

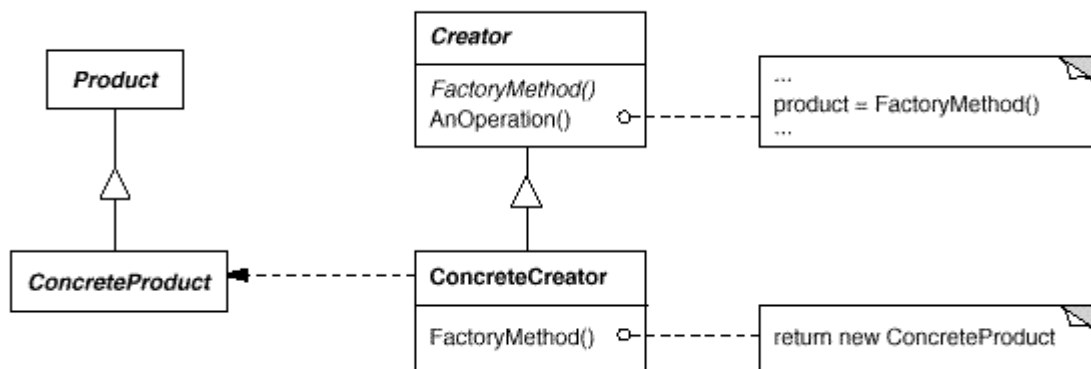
常见设计模式的解析和实现

常见设计模式的解析和实现(C++)之一-Factory 模式

作用:

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。

UML 结构图:



抽象基类:

- 1)Product: 创建出来的对象的抽象基类.
- 2)Factory 创建对象的工厂方法的抽象基类.

接口函数:

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

解析:

在这个模式中, 有两个抽象基类, 一个是 Product 为创建出来的对象的抽象基类, 一个是 Factory 是工厂的抽象基类, 在互相协作的时候都是由相应的 Factory 派生类来生成 Product 的派生类, 也就是说如果要新增一种 Product 那么也要对应的新增一个 Factory, 创建的过程委托给了这个 Factory. 也就是说一个 Factory 和一个 Product 是一一对应的关系.

备注:

设计模式的演示图上把 Factory 类命名为 Creator, 下面的实现沿用了这个命名.

演示实现:

1)Factory.h

```

/*****
|
|   created:   2006/06/30
|   filename:   Factory.h
|   author:    李创
|              http://www.cppblog.com/converse/
|
|   purpose:   Factory 模式的演示代码
|
| *****/

*****/

#ifndef FACTORY_H
#define FACTORY_H

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

class ConcreateProduct
    : public Product
{
public:
    ConcreateProduct();
    virtual ~ConcreateProduct();
};

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

    void AnOperation();

protected:
    virtual Product* FactoryMethod() = 0;
};

class ConcreateCreator
    : public Creator
{

```

```

public:
    ConcreateCreator();
    virtual ~ConcreateCreator();

protected:
    virtual Product* FactoryMethod();
};

#endif

```

2)Factory.cpp

```

/*****
 *
 * created: 2006/06/30
 * filename: Factory.cpp
 * author: 李创
 * http://www.cppblog.com/converse/
 *
 * purpose: Factory 模式的演示代码
 *****/

#include "Factory.h"
#include <iostream>

using namespace std;

ConcreateProduct::ConcreateProduct()
{
    std::cout << "construction of ConcreateProduct\n";
}

ConcreateProduct::~~ConcreateProduct()
{
    std::cout << "destruction of ConcreateProduct\n";
}

void Creator::AnOperation()
{
    Product* p = FactoryMethod();

    std::cout << "an operation of product\n";
}

```

```

ConcreateCreator::ConcreateCreator()
{
    std::cout << "construction of ConcreateCreator\n";
}

ConcreateCreator::~~ConcreateCreator()
{
    std::cout << "destruction of ConcreateCreator\n";
}

Product* ConcreateCreator::FactoryMethod()
{
    return new ConcreateProduct();
}

```

3)Main.cpp(测试代码)

```

/*****
 *
 * created: 2006/06/30
 * filename: Main.cpp
 * author: 李创
 * http://www.cppblog.com/converse/
 *
 * purpose: 测试 Factory 模式的代码
 *****/

#include "Factory.h"
#include <stdlib.h>

int main(int argc, char* argv[])
{
    Creator *p = new ConcreateCreator();
    p->AnOperation();

    delete p;

    system("pause");

    return 0;
}

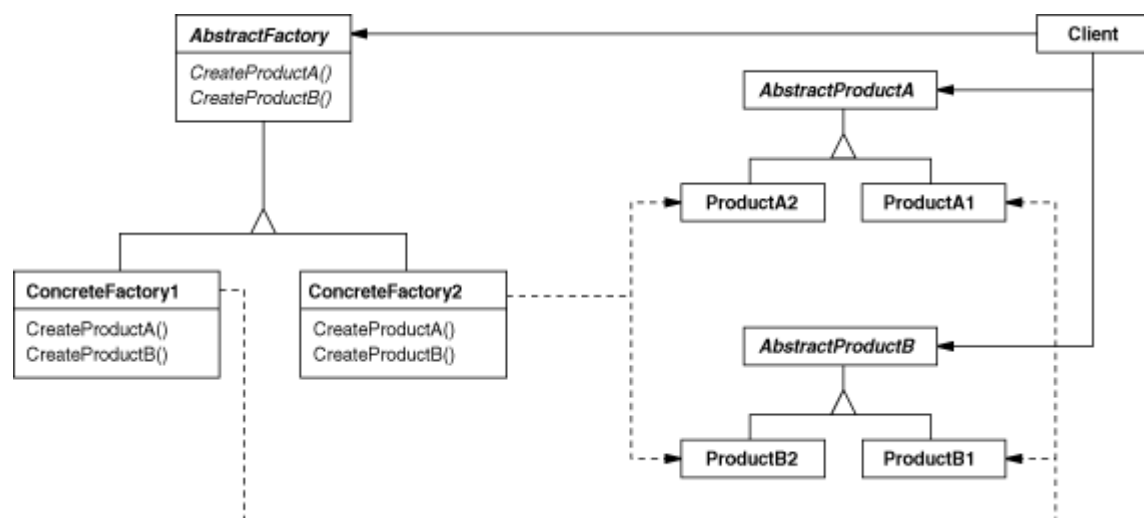
```

常见设计模式的解析和实现(C++)之二-Abstract Factory 模式

作用:

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。

UML 结构图:



抽象基类:

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

2)**AbstractFactory**: 生产这一系列产品的一个抽象工厂,它的派生类是不同的实现.

接口函数:

1)**AbstractFactory::CreateProductA** 和 **AbstractFactory::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.

解析:

Abstract Factory 模式和 **Factory** 最大的差别就是抽象工厂创建的是一系列相关的对象,其中创建的实现其实采用的就是 **Factory** 模式的方法,对于某个实现的有一个派生出来的抽象工厂,另一个实现有另一个派生出来的工厂,等等.

可以举一个简单的例子来解释这个模式: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(也就是 **ProductA** 和 **ProductB** 的不同实现).而负责生产汉堡和鸡腿的就是之前提过的 **Factory** 模式了.

抽象工厂需要特别注意的地方就是区分不同类型的产品和这些产品的不同实现.显而易见的,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.

实现:

1)AbstractFactory.h

```

/*****
|
|   created:   2006/07/19
|   filename:   AbstractFactory.h
|   author:    李创
|              http://www.cppblog.com/converse/
|
|   purpose:   AbstractFactory 的演示代码
|
*****/

// 抽象基类 AbstractProductA,代表产品 A 的抽象
class AbstractProductA
{
public:
    AbstractProductA() {}
    virtual ~AbstractProductA(){};
};

// 派生类 ConcreateProductA1,继承自 AbstractProductA,代表产品 A 的第一种实现
class ConcreateProductA1
    : public AbstractProductA
{
public:
    ConcreateProductA1();
    virtual ~ConcreateProductA1();
};
```

```

// 派生类 ConcreateProductA2,继承自 AbstractProductA,代表产品 A 的第二种实现
class ConcreateProductA2
    : public AbstractProductA
{
public:
    ConcreateProductA2();
    virtual ~ConcreateProductA2();
};

// 抽象基类 AbstractProductB,代表产品 B 的抽象
class AbstractProductB
{
public:
    AbstractProductB() {}
    virtual ~AbstractProductB(){};
};

// 派生类 ConcreateProductB1,继承自 AbstractProductB,代表产品 B 的第一种实现
class ConcreateProductB1
    : public AbstractProductB
{
public:
    ConcreateProductB1();
    virtual ~ConcreateProductB1();
};

// 派生类 ConcreateProductB2,继承自 AbstractProductB,代表产品 B 的第二种实现
class ConcreateProductB2
    : public AbstractProductB
{
public:
    ConcreateProductB2();
    virtual ~ConcreateProductB2();
};

// 抽象基类 AbstractFactory,工厂的抽象类,生产产品 A 和产品 B
class AbstractFactory
{
public:
    AbstractFactory(){}
    virtual ~AbstractFactory(){}

    virtual AbstractProductA* CreateProductA() = 0;
    virtual AbstractProductB* CreateProductB() = 0;
};

```

```

};

// 派生类 ConcreateFactory1,继承自 AbstractFactory
// 生产产品 A 和产品 B 的第一种实现
class ConcreateFactory1
    : public AbstractFactory
{
public:
    ConcreateFactory1();
    virtual ~ConcreateFactory1();

    virtual AbstractProductA* CreateProductA();
    virtual AbstractProductB* CreateProductB();
};

// 派生类 ConcreateFactory2,继承自 AbstractFactory
// 生产产品 A 和产品 B 的第二种实现
class ConcreateFactory2
    : public AbstractFactory
{
public:
    ConcreateFactory2();
    virtual ~ConcreateFactory2();

    virtual AbstractProductA* CreateProductA();
    virtual AbstractProductB* CreateProductB();
};

#endif

```

2)AbstractFactory.cpp

```

/*****
created:    2006/07/19
filename:   AbstractFactory.cpp
author:     李创
           http://www.cppblog.com/converse/

purpose:    AbstractFactory 的演示代码
*****/

#include <iostream>
#include "AbstractFactory.h"

```

```

ConcreateProductA1::ConcreateProductA1()
{
    std::cout << "construction of ConcreateProductA1\n";
}

ConcreateProductA1::~~ConcreateProductA1()
{
    std::cout << "destruction of ConcreateProductA1\n";
}

ConcreateProductA2::ConcreateProductA2()
{
    std::cout << "construction of ConcreateProductA2\n";
}

ConcreateProductA2::~~ConcreateProductA2()
{
    std::cout << "destruction of ConcreateProductA2\n";
}

ConcreateProductB1::ConcreateProductB1()
{
    std::cout << "construction of ConcreateProductB1\n";
}

ConcreateProductB1::~~ConcreateProductB1()
{
    std::cout << "destruction of ConcreateProductB1\n";
}

ConcreateProductB2::ConcreateProductB2()
{
    std::cout << "construction of ConcreateProductB2\n";
}

ConcreateProductB2::~~ConcreateProductB2()
{
    std::cout << "destruction of ConcreateProductB2\n";
}

ConcreateFactory1::ConcreateFactory1()
{
    std::cout << "construction of ConcreateFactory1\n";
}

```

```

    }

    ConcreateFactory1::~ConcreateFactory1()
    {
        std::cout << "destruction of ConcreateFactory1\n";
    }

    AbstractProductA* ConcreateFactory1::CreateProductA()
    {
        return new ConcreateProductA1();
    }

    AbstractProductB* ConcreateFactory1::CreateProductB()
    {
        return new ConcreateProductB1();
    }

    ConcreateFactory2::ConcreateFactory2()
    {
        std::cout << "construction of ConcreateFactory2\n";
    }

    ConcreateFactory2::~ConcreateFactory2()
    {
        std::cout << "destruction of ConcreateFactory2\n";
    }

    AbstractProductA* ConcreateFactory2::CreateProductA()
    {
        return new ConcreateProductA2();
    }

    AbstractProductB* ConcreateFactory2::CreateProductB()
    {
        return new ConcreateProductB2();
    }

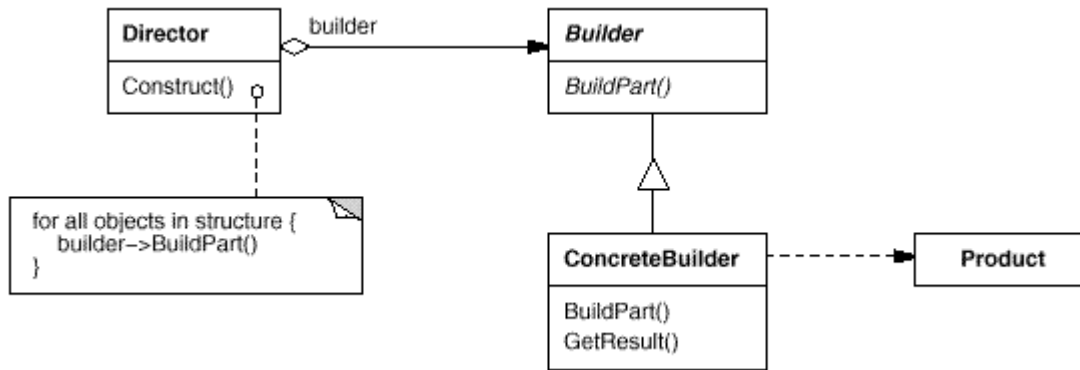
```

3)Main.cpp(测试代码)

```

/*****
*****
|   created:   2006/07/19
|   filename:  Main.cpp
|   author:    李创
|             http://www.cppblog.com/converse/

```

适用于以下情况:

- 1) 当创建复杂对象的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。
- 2) 当构造过程必须允许被构造的对象有不同的表示时。

抽象基类:

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

接口:

1)**Builder::BuildPartA,Builder::BuildPartB**: 是对一个对象不同部分的构建函数接口,**Builder** 的派生类来具体实现.

另外还有一个需要注意的函数,就是 **Director::Construct** 函数,这个函数里面通过调用上面的两个接口函数完成对象的构建--也就是说各个不同部分装配的过程都是一致的(同样的调用的 **Construct** 函数),但是不同的构建方式会有不同的表示(根据 **Builder** 的实际类型来决定如何构建,也就是多态)

解析:

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(哦,你见过车轮长在车座上的么?).也就是说,**Director::Construct** 函数中固定了各个组成部分的装配方式,而具体是装配怎样的组成部分由 **Builder** 的派生类实现.

实现:

Builder 模式的实现基于以下几个面向对象的设计原则: 1)把变化的部分提取出来形成一个基类和对应的接口函数,在这里不会变化的是都会创建 **PartA** 和 **PartB**,变化的则是不同的创建方法,于是就抽取出这里的 **Builder** 基类和 **BuildPartA,BuildPartB** 接口函数 2)采用聚合的方式聚合了会发生变化的基类,就是这里

Director 聚合了 Builder 类的指针.

