



# 行为模式

---



# 行为模式

---

- 行为模式涉及到算法和对象间职责的分配
  - 描述对象或类的模式
  - 描述它们之间的通信模式



# 行为类模式

---

- 使用继承机制在类间分派行为
- Template Method
  - 模板方法是一个算法的抽象定义
  - 子类定义抽象操作以具体实现该算法
- Interpreter
  - 将一个文法表示为一个类层次



# 行为对象模式

---

- 使用对象复合
- 类型1:
  - 描述一组对等的对象怎样相互协作
  - 问题：对等的对象如何了解对方？
  - Mediator
  - Chain of Responsibility
  - Observer



# 行为对象模式

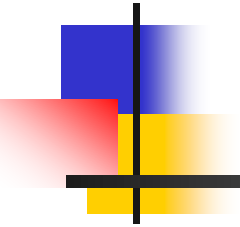
---

- 类型2

- 将行为封装在一个对象中并将请求指派给它
- Strategy
- Command
- State
- Visitor

# 5.1 Chain of Responsibility

职责链





# 1. 意图

---

- 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。



## 2. 动机

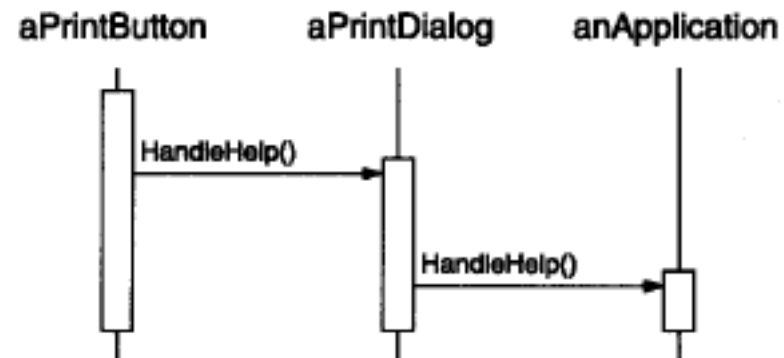
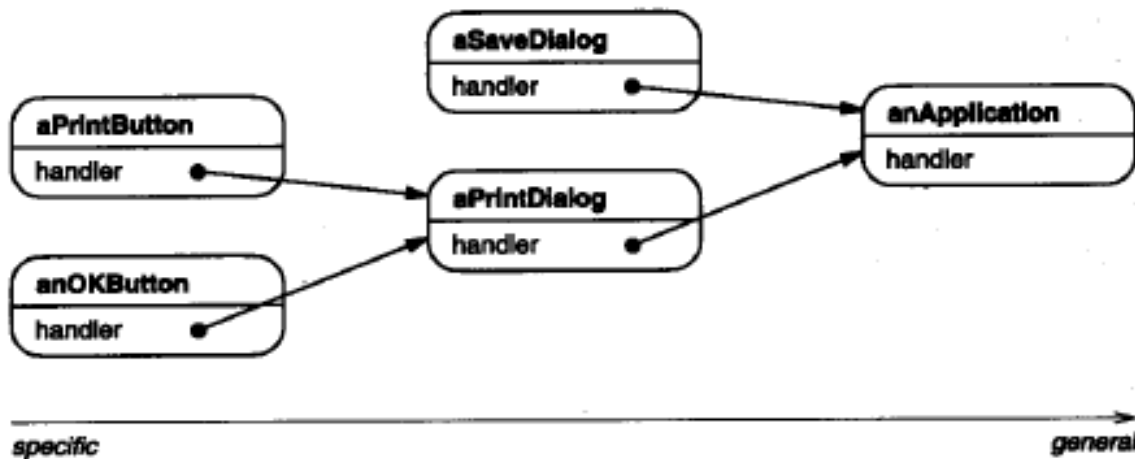
---

- 图形用户界面中的Context有关的帮助机制
  - 根据普遍性（generality）组织帮助信息
    - 从最特殊到最普通
  - 提交请求的对象不知道谁是最终提供响应的对象
    - 使用Chain of Responsibility模式，使得请求者和响应者解耦(decouple)

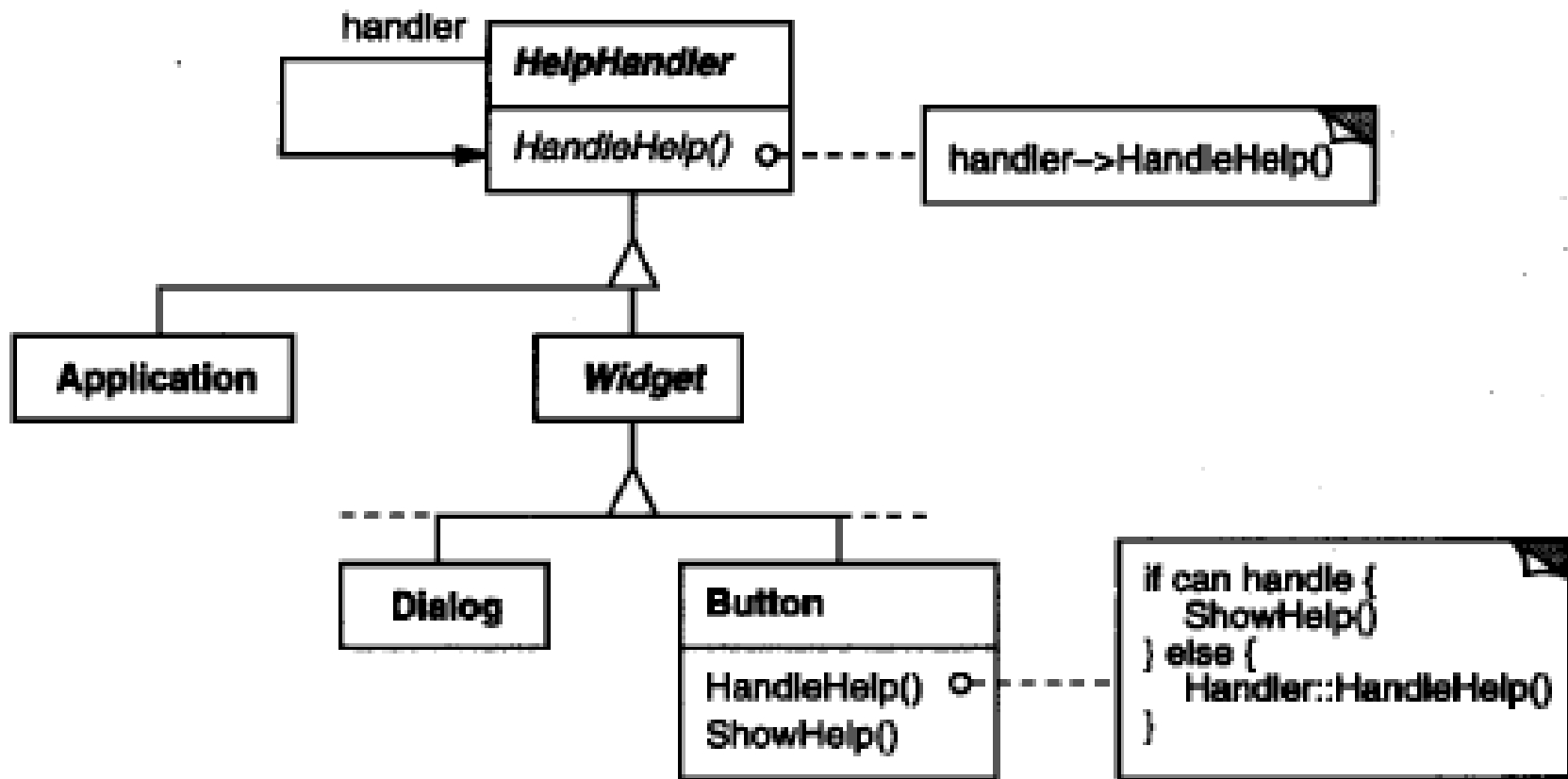


## 2. 动机

- 隐式的接收者(implicit receiver)



## 2. 动机



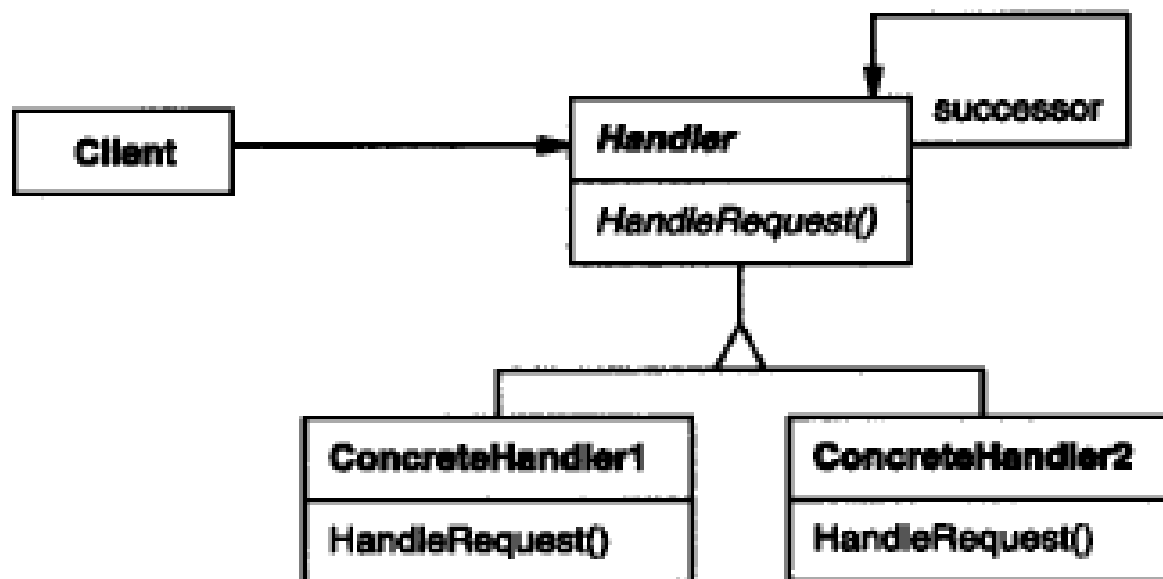


### 3. 适用性

---

- 有多个对象可以处理一个请求
  - 哪个对象处理该请求，由运行时刻自动确定
- 在不明确接收者的情况下，向多个对象中的一个提交一个请求
- 可处理一个请求的对象集合应被动态指定

## 4. 结构





## 5. 参与者

---

- Handler (HelpHandler)
  - 定义一个处理请求的接口
  - (可选)实现后继链
- ConcreteHandler ( PrintButton, PrintDialog)
  - 处理它所负责的请求
  - 可访问它的后继者
  - 如果可处理请求，则处理；否则将请求转发给它的后继者
- Client
  - 向链上的具体处理者(ConcreteHandler)对象提交请求



## 9. 协作

---

- 当客户提交一个请求时，请求沿链传递直至有一个ConcreteHandler对象负责处理它



## 7. 效果

---

### 1) 降低耦合度

- 接收者和发送者都没有对象的明确信息，且链中的对象不需知道链的结构
- 职责链可简化对象的相互连接
  - 仅需保持一个指向后继者的引用
  - 不需保持所有候选接受者的引用



## 7. 效果

---

2) 增强了给对象指派职责(Responsibility)的灵活性

- 在运行时刻对职责链进行动态增加和修改

3) 不保证被接受

- 一个请求，不能保证它一定会被处理





## 8. 实现

---

1) 实现后继者链

a) 定义新的链接

e.g. 使用Handler类...

b) 使用已有的链接

e.g. 复用Composite模式中的链等



## 8.实现

---

### 2) 连接后继者

```
class HelpHandler {
public:
    HelpHandler(HelpHandler* s) : _successor(s) { }
    virtual void HandleHelp();
private:
    HelpHandler* _successor;
};

void HelpHandler::HandleHelp () {
    if (_successor) {
        _successor->HandleHelp();
    }
}
```

Handler不仅定义请求的接口，  
也维护后继链接



## 8.实现

---

### 3) 表示请求

- 方式1：硬编码(hard-coded)的操作调用
  - 方便、安全；
  - 只能处理Handler类中定义的固定的一组请求
- 方式2：使用一个处理函数
  - 处理函数以一个请求码为参数
  - 优点：更为灵活
  - 缺点：无法用类型安全的方法传递请求参数



## 8.实现

---

```
void Handler::HandleRequest (Request* theRequest) {  
    switch (theRequest->GetKind()) {  
        case Help:  
            // cast argument to appropriate type  
            HandleHelp((HelpRequest*) theRequest);  
            break;  
  
        case Print:  
            HandlePrint((PrintRequest*) theRequest);  
            // ...  
            break;  
  
        default:  
            // ...  
            break;  
    }  
}
```

用独立的请求对象来封装请求参数



## 8.实现

---

```
class ExtendedHandler : public Handler {
public:
    virtual void HandleRequest(Request* theRequest);
    // ...
};

void ExtendedHandler::HandleRequest (Request* theRequest) {
    switch (theRequest->GetKind()) {
        case Preview:
            // handle the Preview request
            break;
        default:
            // let Handler handle other requests
            Handler::HandleRequest(theRequest);
    }
}
```

子类扩展

## 9.代码示例

```
typedef int Topic;
const Topic NO_HELP_TOPIC = -1;

class HelpHandler {
public:
    HelpHandler(HelpHandler* = 0, Topic = NO_HELP_TOPIC);
    virtual bool HasHelp();
    virtual void SetHandler(HelpHandler*, Topic);
    virtual void HandleHelp();
private:
    HelpHandler* _successor;
    Topic _topic;
};

HelpHandler::HelpHandler (
    HelpHandler* h, Topic t
) : _successor(h), _topic(t) { }

bool HelpHandler::HasHelp () {
    return _topic != NO_HELP_TOPIC;
}

void HelpHandler::HandleHelp () {
    if (_successor != 0) {
        _successor->HandleHelp();
    }
}
```

检索是否存在相关的帮助主题

HelpHandler类



## 9.代码示例

---

```
class Widget : public HelpHandler {  
protected:  
    Widget(Widget* parent, Topic t = NO_HELP_TOPIC);  
private:  
    Widget* _parent;  
};  
  
Widget::Widget (Widget* w, Topic t) : HelpHandler(w, t) {  
    _parent = w;  
}
```

窗口组件类 Widget



## 9.代码示例

---

```
class Button : public Widget {
public:
    Button(Widget* d, Topic t = NO_HELP_TOPIC);

    virtual void HandleHelp();
    // Widget operations that Button overrides...
};
```

```
Button::Button (Widget* h, Topic t) : Widget(h, t) { }

void Button::HandleHelp () {
    if (HasHelp()) {
        // offer help on the button
    } else {
        HelpHandler::HandleHelp();
    }
}
```

Button类





## 9.代码示例

---

```
class Dialog : public Widget {
public:
    Dialog(Handler* h, Topic t = NO_HELP_TOPIC);
    virtual void HandleHelp();

    // Widget operations that Dialog overrides...
    // ...
};

Dialog::Dialog (Handler* h,  Topic t) : Widget(0) {
    SetHandler(h, t);
}

void Dialog::HandleHelp () {
    if (HasHelp()) {
        // offer help on the dialog
    } else {
        Handler::HandleHelp();
    }
}
```

Dialog类



## 9.代码示例

---

```
class Application : public HelpHandler {  
public:  
    Application(Topic t) : HelpHandler(0, t) { }  
  
    virtual void HandleHelp();  
    // application-specific operations...  
};  
  
void Application::HandleHelp () {  
    // show a list of help topics  
}
```

Application类



## 9.代码示例

---

```
const Topic PRINT_TOPIC = 1;
const Topic PAPER_ORIENTATION_TOPIC = 2;
const Topic APPLICATION_TOPIC = 3;

Application* application = new Application(APPLICATION_TOPIC);
Dialog* dialog = new Dialog(application, PRINT_TOPIC);
Button* button = new Button(dialog, PAPER_ORIENTATION_TOPIC);
```

Client

```
button->HandleHelp();
```



# 11. 相关模式

---

- 职责链常与Composite模式一起使用
  - 一个组件的父组件可作为它的后继

## 5.2 Command

命令

---



# Command模式

---

## 1. 意图

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

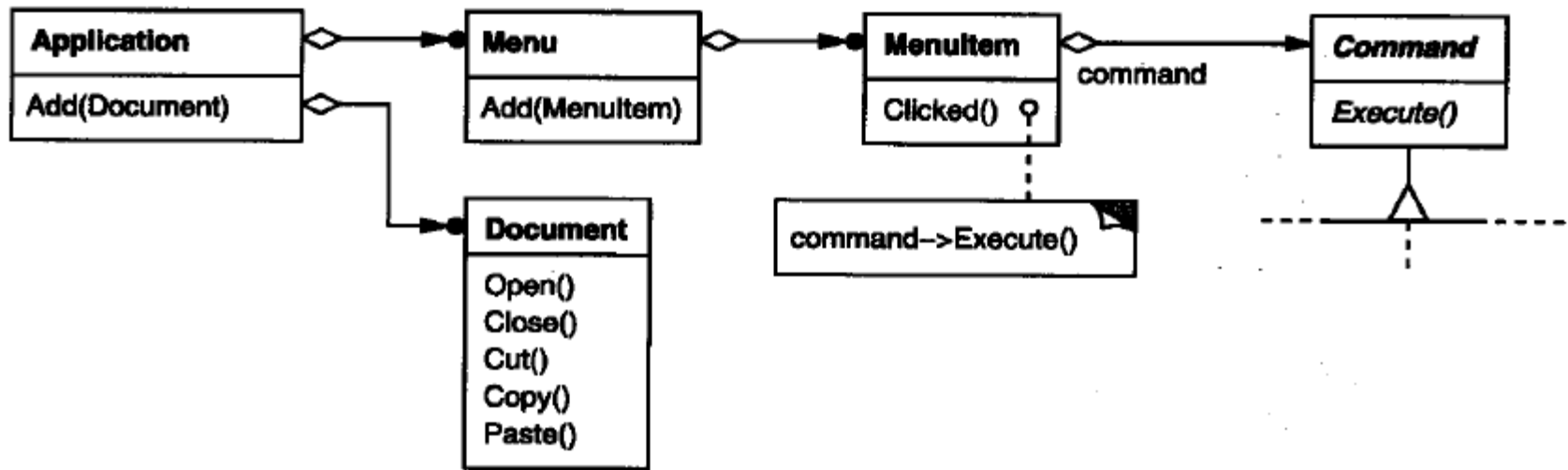
## 2. 别名

动作(Action)

事务(Transaction)

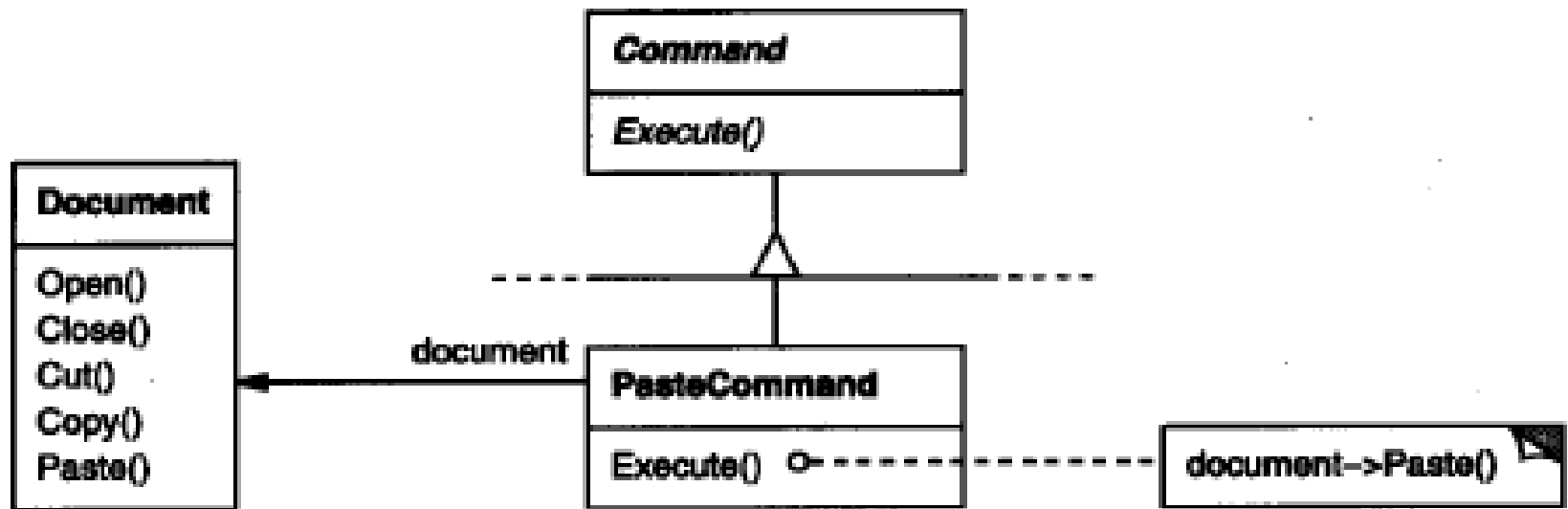
### 3. 动机

应用为每个菜单项配置一个Command对象



有时必须向某对象提交请求，但并不知道被请求的操作或者请求的接受者的信息

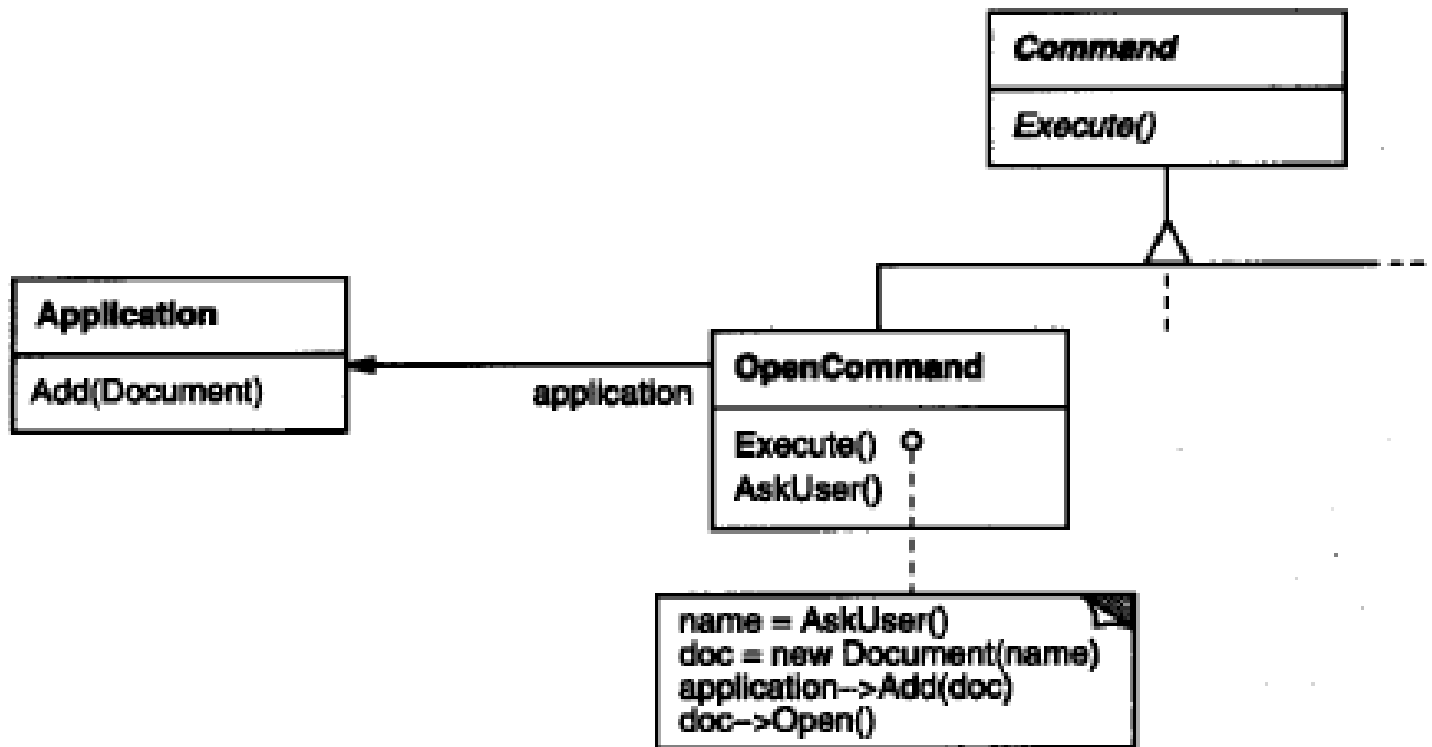
### 3. 动机



PasteCommand对象

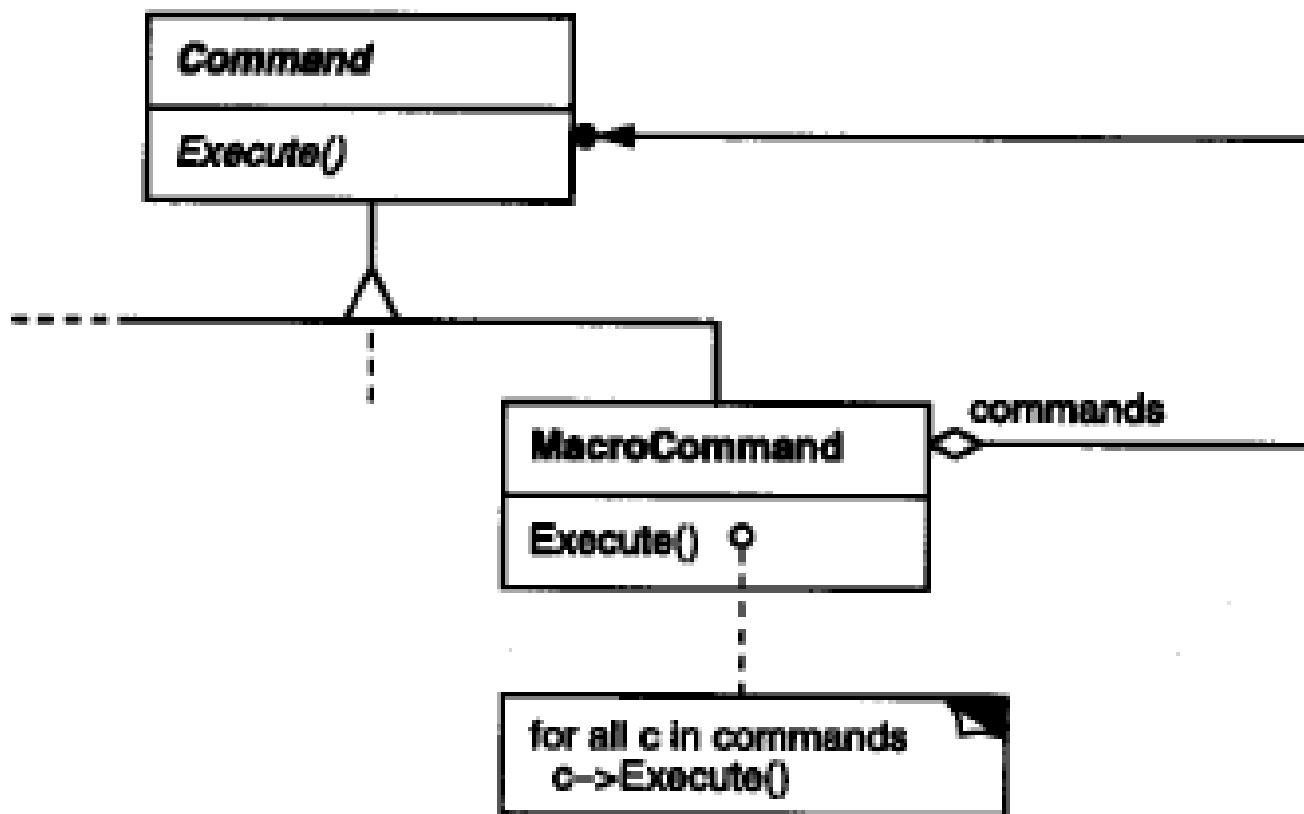


### 3. 动机



OpenCommand

### 3. 动机



宏命令：命令序列



## 3. 动机

---

- 菜单和按钮代表同一功能
  - 共享Command对象
- 命令脚本(command scripting)
- 提交一个请求的对象
  - 仅需知道如何提交它
  - 不需知道该请求将会被如何执行

