《设计模式》学习笔记

Blog: http://chuanqi.name
E-mail：chuanqi.tan(at)gmail.com
所有的模式都有详细的 C++实现代码，完整的代码及工程见 Blog

By Chuanqi Tan @ BIT 2010

引言

- 设计模式可以想象为是面向对象软件的设计经验。可以说设计模式就是解决某个特定的面向对象软件问题的特定方法。
- 23 种设计模式的组织编目
  设计模式在粒度和抽象层次上各不相同。由于存在众多的设计模式，我们希望用一种方式将它们组织起来。这一节将对设计模式进行分类以便于我们对各种相关的模式进行引用。分类有助于更快地学习目录中的模式，且对发现的模式也有指导作用，如下表所示。

		目的		
		创建型	结构型	行为型
范围	类	Factory Method	Adapter(类)	Interpreter Template Method
	对象	Abstract Factory Builder Prototype Singleton	Adapter(对象) Bridge Composite Decorator Facade Flyweight Proxy	Chain of Responsibility Command Iterator Mediator Memento Observer State Strategy Visitor

  我们根据两条准则对模式进行分类。

  - 第一是目的准则，即模式是用来完成什么工作的。模式依据其目的可分为创建型（Creational）、结构型（Structural）、行为型（Behavioral）三种。创建型模式与对象的创建有关；结构型模式处理类或对象的组合；行为型模式对类或对象怎样交互和怎样分配职责进行描述。
  - 第二是范围准则，指定模式主要是用于类还是用于对象。类模式处理类和子类之间的关系，这些关系通过继承建立，是静态的，在编译时刻便确定下来。对象模式处理对象间的关系，这些关系在运行时刻是可以变化的，更具动态性。从某种意义上来说，几乎所有模式都使用继承机制，所以“类模式”只指那些集中于处理类间关系的模式，而大部分模式都属于对象模式的范畴。

- 创建型类模式将对象的部分创建工作延迟到子类，而创建型对象模式则将它延迟到另一个对象中。
  结构型类模式使用继承机制来组合类，而结构型对象模式则描述了对象的组装方式。
  行为型类模式使用继承描述算法和控制流，而行为型对象模式则描述一组对象怎样协作完成单个对象所无法完成的任务。

还有其他组织模式的方式。有些模式经常会被绑定在一起使用。
有些模式是可替代的，有些模式尽管使用意图不同，但产生的设计结果是很相似的。

- 客户请求是使对象执行操作的唯一方法，操作又是对象改变内部数据的唯一方法。由于这些限制，对象内部状态是被封装的，它不能被直接访问，它的表示对于外部是不可见的。
- 针对接口编程，而不是针对实现编程。
- 优先使用对象组合，而不是类继承。
- 因为继承对子类揭示了其父类的实现细节，所以继承常被认为“破坏了封装性”。子类中的实现与它的父类有如此紧密的依赖关系，以至于父类实现中的任何变化必然会导致子类发生变化。所以要优先使用对象组合，而不是类继承。
  一个可用的解决方法就是只继承抽象类，因为抽象类通常提供较少的实现。这个方案在 Effective C++中也被反复的提及。
- 委托是一种组合方法，它使组合具有与继承相同的复用能力，在委托方式下，有两个对象参与处理一个请求，接受请求的对象将操作委托给它的代理者。
  委托是对象组合的特例，它告诉你对象组合作为一个代码复用机制可以替代继承。（委托就是普通的组合嘛，没看到有什么特殊的啊！只是将请求转发而已，但是这种思想可以替代继承了。）
- 现在，可复用面向对象软件系统现在一般划分为三大类：应用程序、工具箱和框架(Framework),我们平时开发的具体软件都是应用程序 ;Java 的 API 属于工具箱;而框架是构成一类特定软件可复用设计的一组相互协作的类。EJB（EnterpriseJavaBeans）是 Java 应用于企业计算的框架.
- 除非是为了学习目的，否则只有当一个模式提供的灵活性是真正需要的时候，才有必要使用。

23 种基本设计模式的简单解释：

1. Abstract Factory：提供一个创建一系列相关或相互依赖对象的接口，而无需指定它们具体的类。
2. Adapter：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。
3. Bridge：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。
4. Builder：将一个复杂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。
5. Chain of Responsibility：为解除请求的发送者和接收者之间耦合，而使多个对象都有机会处理这个请求。将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。
6. Command：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，以及支持可取消的操作。

7. Composite：将对象组合成树形结构以表示“部分-整体”的层次结构。它使得客户对单个对象和复合对象的使用具有一致性。
8. Decorator：动态地给一个对象添加一些额外的职责。就扩展功能而言， 它比生成子类方式更为灵活。
9. Facade：为子系统中的一组接口提供一个一致的界面，Facade 模式定义了一个高层接口，这个接口使得这一子系统更加容易使用。
10. Factory Method：定义一个用于创建对象的接口，让子类决定将哪一个类实例化。Factory Method 使一个类的实例化延迟到其子类。
11. Flyweight：运用共享技术有效地支持大量细粒度的对象。
12. Interpreter：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, 该解释器使用该表示来解释语言中的句子。
13. Iterator：提供一种方法顺序访问一个聚合对象中各个元素，而又不需暴露该对象的内部表示。
14. Mediator：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它们之间的交互。
15. Memento：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恢复到保存的状态。
16. Observer：定义对象间的一种一对多的依赖关系，以便当一个对象的状态发生改变时，所有依赖于它的对象都得到通知并自动刷新。
17. Prototype：用原型实例指定创建对象的种类，并且通过拷贝这个原型来创建新的对象。
18. Proxy：为其他对象提供一个代理以控制对这个对象的访问。
19. Singleton：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。
20. State：允许一个对象在其内部状态改变时改变它的行为。对象看起来似乎修改了它所属的类。
21. Strategy：定义一系列的算法，把它们一个个封装起来，并且使它们可相互替换。本模式使得算法的变化可独立于使用它的客户。
22. Template Method：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。Template Method 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步骤。
23. Visitor：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

创建型模式

创建型模式抽象了实例化过程。在这些模式中有两个不断出现的主旋律：第一，它们都将关于该系统使用哪些具体的类的信息封装起来。第二，它们隐藏了这些类的实例是如何被创建和放在一起的。整个系统关于这些对象所知道的是由抽象类所定义的接口。

创建型模式的总结

Factory Method 使一个设计可以定制且只略微有一点复杂。其它的设计模式需要新的类，而 Factory Method 只需要一个新的操作。人们通常将 Factory Method 作为一种标准的创建对象的方法。（这也是 NVI 原则的标准体现）

使用 Abstract Factory、Prototype 或 Builder 的设计甚至比使用 Factory Method 的那些设计更要灵活，但它们也更加复杂。

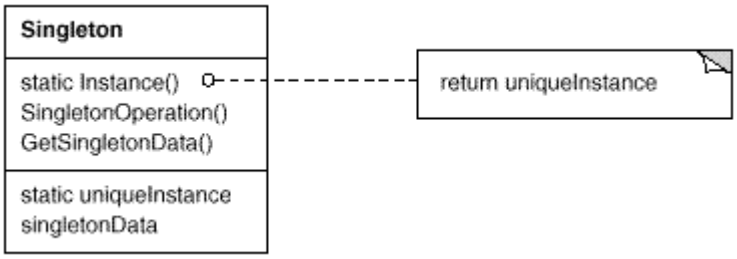
通常，设计以使用 Factory Method 开始，并且当设计者发现需要更大的灵活性时，设计便会向其它创建型模式演化。

各个模式的小结：

- 抽象工厂：生成一系列的产品时使用，对每个系列都有一个具体工厂来负责创建这个系列的产品
- Builder：一步一步的生成复杂的对象产品
- 工厂方法：最基本的创建对象的虚方法
- Prototype：也是生成一系列的产品，而且具有更大的灵活性，更少的类的数目。代价就是每个产品必须实现自我复制
- Singleton：保证一个类只有一个实例，应该使用标准的静态方法来实现，标准的实现方法完美无暇

Singleton（单件）

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。



适用性：

- 当类只能有一个实例而且客户可以从一个从所周知的访问点访问它时
- 当这个唯一的实例应该是通过子类化可扩展的（protected 修饰构造函数），并且客户应该无需更改代码就能使用一个扩展的实例时。

效果：

- Singleton 模式是对全局变量的一种改进，它避免了那些存储唯一实例的全局变量污染命名空间。
- Singleton 可以有子类，而且有这个扩展类的实例来配置一个应用是很容易的
- 使用 Instance()函数来 lazy 初始化 Singleton 避免了 C++中编译单元之间静态对象的依赖问题，这是一种非常好非常常用的手段，用静态对象代替全局变量时都应该使用这种方法

SINGLETON

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

//标准写法实现的单件模式
class Singleton
{
protected:
    //一般将构造方法实现为受保护的，如果不需要被扩展，也可以实现为私有的
    Singleton()
    {
        cout << "A old style singleton constructor" << endl;
    }

private:
    //一个私有的静态成员
    static Singleton * _instance;

public:
    //公有的Singleton方法，返回该类的唯一实例
    static Singleton * Instance()
    {
        if (NULL == _instance)
        {
            //惰性(lazy)初始化，只有使用时才初始化
            _instance = new Singleton();
```

```
        }
        return _instance;
    }

    void Display()
    {
        cout << "Old singleton display" << endl;
    }
};

Singleton * Singleton::_instance = NULL;        //初始化为空指针

//Effective C++上介绍的新的Singleton写法
class NewSingleton
{
private:
    NewSingleton()
    {
        cout << "New style singleton constructor" << endl;
    }

public:
    /*
        新的写法省略了检查NULL==_instance的过程，理论效率更好
        而且减少了实现单件的代码量
        真是太棒了！
    */
    static NewSingleton& Instance()
    {
        static NewSingleton _instance;
        return _instance;
    }

    void Display()
    {
        cout << "New singleton display" << endl;
    }
};

void main()
{
    Singleton::Instance()->Display();
    Singleton::Instance()->Display();
    Singleton::Instance()->Display();
```

```
    Singleton::Instance()->Display();
    Singleton::Instance()->Display();
    Singleton::Instance()->Display();

    NewSingleton::Instance().Display();
    NewSingleton::Instance().Display();
    NewSingleton::Instance().Display();
    NewSingleton::Instance().Display();
    NewSingleton::Instance().Display();
    NewSingleton::Instance().Display();
}
```

事实上，还可以更灵活一点，将 Instance()方法实现如下：

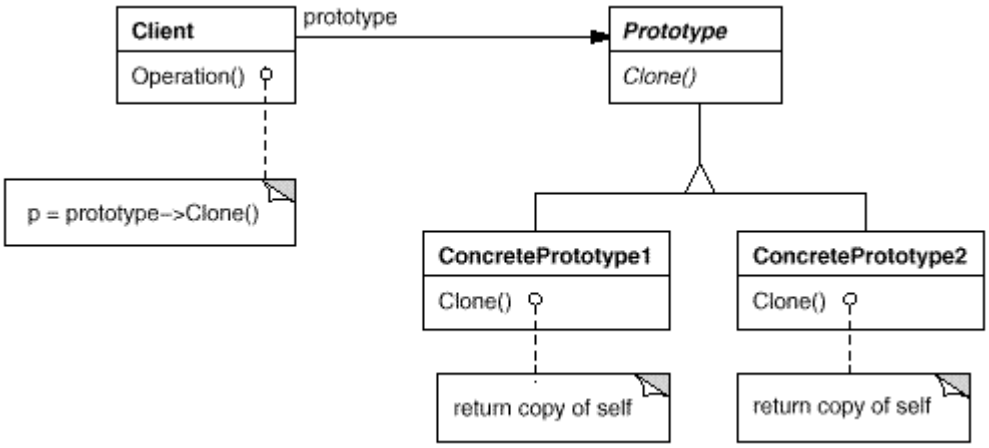
```
//公有的Singleton方法，返回该类的唯一实例
static Singleton* Instance()
{
    if (NULL == _instance)
    { //惰性(lazy)初始化，只有使用时才初始化
        string singleton_set = get_set_from_file("singleton");

        if (singleton_set == "a")
        {
            _instance = new ConcreteSingleton1();
        }
        else if (singleton_set == "b")
        {
            _instance = new ConcreteSingleton2();
        }
        else
        {
            _instance = new DefaultSingleton();
        }
    }
    return _instance;
}
```

不过一般的，基本的实现就非常好了。

Prototype（原型）

用原型实例指定创建对象的种类，并且通过 Copy 这些原型创建新的对象。

在 C++这种静态语言中 ,类不是对象 ,并且运行时刻只能得到很少或者得不到任何类型信息，所以 Prototype 特别有用。而在那些支持反射特性的语言，如 C# java 中，Prototype 就显得不那么重要了。



适用性：

- 当一个系统应该独立于它的产品创建、构成、表示时。
- 当要实例化的类是在运行时刻指定时，例如，通过动态装载。
- 为了避免创建一个与产品类层次平行的工厂类层次时。
- 当一个类的实例只能有几个不同状态组合中的一种时 ,建立相应数目的原型并 Clone 它们可能比每次用合适的状态手工实例化该类更方便一些。

优点：

- 允许在运行时动态加载或动态组合出新类，它比其它创建模式更灵活，因为客户可以在运行时刻建立和删除原型。
- 这种设计使得用户无需编程即可定义新“类”
- 能够大大的减少子类的构造，仅需要定义它们的特点类即可

缺点：

- 主要缺陷是每一个 Prototype 的子类都必须实例 Clone 操作，这可能很困难。例如，当所考虑的类已经存在时就难以新增 Clone 操作。当内部包括一些不支持拷贝或有循环引用的对象时，实现克隆可能也会很困难。

转载一个很好的比喻：

举一个例子来解释这个模式的作用,假设有一家店铺是配钥匙的,他对外提供配制钥匙的服务(提供 Clone 接口函数),你需要配什么钥匙它不知道只是提供这种服务,具体需要配什么钥匙只有到了真正看到钥匙的原型才能配好.也就是说,需要一个提供这个服务的对象,同时还需要一个原型(Prototype),不然不知道该配什么样的钥匙.

Prototype 和 Abstract Factory 模式在某种方面是相互竞争的,它是它们也是可以一起使用。Abstract Factory 可以存储一个被克隆的原型的集合，并且返回产品对象。

感觉上，prototype 模式与 abstract factory 模式都是用来产生一系列的产品，由于 Prototype 模式可以进行随意组合，极大的减少了类的数目，代价就是每个产品必须能够自我复制。

PROTOTYPE

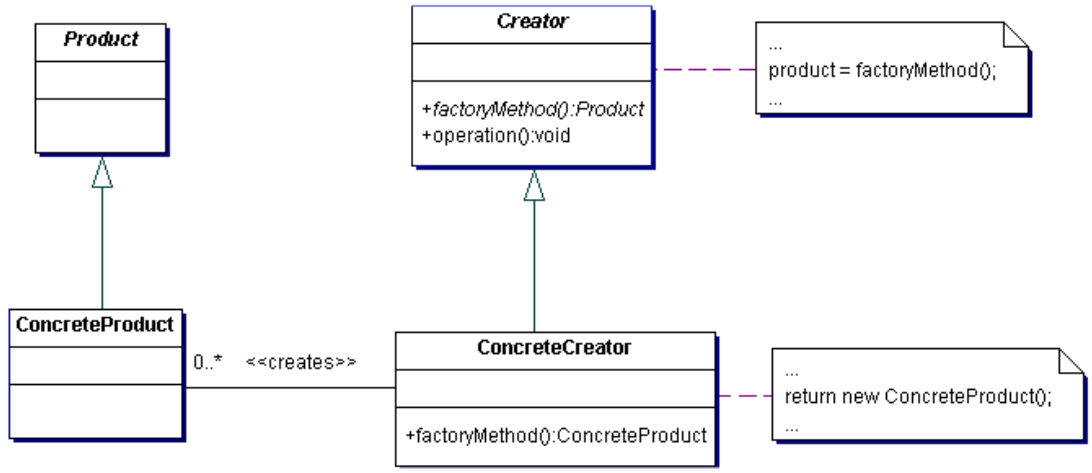
```
#include <vector>
#include <string>
#include <iostream>
#include <sstream>
using namespace std;


//原型的虚基类
class Prototype
{
public:
	Prototype(const string &state)
		: _state(state)
	{}

	string State()
	{
		return _state;
	}

	virtual ~Prototype(){}

	//一般来说,这个都为纯虚函数,很难提供一个好的默认实现
	virtual Prototype* clone() = 0;

protected:
	string _state;
};

//实际派生类1
class ConcretePrototype1 : public Prototype
{
public:
	ConcretePrototype1(const string &state)
		: Prototype(state)
```

```
    {}

    virtual ~ConcretePrototype1(){}

    virtual Prototype* clone()
    {
        return new ConcretePrototype1(_state);
    }
};

//实际派生类2
class ConcretePrototype2 : public Prototype
{
public:
    ConcretePrototype2(const string &state)
        : Prototype(state)
    {}

    virtual ~ConcretePrototype2(){}

    virtual Prototype* clone()
    {
        return new ConcretePrototype2(_state);
    }

private:
    string _state;
};

int _tmain(int argc, _TCHAR* argv[])
{
    Prototype *p1 = new ConcretePrototype1("Prototype state 1");
    Prototype *p2 = new ConcretePrototype1("Prototype state 2");

    cout << p1->State() << endl;
    cout << p2->State() << endl;

    Prototype *p3 = p1->clone();
    Prototype *p4 = p2->clone();

    cout << p3->State() << endl;
    cout << p4->State() << endl;

    return 0;
```

}

Factory Method（工厂方法）

定义一个用于创建对象的接口，让子类决定实例化哪一个类。Factory Method 使一个类的实例化延迟到其子类。



可以这么理解：一个工厂方法就是一个真正负责创建对象的“虚”方法。这个 UML 中的 operation()，它看起来像是在创建对象，实际上它在调用 factoryMethod()来实际创建对象，由于 factoryMethod 是虚方法，所以可以通过子类重载 ConcreteCreator 来重写父类的虚方法。（甚至 Creator 类的 factoryMethod 是一个纯虚方法）。

operator()是 public 的，属于接口的一部分。而 factoryMethod()可以是 protected 或者 private 的，这也满足 NVI（None Virtual Interface 非虚接口）的要求。

工厂方法用来生产单个产品，即使这样实现，所生产的产品也必须满足 product 接口。如果需要生产一系列不同的产品，应该使用抽象工厂模式。

根据情况，工厂方法可以可以有一个缺省的实现，也可以没有一个缺省的实现。

C++中的工厂方法都是虚函数并且常常是纯虚函数。而且一定要注意不能在 Creator 的构造函数中调用工厂方法，因为在构造函数中多态性是不可用的。

C++中可以使用模板来避免创建子类。

Abstract Factory 模式经常是用工厂方法来实现（因为 Abstract Factory 可以视为一系列工厂方法的集合）。工厂方法通常在 Template Methods 中被调用。Prototypes 不需要创建

Creator 的子类。但是，它们通常要求一个针对 Product 类的 Initialize 操作。Creator 使用 Initialize 来初始化对象，而 Factory Method 不需要这样的操作。

FACTORY METHOD

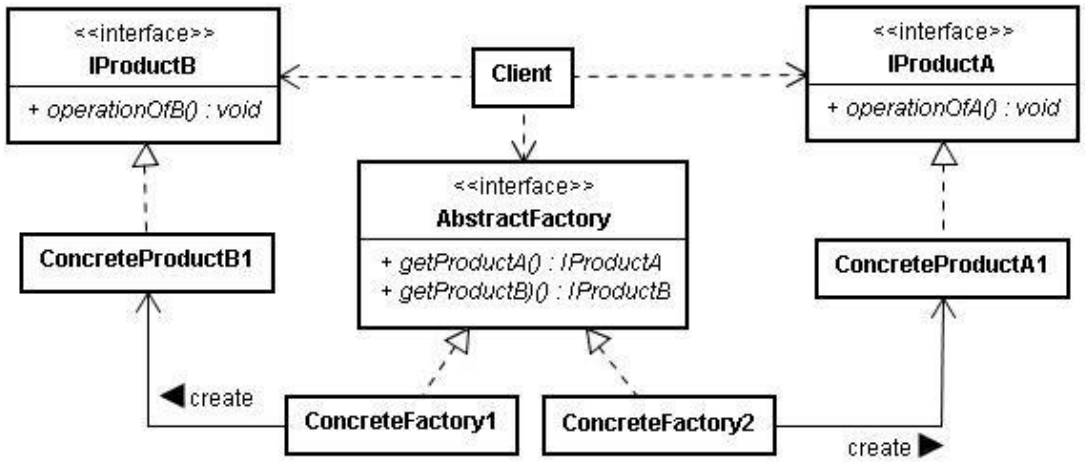
```
#include <vector>
#include <string>
#include <iostream>
#include <sstream>
using namespace std;


class AbstractProduct
{
public:
	virtual ~AbstractProduct(){}

	virtual void product_active() = 0;
};

class DefaultProduct : public AbstractProduct
{
public:
	virtual void product_active()
	{
		cout << "DefaultProduct actived!" << endl;
	}
};

class ConcreteProduct : public AbstractProduct
{
public:
	virtual void product_active()
	{
		cout << "ConcreteProduct actived!" << endl;
	}
};


class Creator
{
public:
	Creator()
	{}
```

```
    //这个方法看起来在创建对象，实际上它是调用虚方法来创建对象
    void GenerateProduct()
    {
        _product = create_product();
    }

