软件设计模式(二)

潘爱民

http://www.icst.pku.edu.cn/compcourse

内容

- 复习
- 续：介绍一些重要的模式
  - Structural Patterns
  - Behavioral Patterns
- 第二次作业

复习：pattern定义

- 定义：
  - 特定环境中问题的成功解决方案中的静态、动态结构，以及结构元素相互之间的协作关系
  - Design patterns represent solutions to problems that arise when developing software within a particular context
- 关于pattern的研究状况
  - 研究历史
  - 现状
- pattern与框架
- pattern的分类
  - 粒度

复习：如何描述一个模式

- 关键要素
  - Design pattern name， Aliases or Also Known As
  - Problem， Intent or Goal
  - Forces， Constraints， Motivation
  - Context， Applicability
  - Solution
    - Structure
    - Participants
    - Collaboration
    - Implementation
  - Evaluation， Resulting Context， Consequences
  - Related Patterns
  - Examples， Known uses

复习：creational patters

- Factory Method
  - 本质：用一个virtual method完成创建过程
- Abstract Factory
  - 一个product族的factory method构成了一个factory接口
- Prototype
  - 通过product原型来构造product，Clone+prototype manager
- Builder
  - 通过一个构造算法和builder接口把构造过程与客户隔离开
- Singleton
  - 单实例类型，如何构造这单个实例？如何访问这单个实例？
- Finder
  - 把对象的获取过程与客户隔离开

Structural Patterns

- Adapter
- Bridge
- Composite
- * *Decorator*
- Facade
- Flyweight
- * *Proxy*

模式 7： Adapter (一)

- Aliases： Wrapper
- Intent
  - Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.
- Motivation
  - Sometimes a toolkit class that's designed for reuse isn't reusable only because its interface doesn't match the domain-specific interface an application requires.

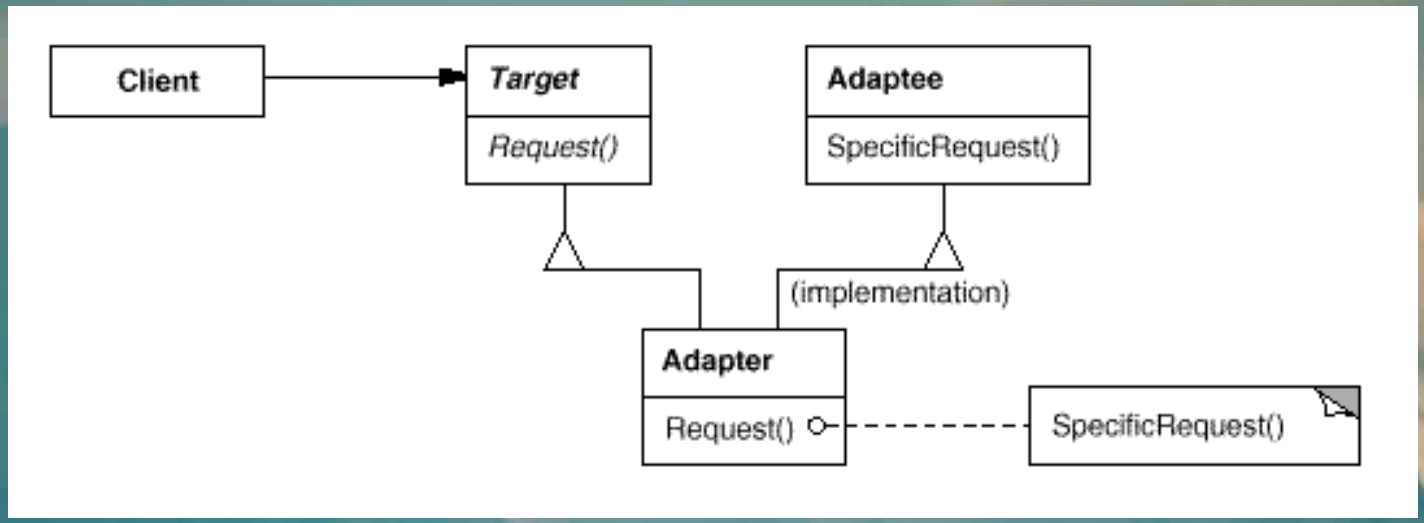
Adapter模式(二)

- Applicability：Use the Adapter pattern when
  - you want to use an existing class, and its interface does not match the one you need.
  - you want to create a reusable class that cooperates with unrelated or unforeseen classes, that is, classes that don't necessarily have compatible interfaces.
  - (object adapter only) you need to use several existing subclasses, but it's impractical to adapt their interface by subclassing every one. An object adapter can adapt the interface of its parent class.

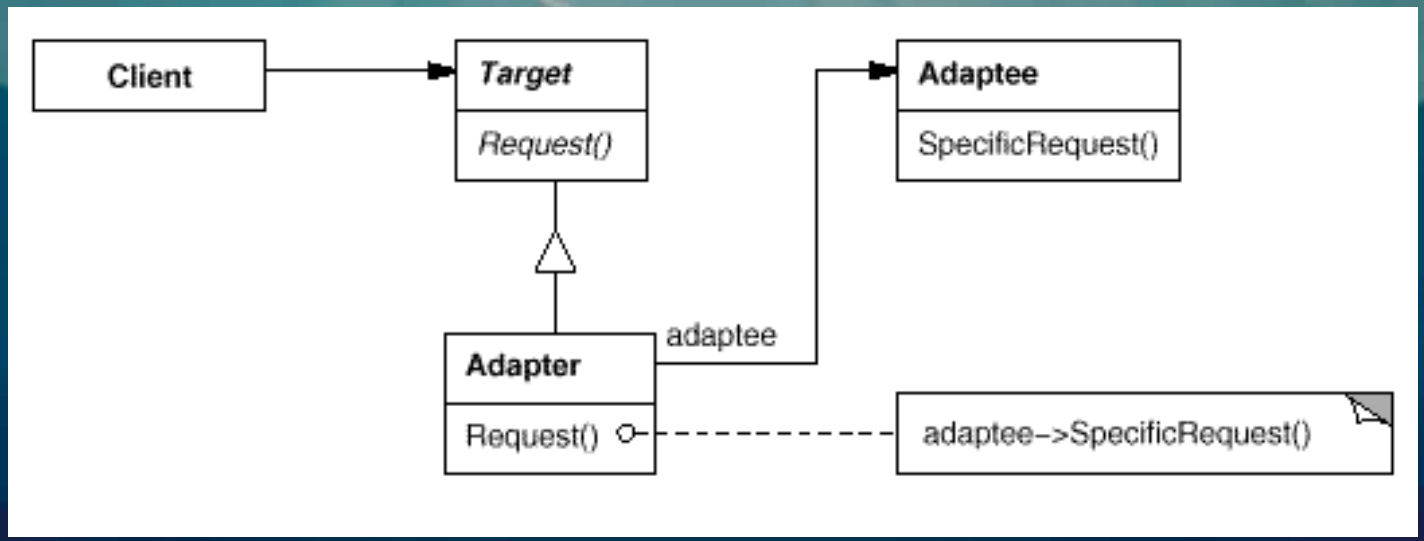
Adapter模式(三)

- Struct

class adapter



object adapter



Adapter模式(三)

- Participants
  - Client、Target、Adaptee、Adapter
- Collaborations
  - class adapter —— delegation
  - object adapter —— container

Adapter模式(四)

- Evaluation
  - 本质上是两种重用模型
  - class adapter：
    - 无法adapt adaptee的子类，但是可以重载adaptee的行为
  - object adapter
    - 可以adapt adaptee的所有子类
  - How much adapting does Adapter do?
  - Pluggable adapters
  - Using two-way adapters to provide transparency
    - 针对class adapter，用多重继承来实现

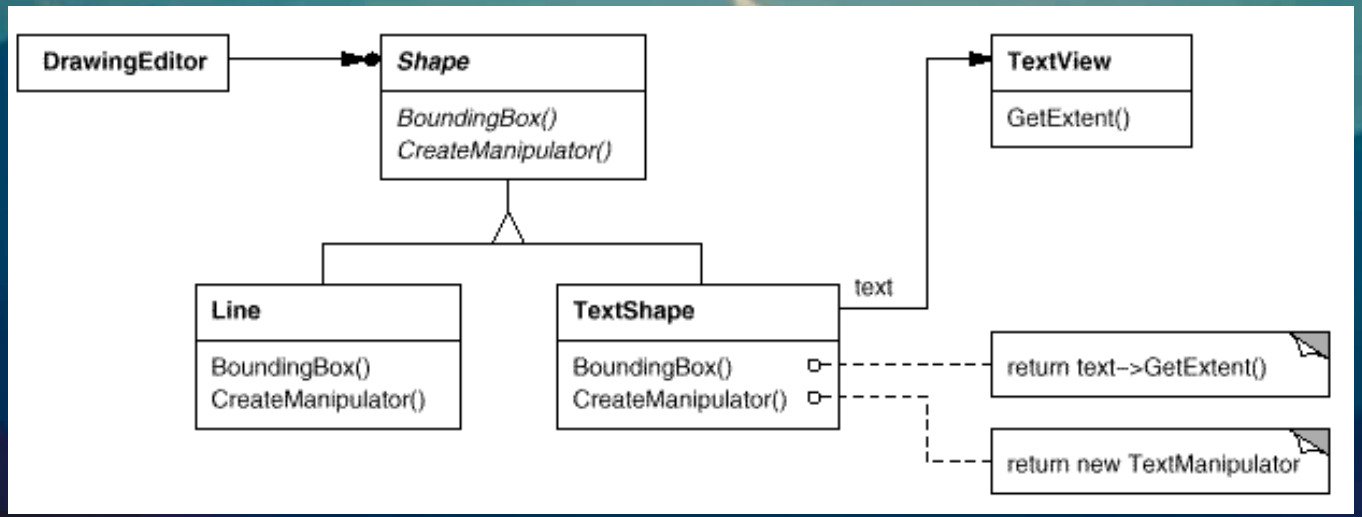
Adapter模式(五)

- Implementation
  - 使用C++继承机制实现class adapter
  - 使用内嵌对象技术实现object adapter
  - Pluggable adapters，三种实现方案
    - 使用抽象方法定义
    - 使用代理对象
    - 参数化技术

    这三种方法的实质：如何在一个类中定义抽象操作，供客户插入？—— hook 技术

Adapter模式(六)

- Related Patterns
  - Bridge
  - Decorator
  - Proxy
- Examples
  - DrawCli：COleDrawObj
  - C++ STL
  - COM中的site



模式 8：Bridge(一)

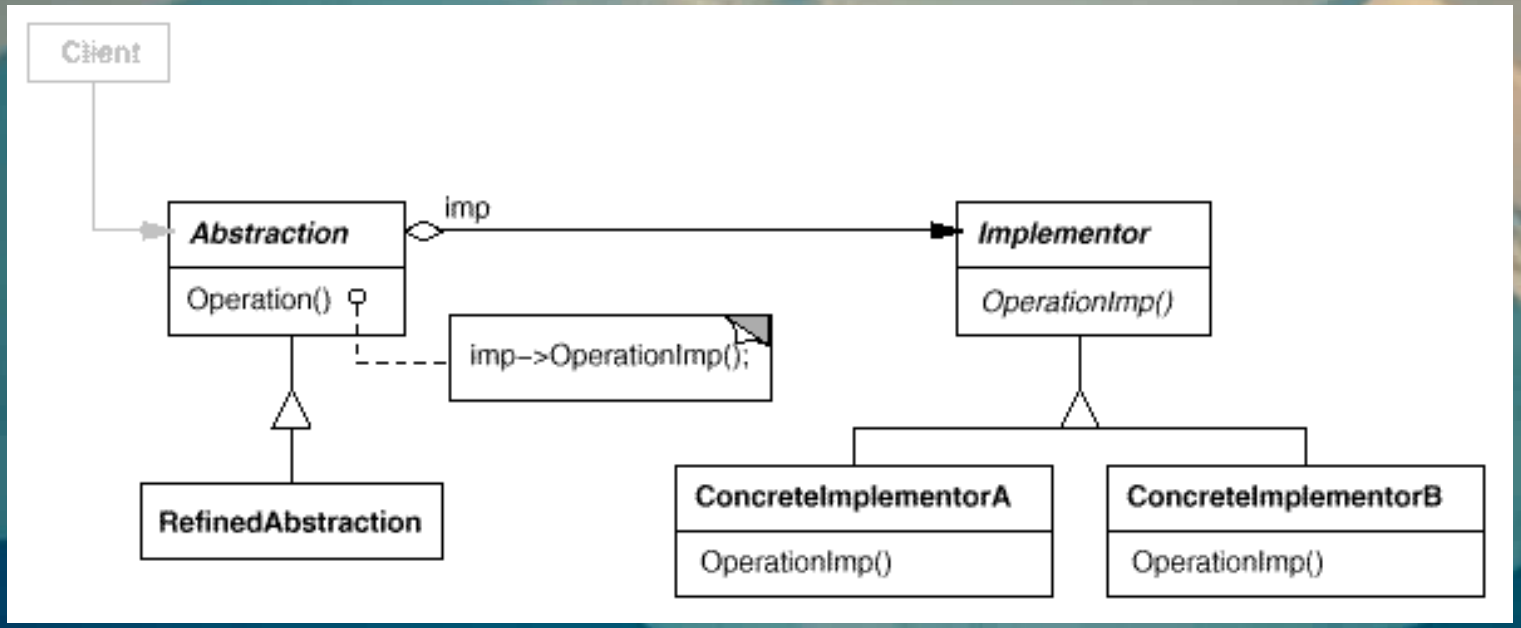
- Aliases：Handle/Body
- Intent
  - Decouple an abstraction from its implementation so that the two can vary independently
- Motivation
  - 要做到“抽象(接口)与实现分离”，最常用的办法是定义一个抽象类，然后在子类中提供实现。也就是说，用继承机制达到“抽象(接口)与实现分离”
  - 但是这种方法不够灵活，继承机制把实现与抽象部分永久地绑定起来，要想独立地修改、扩展、重用抽象(接口)与实现都非常困难。

