结构型设计模式
Structural Patterns

吴映波
wyb@cqu.edu.cn
虎溪Office: 虎溪学院楼410
A区Office：九教205
Tel：13594686661

Structural Patterns

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

模式 7： Adapter (一)

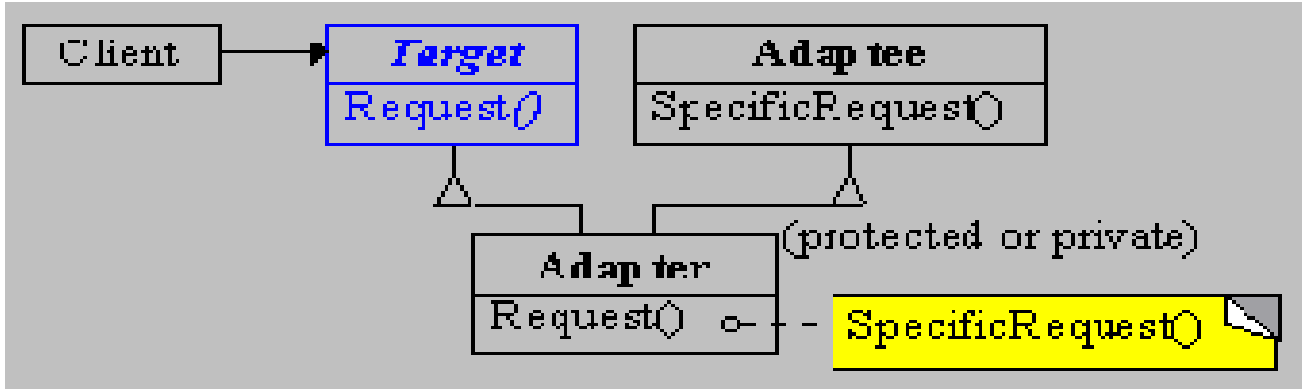
- Aliases：Wrapper
- Intent
  - Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.
- Motivation
  - Sometimes a toolkit class that's designed for reuse isn't reusable only because its interface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

Adapter模式(二)

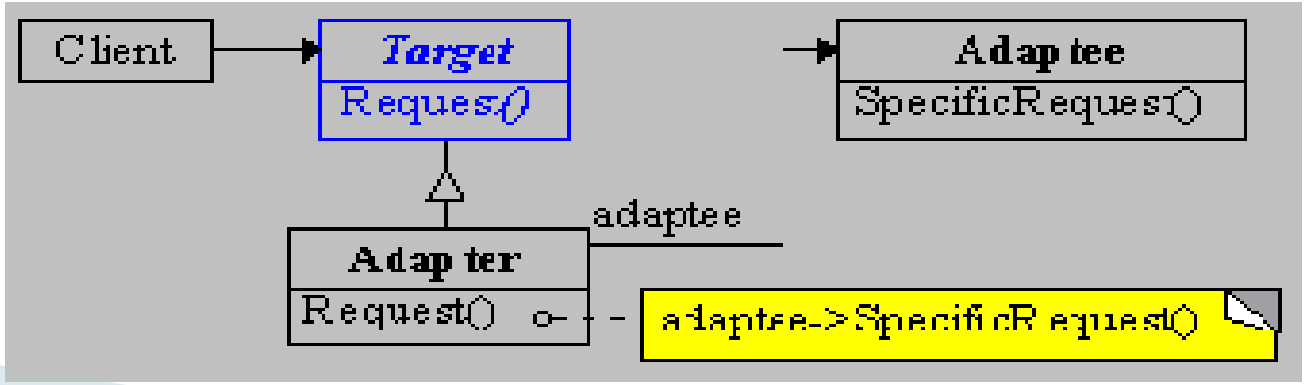
- Applicability：Use the Adapter pattern when
  - you want to use an existing class, and its interface does not match the one you need.
  - you want to create a reusable class that cooperates with unrelated or unforeseen classes, that is, classes that don't necessarily have compatible interfaces.
  - (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

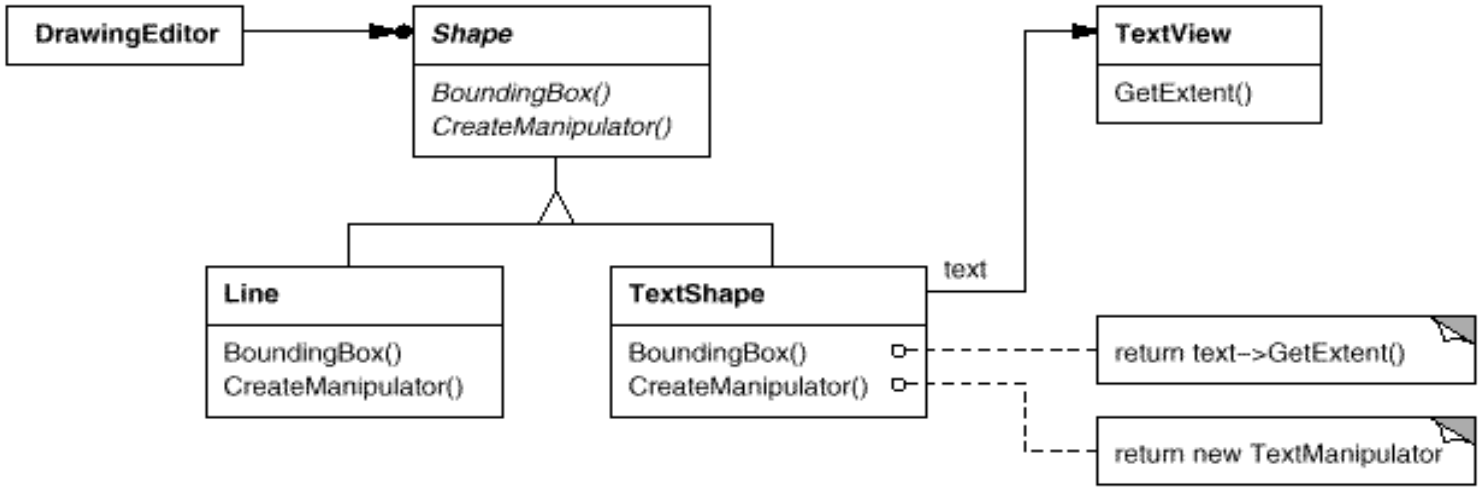
Adapter模式(五)