    AbstractProduct * GetProduct()
    {
        return _product;
    }

protected:
    //Factory Method
    //带缺省实现，如果不需要缺省实现，此方法可以定义为纯虚函数
    virtual AbstractProduct * create_product()
    {
        return new DefaultProduct();
    }


private:
    AbstractProduct *_product;
};


class ConcreteCreator : public Creator
{
public:
    ConcreteCreator(){}

protected:
    //有一个string product_type可以实现子类中仅仅重写感兴趣的产品
    virtual AbstractProduct * create_product()
    {
        return new ConcreteProduct();

        //不感兴趣的产品交给缺省实现来完成
        //return Creator::create_product();
    }

};

int _tmain(int argc, _TCHAR* argv[])
{
```

```
    Creator *c = new ConcreteCreator();
    c->GenerateProduct();
    c->GetProduct()->product_active();

    return 0;
}
```

Abstract Factory（抽象工厂）

抽象工厂提供一个创建**一系列**相关或相互依赖对象的接口，而无需指定它们的具体类。



在以下情况下可以使用 Abstract Factory 模式

- 一个系统要独立于它的产品的创建、组合和表示时
- 一个系统要由多个产品系列中的一个来配置时
- 当你要强调一系列相关的产品对象的设计以便进行联合使用时
- 当你要提供一个产品类库，而只想显示它们的接口而不是实现时

效果与实现：

1. 有几种可变的方案，就有几个具体工厂来产生这些产品
2. 它分离了具体的类，把所有的创建产品的任务交给一个工厂来完成。抽象类只是一个接口
3. 易于交换产品系列：通过使用不同的具体工厂就可以方便的交换整个产品系列
4. 它有利于产品的一致性：当一个系列中的产品对象被设计成一起工作时，一个应用一次只能使用同一个系列中的对象，这点很重要。而 Abstract Factory 可以完美的实现这一点
5. 一个具体的工厂通常是一个 Singleton
6. 最通常的办法就是为每一个产品定义一个工厂方法（Factory Method）。

7. 由于具体工厂通常要重新实现所有的产品生产方法，即使它们的差别很小。这样就可以通过与 Prototype 模式结合来减少工作量
8. 一个常见的技巧就是使得 AbstractFactory 类不是抽象类，但是它的每一个产品创建方法都是虚方法。这样使得从 AbstractFacotry 派生出一个具体工厂来变得非常容易，只需要重写必须改变的方法即可，而不必重写所有的创建产品的方法。
9. Abstract Factory 类通常用工厂方法（Factory Method）实现，但它们也可以用 Prototype 实现

自我总结：

1. 抽象工厂模式一直强制的是创建**一系列**的产品。
2. Abstract Factory 的一个常见应用是用于创建不同视感风格的界面

ABSTRACT FACTORY

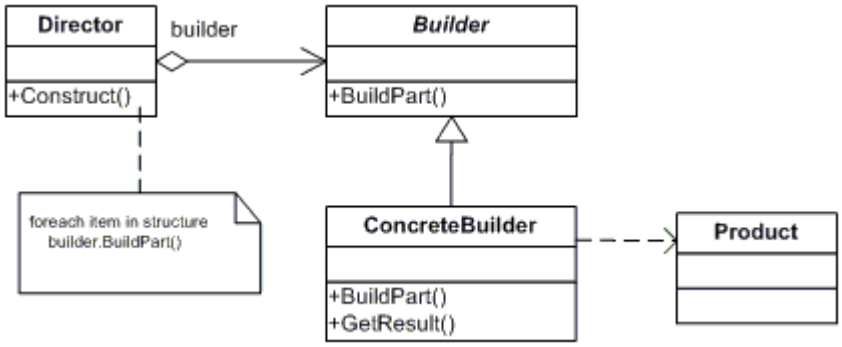
```
#include "stdafx.h"
#include <string>
#include <iostream>
using namespace std;

class ProductA
{
public:
	ProductA(const string &sign)
		: _sign(sign)
	{}

	string Sign()
	{
		return _sign;
	}

private:
	string _sign;
};

class ProductB
{
public:
	ProductB(const string &sign)
		: _sign(sign)
	{}

	string Sign()
	{
```

```
        return _sign;
    }

private:
    string _sign;
};


class AbstractFactory
{
public:
    virtual ~AbstractFactory(){}

    virtual ProductA * GetProductA() = 0;
    virtual ProductB * GetProductB() = 0;
};

class ConcreteFactory1 : public AbstractFactory
{
public:
    virtual ProductA * GetProductA()
    {
        return new ProductA("ProductA made by ConcreteFactory1");
    }
    virtual ProductB * GetProductB()
    {
        return new ProductB("ProductB made by ConcreteFactory1");
    }

};

class ConcreteFactory2 : public AbstractFactory
{
public:
    virtual ProductA * GetProductA()
    {
        return new ProductA("ProductA made by ConcreteFactory2");
    }
    virtual ProductB * GetProductB()
    {
        return new ProductB("ProductB made by ConcreteFactory2");
    }

};
```

```
class Client
{
public:
	Client(AbstractFactory *factory)
	{
		_pA = factory->GetProductA();
		_pB = factory->GetProductB();
	}

	ProductA * GetProductA()
	{
		return _pA;
	}

	ProductB * GetProductB()
	{
		return _pB;
	}

private:
	ProductA * _pA;
	ProductB * _pB;
};

int _tmain(int argc, _TCHAR* argv[])
{
	AbstractFactory *f1 = new ConcreteFactory1();
	AbstractFactory *f2 = new ConcreteFactory2();

	Client c1(f1);
	cout << c1.GetProductA()->Sign() << endl;
	cout << c1.GetProductB()->Sign() << endl;

	Client c2(f2);
	cout << c2.GetProductA()->Sign() << endl;
	cout << c2.GetProductB()->Sign() << endl;

	delete f1;
	delete f2;
	return 0;
}
```

Builder（生成器）

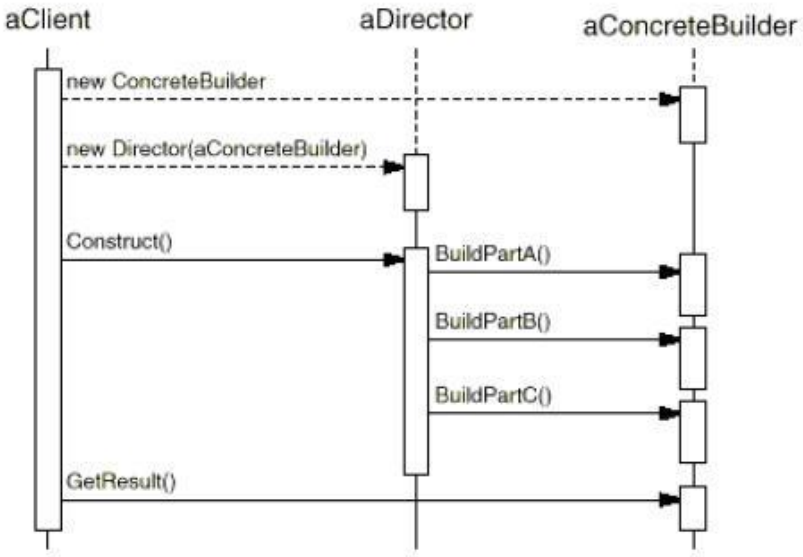
将一个复杂对象的构建与它的表示分离，使得同样的构建过程可以创建不同的表示。



以下情况适用 Builder 模式：

- 当创建复杂算法应该独立于该对象的组成部分以及它们的装配方式时
- 当构造过程必须允许被构造的对象有不同的表示时

各个部件之间的协作流程：



Builder 模式与一下子就生成产品的创建型模式不同，它是在导向者的控制下一步一步构造产品的。仅当该产品完成时导向者才从生成器中取回它。

编译子系统中的 Parser 类就是一个 Director，它以一个 ProgramNodeBuilder 对象作为构造的参数。每当 Parser 对象识别出一个语法结构时，它就通知它的 ProgramNodeBuilder 对象。

Abstract Factory 与 Builder 相似，因为它们都可以用来创建复杂对象。主要的区别是 Builder 模式着重于一步步构造一个复杂对象。而 Abstract Factory 着重于多个系列的产品对象(简单的或复杂的)。Builder 在最后一步返回产品，而对于 Abstract Factory 来说，产品是主即返回的。

生成器模式强调的是一步一步的构建对象！

BUILDER

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

class Builder
{
public:
	// 方法不为纯虚函数
	// 使得子类可以只重写它们感兴趣的实现
	// 当然也可以使用通常的纯虚函数实现方法
	virtual void BuilderPart(string aPart){}

	virtual string GetResult(){  return NULL; }
};

class SimpleBuilder : public Builder
{
public:
	virtual void BuilderPart(string aPart)
	{
		_result << aPart << endl;
	}

	virtual string GetResult()
	{
		return _result.str();
	}

private:
	stringstream _result;
};

class Director
{
public:
```

```
    Director(Builder *builder)
    {
        _builder = builder;

        vector<string> _parts;
        _parts.push_back("Part A");
        _parts.push_back("Part B");
        _parts.push_back("Part C");

        for (vector<string>::iterator iter = _parts.begin(); iter != _parts.end(); ++iter)
        {//一步一步的构建好对象
            _builder->BuilderPart(*iter);
        }
    }

    string GetResult()
    {
        return _builder->GetResult();
    }

private:
    Builder *_builder;
    string _result;
};

int _tmain(int argc, _TCHAR* argv[])
{
    Builder *b = new SimpleBuilder();
    Director d(b);
    cout << d.GetResult() << endl;

    return 0;
}
```

结构型模式

结构型模式涉及到如何组合类和对象以获得更大的结构。结构型类模式采用继承机制来组合接口或实现。

结构型模式的讨论

结构型模式具有很大的相似性，都是通过对象的继承、组合来实现的。

模式之间最重要的是它们之间的差异，理解它们的目的以及使用场合。

Adapter 与 Bridge：

- Adapter 模式主要是为了解决两个已有接口之间的不匹配；Bridge 则对抽象接口与它的实现部分进行分离并桥接
- Adapter 为了使两个独立的接口能相互协调工作；Bridge 则是为了给用户提供一个稳定的接口（类库二进制上的兼容性）
- Adapter 在类已经设计好之后实施；Bridge 模式在设计类之前实施

Facade 与 Adapter 区别在于：Facade 定义了一个新的接口，而 Adapter 则复用了原有的接口。适配器使两个已有的接口协同工作，而不是定义一个新的接口。

Composite 与 Decorator：

- Decorator 旨在使你能够不需要生成子类即可以给对象添加职责；Composite 则旨在构造类，使多个相关的对象能够以统一的方式进行处理，而多重对象可以被当作一个对象来处理。
- Composite 与 Decorator 通常协同使用：这时系统中将会有一个抽象类，它有一些 Composite 子类和 Decorator 子类，还有一些实现系统的基本构建模块。此时，composites 和 decorator 将拥有共同的接口。从 Decorator 角度来看，Composite 是一个 ConcreteComponent。而从 Composite 模式的角度来看，Decorator 则是一个 Leaf。

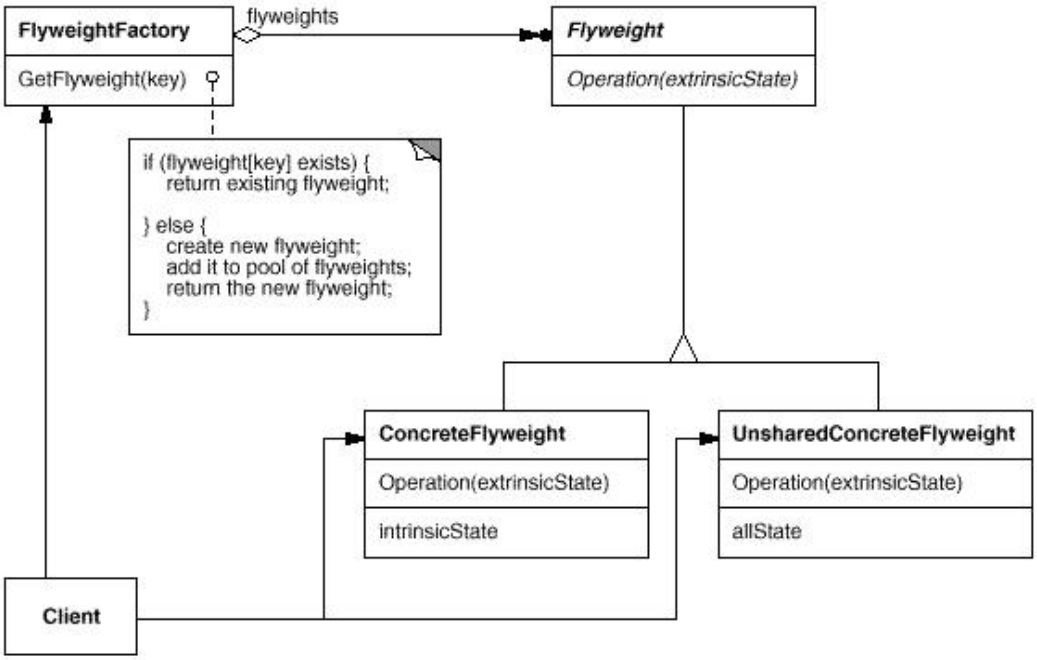
Decorator 与 Proxy：

- Decorator 适用于在编译时不能（至少不方便）确定对象的全部功能的情况；而 Proxy 是不能动态的添加和分离性质，它也不是为递归组合而设计的，它的目的是当访问一个实体不方便或不符合需求时，为这个实体提供一个替代者

- Decorator 组件提供了部分功能，而一个或多个 Decorator 负责完成其它的功能；Proxy 模式中，实体定义了关键功能，而 Proxy 选择提供（或拒绝）对它的访问

Flyweight（享元）

运用共享技术有效地支持大量细粒度的对象。



有些应用程序得益于在整个设计过程中采用对象技术，但简化的实现代价有时候极大。非常显而易见的就是字符的处理程序，在字符的处理程序中有大量的字符对象，而且字符对象的各类却极为有限（ASCII 字符表）。

Flyweight 是一个共享对象，它可以同时在多个场景中使用，并且在每个场景中 Flyweight 都可以作为一个独立的对象。Flyweight 不能对它所运行的场景作任何的假设，这里的关键就是要理解“**内部状态**”与“**外部状态**”之间的区别。内部状态存储于 Flyweight 中，它包含了独立于场景的信息，这些信息使得 Flyweight 可以被共享。而外部状态取决于 Flyweight 场景，并根据场景变化，因此不可以共享。用户对象负责在必要的时候将外部状态传递给 Flyweight。

适用性：

- 一个应用程序使用了大量的对象
- 完全由于使用大量的对象，造成很大的**空间**上的开销
- 对象的大多数状态都可变为外部状态
- 如果删除对象的外部状态，那么可以用相对较少的共享对象取代很多组对象
- 应用程序不依赖于对象标识。由于 Flyweight 对象可以被共享，对于概念上明显有别的对象，标识测试将返回真值

需要注意的：

- Flyweight 是典型的用“时间”换“空间”，因为加了一层中间层。
- 并非所有的 Flyweight 都需要被共享。Flyweight 接口使得共享成为可能，但它并不强制共享。UnsharedConcreteFlyweight 对象通常将 ConcreteFlyweight 对象作为子节点（UnsharedConcreteFlyweight 是字符处理程序中 Row 和 Column 的常见解决方案）
- 用户不应直接对 ConcreteFlyweight 类进行实例化，而只能从 FlyweightFactory 对象得到 ConcreteFlyweight 对象，这可以保证对它们进行适当的共享（我的方法是用友元+私有构造函数）。
- 共享对象可能还需要引用计数和垃圾回收。如果 Flyweight 数目固定而且很少时，这可省略地将 Flyweight 永久保存。

相关模式：

- Flyweight 模式通常和 Composite 模式结合起来，用共享结点的有向无环图实现一个逻辑上的层次结构。
- 通常，最好用 Flyweight 实现 State 和 Strategy 对象。
- FlyweightFactory 通常又可以实现为 Singleton。

FLYWEIGHT

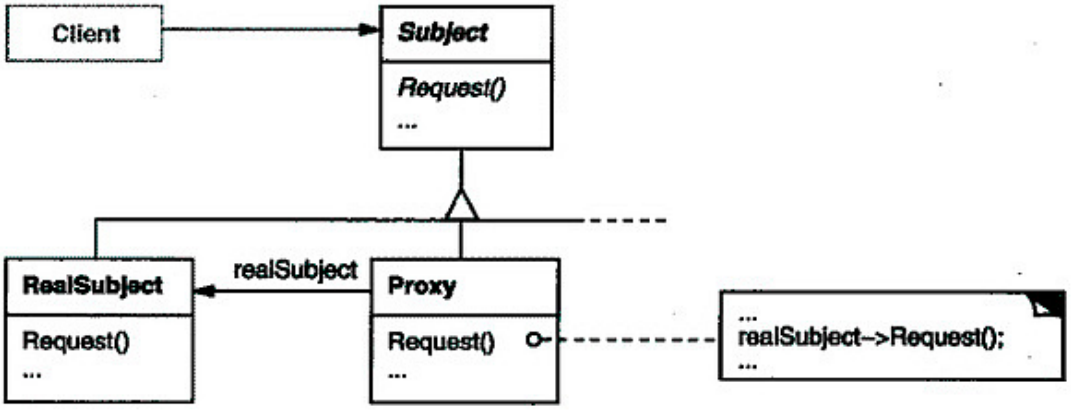
```
#include <vector>
#include <map>
#include <string>
#include <iostream>
#include <sstream>
#include <algorithm>
using namespace std;


//抽象的享元基类
class Flyweight
{
public:
	//基类Flyweight最重要的就是提供了一个根据“外部状态”进行操作的接口
	//这里操作是：打印字符串和当前的位置。
	//其中： 字符串是内部状态、位置信息是外部状态
	virtual void PrintStringAndPostion(const string &extrinsic_state) = 0;

	virtual ~Flyweight(){}
};
```

```
//具体的享元类
class ConcreteFlyweight : public Flyweight
{
	//声明FlyweightFactory为友元
	friend class FlyweightFactory;

public:
	//根据 内部、外部状态 进行打印
	virtual void PrintStringAndPostion(const string &extrinsic_state)
	{
		cout << "Operation with : " << _intrinsic_state << "\t" << extrinsic_state << endl;
	}

private:
	//将构造函数声明为私有，这样就只能由友元FlyweightFactory来产生享元实例了
	ConcreteFlyweight(const string &key)
		: _intrinsic_state(key)
	{
		cout << "Constructor a new Flyweight by key = " << key << endl;
	}

	string _intrinsic_state;
};


/*
享元工厂类
每次都由享元工厂来产生享元可保证相同的享元会共享一个实例
*/
class FlyweightFactory
{
private:
	FlyweightFactory(){}

public:
	//New Signlton Method
	static FlyweightFactory& Instance()
	{
		static FlyweightFactory _instance;
		return _instance;
	}

	Flyweight* GetFlyweight(const string &key)
	{
```

```
        if (_flyweights.count(key) == 0)
        {
            Flyweight *new_flyweight = new ConcreteFlyweight(key);
            _flyweights.insert(pair<string, Flyweight*>(key, new_flyweight));
        }

        return _flyweights[key];
    }

private:
    map<string, Flyweight*> _flyweights;
};


void main()
{
    vector<Flyweight*> vec;
    for (size_t i = 0; i < 10; ++i)
    {
        //十个不同位置的ConcreteFlyweight对象实际上只产生了两个不同的ConcreteFlyweight对象
        vec.push_back(FlyweightFactory::Instance().GetFlyweight(i % 2 == 0 ? "abc" : "xyz"));
    }

    for (size_t i = 0; i < 10; ++i)
    {
        //位置信息作为“外部状态”，在调用时传递给Flyweight
        stringstream position;
        position << "At position: " << i;

        vec[i]->PrintStringAndPostion(position.str());
    }
}
```

