# 状态变化模式

> - 在组件构建过程中，某些**对象的状态经常面临变化**，如何对这些变化进行有效的管理？同时又维持高层模块的稳定？“状态变化？模式为这一问题提供了一种解决方案。
> - 典型模式
>   - State
>   - Memento

# State

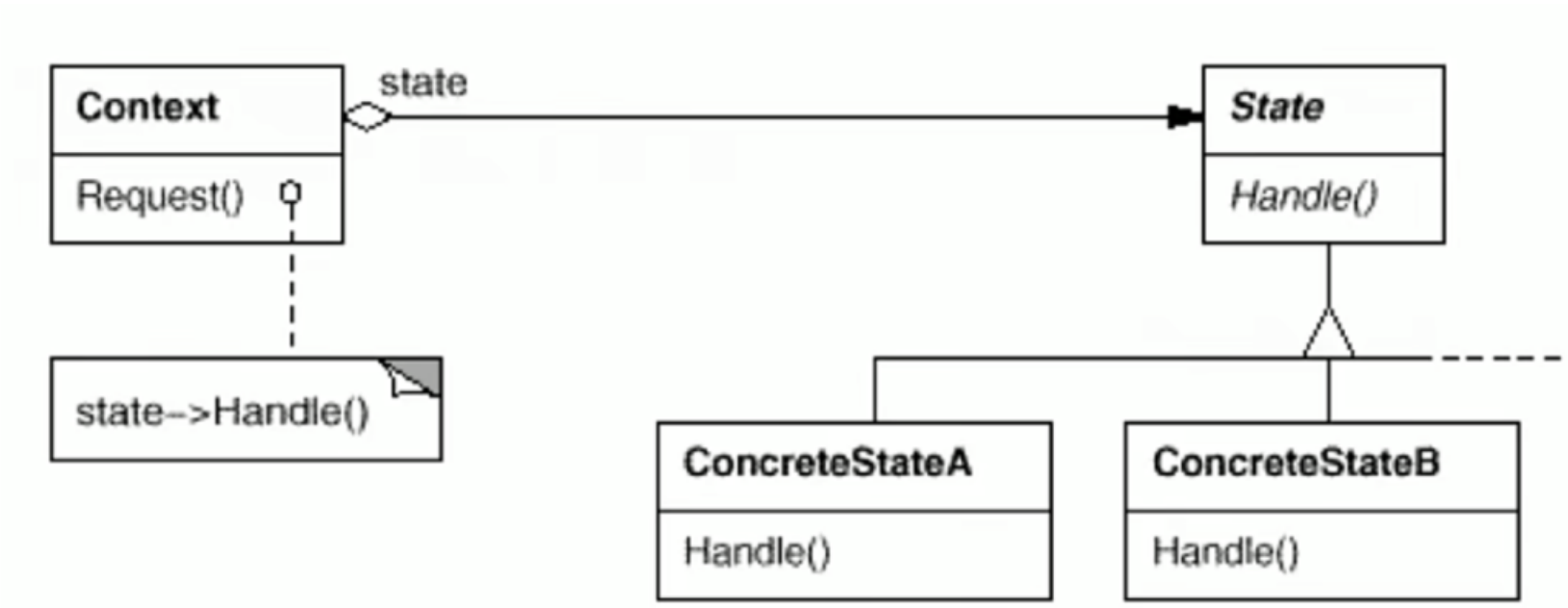
## 动机

> 在软件构建过程中，某些**对象的状态如果发生改变**，其**行为也会随之而发生变化**，比如文档处于只读状态，其支持的行为和读写状态支持的行为就可能完全不同。
>
> 如何在**运行时根据对象的状态来透明地更改对象的行为**？而不会为对象操作和状态转化之间引入紧耦合？

## 模式定义

> 允许一个对象在其内部状态改变时改变它的行为。从而使对象看起来似乎修改了其行为。

## 结构




`Context` 和 `State` 稳定 `ConcreteState` 变化

## 代码

```
class NetworkState{

public:
    NetworkState* pNext;
    virtual void Operation1()=0;
    virtual void Operation2()=0;
    virtual void Operation3()=0;

    virtual ~NetworkState(){}
};


class OpenState :public NetworkState{

    static NetworkState* m_instance;
public:
    static NetworkState* getInstance(){
        if (m_instance == nullptr) {
            m_instance = new OpenState();
        }
        return m_instance;
    }

    void Operation1(){

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

```
        pNext = CloseState::getInstance();
    }

    void Operation2(){

        //.........
        pNext = ConnectState::getInstance();
    }

    void Operation3(){

        //$$$$$$$$$$
        pNext = OpenState::getInstance();
    }


};

class CloseState:public NetworkState{ }
// ...


class NetworkProcessor{

    NetworkState* pState;

public:

    NetworkProcessor(NetworkState* pState){

        this->pState = pState;
    }

    void Operation1(){
        // ...
        pState->Operation1();
        pState = pState->pNext;
        // ...
    }

    void Operation2(){
        // ...
        pState->Operation2();
        pState = pState->pNext;
        // ...
    }

    void Operation3(){
        // ...
        pState->Operation3();
        pState = pState->pNext;
        // ...
    }