1)Builder.h

```
/* *****  
*****  
|   created:   2006/07/19  
|   filename:   Builder.h  
|   author:    李创  
|              http://www.cppblog.com/converse/  
|  
|   purpose:   Builder 模式的演示代码  
|*****  
**** */  
  
#ifndef BUILDER_H  
#define BUILDER_H  
  
// 虚拟基类,是所有 Builder 的基类,提供不同部分的构建接口函数  
class Builder  
{  
public :  
    Builder() {} ;  
    virtual ~ Builder() {}  
  
    // 纯虚函数,提供构建不同部分的构建接口函数  
    virtual void BuilderPartA() = 0 ;  
    virtual void BuilderPartB() = 0 ;  
};  
  
// 使用 Builder 构建产品,构建产品的过程都一致,但是不同的 builder 有不同的实现  
// 这个不同的实现通过不同的 Builder 派生类来实现,存有一个 Builder 的指针,通过这个来实现多态调用  
class Director  
{  
public :  
    Director(Builder * pBuilder);  
    ~ Director();  
  
    void Construct();  
  
private :  
    Builder * m_pBuilder;  
};
```

```

// Builder 的派生类,实现 BuilderPartA 和 BuilderPartB 接口函数
class ConcreateBuilder1
    : public Builder
{
public :
    ConcreateBuilder1() {}
    virtual ~ ConcreateBuilder1() {}

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

// Builder 的派生类,实现 BuilderPartA 和 BuilderPartB 接口函数
class ConcreateBuilder2
    : public Builder
{
public :
    ConcreateBuilder2() {}
    virtual ~ ConcreateBuilder2() {}

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

#endif

```

2)Builder.cpp

```

/* *****
*****

created: 2006/07/19
filename: Builder.cpp
author: 李创
        http://www.cppblog.com/converse/

purpose: Builder 模式的演示代码
*****
**** */

#include " Builder.h "
#include < iostream >

void ConcreateBuilder1::BuilderPartA()

```

```

    {
        std::cout << " BuilderPartA by ConcreteBuilder1\n " ;
    }

    void ConcreteBuilder1::BuilderPartB()
    {
        std::cout << " BuilderPartB by ConcreteBuilder1\n " ;
    }

    void ConcreteBuilder2::BuilderPartA()
    {
        std::cout << " BuilderPartA by ConcreteBuilder2\n " ;
    }

    void ConcreteBuilder2::BuilderPartB()
    {
        std::cout << " BuilderPartB by ConcreteBuilder2\n " ;
    }

    Director::Director(Builder * pBuilder)
        : m_pBuilder(pBuilder)
    {
    }

    Director::~~ Director()
    {
        delete m_pBuilder;
        m_pBuilder = NULL;
    }

    // Construct 函数表示一个对象的整个构建过程,不同的部分之间的装配方式都是一致的,
    // 首先构建 PartA 其次是 PartB,只是根据不同的构建者会有不同的表示
    void Director::Construct()
    {
        m_pBuilder -> BuilderPartA();
        m_pBuilder -> BuilderPartB();
    }

```

3)Main.cpp

```

/* *****
*****
created: 2006/07/20
filename: Main.cpp

```

```

author:      李创
             http://www.cppblog.com/converse/

purpose:     Builder 模式的测试代码
*****
**** */

#include " Builder.h "
#include < stdlib.h >

int main()
{
    Builder * pBuilder1 = new ConcreateBuilder1;
    Director * pDirector1 = new Director(pBuilder1);
    pDirector1 -> Construct();

    Builder * pBuilder2 = new ConcreateBuilder2;
    Director * pDirector2 = new Director(pBuilder2);
    pDirector2 -> Construct();

    delete pDirector1;
    delete pDirector2;

    system( " pause " );

    return 0 ;
}

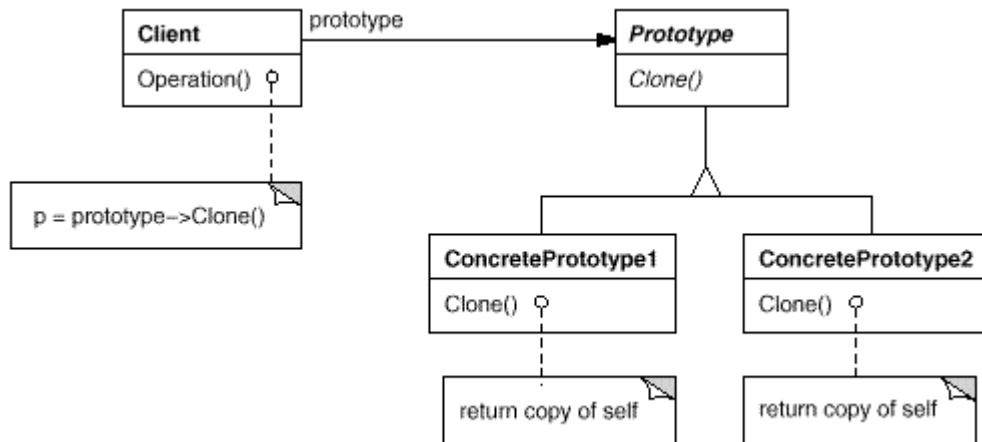
```

常见设计模式的解析和实现(C++)之四-Prototype 模式

作用:

用原型实例指定创建对象的种类，并且通过拷贝这些原型创建新的对象。

UML 结构图:



抽象基类:

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

接口函数:

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

解析:

Prototype 模式其实就是常说的"虚拟构造函数"一个实现, C++ 的实现机制中并没有支持这个特性, 但是通过不同派生类实现的 **Clone** 接口函数可以完成与"虚拟构造函数"同样的效果. 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.

实现:

1) **Prototype.h**

```

/*****
 *
 * created: 2006/07/20
 * filename: Prototype.h
 * author: 李创
 * http://www.cppblog.com/converse/
 *
 * purpose: Prototype 模式的演示代码
 *****/

#ifndef PROTOTYPE_H
#define PROTOTYPE_H

// 虚拟基类, 所有原型的基类, 提供 Clone 接口函数
  
```

```

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

    virtual Prototype* Clone() = 0;
};

// 派生自 Prototype,实现 Clone 方法
class ConcreatePrototype1
: public Prototype
{
public:
    ConcreatePrototype1();
    ConcreatePrototype1(const ConcreatePrototype1&);
    virtual ~ConcreatePrototype1();

    virtual Prototype* Clone();
};

// 派生自 Prototype,实现 Clone 方法
class ConcreatePrototype2
: public Prototype
{
public:
    ConcreatePrototype2();
    ConcreatePrototype2(const ConcreatePrototype2&);
    virtual ~ConcreatePrototype2();

    virtual Prototype* Clone();
};

#endif

```

2)Prototype.cpp

```

/*****
*****
|   created:   2006/07/20
|   filename:   Prototype.cpp
|   author:    李创
|              http://www.cppblog.com/converse/
|
|   purpose:   Prototype 模式的演示代码

```

```

*****
*****/

#include "Prototype.h"
#include <iostream>

ConcreatePrototype1::ConcreatePrototype1()
{
    std::cout << "construction of ConcreatePrototype1\n";
}

ConcreatePrototype1::~~ConcreatePrototype1()
{
    std::cout << "destruction of ConcreatePrototype1\n";
}

ConcreatePrototype1::ConcreatePrototype1(const ConcreatePrototype1&)
{
    std::cout << "copy construction of ConcreatePrototype1\n";
}

Prototype* ConcreatePrototype1::Clone()
{
    return new ConcreatePrototype1(*this);
}

ConcreatePrototype2::ConcreatePrototype2()
{
    std::cout << "construction of ConcreatePrototype2\n";
}

ConcreatePrototype2::~~ConcreatePrototype2()
{
    std::cout << "destruction of ConcreatePrototype2\n";
}

ConcreatePrototype2::ConcreatePrototype2(const ConcreatePrototype2&)
{
    std::cout << "copy construction of ConcreatePrototype2\n";
}

Prototype* ConcreatePrototype2::Clone()
{

```

```

    return new ConcreatePrototype2(*this);
}

```

3)Main.cpp

```

/*****
created:    2006/07/20
filename:   Main.cpp
author:     李创
           http://www.cppblog.com/converse/

purpose:    Prototype 模式的测试代码
*****/

#include "Prototype.h"
#include <stdlib.h>

int main()
{
    Prototype* pPrototype1 = new ConcreatePrototype1();
    Prototype* pPrototype2 = pPrototype1->Clone();

    Prototype* pPrototype3 = new ConcreatePrototype2();
    Prototype* pPrototype4 = pPrototype3->Clone();

    delete pPrototype1;
    delete pPrototype2;
    delete pPrototype3;
    delete pPrototype4;

    system("pause");

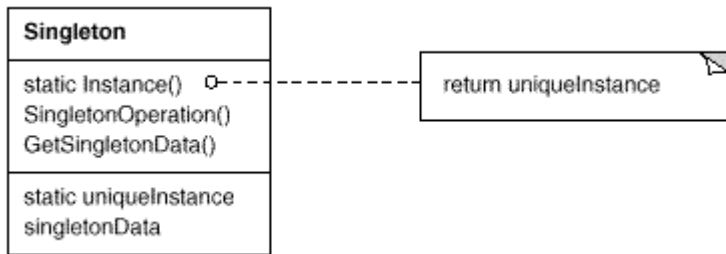
    return 0;
}

```

常见设计模式的解析和实现(C++)之五-Singleton 模式

作用: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。

UML 结构图:



解析:

Singleton 模式其实是对全局静态变量的一个取代策略,上面提到的 **Singleton** 模式的两个作用在 **C++** 中是通过如下的机制实现的: **1)** 仅有一个实例,提供一个类的静态成员变量,大家知道类的静态成员变量对于一个类的所有对象而言是惟一的 **2)** 提供一个访问它的全局访问点,也就是提供对应的访问这个静态成员变量的静态成员函数,对类的所有对象而言也是惟一的.在 **C++** 中,可以直接使用类域进行访问而不必初始化一个类的对象.

下面的实现其实是 **Singleton** 的一个简单实现,并不是特别的通用,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,比如这样实现:

```
template<typename T>
class Singleton
{
    // ...类的声明
};

// 需要采用 singleton 模式的类
class Test
: public Singleton<Test>
{
    // ...类的声明
};
```

但是,下面的实现还是采用最简单的实现办法,起的是演示的作用

实现:

1) Singleton.h

```
/* *****
*****
```

```

created: 2006/07/20
filename: Singleton.h
author: 李创
http://www.cppblog.com/converse/

purpose: Singleton 模式的演示代码
*****

*****/

#ifndef SINGLETON_H
#define SINGLETON_H

class Singleton
{
public:
    Singleton(){};
    ~Singleton(){};

    // 静态成员函数,提供全局访问的接口
    static Singleton* GetInstancePtr();
    static Singleton GetInstance();

    void Test();

private:
    // 静态成员变量,提供全局惟一的一个实例
    static Singleton* m_pStatic;
};

#endif

```

2)Singleton.cpp

```

/*****
created: 2006/07/20
filename: Singleton.cpp
author: 李创
http://www.cppblog.com/converse/

purpose: Singleton 模式的演示代码
*****

*****/

#include "Singleton.h"

```

```

#include <iostream>

// 类的静态成员变量要在类体外进行定义
Singleton* Singleton::m_pStatic = NULL;

Singleton* Singleton::GetInstancePtr()
{
    if (NULL == m_pStatic)
    {
        m_pStatic = new Singleton();
    }

    return m_pStatic;
}

Singleton Singleton::GetInstance()
{
    return *GetInstancePtr();
}

void Singleton::Test()
{
    std::cout << "Test!\n";
}

```

3)Main.cpp

```

/*****
created: 2006/07/20
filename: Main.cpp
author: 李创
http://www.cppblog.com/converse/

purpose: Singleton 模式的测试代码
*****/

#include "Singleton.h"
#include <stdlib.h>

int main()
{
    // 不用初始化类对象就可以访问了
    Singleton::GetInstancePtr()->Test();
}

```

```

Singleton::GetInstance().Test();

system("pause");

return 0;
}

```

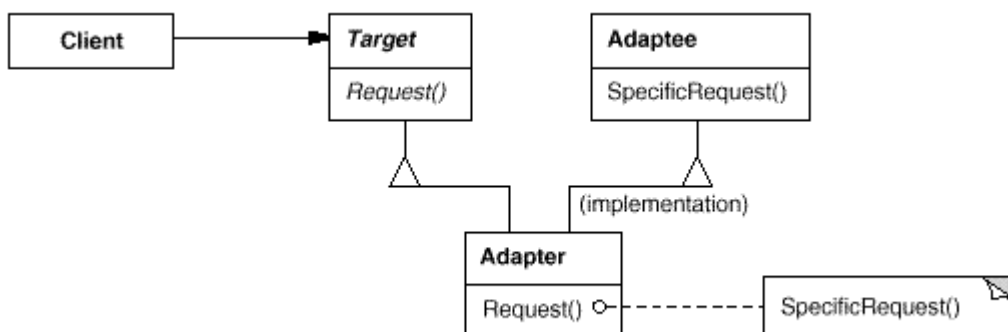
常见设计模式的解析和实现(C++)之六-Adapt 模式

作用:

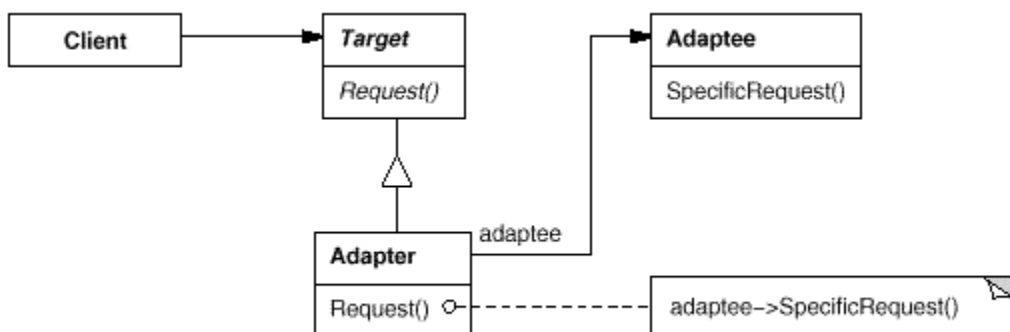
将一个类的接口转换成客户希望的另外一个接口。**Adapt** 模式使得原本由于接口不兼容而不能一起工作的那些类可以一起工作。

UML 示意图

1)采用继承原有接口类的方式



2)采用组合原有接口类的方式



解析:

Adapt 模式其实就是把完成同样的一个功能但是接口不能兼容的类桥接在一起使之可以在一起工作,这个模式使得复用旧的接口成为可能.