- 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



模式 8：Bridge(一)

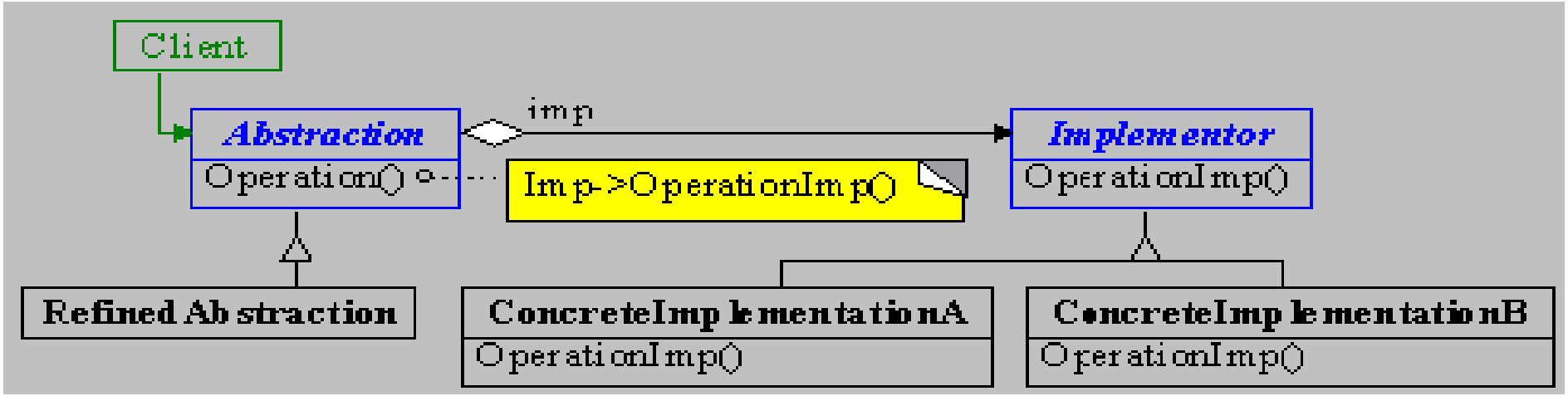
- Aliases：Handle/Body
- Intent
  - Decouple an abstraction from its implementation so that the two can vary independently
- Motivation
  - 要做到“**抽象**(接口)与实现分离”，最常用的办法是定义一个抽象类，然后在子类中提供实现。也就是说，用继承机制达到“**抽象**(接口)与实现分离”
  - 但是这种方法不够灵活，继承机制把实现与抽象部分永久地绑定起来，要想独立地修改、扩展、重用抽象(接口)与实现都非常困难。

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

Bridge模式(四)

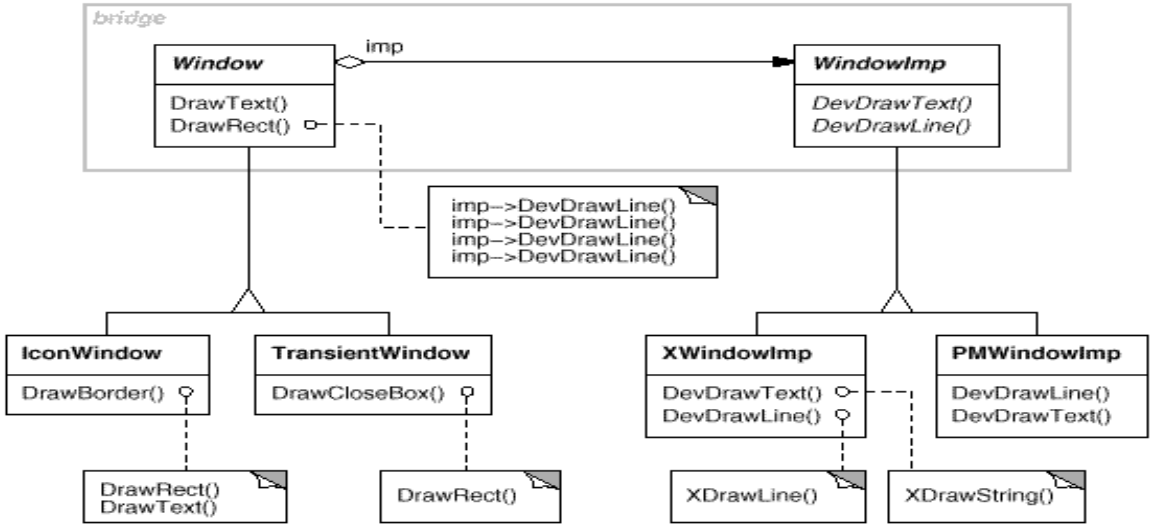
- Evaluation
  - 抽象部分与实现部分的分离，可以在运行时刻连接起来
    二进制兼容性
  - 提高可扩充性：抽象与实现两部分可以单独扩充
  - 对客户隐藏实现细节

Bridge模式(五)

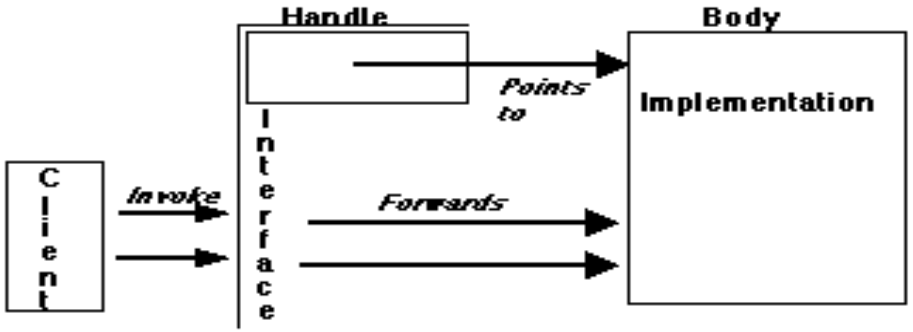
- Implementation
  - Only one Implementor
  - Creating the right Implementor object
    如何创建？根据客户环境，或者通过factory
  - Sharing implementors
    - 资源管理：引用计数技术
  - Using multiple inheritance

Bridge模式(六)