Proxy（代理）

为其它对象提供一种代理以控制对这个对象的访问



图：代理的结构

对一个对象进行访问控制的一个原因是为了只有在我们确实需要这个对象时才对它进行创建和初始化，这种称为虚拟代理；或者是为了对访问进行权限管理，这种称为保护代理；或者是为了取代简单的指针，进行引用计数、自动释放或者是附加一些操作，这种称为智能指针；或者是为了使得其它对象常见远程对象使用起来像本地对象一样方法，这种称为远程代理。

Proxy 与 RealSubject 有一个共同的基类 Subject，这使得在所有可以使用 RealSubject 的地方都可以使用 Proxy 来代替。

Proxy 还可以用来实现 copy-on-write 技术。

适用性：

- 远程代理：为一个对象在不同的地址空间提供局部代理
- 虚代理：根据需要创建开销大的对象
- 保护代理：控制对原始对象的访问，比如在根据权限对行为进行控制的保护代理是非常常见和有用的
- 智能指针：取代简单指针。附加一些操作，如引用计数、自动释放、线程加锁

自动转发和手动转发的区别：

- 自动转发方便，但是无法区别出一些虽然被引用但是仍然可以延迟加载的操作（比如获取图像的大小，不用加载图像也可以得到）
- 手动转发需要重写需要操作，但是可以真正的在必须加载对象时才加载。

相关模式：

- Adapter：适配器为它所适配的对象提供了一个不同的接口。相反，代理提供了与它的实体相同的接口。而且，用于访问保护的代理可能会拒绝一些操作，因此它提供的接口实际上是实体接口的一个子集
- Decorator：尽管 Decorator 的实现部分与代理相似，但是 Decorator 的目的不同。Decorator 为对象添加一个或多个功能，而代理则控制对对象的访问
- 代理的实现与 Decorator 类似，但是在相似程度上有所差别。Protection Proxy 的实现可能与 Decorator 差不多；Remote Proxy 不包含对实体的直接引用，而只是一个间接引用，如“主机 ID，主机上的局部地址”；Virtual Proxy 开始的时候使用一个间接引用，如文件名，但最终将获取并使用一个直接引用

PROXY

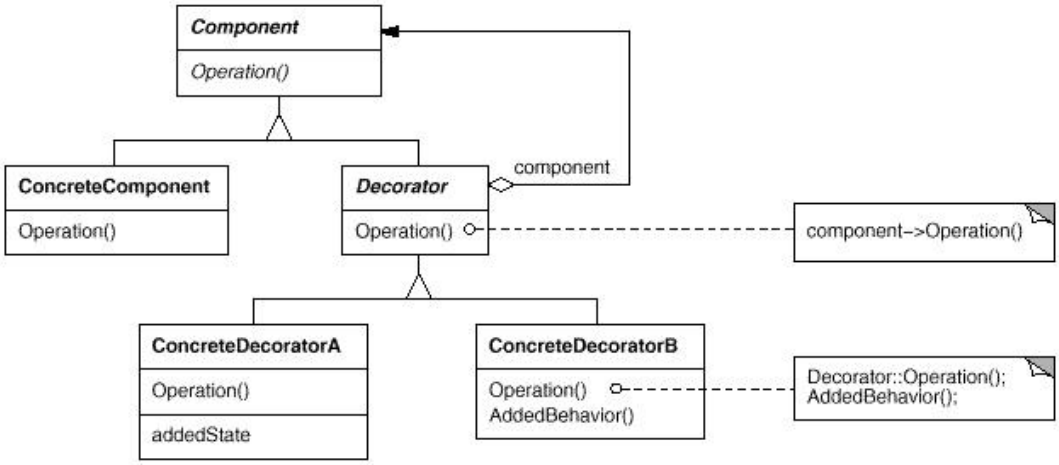
```
#include <vector>
#include <map>
#include <string>
#include <iostream>
#include <sstream>
#include <algorithm>
using namespace std;


/*
共同的抽象基类
由于Image与ImageVirtualProxyPtr具有相同的接口
所以在任何使用Image的地方就可以用ImageVirtualProxyPtr代替
*/
class ImageBase
{
public:
	virtual void DrawSelf() = 0;
};


/*
具体的图像类
*/
class Image : public ImageBase
{
public:
	Image(string file_path)
		: _file_path(file_path)
	{
		cout << "Load Image :" << file_path << endl;
```

```
    }

    virtual void DrawSelf()
    {
        cout << _file_path << " draw it self" << endl;
    }

private:
    string _file_path;
};


/**
 *  图像的虚拟代理指针
 *  只有在的确需要对图像进行绘制的时候才真正的去加载图像，做到了Lazy-Calcuate
 */
class ImageVirtualProxy : public ImageBase
{
public:
    ImageVirtualProxy(string file_path)
        : _file_path(file_path), _img(0)
    {}

    ~ImageVirtualProxy()
    {
        if (_img != 0)
        {
            delete _img;
            _img = 0;
        }
    }

    virtual void DrawSelf()
    {
        //手动转发：手动去写每一个需要转发的函数
        GetImage()->DrawSelf();
    }

    Image * operator->()
    {
        //也可以这样自动实现转发，不过这样的话就把代理类视为一个指针，智能指针多是这么实现的
        return GetImage();
    }
```

```
private:
        Image * GetImage()
        {
                if (_img == 0)
                {
                        _img = new Image(_file_path);
                }
                return _img;
        }

        string _file_path;
        Image *_img;
};


void main()
{
        ImageVirtualProxy img("c:\\love.jpg");
        cout << "Image has not been loaded now" << endl;

        //延迟到使用时才会去加载图像
        img.DrawSelf();         //调用手动的转发
        img->DrawSelf();        //将代理视为指针时的自动转发
}
```

Decorator（修饰者）

动态地给一个对象添加一些额外的职责。就增加功能来说，Decorator 模式相比生成子类更为灵活



有时我们希望给某个对象而不是整个类添加一些功能时，一种较为灵活的方式是将组件嵌入到另一个对象中，由这个对象添加边框。我们称这个嵌入的对象为**装饰**。由于装饰的透明性使得可以递归的嵌套多个装饰，从而可以添加任意多的功能。

适用性：

- 在不影响其它对象的情况下，以动态、透明的方式给单个对象添加职责。
- 处理那些可以撤消的职责。
- 当不能采用生成子类的方法进行扩充时。

装饰对象的接口(Decorator)必须与它所装饰的 Component 的接口是一致的，因此所有的 ConcreteDecorator 类必须有一个公共的父类。

因为修饰者即可以包括其它的修饰者，又可以包括真正的组件，这就要求他们必须有一个公共基类(Component)。甚至可以这么理解：修饰者本身就是一种特殊的组件。

理解 Decorator 与 Strategy 模式的区别与联系：

- Decorator 模式改变的是对象的外观，而 Strategy 模式改变的是对象的内核。
- 由于 Decorator 模式仅仅从外部改变组件，因此组件无需对它的装饰有任何了解；也就是说这些装饰对该组件是透明的，如下所示：
  aDecorator.component --> aDecorator.component --> aComponent
- 在 Strategy 模式中，Component 组件本身是知道可能进行哪些扩充的，因此它必须引用并维护相应的策略，如下所示：
  aComponent.strategies --> aStrategy.next --> aStrategy
- 基于 Strategy 模式的方法可能需要修改 component 组件以适应新的扩充。

相关模式：

- Adapter 模式：Decorator 模式不同于 Adapter 模式，因为装饰仅改变对象的职责而不改变它的接口；而适配器将给对象一个全新的接口。
- Composite 模式：可以将装饰视为一个退化的、仅有一个组件的组合。然而，装饰仅给对象添加一些额外的职责 - - 它的目的不在于对象聚集。
- Strategy 模式：用一个装饰你可以改变对象的外表；而 Strategy 模式使得你可以改变对象的内核。这是改变对象的两种途径。

DECORATOR

```
#include <vector>
#include <string>
#include <iostream>
#include <sstream>
#include <algorithm>
using namespace std;


//组件与修饰者共同的抽象基类
//因为修饰者既可以包含其它的修饰者也可以包含真正的组件，所以它们需要一个共同的基类
class Component
{
public:
	virtual void Operation() = 0;

	virtual ~Component(){}
};


//真正的组件
class ConcreteComponent : public Component
{
public:
	virtual void Operation()
	{
		cout << "ConcreteComponent Operation" << endl;
	}
};


//修饰者的抽象基类，修饰者最重要的特征就是包含一个指向共同基类的指针Component *_component
class Decorator : public Component
{
```

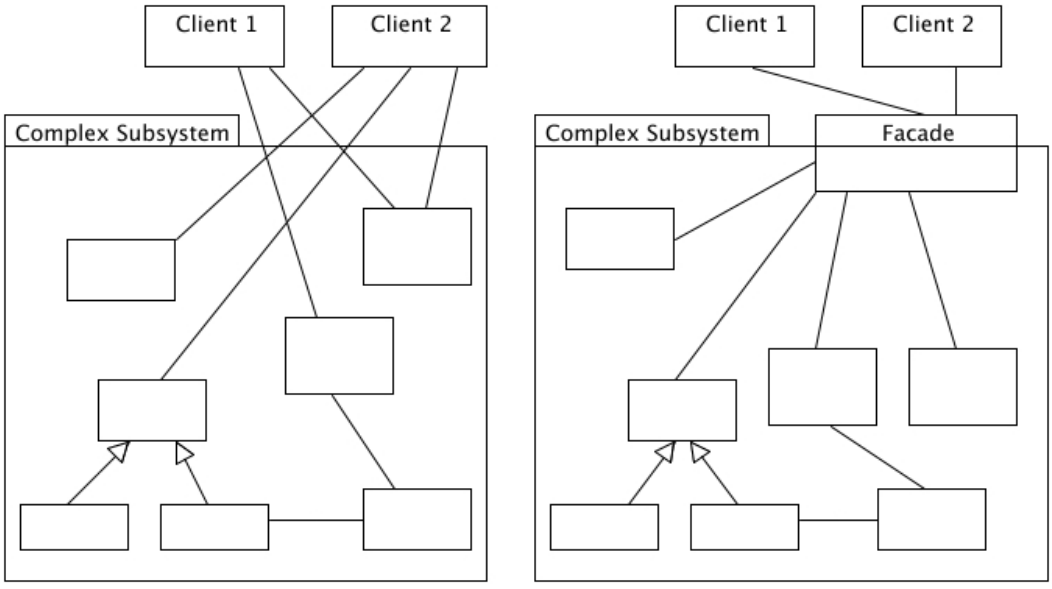
```
public:
    virtual void Operation()
    {
        //缺省的行为：转发
        if (_component != NULL)
        {
            _component->Operation();
        }
    }

protected:
    //修饰者必须包含Component，所以要在构造函数中初始化_component成员
    //不能有无参的构造函数
    Decorator(Component *c) : _component(c)
    {}

private:
    //这里_component用private修饰好，一个修饰者不用去关心它所修饰的组件的细节
    Component *_component;
};


//具体的修饰者
//提供了一个AddBehavior()来附加新的行为
class ConcreteDecorator : public Decorator
{
public:
    ConcreteDecorator(Component *c)
        : Decorator(c)
    {}

    virtual void Operation()
    {
        Decorator::Operation();    //不去关心细节，仅仅链式调用
        AddBehavior();
    }

private:
    void AddBehavior()
    {
        cout << "ConcreteDecorator AddBehavior" << endl;
    }
};
```

```
void main()
{
	ConcreteComponent c;
	ConcreteDecorator d1(&c);
	ConcreteDecorator d2(&d1);
	d2.Operation();
}
```

Facade（外观）

为子系统中的一组接口提供一个一致的界面，Facade 模式定义了一个高层接口，这个接口使得这一子系统更加容易使用。

我觉得可以形像地称之为“系统边界接口”，一个中间层吧！



子系统的外观提供了大多数时候使用子系统的快捷方式，但同时对需要使用底层子系统的人来说，它也不会隐藏这些功能。对那些需要更多的功能的用户可以直接的越过 Facade 层。

尽管是子系统中的有关对象在做实际的工作，但 Facade 模式本身也必须将它的接口转换成子系统的接口。

使用 Facade 模式时可以进一步的降低客户 - 子系统之间的耦合度：用抽象类实现 Facade 而它的具体类对应于不同的子系统实现，这可以进一步的降低客户与子系统之间的耦合度。这样客户就可以通过抽象的 Facade 类接口与子系统通讯。这种抽象耦合使得客户不知道它使用的是

子系统的哪一个实现的。

相关模式：

- Abstract Factory 模式可以与 Facade 模式一起使用以提供一个接口，这一接口可用来以一种子系统独立的方式创建子系统对象。Abstract Factory 也可以代替 Facade 模式隐藏那些与平台相关的类。
- Mediator 模式与 Facade 模式的相似之处是，它抽象了一些已有的类的功能。然而 Mediator 的目的是对同事之间的任意通讯进行抽象，通常集中不属于任何单个对象的功能。Mediator 的同事对象知道中介者并与它通信，而不是直接与其它同类对象通信。相对而言，Facade 模式仅对子系统对象的接口进行抽象，从而使它们更加容易使用；它并不定义新功能，子系统也不知道 Facade 的存在。
- 通常来讲，仅需要一个 Facade 对象，因此 Facade 对象通常属于 Singleton 模式。

FACADE

```
#include <vector>
#include <string>
#include <iostream>
#include <sstream>
#include <algorithm>
using namespace std;


//扫描器
class Scanner
{
public:
	string Scan(string file_name)
	{
		cout << "Scanner scan file : " << file_name << endl;
		return "Scan Result";
	}
};

//预处理器
class PreCopeer
{
public:
	void PreCope(string to_precope)
	{
		cout << "PreCoper do precope" << endl;
	}
```

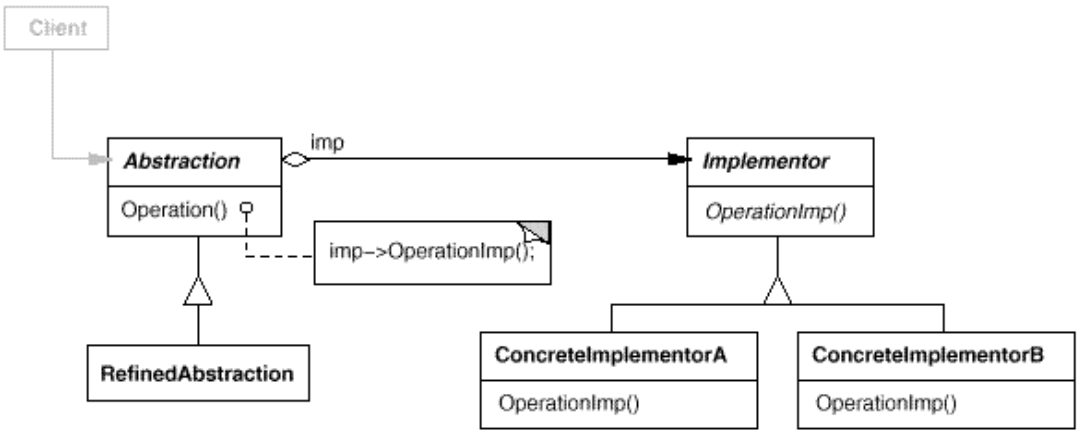
```
};

//编译器
class Complier
{
public:
	string Complie(string to_complie)
	{
		cout << "Complier do complie" << endl;
		return "Complie Result";
	}
};

//链接器
class Linker
{
public:
	string Link(string to_link)
	{
		cout << "Linker do link" << endl;
		return "A execute file";
	}
};

//编译系统的外观
class ComplieSystemFacade
{
public:
	string DoComplie(string file_name)
	{
		/*
		编译的过程：扫描符号——预处理——编译——链接
		*/
		Scanner *scanner = new Scanner();
		PreCopeer *precopeer = new PreCopeer();
		Complier *complier = new Complier();
		Linker *linker = new Linker();

		string scan_result = scanner->Scan(file_name);
		precopeer->PreCope(scan_result);
		string complie_result = complier->Complie(scan_result);
		string linke_result = linker->Link(complie_result);
```

```
            delete scanner;
            delete precopeer;
            delete complier;
            delete linker;
            return linke_result;
        }
};


void main()
{
        ComplieSystemFacade *facade = new ComplieSystemFacade();
        //调用Facade提供的方法可以更方便的使用系统
        facade->DoComplie("abc.cpp");
        delete facade;
}
```

Bridge（桥接）

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。

可以这么理解：在 Client 和最终的 Implement 代码之间加入了 Abstract 这个中间层，以消除 Client 和 Implement 之间的依赖关系。



Bridge 模式将抽象与实现分离开来。抽象与实现之间的关系称之为“桥接”，因为它在抽象类与它的实现之间起到了桥梁的作用，使得它们可以独立地变化。

Bridge 模式的优点：

- 分离接口及其实现部分：一个实现未必不变地绑定在一个接口上。抽象类的实现可以在运行时刻配置，一个对象甚至可以在运行时刻改变它的实现。
  将 Abstarction 与 Implementor 分离有助于降低对实现部分编译时刻的依赖性，当改变一个实现类时，并不需要重新编译 Abstraction 类和它的客户程序。为了保证一个类库的不同版本之间的二进制兼容性，一定要有这个性质。
- 提高可扩充性：你可以独立地对 Abstraction 和 Implementor 层次结构进行扩充。
- 实现细节对客户透明：对客户隐藏实现的细节。

相关的模式：

- Abstract Factory 模式可以用来创建和配置一个特定的 Bridge 模式
- Adapter 模式用来帮助无关的类协同工作，它通常在系统设计完成后才会被使用。
  然而，Bridge 模式则是在系统开始时就被使用，它使得抽象接口和实现部分可以独立进行改变。

C++中 Pimpl 手法与 Bridge 模式的联系：

- Pimpl 手法是 Bridge 模式的一种特别的应用
  Pimpl 手法也是为了隐藏实现的细节，分离抽象与实现
- Pimpl 手法确切的说是属于 Bridge 的退化情况，是仅有一个 Implementor 的情况。
- Pimpl 也减少了编译期间的依赖性，使用客户端的代码仅仅依赖于 Abstraction 而不依赖 Implementor。
  如果是类库的话，就维持了不同版本之间的二进制兼容性。

PIMPL 简单示例

```
//File Widget.h
class Widget
{
public:
    Widget(int x, int y);
    ~Widget();                  //这个析构函数必须提供，因为隐式生成的版本会导致使用上的问题
    int getSum() const;
private:
    struct WidgetImpl;
    WidgetImpl* impl;
};

//File Widget.cpp
#include "Widget.h"
struct Widget::WidgetImpl           //在实现文件里进行定义，这样就减少了依赖，隐藏了实现细节
{
    WidgetImpl(int x, int y) : a(x), b(y) {}
    int a;
    int b;
};
```

```
Widget::Widget(int x, int y) : impl(new WidgetImpl(x, y)) {}
Widget::~Widget() { delete impl; impl = 0; } //而且必须在.h文件中定义，在cpp文件中实现。

int Widget::getSum() const {
	return impl->a + impl->b;
}
```