Bridge模式(二)

- Applicability：Use the Bridge pattern when
  - 编译时刻无法确定抽象(接口)与实现之间的关系
  - 抽象部分与实现部分都可以通过子类化而扩展
  - 对一个实现的修改不影响客户(无须重新编译)
  - 在C++中，对客户完全隐瞒实现细节
  - 因为扩展的原因，需要把一个类分成两部分，(以便灵活组合)
  - 在多个对象之间共享数据，但客户不需要知道

Bridge模式(三)

- Struct



- Participants
  - Client, Abstraction, RefinedAbstraction, Implementor, ConcreteImplementor
- Collaborations

Bridge模式(四)

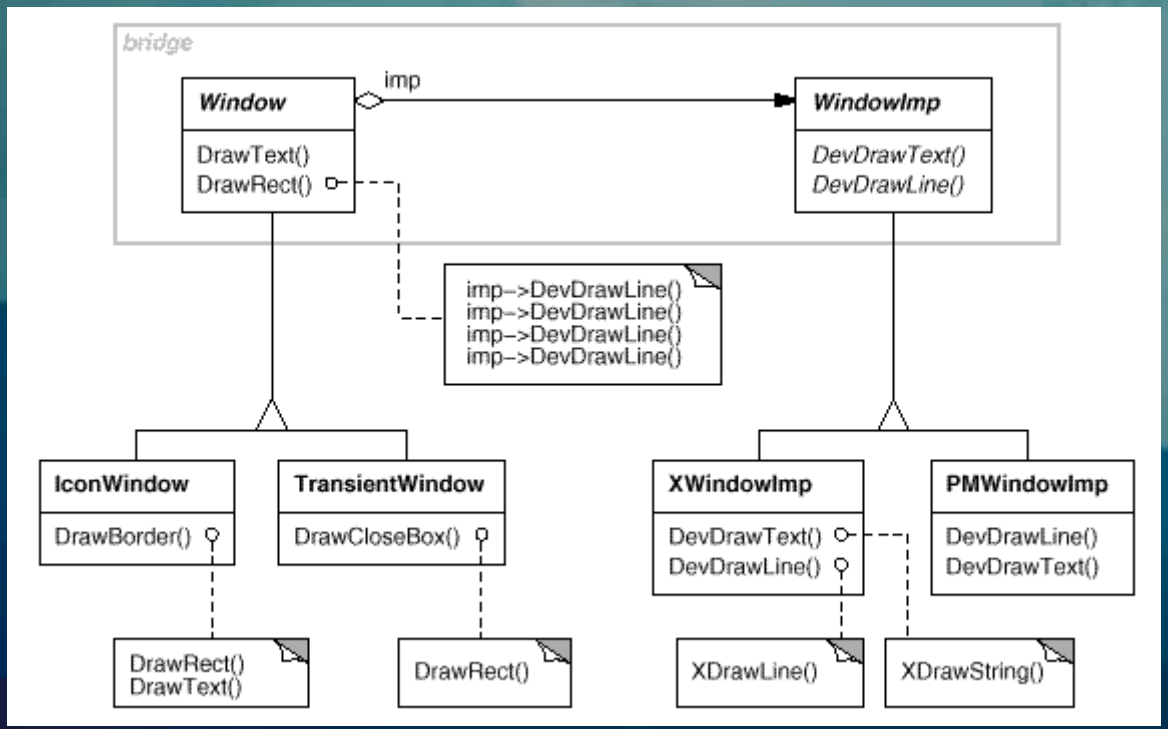
- Evaluation
  - 抽象部分与实现部分的分离，可以在运行时刻连接起来
    二进制兼容性
  - 提高可扩充性：抽象与实现两部分可以单独扩充
  - 对客户隐藏实现细节

Bridge模式(五)

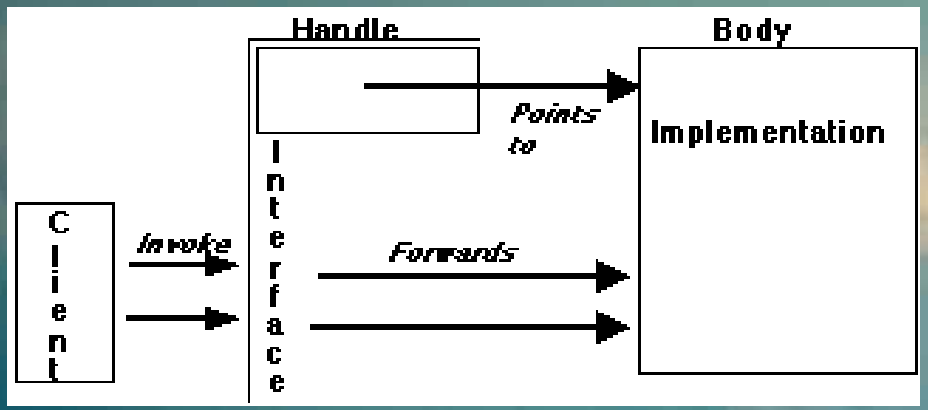
- Implementation
  - Only one Implementor
  - Creating the right Implementor object
    如何创建？根据客户环境，或者通过factory
  - Sharing implementors
    - 资源管理：引用计数技术
  - Using multiple inheritance

Bridge模式(六)

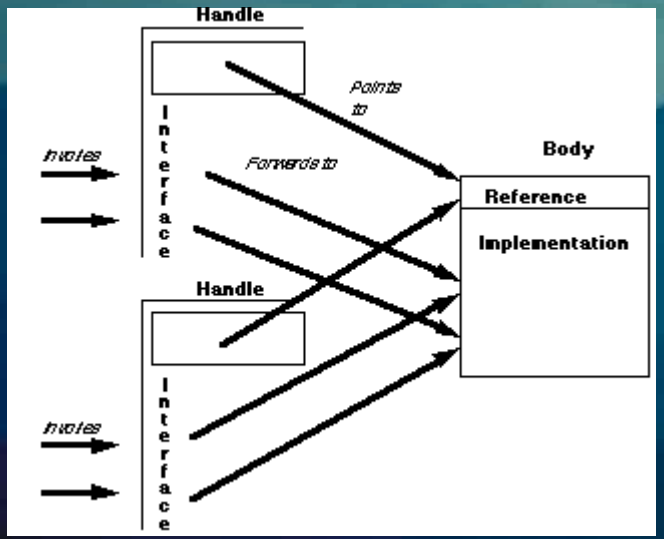
- Related Patterns
  - Abstract Factory可以用来创建和配置Bridge模式
  - 与Adapter模式的区别
- Examples
  - handle：文件handle、窗口handle



插：Handle/Body



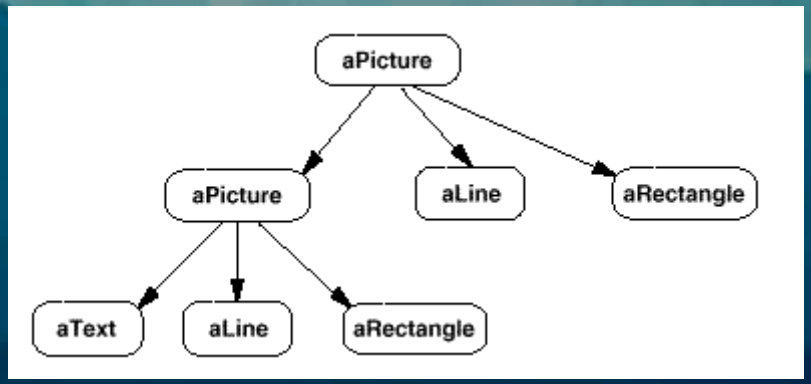
Counted Handle/Body




```
class StringRep {
    friend class String;
    StringRep(const char *s);
    ~StringRep();
    int count;
    char *rep;
};
class String {
public:
    String();
    String(const String &s);
    String &operator=(const String &s);
    ~String();
    String(const char *s);
    . . . .
private:
    StringRep *rep;
};
```

模式 9：Composite(一)

- Intent
  - Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.
- Motivation
  - 一些部件对象经过组合构成的复合部件对象仍然具有单个部件对象的接口，这样的复合部件对象被称为“容器(container)”
  - 复合部件与单个部件具有同样的接口，所有接口包含两部分：单个部件的功能、管理子部件的功能
  - 递归组合

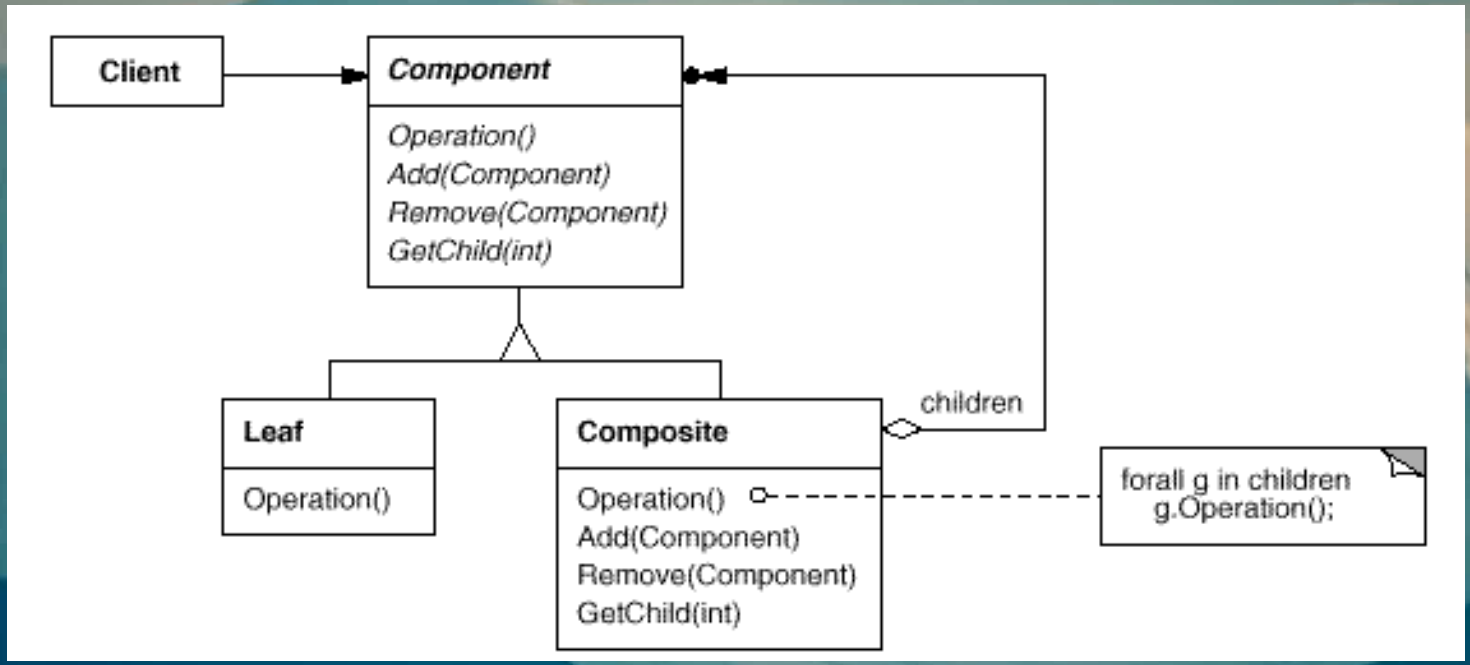


Composite模式(二)

- Applicability：Use the Composite pattern when
  - you want to represent part-whole hierarchies of objects.
  - you want clients to be able to ignore the difference between compositions of objects and individual objects. Clients will treat all objects in the composite structure uniformly.