实现:

Adapt 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,这里采用的是第二种实现方法.

1)Adapt.h

```

/*****
*****
|
|   created:   2006/07/20
|   filename:   Adapter.h
|   author:    李创
|              http://www.cppblog.com/converse/
|
|   purpose:   Adapter 模式的演示代码
|
| *****/

#ifndef ADAPTER_H
#define ADAPTER_H

// 需要被 Adapt 的类
class Target
{
public:
    Target(){}
    virtual ~Target() {}

    virtual void Request() = 0;
};

// 与被 Adapt 对象提供不兼容接口的类
class Adaptee
{
public:
    Adaptee(){}
    ~Adaptee(){}
    void SpecialRequest();
};

// 进行 Adapt 的类,采用聚合原有接口类的方式
class Adapter
    : public Target
{
public:
    Adapter(Adaptee* pAdaptee);
};

```

```

    virtual ~Adapter();

    virtual void Request();

private:
    Adaptee* m_pAdptee;
};

#endif

```

2)Adapt.cpp

```

/*****
*****
|   created:   2006/07/20
|   filename:   Adapter.cpp
|   author:    李创
|              http://www.cppblog.com/converse/
|
|   purpose:   Adapter 模式的演示代码
*****/

#include "Adapter.h"
#include <iostream>

void Adaptee::SpecialRequest()
{
    std::cout << "SpecialRequest of Adaptee\n";
}

Adapter::Adapter(Adaptee* pAdaptee)
    : m_pAdptee(pAdaptee)
{
}

Adapter::~~Adapter()
{
    delete m_pAdptee;
    m_pAdptee = NULL;
}

void Adapter::Request()
{

```

```

std::cout << "Request of Adapter\n";

m_pAdptee->SpecialRequest();
}

```

3)Main.cpp

```

/*****
created: 2006/07/20
filename: Main.cpp
author: 李创
        http://www.cppblog.com/converse/

purpose: Adapter 模式的测试代码
*****/

#include "Adapter.h"
#include <stdlib.h>

int main()
{
    Adaptee *pAdaptee = new Adaptee;
    Target *pTarget = new Adapter(pAdaptee);
    pTarget->Request();

    delete pTarget;

    system("pause");

    return 0;
}

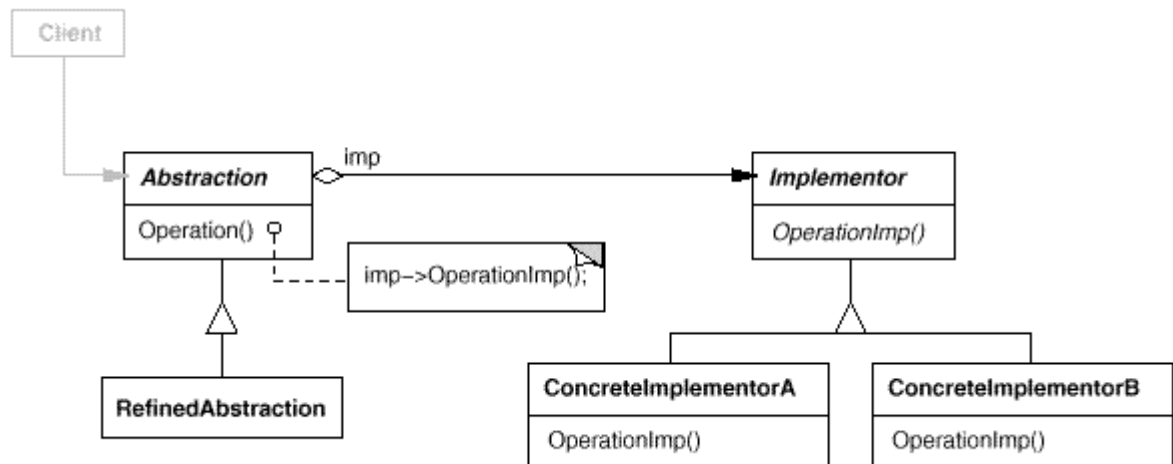
```

常见设计模式的解析和实现(C++)之七-Bridge 模式

作用:

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。

UML 结构图:



抽象基类:

1)**Abstraction**: 某个抽象类,它的实现方式由 **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.

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.

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 **BuildPart** 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.

因此,如果以一些最基本的面向对象的设计原则来分析这些模式的实现的话,还是可以看到很多共同的地方的.

实现:

1)Bridge.h

```

/*****
|
|   created:   2006/07/20
|   filename:   Brige.h
|   author:    李创
|              http://www.cppblog.com/converse/
|
|   purpose:   Brige 模式的演示代码
|
*****/

#ifndef BRIDEG_H
#define BRIDEG_H

class Implementor;

// 维护一个 Implementor 类的指针
class Abstraction
{
public:
    Abstraction(Implementor* pImplementor);
    virtual ~Abstraction();

    void Operation();

protected:
    Implementor* m_pImplementor;
};

// 为实现 Abstraction 定义的抽象基类,定义了实现的接口函数
class Implementor
{
public:
    Implementor(){}
    virtual ~Implementor(){}

    virtual void OperationImpl() = 0;
};

// 继承自 Implementor,是 Implementor 的不同实现之一
class ConcreateImplementorA
```

```

        : public Implementor
    {
    public:
        ConcreateImplementorA(){}
        virtual ~ConcreateImplementorA(){}

        virtual void OperationImpl();
    };

    // 继承自 Implementor,是 Implementor 的不同实现之一
    class ConcreateImplementorB
        : public Implementor
    {
    public:
        ConcreateImplementorB(){}
        virtual ~ConcreateImplementorB(){}

        virtual void OperationImpl();
    };

#endif

```

2)Bridge.cpp

```

/*****
 *
 * created: 2006/07/20
 * filename: Brige.cpp
 * author: 李创
 * http://www.cppblog.com/converse/
 *
 * purpose: Brige 模式的演示代码
 *****/

#include "Brige.h"
#include <iostream>

void ConcreateImplementorA::OperationImpl()
{
    std::cout << "Implementation by ConcreateImplementorA\n";
}

void ConcreateImplementorB::OperationImpl()
{

```

```

    std::cout << "Implementation by ConcreteImplementorB\n";
}

Abstraction::Abstraction(Implementor* pImplementor)
    : m_pImplementor(pImplementor)
{
}

Abstraction::~~Abstraction()
{
    delete m_pImplementor;
    m_pImplementor = NULL;
}

void Abstraction::Operation()
{
    m_pImplementor->OperationImpl();
}

```

3)Main.cpp

```

/*****
 *
 * created: 2006/07/20
 * filename: Main.cpp
 * author: 李创
 * http://www.cppblog.com/converse/
 *
 * purpose: Bridge 模式的测试代码
 *****/

#include "Brige.h"
#include <stdlib.h>

int main()
{
    ConcreteImplementorA *pImplA = new ConcreteImplementorA();
    Abstraction *pAbstraction1 = new Abstraction(pImplA);
    pAbstraction1->Operation();

    ConcreteImplementorB *pImplB = new ConcreteImplementorB();
    Abstraction *pAbstraction2 = new Abstraction(pImplB);
    pAbstraction2->Operation();
}

```

```

delete pAbstraction1;
delete pAbstraction2;

system("pause");

return 0;
}

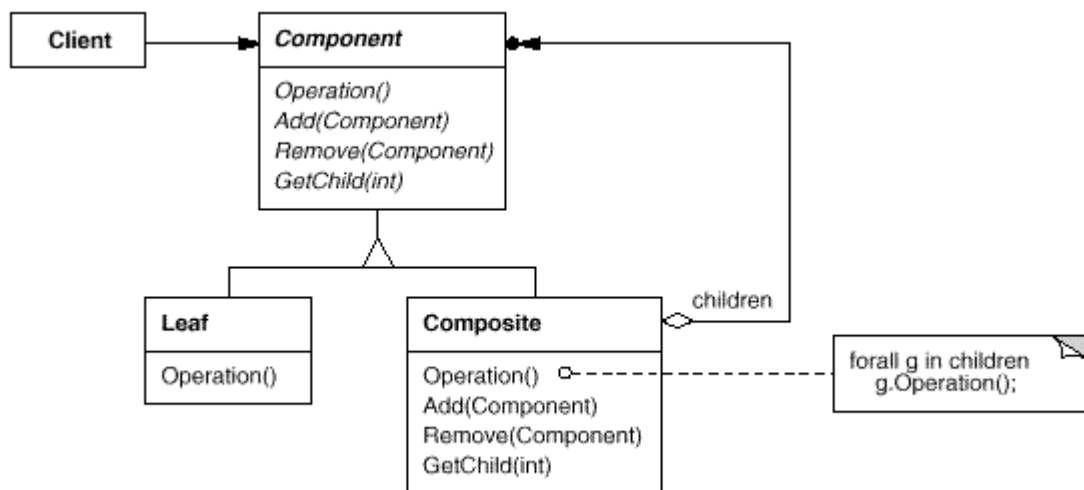
```

常见设计模式的解析和实现(C++)之八-Composite 模式

作用:

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。

UML 结构图:



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(如这里的 **Add, Remove, GetChild** 函数),声明一个接口函数可以访问 **Component** 的子组件.

接口函数:

1)**Component::Operation**: 定义了各个组件共有的行为接口,由各个组件的具体实现.

2)**Component::Add** 添加一个子组件

3)Component::Remove::删除一个子组件.

4)Component::GetChild:获得子组件的指针.

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,它主要分为两个派生类,其中的 **Leaf** 是叶子结点,也就是不含有子组件的结点,而 **Composite** 是含有子组件的类.举一个例子来说明这个模式,在 **UI** 的设计中,最基本的控件是诸如 **Button,Edit** 这样的控件,相当于是这里的 **Leaf** 组件,而比较复杂的控件比如 **List** 则可也看做是由这些基本的组件组合起来的控件,相当于这里的 **Composite**,它们之间有一些行为含义是相同的,比如在控件上作一个点击,移动操作等等的,这些都可以定义为抽象基类中的接口虚函数,由各个派生类去实现之,这些都会有的行为就是这里的 **Operation** 函数,而添加,删除等进行组件组合的操作只有非叶子结点才可能有,所以虚拟基类中只是提供接口而且默认的实现是什么都不做.

实现:

1)Composite.h

```

/*****
|
|   created:   2006/07/20
|   filename:   Composite.h
|   author:    李创
|              http://www.cppblog.com/converse/
|
|   purpose:   Composite 模式的演示代码
|
| *****/

#ifndef COMPOSITE_H
#define COMPOSITE_H

#include <list>

// 组合中的抽象基类
class Component
{
public:
    Component(){}
    virtual ~Component(){}

    // 纯虚函数,只提供接口,没有默认的实现
    virtual void Operation() = 0;

    // 虚函数,提供接口,有默认的实现就是什么都不做

```

```

    virtual void Add(Component* pChild);
    virtual void Remove(Component* pChild);
    virtual Component* GetChild(int nIndex);
};

// 派生自 Component,是其中的叶子组件的基类
class Leaf
    : public Component
{
public:
    Leaf(){}
    virtual ~Leaf(){}

    virtual void Operation();
};

// 派生自 Component,是其中的含有子件的组件的基类
class Composite
    : public Component
{
public:
    Composite(){}
    virtual ~Composite();

    virtual void Operation();

    virtual void Add(Component* pChild);
    virtual void Remove(Component* pChild);
    virtual Component* GetChild(int nIndex);

private:
    // 采用 list 容器去保存子组件
    std::list<Component*> m_ListOfComponent;
};

#endif

```

2)Composite.cpp

```

/*****
*****
    created: 2006/07/20
    filename: Composite.cpp
    author: 李创
    http://www.cppblog.com/converse/

```



```

    for (iter1 = m_ListOfComponent.begin(), iter2 = m_ListOfComponent.end();
        iter1 != iter2;
        )
    {
        temp = iter1;
        ++iter1;
        delete (*temp);
    }
}

void Composite::Add(Component* pChild)
{
    m_ListOfComponent.push_back(pChild);
}

void Composite::Remove(Component* pChild)
{
    std::list<Component*>::iterator iter;

    iter = find(m_ListOfComponent.begin(), m_ListOfComponent.end(), pChild);

    if (m_ListOfComponent.end() != iter)
    {
        m_ListOfComponent.erase(iter);
    }
}

Component* Composite::GetChild(int nIndex)
{
    if (nIndex <= 0 || nIndex > m_ListOfComponent.size())
        return NULL;

    std::list<Component*>::iterator iter1, iter2;
    int i;
    for (i = 1, iter1 = m_ListOfComponent.begin(), iter2 = m_ListOfComponent.
end();
        iter1 != iter2;
        ++iter1, ++i)
    {
        if (i == nIndex)
            break;
    }
}

```

```

    return *iter1;
}

void Composite::Operation()
{
    std::cout << "Operation by Composite\n";

    std::list<Component*>::iterator iter1, iter2;

    for (iter1 = m_ListOfComponent.begin(), iter2 = m_ListOfComponent.end();
         iter1 != iter2;
         ++iter1)
    {
        (*iter1)->Operation();
    }
}

```

3)Main.cpp

```

/*****
created: 2006/07/20
filename: Main.cpp
author: 李创
        http://www.cppblog.com/converse/

purpose: Composite 模式的测试代码
*****/

#include "Composite.h"
#include <stdlib.h>

int main()
{
    Leaf *pLeaf1 = new Leaf();
    Leaf *pLeaf2 = new Leaf();

    Composite* pComposite = new Composite;
    pComposite->Add(pLeaf1);
    pComposite->Add(pLeaf2);
    pComposite->Operation();
    pComposite->GetChild(2)->Operation();

    delete pComposite;
}

```

```

    system("pause");

    return 0;
}

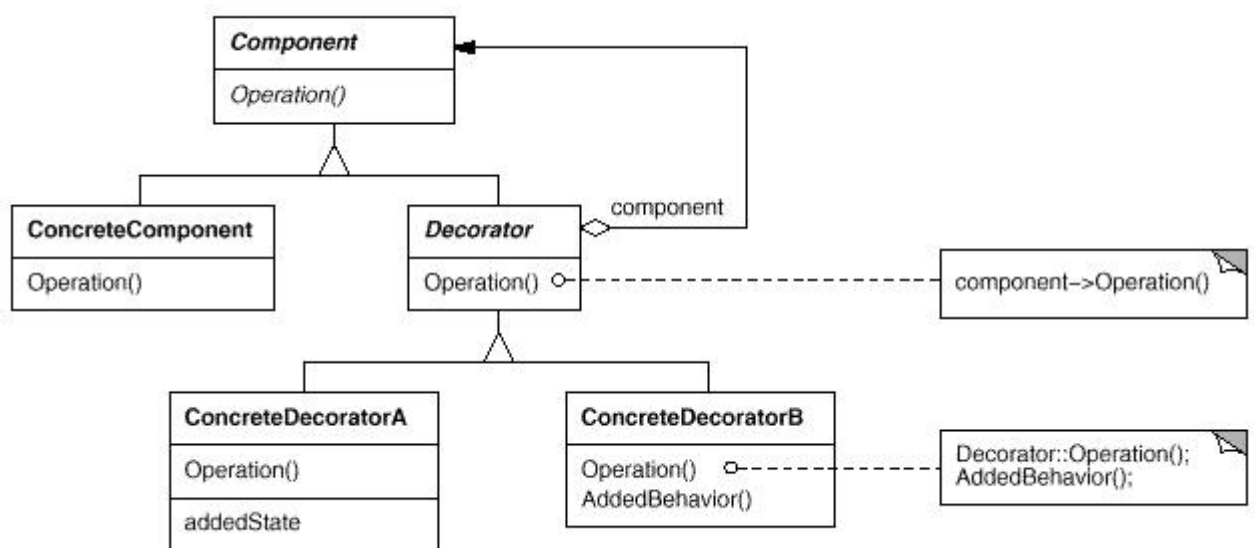
```

常见设计模式的解析和实现(C++)之九-Decorator 模式

作用:

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。

UML 结构图:



抽象基类:

1)**Component**: 定义一个对象接口,可以为这个接口动态的添加职责.