};
```

### 总结

- 和策略模式有点相似，会出现`if-else`
- `State` 模式将所有 **与一个特定状态相关的行为** 都放入一个`State`的子类对象中，在对象状态切换时，切换相应的对象；但同时 **维持`State`的接口**，这样实现了具体操作与状态转换之间的解耦。
- 为 **不同的状态引入不同的对象** 使得状态转换变得更加明确，而且可以保证不会出现状态不一致的情况，因为转换是原子性的—即要么彻底转换过来，要么不转换。

## Memento

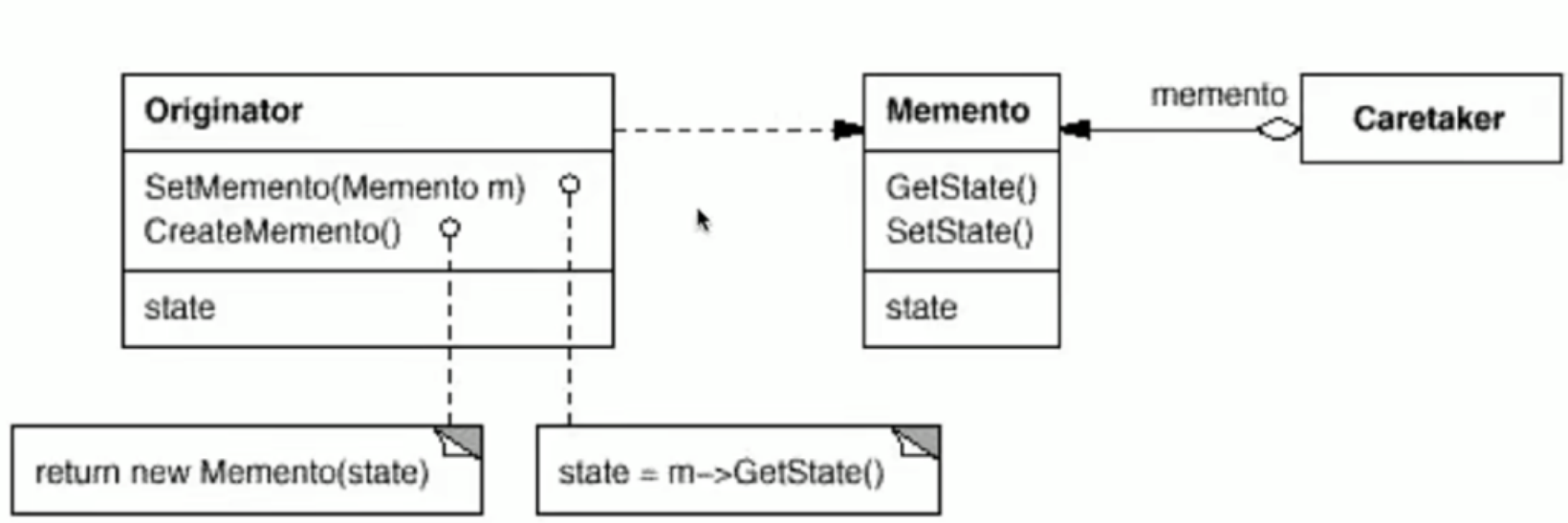
### 动机

> 在软件构建过程中，某些对象的状态在转换过程中，可能由于某种需要，要求程序能够回溯到对象之前处于某个点时的状态。如果使用一些公有接口来让其他对象得到对象的状态，便会暴露对象的细节实现。 如何实现对象状态的良好保存于恢复？但同时又不会因此而破坏对象本身的封装性。

## 模式定义

> 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。这样以后就可以将该对象**恢复到原先保存的状态**。

## 结构




## 代码

```
class Memento
{
    string state;
    //..
public:
    Memento(const string & s) : state(s) {}
    string getState() const { return state; }
    void setState(const string & s) { state = s; }
};



class Originator
{
    string state;
    //....
public:
    Originator() {}
    Memento createMomento() {
        Memento m(state);
        return m;
    }
    void setMomento(const Memento & m) {
        state = m.getState();
    }
};



int main()
{
    Originator orginator;

    //捕获对象状态，存储到备忘录
    Memento mem = orginator.createMomento();

    // ... 改变orginator状态

    //从备忘录中恢复
    orginator.setMomento(mem);
}
```

## 总结

- 备忘录（Memento）存储原发器（Originator）对象的内部状态，在需要时恢复原发器状态。
- Memento 模式的核心是**信息隐藏**，即Originator需要向外隐藏信息，保持其封装性。但同时又需要将状态保持到外界（Memento）。
- 由于现代语言运行时（如C#、Java等）都具有相当的**对象序列化**支持，因此往往采用效率较高、又较容易正确实现的序列化方案来实现Memento模式。

# 数据结构模式

> 常常有一些组件在内部具有特定的数据结构，如果让客户程序依赖这些特定的数据结构，将极大的破坏组件的复用。这时候，将这些特定数据结构封装在内部，在外部提供统一的接口，来实现与特定数据结构无关的访问，是一种行之有效的解决方案。
>
> 典型模式：
>
> - Composite
> - Iterator
> - Chain of Responsibility

# Composite

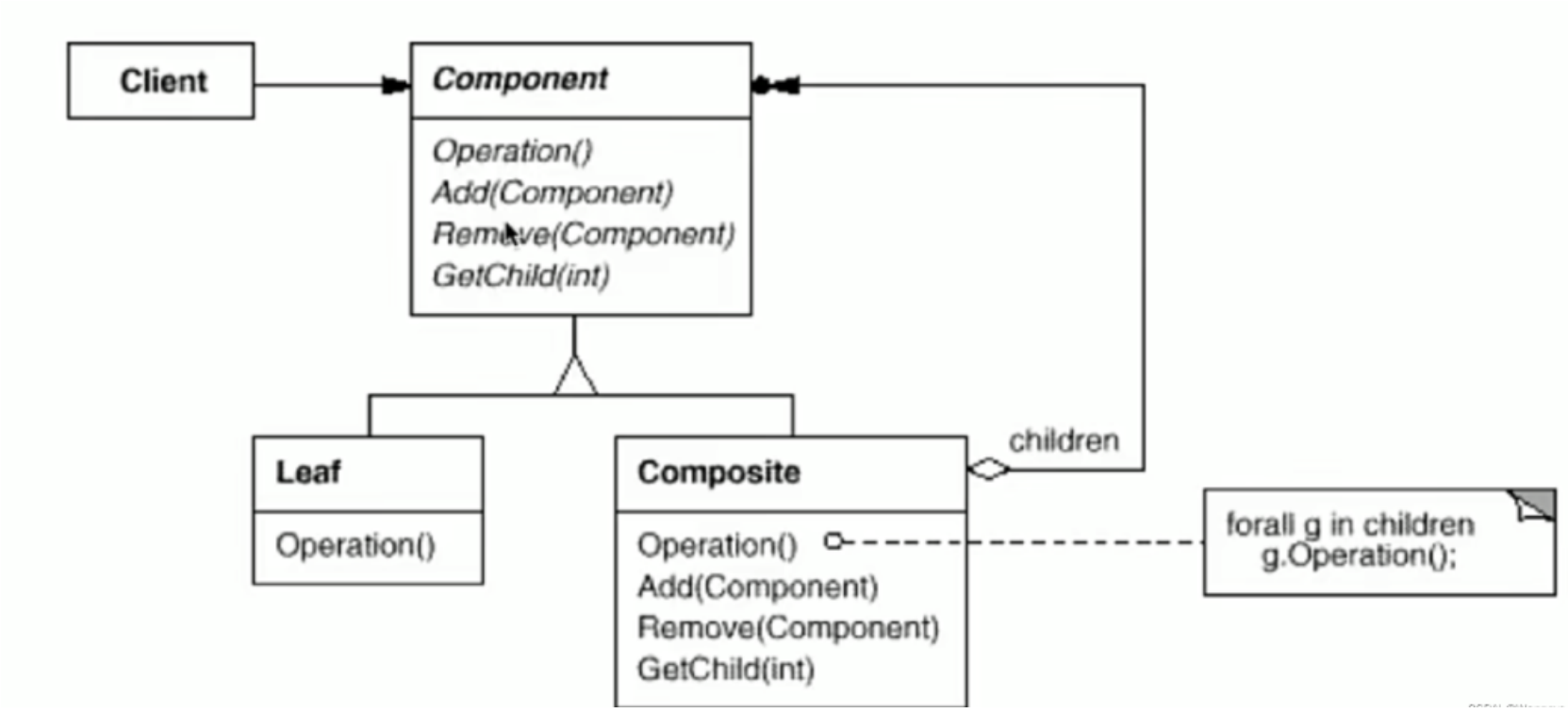
## 动机

> 在软件在某些情况下，客户代码**过多地依赖于对象容器复杂的内部实现结构**，对象容器内部实现结构（而非抽象接口）的变化将引起客户代码的频繁变化，带来代码的维护性、扩展性等弊端。
>
> 如何将“客户代码与复杂的对象容器结构”解耦？让对象容器自己来实现自身的复杂结构，从而使得客户代码就像处理简单对象一样来处理复杂的对象容器。

## 模式定义

> 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**（稳定）。

## 结构




## 代码