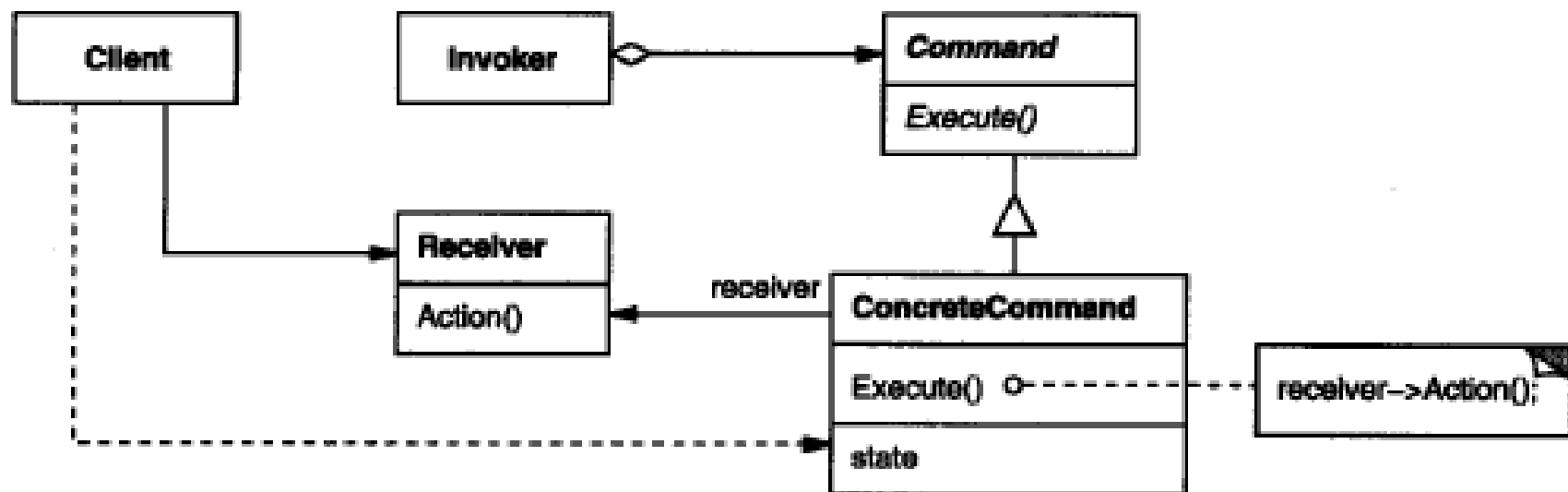


## 4. 适用性

---

- 抽象出待执行的动作以参数化某对象
  - callback
- 在不同的时刻指定、排列和执行请求
- 支持取消操作
- 支持修改日志
- 用构建在原语操作上的高层操作构造一个系统
  - Transaction事务

## 5. 结构



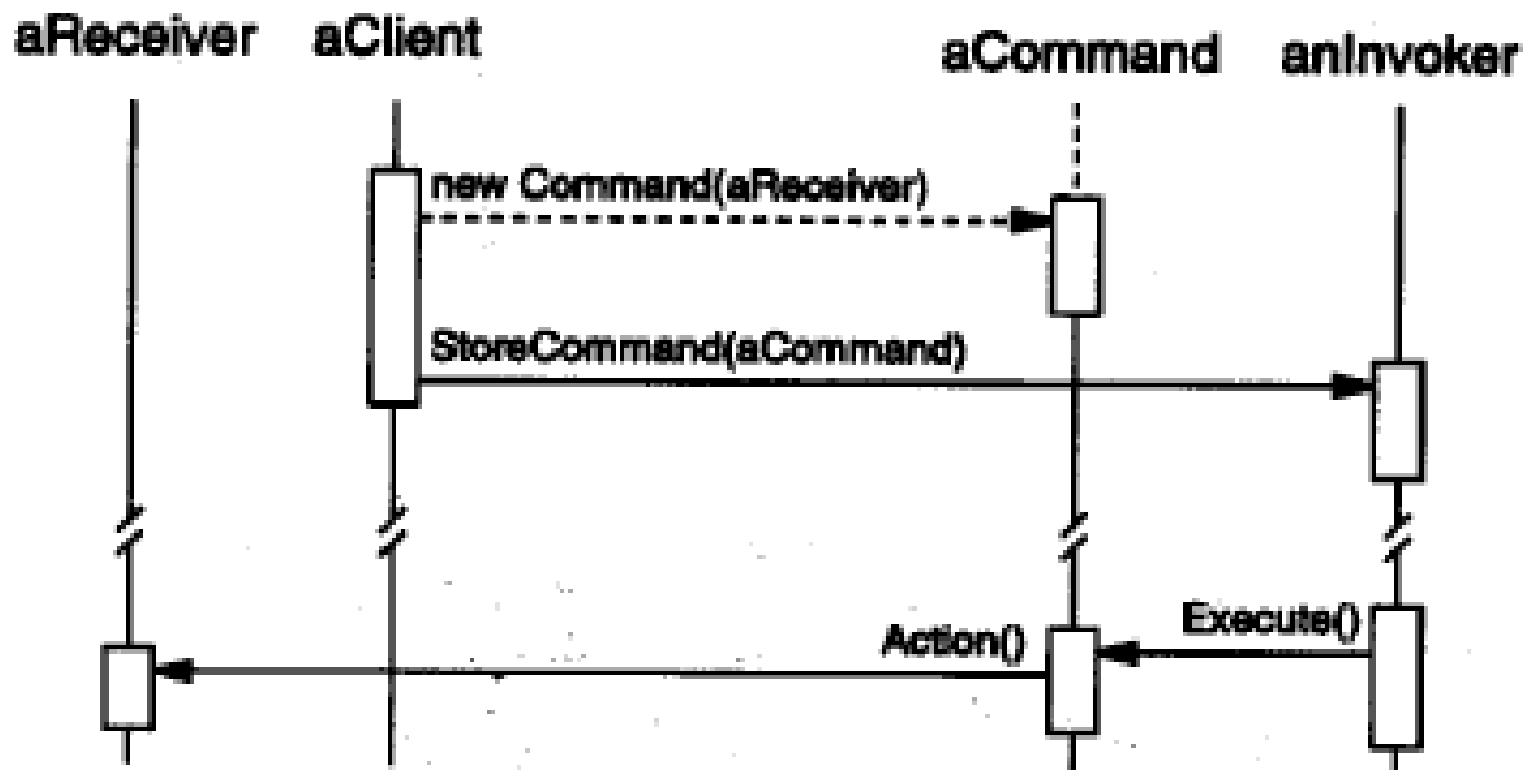


## 6. 参与者

---

- Command
  - 声明执行操作的接口
- ConcreteCommand (PasteCommand, OpenCommand)
  - 将一个接收者对象绑定于一个动作
  - 调用接收者相应的操作，以实现Execute
- Client (Application)
  - 创建一个具体命令对象并设定它的接收者
- Invoker (MenuItem)
  - 要求该命令执行这个请求
- Receiver (Document, Application)
  - 知道如何实施与执行一个请求相关的操作
  - 任何类都可能作为一个请求者

## 7. 协作





## 8. 效果

---

- 将调用操作的对象(Invoker)与知道如何实现该操作的对象(Receiver)解耦
- Command可以像其他对象一样操作以及扩展
- 多个命令装配成一个复合命令
  - Composite模式
- 增加新的Command很容易，无需改变已有的类





## 9.实现

---

### 1) 一个Command对象应达到何种智能程度

- 命令对象的能力可大可小
  - 仅确定一个接收者和执行的动作
  - 或
  - Command对象自己实现所有功能，不需要额外的接收者对象
- 普通情况：
  - Command对象有足够的信息可以动态找到它们的接收者



## 9. 实现

---

### 2) 支持取消(undo)和重做(redo)

- Command提供reverse操作(Unexecute或者Undo操作)
- 此时, ConcreteCommand类需要额外的状态信息
  - 接收者对象, 其上执行的操作
  - 接收者上执行操作的参数
  - 如果该操作改变了接收者对象的值, 则需要存储这些值
    - 接收者需要提供操作, 以便恢复到先前状态



## 9. 实现

---

### 2) 支持取消(undo)和重做(redo)

- 已执行命令的历史序列(history list)
  - 向后遍历: undo
  - 向前遍历: redo
- 如果命令的状态在各次调用之间会发生变化, 则必须进行拷贝以区分不同调用
  - e.g. DeleteCommand命令对象



## 9.实现

---

### 3) 避免取消操作过程中的错误积累

- 在Command中存入更多的信息以保证这些对象被精确复原成其初始状态
- Memento模式

### 4)使用C++模板

对(1)不能撤销(2)不需要参数的命令，  
可使用C++模板来实现



## 10.代码示例

---

```
class Command {  
public:  
    virtual ~Command();  
  
    virtual void Execute() = 0;  
protected:  
    Command();  
};
```

抽象Command类

## 10.代码示例

```
class OpenCommand : public Command {
public:
    OpenCommand(Application*);

    virtual void Execute();
protected:
    virtual const char* AskUser();
private:
    Application* _application;
    char* _response;
};
```

```
OpenCommand::OpenCommand (Application* a) {
    _application = a;
}

void OpenCommand::Execute () {
    const char* name = AskUser();

    if (name != 0) {
        Document* document = new Document(name);
        _application->Add(document);
        document->Open();
    }
}
```

OpenCommand



## 10.代码示例

---

```
class PasteCommand : public Command {
public:
    PasteCommand(Document*);

    virtual void Execute();
private:
    Document* _document;
};

PasteCommand::PasteCommand (Document* doc) {
    _document = doc;
}

void PasteCommand::Execute () {
    _document->Paste();
}
```

PasteCommand



## 10.代码示例

使用类模板处理简单命令  
(不能取消、不需参数)

```
template <class Receiver>
class SimpleCommand : public Command {
public:
    typedef void (Receiver::* Action)();

    SimpleCommand(Receiver* r, Action a) :
        _receiver(r), _action(a) { }

    virtual void Execute();
private:
    Action _action;
    Receiver* _receiver;
};
```

```
template <class Receiver>
void SimpleCommand<Receiver>::Execute () {
    (_receiver->* _action)();
}
```

```
MyClass* receiver = new MyClass;
// ...
Command* aCommand =
    new SimpleCommand<MyClass>(receiver, &MyClass::Action);
// ...
aCommand->Execute();
```



# 10.代码示例

```
class MacroCommand : public Command {
public:
    MacroCommand();
    virtual ~MacroCommand();

    virtual void Add(Command*);
    virtual void Remove(Command*);

    virtual void Execute();
private:
    List<Command*>* _cmds;
};
```

```
void MacroCommand::Execute () {
    ListIterator<Command*> i(_cmds);

    for (i.First(); !i.IsDone(); i.Next()) {
        Command* c = i.CurrentItem();
        c->Execute();
    }
}
```

MacroCommand : 子命令序列

管理子命令

```
void MacroCommand::Add (Command* c) {
    _cmds->Append(c);
}

void MacroCommand::Remove (Command* c) {
    _cmds->Remove(c);
}
```



## 12. 相关模式

---

- Composite模式：用于实现宏命令
- Memento模式
  - 用来保存某个状态，Command对象使用此状态来实现撤销操作
- 在被放入历史列表前必须被拷贝的Command起到原型(prototype)的作用



## 5.3 Interpreter 解释器

---



# 1. 意图

---

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



## 2. 动机

---

- 字符串模式匹配
  - 解释执行一个正则表达式
  - 解释器模式描述了
    - 如何为简单的语言定义一个文法
    - 如何在该语言中表示一个句子
    - 如何解释这些句子



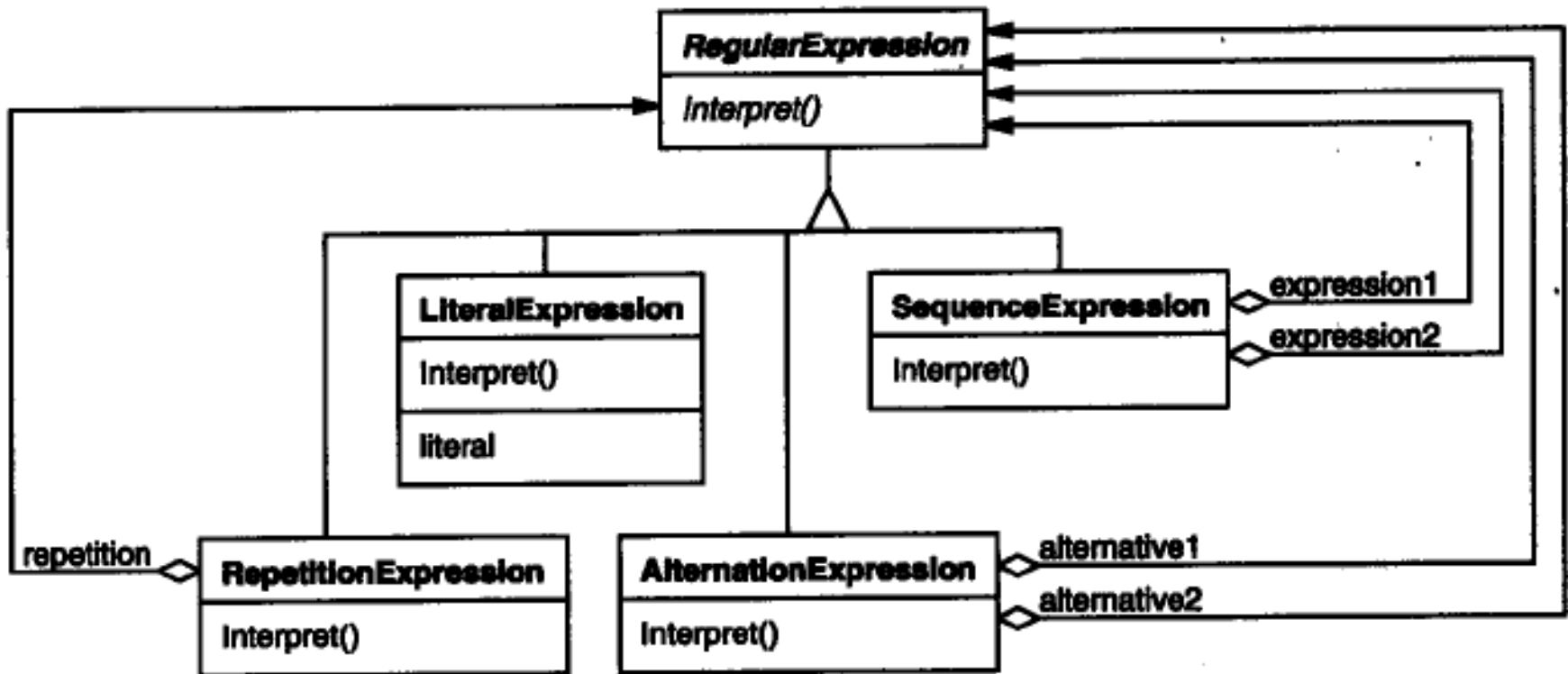
## 2. 动机

---

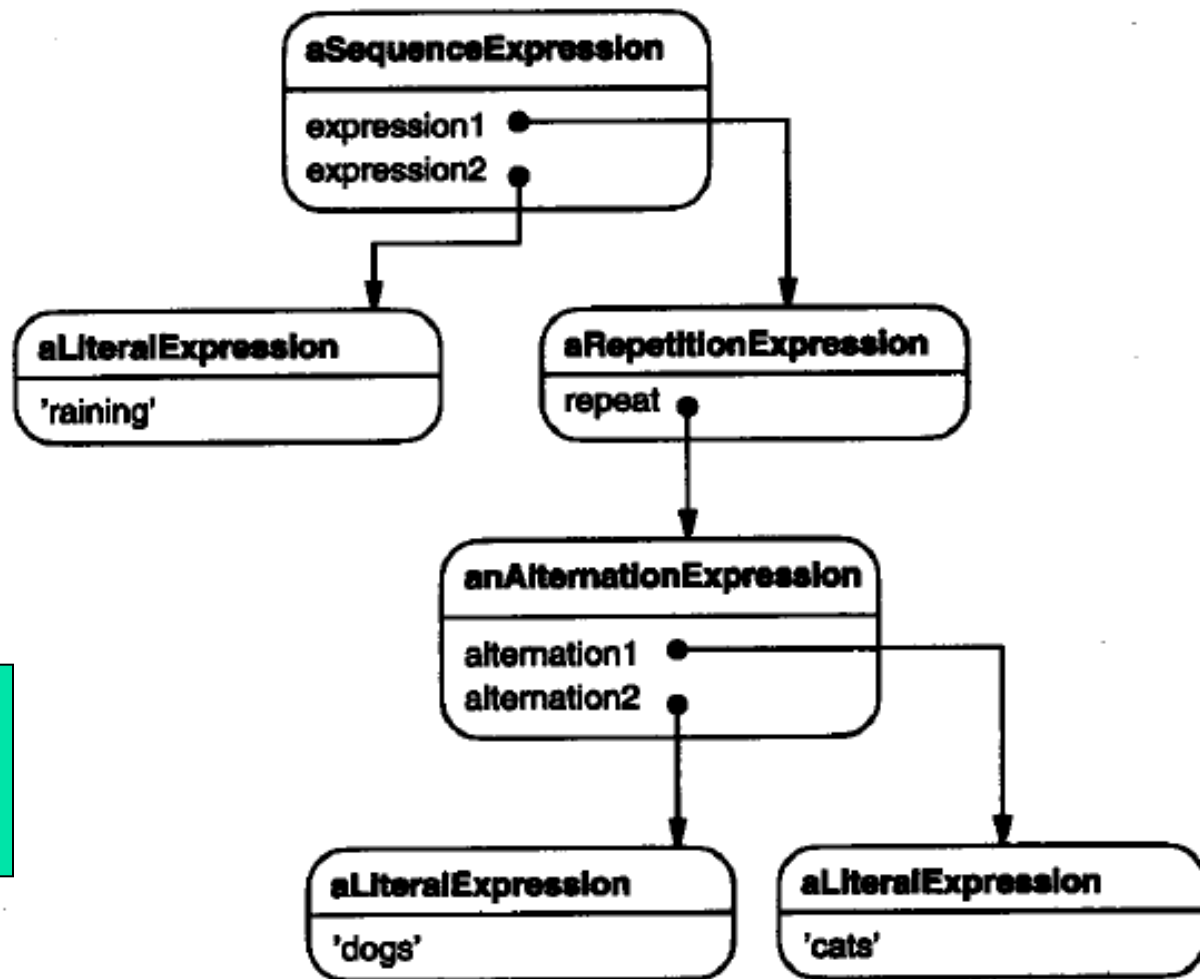
### ■ 正则表达式

```
expression ::= literal | alternation | sequence | repetition |  
              '(' expression ')'  
alternation ::= expression '|' expression  
sequence ::= expression '&' expression  
repetition ::= expression '*'  
literal ::= 'a' | 'b' | 'c' | ... { 'a' | 'b' | 'c' | ... }*
```

## 2. 动机



## 2. 动机



抽象语法树  
`raining & (dogs | cats) *`



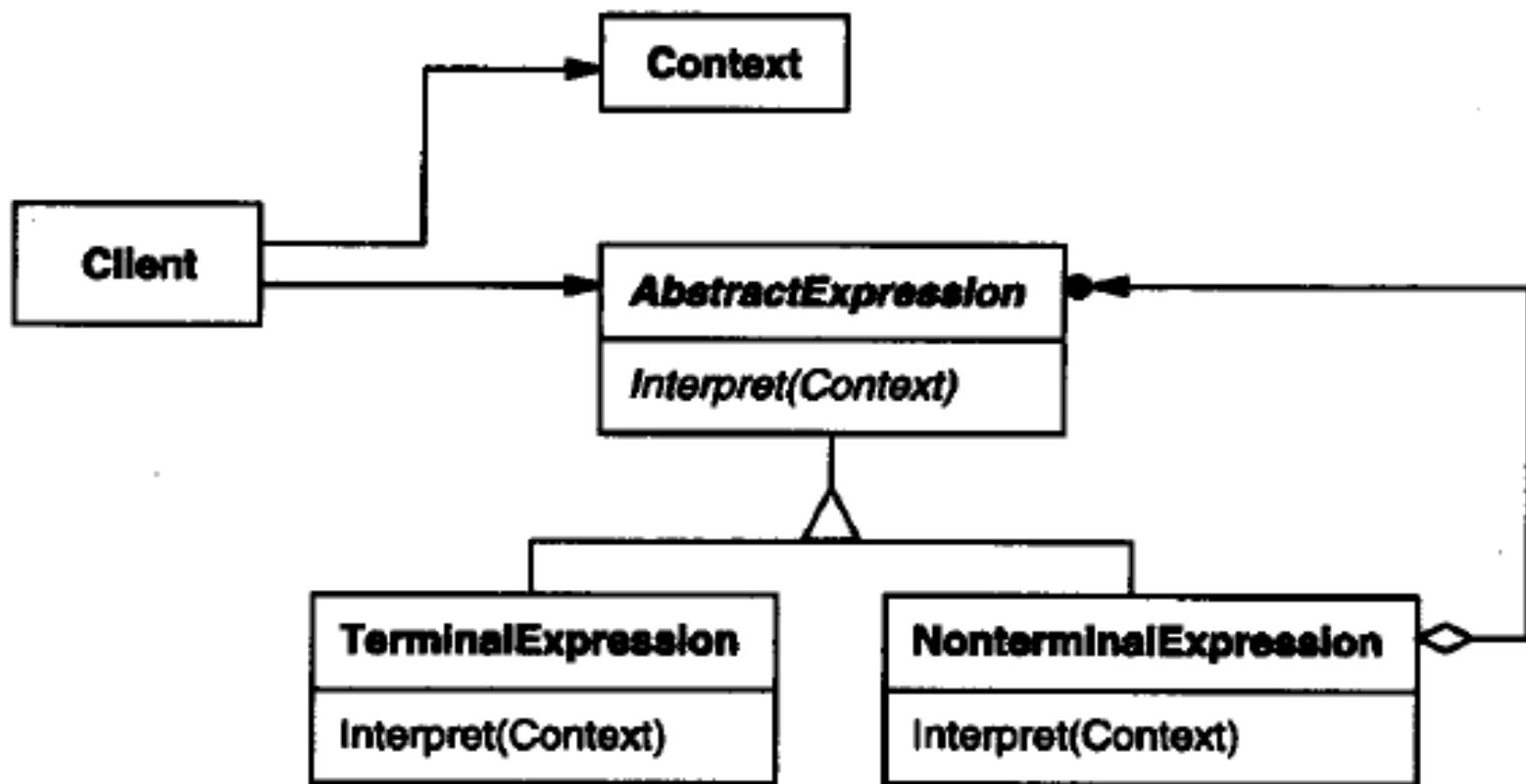


## 3.适用性

---

- 该文法简单
  - 对于复杂的文法，需要使用语法分析程序生成器之类的
- 效率不是一个关键问题

## 4. 结构图





## 5. 参与者

---

- **AbstractExpression (抽象表达式)**
  - 声明一个抽象的解释操作，这个接口被抽象语法树中所有的节点共享
- **TerminalExpression(终结符表达式)**
  - 实现与文法中的终结符相关联的解释操作
  - 一个句子中的每个终结符需要该类的一个实例
- **NonterminalExpression(非终结符表达式)**
  - 针对文法中的规则
- **Context 上下文**
  - 包含解释器之外的一些全局信息
- **Client**
  - 构建抽象语法树
  - 调用解释操作



