Design patterns

Design patterns,

Elements of Reusable Object Oriented Software

Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides

Foreword by Grady Booch

Foreword by Booch

- **All well-structured objected-oriented architectures are full of patterns. Indeed, one of the ways that I measure the quality of an object-oriented system is to judge whether or not its developer have paid careful attention to the common collaborations among its objects. Focusing on such mechanism during a system's development can yield an architecture that is smaller, simpler, and far more understandable than if these patterns are ignored.**

Foreword by Booch

- **The importance of patterns in crafting complex system has been long recognized in other disciplines. In particular, Christopher Alexander and his colleagues were perhaps the first to propose the idea of using a pattern language to architect buildings and cities. His ideas and the contributions of others have now taken root in the object-oriented software community. In short, the concept of the design pattern in software provides a key to helping developers leverage the expertise of other skilled architects**

Foreword by Booch

- **In this book, Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides introduce the principles of design patterns and then offer a catalog of such patterns. Thus, this book makes two important contributions.**
  - First, it shows the role that patterns can play in architecting complex system
  - Second, it provides a very pragmatic reference to a set of well-engineered patterns that the practicing developer can apply to crafting his or her own specific application.
- **I'm honored to have had the opportunity to work directly with some of the authors of this book in architectural design efforts. I have learned much from them, and I suspect that in reading this book, you will also.**

简介

本书分两部分：

- 概念：**ern，以及如何用它**
  设计
- 实用

Concern the process of object creation

Deal with the composition of classes or objects

- **Creational Patterns**
- **Structural Patterns**
- **Behavior Patterns**

Characterize the ways in which Classes or objects interact and Distribute responsibility

重要概念

- 目标：设计可重用OO软件。
- 经验的利用 $\longrightarrow$ **Pattern**

  ＋抽象层次 $\longrightarrow$ **design pattern**

  $\Downarrow$

  有助于重用成功软件的design and architecture

- **Design Patterns are description of communicating objects and classes that are customized to solve a general design problem in a particular context**

重要概念（续）

- 通常，对每个Pattern主要包括四个部分：
  - Pattern name
  - 要解决的Problem
  - Solution （解答，抽象层的描述）
  - Consequences （结果，对性能方面的影响和折中）

- 采用MVC：

  Model —application object
  view – screen presentation
  Control – user interface reacts to user input

重要概念（续）

- 如何用它解决实际问题

1. 找到合适的Object
2. 决定Object 的粒度
3. 确定Object 的接口
4. 指定Object的实现 — 用Class
5. Putting reuse mechanisms to work
6. Relating run-time and compile-time structure
7. Design for change

- 区分两个概念：
  - object’s class —定义object的实现（包括内部状态和操作的实现）
  - Object’s type —指interface

  ∴Class 定义了object’s type

  - Class inheritance
  - Interface inheritance

An object may have many types
and widely different objects can share a type

Program to an interface, not an implementation

reuse mechanisms

1. **Class inheritance – white-box reuse**
2. **Object composition – black-box reuse**
3. **Delegation**
4. **Parameterized type**
   - 功能重用的另一种技术，
   同C++中的templates,
   Ada, Eiffel 中的generics

- 类继承： **white-box reuse**
  - 优点： 1. 类继承在编译是静态定义，可以直接重用（因为编程语言直接支持继承）
    2. 易于修改被重用的实现（override）
  - 缺点： 1. 不能在运行时修改从父类继承的实现
    2. 更糟的是父类定义了子类的部分物理表达
       因继承向子类暴露了父类的实现
       ∴继承破坏了封装性
    3. 且子类实现受限于父类，即父类中实际的修改将强制子类的修改
    ＝＝》限制了灵活性和最终的重用性
    ＝》方法之一：只从抽象类继承，但…reuse

- 对象组装： black-box reuse

  要求被组装的对象具有良好的定义接口

  – 优点：1. 它是运行时通过对象应用动态定义（要求接口一致）
    2. Keep each class encapsulated and focused on one work
    3. 类和类继承相对稳定在较小的规模，不会增加管理难度
  – 缺点：1. 会有很多的对象，系统行为将依赖于其间的对象，而非一个类的定义

- 两者的合作：
  – 对象组合组装对象
  – 继承使对象更易于生成 （用于组装）

O-O design 的两个Principles:

Program to an interface, not an implementation

Favor object composition over class inheritance

- **Delegation:**
  – 使得Composition 的重用性与继承一样Powerful
  – 例：

∴类1欲重用类2的方法有两条路径：<功能重用>
  1. 类1作为类2的子类 （继承）
  2. 类1 have 类2 （即类2是类1的组件－delegate)
     类1引用类2的方法 或说 类1将操作委托给类2

主要优点：方便地支持运行时动态地组合behavior，以及其组合方式
缺点：比静态软件难于理解，对运行效率的影响。

三种重用技术比较

- **Inheritance:**
  – 提供操作的缺省实现，且子类可重定义
- **Object Composition:**
  – 可修改运行时组合的行为，但要求间接，且可能降低效率
- **Parameterized types:**
  – 允许修改类所用到的类型（type），但也支持运行时修改。且只能在支持需要Compile-time type checking的语言中。

Design for change

- 因重用主要取决于：
  – 是否加入新的需求
  – 是否已修改已有功能
  – 是否易于升级
  避免重新设计

- 引起redesign 的几个原因：
  – Creating an object by specifying a class explicitly
  – Dependence on specific operation
  – Dependence on hardware and software platform
  – Dependence on object representations or implementation
  – Algorithmic dependencies
  – Tight coupling
  – Extending functionality by subclassing
  – Inability to alter classes conveniently

- **Design pattern 在三类软件开发中所起的作用：**
  - Application programs
  - Toolkits
  - frameworks

第二部分

实用 Design Pattern 分类

- **Creational Patterns**
  - –Abstract Factory
  - –Builder
  - –Factory Method
  - –Prototype
  - –Singleton
- **Structural Patterns**
  - –Adapter
  - –Bridge
  - –Composite
  - –Decorator
  - –Façade
  - –Flyweight
  - –Proxy
- **Behavior Patterns**
  - –Chain of Responsibility
  - –Command
  - –Interpreter
  - –Iterator
  - –Mediator
  - –Memento
  - –Observer
  - –State
  - –Strategy
  - –Template
  - –Visitor.

Creational Patterns

- Abstract Factory (87)
  - Provide an interface for creating families of related or dependent objects without specifying their concrete classes.
- Builder (97)
  - Separate the construction of a complex object from its representation so that the same construction process can create different representations.
- Factory Method (107)
  - Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.
- Prototype (117)
  - Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype.
- Singleton (127)
  - Ensure a class only has one instance, and provide a global point of access to it.

Structural Patterns

- Adapter (139)
  - Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.
- Bridge (151)
  - Decouple an abstraction from its implementation so that the two can vary independently.
- Composite (163)
  - Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.
- Decorator (175)
  - Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.

Structural Patterns（续）

- Facade (185)
  - Provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.
- Flyweight (195)
  - Use sharing to support large numbers of fine-grained objects efficiently.
- Proxy (207)
  - Provide a surrogate or placeholder for another object to control access to it.

Behavioral Patterns

- Chain of Responsibility (223)
  - Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it.
- Command (233)
  - Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.
- Interpreter (243)
  - Given a language, define a represention for its grammar along with an interpreter that uses the representation to interpret sentences in the language.
- Iterator (257)
  - Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.

Behavioral Patterns （续）

- Mediator (273)
  - Define an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently.
- Memento (283)
  - Without violating encapsulation, capture and externalize an object's internal state so that the object can be restored to this state later.
- Observer (293)
  - Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.
- State (305)
  - Allow an object to alter its behavior when its internal state changes. The object will appear to change its class.

Behavioral Patterns （续）

- Strategy (315)
  - Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.
- Template Method (325)
  - Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.
- Visitor (331)
  - Represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates.

For Every Pattern

- **Pattern Name and Classification**
- **Intent (目的）**
- **Also Known As （别名）**
- **Motivation （问题的提出）**
- **Applicability （什么情况下使用）**
- **Structure （结构）**
- **Participants （元素）**
- **Collaborations** —How the participants collaborate to carry out their responsibilities.