```
#include <iostream>
#include <list>
#include <string>
#include <algorithm>

using namespace std;

class Component
{
public:
    virtual void process() = 0;
    virtual ~Component(){}
};

//树节点
class Composite : public Component{

    string name;
    list<Component*> elements;
public:
    Composite(const string & s) : name(s) {}

    void add(Component* element) {
        elements.push_back(element);
    }
```

```
    void remove(Component* element){
        elements.remove(element);
    }

    void process(){

        //1. process current node


        //2. process leaf nodes
        for (auto &e : elements)
            e->process(); //多态调用

    }
};

//叶子节点
class Leaf : public Component{
    string name;
public:
    Leaf(string s) : name(s) {}

    void process(){
        //process current node
    }
};


void Invoke(Component & c){
    // ...
    c.process();
    // ...
}


int main()
{

    Composite root("root");
    Composite treeNode1("treeNode1");
    Composite treeNode2("treeNode2");
    Composite treeNode3("treeNode3");
    Composite treeNode4("treeNode4");
    Leaf leaf1("left1");
    Leaf leaf2("left2");

    root.add(&treeNode1);
    treeNode1.add(&treeNode2);
    treeNode2.add(&leaf1);

    root.add(&treeNode3);
    treeNode3.add(&treeNode4);
    treeNode4.add(&leaf2);

    process(root);
    process(leaf2);
    process(treeNode3);