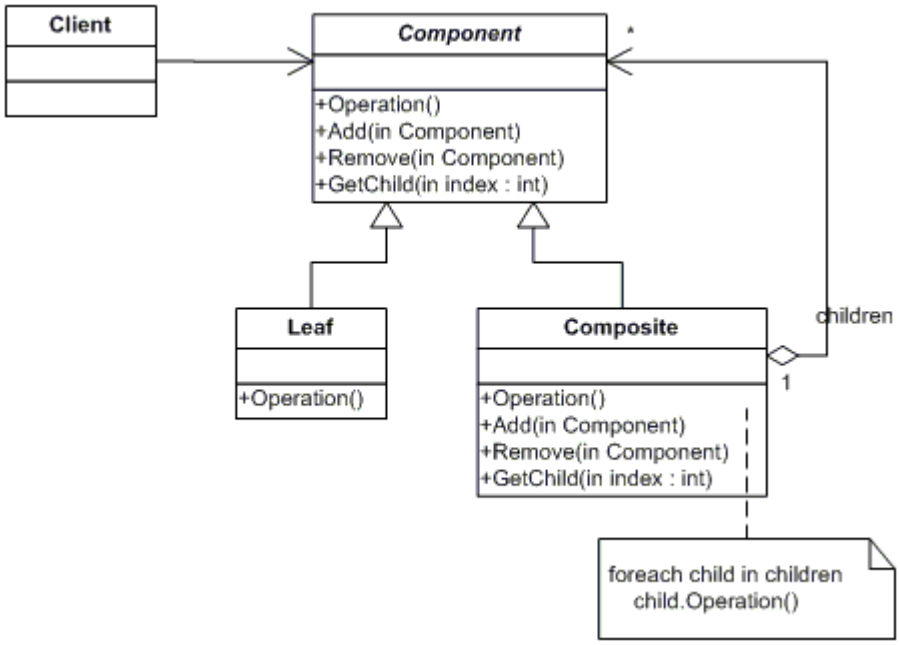
BRIDGE

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

//实现基类
class Implementor
{
public:
	virtual void Operation() = 0;

	virtual ~Implementor(){}
};

//具体的实现类
class ConcreteImplementorA : public Implementor
{
public:
	virtual void Operation()
	{
		cout << "ConcreteImplementorA do Operation" << endl;
	}
};

//抽象
//Abstraction一般为抽象基类，这里为了简便就为具体类
class Abstraction
{
public:
	Abstraction()
	{
		_imp = GetImplementor();
	}
```

```
    virtual ~Abstraction(){}

    virtual void Operation()
    {
        _imp->Operation();   //将对抽象的请求转发给实现
    }

protected:
    virtual Implementor* GetImplementor()      //可以通过继承重写这个方法选择不同的实现者
    {
        return new ConcreteImplementorA();
    }

    Implementor *_imp;
};

//使用的客户
class Client
{
public:
    Client()
    {
        //客户只需要知道Abstraction就可以,不必关心实现(实现了抽象与实现的分离)
        _imp = new Abstraction();
    }

    void Do()
    {
        _imp->Operation();
    }

private:
    Abstraction *_imp;
};


int _tmain(int argc, _TCHAR* argv[])
{
    Client c;
    c.Do();

    return 0;
}
```

Composite（组合）

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



Composite 模式的关键是抽象类 Component ,它即可以表示 Leaf ,又可以表示 Composite。由于 Leaf、Composite 都继承自统一的接口 Component，因此 Composite 可以递归的组合其它 Leaf 或 Composite 对象。由于统一的接口，用户可以一致的对象 Leaf 和 Composite。

Composite 的 children 指针指向其父类 Component 是这种模式的关键所在。

这个组合对象又可被组合，这样不断的递归下去。客户代码中，任何用到基本对象的地方都可以使用组合对象。

容易增加新组合也会产生一些问题，那就是很难限制组合中的组件。有时你希望一个组合只能有某些特定的组合。使用 Composite 时，你不能依赖类型系统施加这些约束，而必须在运行时刻进行检查。

声明管理子部件的操作：(Leaf 不应该操作子部分，Composite 需要操作子部件，这就需要决定在 Component 还是 Composite 来定义管理子部件的操作)

- 在 Component 之中定义具有良好的透明性。代价是安全性，用户可能在 Leaf 上做一些操作子部件的操作。
- 在 Composite 之中定义具有良好的安全性，代价是透明性，需要做很多运行时的类型检查。

在这个模式中，相对于安全性，我们更加强调透明性！

相关模式：

- 通常部件 - 父部件连接用于 Responsibility of Chain 模式
- Decorator 模式经常与 Composite 模式一起使用。当装饰和组合一起使用时，它们通常有一个公共的父类。因些装饰必须支持具有 Add、Remove 和 GetChild 操作的 Composite 接口。
- Flyweight 模式让你共享组件，但不能再引用它们的父部分了（因为共享）。
- Itertor 模式可以用来遍历 Composite。
- Visitor 模式将本来应该分布在 Composite 和 Leaf 类中的操作和行为局部化。

COMPOSITE

```
#include <vector>
#include <string>
#include <iostream>
#include <sstream>
#include <algorithm>
using namespace std;


//组件的抽象接口
//Add, Remove, GetChilds都提供默认的实现.
//Operation为纯虚函数
//这样,无论是叶子结点还是组合结点都可以使用这个接口来进行操作了
class Component
{
public:
	virtual void Operation() = 0;

	virtual void Add(Component *c){}

	virtual void Remove(Component *c){}

	virtual vector<Component*> GetChilds()
	{
		vector<Component*> vec;
		return vec;
	}

	virtual ~Component(){}
};
```

```
//不能包含子结点的叶子结点
//只需要重写必要的函数
//接口中类似于容器的功能函数可以直接使用抽象接口中的默认实现
class Leaf : public Component
{
public:
	Leaf(const string &sign)
		: _sign(sign)
	{}

	virtual void Operation()
	{
		cout << "Leaf " << _sign << " Operation" << endl;
	}

private:
	string _sign;
};


//可以包含子结点的组件
//重写作为一个容器应该提供的功能
class Composite : public Component
{
public:
	Composite(const string &sign)
		: _sign(sign)
	{}

	virtual void Operation()
	{
		cout << "Composite " << _sign << " Operation! And call children to operation" << endl;

		for (vector<Component*>::iterator iter = _childrens.begin(); iter != _childrens.end(); ++iter)
		{
			(*iter)->Operation();
		}
	}

	virtual void Add(Component *c)
	{
		_childrens.push_back(c);
```

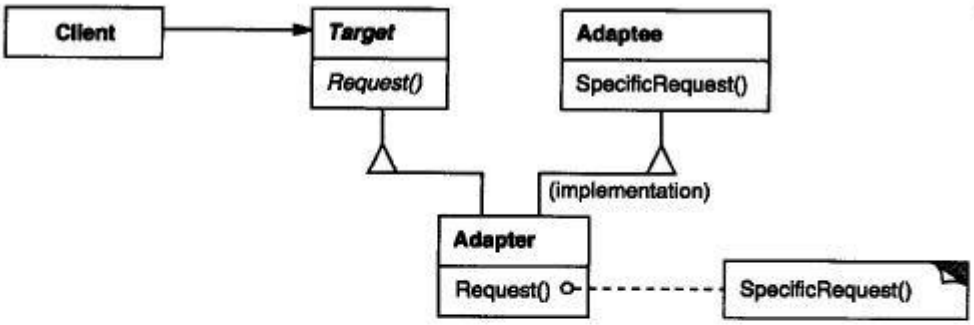
```
        }

        virtual void Remove(Component *c)
        {
            _childrens.erase(remove(_childrens.begin(), _childrens.end(), c));
        }

        virtual vector<Component*> GetChilds()
        {
            return _childrens;
        }

        virtual ~Composite()
        {
            for (vector<Component*>::iterator iter = _childrens.begin(); iter != _childrens.end(); ++iter)
            {
                delete (*iter);
            }
        }

private:
        //有指向父结点的指针是Composite模式的精髓所在,结构型模式多是采用这种继承来实现目的的
        vector<Component*> _childrens;
        string _sign;
};

int _tmain(int argc, _TCHAR* argv[])
{
        Component *leaf1 = new Leaf("a");
        Component *leaf2 = new Leaf("b");
        Component *leaf3 = new Leaf("c");
        Component *leaf4 = new Leaf("d");
        Component *leaf5 = new Leaf("e");

        Component *composite1 = new Composite("A");
        Component *composite2 = new Composite("B");
        Component *composite3 = new Composite("C");
        Component *composite4 = new Composite("D");

        composite1->Add(composite2);
        composite1->Add(composite3);
        composite2->Add(leaf1);
        composite2->Add(leaf2);
        composite3->Add(leaf3);
```

```
    composite3->Add(composite4);
    composite4->Add(leaf4);
    composite4->Add(leaf5);

    composite1->Operation();

    return 0;
}
```

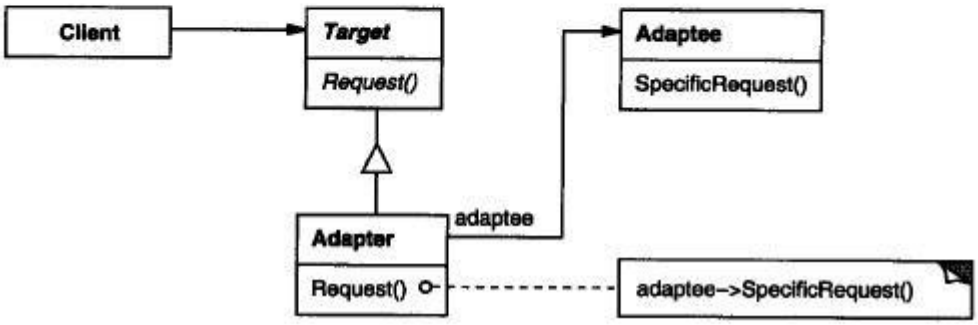
Adapter（适配器）

将一个类的接口转换成客户希望的另外的一个接口（即：用已经存在的 Adaptee 接口去实现 Target 接口）

类适配器使用多重继承对一个接口与另一个接口进行匹配，如下图所示。



对象匹配器依赖于对象组合，如下图所示。



类适配器和对象适配器有着不同的权衡。

类适配器：

- 用一个具体的 Adapter 类对 Adapter 和 Target 进行匹配。结果是当我们想要匹配一个类及所有它的子类时，类 Adapter 将不能胜任工作。
- 使得 Adapter 可以重定义 Adaptee 的部分行为，因为 Adapter 是 Adaptee 的一个子类。

- 仅仅引入了一个对象，并不需要额外的指针以间接得到 Adaptee。

对象适配器则：

- 允许一个 Adapter 与多个 Adaptee - - 即 Adaptee 本身以及它的所有子类同时工作。Adapter 也可以一次给所有的 Adaptee 添加功能。
- 使得重定义 Adaptee 的行为比较困难。这就需要生成 Adaptee 的子类并且使用 Adapter 引用这个子类而不是 Adaptee 本身。

还有一个变形叫作“双向适配器”。双向适配器其它就是用两个 public 继承来同时实现两个接口。

用 C++实现类适配器时：Adapter 类应该采用公共方式继承 Target 类，并且用私有方式继承 Adaptee 类。因此，Adapter 类应该是 Target 的子类型，但不是 Adaptee 的子类型。

相关模式：

- 模式 Bridge 的结构与对象适配器类似，但是 Bridge 模式的出发点不同：Bridge 目的是将接口部分和实现部分分离，从而对它们可以较为容易也相对独立的加以改变。而 Adapter 则意味着改变一个已有对象的接口。（很多时候，不同模式之间的不同点主要就在于它们的目的不同）
- Decorator 模式增强了其它对象的功能而同时又不改变它们的接口。因此 decorator 对应的程序的透明性比适配器要好。结果是 decorator 支持递归组合，而纯粹使用适配器是不可能实现这一点的。
- 模式 Proxy 在不改变它的接口的条件下，为另一个对象定义了一个代理。

ADAPTER

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


namespace ClassAdapter
{
    class Target
    {
    public:
        virtual ~Target(){}
```

```
        virtual void DoA() = 0;

        virtual void DoB() = 0;
    };

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

        void Do1()
        {
            cout << "Do 1" << endl;
        }

        void Do2()
        {
            cout << "Do 2" << endl;
        }
    };

    // 使用已有的Adaptee接口去实现Target接口
    // 注意的是：这里使用public继承Target, private继承Adaptee
    class Adapter : public Target, private Adaptee
    {
    public:
        void DoA()
        {
            Do1();
        }

        void DoB()
        {
            Do2();
        }
    };
}

namespace ObjectAdapter
{
    class Target
    {
    public:
```

```
        virtual ~Target(){}

        virtual void DoA() = 0;

        virtual void DoB() = 0;
    };

    class Adaptee
    {
    public:
        void Do1()
        {
            cout << "Do 1" << endl;
        }

        void Do2()
        {
            cout << "Do 2" << endl;
        }
    };

    //使用已有的Adaptee接口去实现Target接口
    class Adapter : public Target
    {
    public:
        Adapter(Adaptee &e)
            : _adaptee(e)
        {}

        void DoA()
        {
            _adaptee.Do1();
        }

        void DoB()
        {
            _adaptee.Do2();
        }

    private:
        Adaptee &_adaptee;
    };
}
```

```
int _tmain(int argc, _TCHAR* argv[])
{
	{
		ClassAdapter::Target *target = new ClassAdapter::Adapter();
		target->DoA();
		target->DoB();
	}

	cout << endl;

	{
		ObjectAdapter::Adaptee adaptee;
		ObjectAdapter::Target *target = new ObjectAdapter::Adapter(adaptee);
		target->DoA();
		target->DoB();
	}

	return 0;
}
```

行为模式

行为模式的讨论：

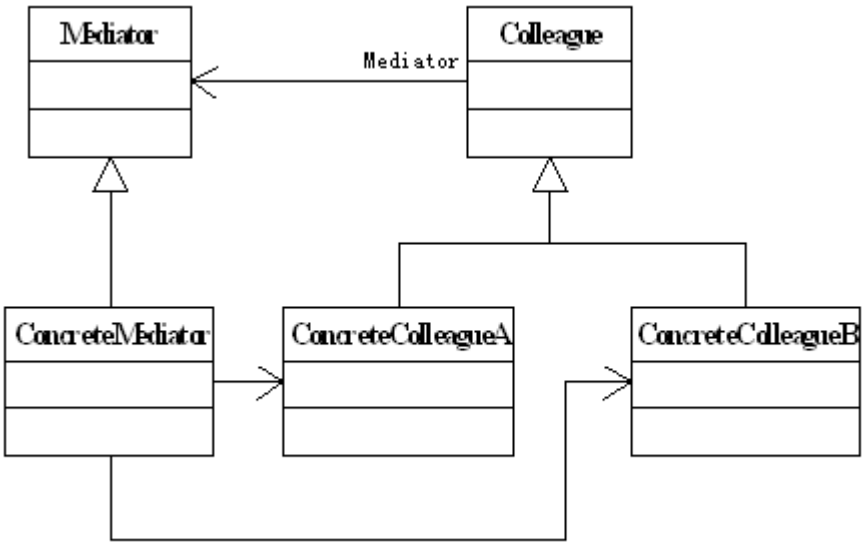
- 封装变化是很多行为模式的主题：当一个程序的某个方面的特征经常发生改变时，这些模式就定义一个封装这个方面的对象。
  - 一个 Strategy 对象封装一个算法
  - 一个 State 对象封装一个与状态相关的行为
  - 一个 Mediator 对象封装对象间的协议
  - 一个 Iterator 对象封装访问和遍历一个聚焦对象中的各个构件的方法
- 一些模式引入总是被用作参数的对象，如 Visitor
- 一些模式定义一些可作为信息到处传递的对象，这些对象在稍后被调用。如 Command 和 Memento
- Mediator 和 Observer 是相互竞争的模式。它们之间的差别是：Observer 通过引入 Observer 和 Subject 对象来分布通信，而 Mediator 对象则封装了其它对象之间的通信与逻辑。
- Mediator 模式的目的是集中而不是分布。生成可复用的 Observer 和 Subject 比生成可复用的 Mediator 更容易一些。Mediator 中的通信流更容易理解，Observer 模式引入的间接性会使得一个系统难以理解
- 观察者模式中的 Subject 和 Observer 接口是为了处理 Subject 的变化而设计的，因此当对象间有数据依赖时，最好用观察者模式来对它们进行解耦
- 中介者模式可以减少一个系统中的子类生成，因为它将通信行为集中到一个类中而不是将其分布在各个子类中。然后特别的分发策略通常会降低类型安全性（职责链模式也一样有这个风险）。
- 除少数例外情况，各个行为设计模式之间是相互补充和相互加强的类型。
- 设计良好的面积对象系统通常有多个模式镶嵌其中，但其设计者却未必使用这些术语进行思考。然后，在模式级别而不是类或对象级别上进行系统组装可以使我们更方便地获取同等的协同性。

一般而言，了解“做什么”要比“为什么”来的容易；而一个模式中的“为什么”就是它要解决的问题。了解一个模式的目的也是重要的，它可以帮助我们选择要使用的模式，也可以帮助我们理解已有系统的设计。

（在设计模式中：模式要解决的问题及目的往往比它所使用的手段更为重要）

Mediator（中介者）

用一个中介对象来封闭一系列的对象交互。中介者使各对象不需要显式地相互引用（甚至它们相互都不知道对方的存在），从而使其耦合松散，而且可以独立地改变它们之间的交互。



通过将集体的行为封装在单独的中介者中。让中介者负责控制和协调一级对象之间的交互，中介者充当一个中介以使组中的对象不再相互显式引用。这些对象仅知道中介者，从而减少了相互连接的数目。

其实这个模式非常地符合我的思维，这是我很喜欢的系统设计方法啊，在体系结构课上使用过很多次的。我把中介者称之为控制模块，用一个控制模块来封装其它模块之间的交互逻辑。其它的模块只与这个控制模块通信，它们相互不知道对方的存在，更不需要相互通信。（中介者可以视为控制者）

与中介者直接通信的可相互称为 Colleague，它们的地位级别是相同的，是同行地位的相互协作的关系。这就像我思维里经常用到的模块的概念。

所有的 Colleague 都知道 Mediator，Mediator 也知道所有的 Colleague。但是 Colleague 相互并不知道对方（这样看来 Colleague 的叫法又有些奇怪了，也许这个名字只是意味着他们在同一个地位等级）。

中介者模式的效果：

- 减少了子类的生成

- 将各个 Colleague 解耦
- 简化了对象协议
- 对对象如何协作进行了抽象，有助于弄清楚系统中的各个对象是如何交互的
- 使得控制集中化：中介者模式将交互的复杂性变为中介者的复杂性。因为中介者封装了协议，它可能变得比任何一个 Colleague 都复杂。这可能使得中介者自身成为一个难于维护的庞然大物

实现的手段：

- 忽略抽象的 Mediator 类：抽象的 Mediator 类只在需要的时候才去定义
- Colleague-Mediator 通信：在 Colleague 感兴趣的事情发生时让 Mediator 去通知 Colleague，这可以用 Observer 模式去实现

相关模式：

- Facade 模式与中介者模式的不同之处在于它是对一个对象子系统进行抽象，从而提供一个更为方便的接口。它的协议是单向的，即 Facade 对象对这个子系统类提出请求，但反之则不行。相反，Mediator 提供了各个 Colleague 对象不支持或不能支持的协作行为，而且协议是多向的。
- Colleague 可使用 Observer 模式与 Mediator 通信。

发现各个设计模式之间最大的不同也是最重要的地方在于 :它们的目的不同 ,要解决的问题不同。其实很多时间用的手段都非常的类似。

MEDIATOR

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

//声明
class Mediator;

class AbstractColleague
{
public:
	//由Colleague通知Mediator
	virtual void ToBeReady() = 0;

	//由Mediator通知Colleague
	virtual void SayName() = 0;

	virtual ~AbstractColleague(){}

protected:
```

```
    AbstractColleague(Mediator * const mediator)
        : _mediator(mediator){}