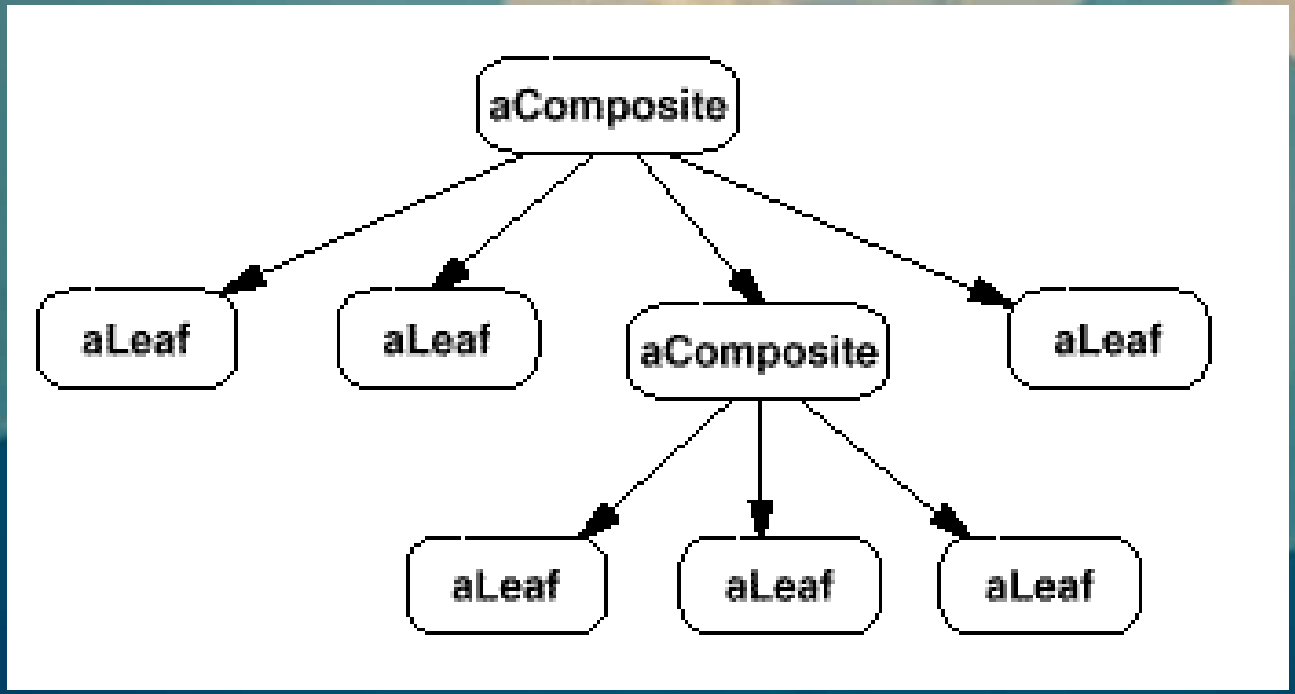
Composite模式(三)

- Struct



- Participants
  - Client, Component, Leaf, Composite
- Collaborations

典型的composite对象结构



Composite模式(四)

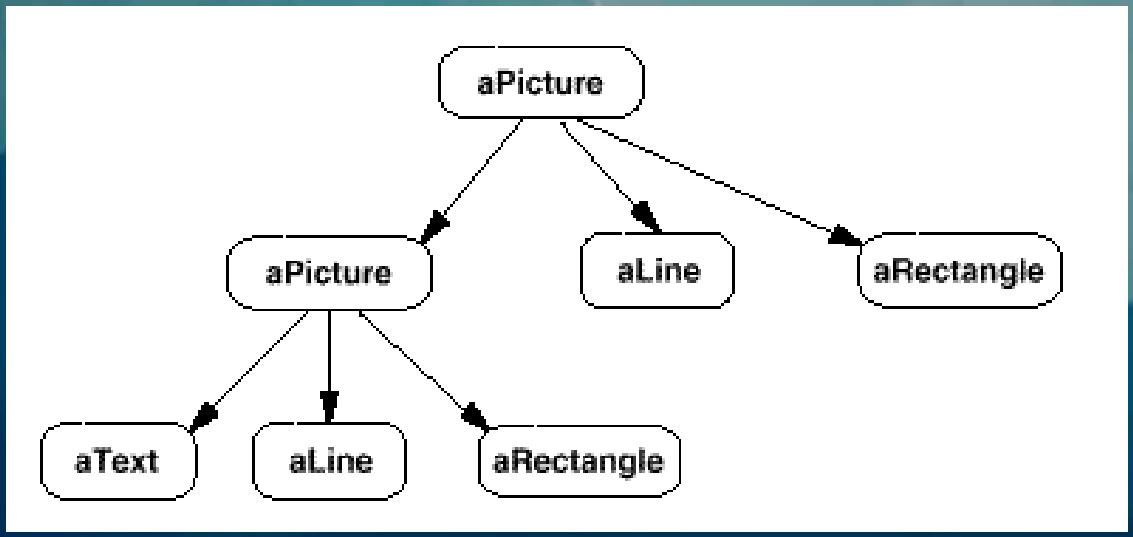
- Evaluation
  - defines class hierarchies consisting of primitive objects and composite objects。定义了包含leaf对象和composite对象的类层次接口。——递归结构
  - makes the client simple。客户一致地处理复合对象和单个对象
  - makes it easier to add new kinds of components。易于增加新类型的组件
  - can make your design overly general。使得系统过于一般化，无法限制类型的组合，可以在运行时刻通过类型检查加以弥补

Composite模式(五)

- Implementation
  - Explicit parent references
  - Sharing components
  - Maximizing the Component interface
  - Declaring the child management operations
  - Should Component implement a list of Components?
  - Child ordering
  - Caching to improve performance
  - Who should delete components?
  - What's the best data structure for storing components?

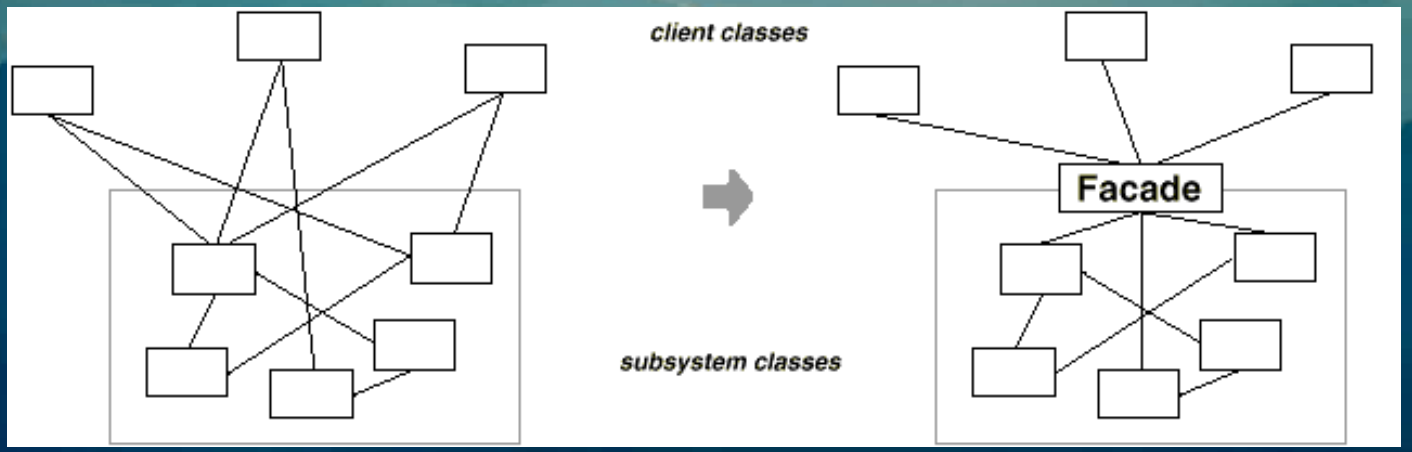
Composite模式(六)

- Related Patterns
  - Decorator、Flyweight、Iterator、Visitor
- Examples
  - 广泛应用于OO领域
  - MFC中的CWnd
  - 组件层次：ActiveX Container



模式 10：Facade(一)

- Intent
  - 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.
- Motivation
  - 使系统的各子系统之间的关联最小，引入一个facade对象，为子系统提供一个简单的、泛化的设施

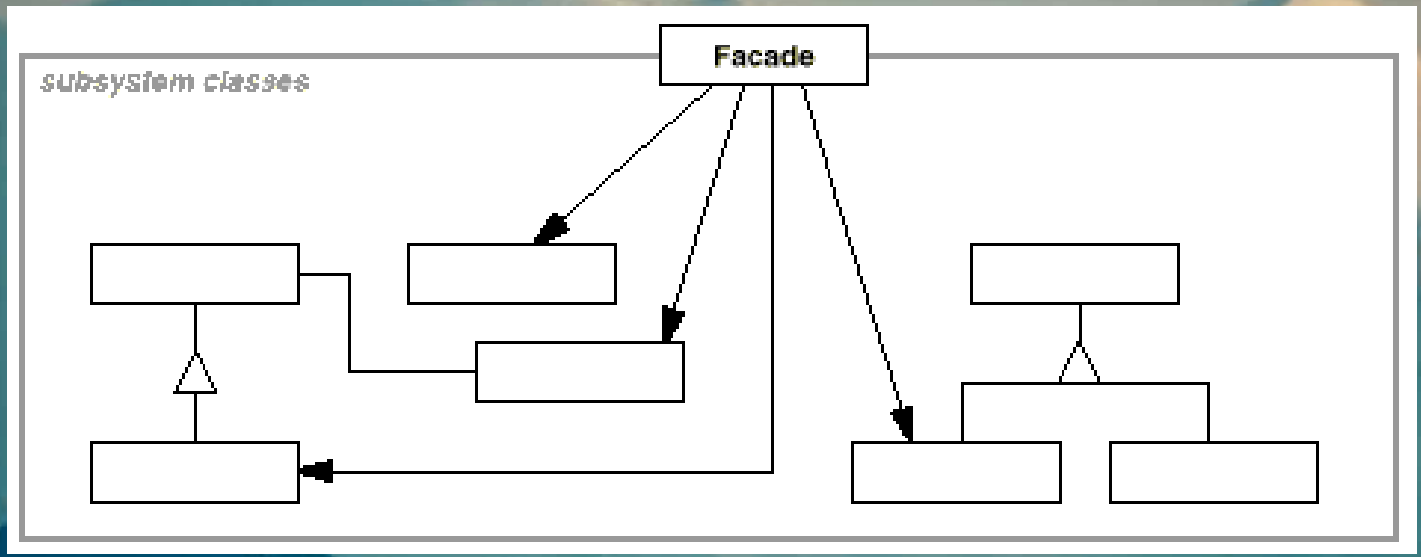


Facade模式(二)

- Applicability： Use the Facade pattern when
  - 为一个复杂的子系统提供一个简单接口时。子系统往往会非常复杂，但是其接口应该尽可能地简单，特别是对于一般用户而言
  - 客户与抽象类的实现部分之间必然存在一定的依赖性，facade可以降低这种依赖性
  - 在多个子系统的结构中，使用facade模式定义子系统的入口点，有助于降低各子系统之间的依赖性

Facade模式(三)

- Struct



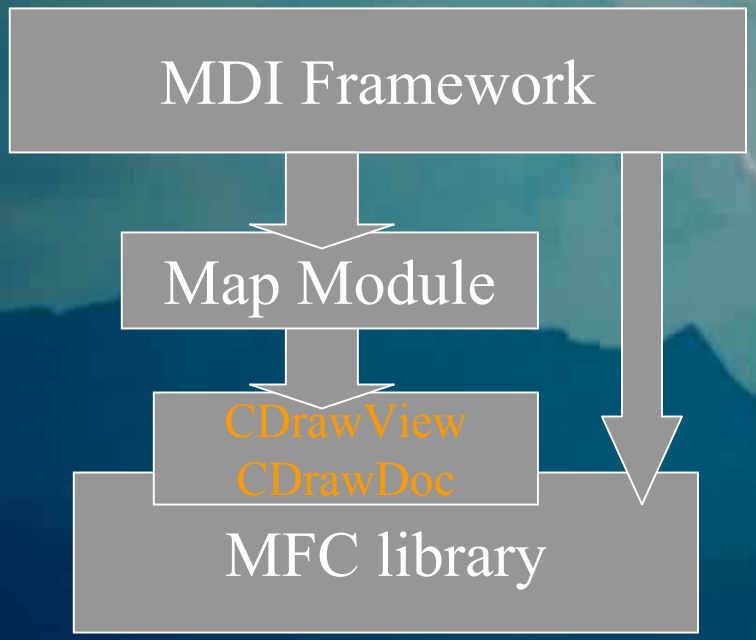
- Participants
  - facade, subsystem classes
- Collaborations

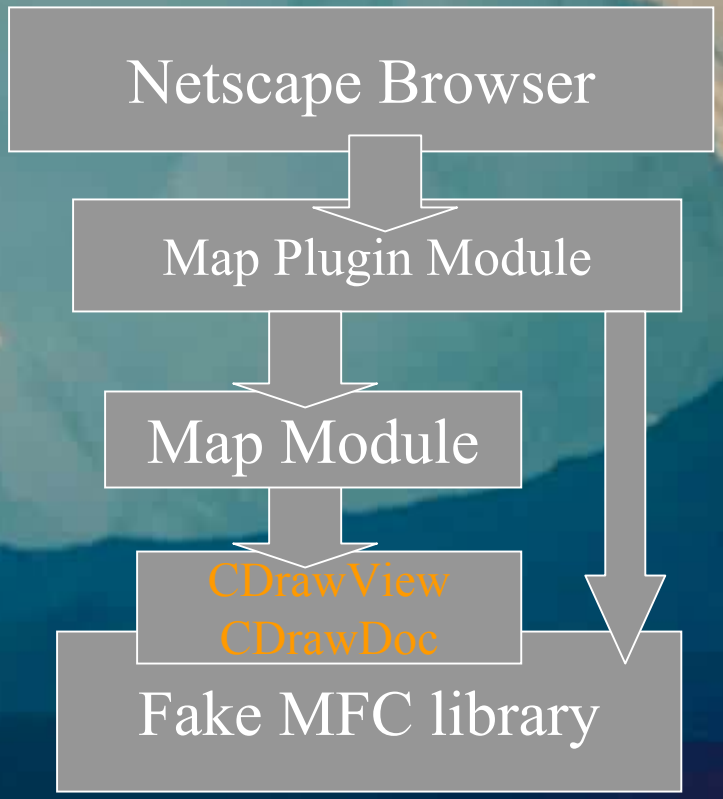
Facade模式(四)

