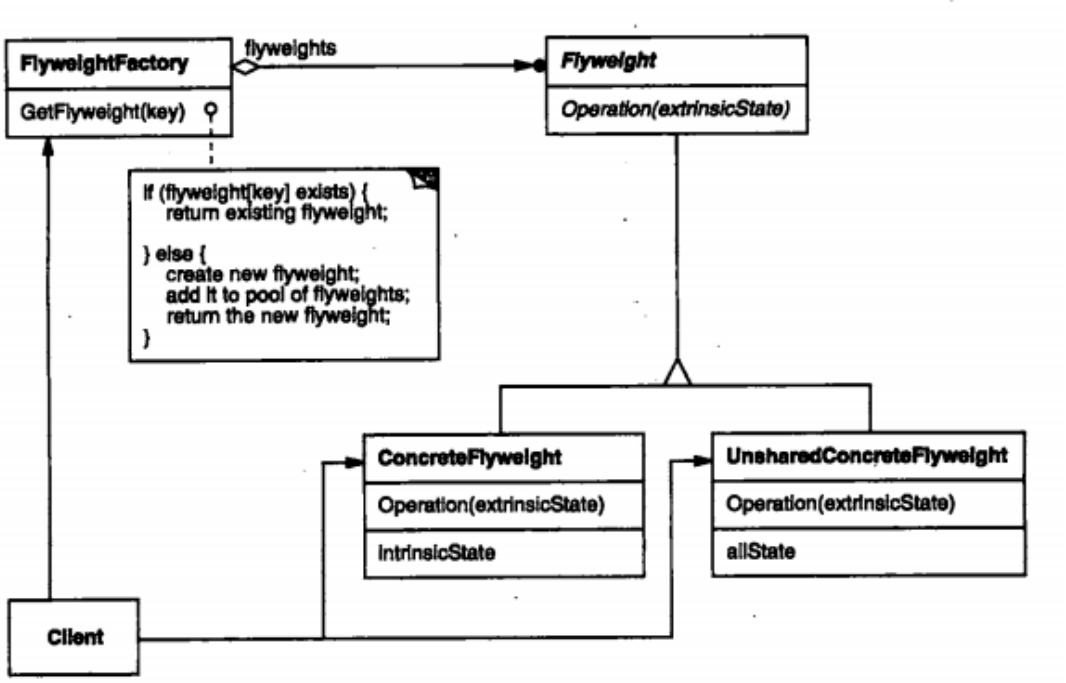
```
    assert(leafX == leaf2);
    composite->Remove(leafX);
    composite->Operation();

    delete composite;

    return 0;
}
```

10. FlyWeight 享元模式：

A) 用途：

运用共享技术有效地支持大量细粒度的对象。

B) UML 图：



C) 抽象基类：

1） Flyweight: 定义抽象方法 Operation()，方便派生类重写方法，实现自己特有的享元。

D) 接口函数：

1） FlyWeight::Operation() : 定义了享元的抽象方法，多态到派生类的具体方法。

E) 解析：

Flyweight 模式在大量使用一些可以被共享的对象的时候经常使用。比如,在 QQ 聊天的时候对于不同的人会有不同的回复,女生的话都喜欢别人说自己漂亮，所以打招呼的时候可以说声 “Hello 美女 XXX”；男生的话一般属于基友关系，所以大招呼的时候以”Hello 骚年 XXX”之类的。这些答复的前面其实都是提前定义好的，在使用的时候根据不同的人名调用出来即可。

Flyweight 就是基于解决这种问题的思路而产生的,当需要一个可以在其它地方共享使用的对象的时候,先去查询是否已经存在了同样样的对象,如果之前有生成就直接使用这个 Flyweight，调用这个 flyweight 的 Operation() 即可。没有的话，重新创建一个。因此,Flyweight 模式和 Factory 模式也经常混用。

F) 代码实现：

```
/**
    Flyweight模式的实现
    File      :flyweight.cpp
    Author    :singmelody
    Date      :2012.11.21
*/
#include <iostream>
#include <list>
#include <algorithm>
#include <string>
using namespace std;

#define FUNCINFO cout<<__FUNCTION__<<endl;
typedef std::string extrinsicState;

class Flyweight
{
public:
    Flyweight(extrinsicState& state):m_intrinsicState(state) { FUNCINFO; }
    virtual ~Flyweight() { FUNCINFO; }

    virtual void Operation(extrinsicState &) = 0;
    extrinsicState getInstrinsicState() const { return m_intrinsicState; }
protected:
    std::string m_intrinsicState;
};

class ConcreteFlywegiht : public Flyweight
{
public:
    ConcreteFlywegiht(extrinsicState & state):Flyweight(state) { FUNCINFO; }
    ~ConcreteFlywegiht() { FUNCINFO; }

    void Operation(extrinsicState & state )
    {
        FUNCINFO; cout<<state<<" "<<m_intrinsicState<<endl; }
};
```

```
class UnsharedConcreateFlyweight : public Flyweight
{
public:
    UnsharedConcreateFlyweight(extrinsicState & state):Flyweight(state) { FUNCINFO;}
    ~UnsharedConcreateFlyweight() { FUNCINFO; }

    void Operation(extrinsicState &state)
    {
        FUNCINFO;
        cout<<state<<endl;
    }
};

class FlyweightFactory
{
public:
    FlyweightFactory(){ FUNCINFO; }
    ~FlyweightFactory()
    {
        FUNCINFO;
        for(FlyweightIterator itr = m_list.begin();
            itr != m_list.end();++itr)
        {
            delete *itr;
        }
        m_list.clear();
    }

    Flyweight* GetFlyweight(extrinsicState & key)
    {
        FUNCINFO;
        for(FlyweightIterator itr = m_list.begin();
            itr != m_list.end(); ++itr)
        {
            if((*itr)->getInstrinsicState() == key)
            {
                cout<<"The Flyweight:"<<key<<" Already Exist"<<endl;
                return (*itr);
            }
        }

        cout<<"Create a new Flyweight: "<<key<<endl;
        Flyweight * flyweight = new ConcreteFlywegiht(key);
        m_list.push_back(flyweight);
```

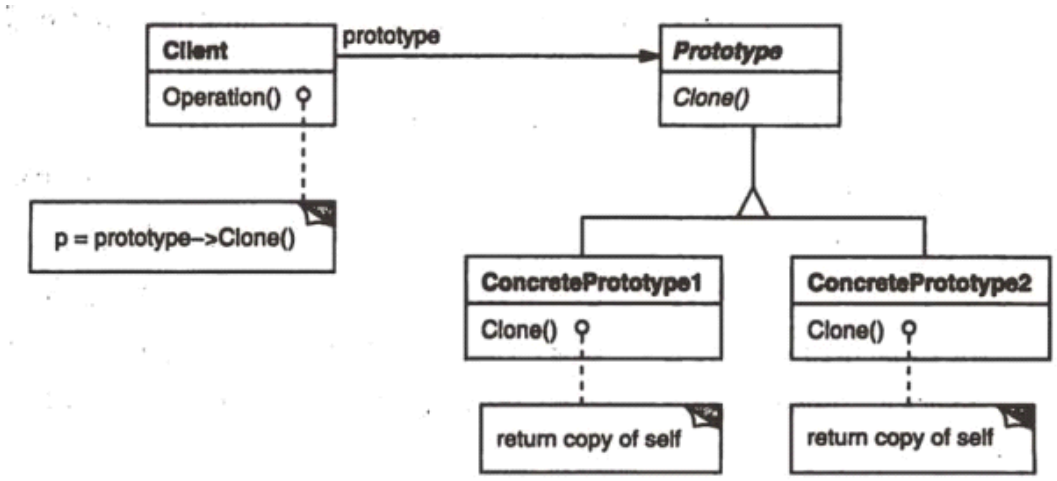
```
        return flyweight;
    }
private:
    list<Flyweight *> m_list;
    typedef list<Flyweight *>::iterator FlyweightIterator;
};

int main()
{
    FlyweightFactory * factory = new FlyweightFactory();
    factory->GetFlyweight(string("Pretty"))->Operation(string("Jack"));
    factory->GetFlyweight(string("Beautiful"))->Operation(string("Mary"));
    factory->GetFlyweight(string("Pretty"))->Operation(string("Maze"));

    delete factory;
    return 0;
}
```

11. Prototype 原型模式：

A) 用途：

用原型实例制定创建对象的种类，并且通过拷贝这些原型创建新的对象。

B) UML 图：



C) 抽象基类：

1） Prototype：虚拟基类，所有原型的基类，提供 Clone 接口函数。

D) 接口函数：

1）Prototype::Clone()：纯虚函数，根据不同的派生类来实例化创建对象。

E) 解析：

Prototype 模式其实就是常说的“虚拟构造函数”的一个实现。在 C++中，构造函数不能是虚函数，因为构造函数是从基类到派生类的，如果基类的构造函数还没有构建完成，就调用派生类的构造方法就会发生严重的错误。但是有些情况，可能需要根据抽象的接口 Clone()，来初始化不同的对象。比如，你配钥匙，配钥匙的师傅可以使用自己的技术配不同的钥匙，而去找师傅去配具体钥匙，就是具体的 ConcretePrototype。最后师傅配完会给你一把和之前一样的钥匙。