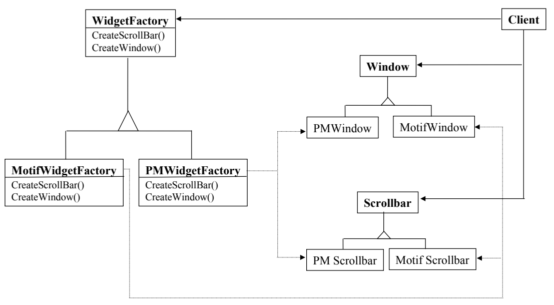
- **Consequences** —How does the pattern support its objectives? What are the trade-offs and results of using the pattern? What aspect of system structure does it let you vary independently?
- **Implementation** —What pitfalls, hints, or techniques should you be aware of when implementing the pattern? Are there language-specific issues?
- **Sample Code** —in C++ or Smalltalk.
- **Known Uses**
- **Related Patterns** —What design patterns are closely related to this one? What are the important differences? With which other patterns should this one be used?

Creational Patterns

- 创建型模式抽象了实例化的过程，从而帮助系统独立于对象的如何创建、组成和表现。类创建模式使用继承了变化实例化的类，而对象创建模式则将实例化委派给其它对象。
- 在该模式中不断出现的主题：
  - 封装了系统所使用的具体类的信息。
  - 隐藏了如何创建和组成这些具体类的实例。
- 对于对象，系统所了解的是抽象类所定义的接口。创建性模式在所创建的对象、创建人、创建方法和时间上提供了灵活性。从而用户可用结构上、功能上不同的对象来配置系统。配置可以是静态的，也可是动态的。

Creational Patterns

- **Abstract Factory:** provide an interface for creating families of related or dependent objects without specifying their concrete classes.

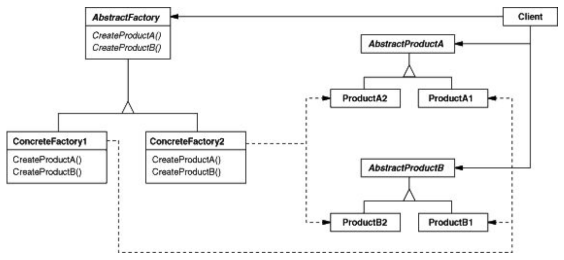


Abstract Factory

- 何时用：
  - a system should be independent of how its products are created, composed, and represented.
  - a system should be configured with one of multiple families of products.
  - a family of related product objects is designed to be used together, and you need to enforce this constraint.
  - you want to provide a class library of products, and you want to reveal just their interfaces, not their implementations.

Abstract Factory

- 结构



Abstract Factory

- **Consequences**

  1. It isolates concrete classes.

  2. It makes exchanging product families easy.

  3. It promotes consistency among products.

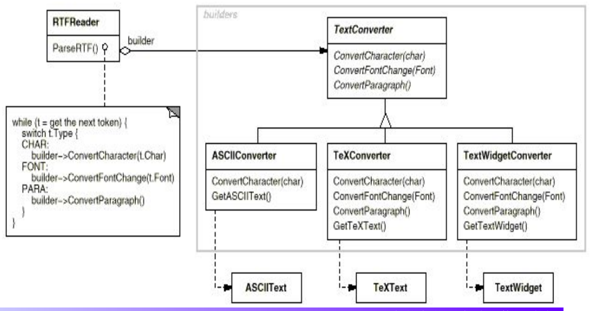
  4. Supporting new kinds of products is difficult.

Creational Patterns

- **Builder**
  - Separate the construction of a complex object from its representation so that the same construction process can create different representations.

Builder

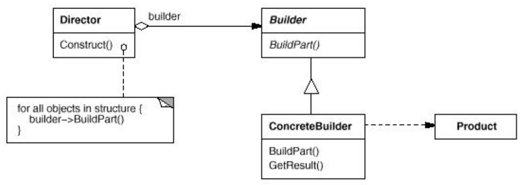
- 来源



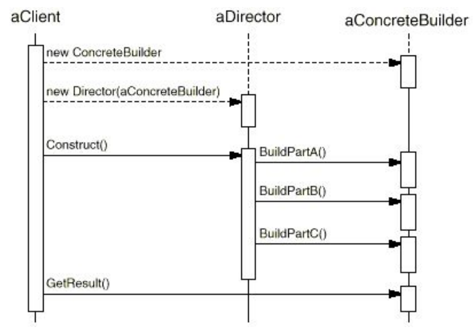
```
while (t = get the next token) {
  switch t.Type {
  CHAR:
    builder->ConvertCharacter(t.Char)
  FONT:
    builder->ConvertFontChange(t.Font)
  PARA:
    builder->ConvertParagraph()
  }
}
```