## 6. 协作

---

- Client构建(或被给定)一个句子
  - 它是NonterminalExpression 和TerminalExpression 对象的一个抽象语法树
  - 然后初始化上下文并调用解释操作
- 每一非终结符节点定义相应子表达式的解释操作。终结符节点的解释操作是递归的基础
- 每一节点的“解释操作”作用Context来存储和访问解释器的状态



## 7. 效果

---

### 1) 易于改变和扩展文法

- 使用继承来改变或扩展文法

### 2) 易于实现文法

- 易于编写抽象语法树中各个节点对应的类
- 也可用编译器或语法分析程序生成器自动生成



## 7. 效果

---

### 3) 复杂的文法难以维护

- 解释器模式中，每个规则至少定义一个类

### 4) 增加了新的解释表达式的方式

- `AbstractExpression`上除了`Interpret`操作外，可以增加新的操作
- 可以考虑`Visitor`模式



## 8.实现

---

### 1)创建抽象语法树

- 解释器模式不涉及语法分析
  - 未解释如何创建一个抽象语法树
- 可行的方式
  - 用一个表驱动的语法分析程序来生成
  - 用手写的(递归下降法)语法分析程序创建
  - 直接由Client提供



## 8. 实现

---

### 2) 定义解释操作

- 不一定要在表达式类中定义解释操作
- 可使用Visitor模式

### 3) 以Flyweight模式共享终结符





## 9.代码示例

### ■ 对布尔表达式进行操作和求值

```
BooleanExp ::= VariableExp | Constant | OrExp | AndExp | NotExp |  
              '(' BooleanExp ')'  
AndExp ::= BooleanExp 'and' BooleanExp  
OrExp  ::= BooleanExp 'or' BooleanExp  
NotExp ::= 'not' BooleanExp  
Constant ::= 'true' | 'false'  
VariableExp ::= 'A' | 'B' | ... | 'X' | 'Y' | 'Z'
```

两个操作:

1. 求值
2. 替换(replace): 用一个表达式来替换一个变量以产生新的表达式



## 9.代码示例

---

```
class BooleanExp {  
public:  
    BooleanExp();  
    virtual ~BooleanExp();
```

父类

```
    virtual bool Evaluate(Context&) = 0;  
    virtual BooleanExp* Replace(const char*, BooleanExp&) = 0;  
    virtual BooleanExp* Copy() const = 0;  
};
```

```
class Context {  
public:  
    bool Lookup(const char*) const;  
    void Assign(VariableExp*, bool);  
};
```

上下文



## 9.代码示例

```
class VariableExp : public BooleanExp {
public:
    VariableExp(const char*);
    virtual ~VariableExp();

    virtual bool Evaluate(Context&);
    virtual BooleanExp* Replace(const char*, BooleanExp&);
    virtual BooleanExp* Copy() const;
private:
    char* _name;
};

VariableExp::VariableExp (const char* name) {
    _name = strdup(name);
}

bool VariableExp::Evaluate (Context& aContext) {
    return aContext.Lookup(_name);
}

BooleanExp* VariableExp::Copy () const {
    return new VariableExp(_name);
}
```

变量

替换

```
BooleanExp* VariableExp::Replace (
    const char* name, BooleanExp& exp
) {
    if (strcmp(name, _name) == 0) {
        return exp.Copy();
    } else {
        return new VariableExp(_name);
    }
}
```



## 9.代码示例

```
class AndExp : public BooleanExp {
public:
    AndExp(BooleanExp*, BooleanExp*);
    virtual ~AndExp();

    virtual bool Evaluate(Context&);
    virtual BooleanExp* Replace(const char*, BooleanExp&);
    virtual BooleanExp* Copy() const;
private:
    BooleanExp* _operand1;
    BooleanExp* _operand2;
};

AndExp::AndExp (BooleanExp* op1, BooleanExp* op2) {
    _operand1 = op1;
    _operand2 = op2;
}

bool AndExp::Evaluate (Context& aContext) {
    return
        _operand1->Evaluate(aContext) &&
        _operand2->Evaluate(aContext);
}
```

And操作

求值



## 9.代码示例

---

```
BooleanExp* AndExp::Copy () const {  
    return  
        new AndExp(_operand1->Copy(), _operand2->Copy());  
}
```

And操作

```
BooleanExp* AndExp::Replace (const char* name, BooleanExp& exp) {  
    return  
        new AndExp(  
            _operand1->Replace(name, exp),  
            _operand2->Replace(name, exp)  
        );  
}
```



## 9.代码示例

---

`(true and x) or (y and (not x))`

求值

```
BooleanExp* expression;  
Context context;  
  
VariableExp* x = new VariableExp("X");  
VariableExp* y = new VariableExp("Y");  
  
expression = new OrExp(  
    new AndExp(new Constant(true), x),  
    new AndExp(y, new NotExp(x))  
);  
  
context.Assign(x, false);  
context.Assign(y, true);  
  
bool result = expression->Evaluate(context);
```



## 9.代码示例

---

### ■ 替换

```
VariableExp* z = new VariableExp("Z");  
NotExp not_z(z);  
  
BooleanExp* replacement = expression->Replace("Y", not_z);  
  
context.Assign(z, true);  
  
result = replacement->Evaluate(context);
```



# 11. 相关模式

---

- Composite模式
  - 抽象语法树是复合模式的实例
- Flyweight模式
  - 如何在抽象语法树中共享终结符
- Iterator
  - 解释器用一个迭代器进行遍历
- Visitor
  - 将解释操作独立出来





## 5.4 Iterator 迭代器

---



# Iterator模式

---

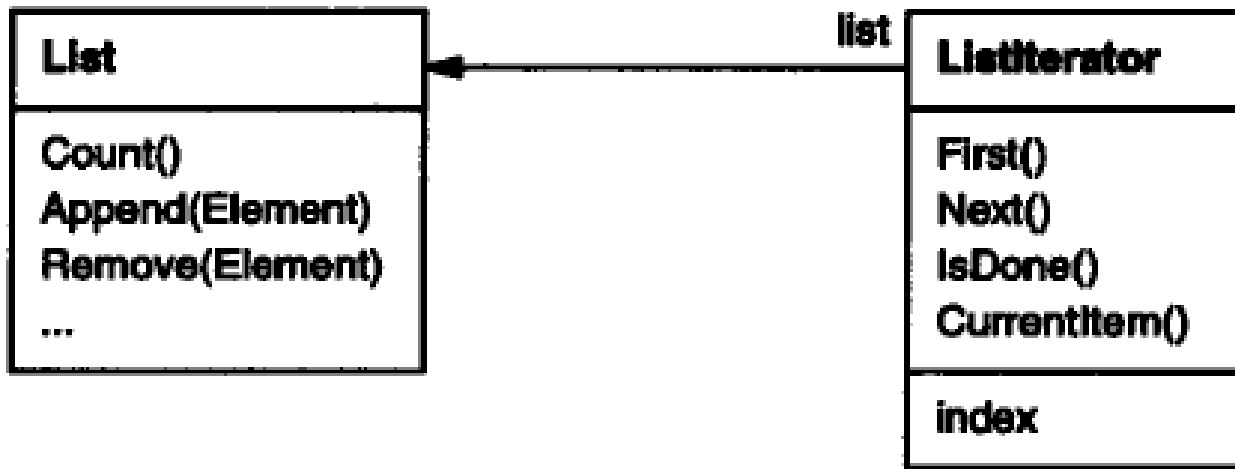
## 1. 意图

- 提供一种方法顺序访问一个聚合对象中各个元素
- 不需暴露该对象的内部表示

## 2. 别名

游标 (Cursor)

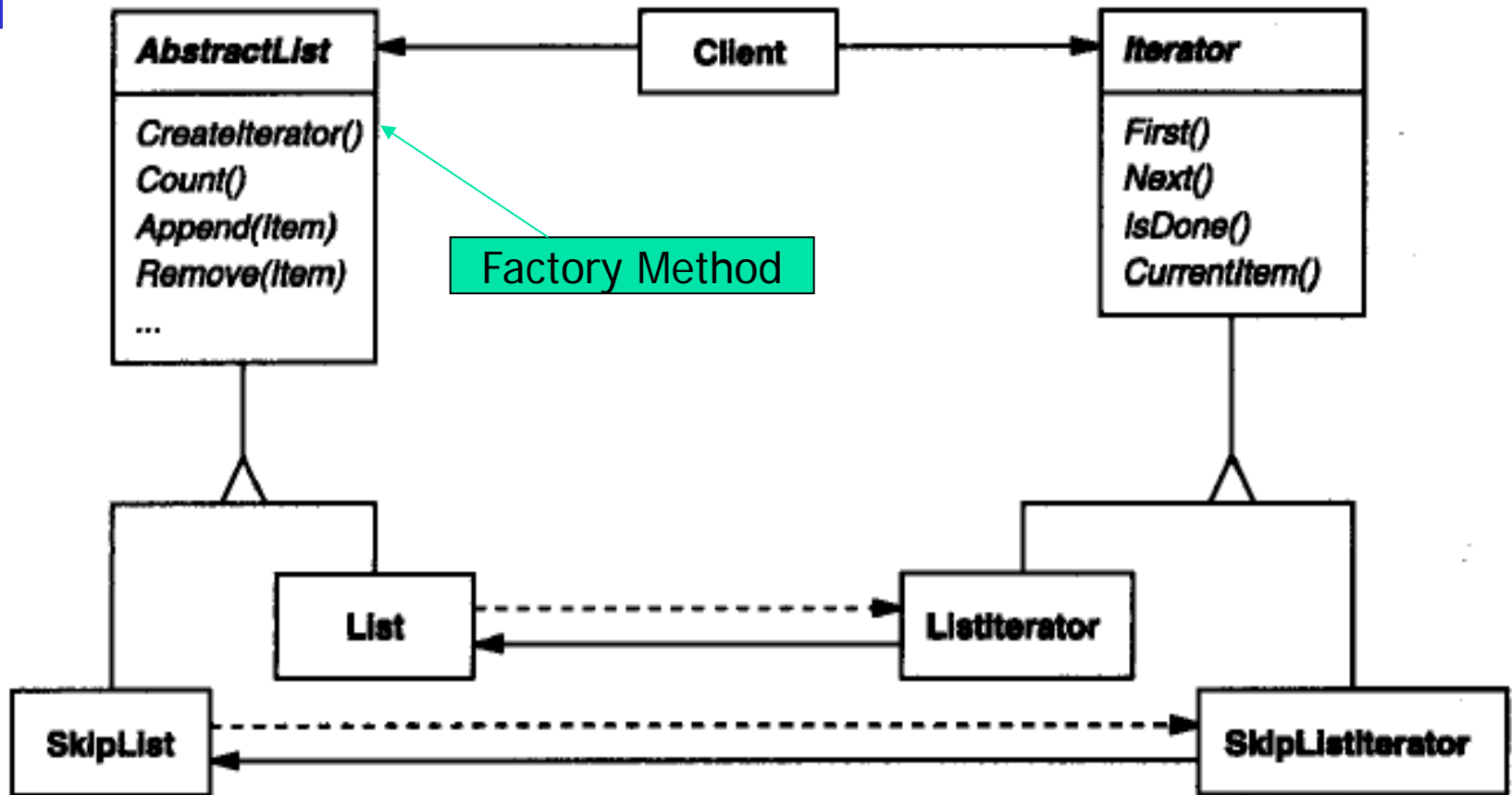
### 3. 动机



将“对列表的访问和遍历”从列表对象中分离出来

此时，迭代器和列表耦合

### 3. 动机



多态迭代(polymorphic iteration)

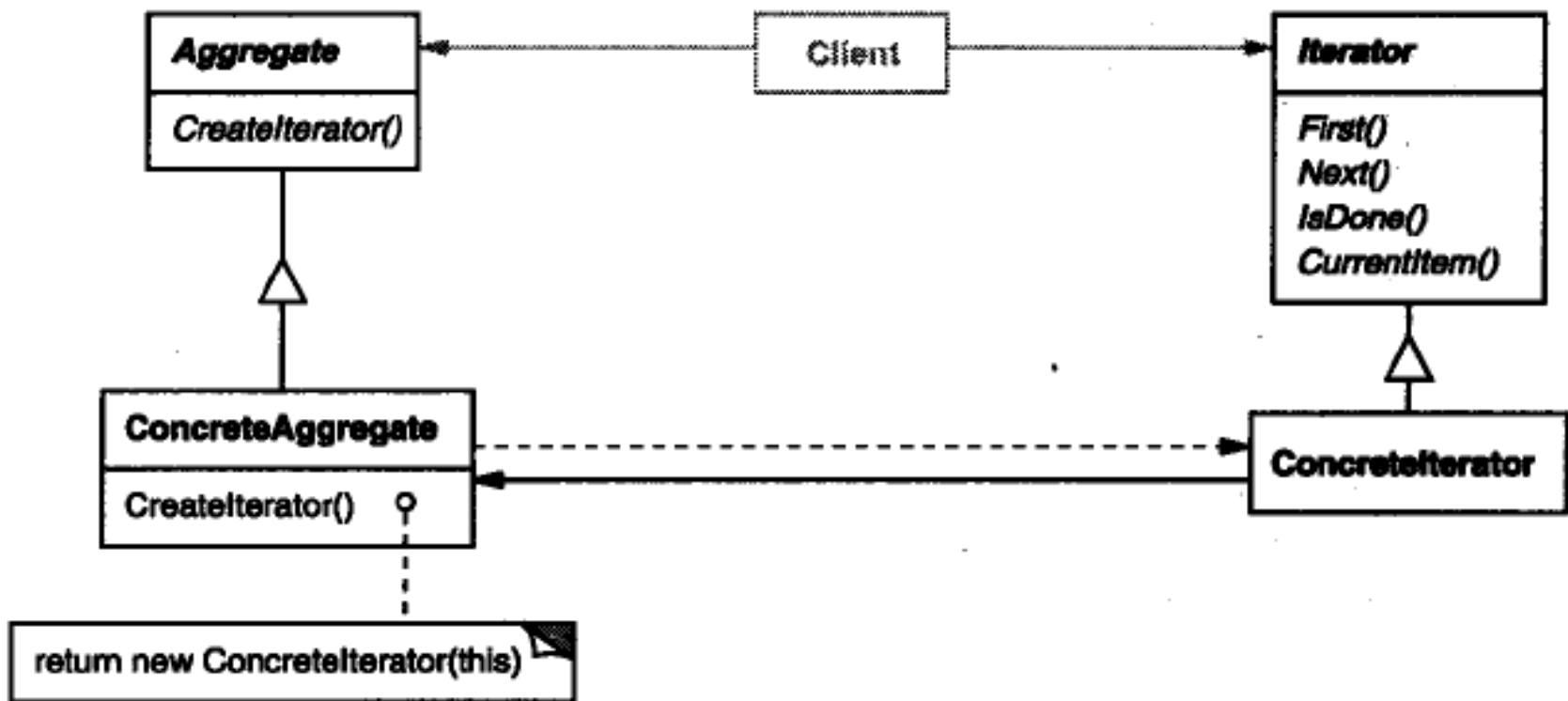


## 4. 适用性

---

- 访问一个聚合对象的内容而无需暴露它的内部表示
- 支持对聚合对象的多种遍历
- 为遍历不同的聚合结构提供一个统一的接口(即, 支持多态迭代)

## 5. 结构





## 6. 参与者

---

- **Iterator**
  - 定义访问和遍历元素的接口
- **ConcreteIterator**
  - 实现Iterator接口
  - 对聚合对象遍历时追踪当前位置
- **Aggregate**
  - 定义创建相应Iterator对象的接口(CreateIterator)
- **ConcreteAggregate**
  - 实现Aggregate接口
  - CreateIterator操作返回一个适当的Iterator实例



## 7.协作

---

- ConcreteIterator跟踪聚合中的当前对象，并能够计算出待遍历的后继对象





## 8.效果

---

- 1) 支持以不同的方式遍历一个聚合
  - 更换迭代器实例即可处理不同的遍历算法
- 2) 迭代器简化了聚合的接口
  - 聚合本身不需提供遍历接口
- 3) 在同一个聚合上可以有多个遍历
  - 每个迭代器保持它自己的遍历状态



## 9. 实现

---

### 1) 谁控制该迭代 迭代器 or client ?

- 外部迭代器(external iterator) client控制
  - Client需要主动推进遍历的步伐(next)
  - 使用灵活
- 内部迭代器(internal iterator) 迭代器控制
  - Client向iterator提交一个操作，迭代器对聚合中每个元素实施该操作
  - 使用简单，client不需考虑迭代逻辑



## 9.实现

---

### 2) 谁定义遍历算法 迭代器 or 聚合

- 聚合中定义遍历算法

- 用迭代器存储当前迭代的状态

- 此时迭代器仅用来只是当前位置，称为“游标” Cursor

- 迭代器负责遍历算法

- 优点：易于在相同聚合上使用不同的迭代算法，或在不同的聚合上重用相同的算法
  - 缺点：遍历算法可能需要访问聚合的私有变量，因此将遍历算法放入迭代器可能会破坏聚合的封装性



## 9. 实现

---

### 3) 迭代器健壮程度如何

- 遍历的同时存在修改(增加或删除聚合元素)可能导致遍历出现问题
- 简单解决方法:
  - 遍历时拷贝聚合, 然后对拷贝进行遍历
  - 问题: 代价太高
- 健壮的迭代器(robust iterator)
  - 保证插入或删除操作不会干扰遍历, 且不需拷贝聚合
  - 方案:
    - 向聚合注册迭代器
    - 聚合修改时, 调整迭代器内部状态, 或在内部维护额外信息以保证正确遍历



## 9. 实现

---

### 4) 附加的迭代器操作

- 迭代器最小接口：
  - First , Next , IsDone, CurrentItem
- 其他操作
  - Previous : 前一个
  - SkipTo : 定位



## 9.实现

---

### 5) 在c++中使用多态的迭代器

- 多态迭代器使用
  - 使用Factory Method动态分配迭代器对象(有代价)
  - 一般情况下，可以直接在栈中分配具体的迭代器
- 问题：Client必须负责删除多态迭代器
  - 使用Proxy模式以便保证迭代器一定被删除



## 9.实现

---

### 6) 迭代器可有特权访问

- 迭代器和聚合紧密耦合
  - C++中，可以将迭代器作为聚合类的friend类
- 问题：使用这种方式，增加新的遍历时，需要在聚合中加入新的友元
- 解决：
  - 迭代器类中包含一些protected操作访问聚合类的非public成员
  - 迭代器子类使用这些protected操作



## 9. 实现

---

### 7) 用于复合对象Composite的迭代器

- 在递归聚合结构上
  - 外部迭代器可能难以实现
    - 需要为跟踪当前的对象存储一个路径
  - 使用内部迭代器
    - 内部迭代器进行递归调用
    - 路径信息隐式存储在调用栈中
- 如果节点提供访问其他节点的接口(兄弟、父、子), 则可以使用基于游标的迭代器
  - 用游标跟踪当前节点





## 9. 实现

---

### 8) 空迭代器      NullIterator

- 退化的迭代器，有助于处理边界条件
  - IsDone总返回true
- 有助于遍历树形结构
  - 每个节点返回一个Iterator对象
  - 叶节点方法NullIterator对象



# 10. 代码示例

---

## 1) 列表与迭代接口

```
template <class Item>
class List {
public:
    List(long size = DEFAULT_LIST_CAPACITY);

    long Count() const;
    Item& Get(long index) const;
    // ...
};
```

访问元素接口，足以进行遍历操作  
因此不再提供专门的特权操作

```
template <class Item>
class Iterator {
public:
    virtual void First() = 0;
    virtual void Next() = 0;
    virtual bool IsDone() const = 0;
    virtual Item CurrentItem() const = 0;
protected:
    Iterator();
};
```



## 2) 迭代器子类

```
template <class Item>
class ListIterator : public Iterator<Item> {
public:
    ListIterator(const List<Item>* aList);
    virtual void First();
    virtual void Next();
    virtual bool IsDone() const;
    virtual Item CurrentItem() const;
private:
    const List<Item>* _list;
    long _current;
};
```

```
template <class Item>
ListIterator<Item>::ListIterator (
    const List<Item>* aList
) : _list(aList), _current(0) {
}
```

```
template <class Item>
void ListIterator<Item>::First () {
    _current = 0;
}
```

```
template <class Item>
void ListIterator<Item>::Next () {
    _current++;
}
```

```
template <class Item>
bool ListIterator<Item>::IsDone () const {
    return _current >= _list->Count();
}
```

```
template <class Item>
Item ListIterator<Item>::CurrentItem () const {
    if (IsDone()) {
        throw IteratorOutOfBounds;
    }
    return _list->Get(_current);
}
```



### 3)使用迭代器

---

```
void PrintEmployees (Iterator<Employee*>& i) {  
    for (i.First(); !i.IsDone(); i.Next()) {  
        i.CurrentItem()->Print();  
    }  
}
```

```
List<Employee*>* employees;  
// ...  
ListIterator<Employee*> forward(employees);  
ReverseListIterator<Employee*> backward(employees);  
PrintEmployees(forward);  
PrintEmployees(backward);
```

## 4)避免限定于一种特定的列表实现

```
SkipList<Employee*>* employees;  
// ...
```

SkipList提供一个SkipListIterator  
(与具体List实现有关)

```
SkipListIterator<Employee*> iterator(employees);  
PrintEmployees(iterator);
```

其中定义一个Factory Method

```
template <class Item>  
class AbstractList {  
public:  
    virtual Iterator<Item>* CreateIterator() const = 0;  
    // ...  
};
```

也可以单独定义一个类Traversable,  
其中定义一个CreateIterator接口

## 4)避免限定于一种特定的列表实现

```
template <class Item>
Iterator<Item>* List<Item>::CreateIterator () const {
    return new ListIterator<Item>(this);
}
```

AbstractList的子类

不依赖于具体列表表示

```
// we know only that we have an AbstractList
AbstractList<Employee*>* employees;
// ...

Iterator<Employee*>* iterator = employees->CreateIterator();
PrintEmployees(*iterator);
delete iterator;
```

## 5) 保证迭代器被删除

- 使用代理机制，保证释放迭代器

```
template <class Item>
class IteratorPtr {
public:
    IteratorPtr(Iterator<Item>* i): _i(i) { }
    ~IteratorPtr() { delete _i; }

    Iterator<Item>* operator->() { return _i; }
    Iterator<Item>& operator*() { return *_i; }

private:
    // disallow copy and assignment to avoid
    // multiple deletions of _i:

    IteratorPtr(const IteratorPtr&);
    IteratorPtr& operator=(const IteratorPtr&);

private:
    Iterator<Item>* _i;
};
```

重载操作符

禁止进行copy和赋值操作

## 5) 保证迭代器被删除

Client的使用

```
AbstractList<Employee*>* employees;  
// ...
```

```
IteratorPtr<Employee*> iterator(employees->CreateIterator());  
PrintEmployees(*iterator);
```

离开作用域后， iterator会被释放





## 6) 一个内部的ListIterator

---

- 内部ListIterator
  - 由迭代器来控制迭代
  - 对列表中每个元素实施同一个操作
- 抽象的迭代器，C++中的实现方法
  - 给迭代器传递一个函数指针
  - 依赖子类生成



## 6) 一个内部的ListIterator

---

```
template <class Item>
class ListTraverser {
public:
    ListTraverser(List<Item>* aList);
    bool Traverse();
protected:
    virtual bool ProcessItem(const Item&) = 0;
private:
    ListIterator<Item> _iterator;
};
```