- Evaluation
  - 简化子系统的接口，方便客户使用子系统
  - 化“紧耦合”为“松耦合”
    —— 实现组件软件的关键技术
  - facade模式并不限制客户直接访问子系统的内部类和对象
- Implementation
  - 以抽象类的形式定义facade，进一步decouple，从而完全隔离子系统的细节
  - Public versus private subsystem classes

Facade模式(五)

- Related Patterns
  - facade对象的创建： singleton、abstract factory
- Examples





模式 11：FlyWeight(一)

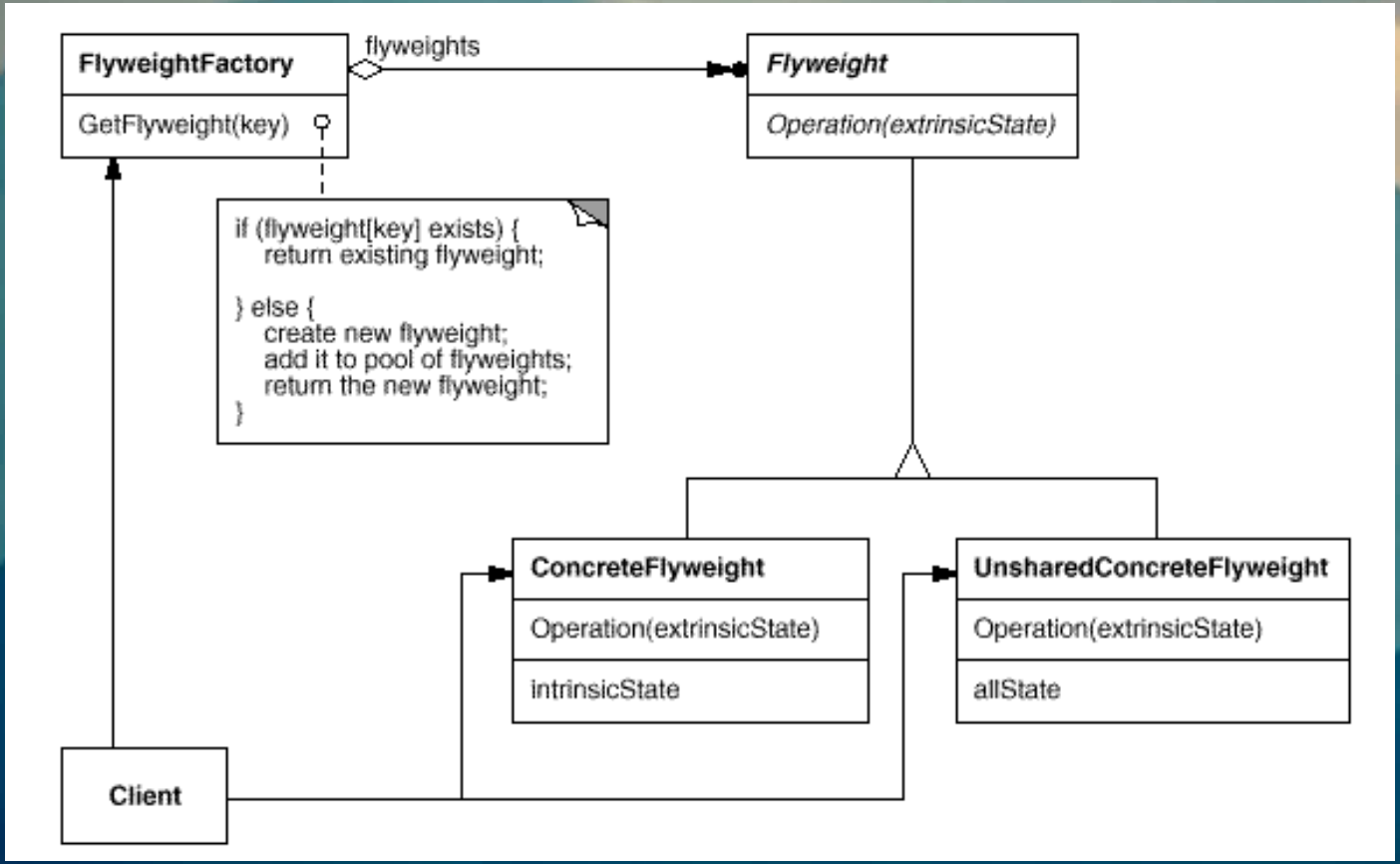
- Intent
  - Use sharing to support large numbers of fine-grained objects efficiently.
- Motivation
  - 当对象的粒度太小的时候，大量对象将会产生巨大的资源消耗，因此考虑用共享对象(flyweight)来实现逻辑上的大量对象。Flyweight对象可用于不同的context中，本身固有的状态不随context发生变化，而其他的状态随context而变化

FlyWeight模式(二)

- Applicability：Use the FlyWeight pattern when all of the following are true:
  - An application uses a large number of objects.
  - Storage costs are high because of the sheer quantity of objects.
  - Most object state can be made extrinsic.
  - Many groups of objects may be replaced by relatively few shared objects once extrinsic state is removed.
  - The application doesn't depend on object identity. Since flyweight objects may be shared, identity tests will return true for conceptually distinct objects.

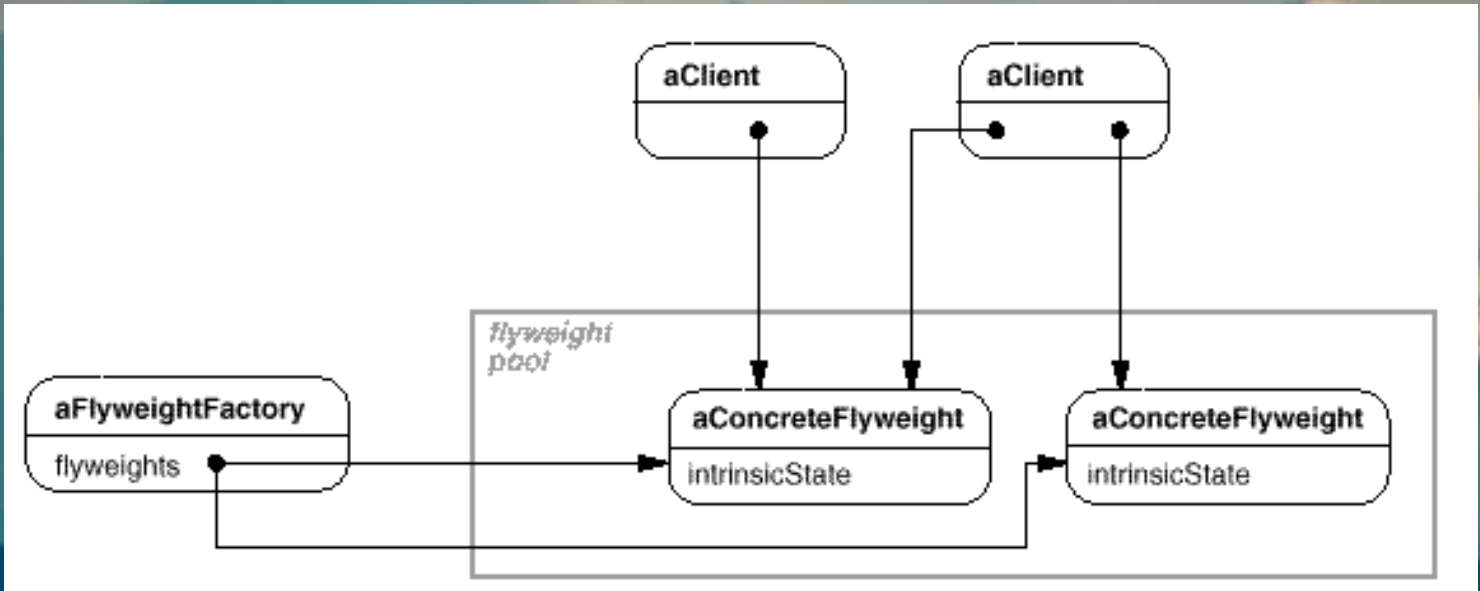
FlyWeight模式(三)

- Struct



FlyWeight模式(四)

- Struct(续)



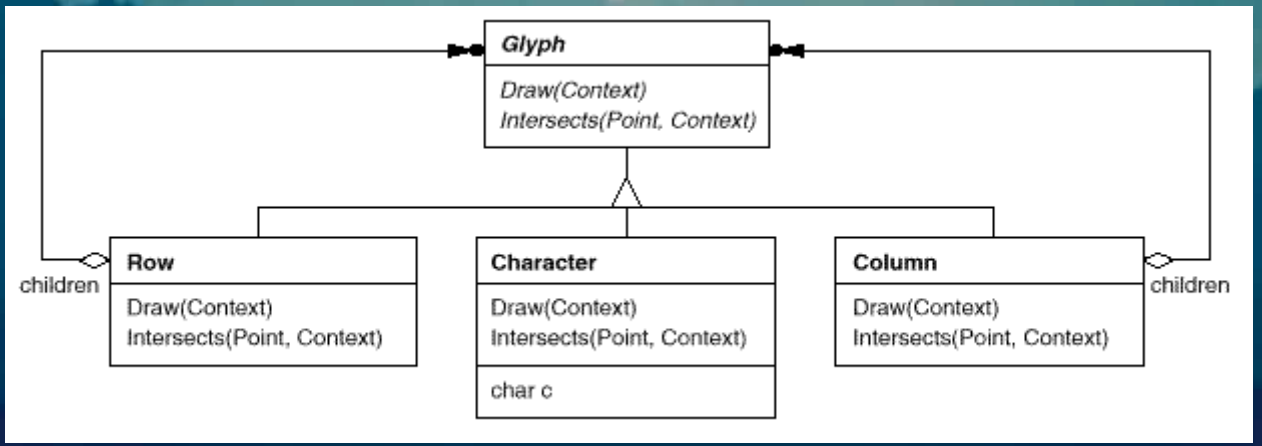
- Participants
  - client, flyweight, concreteFlyweight, FlyweightFactory,
  - UnsharedConcreteFlyweight
- Collaborations

FlyWeight模式(五)

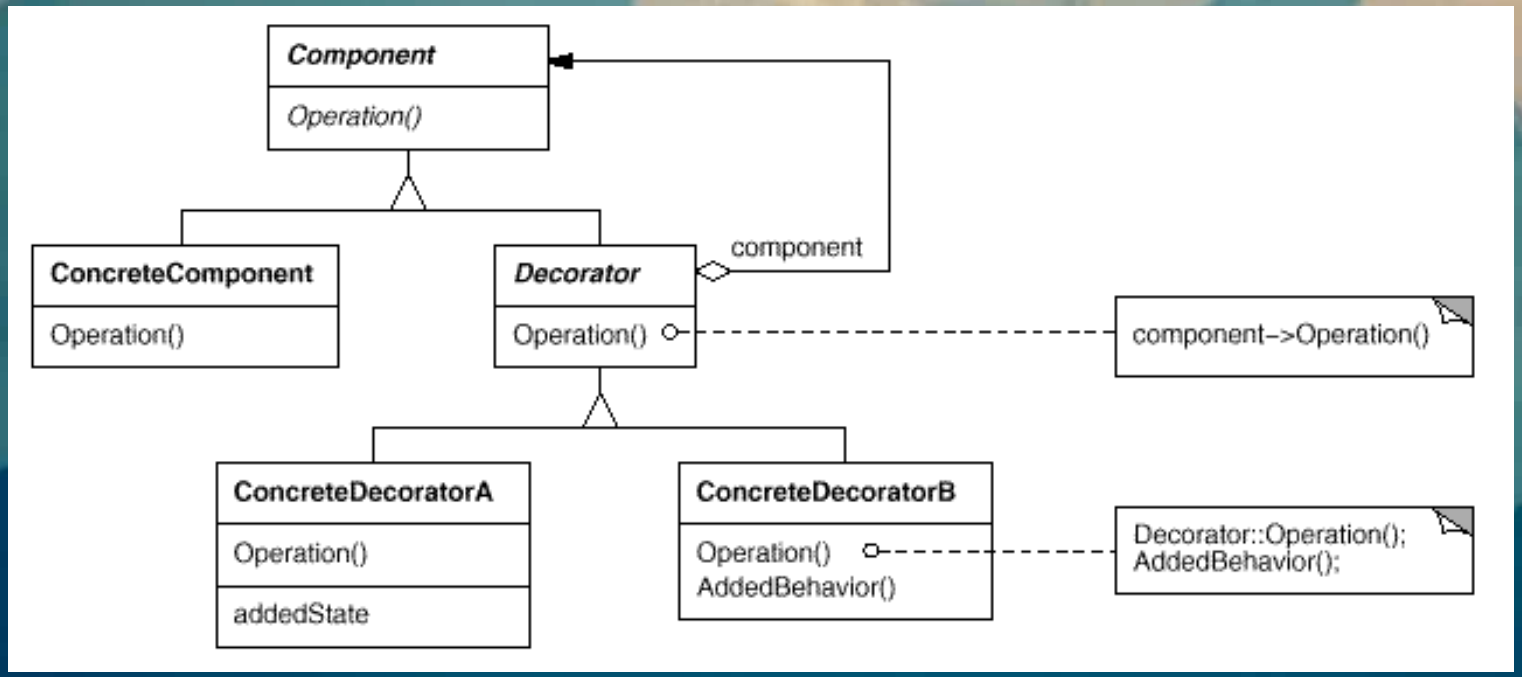
- Evaluation
  - 把对象的状态分开：intrinsic and extrinsic
  - 节约存储空间：内部状态的共享节约了大量空间，外部状态可通过计算获得从而进一步节约空间
  - flyweight与composite结合。Flyweight为leaf节点，并且父节点只能作为外部状态
- Implementation
  - Removing extrinsic state，尽可能做到实时计算(通过一个小的数据结构)
  - Managing shared objects，客户不能直接实例化flyweight，必须通过管理器，例如FlyweightFactory。flyweight的生命周期管理，引用计数和回收