    Mediator *_mediator;
};

class ColleagueA : public AbstractColleague
{
public:
    ColleagueA(Mediator * const mediator)
        : AbstractColleague(mediator)
    {}

    virtual void ToBeReady();

    virtual void SayName()
    {
        cout << "My name is A" << endl;
    }
};

class ColleagueB : public AbstractColleague
{
public:
    ColleagueB(Mediator * const mediator)
        : AbstractColleague(mediator)
    {}

    virtual void ToBeReady();

    virtual void SayName()
    {
        cout << "My name is B" << endl;
    }
};

class ColleagueC : public AbstractColleague
{
public:
    ColleagueC(Mediator * const mediator)
        : AbstractColleague(mediator)
    {}

    virtual void ToBeReady();
```

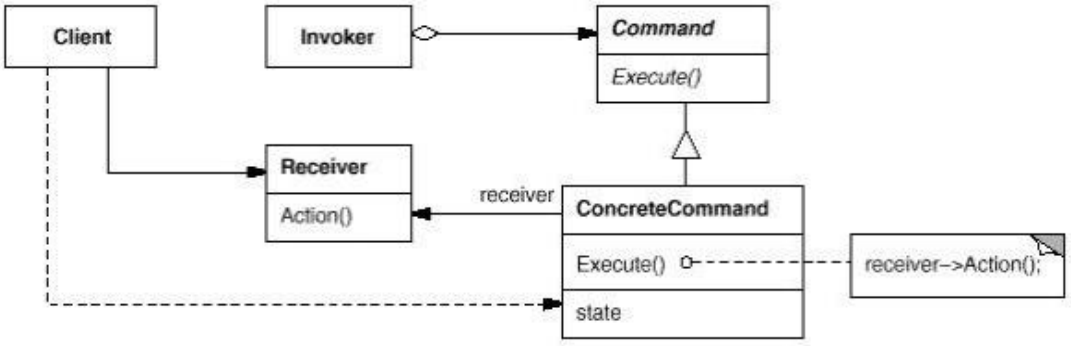
```
    virtual void SayName()
    {
        cout << "My name is C" << endl;
    }
};

class Mediator
{
public:
    static Mediator & Instance()
    {
        static Mediator instance;
        return instance;
    }


    void BeginReadyAndToSayName()
    {
        a->ToBeReady();
    }

    void AReadyNow()
    {
        b->ToBeReady();
    }


    void BReadyNow()
    {
        c->ToBeReady();
    }

    void CReadyNow()
    {
        AllSayName();
    }

private:
    Mediator()
    {
        a = new ColleagueA(this);
        b = new ColleagueB(this);
        c = new ColleagueC(this);
```

```
            _ready_count = 0;
        }

        Mediator(Mediator &);
        Mediator & operator=(Mediator &);

        void AllSayName()
        {
            cout << "Mediator command all to say name" << endl;
            a->SayName();
            b->SayName();
            c->SayName();
        }


        AbstractColleague *a;
        AbstractColleague *b;
        AbstractColleague *c;
        int _ready_count;
};

void ColleagueA::ToBeReady()
{
        cout << "ColleagueA to be ready" << endl;
        _mediator->AReadyNow();
}

void ColleagueB::ToBeReady()
{
        cout << "ColleagueB to be ready" << endl;
        _mediator->BReadyNow();
}

void ColleagueC::ToBeReady()
{
        cout << "ColleagueC to be ready" << endl;
        _mediator->CReadyNow();
}

void main()
{
        //Mediator中的Colleague都相互不知道对方的存在
        //它们之间的交互逻辑被包含在Mediator中了
        //一旦A->B->C的轮流准备,然后在所有的人准备好了之后再轮流报自己的名字
```

```
    //这些逻辑A、B、C都相互不知道（他们甚至不知道对方的存在），它们只做自己的工作
    //只有Mediator知道所有人Colleague的存在，同时所有的Colleague也都知道Mediator的存在
    Mediator::Instance().BeginReadyAndToSayName();
}
```

Command（命令）

将一个请求封装为一个对象，从而使得可以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。



一个命令基本的成员应该包括这个命令所需要的所有参与者，然后通过调用参与者来完成相应的命令。
“参与者”、“执行命令”是一个命令的必须属性。（当然参与者可以为空）

Android 的 intent 不是 Command 模式的应用，intent 的实现应该是有一个 application 级别的东西在管理与协调 intent 发生时应该如何激活相应的 activity。

可以使用 Command+Composite 这两种模式的结合来产生宏命令，这是相当常见和好用的。

Command 模式是回调机制的一个面向对象的替代品。但是好像现在的这些 GUI 框架使用的依然还是回调函数的机制。如.NET 中事件都是+ = 一个函数的回调啊。原因是为什么呢？也许是因为回调高效和简单。

支持取消操作：

- Command 在 execute 之前将参与对象的状态存储起来，在取消该操作时恢复这个状态来消除该操作的影响。
- Command 抽象接口增加一个 Unexecute 操作。
- 支持取消操作是 Command 模式+Memento 模式的一个经典应用。

Command 模式的效果：

- 将调用操作的对象与知道如何实现该操作的对象解耦
- 方便的使用 Composite 模式组合成宏命令
- 增加新的 Command 很容易，因为这无需改变已有的类

Memento 模式可以避免在支持取消操作过程时的错误积累，使得可以完全的恢复参与的对象。使用 Memento 模式可以用空间上的消耗来换取这个精度。

使用 C++的模式来定义一类简单的命令，它们只是去调用另外一个对象的成员函数。

使用 Prototype 模式来实现将已经执行过的命令放置到历史列表中去。

COMMAND

```
#include <vector>
#include <map>
#include <string>
#include <iostream>
#include <sstream>
#include <algorithm>
using namespace std;

/*
定义的参与者,有两个成员函数
*/
class MyClass
{
public:
	void Display1()
	{
		cout << "MyClass.Display1() called" << endl;
	}

	void Display2()
	{
		cout << "MyClass.Display2() called" << endl;
	}
};


/*
一个命令的抽象就应该是这样的，命令最基本的属性其实就只有“执行”
```

```
*/
class Command
{
public:
	virtual ~Command(){}

	virtual void Execute() = 0;
};


/*
这是简单的命令，适用于只需要调用Receiver的一个成员方法的情况
但是这已经是基本完善的：一个命令包括参与者和需要执行的动作两部分。
Receiver代表参与者，Action代表要执行的动作。这两者已经包括了命令的根本属性
*/
template<class ReceiverType>
class SimpleCommand : public Command
{
public:
	typedef void (ReceiverType::* Action)();

	SimpleCommand(ReceiverType &r, Action a)
		: _receiver(r), _action(a)
	{}

	void Execute()
	{
		//.*与->*：成员访问运算符
		(_receiver.*_action)();
	}

private:
	ReceiverType _receiver;
	Action _action;
};


/*
最普通的具体命令
在构造函数中定义好参与者,然后在execute中调用参与者来完成相应的功能
*/
class NormalCommand1 : public Command
{
public:
```

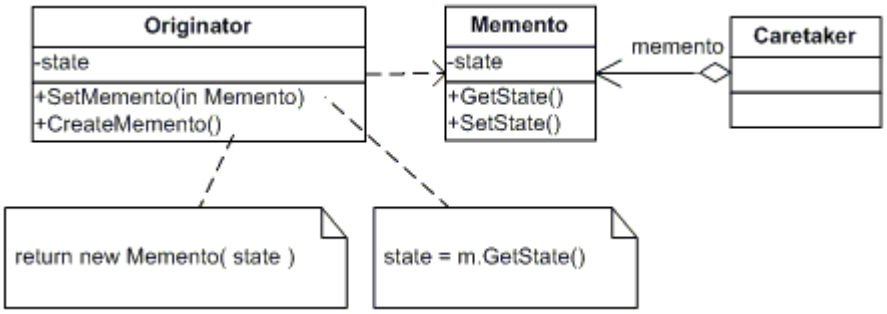
```
        NormalCommand1(MyClass &r)
            : _receiver(r)
        {}

        void Execute()
        {
            _receiver.Display1();
        }

private:
        MyClass _receiver;
};

class NormalCommand2 : public Command
{
public:
        NormalCommand2(MyClass &r)
            : _receiver(r)
        {}

        void Execute()
        {
            _receiver.Display2();
        }

private:
        MyClass _receiver;
};


/*
宏命令：Command模式+Composite模式
*/
class MacroCommand : public Command
{
public:
        MacroCommand(){}

        void Add(Command *c)
        {
            _commands.push_back(c);
        }

        void Remove(Command *c)
```

```
    {
        _commands.erase(remove(_commands.begin(), _commands.end(), c), _commands.end());
    }

    virtual void Execute()
    {
        for (vector<Command *>::iterator iter = _commands.begin(); iter != _commands.end(); ++iter)
        {
            (*iter)->Execute();
        }
    }

private:
    vector<Command *> _commands;            //包括一个指向基类的Command指针的vector是
Composite模式的关键
};

void main()
{
    MyClass m;

    Command *aCommand = new SimpleCommand<MyClass>(m, &MyClass::Display1);
    aCommand->Execute();

    Command *bCommand = new NormalCommand1(m);
    bCommand->Execute();

    Command *cCommand = new NormalCommand2(m);
    cCommand->Execute();

    MacroCommand micro_command;
    micro_command.Add(aCommand);
    micro_command.Add(bCommand);
    micro_command.Add(cCommand);
    micro_command.Execute();
}
```

Memento（备忘录）

在不破坏封装的前提下（如果允许破坏封装，那么备忘录模式的目的太容易就可以达到），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

就可以将该对象恢复到原先保存的状态。



应用要点：记录对象在某个瞬间的内部状态以支持取消操作机制。

备忘录模式存在的原因在于：要有能够恢复对象状态的机制，必须能够将对象的状态信息保存在某处（对象的外部）。但是对象通常封装了其部分或所有的状态信息，使得其状态不能够被其它对象访问，也就不可能在该对象之外保存其状态。而暴露其内部状态又违反了封装的原则，可能有损于应用的可靠性和可扩展性。

一个备忘录是一个对象，它存储另一个对象在某个瞬间的内部状态，而后者称为备忘录的原发器。
只有原发器可以向备忘录中存取信息，备忘录对其它对象“不可见”。（理想状态是只有产生该备忘录的原发器可以访问该备忘录）

参与者：

- Memento(备忘录)
  - 备忘录存储原发器对象的内部状态。原发器根据需要决定备忘录存储原发器的哪些内部状态。
  - 防止原发器以外的其他对象访问备忘录。备忘录实际上有两个接口，管理者（caretaker）只能看到备忘录的窄接口—它只能将备忘录传递给其他对象。相反，原发器能够看到一个宽接口，允许它访问返回到先前状态所需的所有数据（C++中使用友元非常简洁的实现了双接口的设计）。理想的情况是只允许生成本备忘录的那个原发器访问本备忘录的内部状态。
- Originator(原发器)
  - 原发器创建一个备忘录,用以记录当前时刻它的内部状态。
  - 使用备忘录恢复内部状态..
- Caretaker(负责人)
  - 负责保存好备忘录。
  - 不能对备忘录的内容进行操作或检查。

适用性，在以下情况下使用备忘录模式：

- 必须保存一个对象在某一个时刻的(部分)状态，这样以后需要时它才能恢复到先前的状态。
- 如果一个用接口来让其它对象直接得到这些状态，将会暴露对象的实现细节并破坏对象的封装性。

效果：

- 保持封装边界：使用备忘录可以避免暴露一些只应由原发器管理却又必须存储在原发器之外的信息。该模式把可能很复杂的 Originator 内部信息对其他对象屏蔽起来，从而保持了封装边界。
- 它简化了原发器：在其他的保持封装性的设计中，Originator 负责保持客户请求过的内部状态版本。这就把所有存储管理的重任交给了 Originator。让客户管理它们请求的状态将会简化 Originator，并且使得客户工作结束时无需通知原发器。
- 使用备忘录可能代价很高：如果原发器在生成备忘录时必须拷贝并存储大量的信息，或者客户非常频繁地创建备忘录和恢复原发器状态，可能会导致非常大的开销。除非封装和恢复 Originator 状态的开销不大，否则该模式可能并不合适。
- 定义窄接口和宽接口：在一些语言中可能难以保证只有原发器可访问备忘录的状态。
- 维护备忘录的潜在代价：管理器负责删除它所维护的备忘录。然而，管理器不知道备忘录中有多少个状态。因此当存储备忘录时，一个本来很小的管理器，可能会产生大量的存储开销。

相关模式

- Command：命令可使用备忘录来为可撤消的操作维护状态
- Iterator：如前所述备忘录可用于迭代

MEMENTO

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

//实际用来保存游戏状态数据的类
//也可以考虑把这个类处理成GameMemento一样的私有构造函数+friend class GameMemento;
class GameState
{
public:
	GameState(const string &user_name, int step, int score)
		: _user_name(user_name), _step(step), _score(score)
	{}

	string UserName()
	{
		return _user_name;
	}

	int Step()
	{
		return _step;
	}

	int Score()
	{
		return _score;
```

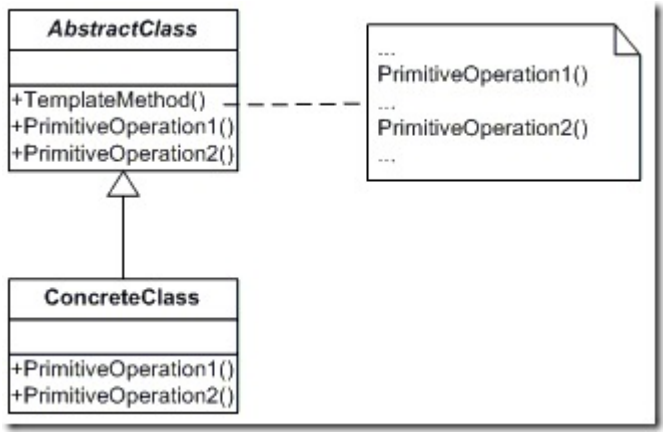
```
    }

private:
    string _user_name;
    int _step;
    int _score;
};


//Game对应的Memento对象
//也可以考虑将GameState所保存的数据保存在这个类中,这样的话类的结构简单,但是可重用性不如分成两个类
//在现在写的这种分成两个类的结构中,GameState类仅仅充当一个保存数据的作用
class GameMemento
{
    //用友元实现对Game的宽接口和对其它类的窄接口
    friend class Game;

public:
    ~GameMemento()
    {
        if (_state != NULL)
        {
            delete _state;
            _state = NULL;
        }
    }

private:
    //只能被Game调用的构造函数
    GameMemento(const string &user_name, int step, int score)
    {
        _state = new GameState(user_name, step, score);
    }

    GameState * GetState()
    {
        return _state;
    }

private:
    GameState *_state;
};
```

```
//某种游戏
class Game
{
public:
	//创建一个全新的游戏
	Game()
		: _user_name("Tan Chuan Qi")
		, _step(6)
		, _score(60)
	{}

	//使用备忘录来恢复一个游戏
	Game(GameMemento *memento)
		: _user_name(memento->GetState()->UserName())
		, _step(memento->GetState()->Step())
		, _score(memento->GetState()->Score())
	{}

	//得到当前对象的备忘录
	GameMemento * GetMemento()
	{
		GameMemento *memento = new GameMemento(_user_name, _step, _score);
		return memento;
	}

	void Score(int score)
	{
		_score = score;
	}

	int Score()
	{
		return _score;
	}

	string UserName()
	{
		return _user_name;
	}

	int Step()
	{
		return _step;
```

```
    }

private:
    string _user_name;
    int _step;
    int _score;
};


void main()
{
    Game g1;                                    //初始化一个全新的游戏
    cout << "g1 : " << g1.Score() << endl;//输出游戏的状态

    g1.Score(100);                              //游戏的状态被改变了
    cout << "g1 : " << g1.Score() << endl;//输出游戏的状态

    GameMemento *m = g1.GetMemento();           //用一个备忘录对象保存游戏的状态
    Game g2(m);                                     //用这个备忘录去恢复一个新游戏
    cout << "g2 : " << g2.Score() << endl;//显示新游戏的状态，可以看到，原来游戏中被改变的状态在新游戏中被保存下来了

    delete m;
}
```

Template Method（模板方法）

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。Template Method 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步骤。



一个模板方法用一些抽象的操作定义一个算法，而子类将重定义这些操作以提供具体的行为。

一次性实现一个算法的不变部分，并将可变的行为留给子类来实现。

模板方法与 C++中的 NVI（非虚接口）非常的相像。

实现：

- 使用 C++访问控制 :在 C++中 ,一个模板方法调用的原语操作可以被定义为保护成员。这保证了它们只能被模板方法访问。而模板方法本身不应该被重定义，所以应该实现为非虚成员函数。
- 尽量减少原语操作
- 命名约定：可以给应该被重定义的那些操作加一个命名前缀以区分它们。如 DoSomething

实现要点：

- Template Method 模式是一种非常基础性的设计模式，在面向对象系统中有着大量的应用。它用最简洁的机制(虚函数的多态性)为很多应用程序框架提供了灵活的扩展点，是代码复用方面的基本实现结构。
- 除了可以灵活应对子步骤的变化外，“不用调用我，让我来调用你”的反向控制结构是 Template Method 的典型应用。
- 在具体实现方面，被 Template Method 调用的虚方法可以具有实现，也可以没有任何实现（抽象方法，纯虚方法），但一般推荐将它们设置为 protected 方法。[李建忠]

适用性：

- 一次性实现一个算法的不变的部分，并将可变的行为留给子类来实现。
- 各子类中公共的行为应被提取出来并集中到一个公共父类中以避免代码重复。这是 Opdyke 和 Johnson 所描述过的“重分解以一般化”的一个很好的例子。首先识别现有代码中的不同之处，并且将不同之处分离为新的操作。最后，用一个调用这些新的操作的模板方法来替换这些不同的代码。
- 控制子类扩展。模板方法只在特定点调用“Hook”操作，这样就只允许在这些点进行扩展。

Template Method 模式是非常简单的一种设计模式，但它却是代码复用的一项基本的技术，在类库中尤其重要。

相关模式：

- Factory Method 模式常被模板方法调用。这么理解：模板方法（Template Method 是骨架）是调用多个 工厂方法（Factory Method 是具体的步骤）的组合。
- Strategy :模板方法使用继承来改变算法的一部分。Strategy 使用委托来改变整个算法。

TEMPLATE METHOD

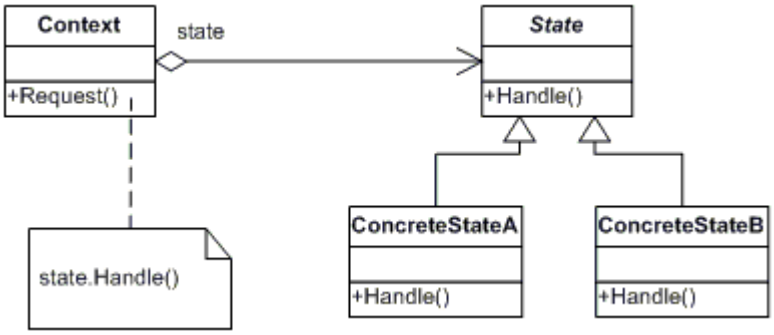
```
#include <iostream>
using namespace std;


class AbstractGenerator
{
public:
	//这就是一个Template Method，它定义了一个算法的框架
	void Display()
	{
		//“模板方法”就是调用了多个“工厂方法”的算法框架
		//其实这是一个非常非常常见且易于理解的方案，平时编程无意中都用过太多次
		if (CalcuteFileSize() > 100)
		{
			DisplayBigFile();
		}
		else
		{
			DisplaySmallFile();
		}
	}


protected: //工厂方法应该是受保护的

	//工厂方法
	virtual int CalcuteFileSize() = 0;

	//工厂方法
	virtual void DisplayBigFile()
	{
		cout << "This is a big file" << endl;
	}

	//工厂方法
	virtual void DisplaySmallFile()
	{
		cout << "This is a small file" << endl;
	}
```

```
};

class ConcreteGenerator : public AbstractGenerator
{
public:
	ConcreteGenerator(int s)
		: _size(s){}

	virtual int CalcuteFileSize()
	{
		//read a file
		//return this file's size
		return _size;
	}

	virtual void DisplayBigFile()
	{
		cout << "Oh~~~    a big file" << endl;
	}

private:
	int _size;
};


void main()
{
	AbstractGenerator *g1 = new ConcreteGenerator(20);
	g1->Display();

	AbstractGenerator *g2 = new ConcreteGenerator(200);
	g2->Display();
}
```