供子类重定义的操作





## 6) 一个内部的ListIterator

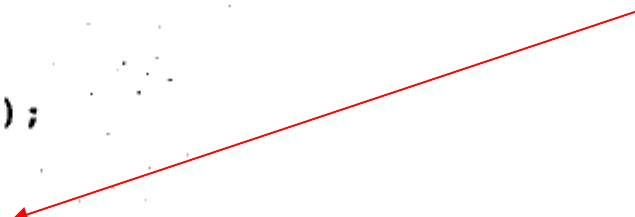
---

```
template <class Item>
ListTraverser<Item>::ListTraverser (
    List<Item>* aList
) : _iterator(aList) { }

template <class Item>
bool ListTraverser<Item>::Traverse () {
    bool result = false;

    for (
        _iterator.First();
        !_iterator.IsDone();
        _iterator.Next()
    ) {
        result = ProcessItem(_iterator.CurrentItem());

        if (result == false) {
            break;
        }
    }
    return result;
}
```



## 6) 一个内部的ListIterator

```
class PrintNEmployees : public ListTraverser<Employee*> {
public:
    PrintNEmployees(List<Employee*>* aList, int n) :
        ListTraverser<Employee*>(aList),
        _total(n), _count(0) { }

protected:
    bool ProcessItem(Employee* const&);
private:
    int _total;
    int _count;
};

bool PrintNEmployees::ProcessItem (Employee* const& e) {
    _count++;
    e->Print();
    return _count < _total;
}
```

子类派生(重载ProcessItem操作)



## 6) 一个内部的ListIterator

---

```
List<Employee*>* employees;  
// ...
```

```
PrintNEmployees pa(employees, 10);  
pa.Traverse();
```

使用内部Iterator

```
ListIterator<Employee*> i(employees);  
int count = 0;  
for (i.First(); !i.IsDone(); i.Next()) {  
    count++;  
    i.CurrentItem()->Print();  
  
    if (count >= 10) {  
        break;  
    }  
}
```

使用外部Iterator

## 6) 一个内部的ListIterator

```
template <class Item>
class FilteringListTraverser {
public:
    FilteringListTraverser(List<Item>* aList);
    bool Traverse();
protected:
    virtual bool ProcessItem(const Item&) = 0;
    virtual bool TestItem(const Item&) = 0;
private:
    ListIterator<Item> _iterator;
};
```

使用内部迭代器封装不同的迭代

只有通过测试的元素才进行处理

```
template <class Item>
void FilteringListTraverser<Item>::Traverse () {
    bool result = false;

    for (
        _iterator.First();
        !_iterator.IsDone();
        _iterator.Next()
    ) {
        if (TestItem(_iterator.CurrentItem())) {
            result = ProcessItem(_iterator.CurrentItem());
            if (result == false) {
                break;
            }
        }
    }
    return result;
}
```



## 12. 相关模式

---

- Composite
  - 迭代器的应用对象
- Factory Method
  - 多态迭代器使用Factory Method
- Memento
  - 迭代器使用memento来存储迭代的状态



## 5.5 Mediator 中介者

---





# 1. 意图

---

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

## 2. 动机

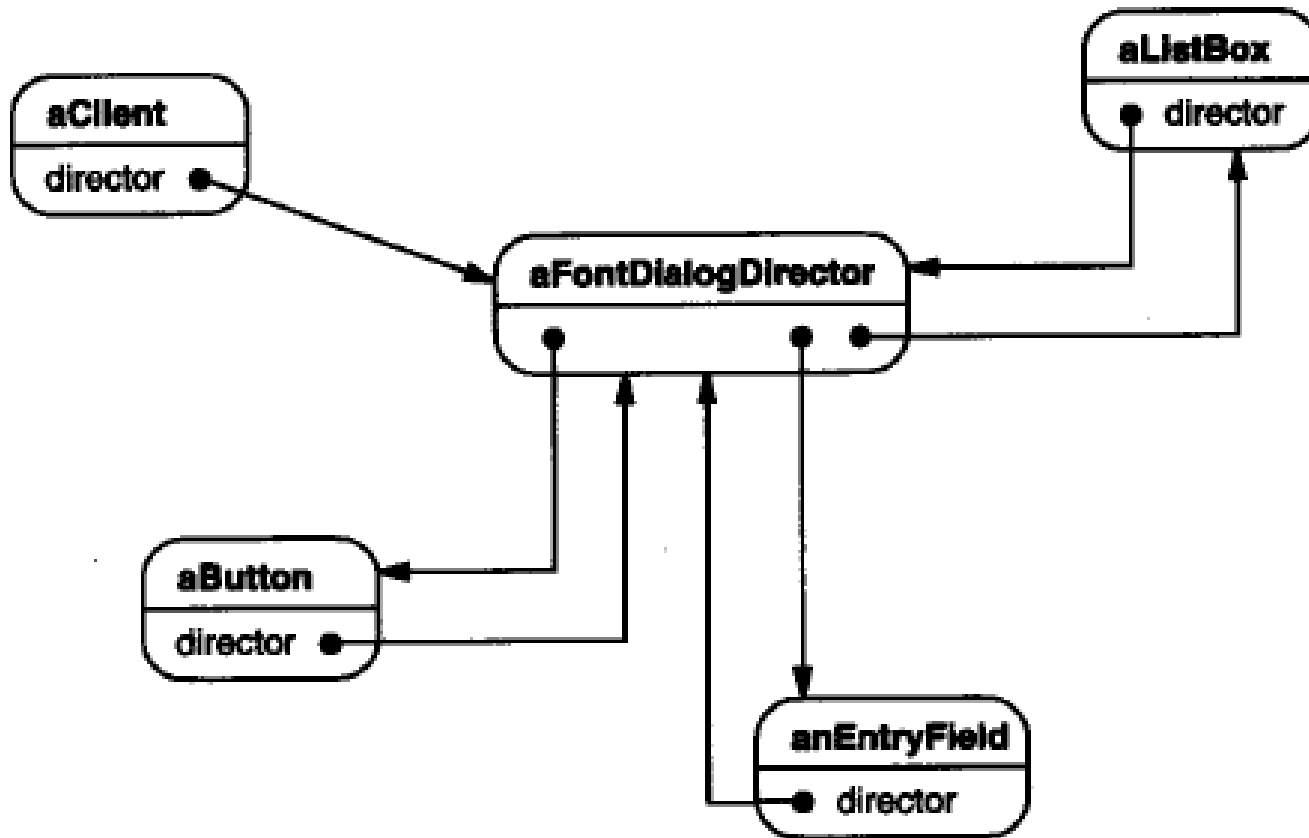
- 对象间的相互连接会降低可复用性



对于不同的对话框，有不同的窗口组件间的依赖关系。

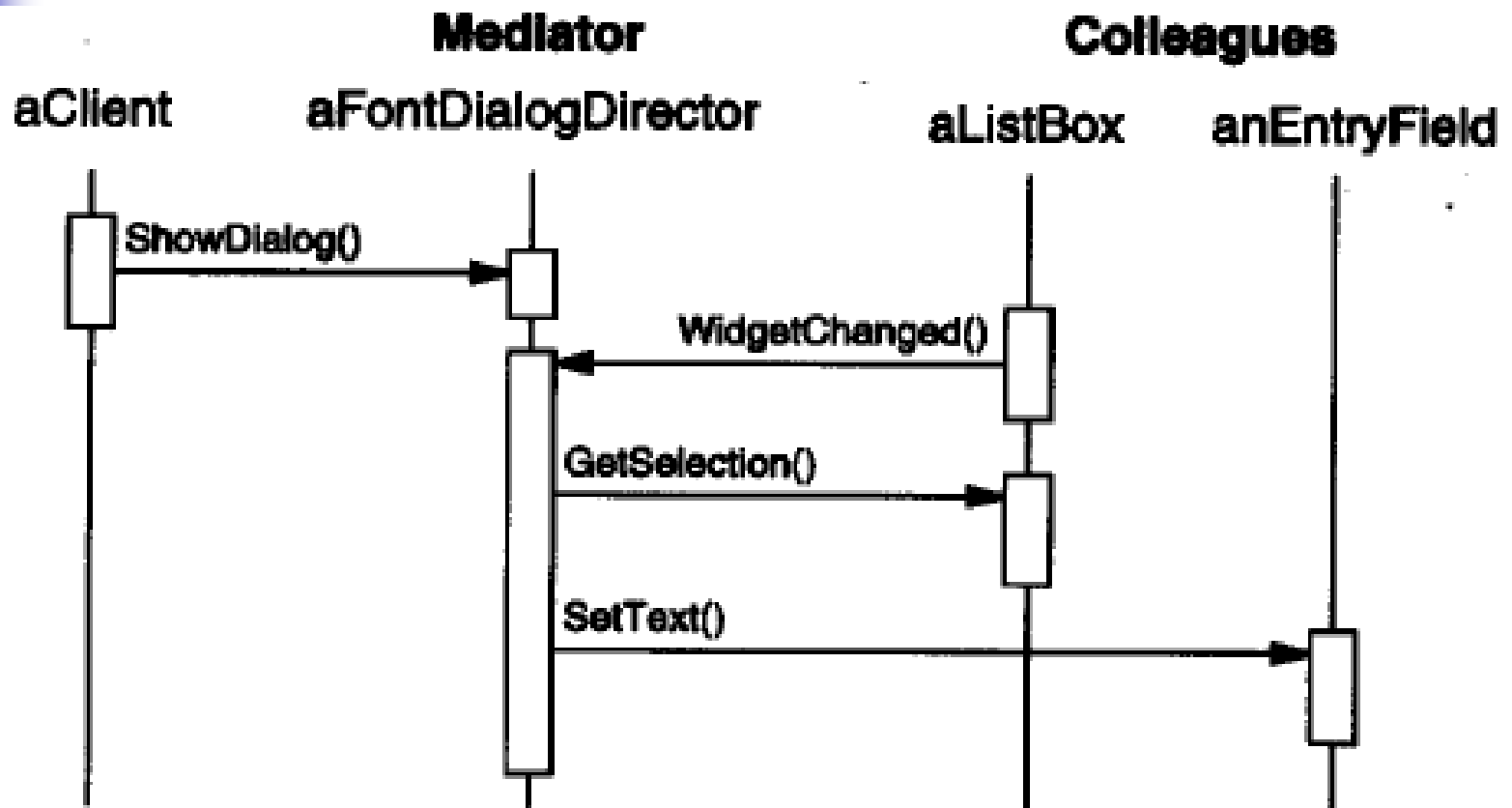
两个对话框显示相同类型的窗口组件...

## 2. 动机



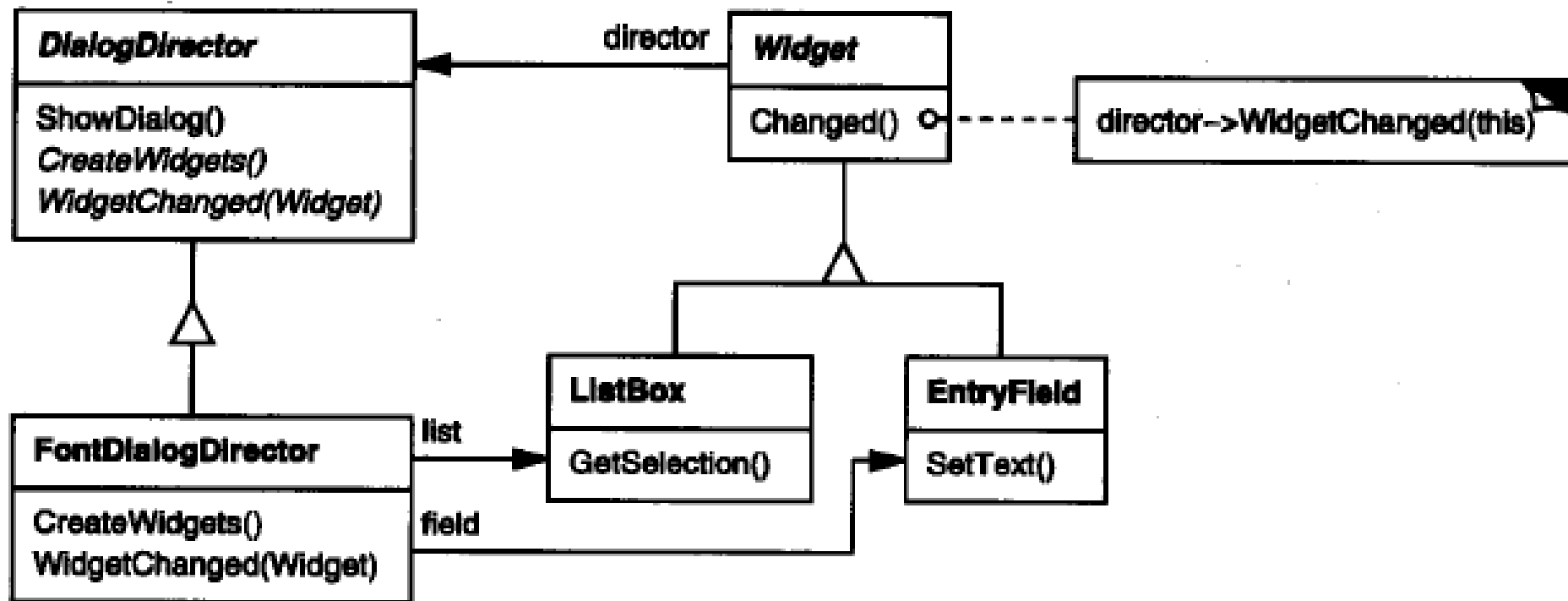
将集体行为封装在一个单独的中介者(mediator)对象中

## 2. 动机



各对象如何协作处理一个列表框中选项的变化

## 2. 动机



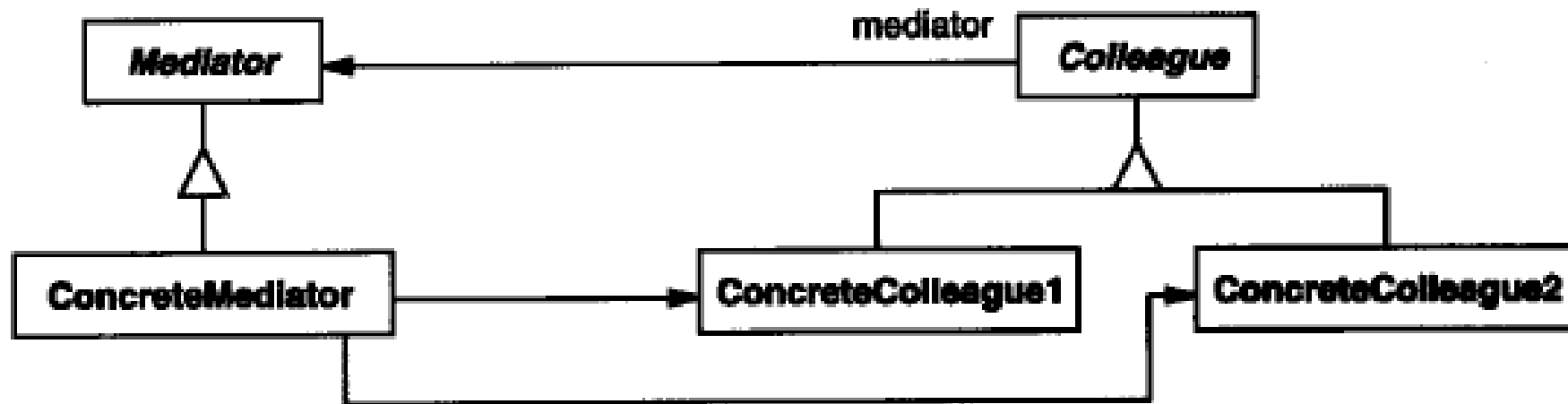


## 3.适用性

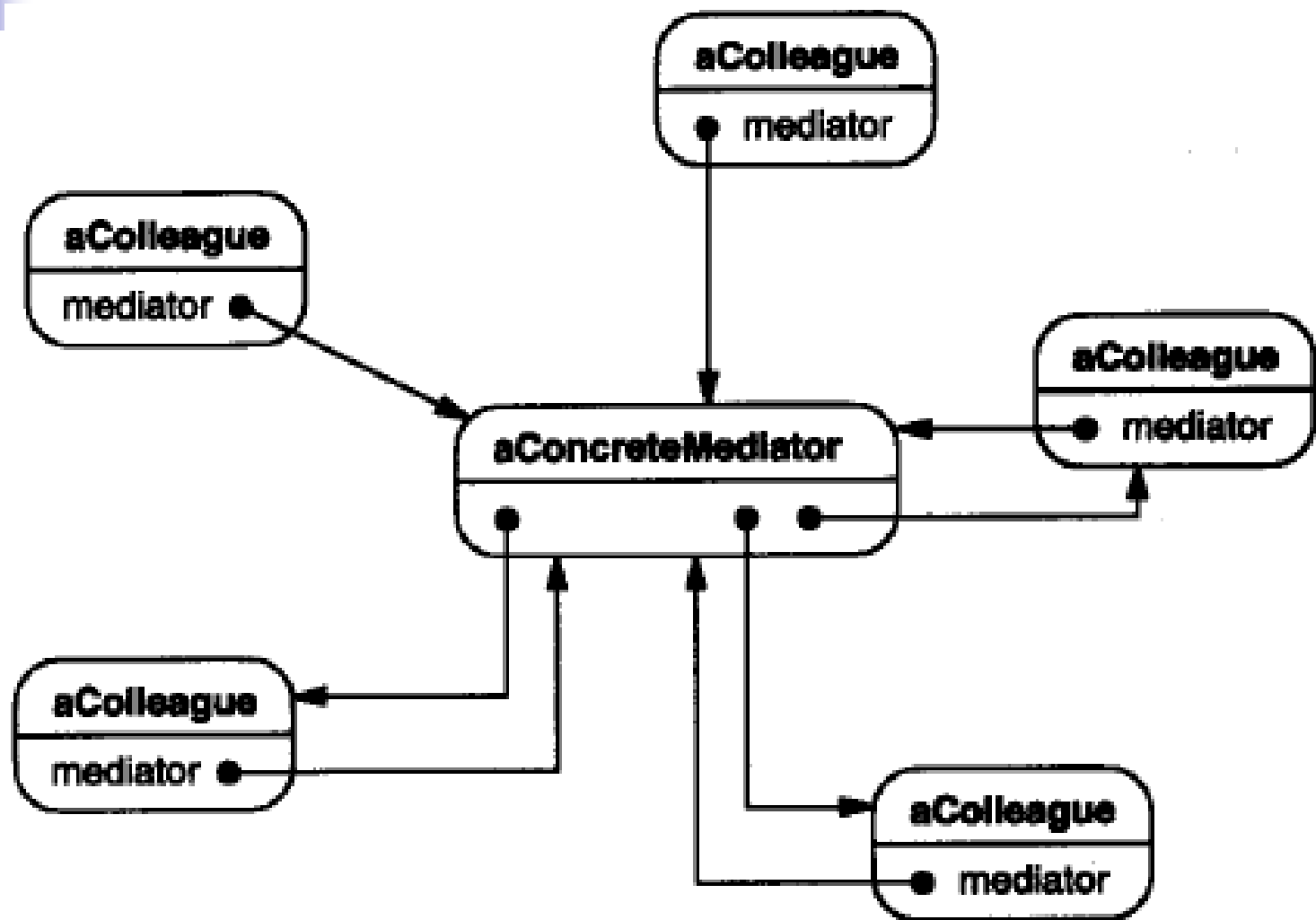
---

- 一组对象以定义良好但是**复杂**的方式进行通信。产生的依赖关系结构混乱且难以理解。
- 一个对象引用其他很多对象并且直接与这些对象通信，导致难以复用该对象
- 想定制一个分布在多个类的行为，而又不想生成太多的子类。

## 4. 结构



## 4. 结构







## 5. 参与者

---

- Mediator 中介者
  - 定义一个接口，用于和各同事(colleague)对象通信
- ConcreteMediator
  - 具体中介者通过协调各同事对象实现协作行为
  - 了解并维护它的各同事
- Colleague class 同事类
  - 每个同事类都知道它的中介者对象
  - 每个同事对象在需要与其他同事通信时，与它的中介者通信



## 6. 协作

---

- 同事向一个中介者对象发送和接收请求
- 中介者在各同事间适当地转发请求以实现协作行为



## 7.效果

---

### 1)减少了子类生成

- 有利于Colleague类的重用

### 2) 将各Colleague解耦

- 可以独立改变和复用Mediator / Colleague

### 3) 简化了对象协议

- Colleague间的多对多 → Mediator与Colleague间的一对多



## 7.效果

---

### 4) 对对象如何协作进行了抽象

- 用独立的对象处理一个系统的各个对象的交互

### 5) 使控制集中化

- 将交互的复杂性变为中介者的复杂性
- 可能使得中介者本身很复杂，难于维护



## 8. 实现

---

### 1) 忽略抽象的Mediator类

- 仅存在一个Mediator时

### 2) Colleague-Mediator通信

- 实现方式1: Observer模式
- 实现方式2
  - 在Mediator中定义一个特殊的通知接口



## 9.代码示例

---

导控者父类

```
class DialogDirector {
public:
    virtual ~DialogDirector();

    virtual void ShowDialog();
    virtual void WidgetChanged(Widget*) = 0;

protected:
    DialogDirector();
    virtual void CreateWidgets() = 0;
};
```

## 9.代码示例

窗口组件父类

```
class Widget {  
public:  
    Widget(DialogDirector*);  
    virtual void Changed();  
  
    virtual void HandleMouse(MouseEvent& event);  
    // ...  
private:  
    DialogDirector* _director;  
};
```

← 导控者指针

onChange事件

```
void Widget::Changed () {  
    _director->WidgetChanged(this);  
}
```

```

class ListBox : public Widget {
public:
    ListBox(DialogDirector*);

    virtual const char* GetSelection();
    virtual void SetList(List<char*>* listItems);
    virtual void HandleMouse(MouseEvent& event);
    // ...
};

```

---

```

class EntryField : public Widget {
public:
    EntryField(DialogDirector*);

    virtual void SetText(const char* text);
    virtual const char* GetText();
    virtual void HandleMouse(MouseEvent& event);
    // ...
};

```

---

Widget子类

```

class Button : public Widget {
public:
    Button(DialogDirector*);

    virtual void SetText(const char* text);
    virtual void HandleMouse(MouseEvent& event);
    // ...
};

void Button::HandleMouse (MouseEvent& event) {
    // ...
    Changed();
}

```





## 9.代码示例

---

```
class FontDialogDirector : public DialogDirector {
public:
    FontDialogDirector();
    virtual ~FontDialogDirector();
    virtual void WidgetChanged(Widget*);

protected:
    virtual void CreateWidgets();

private:
    Button* _ok;
    Button* _cancel;
    ListBox* _fontList;
    EntryField* _fontName;
};
```

具体的Mediator



## 9.代码示例

---

```
void FontDialogDirector::CreateWidgets () {  
    _ok = new Button(this);  
    _cancel = new Button(this);  
    _fontList = new ListBox(this);  
    _fontName = new EntryField(this);  
  
    // fill the listBox with the available font names  
  
    // assemble the widgets in the dialog  
}
```

跟踪窗口组件



## 9.代码示例

---

```
void FontDialogDirector::WidgetChanged (
    Widget* theChangedWidget
) {
    if (theChangedWidget == _fontList) {
        _fontName->SetText(_fontList->GetSelection());

    } else if (theChangedWidget == _ok) {
        // apply font change and dismiss dialog
        // ...
    } else if (theChangedWidget == _cancel) {
        // dismiss dialog
    }
}
```

处理组件间的协同



# 11. 相关应用

---

- Facade

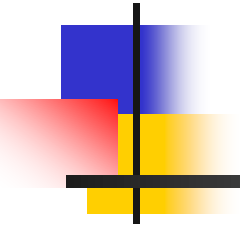
- Facade是单向的
- Mediator是多向的

- Observer

- Colleague可以使用Observer模式与Mediator通信

## 5.6 Memento 备忘录

---





# Memento模式

---

## 1. 意图

- 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。使得以后可以间对象恢复到原先保存的状态

## 2. 别名

Token



### 3. 动机

---

- 记录对象内部状态与保持对象的封装性
- 图形编辑器中连线问题
  - 约束解释系统ConstraintSolver



ConstraintSolver的公共接口可能不足以精确地逆转它对其他对象的作用



## 3. 动机

---

- 使用Memento模式
  - 备忘录memento对象：存储另一个对象(原发器 **originator**)某个瞬间的内部状态
  - 只有原发器可以向备忘录中存取信息，备忘录对其他对象“不可见”
  - 取消操作：
    - 移动操作时，编辑器向ConstraintSolver请求一个Memento
    - ConstraintSolver创建并返回一个Memento
    - 在取消移动操作时，编辑器将Memento送给ConstraintSolver
    - 根据Memento，ConstraintSolver恢复到先前状态