2)**Decorator**: 维持一个指向 **Component** 的指针,并且有一个和 **Component** 一致的接口函数.

接口函数:

1)**Component::Operation**: 这个接口函数由 **Component** 声明,因此 **Component** 的派生类都需要实现,可以在这个接口函数的基础上给它动态添加职责.

解析:

Decorator 的派生类可以为 **ConcreteComponent** 类的对象动态的添加职责,

或者可以这么说:Decorator 的派生类装饰 ConcreateComponent 类的对象.具体是这么实现的,首先初始化一个 ConcreateComponent 类的对象(被装饰者),采用这个对象去生成一个 Decorator 对象(装饰者),之后对 Operation 函数的调用则是对这个 Decorator 对象成员函数的多态调用.这里的实现要点是 Decorator 类和 ConcreateComponent 类都继承自 Component,从而两者的接口函数是一致的;其次,Decorator 维护了一个指向 Component 的指针,从而可以实现对 Component::Operation 函数的动态调用.

实现:

1)Decorator.h

```

/*****
 *
 * created: 2006/07/20
 * filename: Decorator.h
 * author: 李创
 * http://www.cppblog.com/converse/
 *
 * purpose: Decorator 模式的演示代码
 *****/

#ifndef DECORATOR_H
#define DECORATOR_H

// 抽象基类,定义一个对象接口,可以为这个接口动态的添加职责.
class Component
{
public:
    Component(){}
    virtual ~Component(){}

    // 纯虚函数,由派生类实现
    virtual void Operation() = 0;
};

// 抽象基类,维护一个指向 Component 对象的指针
class Decorator
    : public Component
{
public:
    Decorator(Component* pComponent) : m_pComponent(pComponent){}
    virtual ~Decorator();

protected:

```

```

    Component* m_pComponent;
};

// 派生自 Component,在这里表示需要给它动态添加职责的类
class ConcreateComponent
    : public Component
{
public:
    ConcreateComponent(){}
    virtual ~ConcreateComponent(){}

    virtual void Operation();
};

// 派生自 Decorator,这里代表为 ConcreateComponent 动态添加职责的类
class ConcreateDecorator
    : public Decorator
{
public:
    ConcreateDecorator(Component* pComponent) : Decorator(pComponent){}
    virtual ~ConcreateDecorator(){}

    virtual void Operation();

private:
    void AddedBehavior();
};

#endif

```

2)Decorator.cpp

```

/*****
created:    2006/07/20
filename:   Decorator.cpp
author:     李创
           http://www.cppblog.com/converse/

purpose:    Decorator 模式的演示代码
*****/

#include "Decorator.h"
#include <iostream>

```

```

Decorator::~~Decorator()
{
    delete m_pComponent;
    m_pComponent = NULL;
}

void ConcreateComponent::Operation()
{
    std::cout << "Operation of ConcreateComponent\n";
}

void ConcreateDecorator::Operation()
{
    m_pComponent->Operation();
    AddedBehavior();
}

void ConcreateDecorator::AddedBehavior()
{
    std::cout << "AddedBehavior of ConcreateDecorator\n";
}

```

3)Main.cpp

```

/*****
 *
 * created: 2006/07/20
 * filename: Main.cpp
 * author: 李创
 * http://www.cppblog.com/converse/
 *
 * purpose: Decorator 模式的测试代码
 *****/

#include "Decorator.h"
#include <stdlib.h>

int main()
{
    // 初始化一个 Component 对象
    Component* pComponent = new ConcreateComponent();
    // 采用这个 Component 对象去初始化一个 Decorator 对象,
    // 这样就可以为这个 Component 对象动态添加职责

```

```

Decorator* pDecorator = new ConcreateDecorator(pComponent);

pDecorator->Operation();

delete pDecorator;

system("pause");

return 0;
}

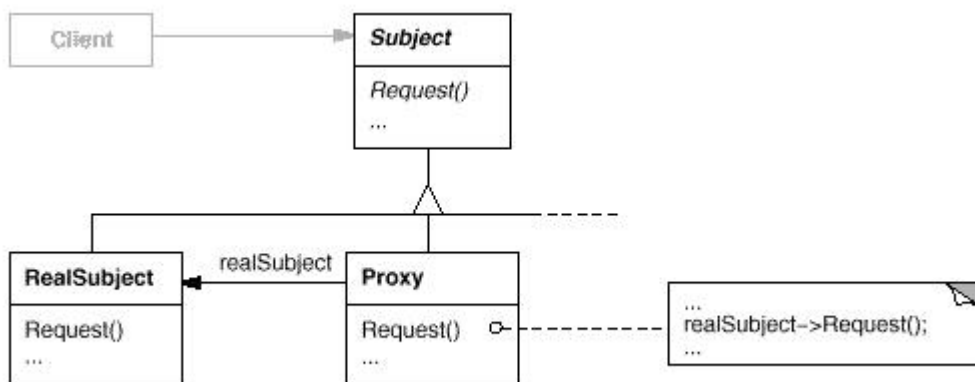
```

常见设计模式的解析和实现(C++)之十-Proxy 模式

作用:

为其他对象提供一种代理以控制对这个对象的访问。

UML 结构图:



抽象基类:

1)Subject: 定义了 **Proxy** 和 **RealSubject** 的公有接口,这样就可以在任何需要使用到 **RealSubject** 的地方都使用 **Proxy**.

解析:

Proxy 其实是基于这样一种时常使用到的技术-某个对象直到它真正被使用到的时候才被初始化,在没有使用到的时候就暂时用 **Proxy** 作一个占位符.这个模式实现的要点就是 **Proxy** 和 **RealSubject** 都继承自 **Subject**,这样保证了两个的接口都是一致的.

实现:

1)Proxy.h

```

/*****
|
|   created:   2006/07/26
|   filename:   Proxy.h
|   author:    李创
|              http://www.cppblog.com/converse/
|
|   purpose:   Proxy 模式的演示代码
|
*****/

#ifndef PROXY_H
#define PROXY_H

// 定义了 Proxy 和 RealSubject 的公有接口,
// 这样就可以在任何需要使用到 RealSubject 的地方都使用 Proxy.
class Subject
{
public:
    Subject(){}
    virtual ~Subject(){}

    virtual void Request() = 0;
};

// 真正使用的实体
class RealSubject
    : public Subject
{
public:
    RealSubject();
    virtual ~RealSubject(){}

    virtual void Request();
};

// 代理类,含有一个指向 RealSubject 对象的指针
class Proxy
    : public Subject
{
public:
    Proxy();

```

```

    virtual ~Proxy();

    virtual void Request();

private:
    RealSubject* m_pRealSubject;
};
#endif

```

2) Proxy.cpp

```

//*****
*****
|   created:   2006/07/26
|   filename:   Proxy.cpp
|   author:    李创
|              http://www.cppblog.com/converse/
|
|   purpose:   Proxy 模式的演示代码
|*****
*****/

#include "Proxy.h"
#include <iostream>

RealSubject::RealSubject()
{
    std::cout << "Constructing a RealSubject\n";
}

void RealSubject::Request()
{
    std::cout << "Request By RealSubject\n";
}

Proxy::Proxy()
    : m_pRealSubject(NULL)
{
    std::cout << "Constructing a Proxy\n";
}

Proxy::~~Proxy()
{
    delete m_pRealSubject;
    m_pRealSubject = NULL;
}

```

```

}

void Proxy::Request()
{
    // 需要使用 RealSubject 的时候才去初始化
    if (NULL == m_pRealSubject)
    {
        std::cout << "Request By Proxy\n";
        m_pRealSubject = new RealSubject();
    }
    m_pRealSubject->Request();
}

```

3)Main.cpp

```

/*****
 *
 * created: 2006/07/26
 * filename: Main.cpp
 * author: 李创
 * http://www.cppblog.com/converse/
 *
 * purpose: Proxy 模式的测试代码
 *****/

#include "Proxy.h"
#include <stdlib.h>

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

    delete pProxy;

    system("pause");

    return 0;
}

```

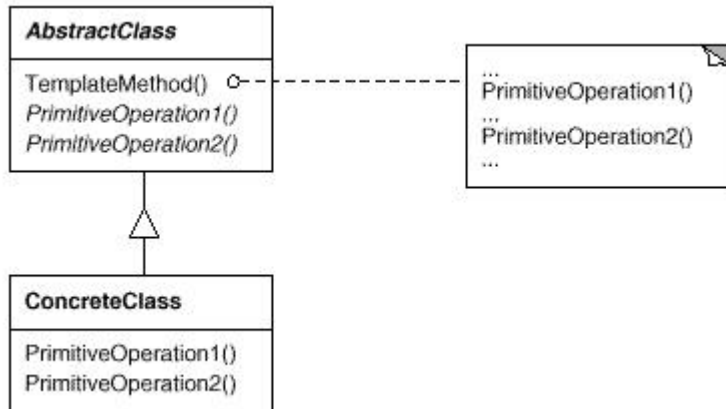
常见设计模式的解析和实现(C++)之十一-Template

Method 模式

作用:

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步骤。

UML 结构图:



抽象基类:

1)**AbstractClass**: 抽象基类, 定义算法的轮廓

解析:

TemplateMethod 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.

实现:

1)TemplateMethod.h

```

/*****
 *
 * created: 2006/07/20
 * filename: TemplateMethod.h
 * author: 李创
 * http://www.cppblog.com/converse/
 *
 * purpose: TemplateMethod 模式的演示代码
 *****/
/
```

```

// 抽象基类,定义算法的轮廓
class AbstractClass
{
public:
    AbstractClass(){}
    virtual ~AbstractClass(){}

    // 这个函数中定义了算法的轮廓
    void TemplateMethod();

protected:
    // 纯虚函数,由派生类实现之
    virtual void PrimitiveOperation1() = 0;
    virtual void PrimitiveOperation2() = 0;
};

// 继承自 AbstractClass,实现算法
class ConcreateClass
: public AbstractClass
{
public:
    ConcreateClass(){}
    virtual ~ConcreateClass(){}

protected:
    virtual void PrimitiveOperation1();
    virtual void PrimitiveOperation2();
};

```

2)TemplateMethod.cpp

```

/*****
created: 2006/07/20
filename: TemplateMethod.cpp
author: 李创
http://www.cppblog.com/converse/

purpose: TemplateMethod 模式的演示代码
*****/

#include "TemplateMethod.h"
#include <iostream>

```

```

void AbstractClass::TemplateMethod()
{
    PrimitiveOperation1();
    PrimitiveOperation2();
}

void ConcreateClass::PrimitiveOperation1()
{
    std::cout << "PrimitiveOperation1 by ConcreateClass\n";
}

void ConcreateClass::PrimitiveOperation2()
{
    std::cout << "PrimitiveOperation2 by ConcreateClass\n";
}

```

3)Main.cpp

```

/*****
 *
 * created: 2006/07/20
 * filename: Main.cpp
 * author: 李创
 * http://www.cppblog.com/converse/
 *
 * purpose: TemplateMethod 模式的测试代码
 *****/

#include "TemplateMethod.h"
#include <stdlib.h>

int main()
{
    AbstractClass* pConcreateClass = new ConcreateClass;
    pConcreateClass->TemplateMethod();

    delete pConcreateClass;

    system("pause");

    return 0;
}

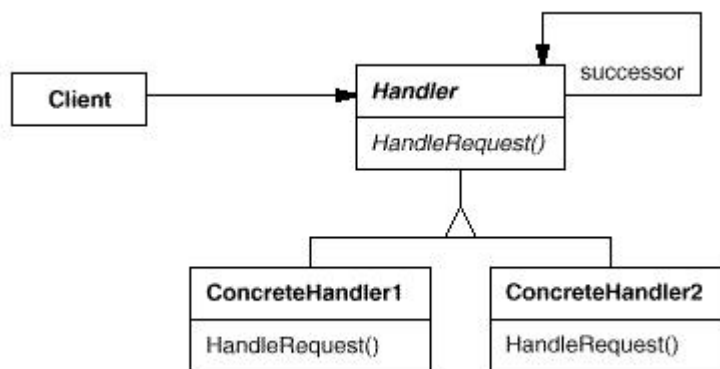
```

常见设计模式的解析和实现(C++)之十二-ChainOfResponsibility 模式

作用:

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.

UML 结构图:



抽象基类:

1)Handler: 定义一个处理请求的接口,在图中这个接口就是 **HandleRequest** 函数,这个类同时有一个指向 **Handler** 对象的指针,指向后续的处理请求的对象(如果有的话).

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.