}
```

### 总结

- Composite 模式采用**树形结构**来实现普遍存在的对象容器，从而将“一对多”的关系转化为“一对一”的关系，使得客户代码可以**一致地（复用）处理对象和对象容器**，无需关心处理的是单个的对象还是组合的对象容器。
- 将“客户代码与复杂的对象容器结构”解耦是Composite的核心思想，解耦之后，客户代码将与纯粹的**抽象接口**——而非对象容器的内部实现结构——发生依赖，从而更能“应对变化”。
- Composite模式在具体实现中，可以让父对象中的子对象反向追溯；如果父对象有频繁的遍历需求，可使用缓存技巧来改善效率。

## Iterator

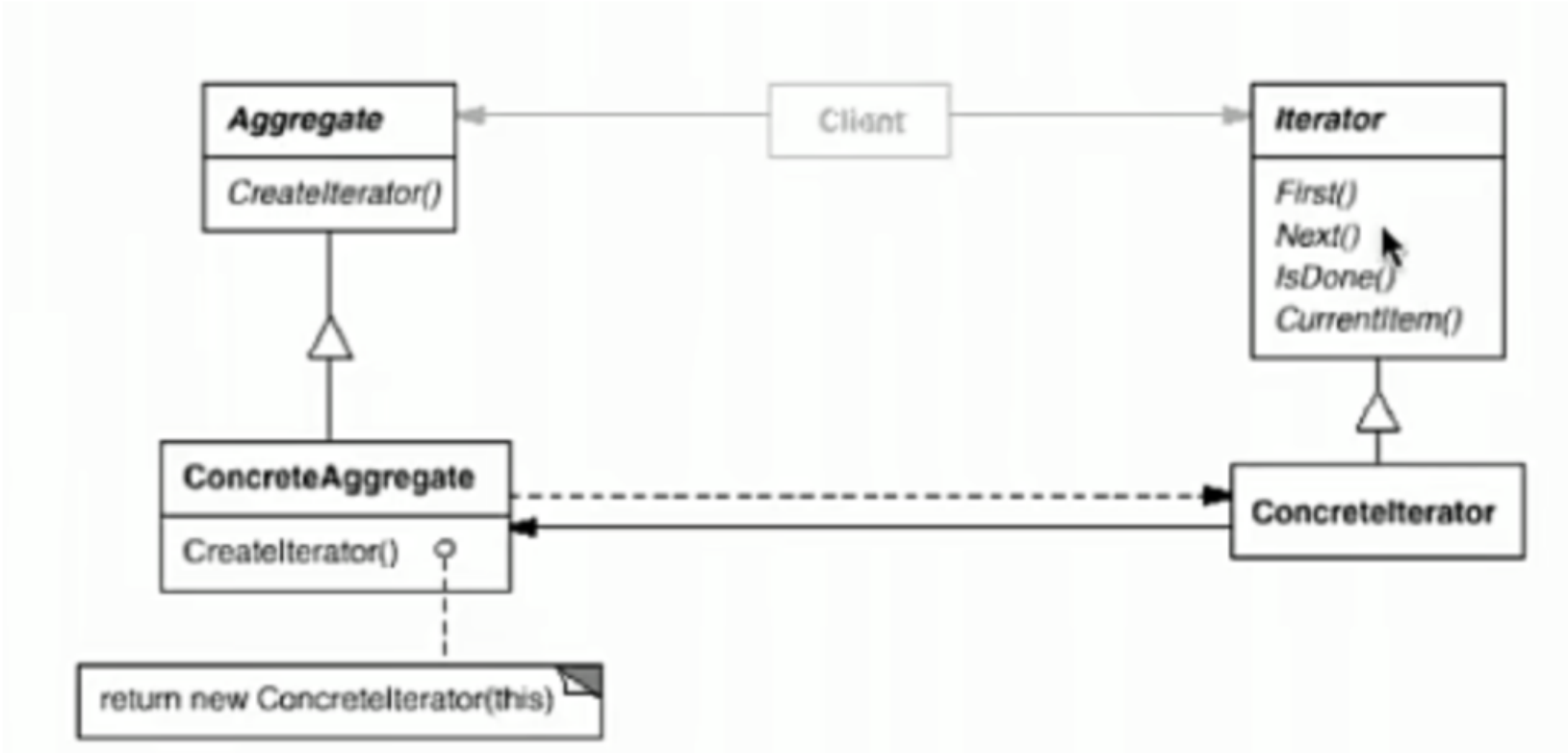
### 动机

> 在软件构建过程中，**集合对象内部结构常常变化各异**。但对于这些集合对象，我们希望在**不暴露其内部结构**的同时，可以让外部客户代码透明地访问其中包含的元素；同时这种“透明遍历”也为“同一种算法在多种集合对象上进行操作”提供了可能。
>
> 使用面向对象技术将这种遍历机制抽象为“迭代器对象”为“应对变化中的集合对象”提供了一种优雅的方式。

## 模式定义

> 提供一种方法顺序访问一个聚合对象中的各个元素，而又不暴露（稳定）该对象的内部表示。

## 结构




## 代码

```
template<typename T>
class Iterator
{
public:
    virtual void first() = 0;
    virtual void next() = 0;
    virtual bool isDone() const = 0;
    virtual T& current() = 0;
};



template<typename T>
class MyCollection{

public:

    Iterator<T> GetIterator(){
        // ...
    }

};

template<typename T>
class CollectionIterator : public Iterator<T>{
    MyCollection<T> mc;
public:

    CollectionIterator(const MyCollection<T> & c): mc(c){ }

    void first() override {

    }
    void next() override {

    }
    bool isDone() const override{

    }
    T& current() override{
```

```
    }
};

void MyAlgorithm()
{
    MyCollection<int> mc;

    Iterator<int> iter= mc.GetIterator();

    for (iter.first(); !iter.isDone(); iter.next()){
        cout << iter.current() << endl;
    }