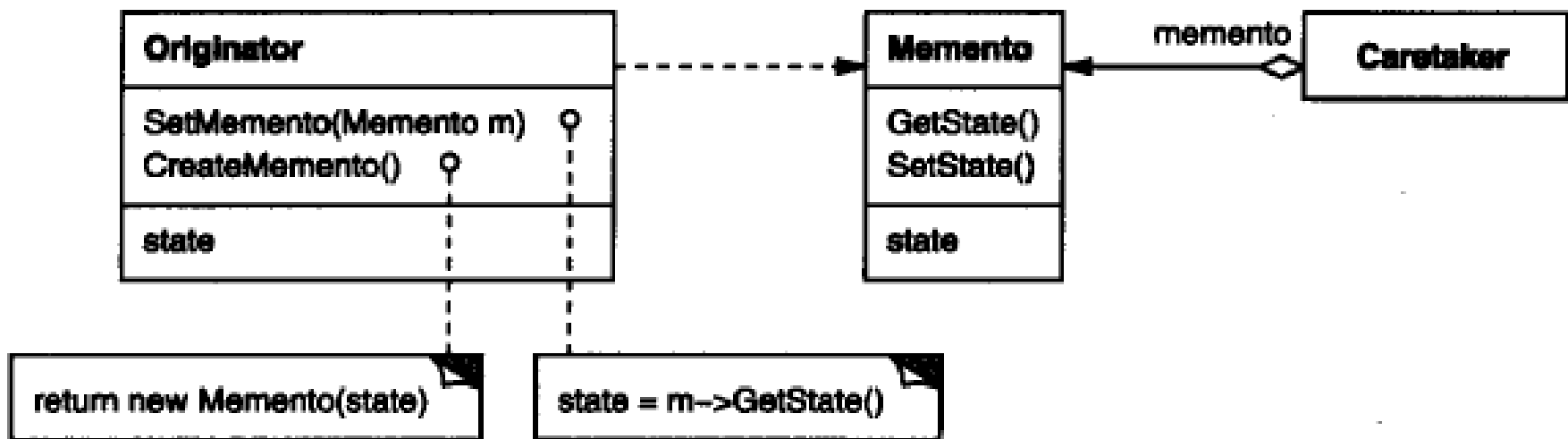


## 4.适用性

---

- 必须保存一个对象在某一个时刻的(部分)状态，以便需要时可以恢复到先前的状态
- 如果用接口来让其他对象直接得到这些状态，将会暴露对象的实现细节并破坏对象的封装性

## 5. 结构





## 6. 参与者

---

- Memento

- 存储原发器对象的内部状态
- 防止原发器以外的对象访问备忘录
  - 窄接口：供管理者(caretaker)访问的
  - 宽接口：供原发器访问的

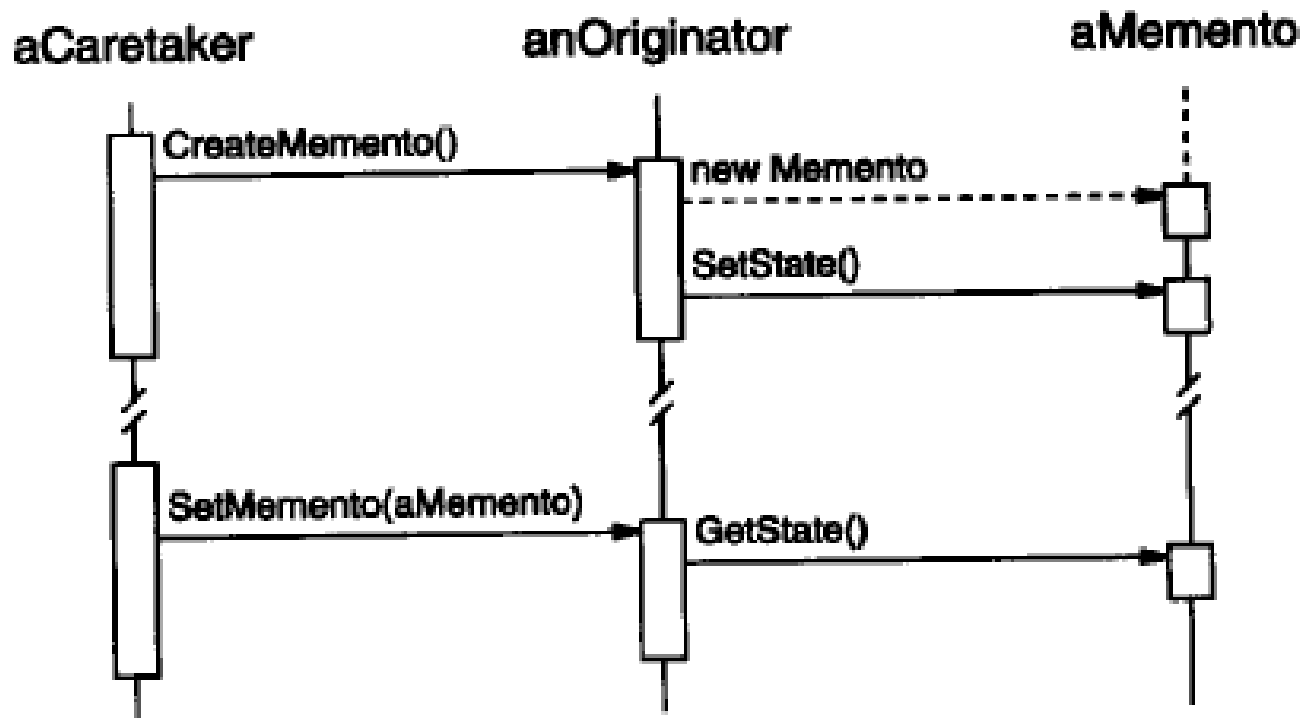
- Originator

- 创建一个Memento，用以记录当时自己的状态
- 使用Memento恢复内部状态

- Caretaker

- 负责保存Memento
- 不能对Memento的内容进行操作或检查

## 7. 协作



Memento是被动的



## 8. 效果

---

- 1) 保持封装边界
- 2) 简化了原发器
  - 由Client进行存储管理
- 3) 使用备忘录可能代价很高
- 4) 定义窄接口和宽接口
  - 有些语言中难以保证只有原发器访问备忘录状态
- 5) 维护备忘录的潜在代价
  - 一个很小的管理器，可能会产生大量的存储开销



## 9.实现

---

### 1)语言支持

- 备忘录中两个接口：宽接口/窄接口
  - 理想的实现语言应支持**两级**的静态保护
- C++实现：
  - 宽接口：private
    - Originator做为Memento的友元
  - 窄接口：public



## 9.实现

---

### 1)语言支持

```
class State;

class Originator {
public:
    Memento* CreateMemento();
    void SetMemento(const Memento*);
    // ...
private:
    State* _state;           // internal data structures
    // ...
};

class Memento {
public:
    // narrow public interface
    virtual ~Memento();
private:
    // private members accessible only to Originator
    friend class Originator;
    Memento();

    void SetState(State*);
    State* GetState();
    // ...
private:
    State* _state;
    // ...
};
```



## 9.实现

---

### 2) 存储增量式改变

- 如果备忘录创建及其返回(给它的原发器)的顺序是可预测的
  - 备忘录可仅存储原发器内部状态的增量改变
- 即：只存储改变的部分

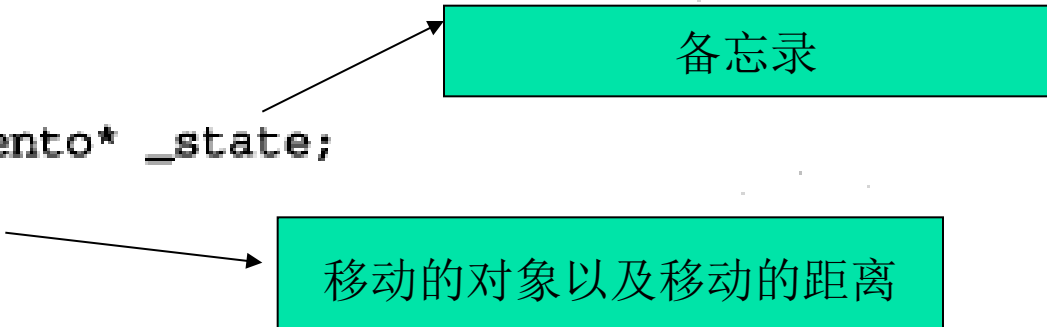




## 10.代码示例

---

```
class Graphic;  
    // base class for graphical objects in the graphical editor  
  
class MoveCommand {  
public:  
    MoveCommand(Graphic* target, const Point& delta);  
    void Execute();  
    void Unexecute();  
private:  
    ConstraintSolverMemento* _state;  
    Point _delta;  
    Graphic* _target;  
};
```



备忘录

移动的对象以及移动的距离

Command对象

# 10.代码示例

原发器

```
class ConstraintSolver {  
public:  
    static ConstraintSolver* Instance();
```

Singleton

根据约束求解

```
    void Solve();  
    void AddConstraint(  
        Graphic* startConnection, Graphic* endConnection  
    );  
    void RemoveConstraint(  
        Graphic* startConnection, Graphic* endConnection  
    );  
    ConstraintSolverMemento* CreateMemento();  
    void SetMemento(ConstraintSolverMemento*);
```

操作Memento

```
private:  
    // nontrivial state and operations for enforcing  
    // connectivity semantics  
};
```

约束  
管理



## 10.代码示例

Memento

```
class ConstraintSolverMemento {  
public:  
    virtual ~ConstraintSolverMemento();  
private:  
    friend class ConstraintSolver;  
    ConstraintSolverMemento();  
  
    // private constraint solver state  
};
```



## 10.代码示例

---

```
void MoveCommand::Execute () {
    ConstraintSolver* solver = ConstraintSolver::Instance();
    _state = solver->CreateMemento(); // create a memento
    _target->Move(_delta);
    solver->Solve();
}

void MoveCommand::Unexecute () {
    ConstraintSolver* solver = ConstraintSolver::Instance();
    _target->Move(-_delta);
    solver->SetMemento(_state); // restore solver state
    solver->Solve();
}
```

Client使用



# 11. 已知应用

## 基于备忘录的迭代

---

```
template <class Item>
class Collection {
public:
    Collection();

    IterationState* CreateInitialState();
    void Next(IterationState*);
    bool IsDone(const IterationState*) const;
    Item CurrentItem(const IterationState*) const;
    IterationState* Copy(const IterationState*) const;

    void Append(const Item&);
    void Remove(const Item&);
    // ...
};
```

# 11.已知应用

## 基于备忘录的迭代

---

```
class ItemType {  
public:  
    void Process();  
    // ...  
};
```

```
Collection<ItemType*> aCollection;  
IterationState* state;
```

```
state = aCollection.CreateInitialState();
```

```
while (!aCollection.IsDone(state)) {  
    aCollection.CurrentItem(state)->Process();  
    aCollection.Next(state);  
}  
delete state;
```

# 11. 已知应用

## 基于备忘录的迭代

---

- 1) 同一个集合上可以有多个状态一起工作
- 2) 不需要为迭代而破坏集合的封装性
  - Iterator需要将迭代器作为集合类的友元
  - 这里，Collection是IteratorState的友元



## 12. 相关模式

---

- Command

- 命令可使用备忘录来为可撤销的操作维护状态

- Iterator

- 备忘录可以用于迭代





## 5.7 Observer 观察者

---



# Observer模式

---

## 1. 意图

- 定义对象间的一对多的依赖关系
- 当一个对象的状态发生变化时，所有依赖于它的对象都得到通知并被更新

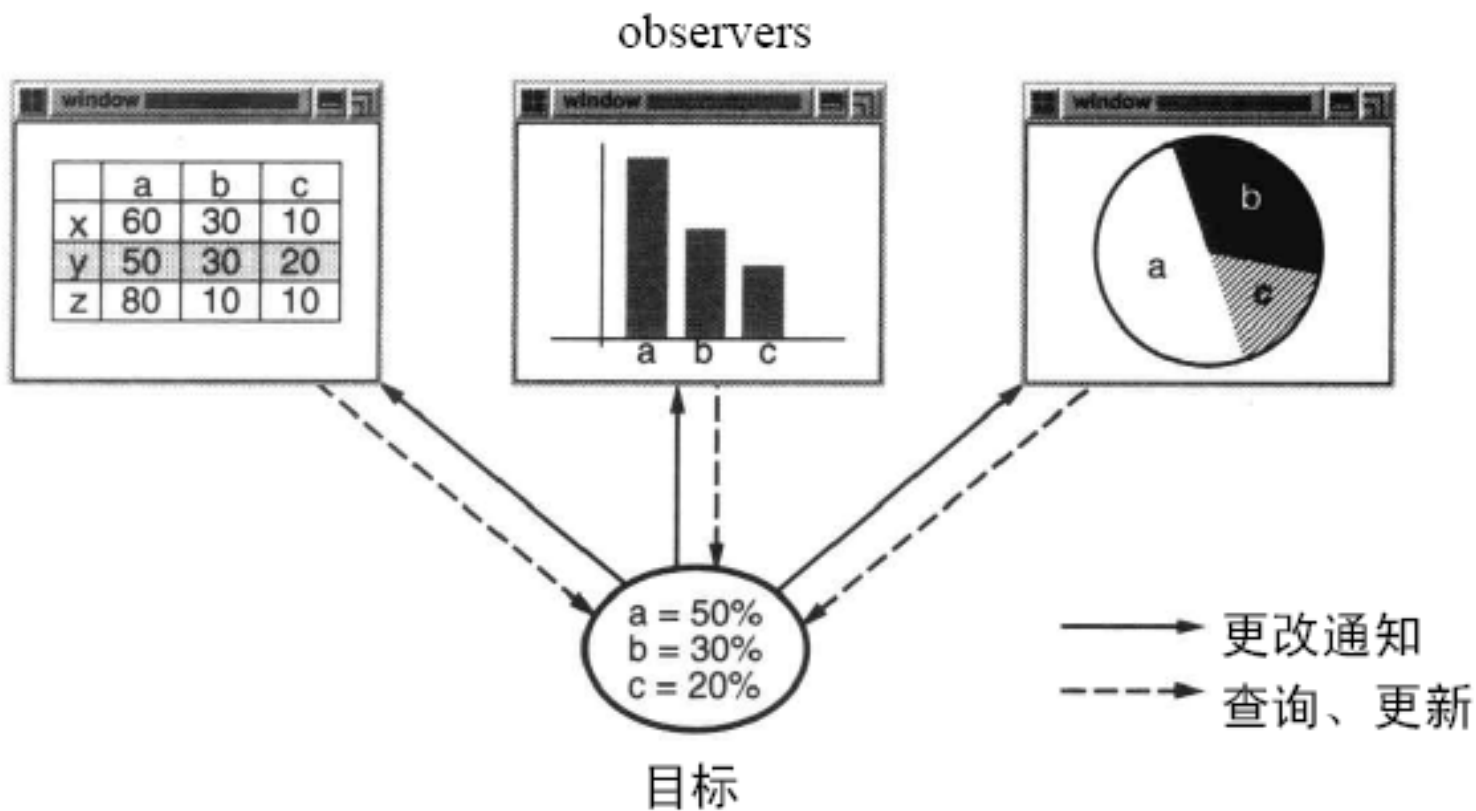
## 2. 别名

依赖 (Dependents)

发布-订阅(Publish-Subscribe)

### 3. 动机

- 维护相关对象间的一致性





## 3. 动机

---

### ■ Observer模式

- 目标(subject)与观察者(observer)
  - 一个目标有任意数目的观察者
  - 目标状态发生改变，所有观察者都得到通知
  - 对通知的响应，观察者查询目标状态，实现状态同步
- 发布-订阅 publish – subscribe
  - 目标是通知的发布者
  - 目标发出通知，并不知道谁是其观察者

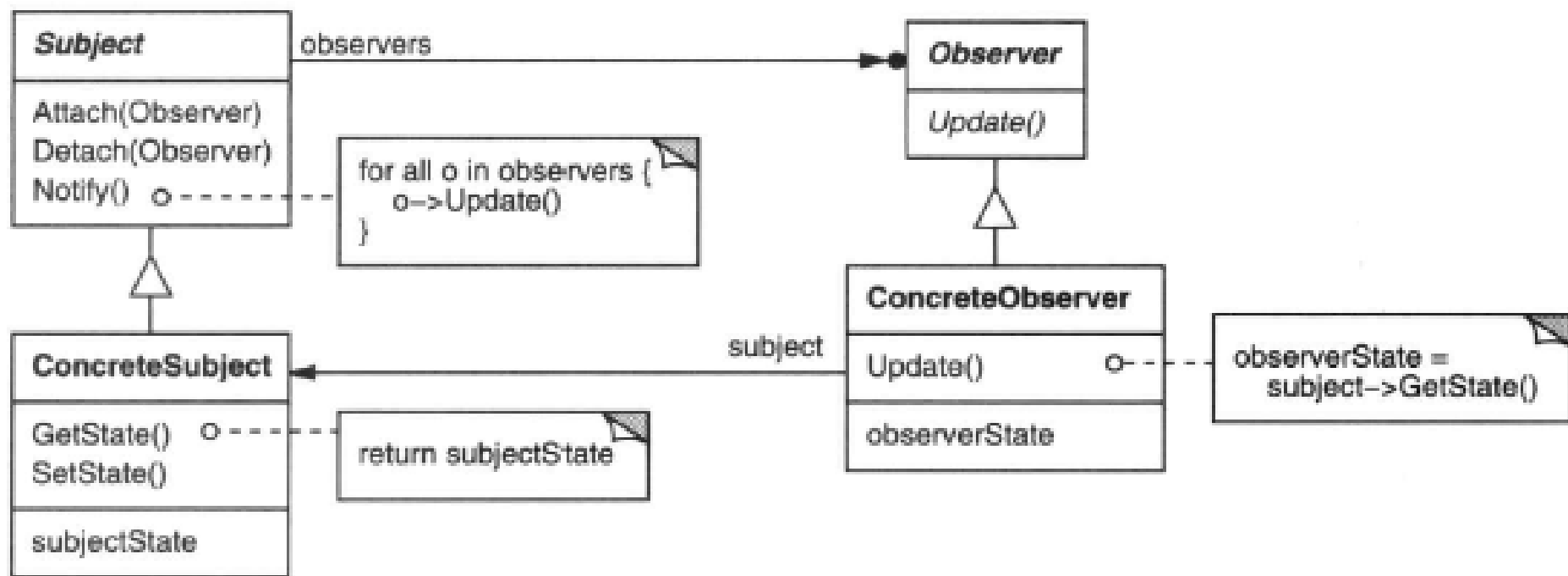


## 4.适用性

---

- 当一个抽象模型有两个方面，其中一个方面依赖于另一个方面
  - 将二者封装在独立对象中以使它们可以独立改变和复用
- 当对一个对象的改变需要改变其他对象，而不知道具体多少对象待改变
- 一个对象必须通知其他对象，而它不能假定其他对象是谁
  - 即：不希望这些对象是紧密耦合的

## 5. 结构





## 6. 参与者

---

- Subject
  - 目标知道它的观察者
  - 提供attach/detach接口
- Observer
  - 定义更新接口
- ConcreteSubject
  - 存储有关状态
  - 状态变化时，通知各个观察者
- ConcreteObserver
  - 指向ConcreteSubject的引用
  - Observer中存储的状态，与Subject的保持一致
  - 实现更新接口

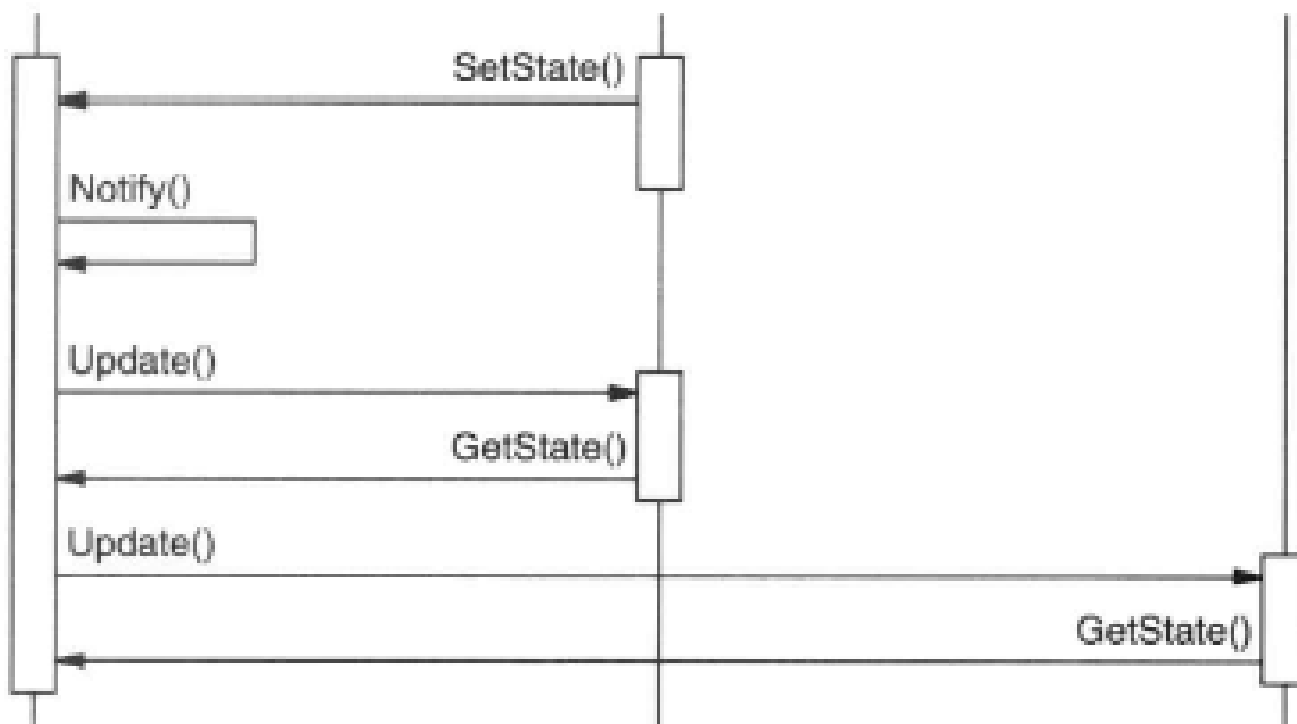
## 7. 协作

不是立即更新

aConcreteSubject

aConcreteObserver

anotherConcreteObserver



Notify也可被一个观察者或其他对象调用





## 8. 效果

---

### 1) 目标和观察者间的抽象耦合

- Subject仅知道一系列符合Observer接口的观察者
- Subject不知道观察者具体的类

### 2) 支持广播通信

- Subject发送通知时不需要指定其接收者

### 3) 意外的更新

- 一个观察者对目标的修改可能引起一系列对象的更新
- e.g. 重复更新等



## 9. 实现

---

### 1) 创建目标到其他观察者之间的映射

- 如何在Subject中保存对Observer的引用
  - 显式在Subject中保存对Observer的引用
  - 或
  - 用关联查找机制(Hash表)维护目标到观察者的映射

### 2) 观察多个目标

- 一个观察者依赖多个目标
- 扩展Update接口  
Update(Subject\* Sender)



## 9.实现

---

### 3) 谁触发更新

- Subject的Set操作触发

- 优点： Client不需要额外操作

- 问题：

- 多个连续的Set操作会导致多次更新，效率低

- Client负责触发

- 优点： 避免不必要的中间更新

- 缺点： 对Client中要求



## 9. 实现

---

### 4) 对已删除目标的悬挂引用

- Subject被删除时，通知观察者
  - 注意：不能直接删除观察者



## 9.实现

---

### 5)在发出通知前确保目标的状态自身是一致的

```
void MySubject::Operation (int newValue) {  
    BaseClassSubject::Operation(newValue);  
    // trigger notification  
  
    _myInstVar += newValue;  
    // update subclass state (too late!)  
}
```

错误情况

```
void Text::Cut (TextRange r) {  
    ReplaceRange(r);          // redefined in subclasses  
    Notify();  
}
```

正确情况



## 9. 实现

---

- 6) 避免特定于观察者的更新协议— 推/拉模型
- Subject调用Observer的Update操作时使用的参数
    - 推模型 push model
      - Subject发送关于改变的详细信息
      - 假定目标知道一些观察者需要的信息
      - 观察者相对难于复用
    - 拉模式 pull model
      - Subject仅进行通知，不发送任何信息
      - 强调的是Subject不知道它的Observer
      - 可能效率较差，Observer难于确定什么改变了



## 9. 实现

---

### 7) 显式指定感兴趣的变化

- 扩展注册接口，观察者可以注册对特定事件感兴趣

```
void Subject::Attach(Observer*, Aspect& interest);
```

```
void Observer::Update(Subject*, Aspect& interest);
```