Builder

何时用：
- the algorithm for creating a complex object should be independent of the parts that make up the object and how they're assembled.
- the construction process must allow different representations for the object that's constructed.



```
for all objects in structure {
    builder->BuildPart()
}
```



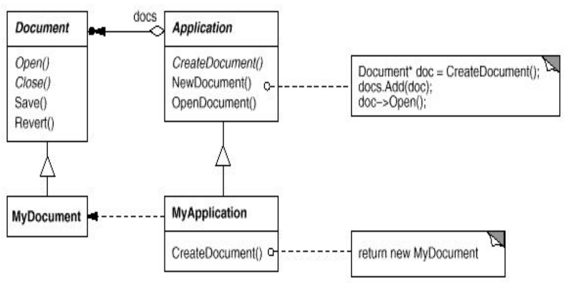
- **Consequences**

1. *It lets you vary a product's internal representation.*
2. *It isolates code for construction and representation.*
3. *It gives you finer control over the construction process.*

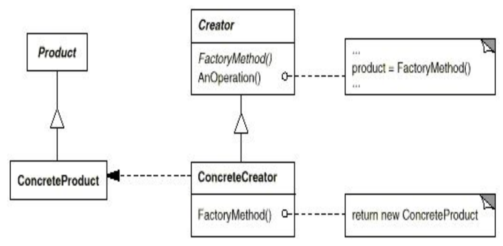
Factory Method

- **Factory Method**
  - Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.
- **Applicability**
  - a class can't anticipate the class of objects it must create.
  - a class wants its subclasses to specify the objects it creates.
  - classes delegate responsibility to one of several helper subclasses, and you want to localize the knowledge of which helper subclass is the delegate.

Factory Method



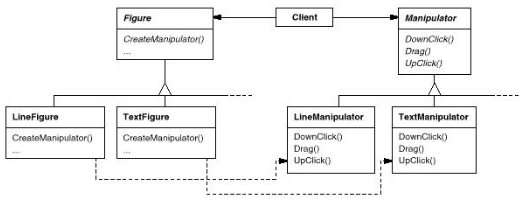
Factory Method



Factory Method

- **Consequences**

1. *Provides hooks for subclasses.*
2. *Connects parallel class hierarchies.*



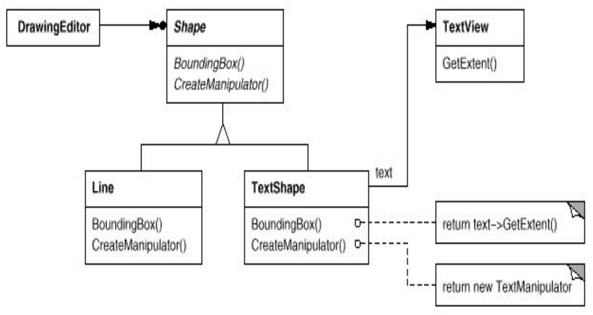
Structural Patterns

- 结构性模式关注于如何将类和对象组合成更大的结构。*结构性类模式*使用*继承*来组合接口或实现。
  - Adapter
  - Bridge
  - Composite
  - Decorator
  - Façade
  - Flyweight
  - Proxy

Adaptor

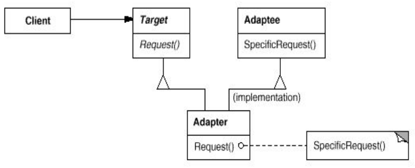
- 目的：
  - *将类接口转化为用户所希望的接口。Adapter使原本由于不兼容接口而无法使用的类一同工作。*
- 问题的提出