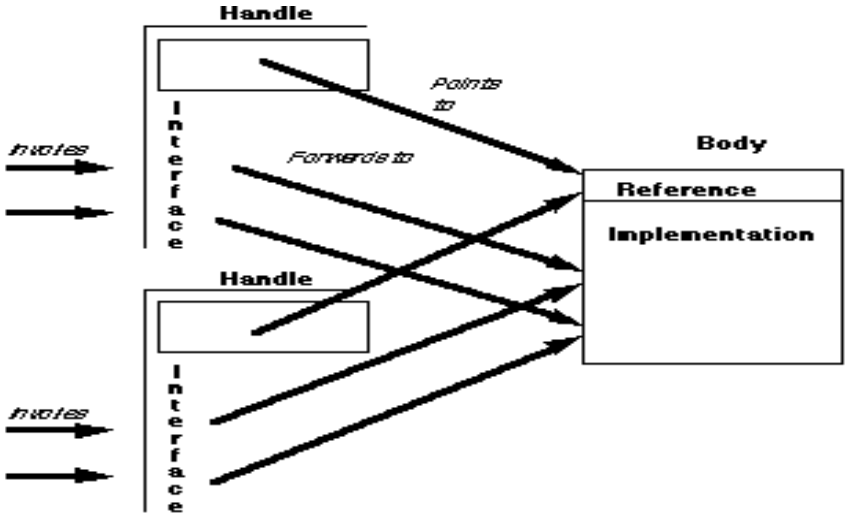
- Related Patterns
  - Abstract Factory可以用来创建和配置Bridge模式
  - 与Adapter模式的区别
- Examples
  - handle：文件handle、窗口handle



插：Handle/Body



Counted Handle/Body

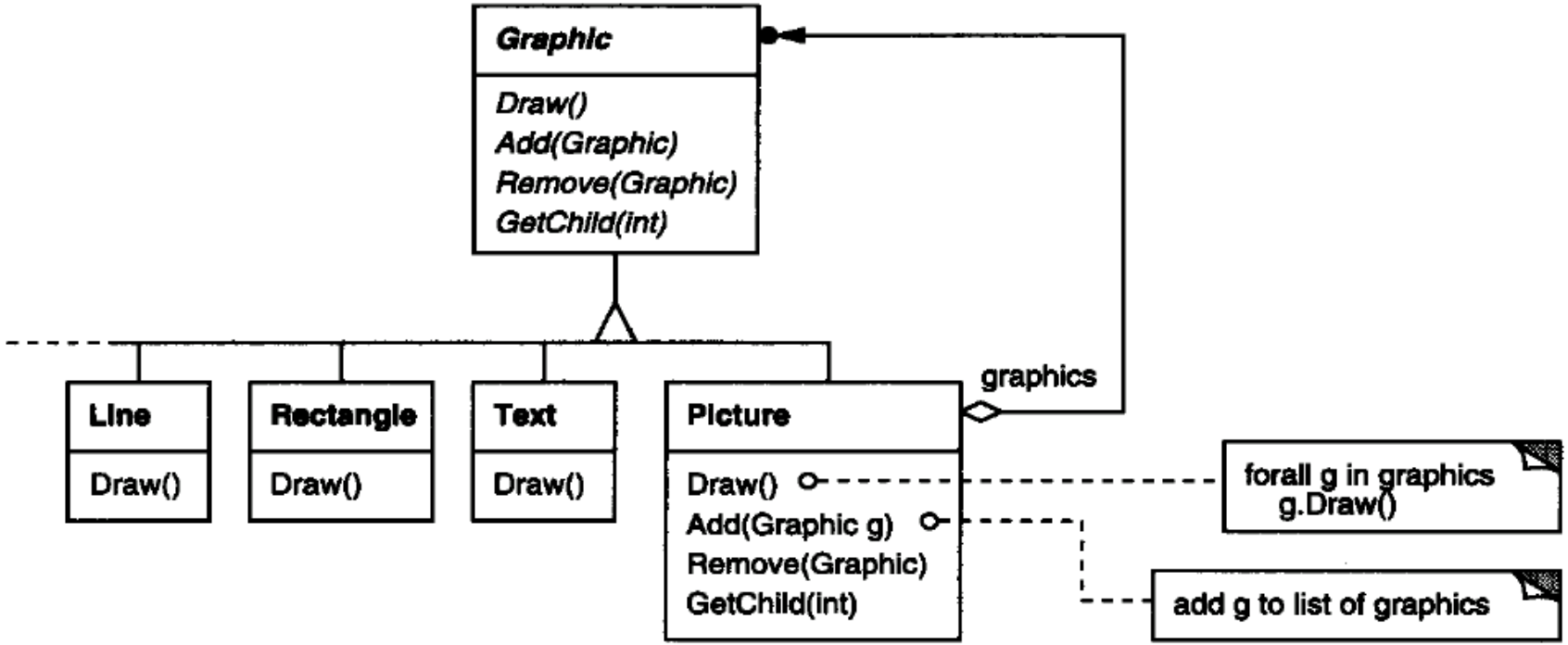


模式 9：Composite(一)

- 在绘图编辑器和图形捕捉系统这样的图形应用程序中，用户可以使用简单的图元创建复杂的图表。用户可以组合多个简单图元以形成一些较大的图表，这些图表又可以组合成更大的图表。一个简单的实现方法是为Text和Line这样的图元定义一些类，另外定义一些类作为这些图元的容器类(Container)。
- 然而这种方法存在一个问题：使用这些类的代码必须区别对待图元对象与容器对象，而实际上大多数情况下用户认为它们是一样的。对这些类区别使用，使得程序更加复杂。

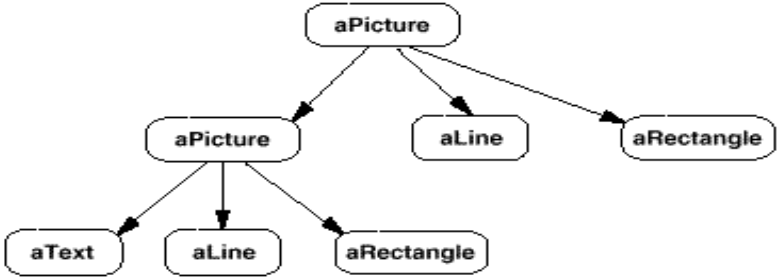
模式 9：Composite(一续)