}
```

## 总结

> - 迭代抽象：访问一个聚合对象的内容而无需暴露它的内部表示。
> - 迭代多态：为遍历不同的集合结构提供一个统一的接口，从而支持同样的算法在不同的集合结构上进行操作。
> - 迭代器的健壮性考虑：遍历的同时更改迭代器所在的集合结构，会导致问题。
> - 在C++里面已经**过时**了，用泛型取代，运行时绑定没有编译时绑定效率高。

# Chain of Responsibility

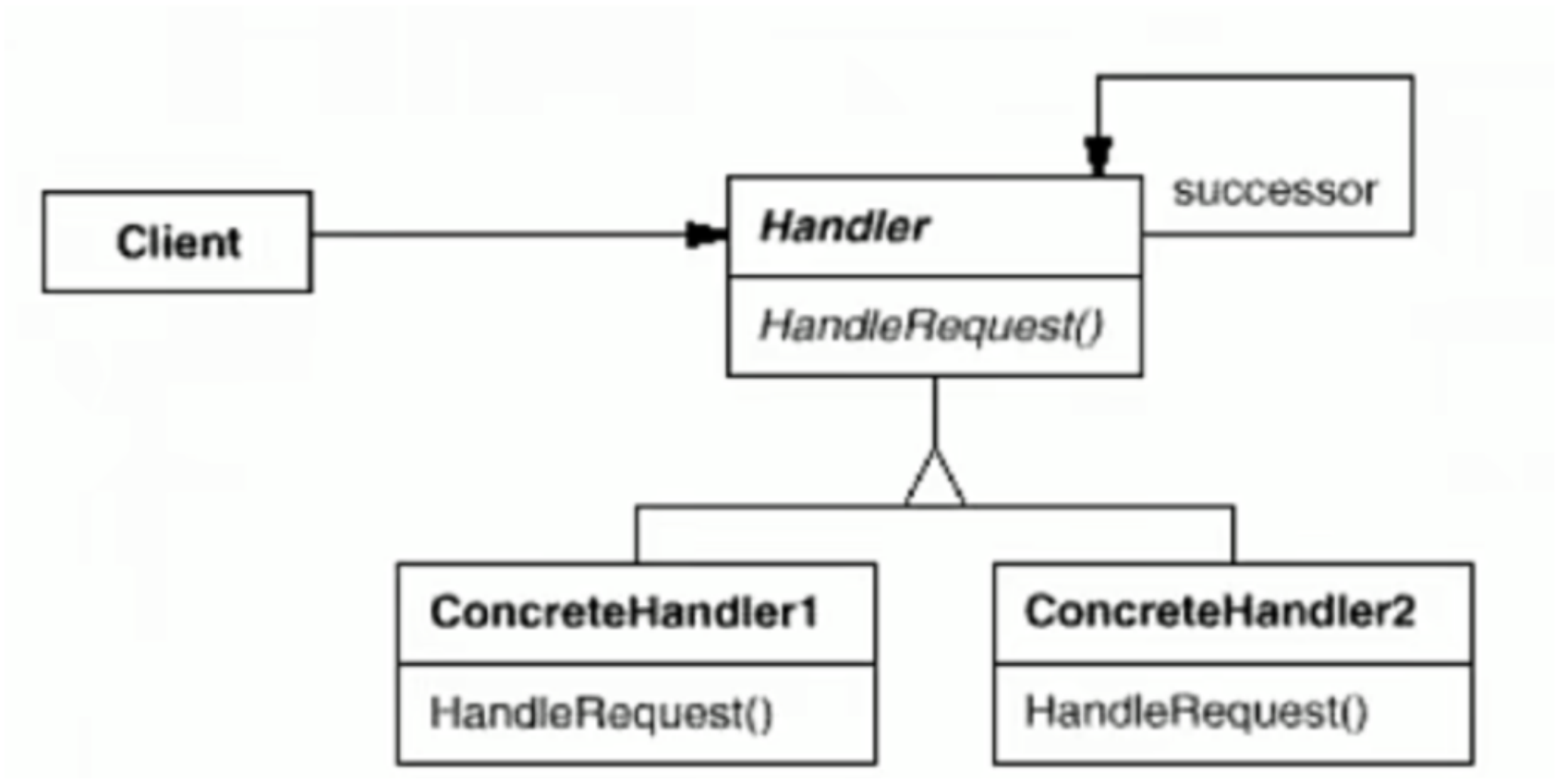
## 动机

> 在软件构建过程中，**一个请求可能被多个对象处理**，但是每个请求在运行时只能有一个接收者，如果显示指定，将必不可少地带来请求发送者与接受者的紧耦合。
>
> 如何使**请求的发送者不需要指定具体的接收者**？让请求的接收者自己在运行时决定来处理请求，从而使两者解耦。

## 模式定义

> 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**沿着这条链传递请求**，直到有一个对象处理它为止。

## 结构




`Client`和`Handler`稳定 `ConcreteHandler`变化

## 代码

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

using namespace std;

enum class RequestType
{
    REQ_HANDLER1,
    REQ_HANDLER2,
    REQ_HANDLER3
};
```

```
class Reqest
{
    string description;
    RequestType reqType;
public:
    Reqest(const string & desc, RequestType type) : description(desc), reqType(type) {}
    RequestType getReqType() const { return reqType; }
    const string& getDescription() const { return description; }
};

class ChainHandler{

    ChainHandler *nextChain;
    void sendReqestToNextHandler(const Reqest & req)
    {
        if (nextChain != nullptr)
            nextChain->handle(req);
    }
protected:
    virtual bool canHandleRequest(const Reqest & req) = 0;
    virtual void processRequest(const Reqest & req) = 0;
public:
    ChainHandler() { nextChain = nullptr; }
    void setNextChain(ChainHandler *next) { nextChain = next; }


    void handle(const Reqest & req)
    {
        if (canHandleRequest(req))
            processRequest(req);
        else
            sendReqestToNextHandler(req);
    }
};


class Handler1 : public ChainHandler{
protected:
    bool canHandleRequest(const Reqest & req) override
    {
        return req.getReqType() == RequestType::REQ_HANDLER1;
    }
    void processRequest(const Reqest & req) override
    {
        cout << "Handler1 is handle reqest: " << req.getDescription() << endl;
    }
};

class Handler2 : public ChainHandler{
protected:
    bool canHandleRequest(const Reqest & req) override
    {
        return req.getReqType() == RequestType::REQ_HANDLER2;
    }
    void processRequest(const Reqest & req) override
    {
        cout << "Handler2 is handle reqest: " << req.getDescription() << endl;
    }
};

class Handler3 : public ChainHandler{
protected:
    bool canHandleRequest(const Reqest & req) override
    {
        return req.getReqType() == RequestType::REQ_HANDLER3;
    }
    void processRequest(const Reqest & req) override
    {
        cout << "Handler3 is handle reqest: " << req.getDescription() << endl;
    }
};

int main(){
    Handler1 h1;
```

```
    Handler2 h2;
    Handler3 h3;
    h1.setNextChain(&h2);
    h2.setNextChain(&h3);

    Reqest req("process task ... ", RequestType::REQ_HANDLER3);
    h1.handle(req);
    return 0;
}
```

### 总结

- Chain of Responsibility 模式的应用场合在于“**一个请求可能有多个接收者，但是最后真正的接收者只有一个**”，这时候请求发送者与接收者的耦合有可能出现“变化脆弱”的症状，职责链的目的就是将二者解耦，从而更好地应对变化。
- 应用了 Chain of Responsibility 模式后，对象的职责分派将更具灵活性。我们可以在**运行时动态添加/修改请求的处理职责**。
- 如果请求传递到职责链的末尾仍得不到处理，应该有一个合理的**缺省机制**。这也是每一个接受对象的责任，而不是发出请求的对象的责任。
- **过时**

## 行为变化模式

> - 在组件的构建过程中，组件行为的变化经常导致组件本身剧烈的变化。“行为变化”模式将组件的行为和组件本身进行解耦，从而支持组件行为的变化，实现两者之间的松耦合。
> - 典型模式
>   - Command
>   - Visitor