F)　代码实现：

```
/**
    Prototype模式的实现
    File      :prototype.cpp
    Author    :singmelody
    Date      :2012.11.21
*/
#include <iostream>
using namespace std;

#define FUNCINFO cout<<__FUNCTION__<<endl;

class Prototype
{
public:
    Prototype() { FUNCINFO; }
    virtual ~Prototype() { FUNCINFO; }

    virtual Prototype* Clone() = 0;
};


class ConcretePrototype1 : public Prototype
{
public:
    ConcretePrototype1() { FUNCINFO; }
    ~ConcretePrototype1() { FUNCINFO; }

    Prototype * Clone() { FUNCINFO; return new ConcretePrototype1(); }
};

class ConcretePrototype2 : public Prototype
{
public:
    ConcretePrototype2() { FUNCINFO; }
    ~ConcretePrototype2() { FUNCINFO; }
```

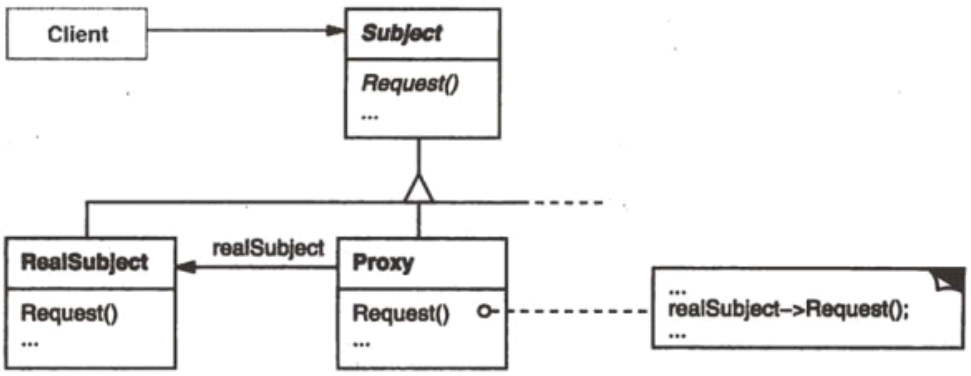
```
    Prototype*    Clone() { FUNCINFO; return new ConcretePrototype2(); }
};

int main()
{
    Prototype* prototype = new ConcretePrototype1();
    Prototype* first = prototype->Clone();

    Prototype* prototype1 = new ConcretePrototype2();
    Prototype* second = prototype1->Clone();

    delete prototype;
    delete prototype1;
    delete first;
    delete second;
    return 0;
}
```

12. Proxy 代理模式：

A) 用途：

为其他对象提供一种代理以控制对这个对象的访问。

B) UML 图：



C) 抽象基类：

1) Subject ：定义了 Proxy 和 RealSubject 的公共接口，这样就可以使用到 RealSubject 的地方都使用 Proxy。

D) 接口函数：

1） Subject::Request()：请求代理的类和代理本身的类都调用这个抽象接口。但是代理本身的类是使用了请求代理的类的请求，来使用代理本身类来代为发送请求。

E) 解析：

Proxy 一般适用于使用统一的接口，但是只有代理本身可以访问的方法。这样实现了请求代理者和代理者所要调用方法的解耦。比如，我们都知道的网页代理程序，假设我现在要翻墙上 twitter，请求代理者本身因为 GFW 的原因翻不上去墙，只能把自己的请求发送给代理者，代理者使用的是美国服务器，很少受 GFW 的控制，所以把你的请求发送给 twitter。

F) 代码实现：

```
/**
    Proxy模式的实现
    File      :proxy.cpp
    Author        :singmelody
    Date      :2012.11.22
*/
#include <iostream>
using namespace std;

#define FUNCINFO cout<<__FUNCTION__<<endl;

class Subject
{
public:
    Subject() { FUNCINFO; }
    virtual ~Subject() { FUNCINFO; }

    virtual void Request() = 0;
};

class RealSubject : public Subject
{
public:
    RealSubject() { FUNCINFO; }
    ~RealSubject() { FUNCINFO; }

    void Request() { FUNCINFO; }
};

class Proxy : public Subject
{
public:
    Proxy():m_realSubject(NULL) { FUNCINFO; }
    ~Proxy() { FUNCINFO; delete m_realSubject; }

    virtual void Request()
    {
```

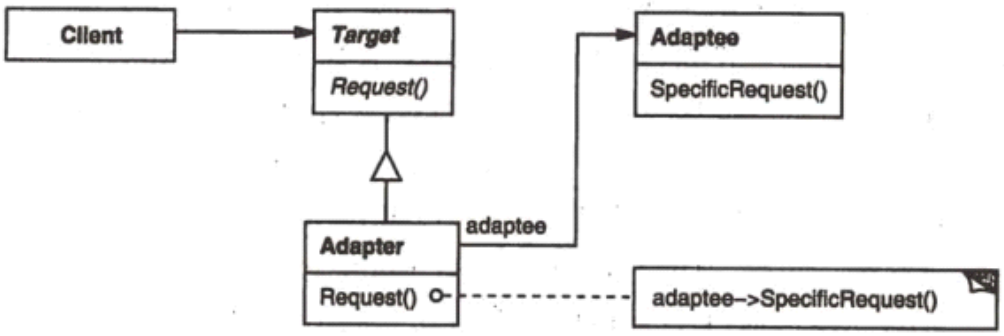
```
        FUNCINFO;
        if(m_realSubject == NULL)
        {
            m_realSubject = new RealSubject();
        }
        m_realSubject->Request();
    }
private:
    Subject* m_realSubject;
};

int main()
{
    Proxy * proxy = new Proxy();
    proxy->Request();

    delete proxy;
    return 0;
}
```

13. Adapter 适配器模式：

A) 用途：

将一个类的接口转换成客户希望的另外一个接口。Adapter 模式使得原本由于接口不兼容而不能一起工作的那些类可以一起工作。

B) UML 图：



C) 抽象基类：

1） Target ：提供需要接口(Request)的抽象积累。

D) 接口函数：

1） Target::Requst() ：提供兼容的接口。

E) 解析：

Adapter 模式有两种实现办法，一种是采用继承原有接口类的方法，一种是采用组合原有接口类的方法，这里使用的是第二种实现方法。

F) 代码实现：

```
/**
    Adapter模式的实现
    File     :Adapter.cpp
    Author       :singmelody
    Date     :2012.11.23
*/
#include <iostream>
using namespace std;

#define FUNCINFO cout<<__FUNCTION__<<endl;

class Adaptee
{
public:
    Adaptee() { FUNCINFO; }
    virtual ~Adaptee() { FUNCINFO; }

    void SpecificRequest(){ FUNCINFO; }
};

class Target
{
public:
    Target() { FUNCINFO; }
    virtual ~Target() { FUNCINFO;  }

    virtual void Request() = 0;
};


class Adapter : public Target
{
public:
    Adapter(Adaptee *adaptee) : m_adaptee(adaptee){ FUNCINFO; }
    ~Adapter(){ FUNCINFO; delete m_adaptee;}

    void Request() { FUNCINFO; m_adaptee->SpecificRequest(); }
protected:
    Adaptee*     m_adaptee;
};
```

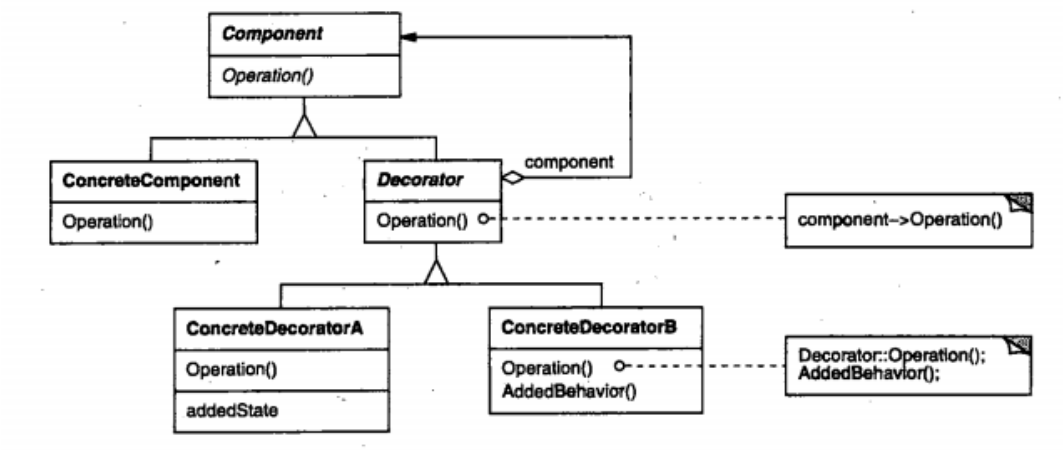
```
int main()
{
    Adaptee*adaptee  = new Adaptee();
    Target*      target   = new Adapter(adaptee);
    target->Request();

    delete target;

    return 0;
}
```

14. Decorator 装饰模式：

A) 用途：

动态地给一个对象添加一些额外的职责。就添加功能来说， Decorator 模式相比生成子类更为灵活。别名也为 Wrapper 模式，Lua 和 C++交互中使用这种模式比较多。