- Composite模式描述了如何使用递归组合，使得用户不必对这些类进行区别，如下图所示。



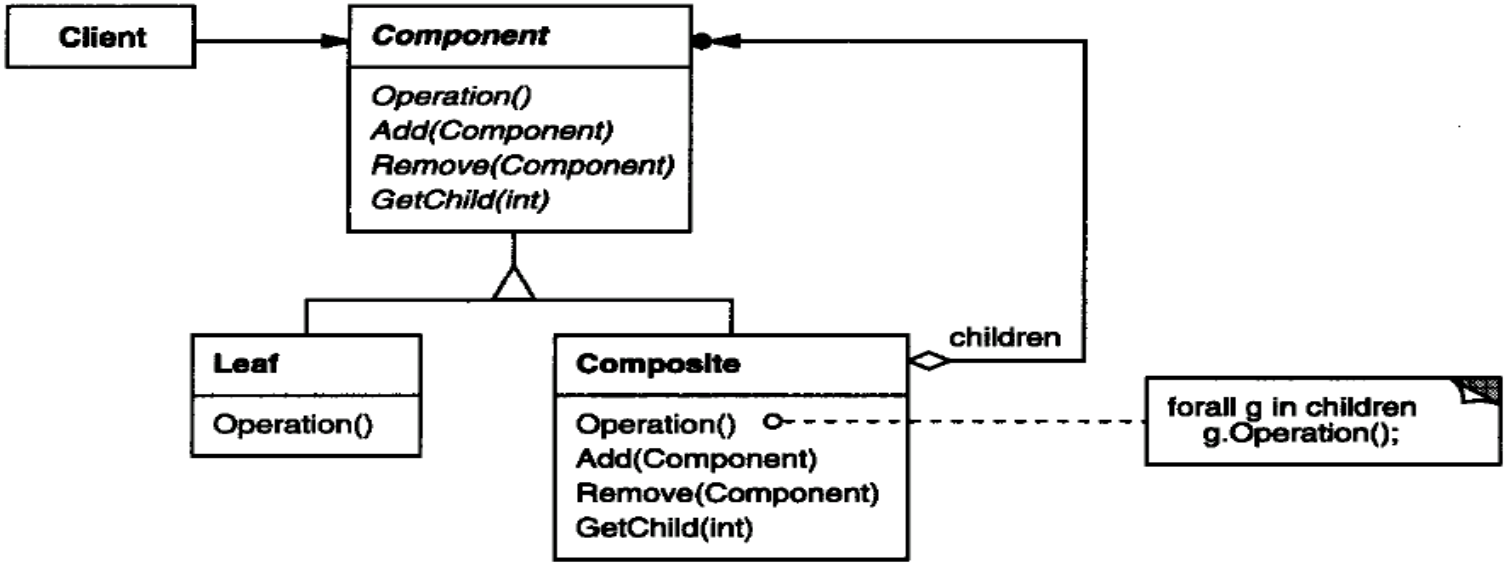
模式 9：Composite(一)

- 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(三)

- Struct



- Participants
  - Client, Component, Leaf, Composite

Composite模式(二)

- Applicability：Use the Composite pattern when
  - you want to represent part-whole hierarchies of objects.你想表示对象的部分-整体层次结构。
  - 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.你希望用户忽略组合对象与单个对象的不同，用户将统一地使用组合结构中的所有对象。

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。容易增加新组件也会产生一些问题，那就是很难限制组合中的组件。有时你希望一个组合只能有某些特定的组件。使用Composite时，你不能依赖类型系统施加这些约束，而必须在运行时刻进行检查。

Composite模式(五)

- Implementation
  - **Explicit parent references**：保持从子部件到父部件的引用能简化组合结构的遍历和管理。父部件引用可以简化结构的上移和组件的删除。通常在Component类中定义父部件引用。Leaf和Composite类可以继承这个引用以及管理这个引用的那些操作。
  - **Maximizing the Component interface**：Composite模式的目的之一是使得用户不知道他们正在使用的具体的Leaf和Composite类。为了达到这一目的，Composite类应为Leaf和Composite类尽可能多定义一些公共操作。Composite类通常为这些操作提供缺省的实现，而Leaf和Composite子类可以对它们进行重定义。

Composite模式(五)

- Implementation（续）
  - Declaring the child management operations(声明管理子部件的操作)
    - 虽然Composite类实现了Add和Remove操作用于管理子部件，但在Composite模式中一个重要的问题是：在Composite类层次结构中哪一些类声明这些操作。
    - 我们是应该在Component中声明这些操作，并使这些操作对Leaf类有意义呢，还是只应该在Composite和它的子类中声明并定义这些操作呢？

Composite模式(五)

- Declaring the child management operations（续）

这需要在安全性和透明性之间做出权衡选择。

  - 在类层次结构的根部定义子节点管理接口的方法具有良好的透明性，因为你可以一致地使用所有的组件，但是这一方法是以安全性为代价的，因为客户有可能会做一些无意义的事情，例如在Leaf中增加和删除对象等。
  - 在Composite类中定义管理子部件的方法具有良好的安全性，因为在象C++这样的静态类型语言中，在编译时任何从Leaf中增加或删除对象的尝试都将被发现。但是这又损失了透明性，因为Leaf和Composite具有不同的接口。

在这一模式中，相对于安全性，我们比较强调透明性。如果你选择了安全性，有时你可能会丢失类型信息，并且不得不将一个组件转换成一个组合。这样的类型转换必定不是类型安全的。

Composite模式(五)

- Declaring the child management operations（续）

一种办法是在Component类中声明一个操作：

Composite* GetComposite()

Component提供了一个返回空指针的缺省操作。

Composite类重新定义这个操作并通过this指针返回它自身。

```
class Component{
public:
    //...
    virtual Composite* GetComposite(){return 0;}
    //...
}
class Composite:public Component{
public:
    void Add(Component*);
    //...
    virtual Composite* GetComposite(){return this;)
};

class Leaf:public Component{
    //...
};
```

Composite模式(五)

- Declaring the child management operations（续）

GetComposite允许你查询一个组件看它是否是一个组合，你可以对返回的组合安全地执行Add和Remove操作。

```
Composite* aComposite=new Composite;
Leaf* aLeaf = new Leaf;

Component* aComponent;
Composite* test;

aComponent = aComposite;
if (test = aComponent->GetComposite()){
    test--> Add(new Leaf);

aComponent = aLeaf
if (test = aComponent->GetComposite()){
test-->Add(new Leaf);//will not add leaf
```