## Command

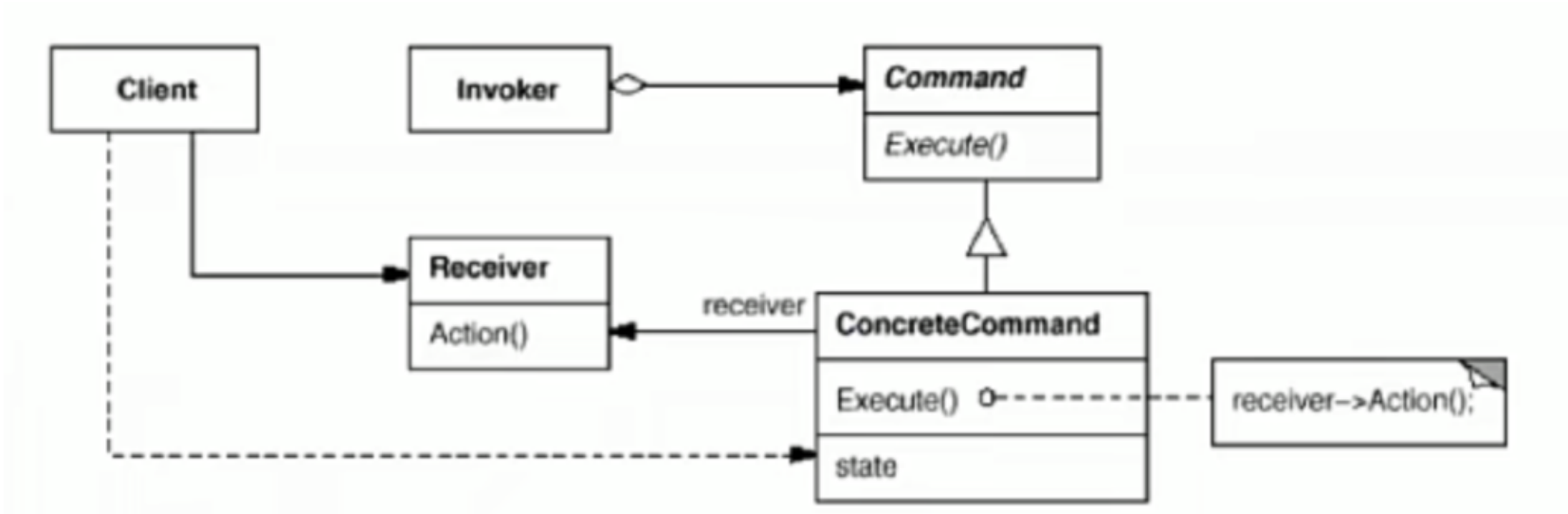
### 动机

> 在软件构建过程中，“行为请求者”与“行为实现者”通常呈现一种“紧耦合”。但在某些场合—比如需要对行为进行“记录、撤销/重做（undo/redo）、事务”等处理，这种无法抵御变化的紧耦合是不适合的。
>
> 在这种情况下，如何将“**行为请求者**”与“**行为实现者**”解耦？将一组行为抽象为对象，可以实现二者之间的松耦合。

### 模式定义

> 将一个**请求（行为）封装为一个对象**，从而使你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。

### 结构




`Command` 稳定 `ConcreteCommand` 变化

### 代码

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


class Command
{
public:
    virtual void execute() = 0;
```

```
};

class ConcreteCommand1 : public Command
{
    string arg;
public:
    ConcreteCommand1(const string & a) : arg(a) {}
    void execute() override
    {
        cout<< "#1 process ..."<<arg<<endl;
    }
};

class ConcreteCommand2 : public Command
{
    string arg;
public:
    ConcreteCommand2(const string & a) : arg(a) {}
    void execute() override
    {
        cout<< "#2 process ..."<<arg<<endl;
    }
};
```

```
class MacroCommand : public Command
{
    vector<Command*> commands;
public:
    void addCommand(Command *c) { commands.push_back(c); }
    void execute() override
    {
        for (auto &c : commands)
        {
            c->execute();
        }
    }
};



int main()
{

    ConcreteCommand1 command1(receiver, "Arg ###");
    ConcreteCommand2 command2(receiver, "Arg $$$");

    MacroCommand macro;
    macro.addCommand(&command1);
    macro.addCommand(&command2);

    macro.execute();