FlyWeight模式(六)

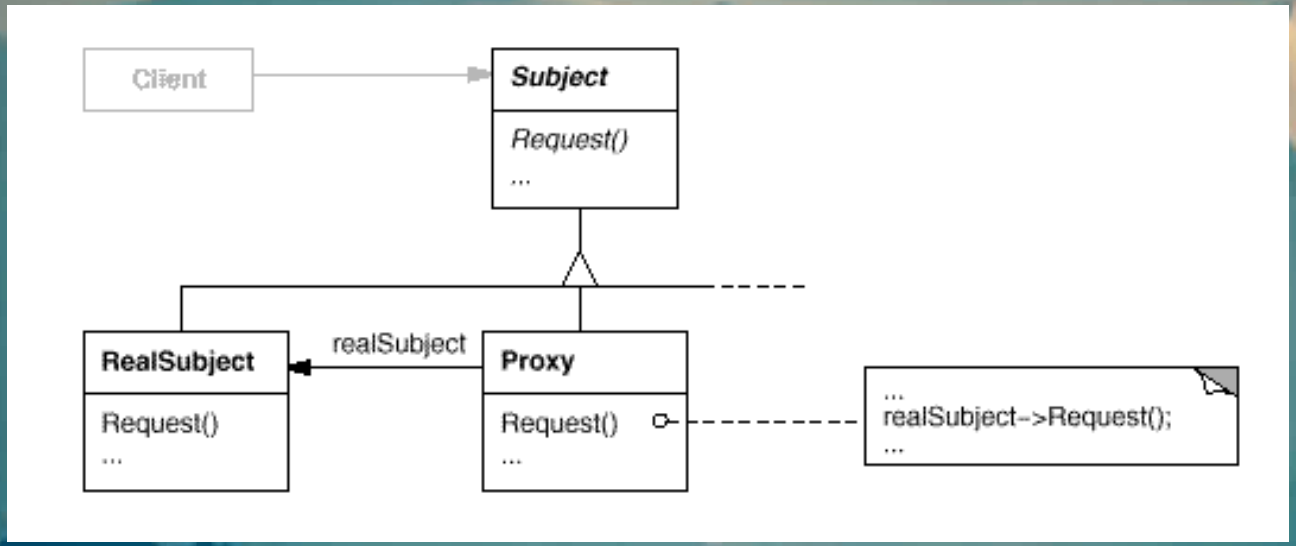
- Related Patterns
  - 与Composite模式组合
  - 可以用flyweight实现State和Strategy模式中的对象
- Examples
  - Excel中cell的管理
    - IOleItemContainer接口允许客户发现每一个cell对象
    - 用flyweight实现cell对象 —— tearoff技术
    - 对状态的有效管理是对象技术的一个进步
  - “Design Patterns”中提到的文档编辑器的例子

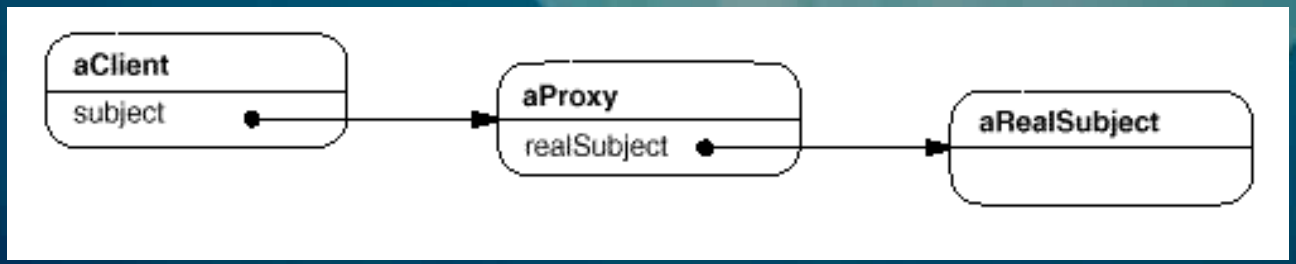


Structural Patterns： Decorator



Structural Patterns： Proxy





Structural patterns小结

- Adapter 、bridge、facade
  - adapter用于两个不兼容接口之间的转接
  - bridge用于将一个抽象与多个可能的实现连接起来
  - facade用于为复杂的子系统定义一个新的简单易用的接口
- composite、decorator和proxy
  - composite用于构造对象组合结构
  - decorator用于为对象增加新的职责
  - proxy为目标对象提供一个替代者
- flyweight
  - 针对细粒度对象的一种全局控制手段

Behavioral Patterns

- *Chain of Responsibility* *
- Command
- *Interpreter* *
- Iterator
- *Mediator* *
- *Memento* *
- Observer
- *State* *
- Strategy
- *Template Method* *
- Visitor

模式 12：Command(一)

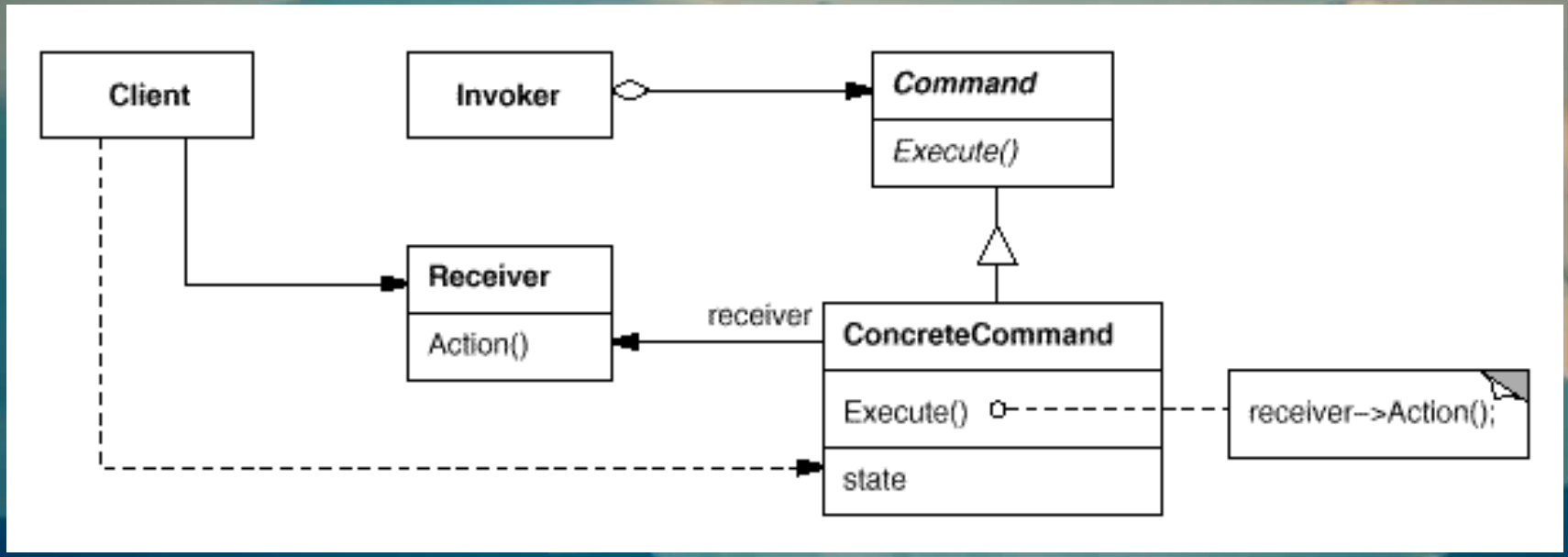
- Aliases
  - Action, Transaction
- Intent
  - Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.
- Motivation
  - 把请求信息和请求执行过程封装起来
  - framework往往需要把命令请求与处理请求的对象分开，command模式可以把调用操作的对象与操作的目标对象分开。
  - 允许通过多种途径调用同一个请求。——请求的重用

Command模式(二)

- Applicability：Use the Command pattern when :
  - parameterize objects by an action to perform，代替回调
  - specify, queue, and execute requests at different times
  - support undo
  - support logging changes so that they can be reapplied in case of a system crash
  - structure a system around high-level operations built on primitives operations —— transactions

Command模式(三)

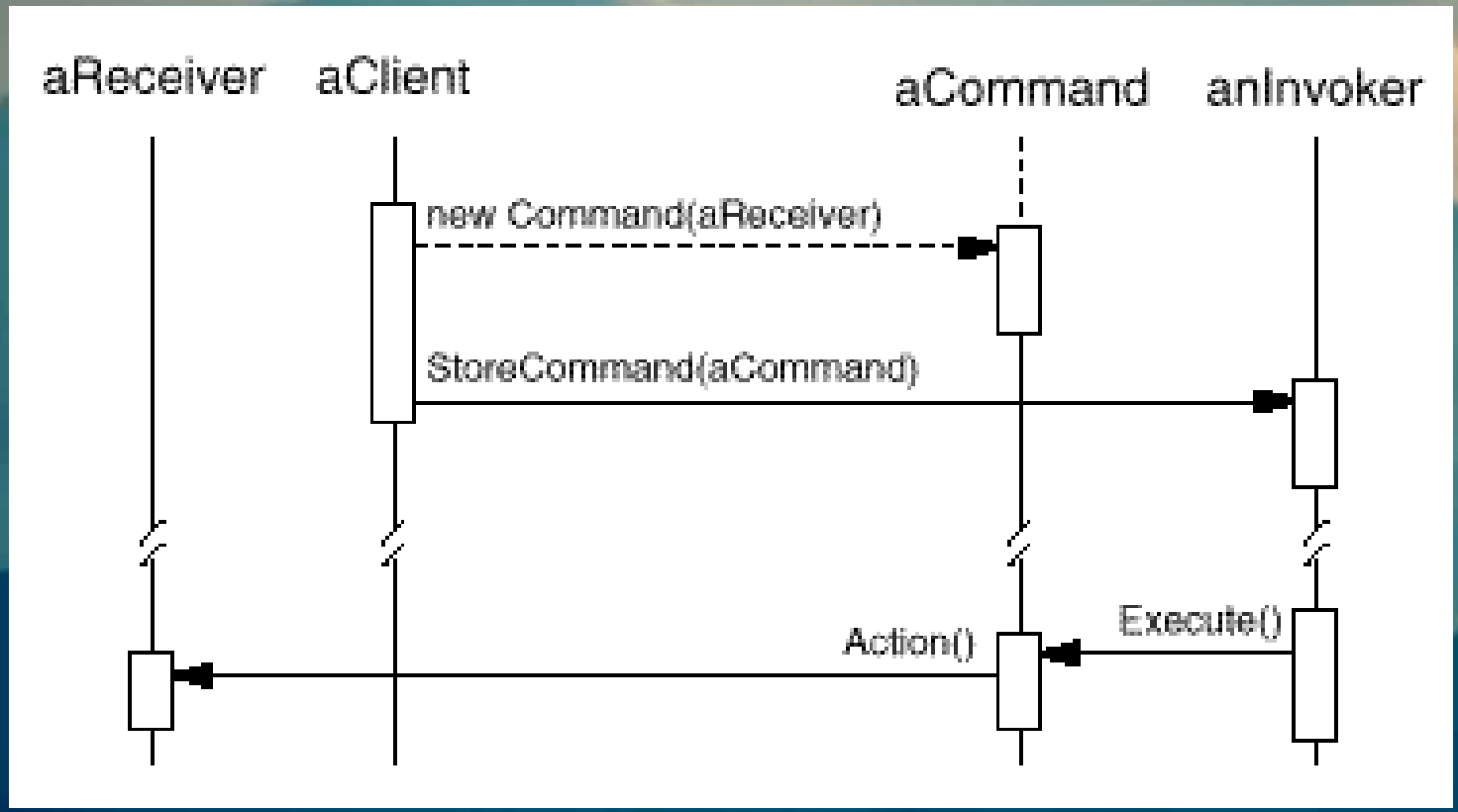
- Struct



- Participants
  - Client, Command、ConcreteCommand、Invoker、Receiver

Command模式(四)