Composite模式(五)

- Child ordering：
  - 在许多设计中可能需要指定Composite的子部件顺序。在前面Graphics例子中，排序可能表示了从前至后的顺序。如果Composite表示语法分析树，Composite子部件的顺序必须反映程序结构，而组合语句就是这样一些Composite的实例。
  - 如果需要考虑子节点的顺序时，必须仔细地设计对子节点的访问和管理接口，以便管理子节点序列。Iterator模式可以在这方面给予一定的指导。

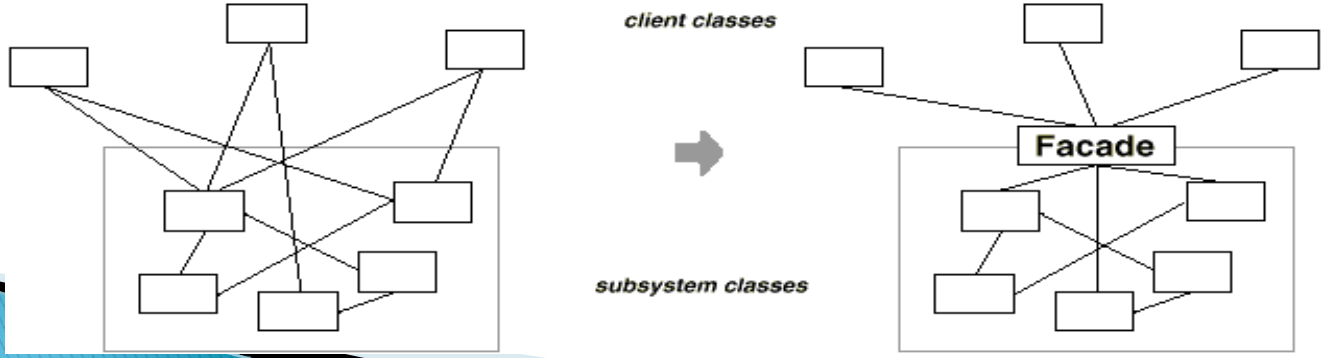
Composite模式(五)

- **What‘s the best data structure for storing components?(存贮组件最好用哪一种数据结构？)**

Composite可使用多种数据结构存贮它们的子节点，包括连接列表、树、数组和hash表。数据结构的选择取决于效率。

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

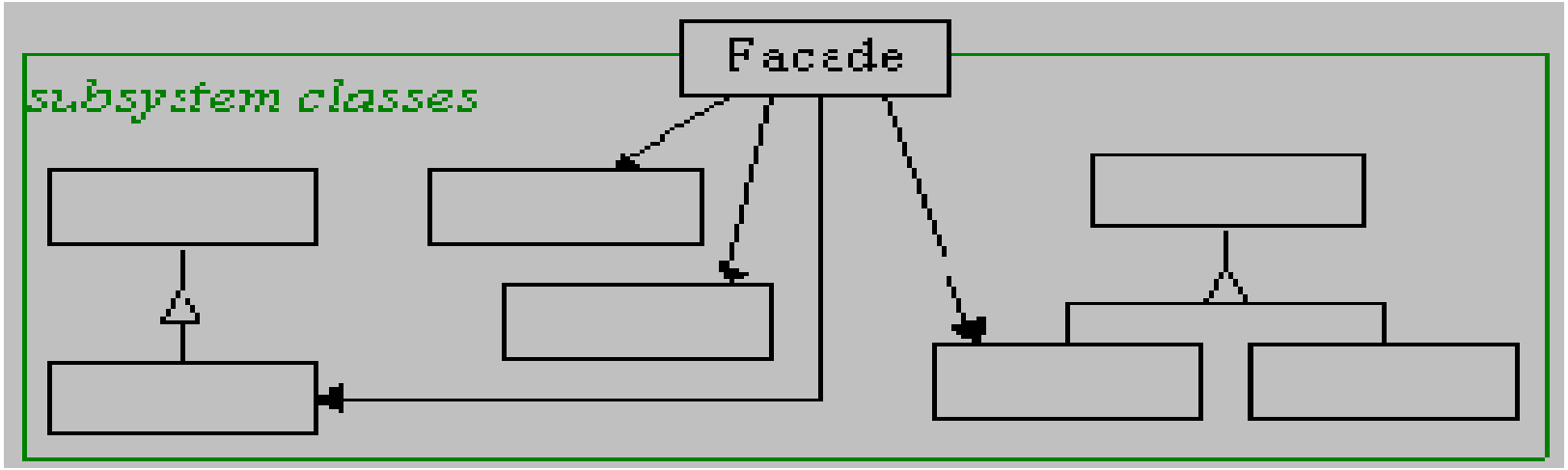


Facade模式(二)