B) UML 图：



C) 抽象基类：

1） Component ：定义一个对象的接口，可以为这个接口动态的添加职责。

2） Decorator：维护一个指向 Component 的指针，并且有一个和 Component 一致的接口函数。

D) 接口函数：

1） Component::Operation()：这个接口函数为纯虚函数，所有的 Component 的派生类都需要实现。可以再这个函数的基础上给他动态的添加职责。比如 ConcreteDecoratorB 的 Operation()方法就添加 AddedBehavior() 操作。

E) 解析：

Decorator 基类初始化的时候放入真正的派生类组件(ConcreteComponent)，这样当 Decorator 的派生类如 ConcreteDecoratorB 继承 Decorator，调用 Decorator 的 Operation 的时候，实际上使用到了多态，调用了 ConcreteComponent 的 Operation()方法。而装饰 ConcreteComponent 的 Operation()方法的是 ConcreteDecoratorB 的 AddedBehavior()方法。这样就实现了 Operation()方法本身和其拓展的解耦。

Decorator 的使用的前提是组件的功能相对稳定，但是使用组件来进行的操作是不稳定的，经常变动的。我们可以通过继承 Decorator 来更好的适应这种变动。

举个实际的例子，你的女朋友要和你装修房子。其中的地板，电视都已经确定好买什么牌子和型号了。但是女孩子嘛比较注重细节，感觉给电视周围装饰一些小饰物。而装饰什么，怎么装饰，就是 Decorator 的派生类所要决断的。

F) 代码实现：

```
/**
    Decorator模式的实现
    File     :decorator.cpp
    Author   :singmelody
    Date     :2012.11.23
*/
#include <iostream>
using namespace std;

#define FUNCINFO cout<<__FUNCTION__<<endl;

class Component
{
public:
    Component(){ FUNCINFO; }
    virtual ~Component() { FUNCINFO; }

    virtual void Operation() = 0;
};

class ConcreteComponent : public Component
{
public:
    ConcreteComponent() { FUNCINFO; }
    ~ConcreteComponent() { FUNCINFO; }

    void Operation() { FUNCINFO; }
};

class Decorator : public Component
{
public:
```

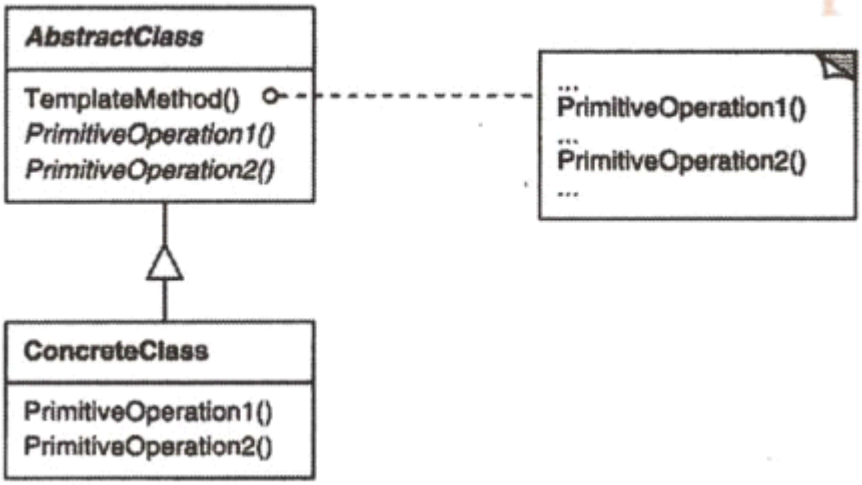
```
    Decorator(Component * component):m_component(component) { FUNCINFO; }
    ~Decorator() { FUNCINFO; delete m_component; }

    void Operation() { m_component->Operation(); }
protected:
    Component *  m_component;
};

class ConcreteDecoratorB : public Decorator
{
public:
    ConcreteDecoratorB(Component * component) : Decorator(component) { FUNCINFO; }
    ~ConcreteDecoratorB() { FUNCINFO; }

    void Operation() { FUNCINFO;  Decorator::Operation(); AddedBehavior(); }
    void AddedBehavior() { FUNCINFO; }
};

class ConcreteDecoratorA : public Decorator
{
public:
    ConcreteDecoratorA(Component * component) : Decorator(component) { FUNCINFO; }
    ~ConcreteDecoratorA() { FUNCINFO; }

    void Operation() {  FUNCINFO; Decorator::Operation(); AddedState(); }
    void AddedState() { FUNCINFO; }
};

int main()
{
    Component* pComponent = new ConcreteComponent();
    Decorator* pDecorator = new ConcreteDecoratorA(pComponent);

    pDecorator->Operation();

    delete pDecorator;
    return 0;
}
```

15. Template Method 模版方法模式：

A) 用途：

定义一个操作中的算法的骨架，而将一些步骤延迟到子类中。TemplateMethod 使得子类可以不改变一个算法的结构即可重定义该算法的某些特定的步骤。

B) UML 图：



C) 抽象基类：

1） AbstractClass ：抽象基类，定义算法的轮廓。

D) 接口函数：

1） AbstractClass::PrimitiveOperationX() ：基类为抽象方法，在 TemplateMethod()操作中使用此方法，派生类重写此接口，实现不改变一个算法的结构即可重定义该算法的某些特定步骤的目地。

2） AbstractClass::TemplateMethod() : 实现算法的结构，此结构是稳定的。算法具体的实现留给了派生类。

E) 解析：

TemplateMethod 模式的关键在于在基类中定义了一个算法的轮廓,但是算法每一步具体的实现留给了派生类。但是这样也会造成设计的灵活性不高的缺点,因为轮廓已经定下来了要想改变就比较难了,这也是为什么优先采用聚合而不是继承的原因。

F) 代码实现：

```
/**
    Template Method模式的实现
    File     :template method.cpp
    Author   :singmelody
    Date     :2012.11.24
*/
#include <iostream>
using namespace std;
```

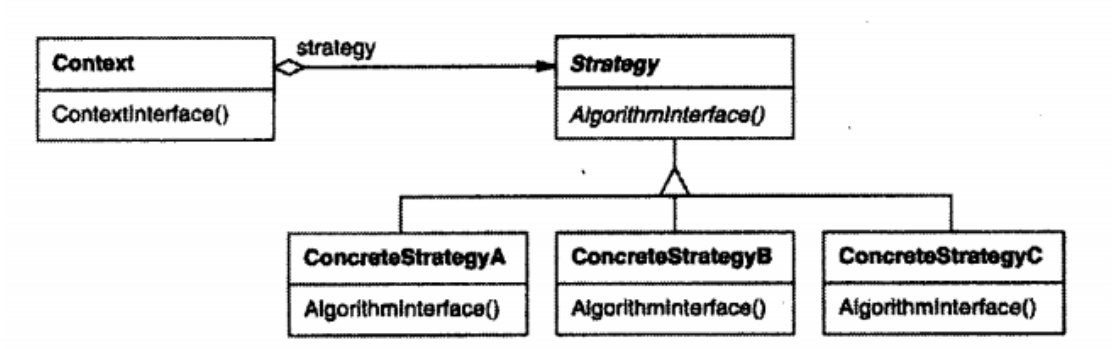
```
#define FUNCINFO cout<<__FUNCTION__<<endl;

class AbstractClass
{
public:
    AbstractClass() { FUNCINFO; }
    virtual ~AbstractClass() { FUNCINFO; }

    void TemplateMethod()
    {
        PrimitiveOperation1();
        PrimitiveOperation2();
    }
protected:
    virtual void PrimitiveOperation1() { FUNCINFO; }
    virtual void PrimitiveOperation2() { FUNCINFO; }
};

class ConcreteClass : public AbstractClass
{
public:
    ConcreteClass() { FUNCINFO; }
    ~ConcreteClass() { FUNCINFO; }

    void PrimitiveOperation1() { FUNCINFO; }
    void PrimitiveOperation2() { FUNCINFO; }
};

int main()
{
    AbstractClass * abstract = new ConcreteClass();
    abstract->TemplateMethod();

    delete abstract;
    return 0;
}
```

16. Strategy 策略模式：

A) 用途：

定义一系列的算法，把它们一个个封装起来，并且使他们可相互替换。本模式使得算法可独立于他们的客户而变化。

B) UML 图：



C) 抽象基类：

1) Strategy ：提供可供重写的算法函数的接口。

D) 接口函数：

1） Strategy::AlgorithmInterface() ：可供重写的算法接口。达到算法可独立于他们客户而变化的目的。

E) 解析：

简而言之一句话,Strategy 模式是对算法的封装.处理一个问题的时候可能有多种算法,这些算法的接口(输入参数,输出参数等)都是一致的,那么可以考虑采用 Strategy 模式对这些算法进行封装,在基类中定义一个函数接口就可以了。

有时候算法可能会有不同的策略，比如寻路算法，可能客户需要更快的寻路，可能需要更精准的寻路。对于这两个不同的算法，我们之前的话可能在基类中实现两个方法，然后供做产品的程序使用。但是这存在两个问题，每次都要把方法和方法所需的数据放在基类里面，增加的基类的耦合度；一般使用你代码的程序员，通常对方法不敏感，而对用什么比较敏感。