## 9.实现

---

### 8)封装复杂的更新语义

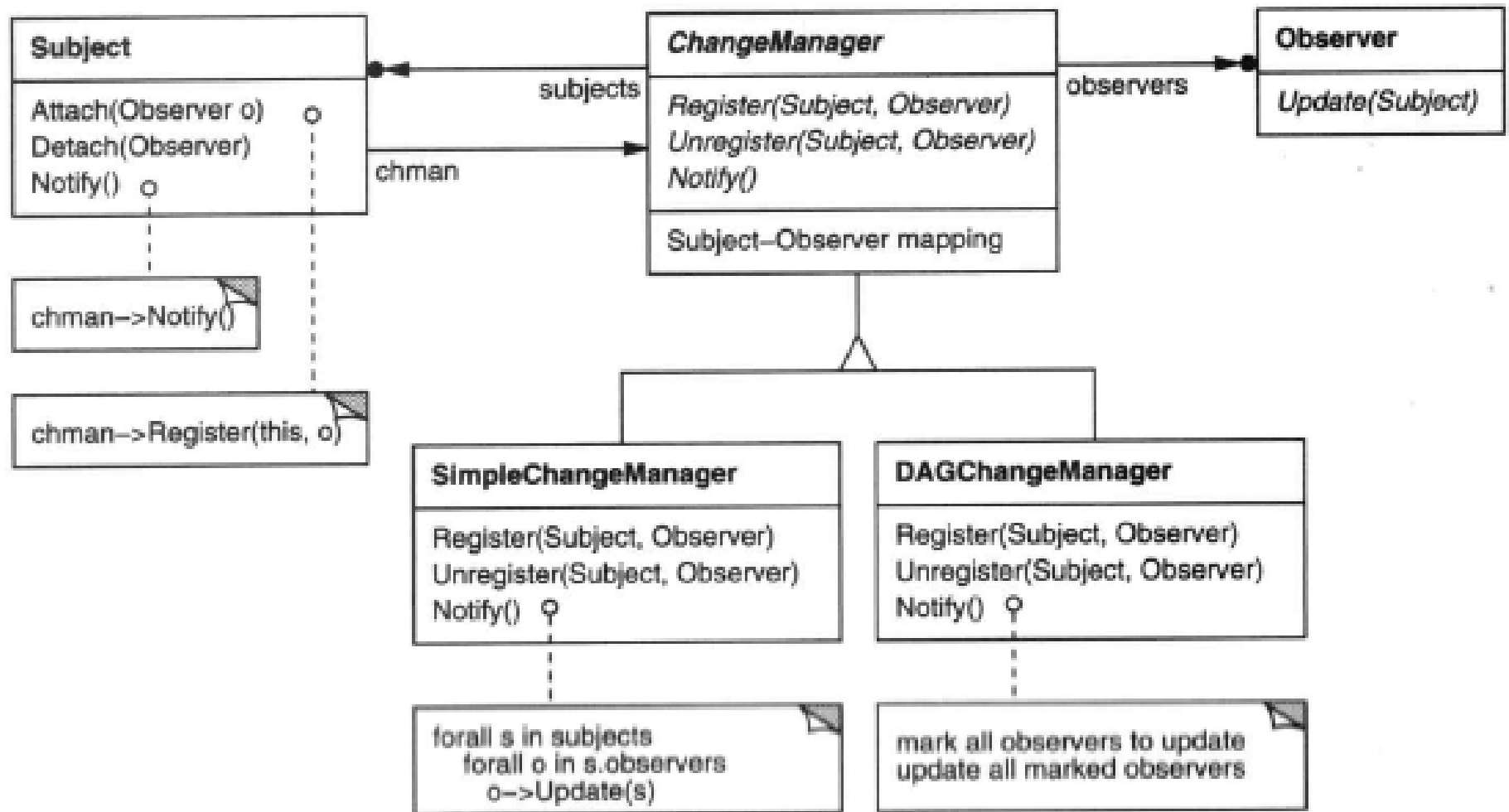
- 更新管理器(ChangeManger)

- 处理多个相互依赖的目标
- 所有目标均更新完毕后，才通知其Observer

- ChangeManager的责任

- 维护Subject—Observer的映射，Subject内不再维护
- 定义一个特定的更新策略
- 根据一个目标的请求，更新所有依赖于这个目标的Observer





ChangeManager是Mediator模式的实例  
(可以使用Singleton模式，如果只有一个ChangeManager的情况)



## 10.代码示例

---

```
class Subject;

class Observer {
public:
    virtual ~Observer();
    virtual void Update(Subject* theChangedSubject) = 0;
protected:
    Observer();
};
```



一个观察者可以有多个目标



## 10.代码示例

```
class Subject {  
public:  
    virtual ~Subject();  
  
    virtual void Attach(Observer*);  
    virtual void Detach(Observer*);  
    virtual void Notify();  
protected:  
    Subject();  
private:  
    List<Observer*> *_observers;  
};
```

```
void Subject::Attach (Observer* o) {  
    _observers->Append(o);  
}  
  
void Subject::Detach (Observer* o) {  
    _observers->Remove(o);  
}  
  
void Subject::Notify () {  
    ListIterator<Observer*> i(_observers);  
    for (i.First(); !i.IsDone(); i.Next()) {  
        i.CurrentItem()->Update(this);  
    }  
}
```

Subject类



## 10.代码示例

---

```
class ClockTimer : public Subject {  
public:  
    ClockTimer();
```

```
    virtual int GetHour();  
    virtual int GetMinute();  
    virtual int GetSecond();
```

```
    void Tick();
```

```
};
```

```
void ClockTimer::Tick () {  
    // update internal time-keeping state  
    // ...  
    Notify();  
}
```

具体Subject类

```
class DigitalClock: public Widget, public Observer {
public:
    DigitalClock(ClockTimer*);
    virtual ~DigitalClock();

    virtual void Update(Subject*);
        // overrides Observer operation

    virtual void Draw();
        // overrides Widget operation;
        // defines how to draw the digital clock
private:
    ClockTimer* _subject;
};
```

具体  
Observer

```
DigitalClock::DigitalClock (ClockTimer* s) {
    _subject = s;
    _subject->Attach(this);
}

DigitalClock::~~DigitalClock () {
    _subject->Detach(this);
}
```

Attach  
/  
Detach



## 10.代码示例

Update操作

```
void DigitalClock::Update (Subject* theChangedSubject) {  
    if (theChangedSubject == _subject) {  
        Draw();  
    }  
}
```

```
void DigitalClock::Draw () {  
    // get the new values from the subject  
  
    int hour = _subject->GetHour();  
    int minute = _subject->GetMinute();  
    // etc.  
  
    // draw the digital clock  
}
```



## 10.代码示例

---

```
class AnalogClock : public Widget, public Observer {
public:
    AnalogClock(ClockTimer*);
    virtual void Update(Subject*);
    virtual void Draw();
    // ...
};
```

另外一个Observer



## 10.代码示例

---

```
ClockTimer* timer = new ClockTimer;  
AnalogClock* analogClock = new AnalogClock(timer);  
DigitalClock* digitalClock = new DigitalClock(timer);
```

Client

两个Observer总是显示相同的时间



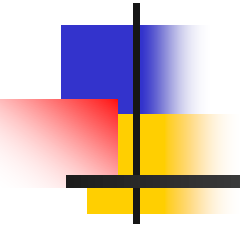


## 12. 相关模式

---

- Mediator
  - 用于ChangeManager
- Singleton
  - 如果只有一个ChangeManager时

## 3.5 State 状态





# State模式

---

## 1. 意图

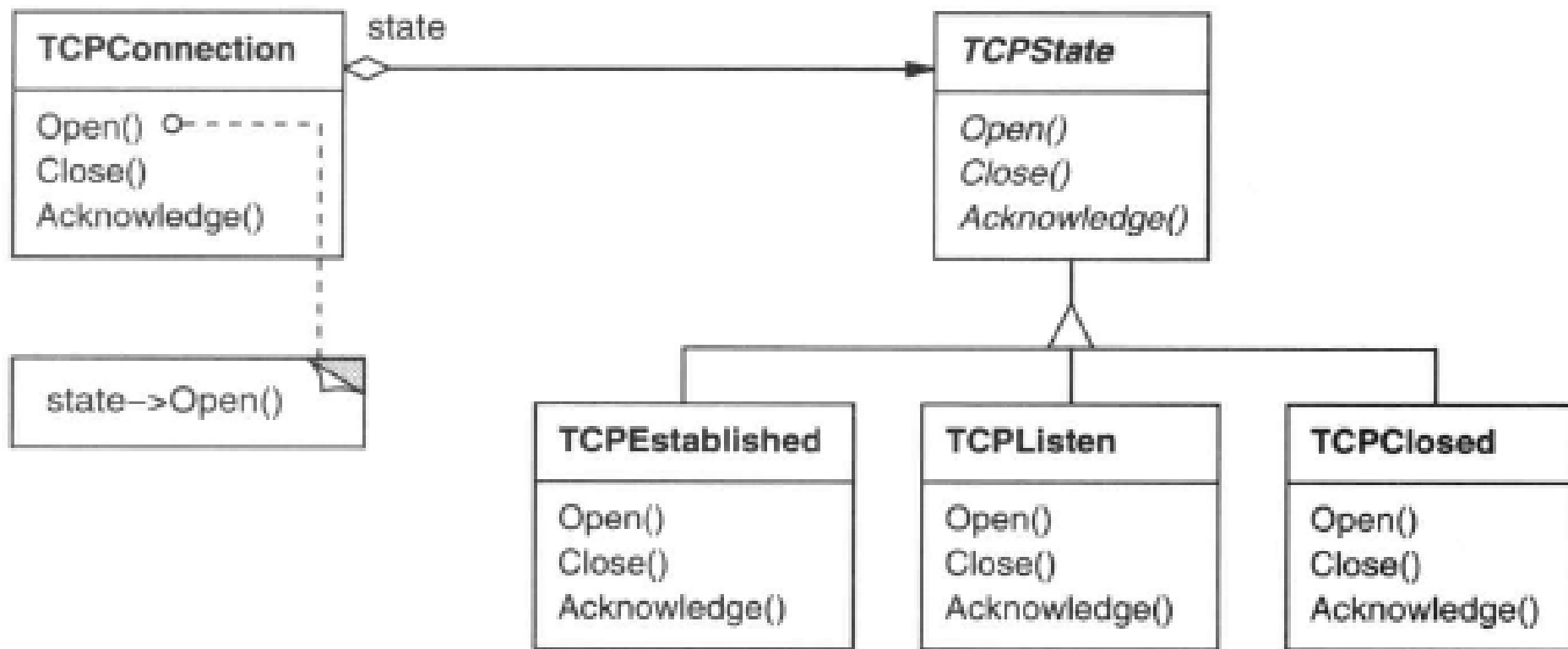
- 允许一个对象在其内部状态改变时改变它的行为
- 对象看起来似乎修改了它的类

## 2. 别名

状态对象(Objects for States)

### 3. 动机

- TCPConnect在不同状态时有不同的反应



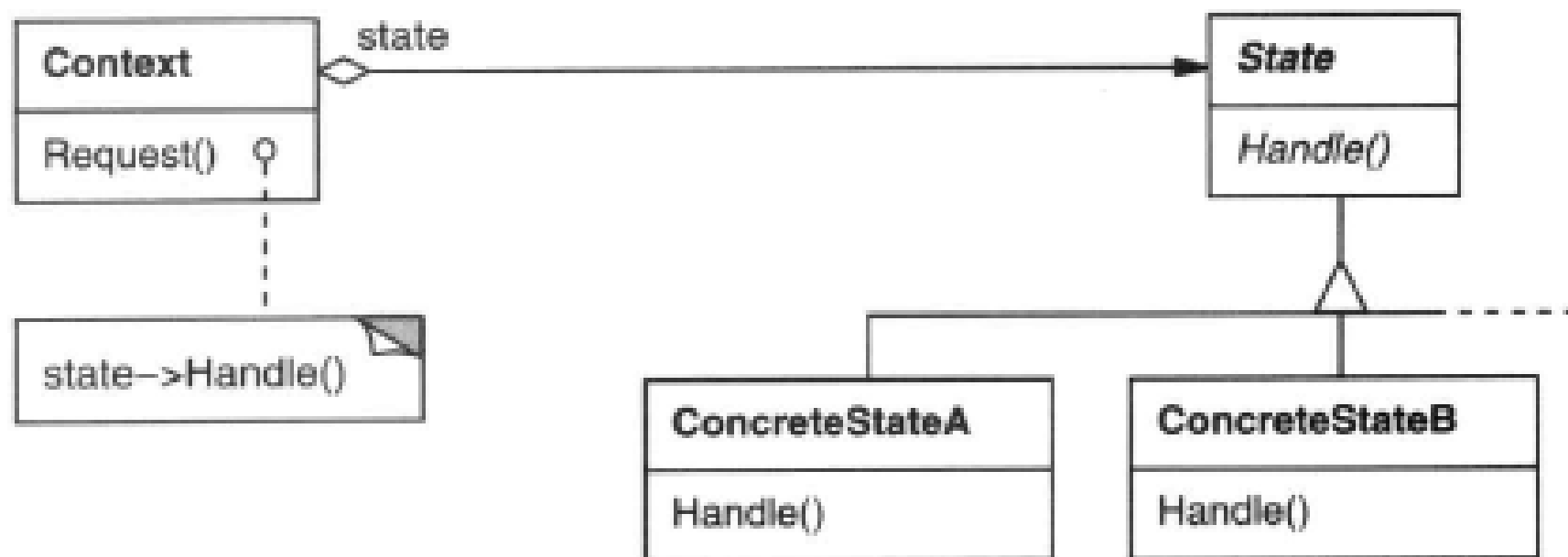


## 4.适用性

---

- 一个对象的行为取决于它的状态，并且它必须在运行时刻根据状态改变它的行为
- 一个操作中含有庞大的多分支的条件语句，且这些分支依赖于该对象的状态

## 5. 结构





## 6. 参与者

---

- Context（环境, TCPConnection）
  - 定义Client感兴趣的接口
  - 维护一个当前状态的实例
- State(状态, TCPState)
  - 定义接口
- ConcreteState(具体状态子类)
  - 实现与Context某个状态相关的行为



## 7.协作

---

- Context将与状态有关的请求委托给当前的ConcreteState对象处理
- Context将自身作为参数传递给状态对象
  - 使得状态对象可以访问Context
- Client不直接与状态对象交互
- Context或ConcreteState可以决定后续状态是什么， 以及进行状态转换





## 8.效果

---

- 1) 将与特定状态相关的行为局部化，并且将不同状态的行为分割开来
- 2) 使得状态转换显式化
- 3) State对象可被共享



## 9. 实现

---

### 1) 谁定义状态转换

- Context

- 如果转换规则是固定的

- State对象

- 需要Context提供接口，State对象可以设置Context的当前状态
  - 优点：容易定义新的State子类来扩展和修改逻辑
  - 缺点：产生了State子类间的关联性



## 9.实现

---

### 2)基于表的另一种方法

- 用表来记录状态转换规则
  - 当前状态 + 输入  $\rightarrow$  后继状态
  - 优点：规则性，可通过更改数据(表)来改变状态转换准则
  - 缺点：
    - 对表的查找效率不如虚函数调用
    - 表格形式进行表示，使得转换规则不明确，难以理解
    - 难以加入伴随状态转换的一些动作
- 表驱动方式：着重定义状态转换
- State模式：对与状态相关的行为进行建模



## 9.实现

---

### 3)创建和销毁State对象

- 方式1：仅当需要State对象时才创建并在随后销毁
  - 适用情况
    - 将要进入的状态在运行时是不可知的，并且Context不经常改变状态
    - State对象存储大量信息时，可避免创建不会用到的对象
- 方式2：提前创建并且始终不销毁State对象
  - 适用：状态改变很频繁时
  - 缺点：可能不方便，Context需要保存所有State对象的指针

# 10. 代码示例

Context

```
class TCPOctetStream;
class TCPState;
class TCPConnection {
public:
    TCPConnection();

    void ActiveOpen();
    void PassiveOpen();
    void Close();
    void Send();
    void Acknowledge();
    void Synchronize();

    void ProcessOctet(TCPOctetStream*);
private:
    friend class TCPState;
    void ChangeState(TCPState*);
private:
    TCPState* _state;
};
```

定义了一个转换状态的接口



## 10. 代码示例

---

State父类

```
class TCPState {
public:
    virtual void Transmit(TCPConnection*, TCPOctetStream*);
    virtual void ActiveOpen(TCPConnection*);
    virtual void PassiveOpen(TCPConnection*);
    virtual void Close(TCPConnection*);
    virtual void Synchronize(TCPConnection*);
    virtual void Acknowledge(TCPConnection*);
    virtual void Send(TCPConnection*);
protected:
    void ChangeState(TCPConnection*, TCPState*);
};
```

---

```
TCPConnection::TCPConnection () {  
    _state = TCPClosed::Instance();  
}
```

```
void TCPConnection::ChangeState (TCPState* s) {  
    _state = s;  
}
```

```
void TCPConnection::ActiveOpen () {  
    _state->ActiveOpen(this);  
}
```

```
void TCPConnection::PassiveOpen () {  
    _state->PassiveOpen(this);  
}
```

```
void TCPConnection::Close () {  
    _state->Close(this);  
}
```

```
void TCPConnection::Acknowledge () {  
    _state->Acknowledge(this);  
}
```

```
void TCPConnection::Synchronize () {  
    _state->Synchronize(this);  
}
```

TCPConnect将与状态有关的操作  
委托给\_state对象



## 10. 代码示例

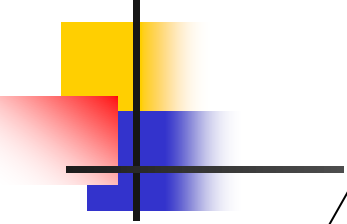
---

State父类定义的缺省实现

```
void TCPState::Transmit (TCPConnection*, TCPOctetStream*) { }
void TCPState::ActiveOpen (TCPConnection*) { }
void TCPState::PassiveOpen (TCPConnection*) { }
void TCPState::Close (TCPConnection*) { }
void TCPState::Synchronize (TCPConnection*) { }

void TCPState::ChangeState (TCPConnection* t, TCPState* s) {
    t->ChangeState(s);
}
```





```
class TCPEstablished : public TCPState {
public:
    static TCPState* Instance();

    virtual void Transmit(TCPConnection*, TCPOctetStream*);
    virtual void Close(TCPConnection*);
};
```

```
class TCPListen : public TCPState {
public:
    static TCPState* Instance();

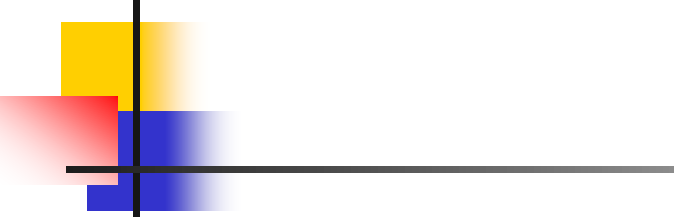
    virtual void Send(TCPConnection*);
    // ...
};
```

```
class TCPClosed : public TCPState {
public:
    static TCPState* Instance();

    virtual void ActiveOpen(TCPConnection*);
    virtual void PassiveOpen(TCPConnection*);
    // ...
};
```

Singleton

TCPState子类



```
void TCPClosed::ActiveOpen (TCPConnection* t) {
    // send SYN, receive SYN, ACK, etc.

    ChangeState(t, TCPEstablished::Instance());
}

void TCPClosed::PassiveOpen (TCPConnection* t) {
    ChangeState(t, TCPListen::Instance());
}

void TCPEstablished::Close (TCPConnection* t) {
    // send FIN, receive ACK of FIN

    ChangeState(t, TCPListen::Instance());
}

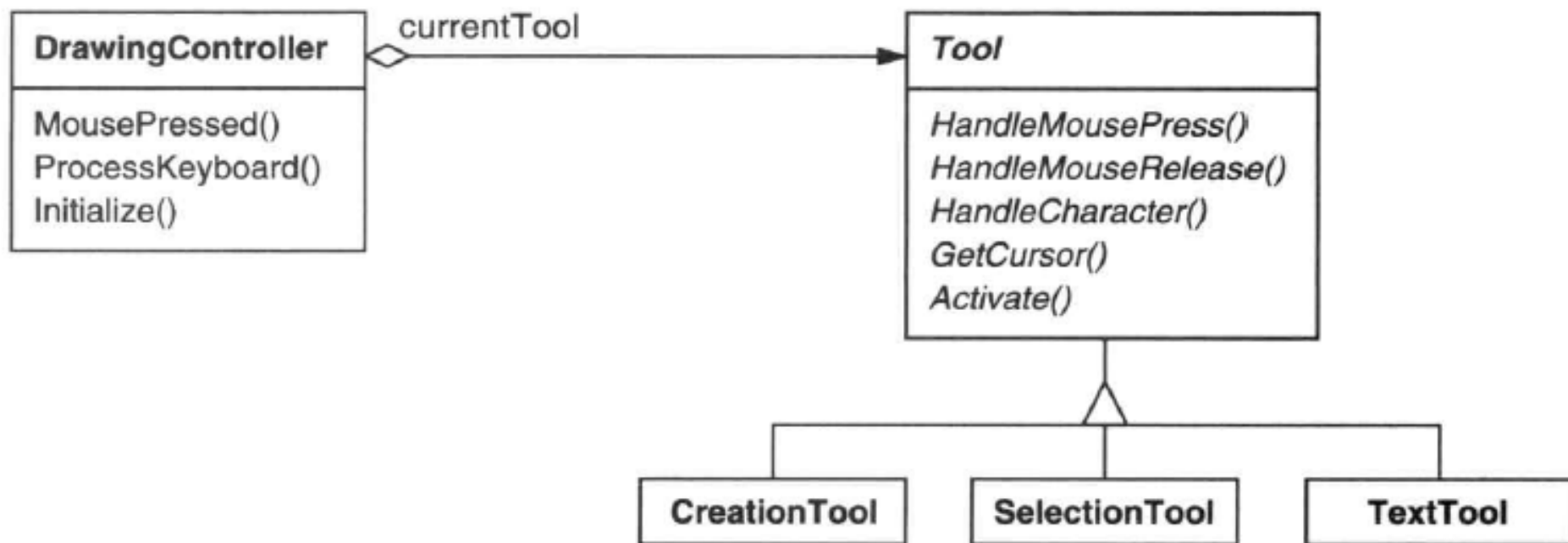
void TCPEstablished::Transmit (
    TCPConnection* t, TCPOctetStream* o
) {
    t->ProcessOctet(o);
}

void TCPListen::Send (TCPConnection* t) {
    // send SYN, receive SYN, ACK, etc.

    ChangeState(t, TCPEstablished::Instance());
}
```

子类中的具体操作

# 11. 已知应用





## 12. 相关模式

---

- Flyweight模式： 共享状态对象
- Singleton： 状态对象通常是Singleton



## 5.9 Strategy 策略模式

---



# Strategy模式

---

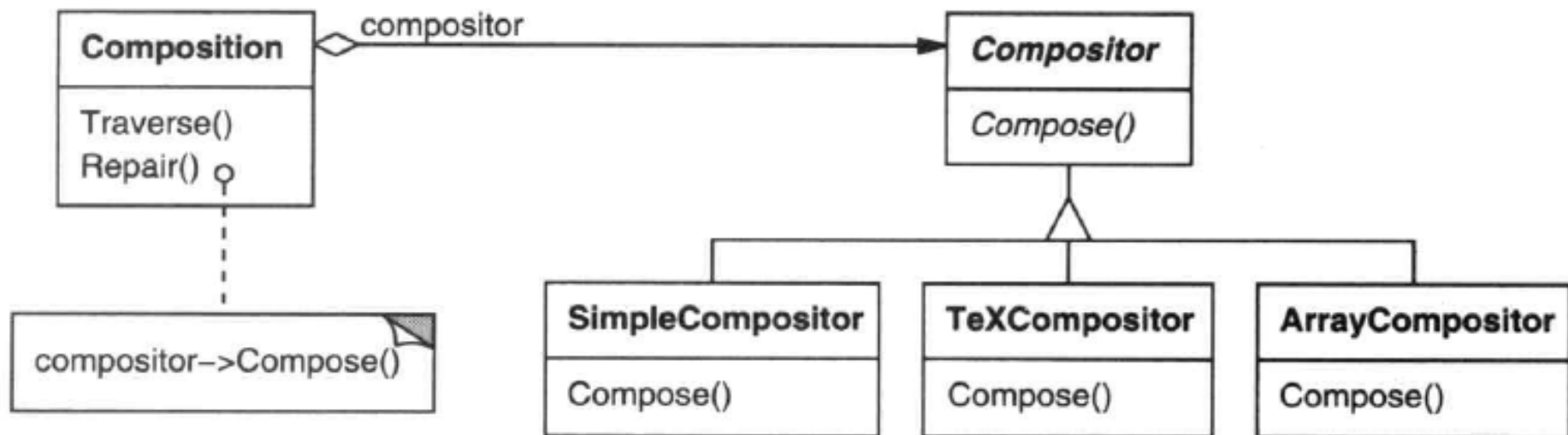
## 1. 意图

- 定义一系列的算法，把它们一个个封装起来，使得它们可以相互替换
- 本模式使得算法可独立于客户而变化

## 2. 别名

政策 Policy

### 3. 动机



多个对正文流分行的算法



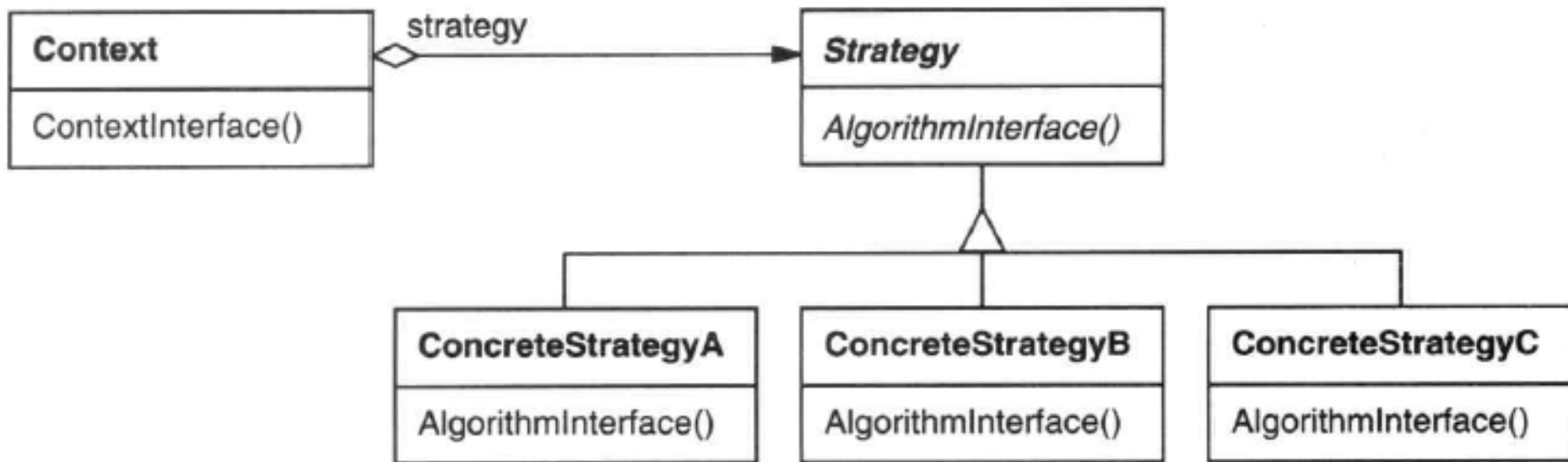
## 4.适用性

---

- 许多相关的类仅仅是行为有异
  - Strategy模式：一个类 + 行为类
- 需要适用一个算法的不同变体
- 算法使用客户不应该知道的数据
  - Strategy模式可以对算法相关的东西进行封装
- 一个类定义了多种行为，以多个条件语句的形式实现
  - Strategy将相应的条件分支移入各自对应的Strategy类