}
```

## 总结

- Command 模式的根本目的在于将“行为请求者”与“行为实现者”解耦，在面向对象语言中，常见的实现手段是“**将行为抽象为对象**”。
- 实现Command接口的具体命令对象ConcreteCommand有时候根据需要可能会保存一些额外的状态信息。通过使用**Composite模式**，可以将多个“命令”封装为一个“复合命令”MacroCommand。
- Command 模式与C++中的**函数对象(functor)**有些类似。但两者定义行为接口的规范有所区别：Command以面向对象中的“接口-实现”来定义行为接口规范，更严格，但有性能损失；C++函数对象以函数签名来定义行为接口规范，更灵活，性能更高。

# Visitor

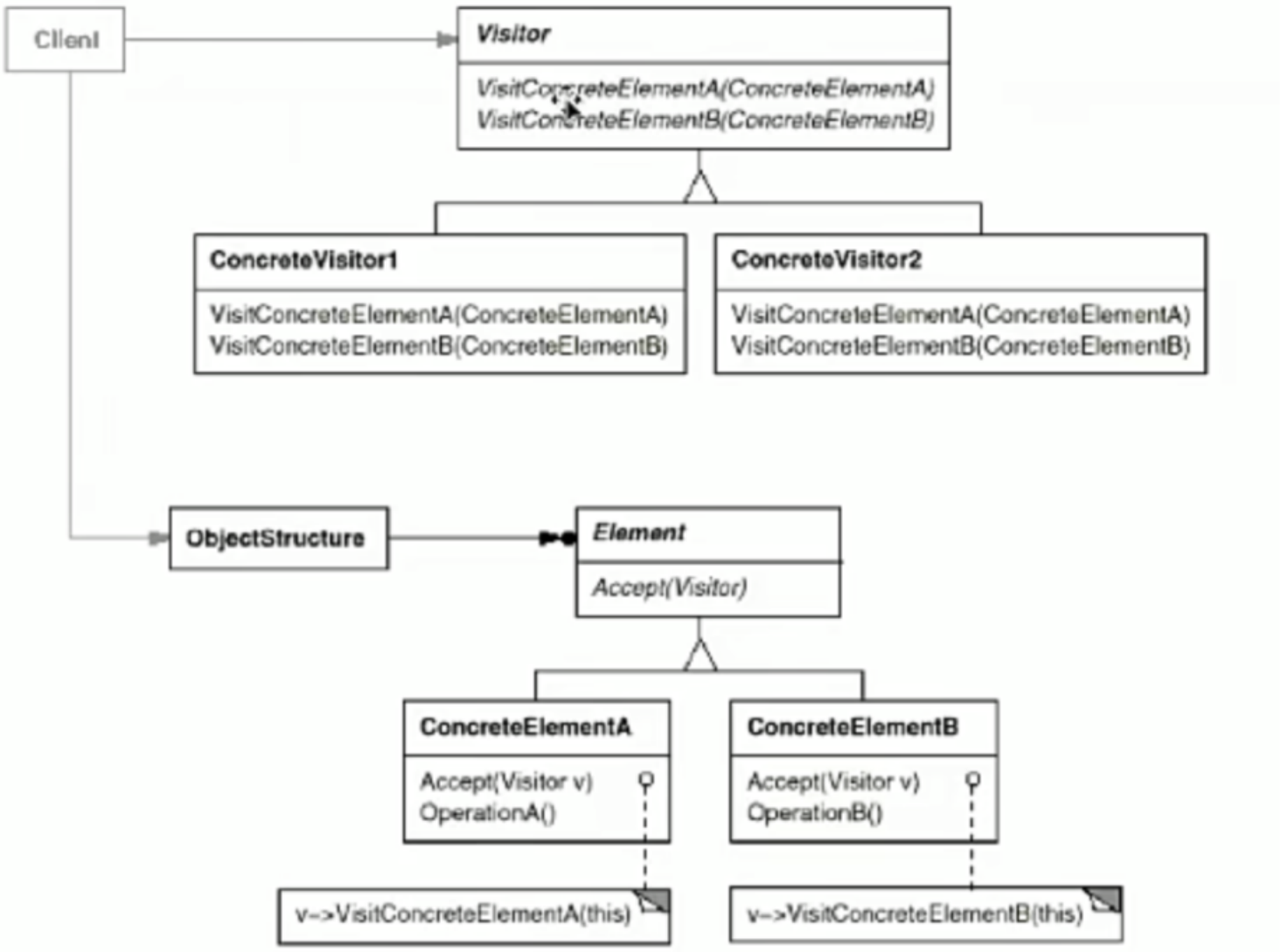
## 动机

> 在软件构件的过程中，由于需求的改变，某些类层次结构中常常**需要增加新的行为**，如果**直接在基类中作出修改，将会给子类带来繁重的变更负担**，甚至破坏原有设计。

## 模式定义

> 表示一个作用于某对象结构中的各元素的操作。使得可以在不改变（稳定）各元素的类的前提下定义（扩展）作用于这些元素的新操作（变化）。

## 结构




`Element` `ConcreteElement` `Visitor` 稳定 `ConcreteVisitor` 变化

## 代码

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

class Visitor;


class Element
{
public:
    virtual void accept(Visitor& visitor) = 0; //第一次多态辨析

    virtual ~Element(){}
};

class ElementA : public Element
{
public:
    void accept(Visitor &visitor) override {
        visitor.visitElementA(*this);
    }


};

class ElementB : public Element
{
public:
    void accept(Visitor &visitor) override {
        visitor.visitElementB(*this); //第二次多态辨析
    }

};


class Visitor{
public:
    virtual void visitElementA(ElementA& element) = 0;
    virtual void visitElementB(ElementB& element) = 0;

    virtual ~Visitor(){}
```

```
};

//=============================

//扩展1
class Visitor1 : public Visitor{
public:
    void visitElementA(ElementA& element) override{
        cout << "Visitor1 is processing ElementA" << endl;
    }

    void visitElementB(ElementB& element) override{
        cout << "Visitor1 is processing ElementB" << endl;
    }
};

//扩展2
class Visitor2 : public Visitor{
public:
    void visitElementA(ElementA& element) override{
        cout << "Visitor2 is processing ElementA" << endl;
    }

    void visitElementB(ElementB& element) override{
        cout << "Visitor2 is processing ElementB" << endl;
    }
};



int main()
{
    Visitor2 visitor;
    ElementB elementB;
    elementB.accept(visitor);// double dispatch

    ElementA elementA;
    elementA.accept(visitor);


    return 0;
}
```

### 总结

- Visitor 模式通过所谓**双重分发（double dispatch）**来实现在不更改（不添加新的操作-编译时）Element类层次结构的前提下，在运行时透明地为类层次结构上的各个类动态添加新的操作（支持变化）。
- 所谓**双重分发**即Visitor模式中间包括了两个多态分发（注意其中的多态机制）：第一个为accept方法的多态辨析；第二个为visitElementX方法的多态辨析。
- Visitor模式的做大缺点在于扩展类层次结构（增添新的Element子类），会导致Visitor类的改变。因此Visitor模式适用于“**Element类层次结构稳定**，而其中的操作却经常面临频繁改动”。

# 领域规则模式

> - 在特定领域中，某些变化虽然频繁，但可以抽象为某种规则。这时候，结合特定领域，将问题抽象为语法规则，从而给出在该领域下的一般性解决方案。
> - 典型模式
>   - Interpreter