- 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

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

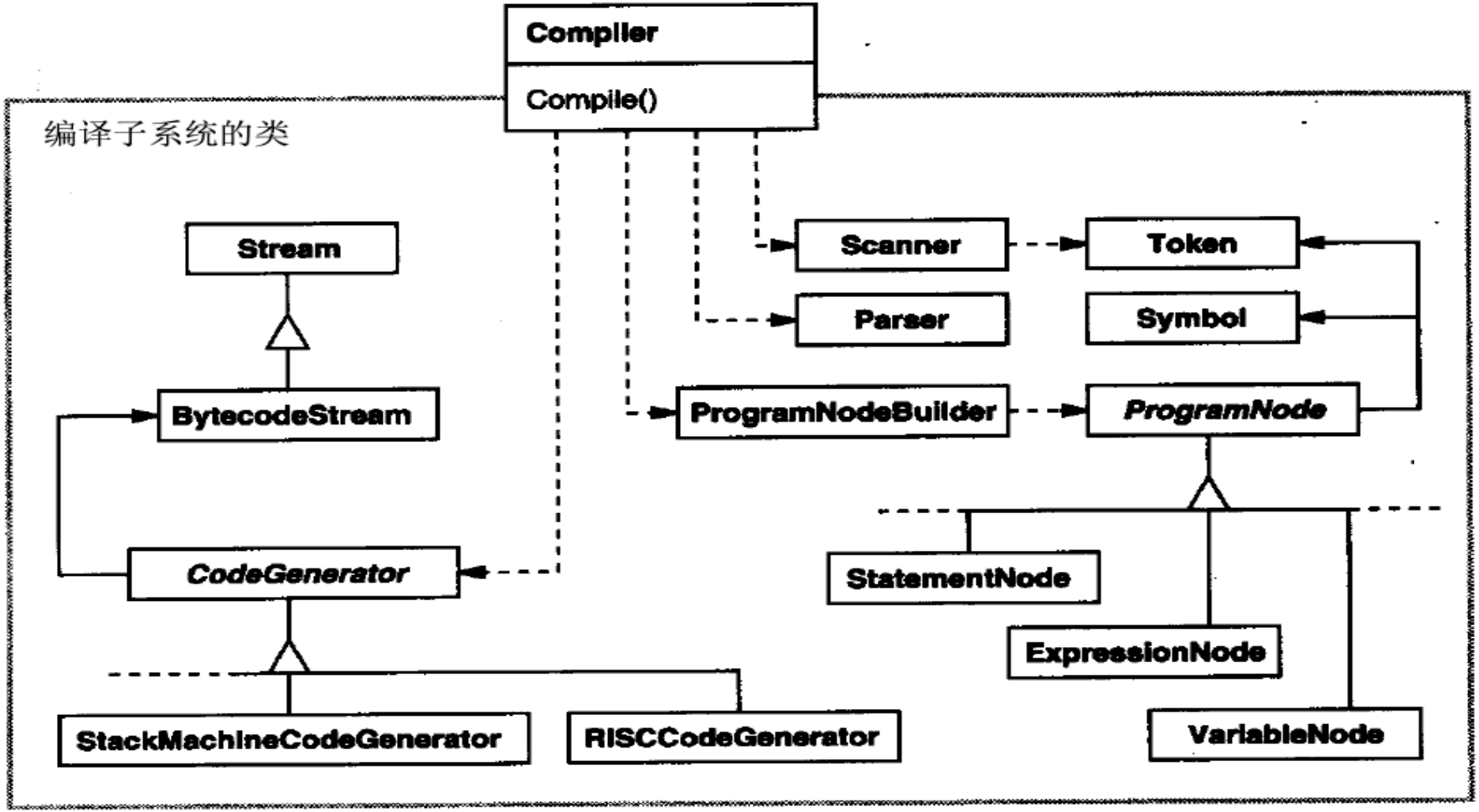
Facade模式(五)

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

例如有一个编程环境，它允许应用程序访问它的编译子系统。这个编译子系统包含了若干个类，如Scanner、Parser、ProgramNode、BytecodeStream和ProgramNodeBuilder，用于实现这一编译器。有些特殊应用程序需要直接访问这些类，但是大多数编译器的用户并不关心

语法分析和代码生成这样的细节；他们只是希望编译一些代码。对这些用户，编译子系统中那些功能强大但层次较低的接口只会使他们的任务复杂化。

- Examples（续）


Compiler
Compile()
编译子系统的类
Stream
Scanner
Token
Parser
Symbol
BytecodeStream
ProgramNodeBuilder
ProgramNode
CodeGenerator
StatementNode
ExpressionNode
StackMachineCodeGenerator
RISCCodeGenerator
VariableNode

模式 11：FlyWeight(一)

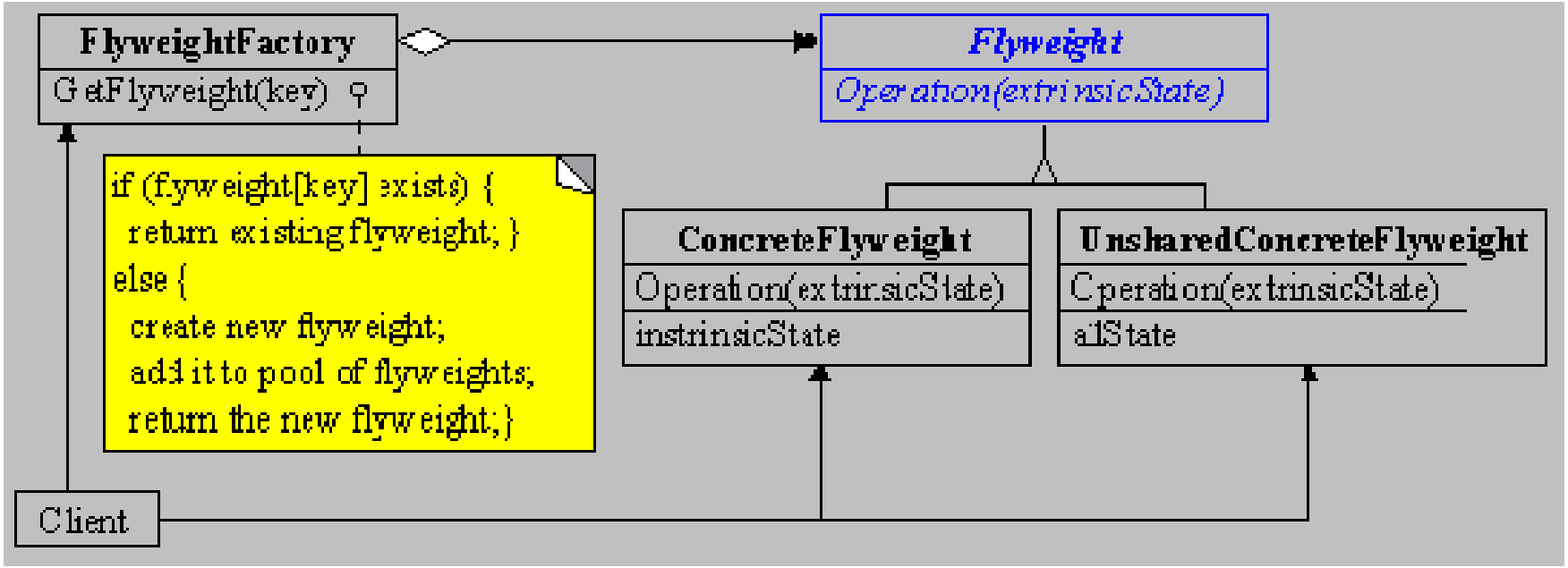
- Intent
  - Use sharing to support large numbers of fine-grained objects efficiently.
- Motivation
  - 当对象的粒度太小的时候，大量对象将会产生巨大的资源消耗，因此考虑用共享对象(flyweight)来实现逻辑上的大量对象。Flyweight对象可用于不同的context中，本身固有的状态不随context发生变化，而其他的状态随context而变化

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.