Structural Patterns -Adaptor

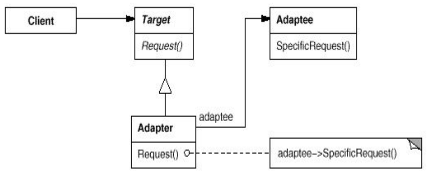


Adaptor

- 多继承



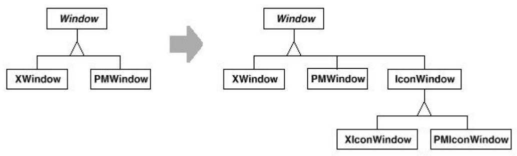
- 对象匹配



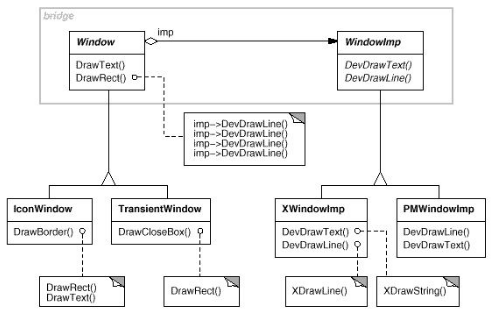
Structural Patterns -bridge

- **Bridge**
  - Decouple an abstraction from its implementation so that the two can vary independently.

  (减少抽象与实现之间的耦合，使它们能对立地变化)
- **Motivation(问题的提出)**

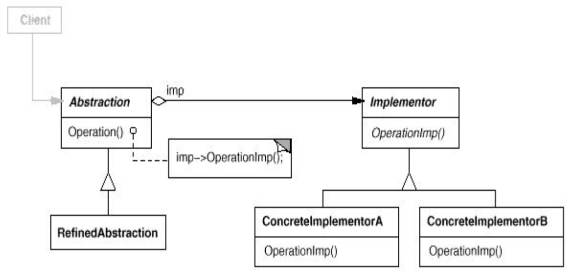


Structural Patterns -bridge



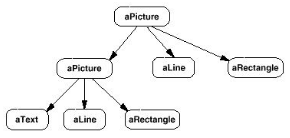
Structural Patterns -bridge

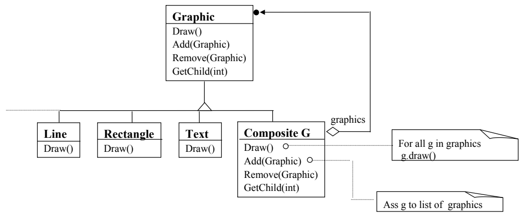
- 结构



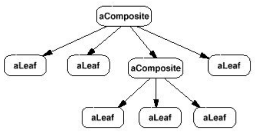
Structural Patterns

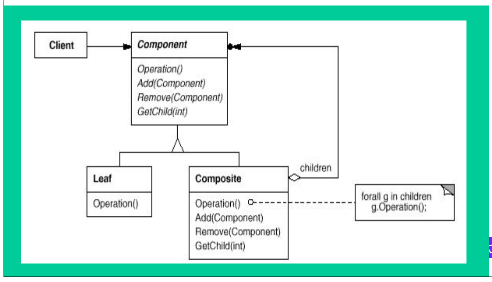
- **Composite:** compose objects into tre
hierarchies. Composite lets clients treat
objects uniformly.





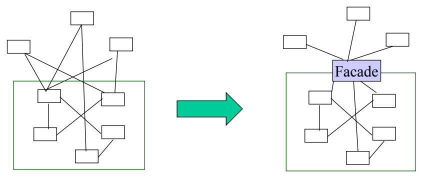
- 结构





Structural Patterns (II)

- **Facade:** provide a unified interface to a set of interface in subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.



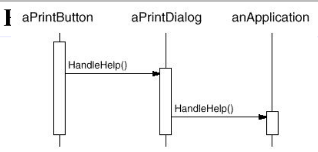
Behavior Patterns

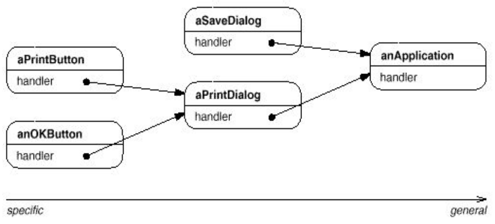
- 行为模式涉及到算法和对象间职责的分配。
- 行为模式不仅描述了对象和类的模式，还描述了它们之间的通讯模式
- 分为两类：
  - 使用继承机制：如Template Method, Interpreter
  - 使用对象组合：如Mediator, Chain of Responsibility 等