实现:

1)ChainOfResponsibility.h

```
*****
*****
| created: 2006/07/20
```

```

filename: ChainOfResponsibility.h
author: 李创
http://www.cppblog.com/converse/

purpose: ChainOfResponsibility 模式的演示代码
*****
*****/

#ifndef CHAINOFRESPONSIBILITY_H
#define CHAINOFRESPONSIBILITY_H

#include <stdio.h>

// 抽象基类,定义一个处理请求的接口
class Handler
{
public:
    Handler(Handler *pSuccessor = NULL);
    virtual ~Handler();

    // 纯虚函数,由派生类实现
    virtual void HandleRequest() = 0;

protected:
    Handler* m_pSuccessor;
};

class ConcreteHandler1
    : public Handler
{
public:
    ConcreteHandler1(Handler *pSuccessor = NULL) : Handler(pSuccessor){}
    virtual ~ConcreteHandler1(){}

    virtual void HandleRequest();
};

class ConcreteHandler2
    : public Handler
{
public:
    ConcreteHandler2(Handler *pSuccessor = NULL) : Handler(pSuccessor){}
    virtual ~ConcreteHandler2(){}
};

```

```

    virtual void HandleRequset();
};

#endif

```

2)ChainOfResponsibility.cpp

```

/*****
created:    2006/07/20
filename:   ChainOfResponsibility.cpp
author:     李创
           http://www.cppblog.com/converse/

purpose:    ChainOfResponsibility 模式的演示代码
*****/

#include "ChainOfResponsibility.h"
#include <iostream>

Handler::Handler(Handler *pSuccessor /* = NULL */)
: m_pSuccessor(pSuccessor)
{

}

Handler::~~Handler()
{
    delete m_pSuccessor;
    m_pSuccessor = NULL;
}

void ConcreateHandler1::HandleRequset()
{
    if (NULL != m_pSuccessor)
    {
        m_pSuccessor->HandleRequset();
    }
    else
    {
        std::cout << "HandleRequset by ConcreateHandler1\n";
    }
}

```

```

void ConcreateHandler2::HandleRequset()
{
    if (NULL != m_pSuccessor)
    {
        m_pSuccessor->HandleRequset();
    }
    else
    {
        std::cout << "HandleRequset by ConcreateHandler2\n";
    }
}

```

3)Main.cpp

```

/*****
*****
|   created:   2006/07/20
|   filename:  Main.cpp
|   author:    李创
|              http://www.cppblog.com/converse/
|
|   purpose:   ChainOfResponsibility 模式的测试代码
| *****/
*****/

#include "ChainOfResponsibility.h"
#include <stdlib.h>

int main()
{
    Handler *p1 = new ConcreateHandler1();
    Handler *p2 = new ConcreateHandler2(p1);

    p2->HandleRequset();

    delete p2;

    system("pause");

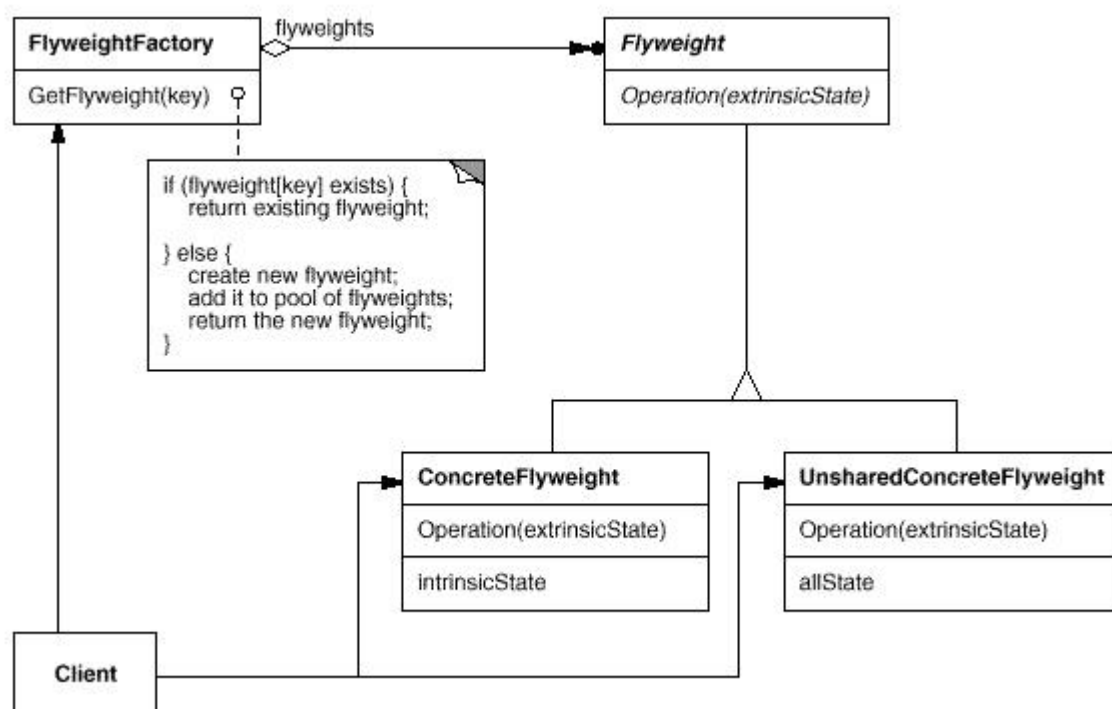
    return 0;
}

```

常见设计模式的解析和实现(C++)之十三-FlyWeight模式

作用：
运用共享技术有效地支持大量细粒度的对象。

UML 结构图：



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 聊天的时候很多时候你懒得回复又不得不回复的时候,一般会用一些客套的话语敷衍别人,如"呵呵","好的"等等之类的,这些简单的答复其实每个人都是提前定义好的,在使用的时候才调用出来.**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的对象,如果没有就生成之有的话就直接使用.因此,**Flyweight** 模式和 **Factory** 模式也经常混用.

实现：

需要说明的是下面的实现仅仅实现了对可共享对象的使用,非可共享对象的使用没有列出,因为这个不是 **Flyweight** 模式的重点.这里的实现要点是采用一个 **list** 链表来保存这些可以被共享的对象,需要使用的时候就到链表中查询是不是已经存在了,如果不存在就初始化一个,然后返回这个对象的指针.

1)Flyweight.h

```

*****
|
|   created:   2006/07/26
|   filename:  FlyWeight.h
|   author:    李创
|              http://www.cppblog.com/converse/
|
|   purpose:   FlyWeight 模式的演示代码
|
*****
*****/

#ifndef FLYWEIGHT_H
#define FLYWEIGHT_H

#include <string>
#include <list>

typedef std::string STATE;

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

    STATE GetIntrinsicState();
    virtual void Operation(STATE& ExtrinsicState) = 0;

protected:
    Flyweight(const STATE& state)
        :m_State(state)
    {
    }

private:
    STATE m_State;
};

class FlyweightFactory
{
public:
    FlyweightFactory(){}
    ~FlyweightFactory();
};
```

```

    Flyweight* GetFlyweight(const STATE& key);

private:
    std::list<Flyweight*>    m_listFlyweight;
};

class ConcreateFlyweight
: public Flyweight
{
public:
    ConcreateFlyweight(const STATE& state)
        : Flyweight(state)
    {
    }
    virtual ~ConcreateFlyweight(){}

    virtual void Operation(STATE& ExtrinsicState);
};

#endif

```

2)Flyweight.cpp

```

/*****
created:    2006/07/26
filename:   FlyWeight.cpp
author:     李创
           http://www.cppblog.com/converse/

purpose:    FlyWeight 模式的演示代码
*****/

#include "FlyWeight.h"
#include <iostream>

inline STATE Flyweight::GetIntrinsicState()
{
    return m_State;
}

FlyweightFactory::~FlyweightFactory()

```

```

{
    std::list<Flyweight*>::iterator iter1, iter2, temp;

    for (iter1 = m_listFlyweight.begin(), iter2 = m_listFlyweight.end();
        iter1 != iter2;
        )
    {
        temp = iter1;
        ++iter1;
        delete (*temp);
    }

    m_listFlyweight.clear();
}

Flyweight* FlyweightFactory::GetFlyweight(const STATE& key)
{
    std::list<Flyweight*>::iterator iter1, iter2;

    for (iter1 = m_listFlyweight.begin(), iter2 = m_listFlyweight.end();
        iter1 != iter2;
        ++iter1)
    {
        if ((*iter1)->GetIntrinsicState() == key)
        {
            std::cout << "The Flyweight:" << key << " already exists"<< std::endl;
            return (*iter1);
        }
    }

    std::cout << "Creating a new Flyweight:" << key << std::endl;
    Flyweight* flyweight = new ConcreateFlyweight(key);
    m_listFlyweight.push_back(flyweight);
}

void ConcreateFlyweight::Operation(STATE& ExtrinsicState)
{
}

```

3)Main.cpp

```

/*****
*****

```

```

created: 2006/07/26
filename: Main.cpp
author: 李创
http://www.cppblog.com/converse/

purpose: FlyWeight 模式的测试代码
*****

*****/

#include "FlyWeight.h"

int main()
{
    FlyweightFactory flyweightfactory;
    flyweightfactory.GetFlyweight("hello");
    flyweightfactory.GetFlyweight("world");
    flyweightfactory.GetFlyweight("hello");

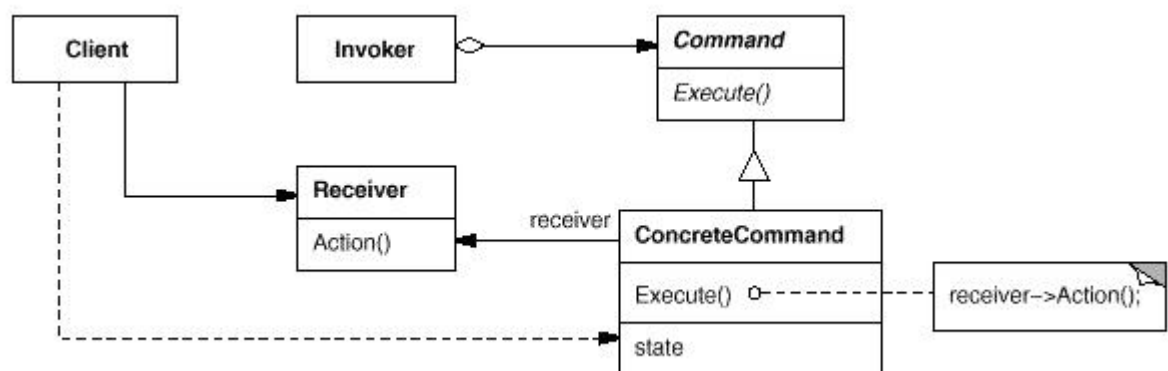
    system("pause");
    return 0;
}

```

常见设计模式的解析和实现(C++)之十四-Command模式

作用：
 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,以及支持可撤消的操作。

UML 结构图:



解析:

Command 模式的思想是把命令封装在一个类中,就是这里的 **Command** 基类,同时把接收对象也封装在一个类中就是这里的 **Receiver** 类中,由调用这个命令的类也就是这里的 **Invoker** 类来调用.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.

实现:

1)Command.h

```

/*****
|
|   created:   2006/08/04
|   filename:   Command.h
|   author:    李创
|              http://www.cppblog.com/converse/
|
|   purpose:   Command 模式的演示代码
*****/

#ifndef COMMAND_H
#define COMMAND_H

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

    virtual void Execute() = 0;
};

class Receiver
{
public:
    void Action();
};

class Invoker
{
public:
    Invoker(Command *pCommand);
    ~Invoker();
};

```

```

        void Invoke();
private:
    Command *m_pCommand;
};

class ConcreateComand
    : public Command
{
public:
    ConcreateComand(Receiver* pReceiver);
    virtual ~ConcreateComand();

    virtual void Execute();

private:
    Receiver* m_pReceiver;
};

#endif

```

2)Command.cpp

```

/*****
 *
 * created:    2006/08/04
 * filename:   Command.cpp
 * author:     李创
 *             http://www.cppblog.com/converse/
 *
 * purpose:    Command 模式的演示代码
 *****/

#include "Command.h"
#include <iostream>

void Receiver::Action()
{
    std::cout << "Receiver Action\n";
}

Invoker::Invoker(Command *pCommand)
    : m_pCommand(pCommand)

```

```

~Invoker()
{
}

Invoker::~Invoker()
{
    delete m_pCommand;
    m_pCommand = NULL;
}

void Invoker::Invoke()
{
    if (NULL != m_pCommand)
    {
        m_pCommand->Execute();
    }
}

ConcreateComand::ConcreateComand(Receiver* pReceiver)
    : m_pReceiver(pReceiver)
{
}

ConcreateComand::~ConcreateComand()
{
    delete m_pReceiver;
    m_pReceiver = NULL;
}

void ConcreateComand::Execute()
{
    if (NULL != m_pReceiver)
    {
        m_pReceiver->Action();
    }

    std::cout << "Execute by ConcreateComand\n";
}

```

3)Main.cpp

```

/*****
*****

```

```

created: 2006/08/04
filename: main.cpp
author: 李创
        http://www.cppblog.com/converse/

purpose: Command 模式的测试代码
*****
*****/

#include "Command.h"
#include <stdlib.h>

int main()
{
    Receiver* pReceiver = new Receiver();
    Command* pCommand = new ConcreateComand(pReceiver);
    Invoker* pInvoker = new Invoker(pCommand);

    pInvoker->Invoke();

    delete pInvoker;

    system("pause");

    return 0;
}

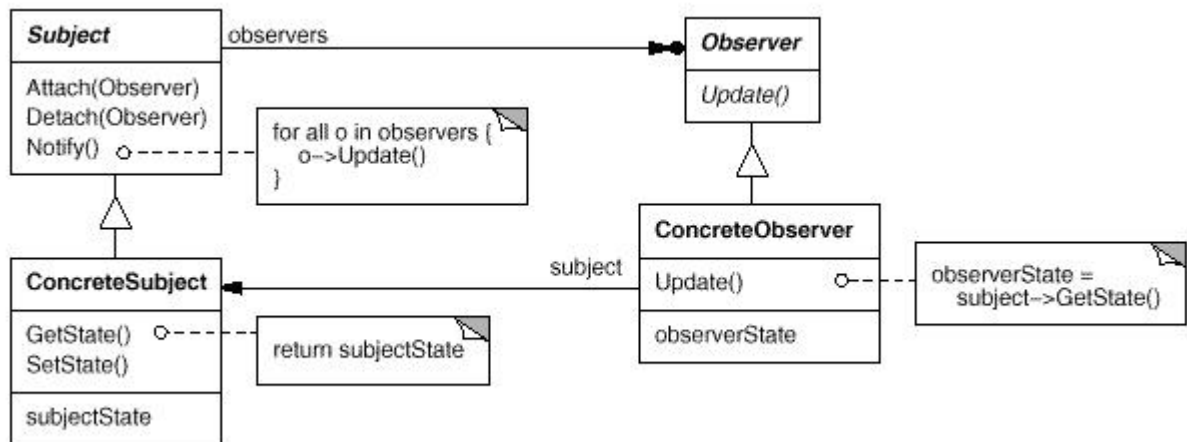
```

常见设计模式的解析和实现(C++)之十五-Observer 模式

作用:

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,所有依赖于它的对象都得到通知并被自动更新.