所以针对问题和两个寻路算法更好的需求，实现不同的派生类以实现不同的寻路算法是我们要考虑的。

F) 代码实现：

```
/**
    Strategy模式的实现
    File      :strategy.cpp
    Author    :singmelody
    Date      :2012.11.24
*/
#include <iostream>
using namespace std;

#define FUNCINFO cout<<__FUNCTION__<<endl;
#define SAFE_DELETE(p) delete p;p = NULL;

class Strategy
{
public:
```

```
    Strategy(){ FUNCINFO; }
    virtual ~Strategy() { FUNCINFO; }

    virtual void AlgorithmInterface() = 0;
};

class ConcreteStrategyA : public Strategy
{
public:
    ConcreteStrategyA(){ FUNCINFO; }
    ~ConcreteStrategyA() { FUNCINFO; }

    void AlgorithmInterface() { FUNCINFO;}

};

class ConcreteStrategyB : public Strategy
{
public:
    ConcreteStrategyB(){ FUNCINFO; }
    ~ConcreteStrategyB() { FUNCINFO; }

    void AlgorithmInterface() { FUNCINFO; }
};

class ConcreteStrategyC : public Strategy
{
public:
    ConcreteStrategyC(){ FUNCINFO; }
    ~ConcreteStrategyC() { FUNCINFO; }
    void AlgorithmInterface() { FUNCINFO; }
};

class Context
{
public:
    Context(Strategy *p):m_strategy(p){ FUNCINFO; }
    ~Context(){ FUNCINFO; SAFE_DELETE(m_strategy); }

    void ContextInterface(){ FUNCINFO; m_strategy->AlgorithmInterface(); }
private:
    Strategy* m_strategy;
};
int main()
```

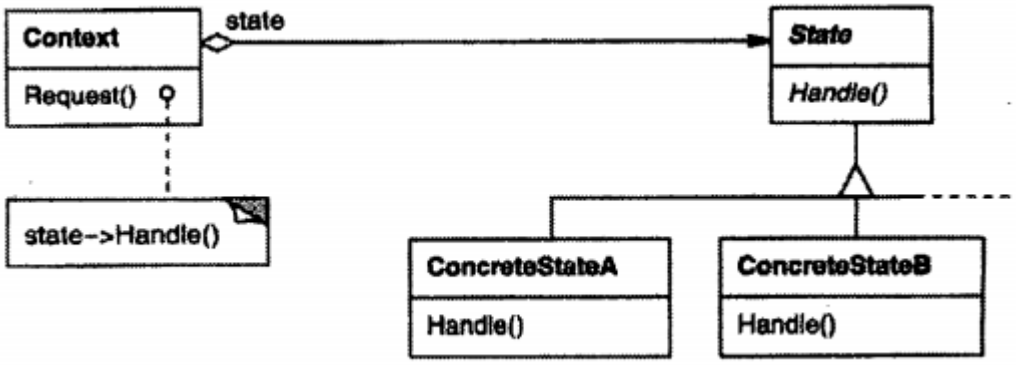
```
{
    Strategy*     strategy = new ConcreteStrategyA();
    Context* context    = new Context(strategy);
    context->ContextInterface();

    delete context;
    return 0;
}
```

17. State 状态模式：

A) 用途：

允许一个对象在其内部状态改变时改变它的应为。对象看起来修改了它的类。

B) UML 图：



C) 抽象基类：

1） State：抽象所有的状态，具体的不同状态派生它并实现之。

D) 接口函数：

1） State::Handle()：处理状态本身，完成之后状态切换的抽象接口。

2） Context::Request()：在这个函数中调用 state 对应 handle 进行状态处理的接口。

E) 解析：

State 模式主要解决的是在开发中时常遇到的根据不同的状态需要进行不同的处理操作的问题,而这样的问题,大部分人是采用 switch-case 语句进行处理的,这样会造成一个问题:分支过多,而且如果加入一个新的状态就需要对原来的代码进行编译.State 模式采用了对这些不同的状态进行封装的方式处理这类问题,当状态改变的时候进行处理然后再切换到另一种状态,也就是说把状态的切换责任交给了具体的状态类去负责。

当每次添加一个新的状态，然后从原有状态切换到这种状态，这种状态切换到原有的一种状态。只需要把状态列表的最后一个状态的 Handle()中，ChangeState 到最新状态，最新状态的 Handle() 再切换到原有的一种状态。

这样就实现了最小的修改来添加了最新的状态。比 if-else 和 switch-case 方式还要每次考虑到底在哪个 if 或者 case 中要方便很多。当状态上几十个的时候，这种优势就体现的非常明显了。Booost 库中的 statechart 库就采用了这种原则的设计。

F) 代码实现：

```
/**
    State模式的实现
    File      :state.cpp
    Author        :singmelody
    Date      :2012.11.25
*/
#include <iostream>
using namespace std;

#define FUNCINFO cout<<__FUNCTION__<<endl;

class Context;

class State
{
public:
    State() { FUNCINFO; }
    virtual ~State() { FUNCINFO; }
    virtual void Handle(Context *context) = 0;
};

class ConcreteStateA : public State
{
public:
    ConcreteStateA() { FUNCINFO; }
    ~ConcreteStateA() { FUNCINFO; }
    void Handle(Context *context);
};

class ConcreteStateB : public State
{
public:
    ConcreteStateB() { FUNCINFO; }
    ~ConcreteStateB() { FUNCINFO; }
    void Handle(Context *context);
};

class Context
{
```

```
public:
    Context(State* state):m_state(state){ FUNCINFO;}
    ~Context() { FUNCINFO; delete m_state; }
    void ChangleState(State *state)
    {
        FUNCINFO;
        if(NULL != m_state)
        {
            delete m_state;
            m_state = NULL;
        }

        m_state = state;
    }

    void Request()
    {
        FUNCINFO;
        if (NULL != m_state)
        {
            m_state->Handle(this);
        }
    }
private:
    State*   m_state;
};

void ConcreteStateA::Handle(Context *context)
{
    FUNCINFO;
    context->ChangleState(new ConcreteStateB() );
}

void ConcreteStateB::Handle(Context *context)
{
    FUNCINFO;
    context->ChangleState(new ConcreteStateA() );
}

int main()
{
    State *pState = new ConcreteStateA();
    Context *pContext = new Context(pState);
    pContext->Request();
```

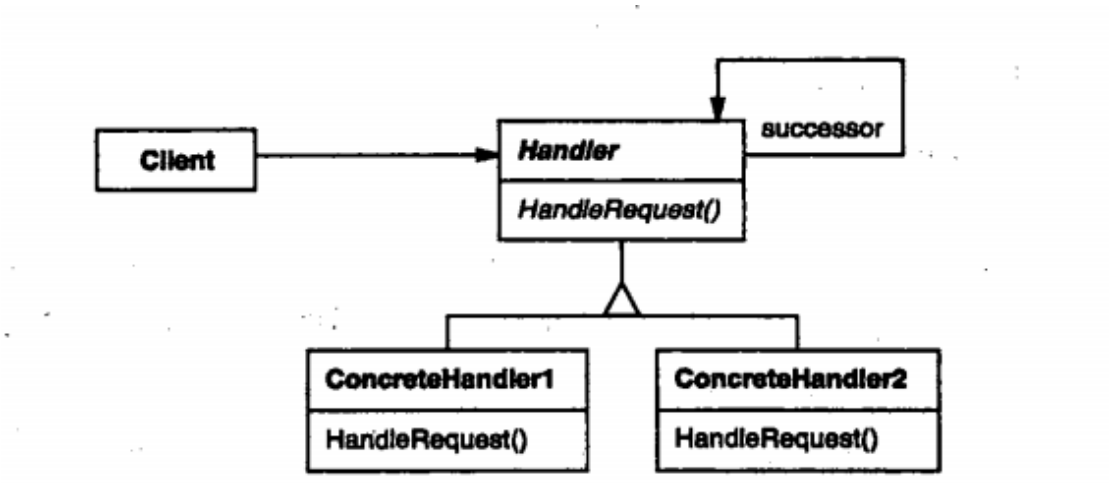
```
    pContext->Request();
    pContext->Request();

    delete pContext;

    return 0;
}
```

18. CHAIN OF RESPONSIBLITY 责任链模式：

A) 用途：

使多个对象都有机会处理请求，从而避免请求的发送者和接受者之间的耦合关系。讲这些对象连成一条链，并沿着这条链传递该请求，直到有一个对象处理它为止。

B) UML 图：



C) 抽象基类：

1) Handler：抽象基类，定义责任链的下一结点与相应的处理方法 HandleRequest()。

D) 接口函数：

1） Handler::HandleRequest() ： 定义责任链每个结点对应的处理方法和访问责任链的下一结点。

E) 解析：

这个模式把可以处理一个请求的对象以链的形式连在了一起,让这些对象都有处理请求的机会.好比原来看古装电视中经常看到皇宫中召见某人的时候,太监们(可以处理一个请求的对象)就会依次的喊:传 XX...这样一直下去直到找到这个人为止.ChainOfResponsibility 模式也是这样的处理请求的,如果有后续的对象可以处理,那么传给后续的对象处理,否则就自己处理请求.这样的设计把请求的发送者和请求这种的处理者解耦了,好比发号的皇帝不知道到底是哪个太监最后会找到他要找到的人一般,只管发出命令就 OK 了。