- Collaborations

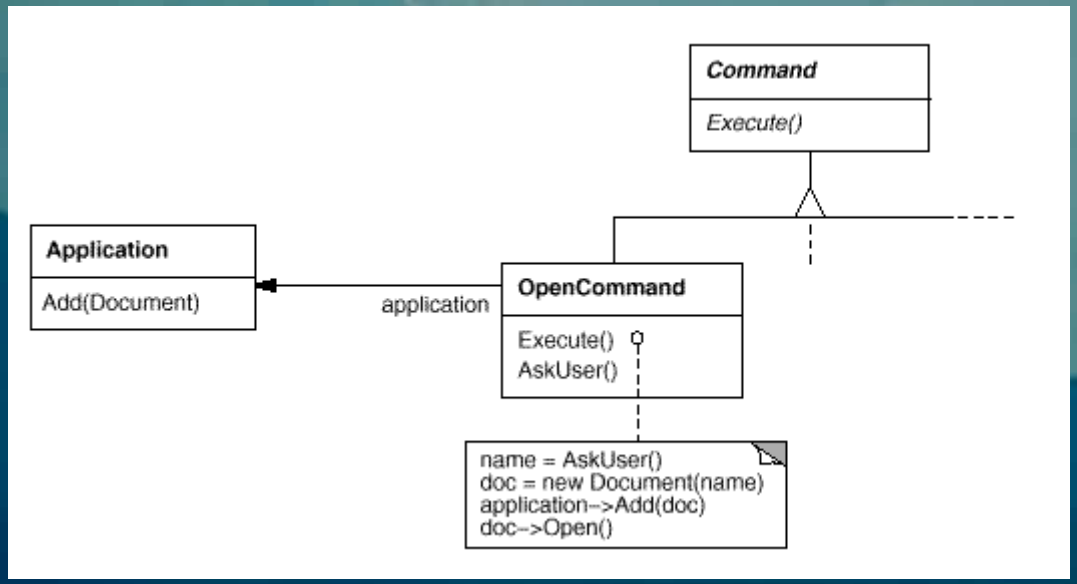


Command模式(五)

- Evaluation
  - Command decouples the object that invokes the operation from the one that knows how to perform it.
  - Commands are first-class objects. They can be manipulated and extended like any other object.
  - You can assemble commands into a composite command. An example is MacroCommand.
  - It's easy to add new Commands, because you don't have to change existing classes.
- Implementation
  - How intelligent should a command be?
  - Supporting undo and redo
  - Avoiding error accumulation in the undo process
  - Using C++ templates

Command模式(六)

- Related Patterns
  - Composite模式可用来实现command组合
  - 为实现undo/redo，可以用其他行为模式来管理状态，如memento模式。Command被放到history list之前，可以用prototype模式复制自身
- Examples



模式 13：Iterator(一)

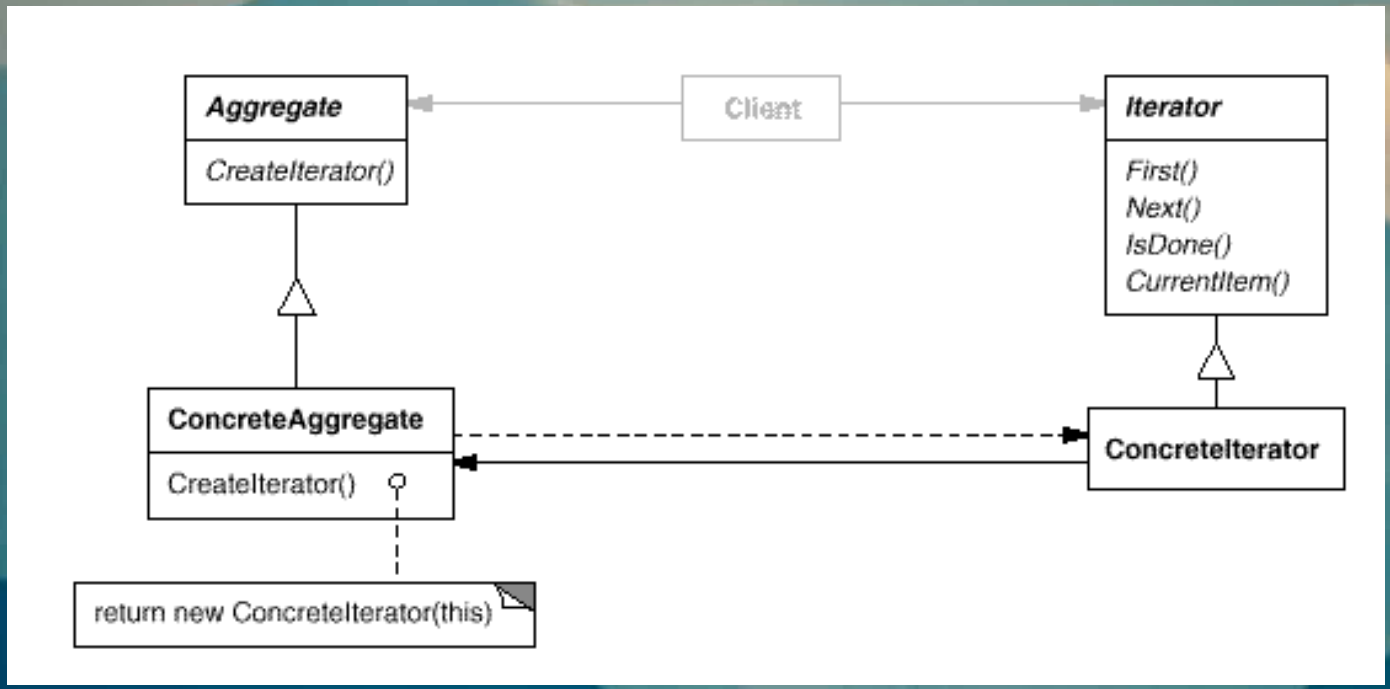
- Aliases： Cursor
- Intent
  - Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.
- Motivation
  - An aggregate object such as a list should give you a way to access its elements without exposing its internal structure.
  - Separating the traversal mechanism from the List object lets us define iterators for different traversal policies without enumerating them in the List interface.

Iterator模式(二)

- Applicability： Use the Iterator pattern when :
  - to access an aggregate object's contents without exposing its internal representation.
  - to support multiple traversals of aggregate objects.
  - to provide a uniform interface for traversing different aggregate structures (that is, to support polymorphic iteration).

Iterator模式(三)

- Struct



- Participants
  - Iterator、ConcreteIterator、Aggregate、ConcreteAggregate
- Collaborations

Iterator模式(四)

- Evaluation
  - It supports variations in the traversal of an aggregate
  - Iterators simplify the Aggregate interface
  - More than one traversal can be pending on an aggregate
- Implementation
  - 实现可以非常灵活
  - Who controls the iteration?
    - external iterator *versus* internal iterator
  - Who defines the traversal algorithm?
    - Aggregate本身定义算法 —— Cursor mode
    - iterator定义算法 —— iterator如何访问数据
  - How robust is the iterator?

Iterator模式(五)

- Implementation(续)
  - Additional Iterator operations.
    - 基本操作：First, Next, IsDone, and CurrentItem
  - Using polymorphic iterators —— iterator资源释放
  - Iterators may have privileged access
  - Iterators for composites —— 适合于internal iterator或者cursor方式的iterator
  - Null iterators

Iterator模式(六)

- Related Patterns
  - Composite：iterator常被用于composite模式的复合结构
  - Polymorphic iterators rely on factory methods to instantiate the appropriate Iterator subclass.

- Examples
  - COM enumerator：connectable object、…
  - C++ STL
    - 在STL中，iterator是连接algorithm和container的桥梁

模式 14：Observer(一)

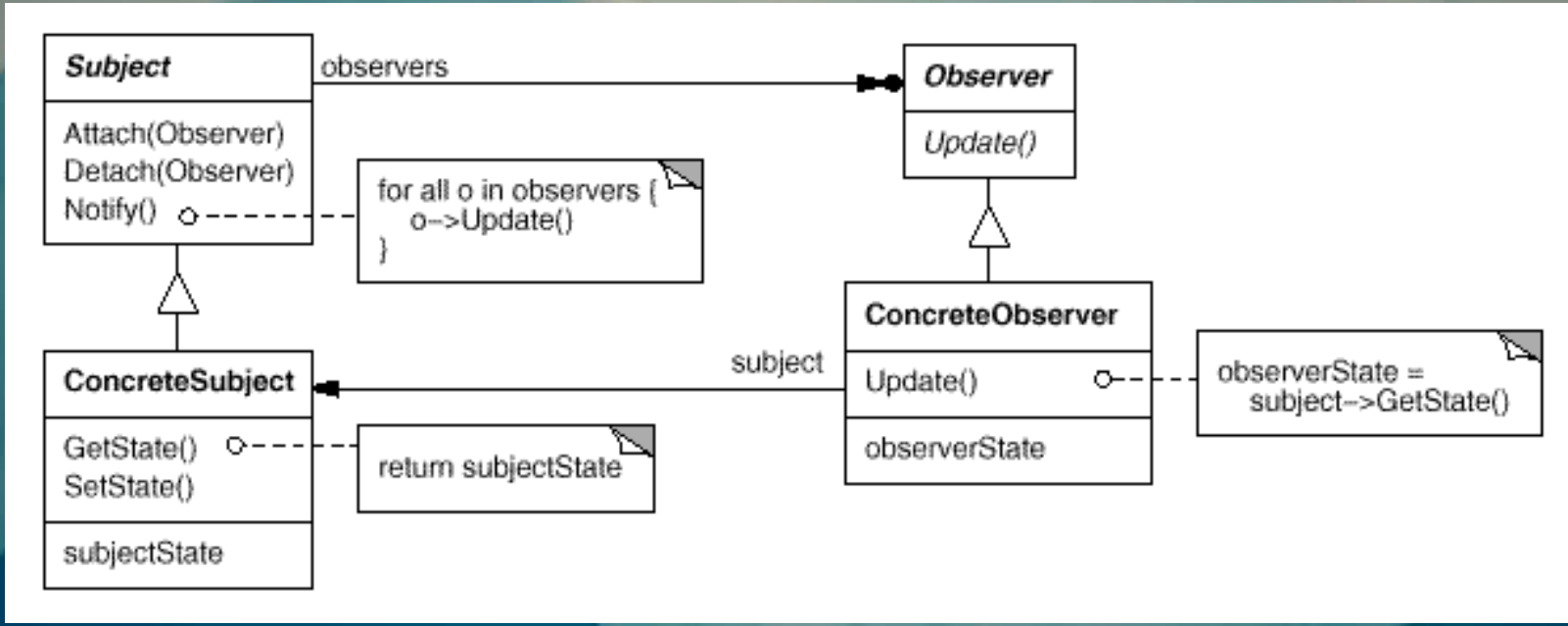
- Aliases： Dependents, Publish-Subscribe
- Intent
  - Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.
- Motivation
  - 把系统分成一些相互关联的类或者对象，如何维护这些类的实例一致性？
  - The key objects in this pattern are subject and observer
    - One-to-many relationship
    - A subject may have any number of dependent observers.
    - All observers are notified whenever the subject undergoes a change in state.

Observer模式(二)

- Applicability：Use the Observer pattern when :
  - When an abstraction has two aspects, one dependent on the other. Encapsulating these aspects in separate objects lets you vary and reuse them independently.
  - When a change to one object requires changing others, and you don't know how many objects need to be changed.
  - When an object should be able to notify other objects without making assumptions about who these objects are. In other words, you don't want these objects tightly coupled.