UML 结构图:



解析:

Observer 模式定义的是一种一对多的关系,这里的一就是图中的 **Subject** 类,而多则是 **Observer** 类,当 **Subject** 类的状态发生变化时通知与之对应的 **Observer** 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。**Observer** 模式的实现要点是,第一一般 **subject** 类都是采用链表等容器来存放 **Observer** 对象,第二抽取出 **Observer** 对象的一些公共的属性形成 **Observer** 基类,而 **Subject** 中保存的则是 **Observer** 类对象的指针,这样就使 **Subject** 和具体的 **Observer** 实现了解耦,也就是 **Subject** 不需要去关心到底是哪个 **Observer** 对象放进了自己的容器中.生活中有很多例子可以看做是 **Observer** 模式的运用,比方说,一个班有一个班主任(**Subject**),他管理手下的一帮学生(**Observer**),当班里有一些事情发生需要通知学生的时候,班主任要做的不是逐个学生挨个的通知而是把学生召集起来一起通知,实现了班主任和具体学生的关系解耦.

实现:

1)Observer.h

```

/*****
|
|   created:   2006/07/20
|   filename:  Observer.h
|   author:    李创
|              http://www.cppblog.com/converse/
|
|   purpose:   Observer 模式的演示代码
|
| *****/

#ifndef OBSERVER_H
#define OBSERVER_H

```

```

#include <list>

typedef int STATE;

class Observer;

// Subject 抽象基类,只需要知道 Observer 基类的声明就可以了
class Subject
{
public:
    Subject() : m_nSubjectState(-1){}
    virtual ~Subject();

    void Notify(); // 通知对象改变状态
    void Attach(Observer *pObserver); // 新增对象
    void Detach(Observer *pObserver); // 删除对象

    // 虚函数,提供默认的实现,派生类可以自己实现来覆盖基类的实现
    virtual void SetState(STATE nState); // 设置状态
    virtual STATE GetState(); // 得到状态

protected:
    STATE m_nSubjectState; // 模拟保存 Subject 状态的变量
    std::list<Observer*> m_ListObserver; // 保存 Observer 指针的链表
};

// Observer 抽象基类
class Observer
{
public:
    Observer() : m_nObserverState(-1){}
    virtual ~Observer(){}

    // 纯虚函数,各个派生类可能有不同的实现
    // 通知 Observer 状态发生了变化
    virtual void Update(Subject* pSubject) = 0;

protected:
    STATE m_nObserverState; // 模拟保存 Observer 状态的变量
};

// ConcreateSubject 类,派生在 Subject 类
class ConcreateSubject

```

```

        : public Subject
    {
    public:
        ConcreateSubject() : Subject(){}
        virtual ~ConcreateSubject(){}

        // 派生类自己实现来覆盖基类的实现
        virtual void SetState(STATE nState); // 设置状态
        virtual STATE GetState(); // 得到状态
    };

    // ConcreateObserver 类派生自 Observer
    class ConcreateObserver
        : public Observer
    {
    public:
        ConcreateObserver() : Observer(){}
        virtual ~ConcreateObserver(){}

        // 虚函数,实现基类提供的接口
        virtual void Update(Subject* pSubject);
    };

#endif

```

2)Observer.cpp

```

/* *****
 *
 * created: 2006/07/20
 * filename: Observer.cpp
 * author: 李创
 * http://www.cppblog.com/converse/
 *
 * purpose: Observer 模式的演示代码
 * *****
 */

#include "Observer.h"
#include <iostream>
#include <algorithm>

/* *****

```

Subject 类成员函数的实现

```
-----*/

void Subject::Attach(Observer *pObserver)
{
    std::cout << "Attach an Observer\n";

    m_ListObserver.push_back(pObserver);
}

void Subject::Detach(Observer *pObserver)
{
    std::list<Observer*>::iterator iter;
    iter = std::find(m_ListObserver.begin(), m_ListObserver.end(), pObserver);

    if (m_ListObserver.end() != iter)
    {
        m_ListObserver.erase(iter);
    }

    std::cout << "Detach an Observer\n";
}

void Subject::Notify()
{
    std::cout << "Notify Observers's State\n";

    std::list<Observer*>::iterator iter1, iter2;

    for (iter1 = m_ListObserver.begin(), iter2 = m_ListObserver.end();
        iter1 != iter2;
        ++iter1)
    {
        (*iter1)->Update(this);
    }
}

void Subject::SetState(STATE nState)
{
    std::cout << "SetState By Subject\n";
    m_nSubjectState = nState;
}
```

```

STATE Subject::GetState()
{
    std::cout << "GetState By Subject\n";
    return m_nSubjectState;
}

Subject::~~Subject()
{
    std::list<Observer*>::iterator iter1, iter2, temp;

    for (iter1 = m_ListObserver.begin(), iter2 = m_ListObserver.end();
        iter1 != iter2;
        )
    {
        temp = iter1;
        ++iter1;
        delete (*temp);
    }

    m_ListObserver.clear();
}

/* -----
   | ConcreateSubject 类成员函数的实现
   |
   | ----- */
void ConcreateSubject::SetState(STATE nState)
{
    std::cout << "SetState By ConcreateSubject\n";
    m_nSubjectState = nState;
}

STATE ConcreateSubject::GetState()
{
    std::cout << "GetState By ConcreateSubject\n";
    return m_nSubjectState;
}

/* -----
   | ConcreateObserver 类成员函数的实现
   |
   | ----- */
void ConcreateObserver::Update(Subject* pSubject)
{

```

```

    if (NULL == pSubject)
        return;

    m_nObserverState = pSubject->GetState();

    std::cout << "The ObeserverState is " << m_nObserverState << std::endl;
}

```

3)Main.cpp

```

//*****
*****
    created:   2006/07/21
    filename:  Main.cpp
    author:    李创
               http://www.cppblog.com/converse/

    purpose:   Observer 模式的测试代码
*****
//*****

#include "Observer.h"
#include <iostream>

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

    Subject* p = new ConcreateSubject;
    p->Attach(p1);
    p->Attach(p2);
    p->SetState(4);
    p->Notify();

    p->Detach(p1);
    p->SetState(10);
    p->Notify();

    delete p;

    system("pause");

    return 0;
}

```

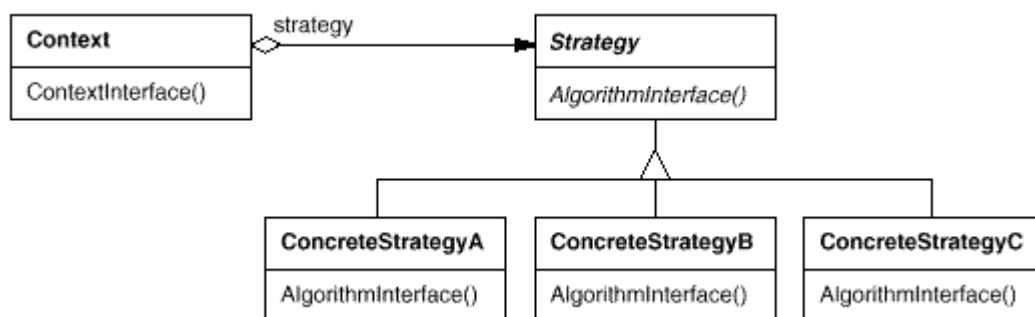
└}

常见设计模式的解析和实现(C++)之十六-Strategy 模式

作用:

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.本模式使得算法可独立于使用它的客户而变化.

UML 结构图:



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.

实现:

1)Strategy.h

```
1/*****
2*****
3|   created:   2006/08/06
4|   filename:   Strategy.h
5|   author:    李创
6|             http://www.cppblog.com/converse/
7|
8|   purpose:   Strategy 模式的演示代码
9|*****
10*****/
11
12#ifndef STRATEGY_H
```

```

#define STRATEGY_H

class Strategy;

class Context
{
public:
    Context(Strategy *pStrategy);
    ~Context();

    void ContextInterface();
private:
    Strategy* m_pStrategy;
};

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

    virtual void AlgorithmInterface() = 0;
};

class ConcreateStrategyA
    : public Strategy
{
public:
    virtual ~ConcreateStrategyA(){}

    virtual void AlgorithmInterface();
};

#endif

```

2)Strategy.cpp

```

/*****
*****
|   created:   2006/08/06
|   filename:   Strategy.cpp
|   author:    李创
|              http://www.cppblog.com/converse/
|
|   purpose:   Strategy 模式的演示代码

```

```

*****/

#include <iostream>
#include "Strategy.h"

Context::Context(Strategy *pStrategy)
    : m_pStrategy(pStrategy)
{

}

Context::~Context()
{
    delete m_pStrategy;
    m_pStrategy = NULL;
}

void Context::ContextInterface()
{
    if (NULL != m_pStrategy)
    {
        m_pStrategy->AlgorithmInterface();
    }
}

void ConcreateStrategyA::AlgorithmInterface()
{
    std::cout << "AlgorithmInterface Implemented by ConcreateStrategyA\n";
}

```

3)Main.cpp

```

*****/
*****
|   created:   2006/08/06
|   filename:  Main.cpp
|   author:    李创
|              http://www.cppblog.com/converse/
|
|   purpose:   Strategy 模式的测试代码
*****/
*****/

#include "Strategy.h"

```

```

int main()
{
    Strategy* pStrategy = new ConcreateStrategyA();
    Context* pContext = new Context(pStrategy);

    pContext->ContextInterface();

    delete pContext;

    return 0;
}

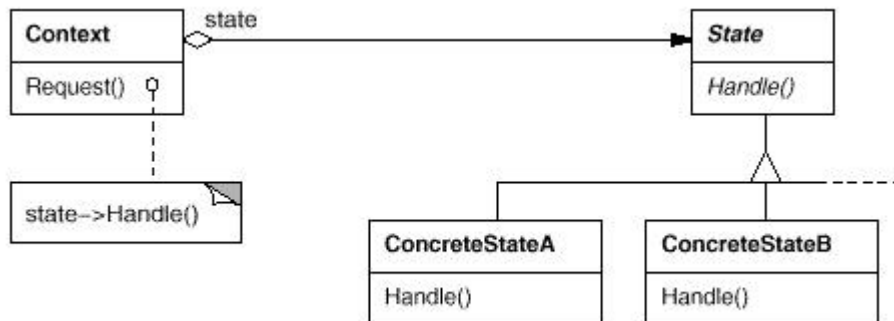
```

常见设计模式的解析和实现(C++)之十七-State 模式

作用:

允许一个对象在其内部状态改变时改变它的行为.

UML 结构图:



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.同时,**State** 模式和 **Strategy** 模式在图示上有很多相似的地方,需要说明的是两者的思想都是一致的,只不过封装的东西不同:**State** 模式封装的是不同的状态,而 **Strategy** 模式封装的是不同的算法.

实现:

1)State.h

```

*****
|
|   created:   2006/08/05
|   filename:   State.h
|   author:    李创
|              http://www.cppblog.com/converse/
|
|   purpose:   State 模式的演示代码
|
*****
*****/

#ifndef STATE_H
#define STATE_H

class State;

class Context
{
public:
    Context(State* pState);
    ~Context();
    void Request();
    void ChangeState(State *pState);

private:
    State *m_pState;
};

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

    virtual void Handle(Context* pContext) = 0;
};

class ConcreteStateA
    : public State
{
public:
    void Handle(Context* pContext);
};
```

```

class ConcreteStateB
: public State
{
public:
    void Handle(Context* pContext);
};

#endif

```

2)State.cpp

```

/*****
 *
 * created: 2006/08/05
 * filename: State.cpp
 * author: 李创
 * http://www.cppblog.com/converse/
 * purpose: State 模式的演示代码
 *****/

#include "State.h"
#include <iostream>

Context::Context(State* pState)
: m_pState(pState)
{
}

Context::~Context()
{
}

void Context::Request()
{
    if (NULL != m_pState)
    {
        m_pState->Handle(this);
    }
}

```

```

    }

    void Context::ChangeState(State *pState)
    {
        if (NULL != m_pState)
        {
            delete m_pState;
            m_pState = NULL;
        }

        m_pState = pState;
    }

    void ConcreateStateA::Handle(Context* pContext)
    {
        std::cout << "Handle by ConcreateStateA\n";

        if (NULL != pContext)
        {
            pContext->ChangeState(new ConcreateStateB());
        }
    }

    void ConcreateStateB::Handle(Context* pContext)
    {
        std::cout << "Handle by ConcreateStateB\n";

        if (NULL != pContext)
        {
            pContext->ChangeState(new ConcreateStateA());
        }
    }

```

3)Main.cpp

```

/*****
*****
    created: 2006/08/05
    filename: Main.cpp
    author: 李创
           http://www.cppblog.com/converse/

    purpose: State 模式的测试代码
*****/

```

```

*****/

#include "State.h"

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

    delete pContext;

    return 0;
}

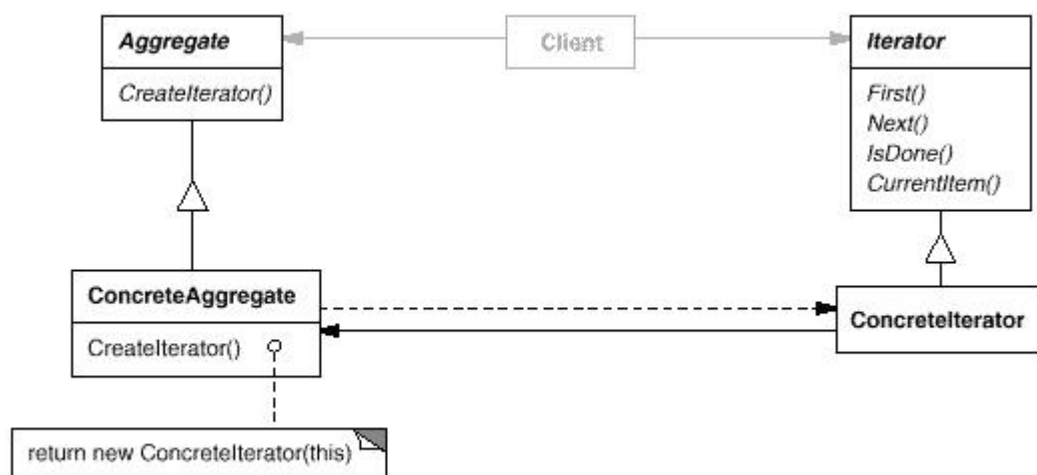
```

常见设计模式的解析和实现(C++)之十八-Iterator 模式

作用:

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.