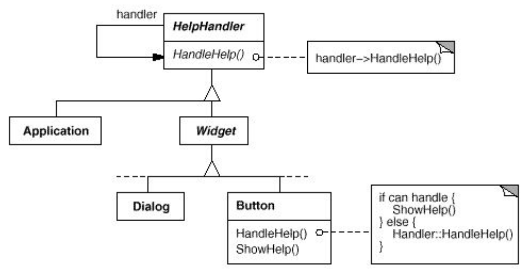
Behavior Patterns—

- Chain of responsibility:
- Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it.

• 问题的提出





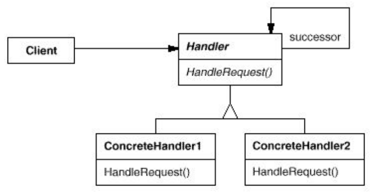


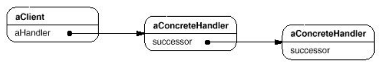
Behavior Patterns

• 何时用
  – more than one object may handle a request, and the handler isn't known *a priori*. The handler should be ascertained automatically.
  – you want to issue a request to one of several objects without specifying the receiver explicitly.
  – the set of objects that can handle a request should be specified dynamically.

Behavior Patterns

• 结构



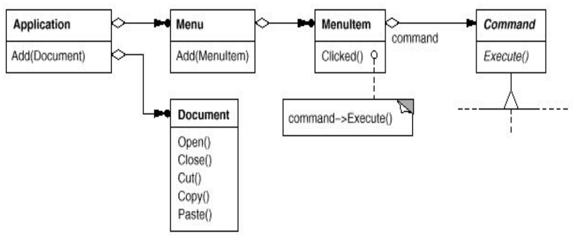


Behavior Patterns—command

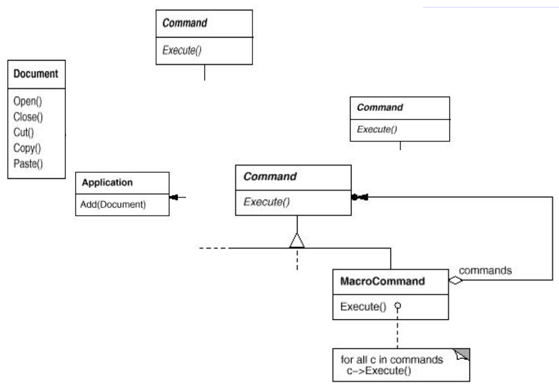
• **command**
  – Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.

• 问题的提出

command1

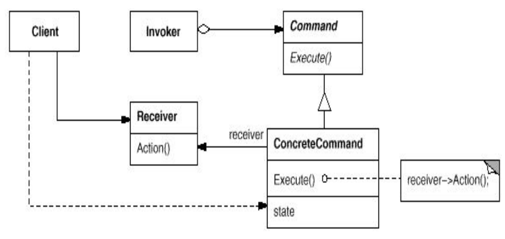


command2



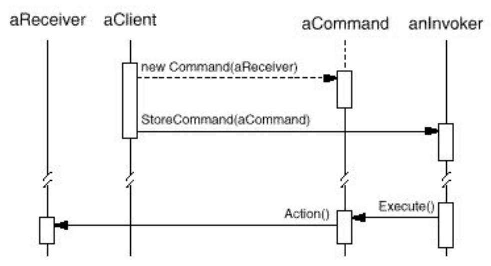
command3

- 结构



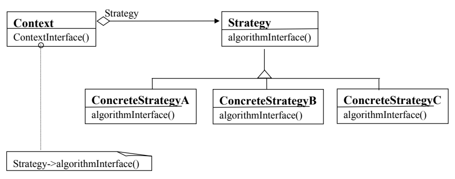
command4

- Collaboration



Behavior Patterns

- **Strategy:** Define a family of algorithms, encapsulate each other, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.


Context
ContextInterface()
Strategy
Strategy
algorithmInterface()
ConcreteStrategyA
algorithmInterface()
ConcreteStrategyB
algorithmInterface()
ConcreteStrategyC
algorithmInterface()
Strategy->algorithmInterface()