Observer模式(三)

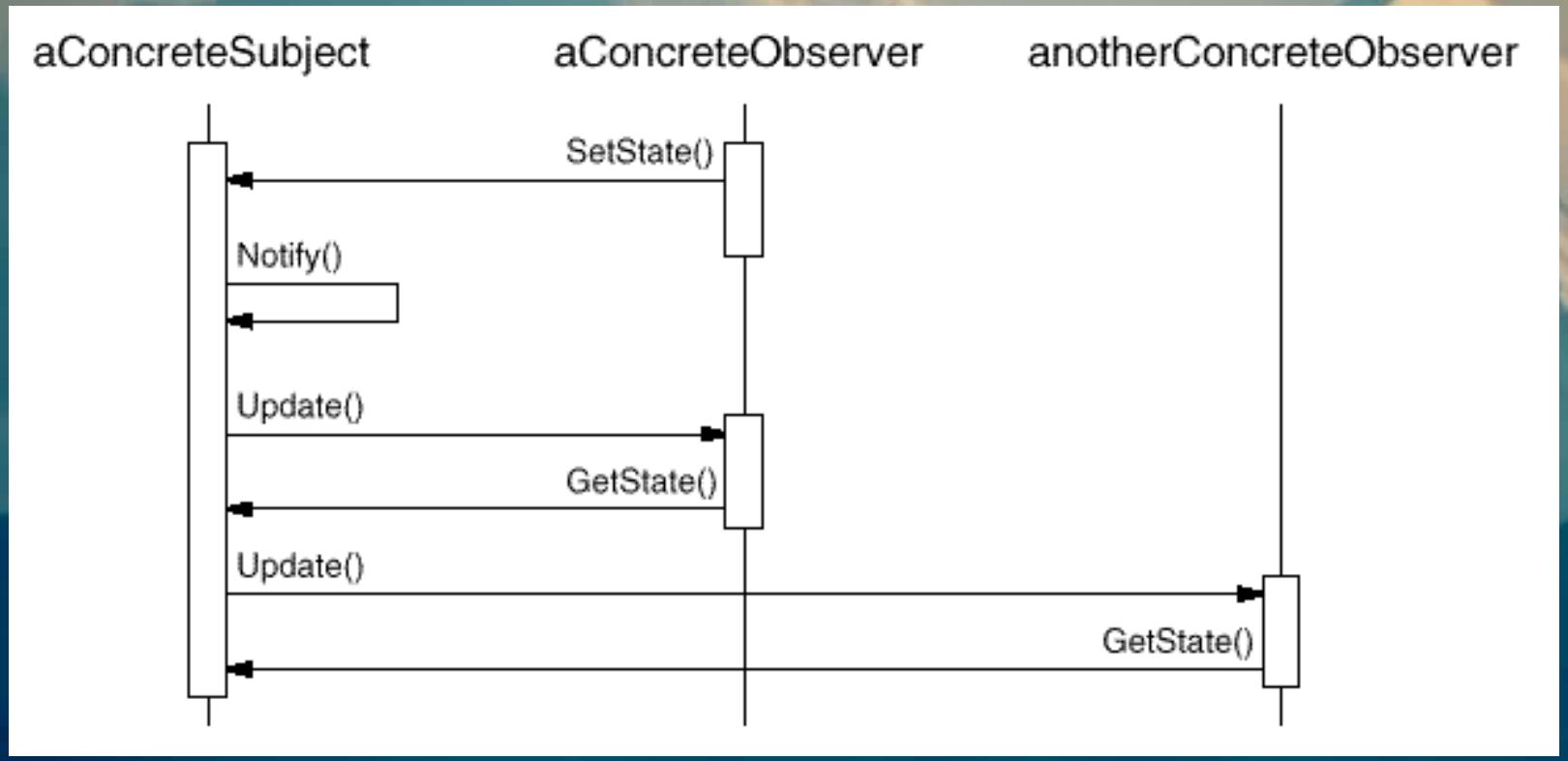
- Struct



- Participants
  - Subject、ConcreteSubject、Observer、ConcreteObserver

Observer模式(四)

- Collaborations

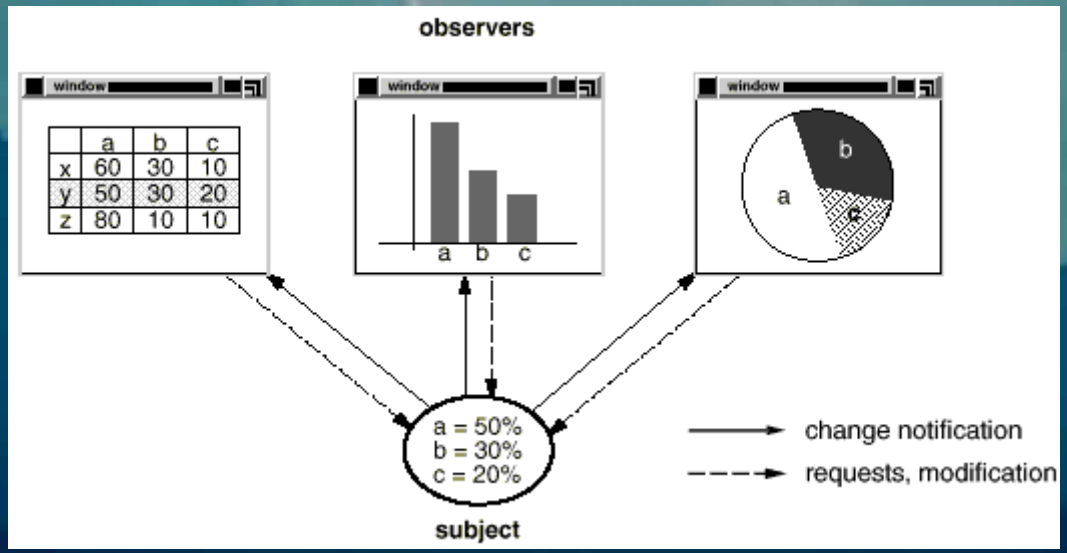


Observer模式(五)

- Evaluation
  - Abstract coupling between Subject and Observer
  - Support for broadcast communication
  - Unexpected updates
- Implementation
  - Mapping subjects to their observers.
  - Observing more than one subject
  - Who triggers the update? Client or subject?
  - Making sure Subject state is self-consistent before notification
  - subject向observer传递变化信息
  - 中间插入ChangeManager

Observer模式(六)

- Related Patterns
  - Mediator：用Mediator模式封装复杂的更新语义
- Examples
  - COM property page
  - COM+ Event Model，
  - MVC



模式 15：Strategy(一)

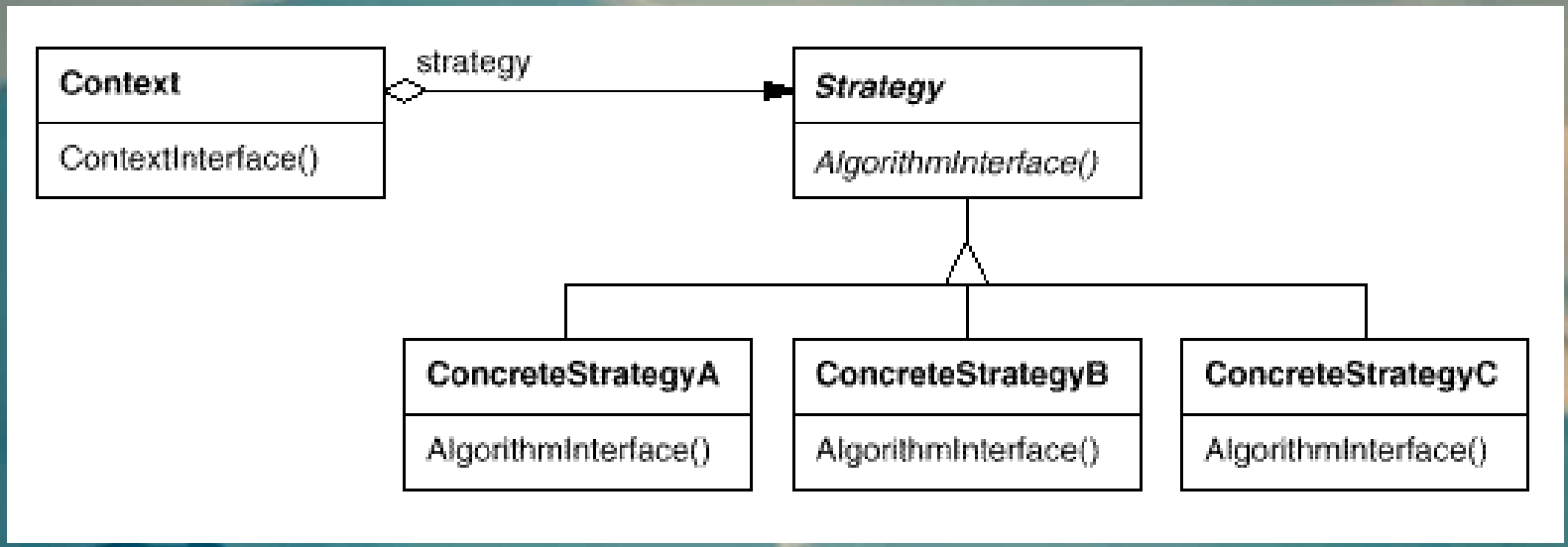
- Aliases ： Policy
- Intent
  - Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.
- Motivation
  - 有些算法对于某些类是必不可少的，但是不适合于硬编进类中。客户可能需要算法的多种不同实现，允许增加新的算法实现或者改变现有的算法实现
  - 我们可以把这样的算法封装到单独的类中，称为strategy

Strategy模式(二)

- Applicability：Use the Strategy pattern when :
  - many related classes differ only in their behavior.
  - you need different variants of an algorithm.
  - an algorithm uses data that clients shouldn't know about.
  - a class defines many behaviors, and these appear as multiple conditional statements in its operations.

Strategy模式(三)

- Struct



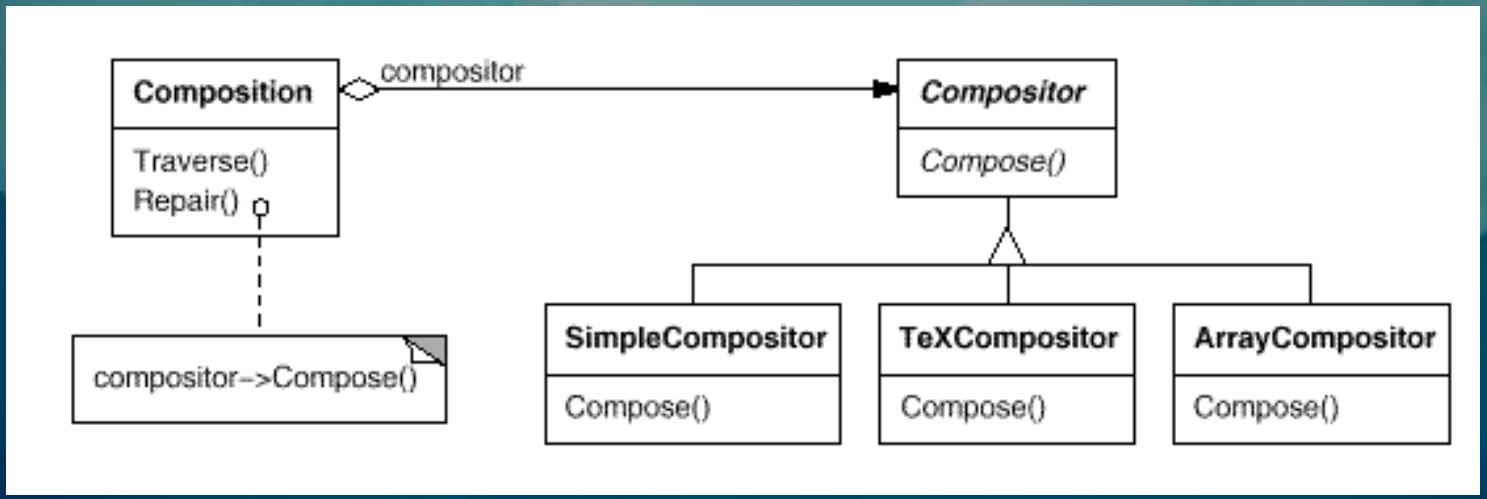
- Participants
  - Strategy、ConcreteStrategy、Context
- Collaborations
  - Strategy and Context interact to implement the chosen algorithm
  - A context forwards requests from its clients to its strategy

Strategy模式(四)

- Evaluation
  - Families of related algorithms
  - An alternative to subclassing
  - Strategies eliminate conditional statements
  - Clients must be aware of different Strategies
  - Communication overhead between Strategy and Context
  - Increased number of objects
- Implementation
  - Defining the Strategy and Context interfaces
  - Strategies as template parameters
  - Making Strategy objects optional

Strategy模式(五)

- Related Patterns
  - flyweight：考虑用flyweight模式来实现strategy对象
- Examples
  - ATL中COM对象的线程模型支持

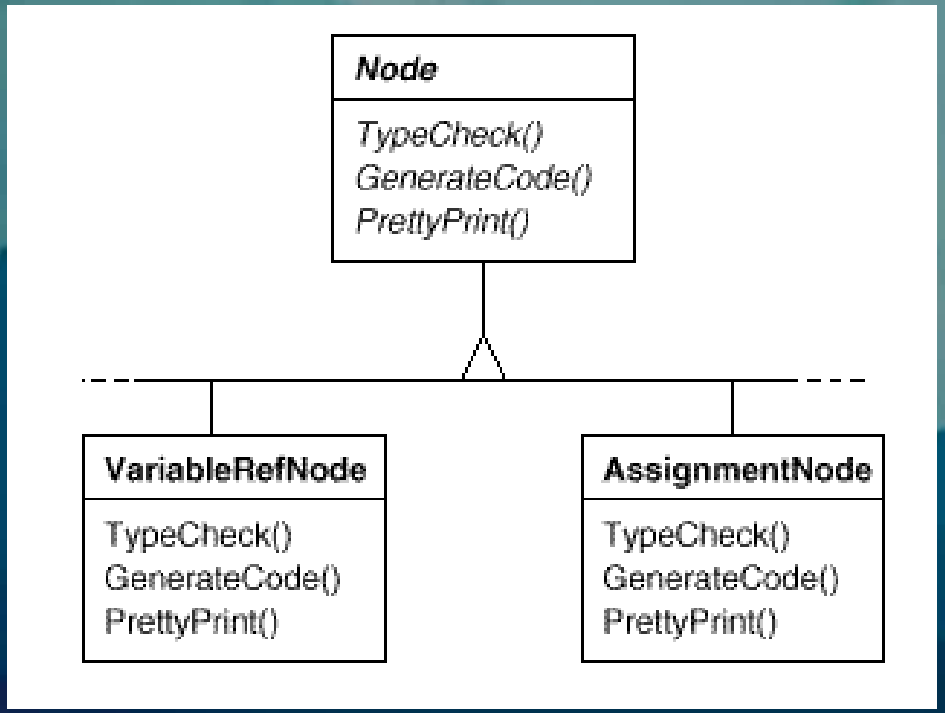


模式 16：Visitor(一)