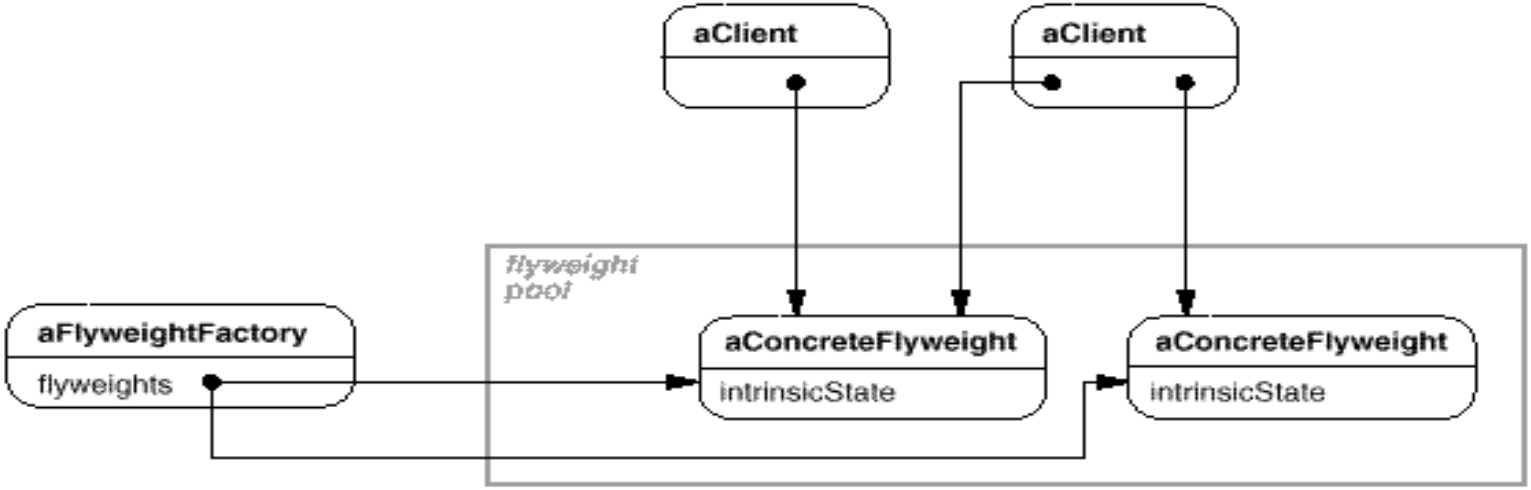
FlyWeight模式(三)

- Struct



FlyWeight模式(四)

- Struct(续)



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

FlyWeight模式(五)

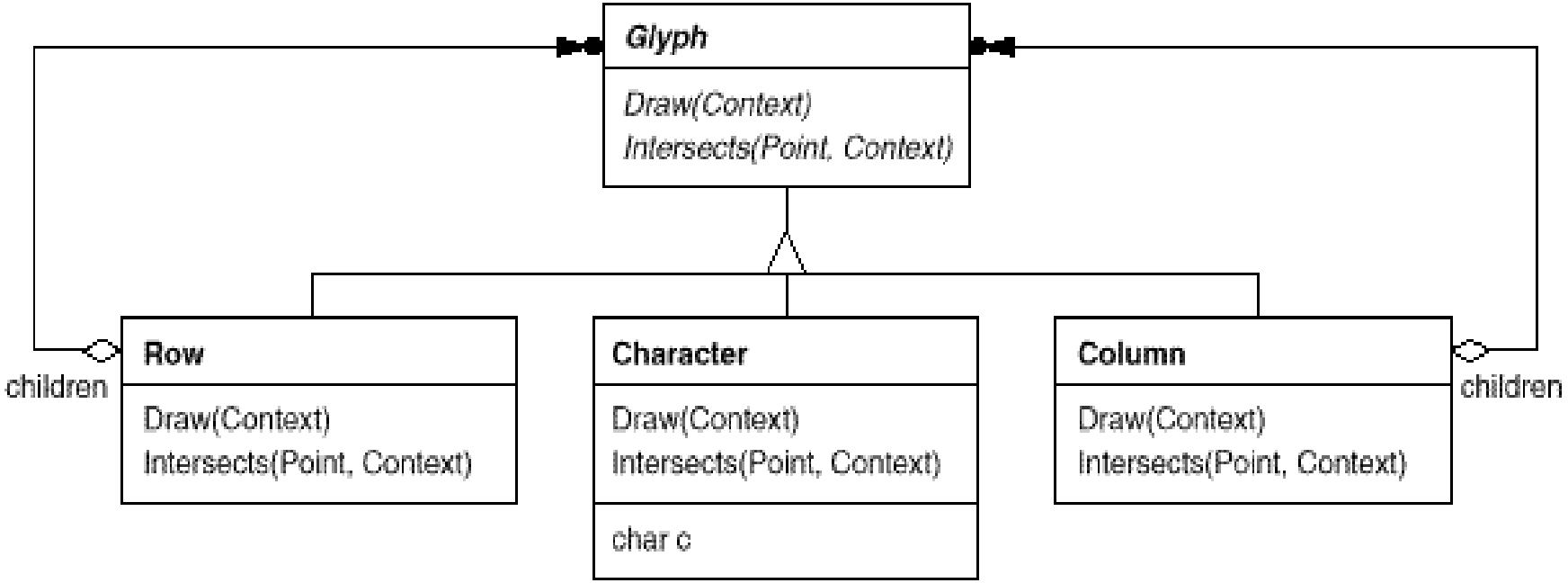
- Evaluation
  - 把对象的状态分开：intrinsic and extrinsic
  - 节约存储空间：内部状态的共享节约了大量空间，外部状态可通过计算获得从而进一步节约空间
  - flyweight与composite结合。Flyweight为leaf节点，并且父节点只能作为外部状态
- Implementation
  - Removing extrinsic state，尽可能做到实时计算(通过一个小的数据结构)
  - Managing shared objects，客户不能直接实例化flyweight，必须通过管理器，例如FlyweightFactory。
   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

FlyWeight模式(六)