# Interpreter

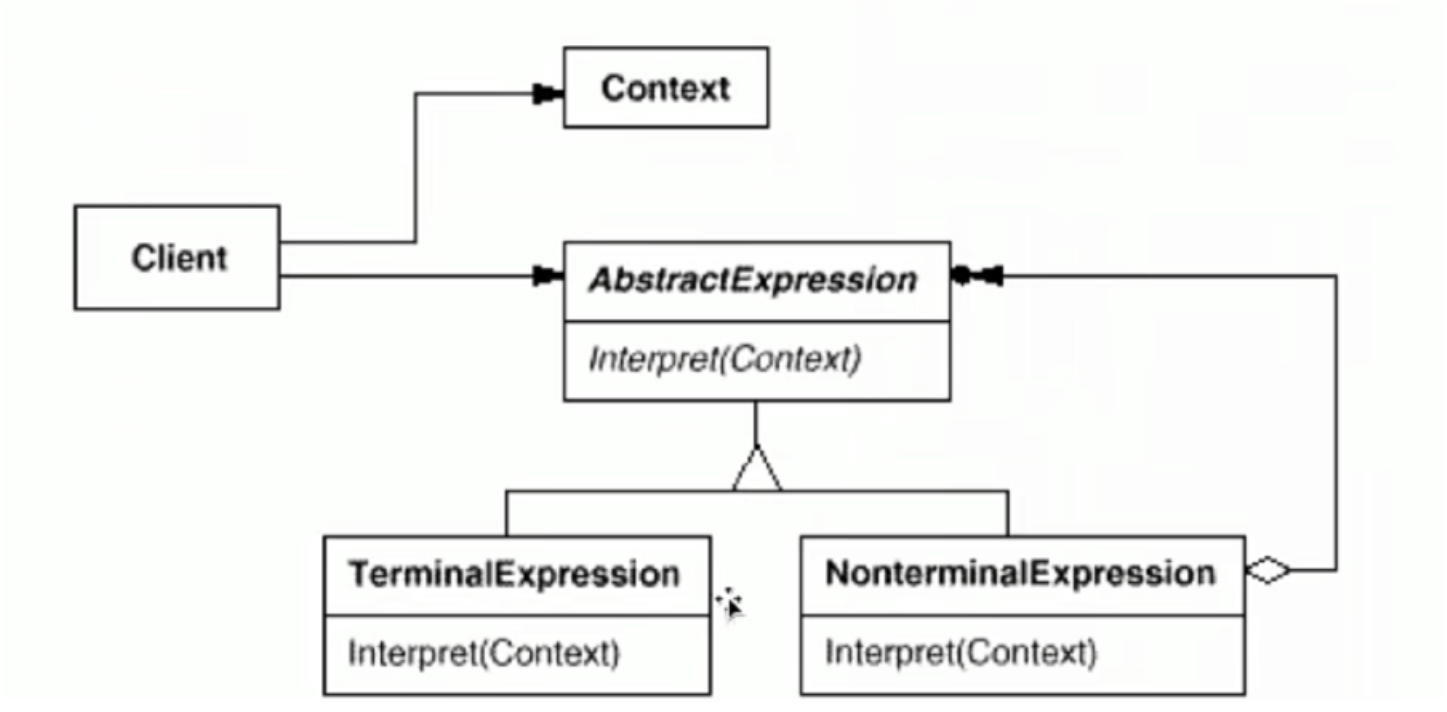
## 动机

> 在软件构建过程中，如果某一特定领域的问题比较复杂，类似的结构不断重复出现，如果使用普通的编程方法来实现将面临非常频繁的变化。
>
> 在这种情况下，将**特定领域的问题表达为某种语法规则下的句子**，然后**构建一个解释器来解释这样的句子**，从而达到解决问题的目的。

## 模式定义

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

## 结构




## 代码

```
#include <iostream>
#include <map>
#include <stack>

using namespace std;

class Expression {
public:
    virtual int interpreter(map<char, int> var)=0;
    virtual ~Expression(){}
};

//变量表达式
class VarExpression: public Expression {

    char key;

public:
    VarExpression(const char& key)
    {
        this->key = key;
    }

    int interpreter(map<char, int> var) override {
        return var[key];
    }

};

//符号表达式
class SymbolExpression : public Expression {

    // 运算符左右两个参数
protected:
    Expression* left;
    Expression* right;

public:
    SymbolExpression( Expression* left,  Expression* right):
        left(left),right(right){

    }

};

//加法运算
class AddExpression : public SymbolExpression {

public:
```

```
    AddExpression(Expression* left, Expression* right):
        SymbolExpression(left,right){

    }
    int interpreter(map<char, int> var) override {
        return left->interpreter(var) + right->interpreter(var);
    }

};

//减法运算
class SubExpression : public SymbolExpression {

public:
    SubExpression(Expression* left, Expression* right):
        SymbolExpression(left,right){

    }
    int interpreter(map<char, int> var) override {
        return left->interpreter(var) - right->interpreter(var);
    }

};


Expression*  analyse(string expStr) {

    stack<Expression*> expStack;
    Expression* left = nullptr;
    Expression* right = nullptr;
    for(int i=0; i<expStr.size(); i++)
    {
        switch(expStr[i])
        {
            case '+':
                // 加法运算
                left = expStack.top();
                right = new VarExpression(expStr[++i]);
                expStack.push(new AddExpression(left, right));
                break;
            case '-':
                // 减法运算
                left = expStack.top();
                right = new VarExpression(expStr[++i]);
                expStack.push(new SubExpression(left, right));
                break;
            default:
                // 变量表达式
                expStack.push(new VarExpression(expStr[i]));
        }
    }

    Expression* expression = expStack.top();

    return expression;
}

void release(Expression* expression){

    //释放表达式树的节点内存 ...
}

int main(int argc, const char * argv[]) {


    string expStr = "a+b-c+d-e";
    map<char, int> var;
    var.insert(make_pair('a',5));
    var.insert(make_pair('b',2));
    var.insert(make_pair('c',1));
    var.insert(make_pair('d',6));
    var.insert(make_pair('e',10));
```

```
    Expression* expression= analyse(expStr);

    int result=expression->interpreter(var);

    cout<<result<<endl;

    release(expression);

    return 0;
}
```

## 总结

- Interpreter模式的应用场合是Interpret模式应用中的难点，只有满足“**业务规则频繁变化，且类似的结构不断重复出现，并且容易抽象为语法规则的问题**”才适合使用Interpreter模式。
- 使用Interpreter模式来表示文法规则，从而可以使用面向对象技巧来方便地“扩展”文法。
- Interpreter模式比较**适合简单的文法**表示，对于复杂的文法表示，Interpreter模式会产生比较大的类层次结构，需要求助于**语法分析生成器这样的标准工具**。
- 应用场合有限