- Intent
  - 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.
- Motivation
  - 为了把一个操作作用于一个对象结构中，一种做法是把这个操作分散到每一个节点上。导致系统难以理解、维护和修改
  - 把这样的操作包装到一个独立的对象(visitor)中。然后在遍历过程中把此对象传递给被访问的元素。

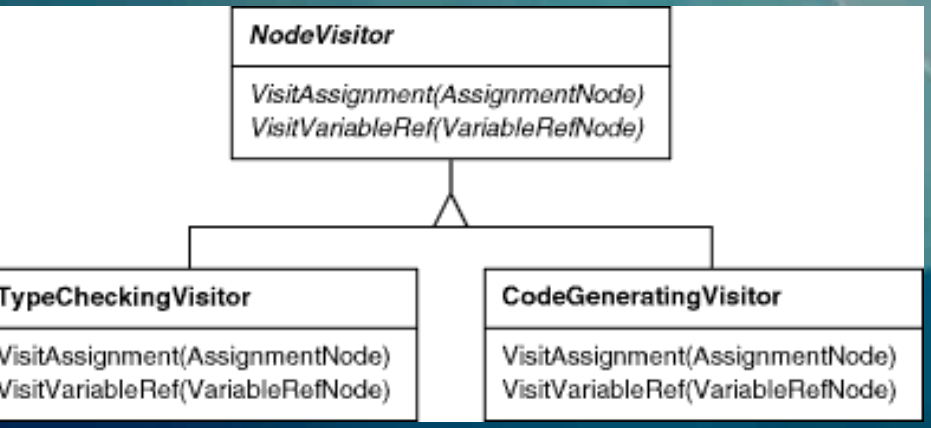
不用visitor的compiler例子

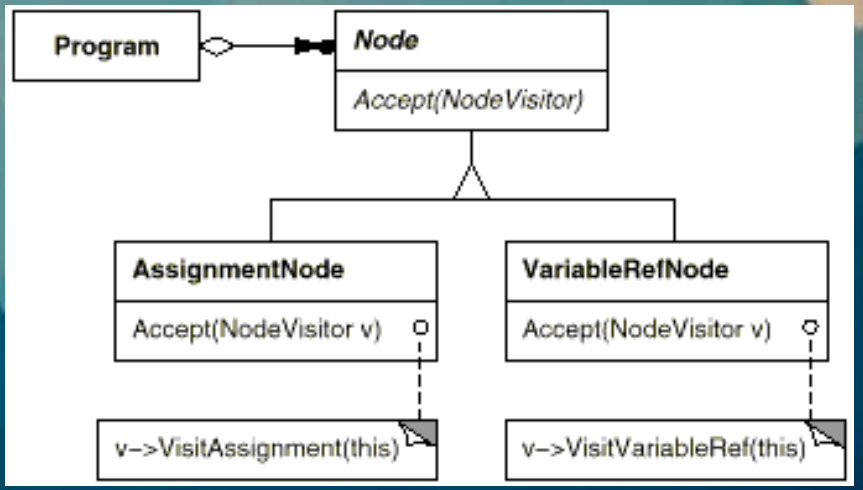
Class\operation	TypeCheck	GenerateCode	PrettyPrint
VariableRefNode			
AssignmentNode			
...			



使用visitor的compiler例子

class\class	VariableRefNode	AssignmentNode
TypeCheckVisitor	VisitVariableRef	VisitAssignment
GenerateCodeVisitor	VisitVariableRef	VisitAssignment
PrettyPrintVisitor	VisitVariableRef	VisitAssignment



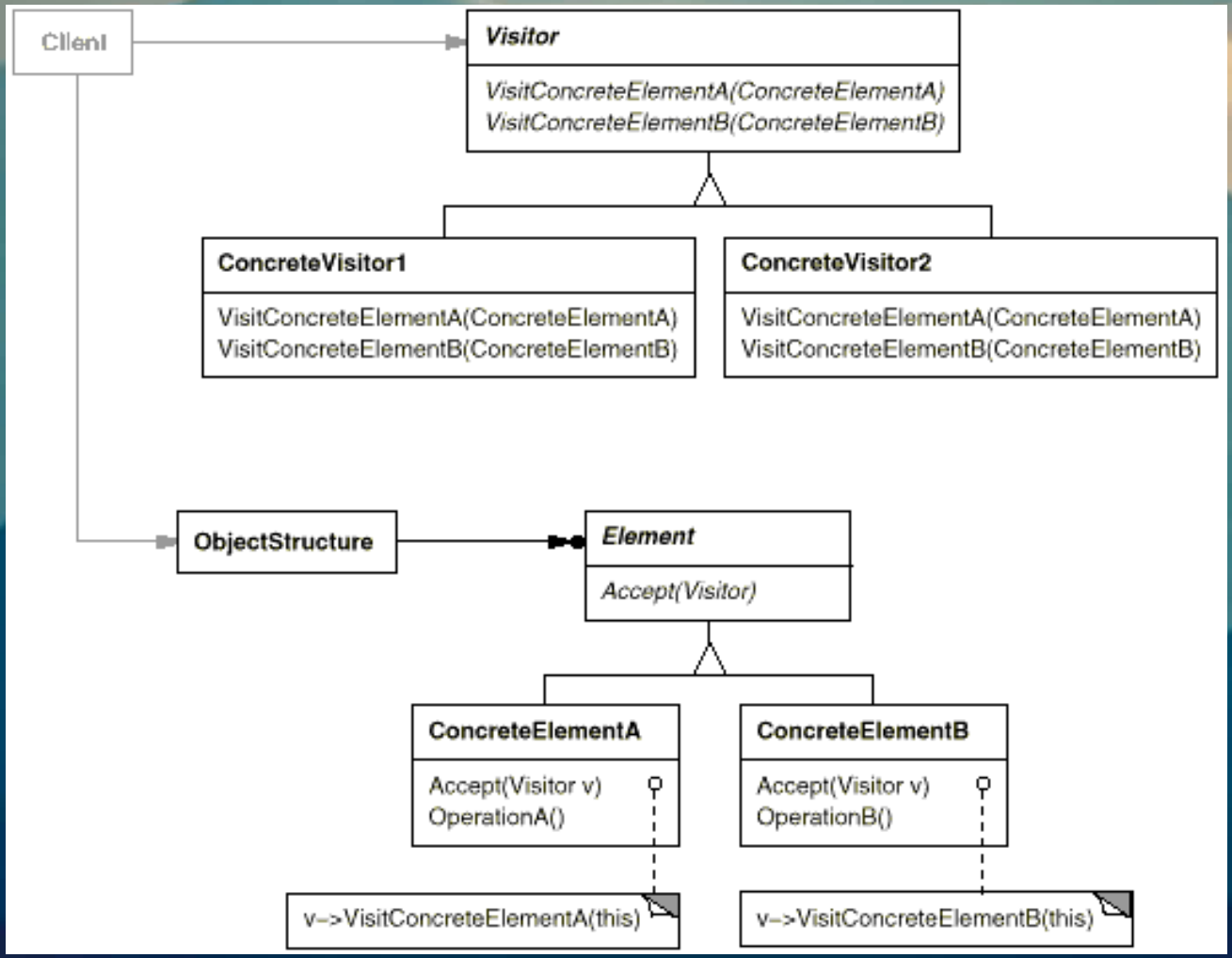


Visitor模式(二)

- Applicability：Use the Visitor pattern when
  - 一个对象结构包含许多对象类，我们想执行一些依赖于具体类的操作
  - 要对一个对象结构中的对象进行很多不同的并且不相关的操作，又不想改变这些对象类
  - 定义对象结构的类很少改变，但是经常要在此结构上定义新的操作。改变对象结构类，需要重定义所有visitor的接口

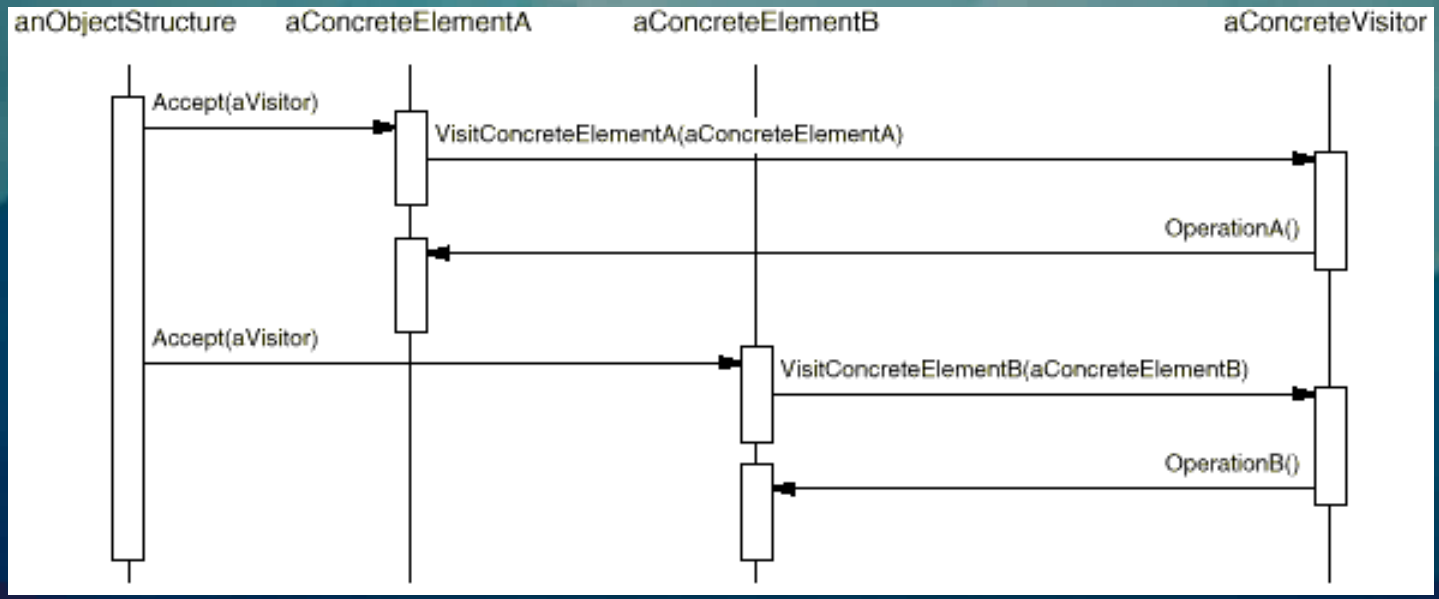
Visitor模式(三)

- Struct



Visitor模式(四)

- Participants
  - Client、Visitor、ConcreteVisitor、ObjectStructure、Elemen
    ConcreteElement
- Collaborations
  - client先创建一个ConcreteVisitor，然后遍历ObjectStructure



Visitor模式(五)

- Evaluation
  - Visitor makes adding new operations easy
  - A visitor gathers related operations and separates unrelated ones
  - Adding new ConcreteElement classes is hard.
  - 即使不是类层次，visitor也可以实施
  - 状态累积
  - visitor要访问每个元素的状态，所以要打破封装
- Implementation
  - double-dispatch，Accept实现了double dispatch
    This is the key to the Visitor pattern: The operation that gets executed depends on both the type of Visitor and the type of Element it visits.
  - Who is responsible for traversing the object structure?

Visitor模式(六)

- Related Patterns
  - Composite：visitor常常被用于composite模式组成的结构中
- Examples
  - 编译器实现

其他Behavioral Patterns

- Chain of Responsibility
  - 请求的处理过程，沿着链传递，decouple发送和接收方
- Interpreter
  - 在类层次结构中，在特定环境的“interpret”过程
- Mediator
  - 用一个mediator来decouple各同等单元
- Memento
  - 在对象之外保存对象的内部状态
- State
  - 把一个对象的状态独立出来，动态可变换状态对象的类型
- Template Method
  - 在基类中定义算法的骨架，把某些细节延迟到子类中

Behavioral Patterns小结

- Strategy、Iterator、Mediator、State、command
  - 用一个对象来封装某些特性，比如变化、交互、状态、行为、命令
- Mediator、Observer
  - Observer建立起subject和observer之间的松耦合连接
  - mediator把约束限制集中起来，-〉中心控制
- command、Chain of Responsibility、interpreter
  - command模式侧重于命令的总体管理
  - Chain of Responsibility侧重于命令被正确处理
  - interpreter用于复合结构中操作的执行过程

第二次作业

写出你认为理解最深刻的一个pattern，
可以是你自己学习、工作中体会出来的，
也可以是书上已经总结好的。
请使用给定的pattern格式书写作业！