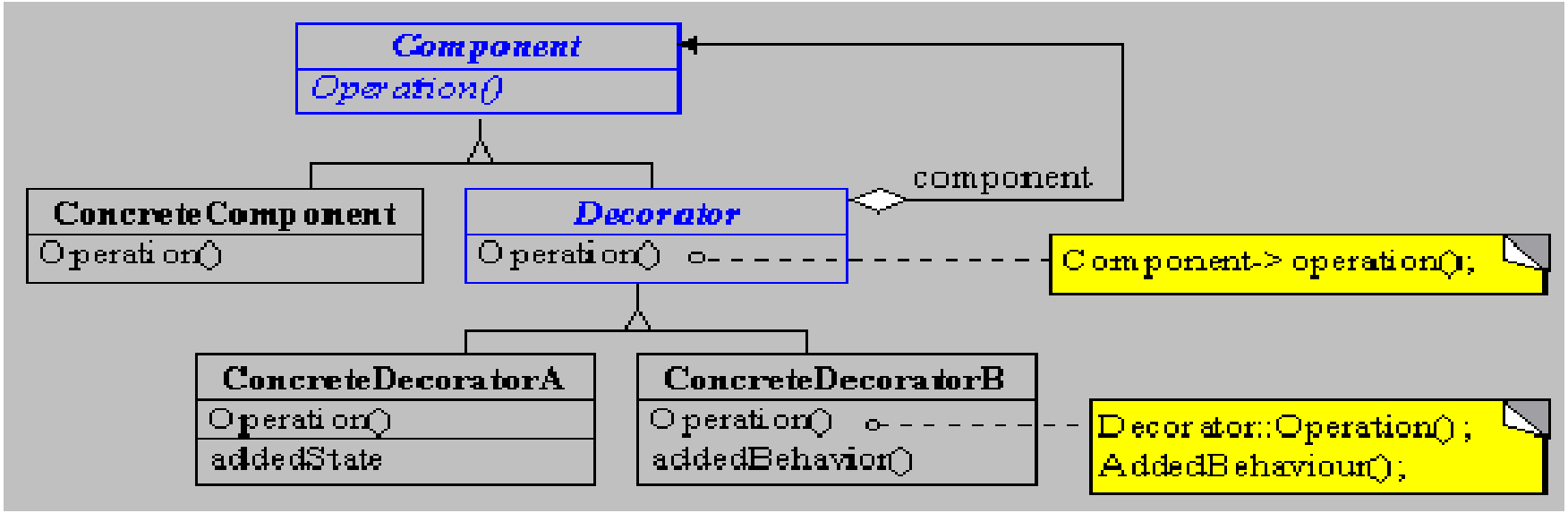
- Related Patterns
  - 与Composite模式组合
  - 可以用flyweight实现State和Strategy模式中的对象
- Examples
  - Excel中cell的管理
    - IOleItemContainer接口允许客户发现每一个cell对象
    - 用flyweight实现cell对象 —— tearoff技术
    - 对状态的有效管理是对象技术的一个进步
  - “ Design Patterns”中提到的文档编辑器的例子

FlyWeight模式(六)

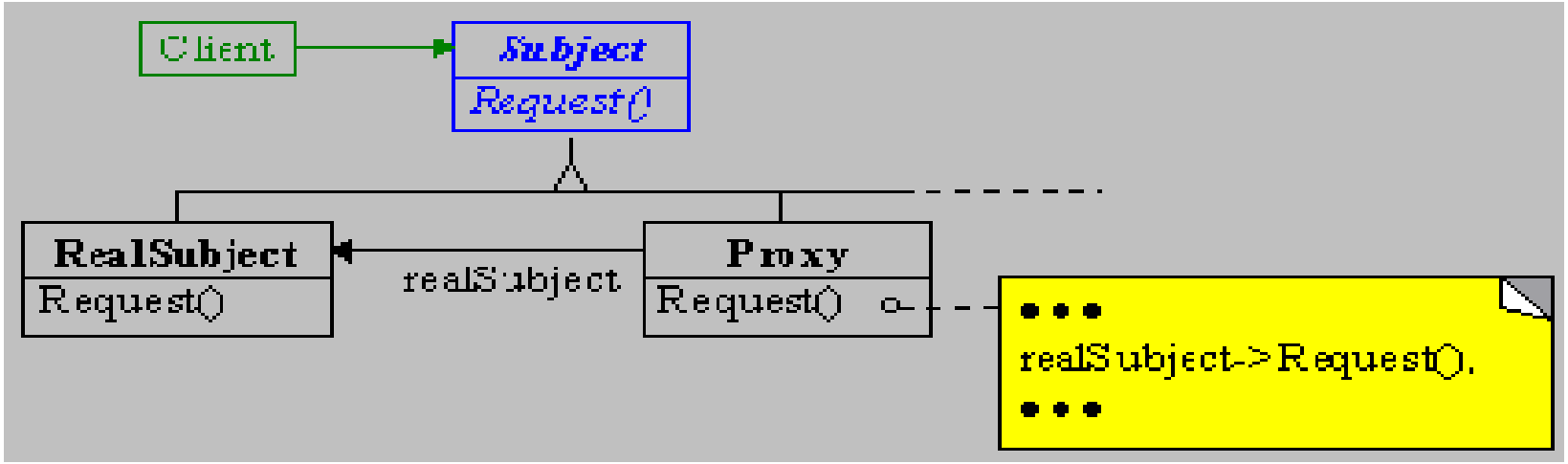
- Examples

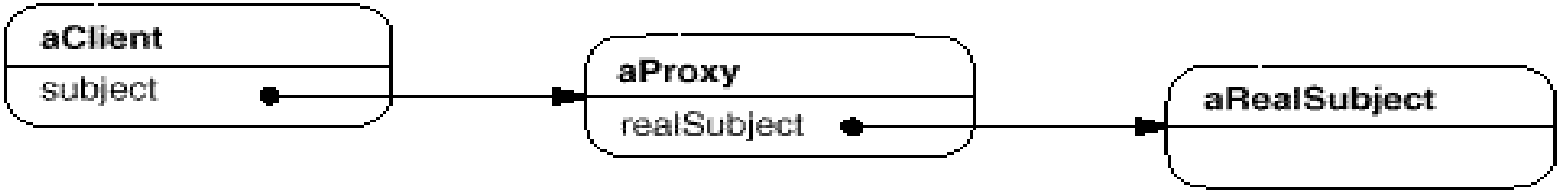


Structural Patterns : Decorator



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