如果对 HandleRequest 稍作修改，就可以实现另一种功能，就是类似递归的责任链模式。还是刚才那个例子，假设现在皇上要传御膳房给自己做饭。这件事传给大太监，这个太监跑百米之后，让自己的小弟把这件事继续传下去，最后传到第 4 个最靠御膳房的太监，御膳房把 KFC 做好，第四个太监传回去，传到大太监手上，最后送给皇上。这时候 HandleRequest 需要先访问下一结点，等结点递归回来再做自己的响应处理。

责任链模式是在对象初始化的时候确定任务处理顺序的。而 State 模式是在每个状态对应处理的时候来决定顺序的，State 模式中的 Context 只起保存上下文的作用。

F) 代码实现：

```
/**
    Chain Of Responsibility模式的实现
    File      :chain of responsibility.cpp
     Author        :singmelody
     Date      :2012.11.25
*/
#include <iostream>
using namespace std;

#define FUNCINFO cout<<__FUNCTION__<<endl;

class Handler
{
public:
    Handler(Handler * handler):m_successor(handler) {  FUNCINFO; }
    virtual ~Handler()
    {
        FUNCINFO;
        if(m_successor != NULL)
        {
            delete m_successor;
            m_successor = NULL;
        }
    }
    virtual void HandleRequest() = 0;
protected:
    Handler* m_successor;
};

class ConcreteHandler1 : public Handler
{
public:
    ConcreteHandler1(Handler * handler = NULL):Handler(handler){ FUNCINFO; }
    ~ConcreteHandler1() { FUNCINFO; }
```

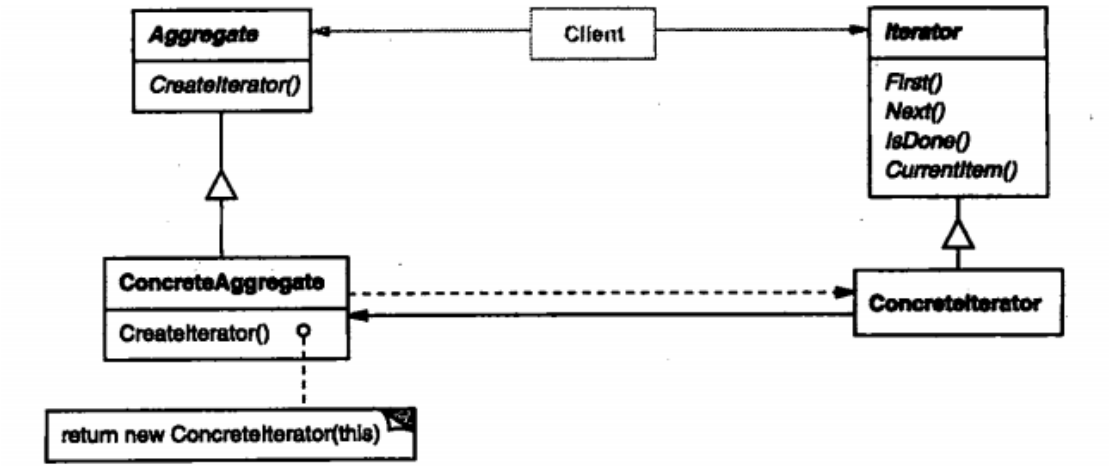
```
    void HandleRequest()
    {
        FUNCINFO;
        if (NULL != m_successor)
        {
            m_successor->HandleRequest();
        }
    }
};

class ConcreteHandler2 : public Handler
{
public:
    ConcreteHandler2(Handler * handler = NULL ):Handler(handler){ FUNCINFO; }
    ~ConcreteHandler2() { FUNCINFO; }

    void HandleRequest()
    {
        FUNCINFO;
        if (NULL != m_successor)
        {
            m_successor->HandleRequest();
        }
    }
};

int main()
{
    Handler *p2 = new ConcreteHandler2();
    Handler *p1 = new ConcreteHandler1(p2);

    p1->HandleRequest();

    delete p1;
    return 0;
}
```

19. Iterator 迭代器模式：

A) 用途：

提供一种方法顺序访问一个聚合对象中各个元素，而不需要暴露该对象的内部表示。

B) UML 图：



C) 抽象基类：

1） Iterator：Iterator 可以看做是一种的规范,它提供了所有 Iterator 需要遵守的规范也就是对外的接口,而它的派生类 ConcreateIterator 则是 ConcreateAggregate 容器的迭代器,它遵照这个规范对容器进行迭代和访问操作。

2） Aggregate：提供创建和自己类型相同迭代器的一个接口。

D) 接口函数：

1） Iterator::First(),Iterator::IsDone(),Iterator::Next(),Iterator::CurrentItem()：提供访问 Aggregate 内部数据的三个接口。

2） Aggregate::CreateIterator()：创建自己类型相同的一个迭代器。其功能和 STL 中 container<Type>::iterator 相似。

E) 解析：

这个模式是 STL 中很常用的模式，但是 STL 把 First()和 End()接口放到 container 中来实现；Iterator::Next()在 STL 一般用重载前置运算符++来替代；Iterator::CurrentItem()在 STL 中用重载*号运算符来计算。

STL 中 Iterator 和 Container 是符合统一的标准的，这样对于外部用户来说，不用关心内部的实现和别的容器的差别，统一的使用即可，比如想获得容器迭代器直接使用 container<Type>::iterator 即可。

下面是 STL 源码中的一些规范：

```
typedef typename _Mybase::value_type value_type;
typedef typename _Mybase::size_type size_type;
typedef typename _Mybase::difference_type difference_type;
typedef typename _Mybase::pointer pointer;
typedef typename _Mybase::const_pointer const_pointer;
typedef typename _Mybase::reference reference;
typedef typename _Mybase::const_reference const_reference;

typedef typename _Mybase::const_iterator const_iterator;
typedef typename _Mybase::iterator iterator;
```

如果你经常留意 STL 不同容器的代码，就会发现所有的容器都会有这样一套统一标准的声明。

F) 代码实现：

```
/**
    Iterator模式的实现
    File      :iterator.cpp
    Author        :singmelody
    Date      :2012.11.26
*/
#include <iostream>
#include <vector>
using namespace std;

#define FUNCINFO cout<<__FUNCTION__<<endl;
#define SAFE_DELETE(p) if(p) {delete p; p = NULL;}
typedef int  DataType;

class ConcreteIterator;
class Iterator;

class Aggregate
{
public:
    Aggregate(int size) { FUNCINFO; m_vec.reserve(size); }
    virtual ~Aggregate() { FUNCINFO; m_vec.clear(); }

    virtual Iterator& CreateIterator() = 0;
    int                   GetSize() { return m_vec.size(); }
    void              Push(int number) { m_vec.push_back(number); }
    DataType&             operator[](int offset){ return m_vec[offset]; }
protected:
    vector<DataType>  m_vec;
};

class ConcreteAggregate : public Aggregate
{
public:
    ConcreteAggregate(int size) :Aggregate(size){ FUNCINFO; }
    ~ConcreteAggregate() { FUNCINFO; }

    Iterator& CreateIterator();
};
```

```
class Iterator
{
public:
    Iterator(Aggregate*   aggregate):m_aggregate(aggregate) { FUNCINFO; }
    virtual ~Iterator() { FUNCINFO; }

    virtual bool      IsDone()          = 0;
    virtual void      Next()                = 0;
    virtual void      First()               = 0;
    virtual DataType& CurrentItem() const   = 0;
protected:
    Aggregate*    m_aggregate;
    int           m_curIndex;
};

class ConcreteIterator : public Iterator
{
public:
    ConcreteIterator(Aggregate* aggregate) : Iterator(aggregate){ FUNCINFO;}
    ~ConcreteIterator() { FUNCINFO; }

    bool IsDone()
    {
        return m_curIndex <   m_aggregate->GetSize();
    }

    void Next()
    {
        m_curIndex++;
    }

    void First()
    {
        m_curIndex = 0;
    }

    DataType& CurrentItem() const
    {
         return (*m_aggregate)[m_curIndex];
    }
};

Iterator& ConcreteAggregate::CreateIterator()
{
```

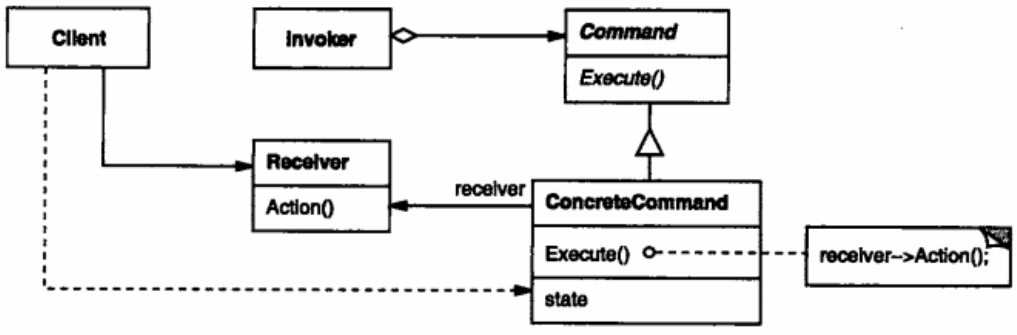
```
    Iterator* p = new ConcreteIterator(this);
    return *p;
}

int main()
{
    Aggregate* aggregate = new ConcreteAggregate(4);
    aggregate->Push(10);
    aggregate->Push(20);
    aggregate->Push(30);

    Iterator& iterator  = aggregate->CreateIterator();

    for(iterator.First();
        false != iterator.IsDone(); iterator.Next() )
    {
        cout<<iterator.CurrentItem()<<endl;
    }

    SAFE_DELETE(aggregate);

    Iterator* p = &iterator;
    SAFE_DELETE(p);

    return 0;
}
```