## 5. 结构





## 6. 参与者

---

- Strategy(Compositor)
  - 定义所支持算法的公共接口
- ConcreteStrategy
  - 实现具体算法，符合Strategy定义的接口
- Context(Composition)
  - 用一个ConcreteStrategy对象来配置
  - 维护一个Strategy对象的引用
  - 可定义一个接口让Strategy访问Context的数据



## 7.协作

---

- Strategy和Context相互作用以实现特定的算法

Context :

`_strategy->AlogrithmInterface(data...)`

或

`_strategy->AlogrithmInterface(this)`



## 7.协作

---

- Context将Client的请求转发给Strategy
  - 1) Client : 用具体Strategy对象配置context  
new context(new concretestrategyA)
  - 2) Client : 仅与context交互  
context->ContextInterface()



## 8.效果

---

### 1) 相关算法系列

- Strategy的类层次结构支持一系列的算法
- 继承有利于析取出这些算法中的公共功能

### 2) 一个替换继承的方法

- 继承方式：
  - Context父类 + ConcreteContext子类的方式
  - 算法实现与Context实现混合，复杂化了




## 8.效果

---

### 3)消除了一些条件语句

```
void Composition::Repair () {  
    switch (_breakingStrategy) {  
        case SimpleStrategy:  
            ComposeWithSimpleCompositor();  
            break;  
        case TeXStrategy:  
            ComposeWithTeXCompositor();  
            break;  
        // ...  
    }  
    // merge results with existing composition, if necessary  
}
```

含有许多条件语句的代码通常意味着需要使用Strategy模式



```
void Composition::Repair () {  
    _compositor->Compose();  
    // merge results with existing composition, if necessary  
}
```



## 8.效果

---

### 4)实现的选择

- Strategy模式提供相同行为的不同实现
- Client根据需从不同的策略中选择

### 5) Client必须了解不同的Strategy

- Client进行选择时必须了解各个Strategy的不同



## 8. 效果

---

### 6) Strategy和Context之间的通信开销

Context调用

```
_strategy->AlogrithmInterface(data...)
```

可能对于简单的strategy，不用到传入的参数

### 7)增加了对象的数目

- 原本单个的对象变成(Context + Strategy)
- 使用Flyweight模式共享Strategy对象可以减少此开销





## 9.实现

---

### 1) 定义Strategy和Context接口

- 目的：Strategy能够访问到需要的数据(存在于Context中的)
- 方法1：Context调用时传入数据
  - 优点：Context与Strategy解耦
  - 缺点：有些数据可能是Strategy不要的，额外开销
- 方法2：
  - Context将自己指针传入，然后Strategy对象去读取数据
  - Strategy对象本身存储对Context的引用
  - 优点：Strategy可以只获取自己需要的数据
  - 缺点：
    - Context必须定义接口，方便Strategy对象获取数据
    - Strategy与Context耦合



## 9.实现

---

### 2) 将Strategy作为模板参数

```
template <class AStrategy>
class Context {
    void Operation() { theStrategy.DoAlgorithm(); }
    // ...
private:
    AStrategy theStrategy;
};
```

```
class MyStrategy {
public:
    void DoAlgorithm();
};
```

```
Context<MyStrategy> aContext;
```

1.可以在编译时选择Strategy

2.不需在运行时改变



## 9. 实现

---

### 3) 使Strategy对象成为可选的

- 即：不配置Strategy时，Context可以执行缺省的操作



## 10.代码示例

---

```
class Composition {  
public:  
    Composition(Compositor*);  
    void Repair();  
private:  
    Compositor* _compositor;  
    Component* _components;  
    int _componentCount;  
    int _lineWidth;  
    int* _lineBreaks;  
  
    int _lineCount;  
};
```

Context

```
// the list of components  
// the number of components  
// the Composition's line width  
// the position of linebreaks  
// in components  
// the number of lines
```



## 10.代码示例

---

Strategy父类

```
class Compositor {  
public:  
    virtual int Compose(  
        Coord natural[], Coord stretch[], Coord shrink[],  
        int componentCount, int lineWidth, int breaks[]  
    ) = 0;  
protected:  
    Compositor();  
};
```



将所有数据传入



## 10.代码示例

---

```
void Composition::Repair () {
    Coord* natural;
    Coord* stretchability;
    Coord* shrinkability;
    int componentCount;
    int* breaks;

    // prepare the arrays with the desired component sizes
    // ...

    // determine where the breaks are:
    int breakCount;
    breakCount = _compositor->Compose(
        natural, stretchability, shrinkability,
        componentCount, _lineWidth, breaks
    );

    // lay out components according to breaks
    // ...
}
```

Context调用Strategy的操作



## 10.代码示例

---

- SimpleCompositor
  - 一次检查一行Component，并决定在哪里换行
- TextCompositor
  - 一次检查一个段落
- ArrayCompositor
  - 用规则的间距将构件分割成行

都没有使用所有Context传入的信息



## 10.代码示例

---

- Client实例化Context

```
Composition* quick = new Composition(new SimpleCompositor);  
Composition* slick = new Composition(new TeXCompositor);  
Composition* iconic = new Composition(new ArrayCompositor(100));
```





## 12. 相关模式

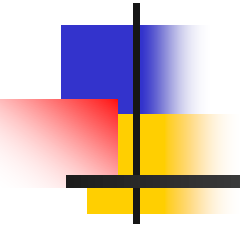
---

- Flyweight模式
  - 共享Strategy对象

## 5.10 Template Method

### 模板方法

---





# 1. 意图

---

- 定义一个操作中算法的骨架，而将一些步骤延迟到子类中
- TemplateMethod使得子类不改变算法的结构即可重定义算法中某些特定步骤



## 2. 动机

---

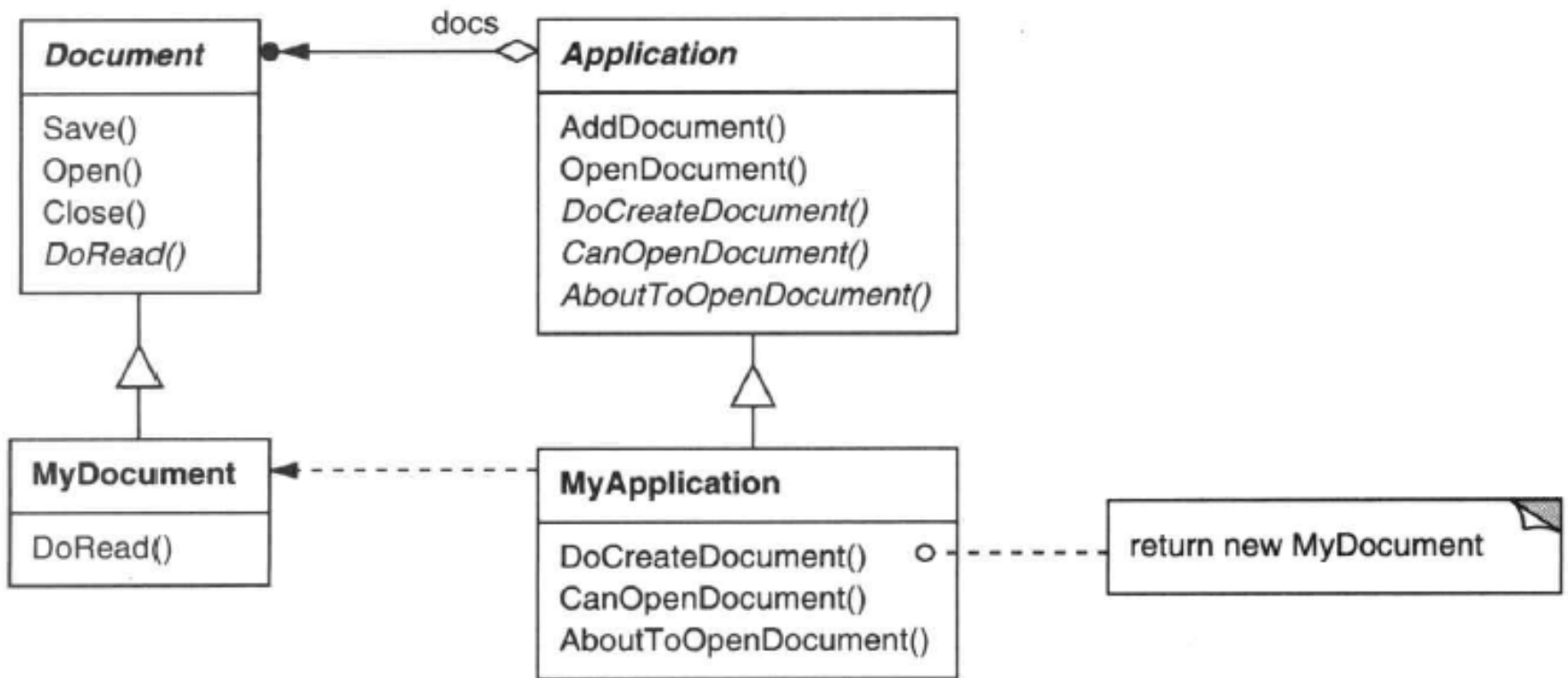
### ■ Application – Document类的应用框架

```
void Application::OpenDocument (const char* name) {  
    if (!CanOpenDocument(name)) {  
        // cannot handle this document  
        return;  
    }  
  
    Document* doc = DoCreateDocument();  
  
    if (doc) {  
        _docs->AddDocument (doc);  
        AboutToOpenDocument (doc);  
        doc->Open();  
        doc->DoRead();  
    }  
}
```



模板方法

## 2. 动机



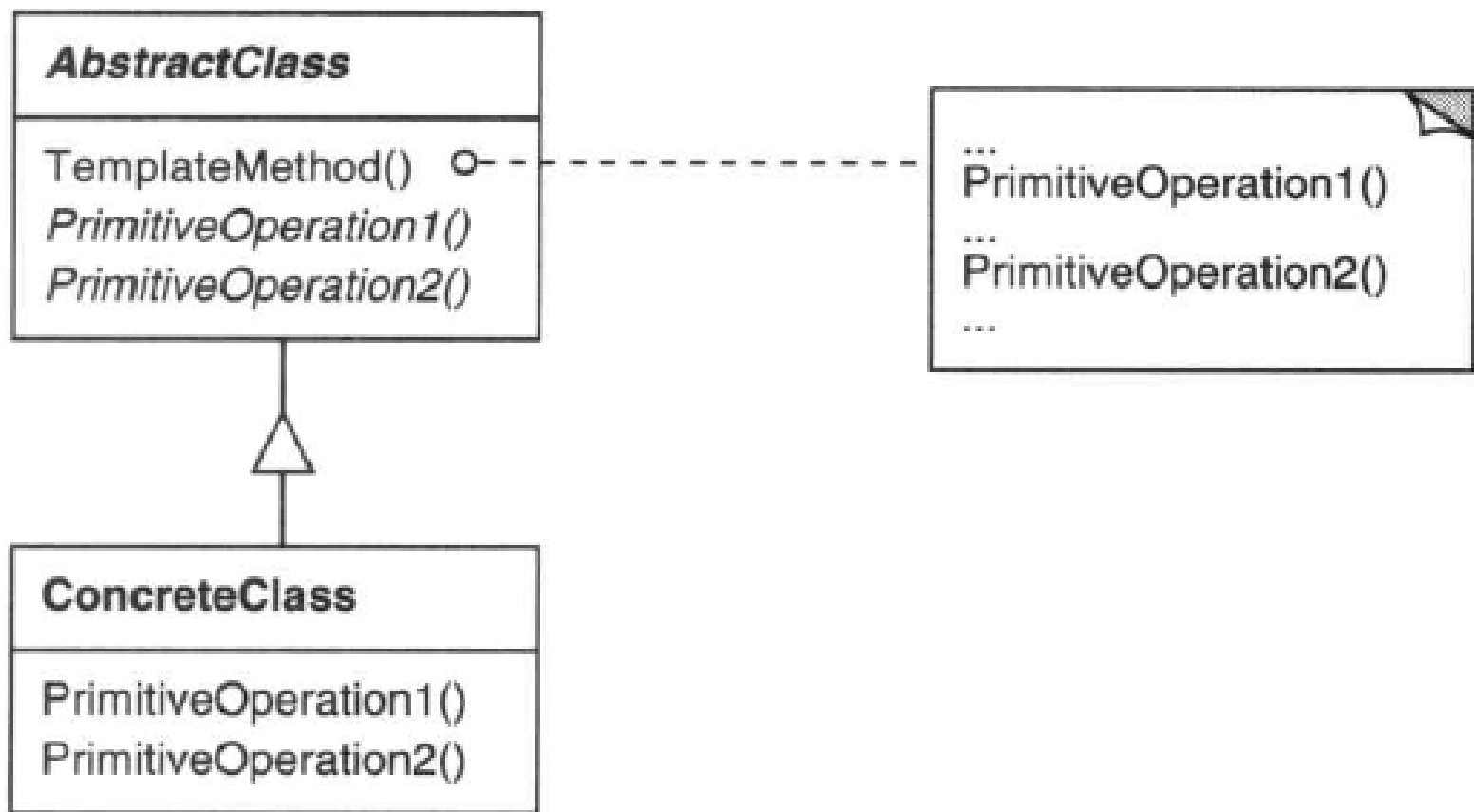


## 3.适用性

---

- 一次性实现一个算法中不变的部分，将可变的行为留给子类来实现
- 各子类中共同的行为应被提取出来并集中到一个公共父类中以避免代码重复
- 控制子类扩展
  - 模板方法只能在特定点进行扩展

## 4. 结构





## 5. 参与者

---

- AbstractClass

- 定义操作的原语操作(primitive operation)
  - 由子类进行重定义
- 实现一个模板方法，定义算法的骨架
  - 该方法调用原语操作、以及其他操作

- ConcreteClass

- 实现原语操作





## 6.协作

---

- ConcreteClass靠AbstractClass来实现算法中不变的步骤



## 7.效果

---

- 模板方法
  - 一种代码复用的基本计数
  - 反向的控制结构（即父类去调用子类的操作）
- 模板方法调用下列类型的操作
  - 具体的操作(ConcreteClass或对Client类的操作)
  - 具体的AbstractClass的操作(即通常对子类有用的操作)
  - 原语操作(即：抽象操作) : 必须被重定义
  - Factory Method
  - 钩子操作(hook operation) : 可以被重定义

## 7.效果

```
void DerivedClass::Operation () {  
    ParentClass::Operation();  
    // DerivedClass extended behavior  
}
```

转换为模板方法

```
void ParentClass::Operation () {  
    // ParentClass behavior  
    HookOperation();  
}
```

调用钩子函数

```
void ParentClass::HookOperation () { }
```

父类的钩子函数

```
void DerivedClass::HookOperation () {  
    // derived class extension  
}
```

子类的钩子函数



## 8. 实现

---

### 1) 使用C++访问控制

- 原语操作: `private`成员
- 原语操作定义为虚函数, 模板方法本身非虚函数

### 2) 尽量减少原语操作

### 3) 命名约定: 通常需要在需要重定义的操作加前缀 e.g. `DoCreateDocument`, `DoRead...`



## 9.代码示例

---

```
void View::Display () {  
    SetFocus();  
    DoDisplay();  
    ResetFocus();  
}
```

进行绘制前必须设置焦点SetFocus

```
void View::DoDisplay () { }
```

父类的抽象操作

```
void MyView::DoDisplay () {  
    // render the view's contents  
}
```

子类进行重定义



# 11. 相关模式

---

- Factory Method

- 被模板方法调用

- Strategy

- 模板方法使用继承来改变算法的一部分
  - Strategy使用委托来改变整个算法



## 5.11 Visitor访问者模式

---



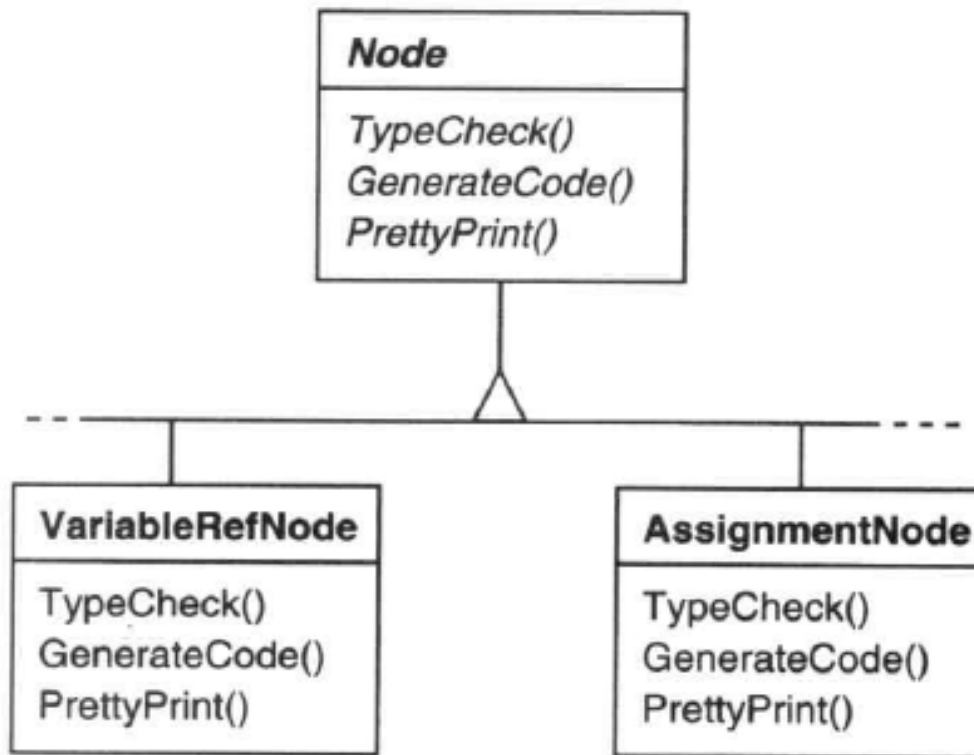
# 1. 意图

---

- 定义一个作用于某对象结构中各元素的操作
- 可以在不改变各元素的类的前提下定义新的操作(作用于这些元素)



## 2. 动机



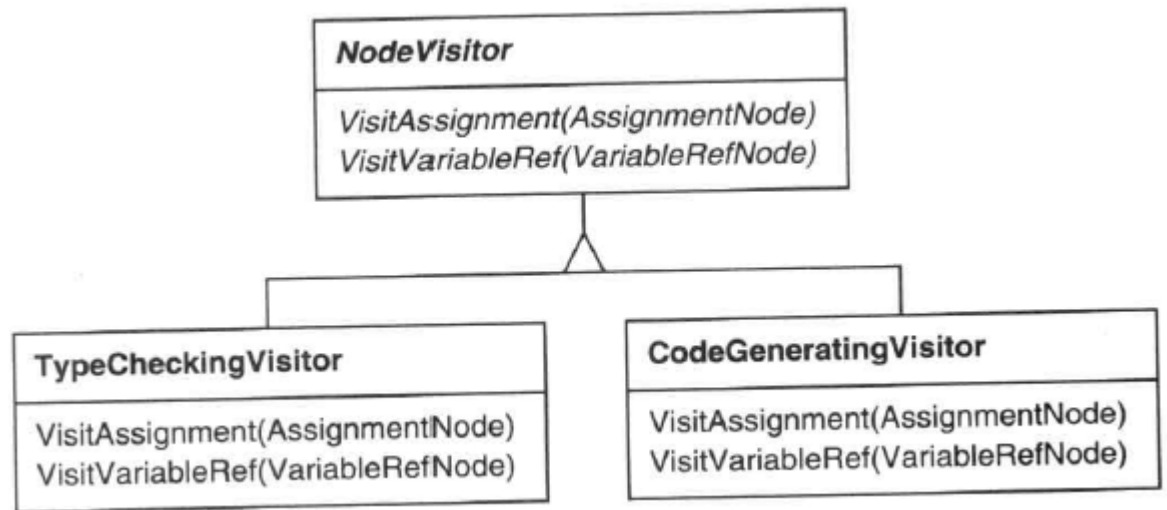
- 背景:

编译器，需要在抽象语法树上实施某些操作

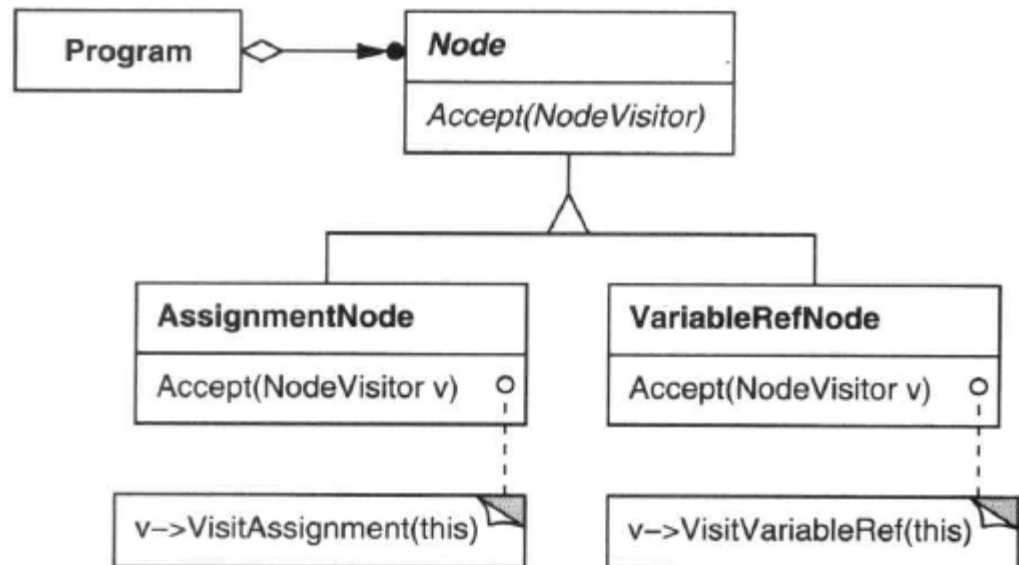
将操作分散到各个节点类中会导致系统难以理解、维护和修改

## 2. 动机

NodeVisitor层次



Node层次



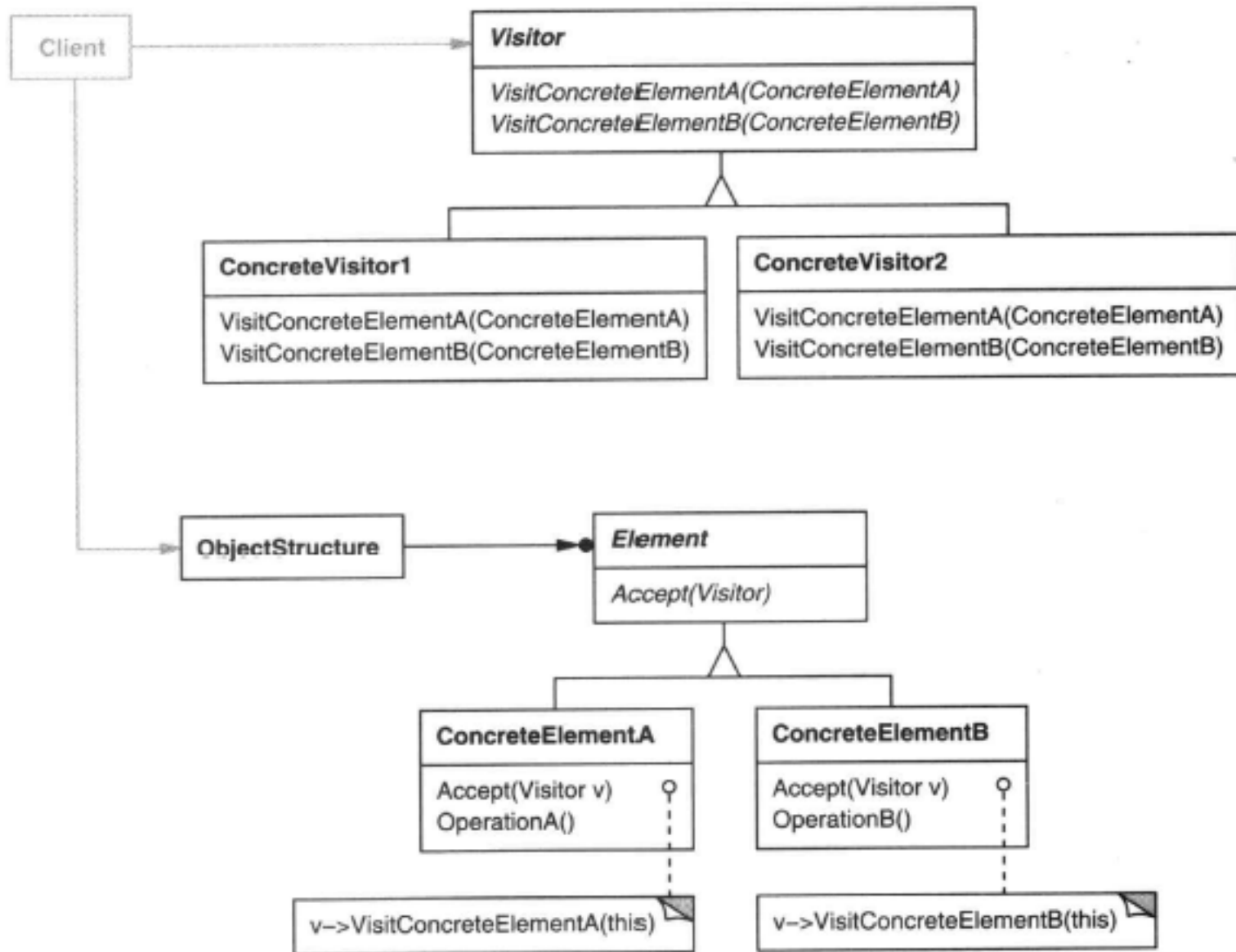


## 3.适用性

---

- 一个对象结构包含很多类对象，它们有不同的接口。需要对这些对象实施一些**依赖于具体类**的操作
- 需要对一个对象结构中的对象进行很多不同且不相关的操作，而不想让这些操作“污染”这些对象的类
- 定义对象结构的类很少改变，但经常需要在此结构上定义新的操作