UML 结构图:



解析:

Iterator 几乎是大部分人在初学 C++ 的时候就无意之中接触到的第一种设计模式,因为在 **STL** 之中,所有的容器类都有与之相关的迭代器.以前初学 **STL** 的时候,时常在看到讲述迭代器作用的时候是这么说的:提供一种方式,使得算法和容器可以独立的变化,而且在访问容器对象的时候不必暴露容器的内部细节,具体是怎么做到这一点的呢?在 **STL** 的实现中,所有的迭代器(**Iterator**)都必须遵照一套规范,这套规范里面定义了几种类型的名称,比如对象的名称,指向对象的指针的名称,指向对象的引用的名称....等等,当新生成一个容器的时候与之对应的 **Iterator** 都要遵守这个规范里面所定义的名称,这样在外部看来虽然里面的实现细节不一样,但是作用(也就是对外的表象)都是一样的,通过某个名称可以得到容器包含的对象,通过某个名称可以得到容器包含的对象的指针等等的.而且,采用这个模式把访问容器的重任都交给了具体的 **iterator** 类中.于是,在使用 **Iterator** 来访问容器对象的算法不需要知道需要处理的是什么容器,只需要遵守事先约定好的 **Iterator** 的规范就可以了;而对于各个容器类而言,不管内部的事先如何,是树还是链表还是数组,只需要对外的接口也遵守 **Iterator** 的标准,这样算法(**Iterator** 的使用者)和容器(**Iterator** 的提供者)就能很好的进行合作,而且不必关心对方是如何事先的,简而言之,**Iterator** 就是算法和容器之间的一座桥梁.

在下面的实现中,抽象基类 **Iterator** 可以看做是前面提到的 **Iterator** 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 **ConcreteIterator** 则是 **ConcreteAggregate** 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.

实现:

1)Iterator.h

```

/*****
|
|   created:   2006/08/04
|   filename:  Iterator.h
|   author:    李创
|              http://www.cppblog.com/converse/
|
|   purpose:   Iterator 模式的演示代码
|
| *****/

#ifndef ITERATOR_H
#define ITERATOR_H

typedef int DATA;

class Iterator;

// 容器的抽象基类
class Aggregate
```

```

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

    virtual Iterater* CreateIterater(Aggregate *pAggregate) = 0;
    virtual int GetSize() = 0;
    virtual DATA GetItem(int nIndex) = 0;
};

```

// 迭代器的抽象基类

class Iterater

```

{
public:
    virtual ~Iterater(){}

    virtual void First() = 0;
    virtual void Next() = 0;
    virtual bool IsDone() = 0;
    virtual DATA CurrentItem() = 0;

private:
};

```

// 一个具体的容器类,这里是用数组表示

class ConcreateAggregate

: public Aggregate

```

{
public:
    ConcreateAggregate(int nSize);
    virtual ~ConcreateAggregate();

    virtual Iterater* CreateIterater(Aggregate *pAggregate);
    virtual int GetSize();
    virtual DATA GetItem(int nIndex);

private:
    int m_nSize;
    DATA *m_pData;
};

```

// 访问 ConcreateAggregate 容器类的迭代器类

class ConcreateIterater

: public Iterater

```

{

```

```

public:
    ConcreateIterater(Aggregate* pAggregate);
    virtual ~ConcreateIterater(){}

    virtual void First();
    virtual void Next();
    virtual bool IsDone();
    virtual DATA CurrentItem();

private:
    Aggregate *m_pConcreateAggregate;
    int m_nIndex;
};

#endif

```

2) Iterator.cpp

```

/*****
created: 2006/08/04
filename: Iterator.cpp
author: 李创
        http://www.cppblog.com/converse/

purpose: Iterator 模式的演示代码
*****/

#include <iostream>
#include "Iterator.h"

ConcreateAggregate::ConcreateAggregate(int nSize)
: m_nSize(nSize)
, m_pData(NULL)
{
    m_pData = new DATA[m_nSize];

    for (int i = 0; i < nSize; ++i)
    {
        m_pData[i] = i;
    }
}

```

```

ConcreateAggregate::~ConcreateAggregate()
{
    delete [] m_pData;
    m_pData = NULL;
}

Iterator* ConcreateAggregate::CreateIterater(Aggregate *pAggregate)
{
    return new ConcreateIterater(this);
}

int ConcreateAggregate::GetSize()
{
    return m_nSize;
}

DATA ConcreateAggregate::GetItem(int nIndex)
{
    if (nIndex < m_nSize)
    {
        return m_pData[nIndex];
    }
    else
    {
        return -1;
    }
}

ConcreateIterater::ConcreateIterater(Aggregate* pAggregate)
    : m_pConcreateAggregate(pAggregate)
    , m_nIndex(0)
{
}

void ConcreateIterater::First()
{
    m_nIndex = 0;
}

void ConcreateIterater::Next()
{
    if (m_nIndex < m_pConcreateAggregate->GetSize())
    {

```

```

        ++m_nIndex;
    }
}

bool ConcreateIterater::IsDone()
{
    return m_nIndex == m_pConcreateAggregate->GetSize();
}

DATA ConcreateIterater::CurrentItem()
{
    return m_pConcreateAggregate->GetItem(m_nIndex);
}

```

3_Main.cpp

```

/*****
*****
    created:    2006/08/08
    filename:   Main.cpp
    author:     李创
               http://www.cppblog.com/converse/

    purpose:    Iterater 模式的演示代码
*****
*****/

#include "Iterator.h"
#include <iostream>

int main()
{
    Aggregate* pAggregate = new ConcreateAggregate(4);
    Iterater* pIterater = new ConcreateIterater(pAggregate);

    for (; false == pIterater->IsDone(); pIterater->Next())
    {
        std::cout << pIterater->CurrentItem() << std::endl;
    }

    return 0;
}

```

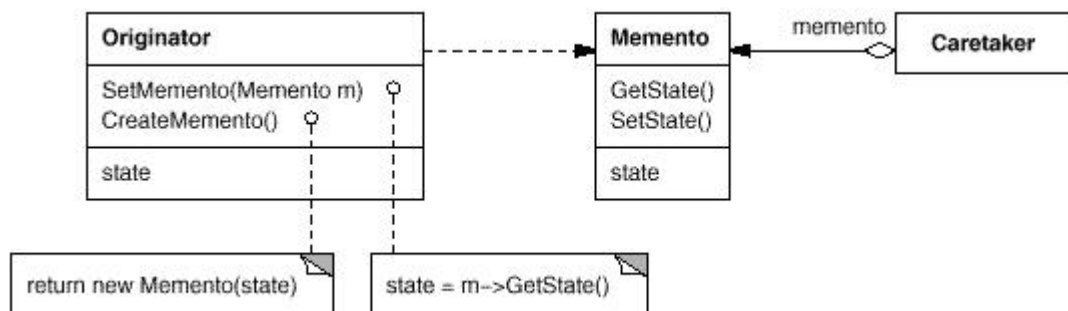
常见设计模式的解析和实现(C++)之十九-Memento

模式

作用:

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原先保存的状态.

UML 结构图:



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.

实现:

1)Memento.h

```

/*****
 *
 * created: 2006/08/09
 * filename: Memento.h
 * author: 李创
 * http://www.cppblog.com/converse/
 *
 * purpose: Memento 模式的演示代码
 *****/

#ifndef MEMENTO_H
```

```

#define MEMENTO_H

#include <string>

typedef std::string State;

class Memento;

class Originator
{
public:
    Originator(const State& rState);
    Originator();
    ~Originator();

    Memento* CreateMemento();
    void SetMemento(Memento* pMemento);
    State GetState();
    void SetState(const State& rState);
    void RestoreState(Memento* pMemento);
    void PrintState();

private:
    State m_State;
};

// 把 Memento 的接口函数都设置为私有的,而 Originator 是它的友元,
// 这样保证了只有 Originator 可以对其访问
class Memento
{
private:
    friend class Originator;
    Memento(const State& rState);
    void SetState(const State& rState);
    State GetState();

    State m_State;
};

#endif

```

2)Memento.cpp

```

/*****
|
|   created:   2006/08/09
|   filename:  Memento.cpp
|   author:    李创
|              http://www.cppblog.com/converse/
|
|   purpose:   Memento 模式的演示代码
| *****/
*****/

#include "Memento.h"
#include <iostream>

Originator::Originator()
{

}

Originator::Originator(const State& rState)
    : m_State(rState)
{

}

Originator::~Originator()
{

}

State Originator::GetState()
{
    return m_State;
}

void Originator::SetState(const State& rState)
{
    m_State = rState;
}

Memento* Originator::CreateMemento()
{
    return new Memento(m_State);
}

```

```

void Originator::RestoreState(Memento* pMemento)
{
    if (NULL != pMemento)
    {
        m_State = pMemento->GetState();
    }
}

void Originator::PrintState()
{
    std::cout << "State = " << m_State << std::endl;
}

Memento::Memento(const State& rState)
: m_State(rState)
{
}

State Memento::GetState()
{
    return m_State;
}

void Memento::SetState(const State& rState)
{
    m_State = rState;
}

```

3)Main.cpp

```

/*****
 *
 * created: 2006/08/09
 * filename: Main.cpp
 * author: 李创
 * http://www.cppblog.com/converse/
 *
 * purpose: Memento 模式的测试代码
 *****/

#include "Memento.h"

```

```

int main()
{
    // 创建一个原发器
    Originator* pOriginator = new Originator("old state");
    pOriginator->PrintState();

    // 创建一个备忘录存放这个原发器的状态
    Memento *pMemento = pOriginator->CreateMemento();

    // 更改原发器的状态
    pOriginator->SetState("new state");
    pOriginator->PrintState();

    // 通过备忘录把原发器的状态还原到之前的状态
    pOriginator->RestoreState(pMemento);
    pOriginator->PrintState();

    delete pOriginator;
    delete pMemento;

    return 0;
}

```

常见设计模式的解析和实现(C++)之二十-Visitor 模式

作用:

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.

UML 结构图:


```

author:      李创
             http://www.cppblog.com/converse/

purpose:     Visitor 模式的演示代码
*****
*****/

#ifndef VISITOR_H
#define VISITOR_H

class Visitor;

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

    virtual void Accept(Visitor &rVisitor) = 0;

protected:
    Element(){}
};

class ConcreateElementA
    : public Element
{
public:
    virtual ~ConcreateElementA() {}

    virtual void Accept(Visitor &rVisitor);
};

class ConcreateElementB
    : public Element
{
public:
    virtual ~ConcreateElementB() {}

    virtual void Accept(Visitor &rVisitor);
};

class Visitor
{
public:

```

```

virtual ~Visitor(){}

    virtual void VisitConcreateElementA(ConcreateElementA *pConcreateElement
A) = 0;
    virtual void VisitConcreateElementB(ConcreateElementB *pConcreateElement
B) = 0;

protected:
virtual Visitor(){}
};

class ConcreateVisitorA
    : public Visitor
{
public:
    virtual ~ConcreateVisitorA(){}

    virtual void VisitConcreateElementA(ConcreateElementA *pConcreateElement
A);
    virtual void VisitConcreateElementB(ConcreateElementB *pConcreateElement
B);
};

class ConcreateVisitorB
    : public Visitor
{
public:
    virtual ~ConcreateVisitorB(){}

    virtual void VisitConcreateElementA(ConcreateElementA *pConcreateElement
A);
    virtual void VisitConcreateElementB(ConcreateElementB *pConcreateElement
B);
};

#endif

```

2)Visitor.cpp

```

/*****
*****
|   created:   2006/08/09
|   filename:   Visitor.cpp
|   author:    李创

```

<http://www.cppblog.com/converse/>

purpose: Visitor 模式的演示代码

```
*****  
*****/
```

```
#include "Visitor.h"  
#include <iostream>
```

```
void ConcreateElementA::Accept(Visitor &rVisitor)
```

```
{  
    rVisitor.VisitConcreateElementA(this);  
}
```

```
void ConcreateElementB::Accept(Visitor &rVisitor)
```

```
{  
    rVisitor.VisitConcreateElementB(this);  
}
```

```
void ConcreateVisitorA::VisitConcreateElementA(ConcreateElementA *pConcreat  
eElementA)
```

```
{  
    std::cout << "VisitConcreateElementA By ConcreateVisitorA\n";  
}
```

```
void ConcreateVisitorA::VisitConcreateElementB(ConcreateElementB *pConcreat  
eElementA)
```

```
{  
    std::cout << "VisitConcreateElementB By ConcreateVisitorA\n";  
}
```

```
void ConcreateVisitorB::VisitConcreateElementA(ConcreateElementA *pConcreat  
eElementA)
```

```
{  
    std::cout << "VisitConcreateElementA By ConcreateVisitorB\n";  
}
```

```
void ConcreateVisitorB::VisitConcreateElementB(ConcreateElementB *pConcreat  
eElementA)
```

```
{  
    std::cout << "VisitConcreateElementB By ConcreateVisitorB\n";  
}
```

3)Main.cpp

```

/*****
 *
 * created: 2006/08/09
 * filename: Main.cpp
 * author: 李创
 *
 * http://www.cppblog.com/converse/
 *
 * purpose: Visitor 模式的测试代码
 *****/

#include "Visitor.h"

int main()
{
    Visitor *pVisitorA = new ConcreateVisitorA();
    Element *pElement = new ConcreateElementA();

    pElement->Accept(*pVisitorA);

    delete pElement;
    delete pVisitorA;

    return 0;
}
```

常见设计模式的解析和实现(C++)之二十一-完结篇

一个月下来,把常见的 20 个设计模式好好复习并且逐个用 C++ 实现了一遍,收获还是很大的,很多东西看上去明白了但是真正动手去做的时候发现其实还是不明白--我深知这个道理,于是我不敢怠慢,不敢写什么所谓的解释原理的伪代码,不敢说所谓的"知道原理就可以了"....因为我知道,我还还没有资格说这个话,至少对于设计模式而言我还是一个初学者,唯有踏实和实干才能慢慢的掌握到知识.

在我学习设计模式的过程中,觉得造成理解困难的主要是以下几点,谈一下自己的体会,希望对他人有帮助,不要走上我的老路上,毕竟我花了 N 长的时间才敢号称自己入门了~~!!-_-:

1)Gof 并不适合于初学者.初学设计模式的一般都是从 Gof 入门开始学习

的,不幸的是,这不是一本好的教科书,而把这本书称为一本奠定了设计模式理论基础的开山之作也许好一些,它把这些散落在各个设计中的常见模式收集起来,从此开始有了一个名词叫做"Design Pattern".说这本书不是一本好的教科书主要是以下的几个原因:**a)**对设计模式或者说面向对象里面的一些原则性的东西解释的不够多不够彻底,比如"面向接口编程而不是对实现编程","优先采用组合而不是继承"等等,以至于后面看到各个模式的实现的时候很多模式看起来很相似却找不到区别和共性的地方.**b)**对各个模式的解释或者举出来的例子不是特别的好,大部分都是为了讲解模式而讲解,没有加入前面提到过的一些基本原则的考量在里面,也就是说:原理性的东西和实现(各个设计模式)脱节.

2)初学者对语言或者说一些概念理解的不好.拿 C++ 来说,为了做到面向对象需要提供的语言上的支持有继承,多态,封装,虚函数,抽象等等,我以前初学 C++ 的时候,只为了学这些概念而去学习,不知道为什么提供这些特性,这也是造成我走弯路的重要原因之一.当然,指望一个初学者在初学语言的时候就知道 **why** 是一件很困难的事情,也许结合着对设计模式的理解可以帮助你消化这些概念(我就是这样的).