State（状态）

允许一个对象在其内部状态改变时改变它的行为（那些会随着状态改变的行为）。

对象看起来似乎修改了它的类。



应用要点：将所有与一个特定状态相关的行为都放入一个对象中。
Context 把所有与状态相关的行为都转发给 State 对象来处理。

适用性：

- 一个对象的行为取决于它的状态，并且它必须在运行时刻根据状态改变它的行为。
- 一个操作中含有庞大的多分支的条件语句，且这些分支依赖于该对象的状态。（以前我都是用大量的分支结构来，这个模式可以优美地消除大量的分支结构，我非常喜欢啊！）
  这个状态通常用一个或多个枚举常量表示。通常，有多个操作包含这一相同的条件结构。State 模式将每一个条件分支放入一个独立的类中。这使得你可以根据对象自身的情况将对象的状态作为一个对象，这一对象可以不依赖于其他对象而独立变化。

协作：

- Context 将与状态相关的请求委托给当前的 ConcreteState 对象处理。
- Context 可将自身作为一个参数传递给处理该请求的状态对象。这使得状态对象在必要时可访问 Context。
- Context 是客户使用的主要接口。客户可用状态对象来配置一个 Context，一旦一个 Context 配置完毕，它的客户不再需要直接与状态对象打交道。
- Context 或 ConcreteState 子类都可决定哪个状态是另外哪一个的后继者，以及是在种条件下进行状态转换。

效果：

- 它将与特定状态相关的行为局部化，并且将不同状态的行为分割开来：State 模式将所有与一个特定的状态相关的行为都放入一个对象中。因为所有与状态相关的代码都存在于某一个 State 子类中，所以通过定义新的子类可以很容易的增加新的状态和转换。
  另一个方法是使用数据值定义内部状态并且让 Context 操作来显式地检查这些数据。但这样将会使整个 Context 的实现中遍布看起来很相似的条件语句或 case 语句。增加一个新的状态可能需要改变若干个操作，这就使得维护变得复杂了。（这就是我以前用的方法，State 模式优美的解决了这个问题）
  State 模式避免了这个问题，但可能会引入另一个问题，因为该模式将不同状态的行为分布在多个 State 子类中。这就增加了子类的数目，相对于单个类的实现来说不够紧凑。但是如果有许多状态时这样的分布实际上更好一些，否则需要使用巨大的条件语句。

正如很长的过程一样，巨大的条件语句是不受欢迎的。它们形成一大整块并且使得代码不够清晰，这又使得它们难以修改和扩展。State 模式提供了一个更好的方法来组织与特定状态相关的代码。决定状态转移的逻辑不在单块的 if 或 switch 语句中，而是分布在 State 子类之间。(以编译器自动实现的多态来代替手工的分支结构 ,这真是相当的棒！) 将每一个状态转换和动作封装到一个类中，就把着眼点从执行状态提高到整个对象的状态。这将使代码结构化并使其意图更加清晰。

- 它使得状态转换显式化：当一个对象仅以内部数据值来定义当前状态时，其状态仅表现为对一些变量的赋值，这不够明确。为不同的状态引入独立的对象使得转换变得更加明确。而且, State 对象可保证 Context 不会发生内部状态不一致的情况 , 因为从 Context 的角度看 ,状态转换是原子的— 只需重新绑定一个变量(即 Context 的 State 对象变量) , 而无需为多个变量赋值。
- State 对象可被共享 : 如果 State 对象没有实例变量— 即它们表示的状态完全以它们的类型来编码—那么各 Context 对象可以共享一个 State 对象。当状态以这种方式被共享时，它们必然是没有内部状态，只有行为的轻量级对象(参见 Flyweight)

实现：

- 谁定义转换：两种方案，
  （ 1 ）在 Context 是定义逻辑
  （ 2 ）由 State 的子类自身指定它们的后继状态以及何时进行转换，通常更灵活合适。但是这需要 Context 增加一个接口，让 State 对象显式地设定 Context 的当前状态。
  （ void Context::SetState(State *s); ）
- State 对象是动态创建的，涉及到创建后的销毁问题，使用智能指针（Proxy 模式）可以解决。

相关模式：

- Flyweight 模式解释了何时以及怎么共享状态对象
- 状态对象通常是 Singleton

STATE

```
#include <vector>
#include <list>
#include <map>
#include <string>
#include <iostream>
#include <sstream>
#include <memory>
#include <algorithm>
#include <ctime>
using namespace std;
```

```
class Game;

class AbstractState
{
public:
	virtual void Display() = 0;

protected:
	AbstractState(Game *g)
		: _game(g)
	{}

	Game *_game;
};

class WaitingStartState : public AbstractState
{
public:
	WaitingStartState(Game *g)
		: AbstractState(g)
	{}

	virtual void Display();
};

class PlayingState : public AbstractState
{
public:
	PlayingState(Game *g)
		: AbstractState(g)
	{}

	virtual void Display();
};

class Game
{
public:
	Game();

	void SetState(AbstractState *s);

	void Display();
```

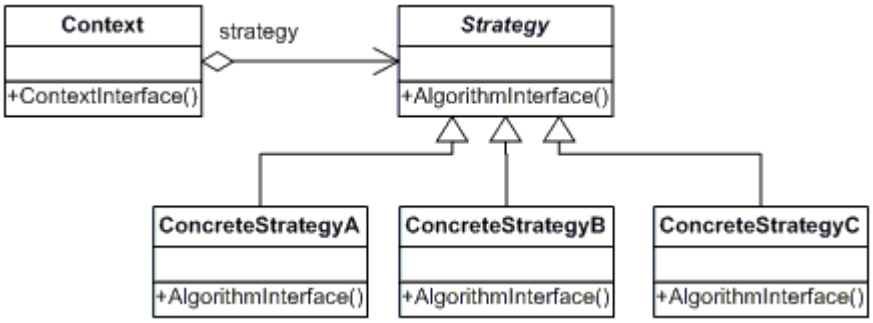
```
    ~Game();

private:
    AbstractState *_state;
};


Game::Game()
{
    _state = new WaitingStartState(this);
}

Game::~Game()
{
    delete _state;
}

void Game::SetState(AbstractState *s)
{
    if (s != _state)
    {
        delete _state;
        _state = s;
    }
}

void Game::Display()
{
    //将与状态相关的操作转发给 状态对象 处理
    _state->Display();
}

void WaitingStartState::Display()
{
    cout << "Waiting to start!" << endl;
    _game->SetState(new PlayingState(_game));
}

void PlayingState::Display()
{
    for (int i = 0; i < 10; ++i)
    {
        cout << "Playing step " << i << endl;
    }
```

```
}

void main ()
{
	Game g;
	g.Display();
	g.Display();
}
```

Strategy（策略）

定义一系列的算法，把他们一个个封装起来，并使他们可以互相替换。

本模式使得算法可以独立于使用它们的客户。



直用要点：用类封装不同的算法而不是将算法直接硬编码到代码中。

看上去就是一个选择的功能，但是这跟 if 有着本质上的区别，这是使用“编译器的多态”代替“手工的分支结构”的又一个好手段。Strategy 和 State 都是在想尽各种方法来消除以往的这种条件分支结构。

效果：

- 可以方便地使用一系列的相关算法
- Strategy 是一个好的替代继承的方法。可以通过继承 Context 类也达到这个目的，但是远不如 Strategy 优美。
- 消除条件语句

缺点：

- Strategy 和 Context 之间的通信开销
- 增加了对象的数目：Strategy 增加了一个应用中的对象的数目。有时你可以将 Strategy 实现为可供各 Context 共享的无状态对象来减少这一开销。任何其余的状态都由

Context 来维护。Context 在每一次对 Strategy 对象的请求中都将这个状态传递过去。共享的 Strategy 不应该在各次调用之间维护状态。

实现：

- 定义 Strategy 与 Context 的接口：Strategy 和 Context 的接口必须使得 ConcreteStrategy 能够有效地访问它所需要的 Context 中的数据，反之亦然。有两种方法：
  - 让 Context 将数据放在参数中传递给 Strategy。这使得 Strategy 与 Context 解耦，但是 Context 可能发送很多不需要的数据，影响效率
  - 让 Context 将自身作为一个参数传递给 Strategy，Strategy 再显式地向 Context 请求所需要的内容
- 将 Strategy 作为模板参数 :在 C++中可以利用模式机制用一个 Strategy 类来配置一个 Context。然后这仅在以下两点满足时才可用：在编译时选择 Strategy + 不需要在运行时改变 Strategy。

  template <typename AStrategy>
  class Context
  {
  public:
   void Operation(){_strategy.DoA();}
  private:
   AStrategy _strategy;
  };

  使用模板不再需要定义给 Strategy 定义接口的抽象类。把 Strategy 作为一个模板参数也使得可以将一个 Strategy 与 Context 静态的绑定在一起，从而提高效率。
- Strategy 可以定义为可选的：示例代码中就是这么做的，这么做的好处在于客户根本不需要了解 Strategy 对象，除非它们不喜欢默认的行为，需要进行定制。

相关模式：Strategy 对象经常是很好的轻量级对象（Flyweight）。

STRATEGY

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

class AbstractStrategy
{
public:
	virtual void ShowWelcome(const string &game_name) const = 0;

	virtual ~AbstractStrategy(){}
};
```

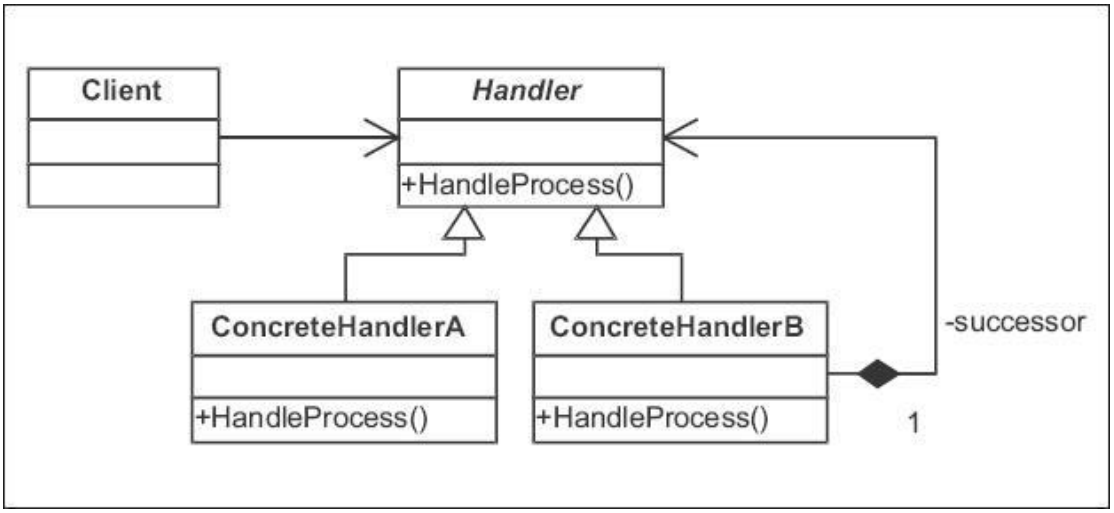
```
class SimpleStrategy : public AbstractStrategy
{
public:
	virtual void ShowWelcome(const string &game_name) const
	{
		cout << endl << "Welcom to " << game_name << endl << endl;
	}
};


class MagnificentStrategy : public AbstractStrategy
{
public:
	virtual void ShowWelcome(const string &game_name) const
	{
		cout << "------------------------------------------------" << endl;
		cout << "\t" << game_name << endl;
		cout << endl;
		cout << "                    Welcome to this game !                    " << endl;
		cout << endl;
		cout << "------------------------------------------------" << endl;
	}
};



class Game
{
public:
	Game(string game_name)
		: _game_name(game_name)
	{
		_strategy = new SimpleStrategy();	//默认的策略
	}

	void SetShowWelcomeStrategy(AbstractStrategy *s)
	{
		delete _strategy;
		_strategy = s;
	}

	void ShowWelcome()
```

```
    {
        //转发给策略对象
        _strategy->ShowWelcome(_game_name);
    }

    ~Game()
    {
        if (_strategy != NULL)
        {
            delete _strategy;
            _strategy = NULL;
        }
    }

private:
    string _game_name;
    AbstractStrategy *_strategy;
};


void main()
{
    Game g("Game:Test");
    g.ShowWelcome();          //使用默认的策略
    g.SetShowWelcomeStrategy(new MagnificentStrategy());    //更改策略
    g.ShowWelcome();          //使用新的策略
}
```

Chain Of Responsibility（职责链）

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收者之间的耦合关系。

将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。



这个模式的想法是：给多个对象处理一个请求的机会，从而解耦发送者和接受者。该请求沿着职责链传递，直到有一个对象处理它。

职责链的应用应该非常的广，比如消息的机制，还有很多 GUI 框架的事件传递机制（如：WPF 的路由事件）。

消息机制中：消息沿着一定的路径进行传递，路径上的所有对象都有机会进行处理或者转发给下一个对象。

事件的传递也是，用户在某个 Panel 中按下鼠标，引发 Mouse_Press 事件，如果 Panel 没有处理这个事件，就会自动转发给 Panel 的上层窗口比如 MainForm 进行处理。

Chain Of Responsibility 中的接收者是隐式的，由一个职责链来代替具体的接收者。消息的发送者并不知道链上的哪个对象会处理请求。

适用性：

- 有多个对象可以处理一个请求，哪个对象处理该请求由运行时刻自动确定。
- 你想在不明确指定接收者的情况下，向多个对象中的一个提交一个请求。
- 可处理一个请求的对象应该被动态的指定。

效果：

- 降低耦合度：接收者和发送者都没有对方的明确信息，且链中的对象不需要知道链的结构。它们仅需要保持一个指向后继者的引用，而不需要保持所有候选接受者的信息。
- 增强了给对象指派职责时的灵活性：可以灵活的地运行时动态的决定谁来处理请求。
- 不保证被接受：如果没有人处理，消息可以被抛弃。

相关模式：

- 职责链常与 Composite 模式一起使用。这种情况下，一个构件的父构造可以作为它的后继

Chain Of Responsibility

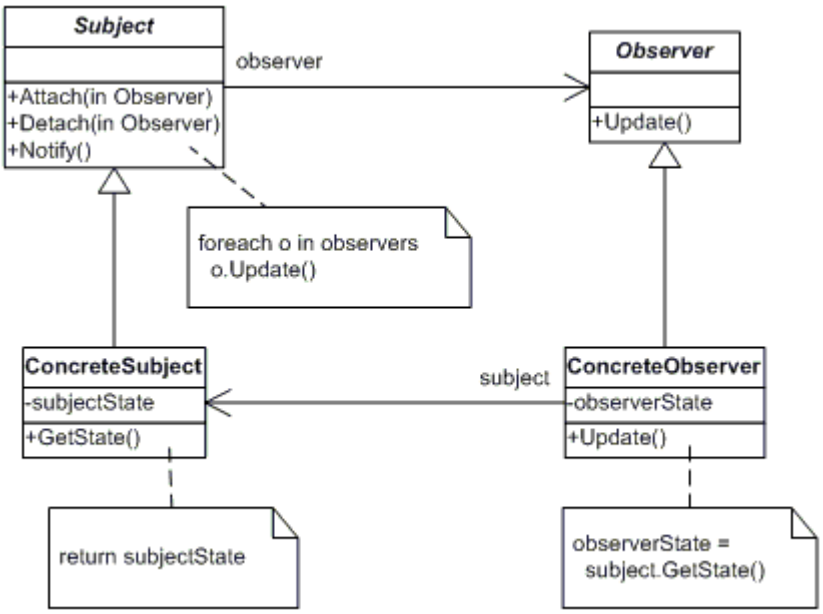
```
#include <vector>
#include <map>
#include <string>
#include <iostream>
#include <sstream>
#include <algorithm>
using namespace std;

//处理者的抽象基类
class Handler
{
public:
	virtual void HandleRequester(string request_id)
	{
		//基类提供默认的处理,如果chain还有下一个处理者就转发,否则提示无人处理
		if (_successor)
		{
			_successor->HandleRequester(request_id);
		}
		else
		{
			cout << "Nobody cope request " << request_id << endl;
		}
	}

	virtual ~Handler(){}

protected:
	Handler(Handler *p = 0)
		: _successor(p)
	{}

	Handler *_successor; //职责链上的下一个处理者
};

//具体处理者1
class ConcreteHandler1 : public Handler
{
public:
```

```
    ConcreteHandler1(string hander_name, Handler *h = 0)
        : Handler(h)
        , _hander_name(hander_name)
    {}

    virtual void HandleRequester(string request_id)
    {
        //如果能处理就处理,不能处理就调用基类提供的默认行为
        if (request_id == "request0" || request_id == "request1")
        {
            cout << "ConcreteHandler1 \"" << _hander_name << "\" cope " << request_id << endl;
        }
        else
        {
            Handler::HandleRequester(request_id);
        }
    }

private:
    string _hander_name;
};

//具体处理者2
class ConcreteHandler2 : public Handler
{
public:
    ConcreteHandler2(string hander_name, Handler *h = 0)
        : Handler(h)
        , _hander_name(hander_name)
    {}

    virtual void HandleRequester(string request_id)
    {
        //如果能处理就处理,不能处理就调用基类提供的默认行为
        if (request_id == "request3" || request_id == "request4")
        {
            cout << "ConcreteHandler2 \"" << _hander_name << "\" cope " << request_id << endl;
        }
        else
        {
            Handler::HandleRequester(request_id);
        }
    }
```

```
private:
	string _hander_name;
};

void main()
{
	//ConcreteHandler之间的关系就是典型的Compsite关系
	//它们之间的Composite关系可以很方便的利用在Chain of Responsibility上

	//构造职责链
	ConcreteHandler1 a("a");//事实上a永远没有机会处理消息，因为Chain上的c会把所有需要a来处理的消息都处理掉并停止传递
	ConcreteHandler2 b("b", &a);
	ConcreteHandler1 c("c", &b);
	Handler *chain_of_responsibility = &c;


	for (int i = 0; i < 5; ++i)
	{
		stringstream ss;
		ss << "request" << i;
		//在职责链上进行传递
		chain_of_responsibility->HandleRequester(ss.str());
	}
}
```

Observer（观察者）

定义对象间的一种一对多的依赖关系，当一个对象的状态发生改变时，所以依赖于它的对象都得到通知并被自动更新。



各个语言中常用的手段：

- Java：使用接口来实现
- C++：使用回调函数指针来实现
- C#：则使用委托比较多
- 总结：使用接口的方式比较 OO，而且在任何语言中都可用，不像委托只能用在 C#中、函数指针只能用在 C++中，但是它的缺点就是实现的起来比较麻烦

一个目标可能有任意数目的依赖它的观察者。一旦目标的状态发生改变，所有的观察者都得到通知。作为对这个通知的响应，每个观察者都将查询目标以使其状态与目标的状态同步。
观察者可能需要保存目标的状态以了解目标到底发生了什么变化，也可以不保存而直接根据目标的当前状态更新自己。

适用性：

- 当一个抽象模型有两个方面，其中一个方面依赖于另一方面。将这二者封装在独立的对象中以使它们可以各自独立地改变和复用
- 当一个对象改变需要同时改变其它对象，而不知道具体有多少对象有待改变
- 当一个对象必须通知其它对象，而它又不能假定其它对象是谁

如何得到目标的当前状态：

- 由 Update 把目标的当前状态传递给观察者（也可以定义更新协议，告诉观察者改变了什么而不是仅仅告诉它们状态改变了）
- 由观察者在得到更新的消息后主动向目标请求

在发出通知前确保目标的状态自身是一致的：这点看起来很容易保证，但是在支持多态的机制中

却很容易出错。比如：

```
void MySubject::Operation(int newValue){
    BaseClassSubject::Operation(newValue);  //由于基类定义了这这里就触发了观察者的 update 操作。