20 Command 命令模式：

A) 用途：

将一个请求封装为一个对象，从而使你可用不同的请求对客户进行参数化；对请求排队或记录请求日志，以及支持可撤销的操作。

B) UML 图：



C) 抽象基类：

1) Command：定义命令的接收者，当命令通过 Invoker 执行时，命令把其自带的信息传送给接收者。

D) 接口函数：

1） Command::Execute()：把命令传送给接收者执行。

E) 解析：

一般情况下，系统都会采用消息的设计，而如何实现消息发送者，接收者和具体命令的解耦是这种系统设计时考虑的主要问题。Command 模式就提供了很好的参照。把消息的封装起来，需要的时候传送给接收者。

其实,如果弄清楚了 Command 模式的原理,就会发现其实它和注册回调函数的原理是很相似的,而在面向过程的设计中的回调函数其实和这里的 Command 类的作用是一致的.采用 Command 模式解耦了命令的发出者和命令的执行者。每次发送消息，只需要在意接收者和具体的指令，实现了最大的解耦。

下面的代码如果你有兴趣，可以修改为 1 对多，多对 1，多对多的传送消息。

F) 代码实现：

```
/**
    Command模式的实现
    File      :command.cpp
    Author    :singmelody
    Date      :2012.11.26
*/
#include <iostream>
using namespace std;

#define FUNCINFO cout<<__FUNCTION__<<endl;
#define SAFE_DELETE(p) if(NULL != p) { delete p,p=NULL; }
enum STATE
{
    STATE_WAIT,
    STATE_SLEEP,
    STATE_RUN
};

char* str[] = {"WAIT","SLEEP","RUN"};

class Receiver
{
public:
    Receiver(){ FUNCINFO; m_state = STATE_WAIT;  }
    ~Receiver(){ FUNCINFO; }

    void Action(STATE state){ FUNCINFO; m_state = state; cout<<str[m_state]<<endl;}
```

```
private:
    STATE m_state;
};

class Command
{
public:
    Command(Receiver * receiver = NULL,STATE state = STATE_WAIT) :
m_receiver(receiver),m_state(state){ FUNCINFO; }
    virtual ~Command(){ FUNCINFO; }

    virtual void Execute() = 0;
protected:
    Receiver*    m_receiver;
    STATE        m_state;
};

class ConcreteCommand : public Command
{
public:
    ConcreteCommand(Receiver* receiver,STATE state) : Command(receiver,state)
{ FUNCINFO; }
    ~ConcreteCommand() { FUNCINFO; }
    void Execute()
    {
        FUNCINFO;
        if(NULL != m_receiver)
        {
            m_receiver->Action(m_state);
        }
    }
};

class Invoker
{
public:
    Invoker(Command* command = NULL):m_command(command) { FUNCINFO; }
    ~Invoker() { FUNCINFO; }

    void Invoke()
    {
        FUNCINFO;
        if (NULL != m_command)
        {
```

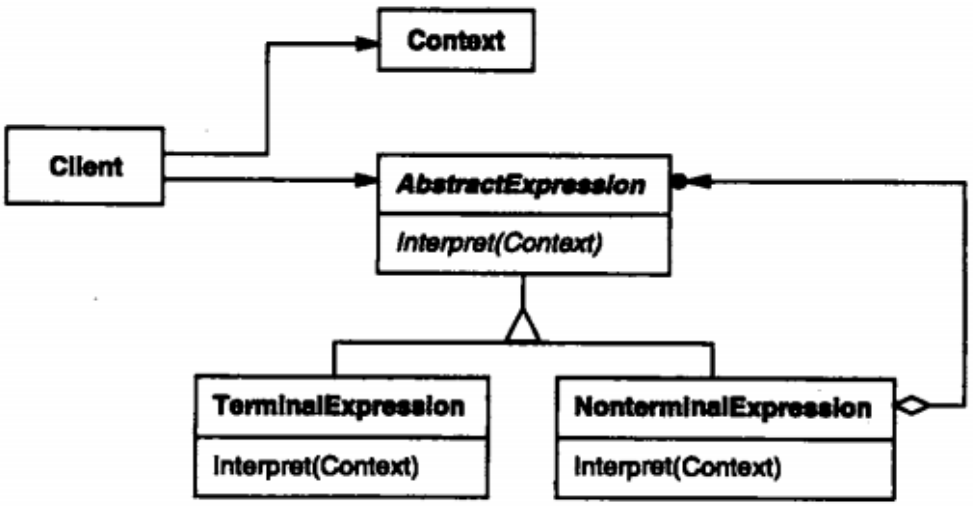
```
            m_command->Execute();
        }
    }
private:
    Command* m_command;
};

int main()
{
    Receiver* receiver = new Receiver();
    Command* command = new ConcreteCommand(receiver,STATE_RUN);
    Invoker* invoker = new Invoker(command);

    invoker->Invoke();

    SAFE_DELETE(receiver);
    SAFE_DELETE(command);
    SAFE_DELETE(invoker);

    return 0;
}
```

21 Interpreter 翻译器模式：

A) 用途：

给定一个语言，定义它的文法的一种表示，并定义一个解释器，这个解释器使用该表示来解释语言中的句子。

B) UML 图：



C) 抽象基类：

1） AbstractExpression：定义抽象表达式和抽象接口，派生类代表文法中不同类型的文法符号。

D) 接口函数：

1） AbstractExpression::Interpret()：定义抽象翻译接口，派生类是实现不同的翻译方法。

E) 解析：

Interpreter 模式最重要的作用，是用来扩展文法。当你有一个比较稳定的结构生成算法时，Interpreter 通过派生 AbstractExpression 来实现不同的扩展文法。具体文法符号的算法是需要重写 AbstractExpression::Interpret() 即可。这种使用指向实际对象的基类指针多态到具体派生类的 Interpret()函数，实现不同的文法。

Interpreter 模式和 Strategy 模式有相似之处，都是通过派生基类来实现不同的算法。但是 Interpreter 模式是拓展算法，而 Strategy 是替换算法。

下面的代码很好的使用了 Interpreter 模式，通过逆波兰表达式来实现一个比较稳定的结构生成算法。而以后如果需要拓展自己需要的符号，只需要继承基类方法并实现具体文法即可。实现了多个文法之间的解耦。

F) 代码实现：

```
/**
    Interpreter模式的实现
    File      :Interpreter.cpp
    Author    :singmelody
    Date      :2012.11.26
*/

#include <map>
#include <stack>
#include <string>
#include <iostream>
using namespace std;

#define FUNCINFO cout<<__FUNCTION__<<endl;
#define SAFE_DELETE(p) if(p!=NULL){ delete p,p=NULL; }

class Expression;
typedef map<string,Expression *> StrExp;

class Expression
{
public:
    Expression() { FUNCINFO; }
    virtual ~Expression() { FUNCINFO; }
    virtual int Interpret(StrExp & strexp) = 0;
```

```
};

class Number : public Expression
{
public:
    Number(int number) : m_number(number){ FUNCINFO; }
    ~Number() { FUNCINFO; }
    int Interpret(StrExp &strexp)
    {
        FUNCINFO;
        return m_number;
    }
private:
    int m_number;
};

class Plus : public Expression
{
private:
    Expression *m_leftExp;
    Expression *m_rightExp;
public:
    Plus(Expression                                                    *left,Expression
*right):m_leftExp(left),m_rightExp(right){ FUNCINFO;}
    ~Plus() { FUNCINFO;  }

    int Interpret(StrExp & strexp)
    {
        FUNCINFO;
        return m_leftExp->Interpret(strexp) + m_rightExp->Interpret(strexp);
    }
};

class Minus : public Expression
{
private:
    Expression *m_leftExp;
    Expression *m_rightExp;
public:
    Minus(Expression                                                    *left,Expression
*right):m_leftExp(left),m_rightExp(right){ FUNCINFO;}
    ~Minus(){ FUNCINFO; }

    int Interpret(StrExp & strexp)
```

```
    {
        FUNCINFO;
        return m_leftExp->Interpret(strexp) - m_rightExp->Interpret(strexp);
    }
};

class Variable : public Expression
{
private:
    string   m_name;
public:
    Variable(string name):m_name(name) { FUNCINFO; }
    ~Variable(){ FUNCINFO; }

    int Interpret(StrExp &strexp)
    {
        FUNCINFO;
        if(strexp.find(m_name) != strexp.end())
            return strexp[m_name]->Interpret(strexp);

        return 0;
    }

};

class Evaluator : public Expression
{
private:
    Expression* m_syntaxTree;
public:
    Evaluator(string & exp)
    {
        FUNCINFO;
        std::stack<Expression *> expressionStack;

        for(int i = 0; i < exp.size();++i)
        {
            char token = exp[i];
            if(token == '+')
            {
                Expression *right = expressionStack.top();
                expressionStack.pop();
                Expression *left  = expressionStack.top();
                expressionStack.pop();
```

```
                Expression *subExpression = new Plus(left,right);
                expressionStack.push(subExpression);
            }
            else if(token == '-')
            {

                Expression *right = expressionStack.top();
                expressionStack.pop();
                Expression *left  = expressionStack.top();
                expressionStack.pop();
                Expression *subExpression = new Minus(left,right);
                expressionStack.push(subExpression);
            }
            else
            {
                char tmp[2] = {token,'\0'};
                string str(tmp);
                expressionStack.push(new Variable(str));
            }
        }
        m_syntaxTree = expressionStack.top();
        expressionStack.pop();
    }

    ~Evaluator() { FUNCINFO; }

    int Interpret(StrExp & exp) {
        FUNCINFO;
        return m_syntaxTree->Interpret(exp);
    }
};

int main()
{
    string expression = "wxz-+";
    Expression* sentence = new Evaluator(expression);
    StrExp variables;
    variables["x"] = new Number(5);
    variables["w"] = new Number(10);
    variables["z"] = new Number(42);

    cout<<sentence->Interpret(variables)<<endl;

    for(StrExp::iterator itr = variables.begin();
```