## 4. 结构



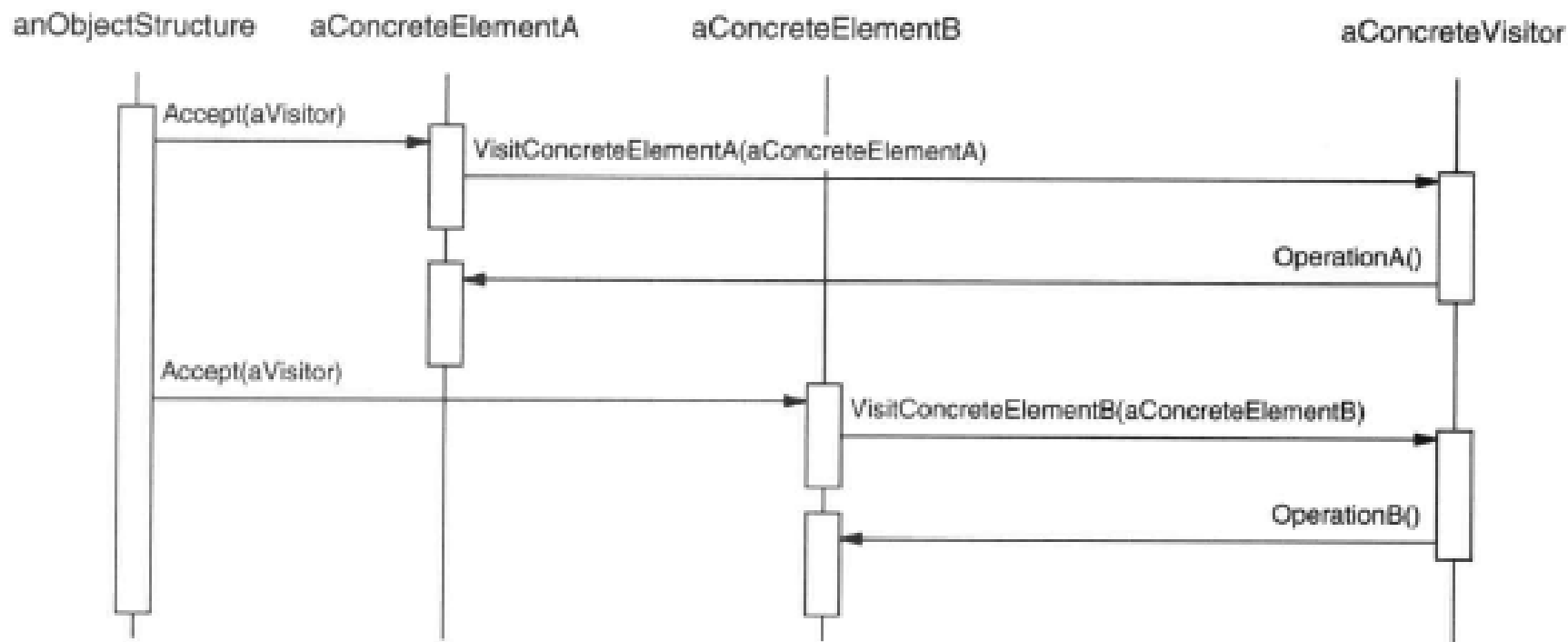


## 5. 参与者

---

- Visitor (NodeVisitor)
  - 为对象结构中的每个类声明一个Visit操作
- ConcreteVisitor
  - 实现父类中定义的操作
- Element
  - 定义一个Accept操作
- ConcreteElement
  - 实现Accept操作
- ObjectStructure
  - 能枚举它的元素
  - 可以提供一个高层结构以允许访问者访问它的元素
  - 可以是一个Composite或集合(e.g. 列表或无序集合)

## 6. 协作





## 7.效果

---

### 1) 访问者模式使得易于增加新的操作

- 仅需要增加一个新的Visitor

### 2) 访问者集中相关的操作而分离无关的操作

- 相关的操作→ Visitor (不是分布在各数据对象类上)
- 无关的操作→ 访问者子类



## 7.效果

---

### 3) 增加新的ConcreteElement类很困难

- Visitor模式适用于Element类层次稳定的情况

### 4) 通过类层次进行访问

```
template <class Item>
class Iterator {
    // ...
    Item CurrentItem() const;
};
```

Iterator模式访问的元素有共同父类

```
class Visitor {
public:
    // ...
    void VisitMyType(MyType*);
    void VisitYourType(YourType*);
};
```

Visitor模式访问的元素可以不具备共同父类





## 7. 效果

---

### 5) 累计状态

- 访问者访问各个元素时，可以累计状态(e.g. 计数)
- 如果不使用Visitor模式...

### 6) 破坏封装

- Visitor模式往往迫使Element提供访问内部状态的公共操作，破坏了Element的封装性



## 8. 实现

---

```
class Visitor {  
public:  
    virtual void VisitElementA(ElementA*);  
    virtual void VisitElementB(ElementB*);  
  
    // and so on for other concrete elements  
protected:  
    Visitor();  
};
```

Visitor需要针对各个ConcreteElement定义  
Visit操作



## 8.实现

---

Element需要定义Accept操作

```
class Element {
public:
    virtual ~Element();
    virtual void Accept(Visitor&) = 0;
protected:
    Element();
};

class ElementA : public Element {
public:
    ElementA();
    virtual void Accept(Visitor& v) { v.VisitElementA(this); }
};

class ElementB : public Element {
public:
    ElementB();
    virtual void Accept(Visitor& v) { v.VisitElementB(this); }
};
```



## 8.实现

---

```
class CompositeElement : public Element {
public:
    virtual void Accept(Visitor&);
private:
    List<Element*>* _children;
};

void CompositeElement::Accept (Visitor& v) {
    ListIterator<Element*> i(_children);

    for (i.First(); !i.IsDone(); i.Next()) {
        i.CurrentItem()->Accept(v);
    }
    v.VisitCompositeElement(this);
}
```

CompositeElement类可能的实现方式



## 8. 实现

---

- 谁负责遍历对象结构
  - Element对象结构
  - 使用Iterator(内部或外部)
  - 访问者中
    - 可以实现特别复杂的遍历



## 9.代码示例

---

```
class Equipment {  
public:  
    virtual ~Equipment();  
  
    const char* Name() { return _name; }  
  
    virtual Watt Power();  
    virtual Currency NetPrice();  
    virtual Currency DiscountPrice();  
  
    virtual void Accept(EquipmentVisitor&);  
protected:  
    Equipment(const char*);  
private:  
    const char* _name;  
};
```

Element



## 9.代码示例

---

```
class EquipmentVisitor {
public:
    virtual ~EquipmentVisitor();

    virtual void VisitFloppyDisk(FloppyDisk*);
    virtual void VisitCard(Card*);
    virtual void VisitChassis(Chassis*);
    virtual void VisitBus(Bus*);

    // and so on for other concrete subclasses of Equipment
protected:
    EquipmentVisitor();
};
```

Visitor父类



## 9.代码示例

ConcreteElement

```
void FloppyDisk::Accept (EquipmentVisitor& visitor) {  
    visitor.VisitFloppyDisk(this);  
}
```

```
void Chassis::Accept (EquipmentVisitor& visitor) {  
    for (  
        ListIterator<Equipment*> i(_parts);  
        !i.IsDone();  
        i.Next()  
    ) {  
        i.CurrentItem()->Accept(visitor);  
    }  
    visitor.VisitChassis(this);  
}
```

Composite对象



## 9.代码示例

```
class PricingVisitor : public EquipmentVisitor {
public:
    PricingVisitor();

    Currency& GetTotalPrice();

    virtual void VisitFloppyDisk(FloppyDisk*);
    virtual void VisitCard(Card*);
    virtual void VisitChassis(Chassis*);
    virtual void VisitBus(Bus*);
    // ...
private:
    Currency _total;
};

void PricingVisitor::VisitFloppyDisk (FloppyDisk* e) {
    _total += e->NetPrice();
}

void PricingVisitor::VisitChassis (Chassis* e) {
    _total += e->DiscountPrice();
}
```

ConcreteVisitor

# 9.代码示例

```
class InventoryVisitor : public EquipmentVisitor {
public:
    InventoryVisitor();

    Inventory& GetInventory();

    virtual void VisitFloppyDisk(FloppyDisk*);
    virtual void VisitCard(Card*);
    virtual void VisitChassis(Chassis*);
    virtual void VisitBus(Bus*);
    // ...

private:
    Inventory _inventory;
};

void InventoryVisitor::VisitFloppyDisk (FloppyDisk* e) {
    _inventory.Accumulate(e);
}

void InventoryVisitor::VisitChassis (Chassis* e) {
    _inventory.Accumulate(e);
}
```

状态累计

ConcreteVisitor



## 9.代码示例

---

### ■ client使用

```
Equipment* component;  
InventoryVisitor visitor;  
  
component->Accept(visitor);  
cout << "Inventory "  
      << component->Name()  
      << visitor.GetInventory();
```



# 11. 相关模式

---

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



## 5.12 行为模式的讨论

---



## 5.12.1 封装变化

---

- 行为模式的主题：封装变化
  - Strategy: 封装算法
  - State: 封装与状态有关的行为
  - Mediator: 封装对象间的协议
  - Iterator: 封装访问和遍历聚集对象中各构件的方法



## 5.12.2对象作为参数

---

- Visitor模式
  - 访问者对象总是被用作参数对象
- Command模式
  - Command对象代表一个请求
- Memento模式
  - Memento对象代表一个对象在某个特定时刻额内部状态

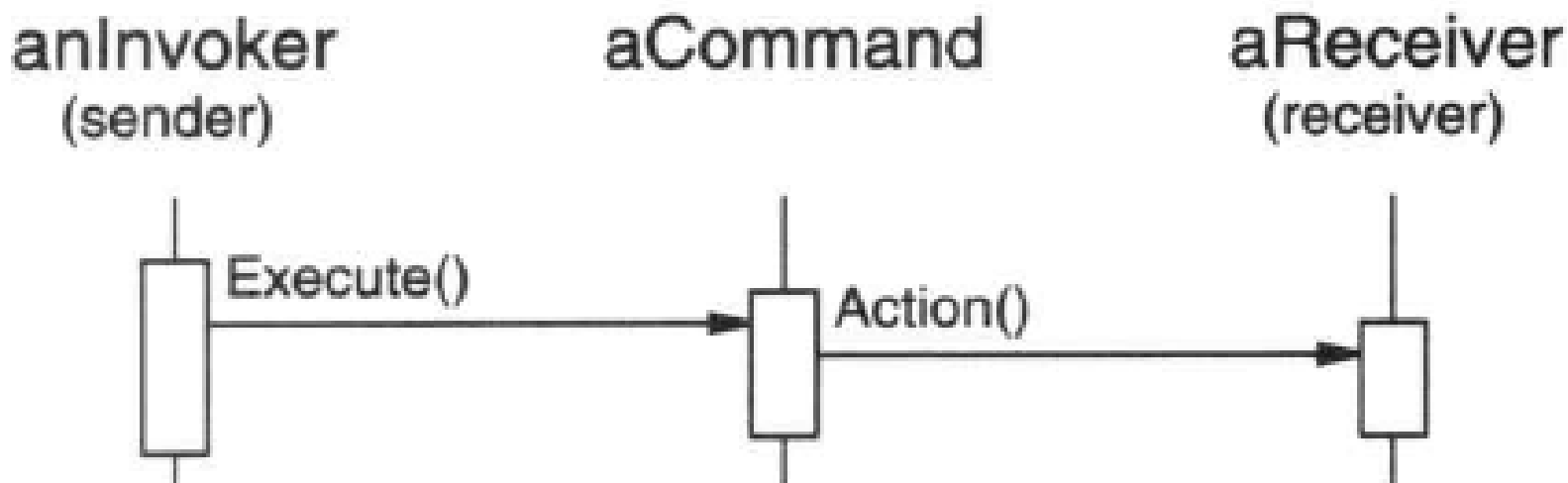
## 5.12.3通信应该被封装还是被分布

- Mediator与Observer
  - Observer模式中引入Observer和Subject对象来分布通信
    - 有利于Observer和Subject间的分割和松耦合
  - Mediator对象封装了其他对象间的通信
    - Mediator中的通信流更容易理解



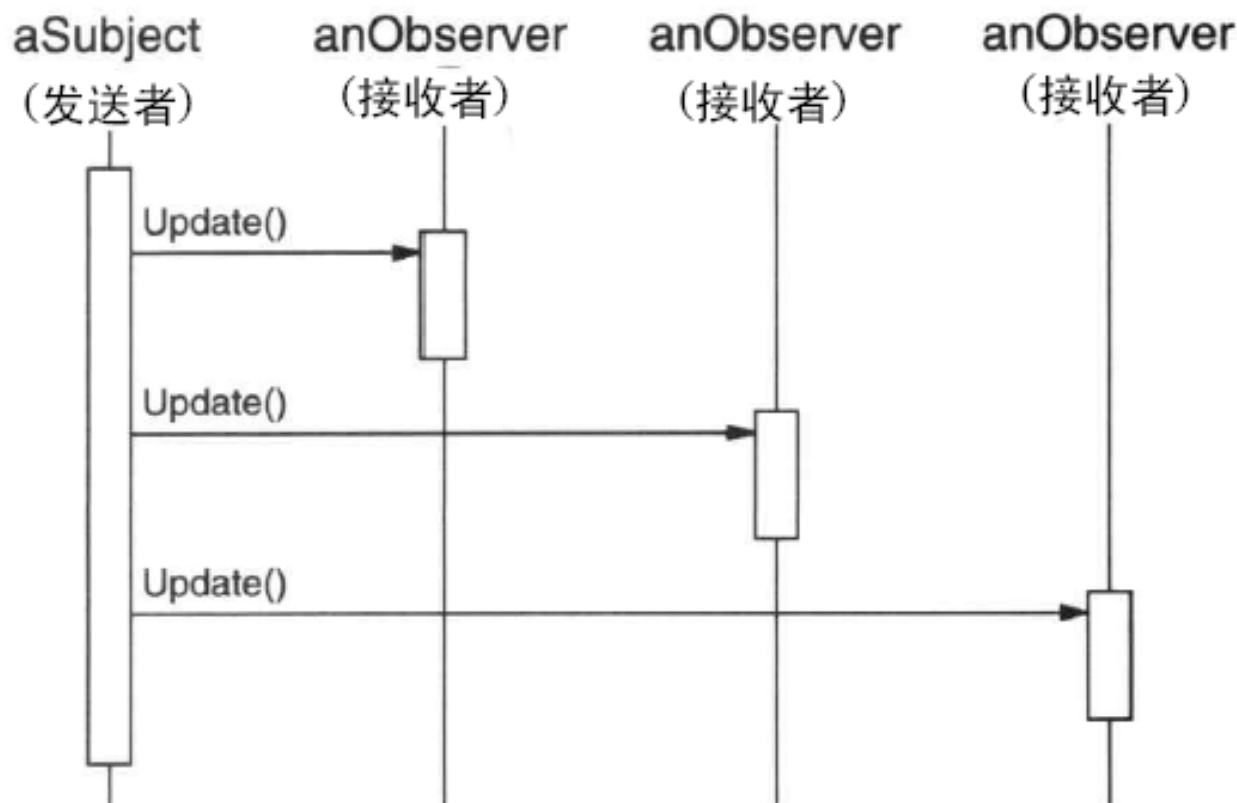
## 5.12.4对发送者和接收者解耦

- Command模式使用Command对象来定义发送者和接收者之间的耦合关系



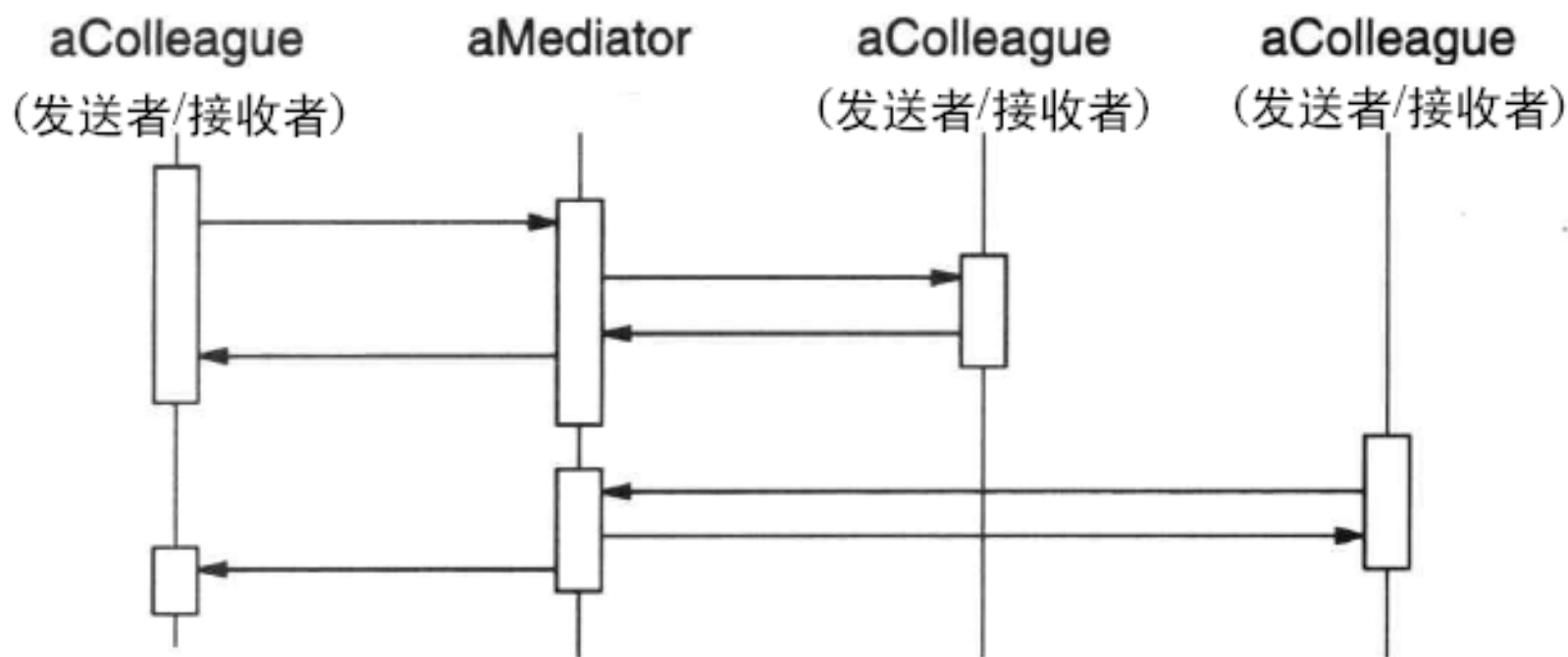
## 5.12.4对发送者和接收者解耦

- Observer: 定义一个接口来通知目标中发生的变化



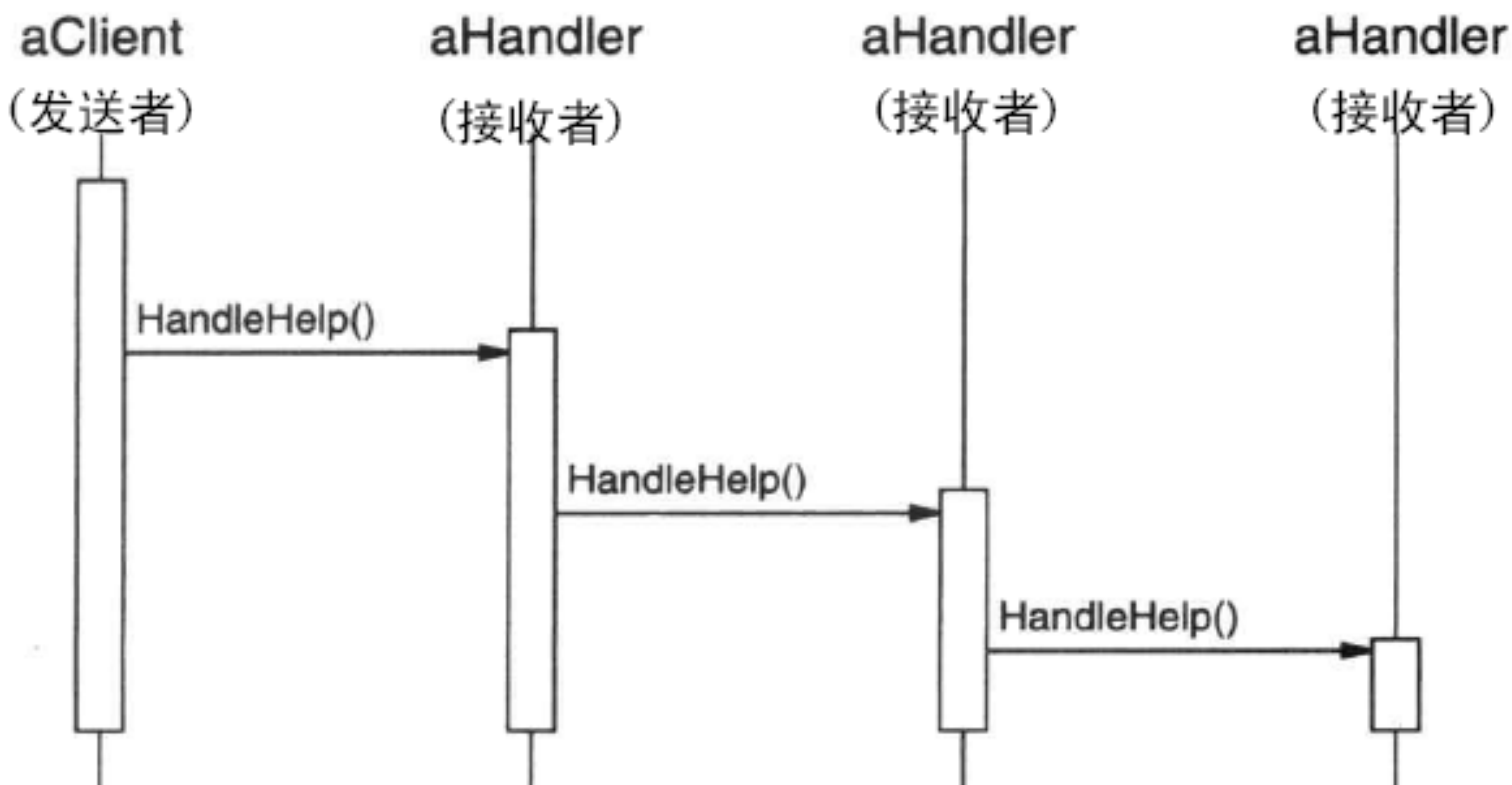
## 5.12.4对发送者和接收者解耦

- Mediator：对象通过Mediator对象间接调用



## 5.12.4对发送者和接收者解耦

- 职责链：沿潜在的接收者链传递请求





谢谢!

---