    _myInstVar += newValue;    //实际上的更新在这里，在观察者收到通知时真正的更新还没有发生。
}
```

解决办法是：基类在其调用更新操作之前调用一个虚方法，子在可以重写这个虚方法，在这个虚方法中去进行实际的更新操作。

更新协议（推/拉模型）：

- 推模型：目标向观察者发送关于改变的详细信息，而不管它们需不需要
- 拉模型：目标除了最小通知外什么也不送出，而是在此之后由观察者显式地向目标询问细节
- 拉模型强调是目标不知道它的观察者，而推模型假定目标知道一些观察者需要的信息
- 推模型可能使得观察都相对难以复用，因为目标对观察者进行的假设可能并不正确
- 拉模型可能效率比推模型差

可以使用 aspect 的概念来注册对不同事情感兴趣的观察者。

```
void Subject::Attach(Aspect &interest);
void Subject::Notify(Aspect &interest){
    foreach( o in observers){
        if (o.Aspect() == interest) o.Update();
    }
}
```

当目标和观察者之间的关系特别复杂时，可以维护一个更改管理器，Subject 和 Observers 都只与 ChangeManger 进行交互。ChangeManager 是一个 Mediator 模式的实例。通常也只有一个 ChangeManager 的 Singleton。

OBSERVER

```
#include <vector>
#include <list>
#include <map>
#include <string>
#include <iostream>
#include <sstream>
#include <memory>
#include <algorithm>
```

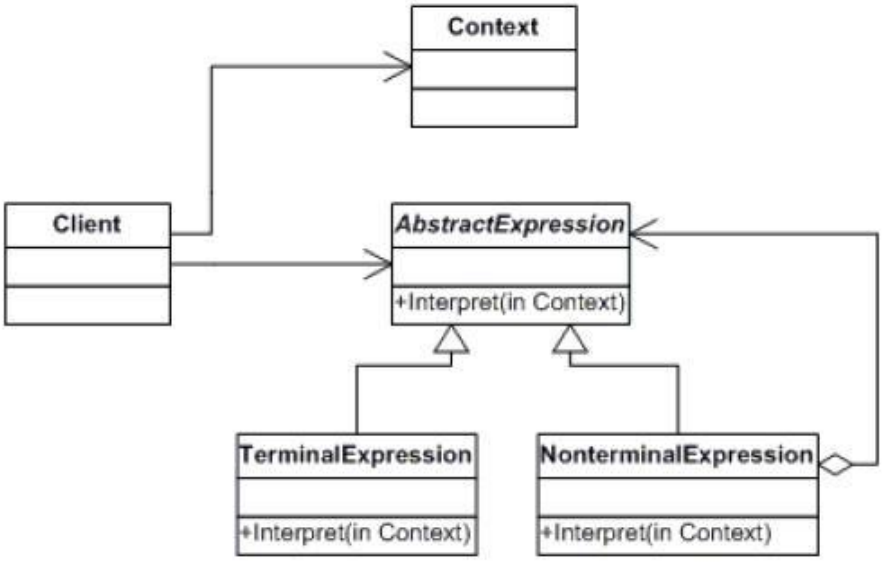
```
#include <ctime>
using namespace std;

//观察者的抽象基类
class AbstractObserver
{
public:
	virtual ~AbstractObserver(){}

	// Update操作时把观察者需要的信息一起发送过去了.属于“推模型”
	// 因为所有的改变的详细信息（这里只有一个时间t）都通过参数被推出了
	virtual void Update(const time_t &t) = 0;
};

class OriginStyleObserver : public AbstractObserver
{
public:
	virtual void Update(const time_t &t)
	{
		cout << "Origin Style: " << t << endl;
	}

};

class NewStyleObserver : public AbstractObserver
{
public:
	virtual void Update(const time_t &t)
	{
		struct tm * timeinfo;
		timeinfo = localtime ( &t );

		cout << "New Style: " << asctime (timeinfo);
	}
};


class Timer
{
public:
	void Attach(AbstractObserver *o)
	{
		_observers.push_back(o);
	}
```

```
    void Detach(AbstractObserver *o)
    {
        _observers.erase(remove(_observers.begin(), _observers.end(), o));
    }

    void Start()
    {
        time_t begin_time = clock();

        while(true)
        {
            time_t end_time = clock();

            if (end_time - begin_time >= 1000)
            {
                begin_time = end_time;
                Trick(time(NULL));
            }
        }
    }


private:
    /// 通知所有的注册观察者有事情发生
    void Trick(time_t t)
    {
        for (vector<AbstractObserver *>::iterator iter = _observers.begin(); iter != _observers.end();
++iter)
        {
            (*iter)->Update(t);
        }
    }


    vector<AbstractObserver *> _observers;
};


void main ()
{
    Timer timer;

    //注册了2个观察者
```

```
        timer.Attach(new OriginStyleObserver());
        timer.Attach(new NewStyleObserver());

        //开始计时
        timer.Start();
}
```

Interpreter（解释器）

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。



解释器模式使用“类”来表示每一条文法。(示例中的 :VariableExpression、AndExpression、OrExpression、 NotExpression) 它们都属于"NonterminalExpression"

突然觉得这种解释器模式和 “递归” 很象啊 !各个文法结点调用子结点的方法进行组合来生成本结点的 Evaluate 值。

Interpreter 和 Composite 模式在实现上有许多相通的地方。(的确可以这样来想 : Interpreter 只是组合方式比较特殊的 Composite)。

语法树的建立过程 : 解释器模式并未涉及如何创建一棵语法树。换言之，它不涉及语法分析。抽象语法树由其它的过程来生成。

这里需要改变对” 解释语言 “的固有观念 :我们可以把解释一种语言想象的更宽泛一些，这样的

话解释语言这种操作就可以解决很多问题。很多问题的解决都可以用宽泛的解释语言的概念去解决。
在最宽泛的概念下，几乎每一个使用复合模式的系统也都使用了解释器模式。

相关模式：

- Composite 模式：抽象语法树是一个复合模式的实例
- Flyweight 模式：可以在抽象语法树中共识终结符
- Iterator 模式：解释器可以使用迭代器遍历语法树
- Visitor 模式：可以用来在一个类中维护抽象语法树中的各节点的行为，使用 Visitor 模式来实现解释的操作。

INTERPRETER

```
#include <vector>
#include <map>
#include <string>
#include <iostream>
#include <sstream>
#include <algorithm>
using namespace std;


//上下文环境
class Context
{
public:
	void Add(const string &variable, bool value)
	{
		//利用了map的特点：如果[]运算符找不存在的元素则会自动添加。
		_vars[variable] = value;
	}

	bool Lookup(const string &variable)
	{
		if (_vars.count(variable) != 0)
		{
			return _vars[variable];
		}

		throw exception("cannot find this int context");
	}
```

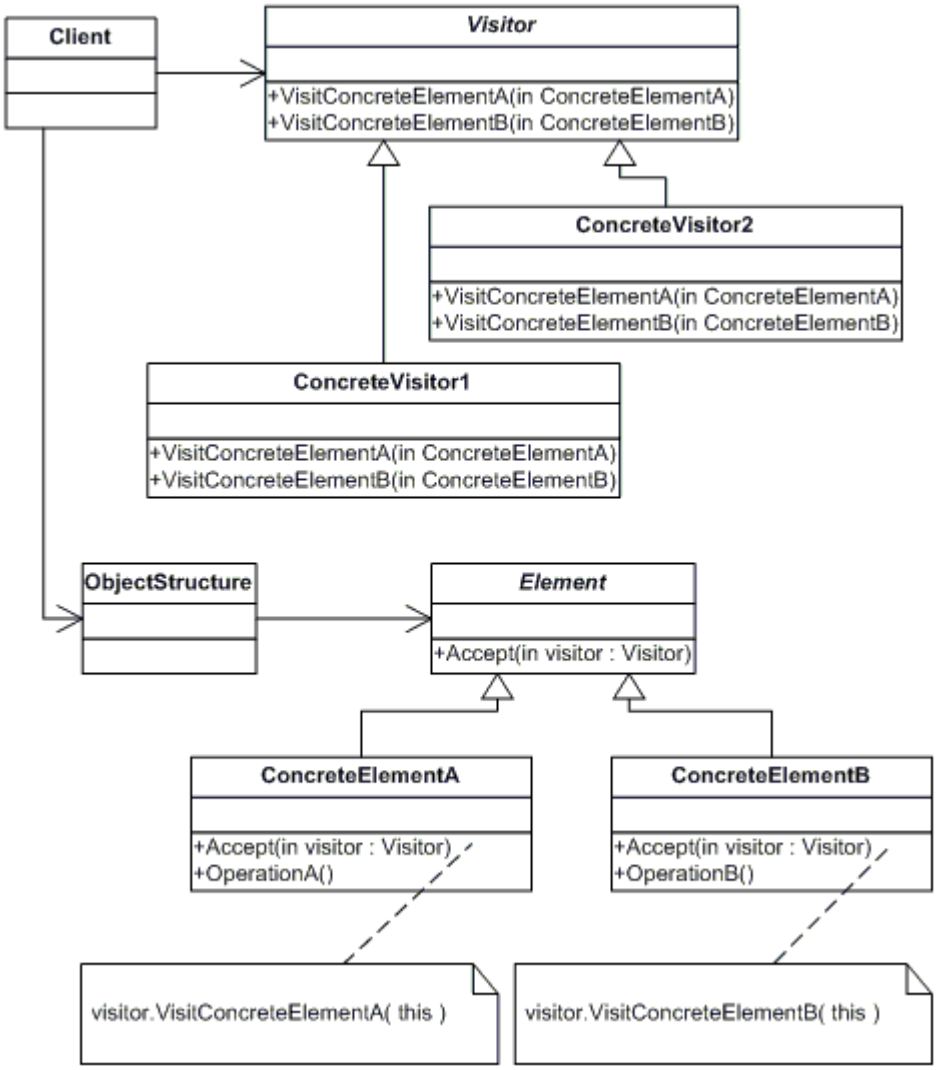
```
private:
    map<string, bool> _vars;
};


//表达式的抽象基类
class AbstractExpression
{
public:
    virtual ~AbstractExpression(){}

    virtual bool Evaluate(Context &c) = 0;
};


//变量表达式
class VariableExpression : public AbstractExpression
{
public:
    VariableExpression(const string &s)
        : _var(s)
    {}

    virtual bool Evaluate(Context &c)
    {
        return c.Lookup(_var);
    }

private:
    string _var;
};


//与 表达式
class AndExpression : public AbstractExpression
{
public:
    AndExpression(AbstractExpression *left, AbstractExpression *right)
        : _left(left), _right(right)
    {}

    virtual bool Evaluate(Context &c)
    {
        return _left->Evaluate(c) && _right->Evaluate(c);
```

```
    }

private:
    AbstractExpression *_left;
    AbstractExpression *_right;
};


//或 表达式
class OrExpression : public AbstractExpression
{
public:
    OrExpression(AbstractExpression *left, AbstractExpression *right)
        : _left(left), _right(right)
    {}

    virtual bool Evaluate(Context &c)
    {
        return _left->Evaluate(c) || _right->Evaluate(c);
    }

private:
    AbstractExpression *_left;
    AbstractExpression *_right;
};


//非 表达式
class NotExpression : public AbstractExpression
{
public:
    NotExpression(AbstractExpression *origin)
        : _origin(origin)
    {}

    virtual bool Evaluate(Context &c)
    {
        return !_origin->Evaluate(c);
    }

private:
    AbstractExpression *_origin;
};
```

```
void main()
{
	AbstractExpression *finial;
	Context c;
	c.Add("x", false);
	c.Add("y", true);
	c.Add("z", true);

	AbstractExpression *x = new VariableExpression("x");		//false
	finial = x;
	cout << (finial->Evaluate(c) ? "true" : "false") << endl;

	AbstractExpression *y = new VariableExpression("y");		//true
	finial = y;
	cout << (finial->Evaluate(c) ? "true" : "false") << endl;

	AbstractExpression *z = new VariableExpression("z");	//true
	finial = z;
	cout << (finial->Evaluate(c) ? "true" : "false") << endl;

	AbstractExpression *a1 = new AndExpression(x, y);		//(false && true) = false
	finial = a1;
	cout << (finial->Evaluate(c) ? "true" : "false") << endl;

	AbstractExpression *a2 = new OrExpression(a1, z);		//((false && true) || true) = true
	finial = a2;
	cout << (finial->Evaluate(c) ? "true" : "false") << endl;

	AbstractExpression *a3 = new NotExpression(a2);		//!((false && true) || true) = false
	finial = a3;
	cout << (finial->Evaluate(c) ? "true" : "false") << endl;
}
```

Visitor（访问者）

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。



经典的 Visitor 模式中重要一点是 :Visitor 接口的方法个数与 Element 层次的子类的数目相同。
所以这个模式适用于 Element 层次不怎么变动，而 Visitor 层次可以进行变动的情况。
需要对不能修改的已有的库进行添加方法时就是访问者模式最佳的应用时机。

访问者模式使得我们可以在传统的单分派语言（如 Smalltalk、Java 和 C++）中模拟双分派技术。对于支持多分派的语言（如 CLOS），访问者模式已经内置于语言特性之中了，从而不再重要。
由于要在 C++中实现 双分派 ，所以 Visitor 接口就需要知道 Element 层次中所有的类。

当一个元素接受访问者访问时，该元素就向访问者发送一个包含自身类信息的请求。该请求同时也将该元素本身作为一个参数。然后访问将为该元素执行该操作 - - 这一操作以前是在该元素的类中的。

适用性：

- 一个对象结构包含很多类对象。它们有不同的接口，而你想对这些对象实施一些依赖于其具体类的操作（知道这个应用目的很重要，因为如果不必依赖于具体类的操作，就可

以使用对普通的抽象基类来实现，没必要使用访问者模式。但是这种依赖于具体类的操作是非常的常见的，基本都是这样，所以访问者模式也就很必要了）。

- 需要对一个对象结构中的对象进行很多不同的并且不相关的操作，而你想避免让这些操作“污染”（Visitor 之间不会相互干扰，也不会污染 Element）这些对象的类。Visitor 使得你可以将相关的操作集中起来定义在一个类中。
  当该对象结构被很多应用共享时，用 Visitor 模式让每个应用仅包含需要用到的操作。
- 定义对象结构的类很少改变，但经常需要在此结构上定义新的操作。改变对象结构类需要重定义对所有访问者的接口，这可能需要很大的代价。如果对象结构类经常改变，那么可能还是在这些类中定义这些操作较好。
  因为 Element 的层次发生了变化，如增加或者减少了一个 ConcreteElement，就必须修改 Visitor 层次中的所有的类的接口以适应 Element 层次的改变。代价太大了！

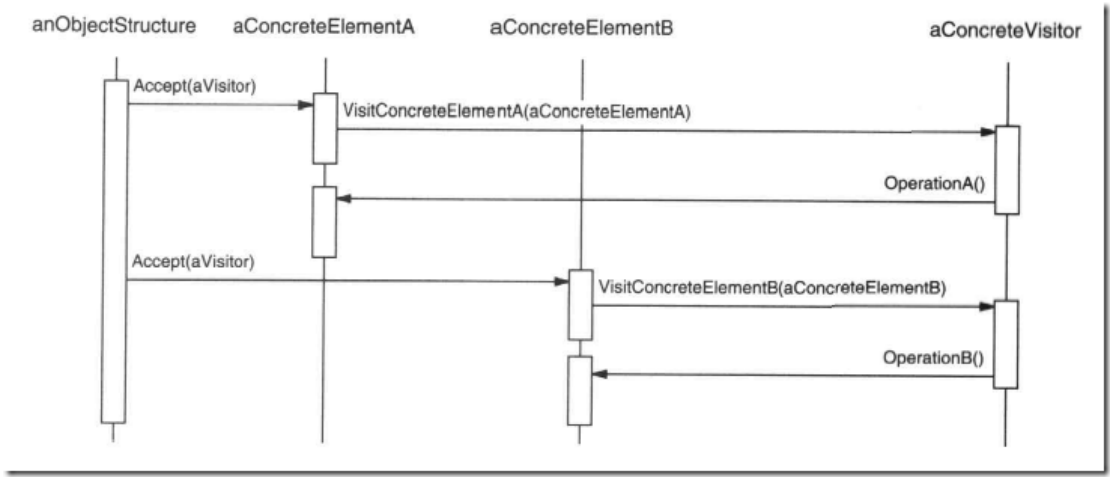
参与者：

- Visitor（访问者，如 NodeVisitor）
  - 为该对象结构中 ConcreteElement 的每一个类声明一个 Visitort 操作。该操作的名字和特征标识了发送 Visit 请求给该访问者的那个类。这使得访问者可以确定正被访问元素的具体的类。这样访问者就可以通过该元素的特定接口直接访问它。
- ConcreteVisitor（具体访问者，如 TypeCheckingVisitor）
  - 实现每个由 Visitor 声明的操作。每个操作实现本算法的一部分，而该算法片断乃是对应于结构中对象的类。ConcreteVisitor 为该算法提供了上下文并存储它的局部状态。这一状态常常在遍历该结构的过程中累积结果。
- Element（元素，如 Node）
  - 定义一个 Accept 操作，它以一个访问者为参数。
- ConcreteElement（具体元素，如 AssignmentNode，VariableRefNode）
  - 实现 Accept 操作，该操作以一个访问者为参数。
- ObjectStructure（对象结构，如 Program）
  - **能枚举它的元素。**
  - 可以提供一个高层的接口以允许该访问者访问它的元素。
  - 可以是一个复合（参见 Composite）或是一个集合，如一个列表或一个无序集合。

协作：

- • 一个使用 Visitor 模式的客户必须创建一个 ConcreteVisitor 对象，然后遍历该对象结构，并用该访问者访问每一个元素。
- • 当一个元素被访问时，它调用对应于它的类的 Visitor 操作。如果必要，该元素将自身作为这个操作的一个参数以便该访问者访问它的状态。

下面的交互框图说明了一个对象结构、一个访问者和两个元素之间的协作。



优缺点：

- 访问者模式使得易于增加新的操作访问者使得增加依赖于复杂对象结构的构件的操作变得容易了。仅需增加一个新的访问者即可在一个对象结构上定义一个新的操作。相反，如果每个功能都分散在多个类之上的话，定义新的操作时必须修改每一类。
- 访问者集中相关的操作而分离无关的操作相关的行为不是分布在定义该对象结构的各个类上，而是集中在一个访问者中。无关行为却被分别放在它们各自的访问者子类中。
- 就既简化了这些元素的类，也简化了在这些访问者中定义的算法。所有与它的算法相关的数据结构都可以被隐藏在访问者中。
- 增加新的ConcreteElement类很困难Visitor模式使得难以增加新的Element的子类。每添加一个新的 ConcreteElement 都要在 Vistor 中添加一个新的抽象操作，并在每一个 ConcretVisitor 类中实现相应的操作。有时可以在 Visitor 中提供一个缺省的实现，这一实现可以被大多数的 ConcreteVisitor 继承，但这与其说是一个规律还不如说是一种例外。

  所以在应用访问者模式时考虑关键的问题是系统的哪个部分会经常变化，是作用于对象结构上的算法呢还是构成该结构的各个对象的类。如果老是有新的 ConcretElement 类加入进来的话，Visitor 类层次将变得难以维护。在这种情况下，直接在构成该结构的类中定义这些操作可能更容易一些。如果 Element 类层次是稳定的，而你不断地增加操作获修改算法，访问者模式可以帮助你管理这些改动。
- 通过类层次进行访问一个迭代器（参见 Iterator）可以通过调用节点对象的特定操作来遍历整个对象结构，同时访问这些对象。但是迭代器不能对具有不同元素类型的对象结构进行操作。
- 累积状态当访问者访问对象结构中的每一个元素时，它可能会累积状态。如果没有访问者，这一状态将作为额外的参数传递给进行遍历的操作，或者定义为全局变量。
- 破坏封装访问者方法假定 ConcreteElement 接口的功能足够强，足以让访问者进行它们的工作。结果是，该模式常常迫使你提供访问元素内部状态的公共操作，这可能会破坏它的封装性。

Visitor 模式中最关键的技术就是双分派技术（C++中默认支持的是单分派，在 C++中普通的方

法调用就是一个单分派）：
双分派意味着得到执行的操作决定于请求的种各类和**两个**接收者的类型。Accept 是一个 double-dispatch 操作。它的含义取决于两个类型：Visitor 类型和 Element 类型。双分派使得访问者可以对每一个类的元素请求不同的操作。
这是 Visitor 模式的关键所在：得到执行的操作不仅决定于 Visitor 类型还决定于它访问的 Element 类型。
双分派的实质就是由两个类的类型来决定具体调用的代码，没什么神奇的！

谁负责遍历：

- 通常由对象结构负责迭代
- 另一个解决方案是使用一个迭代器来访问各个元素
- 甚至可以将遍历算法放在访问者中，但这样需要在每一个 ConcoreteElement 中实现遍历算法。这样做的原因一般在于要实现非常复杂的遍历算法，它依赖于该对象结构的操作结果。

讨论：
有时候我们需要为 Element 提供更多的修改，这样我们就可以通过为 Element 提供一系列的 Visitor 模式可以使得 Element 在不修改自己的同时增加新的操作，但是这也带来了至少以下的两个显著问题：

- 破坏了封装性。Visitor 模式要求 Visitor 可以从外部修改 Element 对象的状态，这一般通过两个方式来实现：a）Element 提供足够的 public 接口，使得 Visitor 可以通过调用这些接口达到修改 Element 状态的目的；b）Element 暴露更多的细节给 Visitor，或者让 Element 提供 public 的实现给 Visitor（当然也给了系统中其他的对象），或者将 Visitor 声明为 Element 的 friend 类，仅将细节暴露给 Visitor。但是无论那种情况，特别是后者都将是破坏了封装性原则（实际上就是 C++的 friend 机制得到了很多的面向对象专家的诟病）。
- ConcreteElement 的扩展很困难：每增加一个 Element 的子类，就要修改 Visitor 的接口，使得可以提供给这个新增加的子类的访问机制。从上面我们可以看到，或者增加一个用于处理新增类的 Visit（ ）接口，或者重载一个处理新增类的 Visit（ ）操作，或者要修改 RTTI 方式实现的 Visit（ ）实现。无论那种方式都给扩展新的 Element 子类带来了困难。

相关模式：

- Composite：访问者可以用于对一个由 Composite 模式定义的对象结构进行操作。
- Interpreter：访问者可以用于解释。

VISITOR.H