3)看不懂 UML 结构图和时序图,UML 图解释的类与类之间的关系,时序图解释的是各个对象的实现方式,两者结合在一起看才能加深对设计模式的理解,事实上,我现在已经可以做到仅仅看这两个图示就掌握一个模式的原理和实现了.

4)写的代码和参与过的项目不够多.设计模式和很多东西的产生过程都是一样的,首先人们遇到了问题,然后很多人解决了这个问题,于是渐渐的有人出来总结出解决这些问题所要遵守的一些原理和常用方法(我们称之为"模式"),久而久之就形成了一个理论或者说一个学科.而后人在讲述这些理论的时候大都是照本宣科,这对于计算机这样一个强调实践的学科或者说对于设计模式这样一个理论而言要理解起来是很困难的.前人在提出这些理论的时候一些考量,权衡等等只有在你自己遇到了这些问题的时候才能慢慢的体会.有一种说法是,没有写上 **10W** 行代码不要空谈什么设计模式大概就是这个意思吧.

综上所述,造成初学者学习设计模式困难的原因,一个是对基本的原则理解的不够透彻,一个选的入门教材不合理,还有一个就是对各个模式的表述不明白,再次是实践不够多.

有几本书籍,我看过,我想可以谈谈我的看法.

第一本,<<敏捷软件开发:原则,模式与实践>>,这本书对于设计模式最大的贡献在于专门有几个章节讲述了面向对象的几个原则,比如 **Likov** 原则,开放封闭原则等等的,这几个章节在我学习设计模式的过程中起了关键的作用,因为当我理解了这些原则之后开始慢慢明白为什么要有纯虚函数提供接口,为什么要有抽象基类,为什么要封装....我开始用这些原则去理解各个设计模式,开始慢慢体会各个模式的区别和共性.

另外看过的两本书,我觉得性质都一样,如果你缺钱,任选其一吧.第一本是<<设计模式精解>>,第二本是<<深入浅出设计模式>>,都是我花上几个晚上

就可以看完的书.这两本的立足点都是以生动的例子结合面向对象的基本原理来讲解模式,我更喜欢前者一些(后者太贵,要不是打 5 折我才不买呐:)

其次,要多接触项目或者可以找一些好的代码来看看,自己也多写一些代码.基本上,只要是用面向对象的语言开发的项目,里面没有几个模式的运用是不可能的了.因此,要戒除那些一开始接触设计模式就想整明白的幻想,因为要真正的理解需要很多的实践,同样的一时半会理解不了的也不必气馁(GOF 的 E 文版我看了好多遍了:),坚信自己多实践一定可以慢慢的悟道的.

关于设计模式的一个疑问:非面向对象语言中有没有所谓的"设计模式"?设计模式最初的定义是解决一些问题的惯用方法(大意如此),并没有明确的说要支持某种特性的语言.我用纯 C 开发的项目实在是有限,平时也只是自己作一些小东西玩玩,没有做过任何一个上万行的纯 C 开发的项目,所以一直对这个问题抱有疑问~~**anyway**,有问题是好事,说明我在思考~~把这个问题放在这里,以后慢慢实践之琢磨之~~

博君一笑.

关于设计模式,还有一篇有意思的文章--<<追 MM 与设计模式>>,这篇文章可谓是"寓教于乐"的典范,讲述了 23 个模式在日常生活中的原型,虽然没有具体讲述如何实现,但是对于理解各个设计模式的运用场合还是很有帮助的.相信对设计模式已经有了一定了解的人看了这篇文章都会会心一笑:),作者和出处已经不详了.

追 MM 与设计模式

作者: 佚名 来自: CSDN

在 CSDN 看见了这篇文章,作者以轻松的语言比喻了 java 的 32 种模式,有很好的启发作用,但可惜没有给出具体的意思,我就在后边加上了:) 这些都是最简单的介绍,要学习的话建议你看一下《java 与模式》这本书。

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**

工厂模式: 客户类和工厂类分开。消费者任何时候需要某种产品,只需向工厂请求即可。消费者无须修改就可以接纳新产品。缺点是当产品修改时,工厂类也要做相应的修改。如: 如何创建及如何向客户端提供。

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)

建造模式：将产品的内部表象和产品的生成过程分割开来，从而使一个建造过程生成具有不同的内部表象的产品对象。建造模式使得产品内部表象可以独立的变化，客户不必知道产品内部组成的细节。建造模式可以强制实行一种分步骤进行的建造过程。

3 、FACTORY METHOD—请 MM 去麦当劳吃汉堡，不同的 MM 有不同的口味，要每个都记住是一件烦人的事情，我一般采用 **Factory Method** 模式，带着 MM 到服务员那儿，说“要一个汉堡”，具体要什么样的汉堡呢，让 MM 直接跟服务员说就行了。

工厂方法模式：核心工厂类不再负责所有产品的创建，而是将具体创建的工作交给子类去做，成为一个抽象工厂角色，仅负责给出具体工厂类必须实现的接口，而不接触哪一个产品类应当被实例化这种细节。

4 、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）

原始模型模式：通过给出一个原型对象来指明所要创建的对象类型，然后用复制这个原型对象的方法创建出更多同类型的对象。原始模型模式允许动态的增加或减少产品类，产品类不需要非得有任何事先确定的等级结构，原始模型模式适用于任何的等级结构。缺点是每一个类都必须配备一个克隆方法。

5 、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 **Singleton**，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)

单例模式：单例模式确保某一个类只有一个实例，而且自行实例化并向整个系统提供这个实例单例模式。单例模式只应在有真正的“单一实例”的需求时才可使用。

结构型模式

6 、ADAPTER—在朋友聚会上碰到了个美女 **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(也不知道他会不会耍我)

适配器（变压器）模式：把一个类的接口变换成客户端所期待的另一种接口，从而使原本因接口原因不匹配而无法一起工作的两个类能够一起工作。适配类可以根据参数返还一个合适的实例给客户端。

7 、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

桥梁模式：将抽象化与实现化脱耦，使得二者可以独立的变化，也就是说将他们之间的强关联变成弱关联，也就是指在一个软件系统的抽象化和实现化之间使用组合 / 聚合关系而不是继承关系，从而使两者可以独立的变化。

8 、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？

合成模式：合成模式将对象组织到树结构中，可以用来描述整体与部分的关系。合成模式就是一个处理对象的树结构的模式。合成模式把部分与整体的关系用树结构表示出来。合成模式使得客户端把一个个单独的成分对象和由他们复合而成的合成对象同等看待。

9 、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 **Mik e** 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？

装饰模式：装饰模式以对客户端透明的方式扩展对象的功能，是继承关系的一个替代方案，提供比继承更多的灵活性。动态给一个对象增加功能，这些功能可以再动态的撤消。增加由一些基本功能的排列组合而产生的非常大量的功能。

10 、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。

门面模式：外部与一个子系统的通信必须通过一个统一的面对象进行。门面模式提供一个高层次的接口，使得子系统更易于使用。每一个子系统只有一个门面类，而且此门面类只有一个实例，也就是说它是一个单例模式。但整个系统可以有多个门面类。

11 、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，再也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。

享元模式：**FLYWEIGHT** 在拳击比赛中指最轻量级。享元模式以共享的方式高效的支持大量的细粒度对象。享元模式能做到共享的关键是区分内蕴状态和外蕴状态。内蕴状态存储在享元内部，不会随环境的改变而有所不同。外蕴状态是随环境的改变而改变的。外蕴状态不能影响内蕴状态，它们是相互独立的。将可以共享的状态和不可以共享的状态从常规类中区分开来，将不可以共享的状态从类里剔除出去。客户端不可以直接创建被共享的对象，而应当使用一个工厂对象负责创建被共享的对象。享元模式大幅度的降低内存中对象的数量。

12 、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。

代理模式：代理模式给某一个对象提供一个代理对象，并由代理对象控制对源对象的引用。代理就是一个人或一个机构代表另一个人或者一个机构采取行动。某些情况下，客户不想或者不能够直接引用一个对象，代理对象可以在客户和目标对象直接起到中介的作用。客户端分辨不出代理主题对象与真实主题对象。代理模式可以并不知道真正的被代理对象，而仅仅持有一个被代理对象的接口，这时候代理对象不能够创建被代理对象，被代理对象必须有系统的其他角色代为创建并传入。

行为模式

13 、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 ！

责任链模式：在责任链模式中，很多对象由每一个对象对其下家的引用而接 起来形成一条链。请求在这个链上传递，直到链上的某一个对象决定处理此请求。客户并不知道链上的哪一个对象最终处理这个请求，系统可以在不影响客户端的情况下动态的重新组织链和分配责任。处理者有两个选择：承担责任或者把责任推给下家。一个请求可以最终不被任何接收端对象所接受。

14 、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。”，： - (

命令模式：命令模式把一个请求或者操作封装到一个对象中。命令模式把发出命令的责任和执行命令的责任分割开，委派给不同的对象。命令模式允许请求的一方和发送的一方独立开来，使得请求的一方不必知道接收请求的一方的接口，更不必知道请求是怎么被接收，以及操作是否执行，何时被执行以及是怎么被执行的。系统支持命令的撤消。

15 、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。

解释器模式：给定一个语言后，解释器模式可以定义出其文法的一种表示，并同时提供一个解释器。客户端可以使用这个解释器来解释这个语言中的句子。解释器模式将描述怎样在有了一个简单的文法后，使用模式设计解释这些语句。在解释器模式里面提到的语言是指任何解释器对象能够解释的任何组合。在解释器模式中需要定义一个代表文法的命令类的等级结构，也就是一系列的组合规则。每一个命令对象都有一个解释方法，代表对命令对象的解释。命令对象的等级结构中的对象的任何排列组合都是一个语言。

16 、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？”

.....

迭代子模式: 迭代子模式可以顺序访问一个聚集中的元素而不必暴露聚集的内部表象。多个对象聚在一起形成的总体称之为聚集，聚集对象是能够包容一组对象的容器对象。迭代子模式将迭代逻辑封装到一个独立的子对象中，从而与聚集本身隔开。迭代子模式简化了聚集的界面。每一个聚集对象都可以有一个或一个以上的迭代子对象，每一个迭代子的迭代状态可以是彼此独立的。迭代算法可以独立于聚集角色变化。

17 、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。

调停者模式: 调停者模式包装了一系列对象相互作用的方式，使得这些对象不必相互明显作用。从而使他们可以松散耦合。当某些对象之间的作用发生改变时，不会立即影响其他的一些对象之间的作用。保证这些作用可以彼此独立的变化。调停者模式将多对多的相互作用转化为一对多的相互作用。调停者模式将对象的行为和协作抽象化，把对象在小尺度的行为上与其他对象的相互作用分开处理。

18 、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。

备忘录模式: 备忘录对象是一个用来存储另外一个对象内部状态的快照的对象。备忘录模式的用意是在不破坏封装的条件下，将一个对象的状态捉住，并外部化，存储起来，从而可以在将来合适的时候把这个对象还原到存储起来的状态。

19 、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

观察者模式：观察者模式定义了一种一对多的依赖关系，让多个观察者对象同时监听某一个主题对象。这个主题对象在状态上发生变化时，会通知所有观察者对象，使他们能够自动更新自己。

20 、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？看完电影再去酒吧怎么样？”，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。

状态模式：状态模式允许一个对象在其内部状态改变的时候改变行为。这个对象看上去象是改变了它的类一样。状态模式把所研究的对象的行为包装在不同的状态对象里，每一个状态对象都属于一个抽象状态类的一个子类。状态模式的意图是让一个对象在其内部状态改变的时候，其行为也随之改变。状态模式需要对每一个系统可能取得的状态创立一个状态类的子类。当系统的状态变化时，系统便改变所选的子类。

21 、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，单目的都是为了得到 MM 的芳心，我的追 M 锦囊中有好多 Strategy 哦。

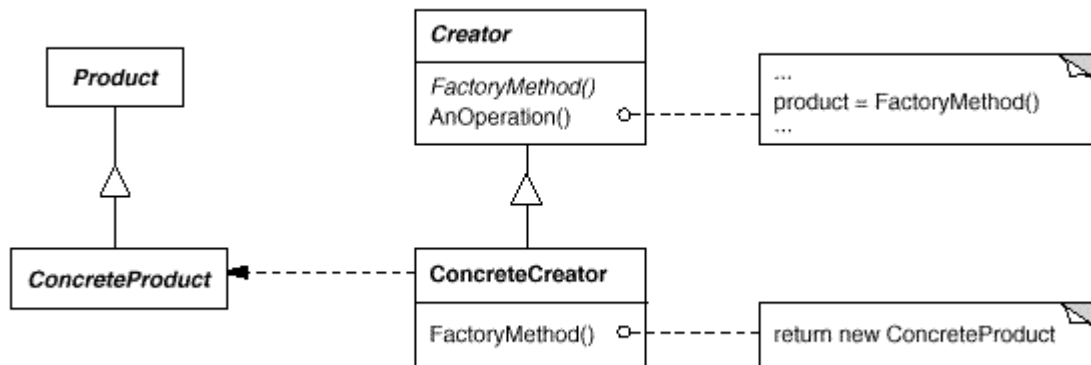
策略模式：策略模式针对一组算法，将每一个算法封装到具有共同接口的独立的类中，从而使得它们可以相互替换。策略模式使得算法可以在不影响到客户端的情况下发生变化。策略模式把行为和环境分开。环境类负责维持和查询行为类，各种算法在具体的策略类中提供。由于算法和环境独立开来，算法的增减，修改都不会影响到环境和客户端。

22 、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)；

模板方法模式：模板方法模式准备一个抽象类，将部分逻辑以具体方法以及具体构造子的形式实现，然后声明一些抽象方法来迫使子类实现剩余的逻辑。不同的子类可以以不同的方式实现这些抽象方法，从而对剩余的逻辑有不同的实现。先制定一个顶级逻辑框架，而将逻辑的细节留给具体的子类去实现。

23 、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；

访问者模式：访问者模式的目的是封装一些施加于某种数据结构元素之上的操作。一旦这些操作需要修改的话，接受这个操作的数据结构可以保持不变。访问者模式适用于数据结构相对未定的系统，它把数据结构和作用于结构上的操作之间的耦合解脱开，使得操作集合可以相对自由的演化。访问者模式使得增加新的操作变的很容易，就是增加一个新的访问者类。访问者模式将有关的行为集中到一个访问者对象中，而不是分散到一个个的节点类中。当使用访问者模式时，要将尽可能多的对象浏览逻辑放在访问者类中，而不是放到它的子类中。访问者模式可以跨过几个类的等级结构访问属于不同的等级结构的成员类。



至此,这一个系列的文章就要告一个段落了.写罢这些文章,我也可以正式踏入设计模式入门者的行列了,希望这些文章对他人能有所帮助.