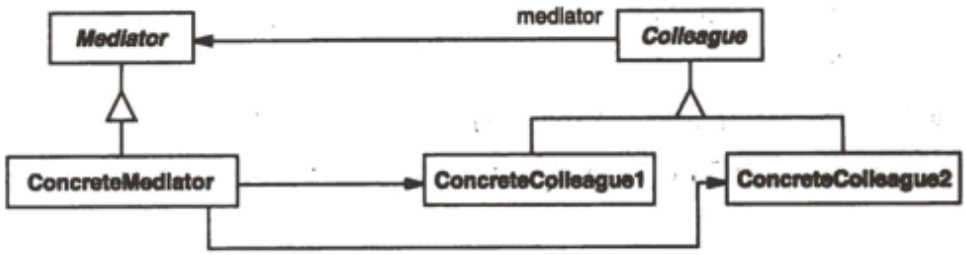
```
        itr != variables.end();++itr)
    {
        SAFE_DELETE(itr->second);
    }
    variables.clear();
    SAFE_DELETE(sentence);
    return 0;
}
```

22 MEDIATOR 中介者模式：

A) 用途：

用一个中介对象来封装一系列的对象交互。中介者使各对象不需要显式地相互引用，从而使其耦合松散，而且可以独立地改变他们之间的交互。

B) UML 图：



C) 抽象基类：

1） Colleague：定义 ConreteColleague1 和 ConreteColleague2 抽象操作的抽象基类。

2） Mediator ：定义中介者抽象操作的抽象基类。

D) 接口函数：

视具体情况。

E) 解析：

和 Composite 模式很类似，但是没有 Composite 派生类对象之间的联系紧密，主要它是通过中介者来实现解耦的。与 Composite 比较相似的地方就是可以有不同的组合，实现不同的中介和效果。

F) 代码实现：

```
/**
    Mediator模式的实现
    File      :mediator.cpp
    Author        :singmelody
    Date      :2012.11.28
*/
#include <iostream>
```

```
using namespace std;

#define FUNCINFO cout<<__FUNCTION__<<endl;

class Colleage
{
public:
    Colleage() { FUNCINFO; }
    virtual ~Colleage() { FUNCINFO; }

    virtual void Command() = 0;
};

class ConcreteColleage1 : public Colleage
{
public:
    ConcreteColleage1(){ FUNCINFO; }
    ~ConcreteColleage1(){ FUNCINFO; }

    void Command() { FUNCINFO; }
};

class ConcreteColleage2 : public Colleage
{
public:
    ConcreteColleage2(){ FUNCINFO; }
    ~ConcreteColleage2(){ FUNCINFO; }

    void Command() { FUNCINFO; }
};

class Mediator
{
public:
    Mediator(){ FUNCINFO; }
    virtual ~Mediator() { FUNCINFO; }

    virtual void Execute() = 0;
};

class ConcreteMediator : public Mediator
{
public:
    ConcreteMediator(Colleage* p,Colleage* q):m_colleage1(p),m_colleage2(q){ FUNCINFO; }
```

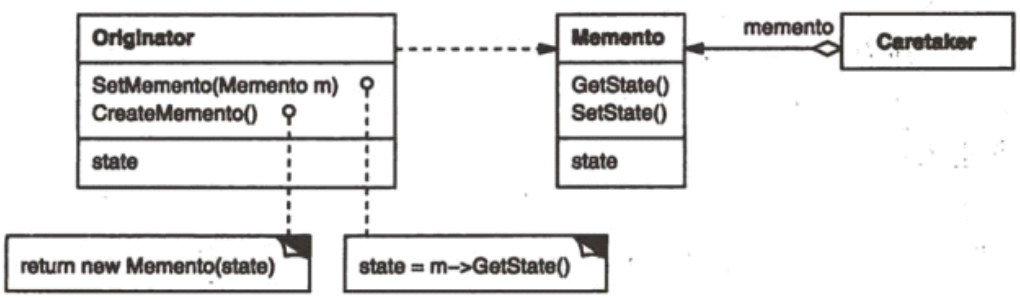
```
    ~ConcreteMediator(){ FUNCINFO; }

    void Execute()
    {
        FUNCINFO;
        m_colleage1->Command();
        m_colleage2->Command();
    }
private:
    Colleage * m_colleage1;
    Colleage * m_colleage2;
};

int main()
{
    Colleage * colleage1 = new ConcreteColleage1();
    Colleage * colleage2 = new ConcreteColleage1();

    Mediator * mediator = new ConcreteMediator(colleage1,colleage2);

    mediator->Execute();

    delete colleage1;
    delete colleage2;
    delete mediator;
    return 0;
}
```

23 MEMENTO 备忘录模式：

A) 用途：

在不破坏封装性的前提下，捕获一个对象的内部状态，并在该对象之外保存这个状态。这样以后就可将该对象回复到原先保存的状态。

B) UML 图：



C) 抽象基类：

1) 无

D) 接口函数：

1） Originator::CreateMemento() ：创建备忘并保存下来，方便之后使用备忘。

2) 对应类的 getter/setter：获取和改变类内部的状态。

E) 解析：

Memento 模式中封装的是需要保存的状态，当需要恢复的时候才取出来进行恢复.原理很简单,实现的时候需要注意一个地方：窄接口和宽接口。所谓的宽接口就是一般意义上的接口,把对外的接口作为 public 成员;而窄接口反之，把接口作为 private 成员,而把需要访问这些接口函数的类作为这个类的友元类,也就是说接口只暴露给了对这些接口感兴趣的类,而不是暴露在外部.下面的实现就是窄实现的方法来实现的。

Memento 模式可运用的场景很多，比如在游戏的不同场景需要切换不同的场景管理器,(如室内 BSP，室外 Octree)，当你需要从室外切向室内管理器，可以先保存室外管理器，等到从室内出去到室外，再切换室外管理器。这样首先可以起到记忆的作用，而且不需要频繁的创建和释放管理器，创建管理器是很耗费时间的。

F) 代码实现：

```
/**
    memento模式的实现
    File      :memento.cpp
    Author    :singmelody
    Date      :2012.11.28
*/
#include <iostream>
#include <assert.h>
using namespace std;

#define FUNCINFO cout<<__FUNCTION__<<endl;
#define SAFE_DELETE(p) if(NULL != p){ delete p,p=NULL;}

enum STATE
{
    STATE_WAIT,
    STATE_WALK,
    STATE_RUN,

    STATE_SUM
};

char * StateStr[] = {"WAIT","WALK","RUN"};
```

```
class Memento
{
public:
    ~Memento() { FUNCINFO; }
private:
    friend class Originator;
    Memento(STATE state = STATE_WAIT):m_state(state){ FUNCINFO; }

    void SetState(STATE state) {
        FUNCINFO;
        if(state < STATE_SUM)
            m_state = state;
        else
            assert("State is Not Right");

    }
    STATE GetState()  { FUNCINFO; return m_state; }

    STATE m_state;
};

class Originator
{
public:
    Originator(STATE state = STATE_WAIT):m_state(state) { FUNCINFO; }
    ~Originator() { FUNCINFO; }

    void SetMemento(Memento *m)
    {
        FUNCINFO;
        if(NULL != m)
        {
            m_state = m->GetState();
        }
    }

    Memento* CreateMemento(STATE state)
    {
        FUNCINFO;
        return new Memento(state);
    }

    Memento* CreateMemento()
    {
```

```
        FUNCINFO;
        return new Memento(m_state);
    }

    void PrintState()
    {
        cout<<StateStr[m_state]<<endl;
    }

private:
    STATE m_state;
};

int main()
{
    Originator * originator = new Originator(STATE_WAIT);
    originator->PrintState();

    // store the old state
    Memento *oldMemento = originator->CreateMemento();
    Memento *newMemento = originator->CreateMemento(STATE_RUN);

    originator->SetMemento(newMemento);
    originator->PrintState();

    // restore the old state
    originator->SetMemento(oldMemento);
    originator->PrintState();

    SAFE_DELETE(oldMemento);
    SAFE_DELETE(newMemento);
    SAFE_DELETE(originator);

    return 0;
}
```

PS:

OK，23种设计模式就到此为止了。

当看完这个，很多人会很困惑，设计模式到底有什么用?感觉好抽象，还不如写一个XX算法来的实在。

首先说一点，就算开始不会使用设计模式，但是等到工作的时候你总要看公司的代码吧。公司的代码基本是稳定的，具有很强拓展性的。而这些是有意识，无意识的用到设计模式所造成的。如果你想更深的理解公司的代码，设计模式是一定要懂的。

第二，不要局限于设计模式，很多人学习完设计模式，感觉自己的所有代码都可以使用设计模式

了，会出现自己往模式里面套写的代码。对于这种现象我想说，这样虽然不好，会出现过度设计的情况，但是这是每个设计人员必须要经过的道路。当你经过这个过程，你会在需要使用模式的地方再使用，而这些你需要使用的地方，写符合设计模式的代码会变为你无意识的行为，而这样你也会写出自己特有的设计模式出来。

写到这里，深感自己的工程经验还不多，很多的模式还不能深入的理解，等到我更加的深入的理解某一个模式的使用，我会把我体会的放到我的博客 http://singmelody.com 上。

我的代码中还有很多不足和改进之处，希望大牛们在我的博客上多多指出来。

参考资料：

1. 设计模式 机械工业出版社出版
2. 常见设计模式的解析和实现(C++) “那谁”大牛 博客地址：http://www.codedump.info/

最后附上一篇有意思的博文：

追MM与设计模式

作者：佚名 来自：CSDN

创建型模式

1、FACTORY—追 MM 少不了请吃饭了，麦当劳的鸡翅和肯德基的鸡翅都是 MM 爱吃的东西，虽然口味有所不同，但不管你带 MM 去麦当劳或肯德基，只管向服务员说“来四个鸡翅”就行了。麦当劳和肯德基就是生产鸡翅的 Factory

2、BUILDER—MM 最爱听的就是“我爱你”这句话了，见到不同地方的 MM,要能够用她们的方言跟她说这句话哦，我有一个多种语言翻译机，上面每种语言都有一个按键，见到 MM 我只要按对应的键，它就能够用相应的语言说出“我爱你”这句话了，国外的 MM 也可以轻松搞掂，这就是我的“我爱你”builder。（这一定比美军在伊拉克用的翻译机好卖）

3、PROTOTYPE—跟 MM 用 QQ 聊天，一定要说些深情的话语了，我搜集了好多肉麻的情话，需要时只要 copy 出来放到 QQ 里面就行了，这就是我的情话 prototype 了。（100 块钱一份，你要不要）

4、SINGLETON—俺有 6 个漂亮的老婆，她们的老公都是我，我就是我们家里的老公 Sigleton，她们只要说道“老公”，都是指的同一个人，那就是我(刚才做了个梦啦，哪有这么好的事)

结构型模式

5、ADAPTER—在朋友聚会上碰到了一个美女 Sarah，从香港来的，可我不会说粤语，她不会说普通话，只好求助于我的朋友 kent 了，他作为我和 Sarah 之间的 Adapter，让我和 Sarah 可以相互交谈了(也不知道他会不会要我)

6、BRIDGE—早上碰到 MM，要说早上好，晚上碰到 MM，要说晚上好；碰到 MM 穿了件新衣服，要说你的衣服好漂亮哦，碰到 MM 新做的发型，要说你的头发好漂亮哦。不要问我“早上碰到 MM 新做了个发型怎么说”这种问题，自己用 BRIDGE 组合一下不就行了

7、COMPOSITE—Mary 今天过生日。“我过生日，你要送我一件礼物。”“嗯，好吧，去商店，

你自己挑。”“这件 T 恤挺漂亮，买，这条裙子好看，买，这个包也不错，买。”“喂，买了三件了呀，我只答应送一件礼物的哦。”“什么呀，T 恤加裙子加包包，正好配成一套呀，小姐，麻烦你包起来。”“……”，MM 都会用 Composite 模式了，你会了没有？

8、DECORATOR—Mary 过完轮到 Sarly 过生日，还是不要叫她自己挑了，不然这个月伙食费肯定玩完，拿出我去年在华山顶上照的照片，在背面写上“最好的的礼物，就是爱你的 Fita”，再到街上礼品店买了个像框（卖礼品的 MM 也很漂亮哦），再找隔壁搞美术设计的 Mike 设计了一个漂亮的盒子装起来……，我们都是 Decorator，最终都在修饰我这个人呀，怎么样，看懂了吗？

9、FACADE—我有一个专业的 Nikon 相机，我就喜欢自己手动调光圈、快门，这样照出来的照片才专业，但 MM 可不懂这些，教了半天也不会。幸好相机有 Facade 设计模式，把相机调整到自动档，只要对准目标按快门就行了，一切由相机自动调整，这样 MM 也可以用这个相机给我拍张照片了。

10、FLYWEIGHT—每天跟 MM 发短信，手指都累死了，最近买了个新手机，可以把一些常用的句子存在手机里，要用的时候，直接拿出来，在前面加上 MM 的名字就可以发送了，再不用一个字一个字敲了。共享的句子就是 Flyweight，MM 的名字就是提取出来的外部特征，根据上下文情况使用。

11、PROXY—跟 MM 在网上聊天，一开头总是“hi,你好”,“你从哪儿来呀？”“你多大了？”“身高多少呀？”这些话，真烦人，写个程序做为我的 Proxy 吧，凡是接收到这些话都设置好了自动的回答，接收到其他的话时再通知我回答，怎么样，酷吧。

行为模式

12、CHAIN OF RESPONSIBLEITY—晚上去上英语课，为了好开溜坐到了最后一排，哇，前面坐了好几个漂亮的 MM 哎，找张纸条，写上“Hi,可以做我的女朋友吗？如果不愿意请向前传”，纸条就一个接一个的传上去了，糟糕，传到第一排的 MM 把纸条传给老师了，听说是个老处女呀，快跑!

13、COMMAND—俺有一个 MM 家里管得特别严，没法见面，只好借助于她弟弟在我们俩之间传送信息，她对我有什么指示，就写一张纸条让她弟弟带给我。这不，她弟弟又传送过来一个 COMMAND，为了感谢他，我请他吃了碗杂酱面，哪知道他说：“我同时给我姐姐三个男朋友送 COMMAND，就数你最小气，才请我吃面。”，:-(

14、INTERPRETER—俺有一个《泡 MM 真经》，上面有各种泡 MM 的攻略，比如说去吃西餐的步骤、去看电影的方法等等，跟 MM 约会时，只要做一个 Interpreter，照着上面的脚本执行就可以了。

15、ITERATOR—我爱上了 Mary，不顾一切的向她求婚。

Mary：“想要我跟你结婚，得答应我的条件”

我：“什么条件我都答应，你说吧”

Mary：“我看上了那个一克拉的钻石”

我：“我买，我买，还有吗？”
Mary：“我看上了湖边的那栋别墅”
我：“我买，我买，还有吗？”
Mary：“你的小弟弟必须要有 50cm 长”
我脑袋嗡的一声，坐在椅子上，一咬牙：“我剪，我剪，还有吗？”
……

16、MEDIATOR—四个 MM 打麻将，相互之间谁应该给谁多少钱算不清楚了，幸亏当时我在旁边，按照各自的筹码数算钱，赚了钱的从我这里拿，赔了钱的也付给我，一切就 OK 啦，俺得到了四个 MM 的电话。

17、MEMENTO—同时跟几个 MM 聊天时，一定要记清楚刚才跟 MM 说了些什么话，不然 MM 发现了会不高兴的哦，幸亏我有个备忘录，刚才与哪个 MM 说了什么话我都拷贝一份放到备忘录里面保存，这样可以随时察看以前的记录啦。

18、OBSERVER—想知道咱们公司最新 MM 情报吗？加入公司的 MM 情报邮件组就行了，tom 负责搜集情报，他发现的新情报不用一个一个通知我们，直接发布给邮件组，我们作为订阅者（观察者）就可以及时收到情报啦

19、STATE—跟 MM 交往时，一定要注意她的状态哦，在不同的状态时她的行为会有不同，比如你约她今天晚上去看电影，对你没兴趣的 MM 就会说“有事情啦”，对你不讨厌但还没喜欢上的 MM 就会说“好啊，不过可以带上我同事么？”，已经喜欢上你的 MM 就会说“几点钟？看完电影再去泡吧怎么样？”，当然你看电影过程中表现良好的话，也可以把 MM 的状态从不讨厌不喜欢变成喜欢哦。

20、STRATEGY—跟不同类型的 MM 约会，要用不同的策略，有的请电影比较好，有的则去吃小吃效果不错，有的去海边浪漫最合适，但目的都是为了得到 MM 的芳心，我的追 MM 锦囊中有好多 Strategy 哦。

21、TEMPLATE METHOD——看过《如何说服女生上床》这部经典文章吗？女生从认识到上床的不变的步骤分为巧遇、打破僵局、展开追求、接吻、前戏、动手、爱抚、进去八大步骤(Template method)，但每个步骤针对不同的情况，都有不一样的做法，这就要看你随机应变啦(具体实现)；

22、VISITOR—情人节到了，要给每个 MM 送一束鲜花和一张卡片，可是每个 MM 送的花都要针对她个人的特点，每张卡片也要根据个人的特点来挑，我一个人哪搞得清楚，还是找花店老板和礼品店老板做一下 Visitor，让花店老板根据 MM 的特点选一束花，让礼品店老板也根据每个人特点选一张卡，这样就轻松多了；