```
#pragma once

//头文件中不应该包含其它的类的头文件，因为头文件中仅仅有只有包括指向这两个类的指针
//指针的大小是固定的，所以只需要一个普通的声明语句就OK了
//所以对于需要用到的类，仅仅只要声明一个这个类型就可以了
class ConcreteElement1;
class ConcreteElement2;

class AbstractVisitor
{
public:
	virtual ~AbstractVisitor(){}

	// 这种针对继承层次中每一个类留一个接口就可以利用各个子类的特定接口，不一定非要使用抽象基类的接口，可行性更好
	virtual void VisitConcreteElement1(ConcreteElement1 * const){}
	virtual void VisitConcreteElement2(ConcreteElement2 * const){}

protected:
	AbstractVisitor(){}
};

class ConcreteVisitor1 : public AbstractVisitor
{
public:
	virtual void VisitConcreteElement1(ConcreteElement1 * const e);

	virtual void VisitConcreteElement2(ConcreteElement2 * const e);
};

class ConcreteVisitor2 : public AbstractVisitor
{
public:
	virtual void VisitConcreteElement1(ConcreteElement1 * const e);

	virtual void VisitConcreteElement2(ConcreteElement2 * const e);
};
```

VISITOR.CPP

```
#include <iostream>
#include "Element.h"
#include "Visitor.h"
using namespace std;

//ConcreteVistor1相当于将Elements的一个SimpleShowName操作提取到了类的外部
```

```
void ConcreteVisitor1::VisitConcreteElement1(ConcreteElement1 * const e)
{
	cout << e->Name() << endl;
}

void ConcreteVisitor1::VisitConcreteElement2(ConcreteElement2 * const e)
{
	cout << e->Name() << endl;
}


//ConcreteVistor2相当于将Elements的一个ComplexShowName操作提取到了类的外部
void ConcreteVisitor2::VisitConcreteElement1(ConcreteElement1 * const e)
{
	cout << "This is a ConcreteElement1 : " << e->Name() << endl;
}

void ConcreteVisitor2::VisitConcreteElement2(ConcreteElement2 * const e)
{
	cout << "This is a ConcreteElement1 : " << e->Name() << ", " << e->Describe() << endl;
}
```

ELEMENT.H

```
#pragma once

#include <string>
using namespace std;

class AbstractVisitor;

class AbstractElement
{
public:
	virtual ~AbstractElement(){}
	virtual void Accpet(AbstractVisitor *v) = 0;
};

class ConcreteElement1 : public AbstractElement
{
public:
	ConcreteElement1(const string &name)
		: _name(name)
	{}

	virtual void Accpet(AbstractVisitor *v);
```

```
        string Name(){  return _name; }

private:
        string _name;
};

class ConcreteElement2 : public AbstractElement
{
public:
        ConcreteElement2(const string &name, const string &describe)
                : _name(name), _describe(describe)
        {}

        virtual void Accpet(AbstractVisitor *v);

        string Name(){  return _name; }

        string Describe(){      return _describe; }

private:
        string _name;
        string _describe;
};
```

ELEMENT.CPP

```
#include "Visitor.h"
#include "Element.h"

void ConcreteElement1::Accpet(AbstractVisitor *v)
{
        //双分派：
        // 1－Accept是一个虚函数，在main中调用“(*iter)->Accpet(visitor1);”时完成一次分派
        // 2－这里由于v是一个指针且VisitConcreteElement1是虚函数，于是再借由v可以指向AbstractVisitor继承层次再实现了一次分派
        //
        // 注：    也可以不使用双分派，AbstractVisitor只有一个接口VisitElemtn(AbstractVisitor *);
        //         但是这样的话就太浪费了，因为在ConcreteElemtnes的Accept方法中可以区分出来访问的是那一个Element
        //         仅仅使用抽象基类的接口相对于使用各个子类的特定接口来讲就丢失了一定的信息
        //         这种双分派虽然在编码上比较麻烦，需要Visitor知道Elements整个继承层次，但是往往由于可以
        //         利用各个子类的特殊接口，在自由度上更大，可行性也更好！
        v->VisitConcreteElement1(this);
}
```

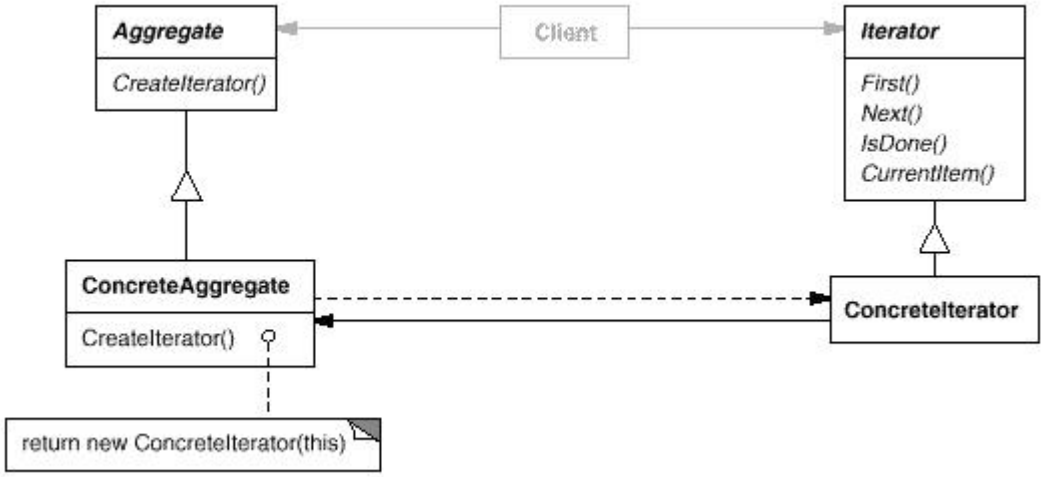
```
void ConcreteElement2::Accpet(AbstractVisitor *v)
{
	v->VisitConcreteElement2(this);
}
```

MAIN

```
#include <vector>
#include "Element.h"
#include "Visitor.h"

void main()
{
	//构造对象结构（一般来说访问者用于组合型的对象结构较多，这里为了简单仅仅用了数组型的对象结构）
	vector<AbstractElement *> object_structre;
	object_structre.push_back(new ConcreteElement1("product1"));
	object_structre.push_back(new ConcreteElement2("product2", "Good"));
	object_structre.push_back(new ConcreteElement1("product3"));
	object_structre.push_back(new ConcreteElement1("product4"));
	object_structre.push_back(new ConcreteElement2("product5", "Bad"));

	//定义访问者
	AbstractVisitor *visitor1 = new ConcreteVisitor1();		//可以任意的添加访问者，而对原来的对象结构不造成影响
	AbstractVisitor *visitor2 = new ConcreteVisitor1();		//可以任意的添加访问者，而对原来的对象结构不造成影响
	AbstractVisitor *visitor3 = new ConcreteVisitor2();		//可以任意的添加访问者，而对原来的对象结构不造成影响
	AbstractVisitor *visitor4 = new ConcreteVisitor2();		//可以任意的添加访问者，而对原来的对象结构不造成影响
	AbstractVisitor *visitor5 = new ConcreteVisitor2();		//可以任意的添加访问者，而对原来的对象结构不造成影响

	//由对象结构来负责遍历迭代
	for (vector<AbstractElement *>::iterator iter = object_structre.begin(); iter != object_structre.end(); ++iter)
	{
		(*iter)->Accpet(visitor1);
		(*iter)->Accpet(visitor2);
		(*iter)->Accpet(visitor3);
		(*iter)->Accpet(visitor4);
		(*iter)->Accpet(visitor5);
	}
}
```

Iterator（迭代器）

提供一种方法访问一个聚合对象中各个元素，而又不需要暴露该对象的内部表示。



在上图中，CreateIterator 方法连接了集合类层次与迭代器层次。

迭代器负责跟踪当前的元素，即它知道哪些元素已经被遍历过了，并能够计算出待遍历的后继对象。

这种将遍历机制与列表对象分离使我们可以定义不同的迭代器来实现不同的遍历策略，而无需在列表接口中列举它们。

通过每一个迭代器来保持它自己的状态，从而解耦了集合的表示与迭代的过程，使得可以同时进行多个不同的遍历。（因为由各个迭代器自己维护当前的迭代状态）

该控制迭代：

- 外部迭代器：当由客户来控制迭代的过程时，该迭代器称为一个外部迭代器。使用外部迭代器需要客户主动推进遍历的步伐，显式地向迭代器请求下一个元素。外部迭代器更符合我们同常对迭代器的感觉，常见的普通迭代器都属于外部迭代器（如 vector<int>::iteartor 等）
- 内部迭代器：当由迭代器自己来控制及完成所有的迭代过程时，该迭代器称为一个内部迭代器。使用内部迭代器时只需要向其提交一个待执行的操作，迭代器内部负责对每一个元素执行该操作。内部迭代器与平时的 foreach 很象。

该定义遍历算法：

- 一般的迭代器都是由迭代器自己定义遍历算法，但是这样的话需要将集合的内部结构暴露给迭代器，可能破坏了封装性（但是由于迭代器和集合是属于一个整体的，也谈不上什么破坏封装）。
- 由集合自己定义遍历算法称为“游标”，这样迭代器的功能就仅仅是保存当前的游标位置信息。

迭代器的健壮性：在进行遍历的过程之中更改集合应该造成所有的迭代器失效，在实现迭代器时需要对这点进行检测。在示例中我实现了普通的、非”线程安全“的检测方法。总之，集合在修改之后应该让该集合的所有迭代器都知道它们都失效了。

在示例中 Collection 层次的类都将 Iterator 层次中相应的类都声明为了友元，以使用迭代器可以对集合进行特权访问。但是这样的特权访问可能使定义新的遍历变得很难，因为它将要求改变该聚合的接口增加另一个友元。为避免这个问题，迭代器可包含一些 protected 操作来访问聚合类的重要的非公共可见成员。迭代器子类（且只有迭代器子类）可以使用这些 protected 操作来得到对该聚合的特权访问。

空迭代器：一个空迭代器是一个退化的迭代器，它有助于处理边界条件。根据定义，一个 NullIterator 总是已经完成遍历的，即它的 IsDone 操作总返回 true
空迭代器的一个常见应用在于树型的聚合结构。在遍历过程中，非叶子结点返回它所有的直接子结点的集合的迭代器，而叶子结点则返回一个空迭代器。这样就可以使用一个统一的访问实现在整个结构上的遍历。

相关模式：

- Composite 模式：迭代器常被应用到象复合这样的递归结构上
- Factory Method 模式：多态迭代器靠 Factory Method 来实例化适当的迭代器子类，即示例中的 CreateIterator 工厂方法
- Memento：常与迭代器模式一起使用，迭代器可使用一个 Memento 来捕获一个迭代的状态。迭代器在其内部使用 Memento

INTERATOR

```
#include <vector>
#include <list>
#include <map>
#include <string>
#include <iostream>
#include <sstream>
#include <memory>
#include <algorithm>
#include <ctime>
using namespace std;
```

```
//提前声明
template<typename ItemType> class VectorCollection;
template<typename ItemType> class ListCollection;


//迭代器基类,完成迭代所需的接口
template<typename ItemType>
class AbstractIterator
{
public:
	virtual void First() = 0;
	virtual void Next() = 0;
	virtual bool IsDone() = 0;
	virtual ItemType& CurrentItem() = 0;
};


//用以迭代vector的迭代器
template<typename ItemType>
class VectorIterator : public AbstractIterator<ItemType>
{
public:
	VectorIterator(VectorCollection<ItemType> * const collection)
		: _collection(collection), _collection_time(collection->_collection_altered_time)
	{}

	void First()
	{
		_collection_time = _collection->_collection_altered_time;   //重新迭代时应该更新记录时间
		_postion = 0;
	}

	void Next()
	{
		CheckValid();
		++_postion;
	}

	bool IsDone()
	{
		CheckValid();
		return _postion >= _collection->_collection.size();
```

```
    }

    ItemType& CurrentItem()
    {
        CheckValid();

        //迭代器超过了末尾
        if (IsDone())
        {
            throw new exception("out of range");
        }

        return _collection->_collection[_postion];
    }

private:
    //迭代器做任何操作之前都应该检查一下迭代器是否有效
    //这种记录时间的方法不是"线程安全的",但是对单线程的情况是适用且有效的
    void CheckValid()
    {
        //如果当前的集合的最后修改时间与记录的不符,表示集合被修改,迭代器失效,迭代应该立
即中止
        if (_collection->_collection_altered_time != _collection_time)
        {
            throw new exception("collection has been modify, iterator are invalid");
        }
    }

private:
    size_t _postion;
    VectorCollection<ItemType> *_collection;
    time_t _collection_time;          //用于记录开始迭代时集合的最后修改时间
};


//用以迭代list的迭代器
template<typename ItemType>
class ListIterator : public AbstractIterator<ItemType>
{
public:
    ListIterator(ListCollection<ItemType> * const collection)
        : _collection(collection), _collection_time(collection->_collection_altered_time)
    {}
```

```
    void First()
    {
        _collection_time = _collection->_collection_altered_time;   //重新迭代时应该更新记录时间
        _postion = 0;
    }

    void Next()
    {
        CheckValid();
        ++_postion;
    }

    bool IsDone()
    {
        CheckValid();
        return _postion >= _collection->_list.size();
    }

    ItemType& CurrentItem()
    {
        CheckValid();

        if (IsDone())
        {
            throw new exception("out of range");
        }

        list<ItemType>::iterator iter = _collection->_list.begin();
        for (size_t pos = 0; pos < _postion && iter != _collection->_list.end(); ++pos)
        {
            ++iter;
        }

        return *iter;
    }

private:
    //迭代器做任何操作之前都应该检查一下迭代器是否有效
    //这种记录时间的方法不是"线程安全的",但是对单线程的情况是适用且有效的
    void CheckValid()
    {
        //如果当前的集合的最后修改时间与记录的不符,表示集合被修改,迭代器失效,迭代应该立即中止
        if (_collection->_collection_altered_time != _collection_time)
```

```
            {
                throw new exception("collection has been modify, iterator are invalid");
            }
        }

private:
        ListCollection<ItemType> *_collection;
        size_t _postion;
        time_t _collection_time;            //用于记录开始迭代时集合的最后修改时间
};


//集合的抽象基类
template<typename ItemType>
class AbstractCollection
{
public:
        virtual void Add(ItemType) = 0;

        //用以创建相应的迭代器的Factory Method方法
        virtual auto_ptr< AbstractIterator<ItemType> > CreateIterator() = 0;
};


//以vector为基础的集合类
template<typename ItemType>
class VectorCollection : public AbstractCollection<ItemType>
{
public:
        friend class VectorIterator<ItemType>;              //声明迭代器为友元

        VectorCollection()
            : _collection_altered_time(time(NULL))
        {}

        void Add(ItemType item)
        {
            //在做任何更新集合的操作(如添加,修改,删除)时,都必须更新集合的最后修改时间
            _collection_altered_time = time(NULL);

            _collection.push_back(item);
        }

        virtual auto_ptr< AbstractIterator<ItemType> > CreateIterator()
```

```
    {
        auto_ptr< AbstractIterator<ItemType> > ptr(new VectorIterator<ItemType>(this));

        return ptr;
    }

private:
    vector<ItemType> _collection;
    time_t _collection_altered_time;        //标识该集合的最后修改时间
};


//以list为基础的集合类
template<typename ItemType>
class ListCollection : public AbstractCollection<ItemType>
{
public:
    friend class ListIterator<ItemType>;          //声明迭代器为友元

    ListCollection()
        : _collection_altered_time(time(NULL))
    {}

    void Add(ItemType item)
    {
        //在做任何更新集合的操作(如添加,修改,删除)时,都必须更新集合的最后修改时间
        _collection_altered_time = time(NULL);

        _list.push_back(item);
    }

    virtual auto_ptr< AbstractIterator<ItemType> > CreateIterator()
    {
        auto_ptr< AbstractIterator<ItemType> > ptr(new ListIterator<ItemType>(this));

        return ptr;
    }

private:
    list<ItemType> _list;
    time_t _collection_altered_time;        //标识该集合的最后修改时间
};
```

```
//对集合中每个元素使用的模板方法
template<typename ItemType>
void display(ItemType &item)
{
	cout << item << endl;
}

//过滤器的模板方法
template<typename ItemType>
bool Filter(ItemType item)
{
	return item % 2 == 0;
}


//内部迭代器，带过滤功能的
template<typename ItemType>
class InnerFilterIterator
{
public:
	InnerFilterIterator(AbstractCollection<ItemType> &collection, bool (*filter)(ItemType))
		: _collection(collection), _filter(filter)
	{}

	void ProcessItems(void (*func)(ItemType &))
	{
		auto_ptr< AbstractIterator<ItemType> > iter = _collection.CreateIterator();
		for (iter->First(); !iter->IsDone(); iter->Next())
		{
			if (_filter(iter->CurrentItem()))   //演示过滤功能，很容易在内部迭代时实现
			{
				func(iter->CurrentItem());        //对通过了过滤的元素调用需要处理的方法
			}
		}
	}

private:
	AbstractCollection<ItemType> &_collection;
	bool (*_filter)(ItemType);
};


void main()
{
```

```
    AbstractCollection<int> *abc;

    //仅使用抽象接口,无所谓具体类型是什么.
    //在这里new一个VectorCollection或ListCollection对下面的代码来说没有区别
    abc = new VectorCollection<int>();
    //abc = new ListCollection<int>();

    for (int i = 0; i < 10; ++i)
    {
        abc->Add(i);
    }

    cout << "Show outer-iterator" << endl;
    auto_ptr< AbstractIterator<int> > iter = abc->CreateIterator();
    for (iter->First(); !iter->IsDone(); iter->Next())
    {
        cout << iter->CurrentItem() << endl;
    }

    cout << "Show inner-iterator" << endl;
    InnerFilterIterator<int> inner_iter(*abc, Filter<int>);
    inner_iter.ProcessItems(display<int>);
}
```

完结
