设计模式

设计模式简介

课程目标

1. 理解松耦合设计思想
2. 掌握面向对象设计原则
3. 掌握重构技法改善设计
4. 掌握 GOF 核心设计模式

什么是设计模式

> 每一个模式描述了一个在我们周围不断重复发生的问题以及该问题的解决方案的核心。

参考教材

《设计模式：可复用面向对象软件的基础》 （简称： GOF ）

从面向对象谈起

> 1. **底层思维**：向下，如何把握机器底层从微观理解对象构造（语言构造 -> 编译转换 -> 内存模型 -> 运行时机制）
> 2. **抽象思维**：向上，如何将我们周围的实际抽象为程序代码（面向对象 -> 组件封装 -> 设计模式 -> 架构模式）

深入理解面向对象

> 1. 向下：面向对象三大机制
>   1. 封装：隐藏内部实现
>   2. 继承：复用现有代码
>   3. 多态：改写对象行为
> 2. 向上：什么是好的面向对象设计?

软件设计固有的复杂性

变化 ：客户需求、技术平台、开发团队、市场环境

如何解决复杂性

> 1. 分解：分而治之(**不容易复用**)，大问题 -> 小问题 ; 复杂问题 -> 简单问题
> 2. 抽象：忽略事物非本质细节，而去处理泛化和理想化了的对象模型（**易复用**）

软件设计目标：***复用***

面向对象设计原则

面向对象优点：抵御变化

重新认识面向对象

1. 理解隔离变化：适应变化，将变化带来影响减为最小
2. 各司其职：各个类的职责
3. 对象是什么：对象是拥有某种责任的抽象

面向对象设计原则

1. **依赖倒置原则（DIP）**： 高层模块(稳定)不应该依赖于低层模块(变化)，二者都应该依赖于抽象(稳定)； 抽象(稳定)不应该依赖于实现细节(变化)，实现细节应该依赖于抽象(稳定)。
2. **开放封闭原则（OCP）**：对扩展开放，对更改封闭。 类模块应该是可扩展的，但是不可修改。
3. **单一职责原则（SRP）**： 一个类应该仅有一个引起它变化的原因。变化的方向隐含着类的责任。
4. **Liskov 替换原则（LSP）**： 子类必须能够替换它们的基类(IS-A)。继承表达类型抽象。
5. **接口隔离原则（ISP）**： 不应该强迫客户程序依赖它们不用的方法。 接口应该小而完备。
6. **优先使用对象组合，而不是类继承**：类继承通常为“白箱复用”，对象组合通常为“黑箱复用”。 继承在某种程度上破坏了封装性，子类父类耦合度高。而对象组合则只要求被组合的对象具有好定义的接口，耦合度低。
7. **封装变化点**： 使用封装来创建对象之间的分界层，让设计者可以在分界层的一侧进行修改，而不会对另一侧产生不良的影响，从而实现层次间的松耦合。
8. **针对接口编程，而不是针对实现编程**： 不将变量类型声明为某个特定的具体类，而是声明为某个接口。 客户程序无需获知对象的具体类型，只需要知道对象所具有的接口。 减少系统中各部分的依赖关系，从而实现“高内聚、松耦合”的类型设计方案。

面向接口设计

产业强盛的标志：**接口标准化**

将设计原则提升为设计经验

1. **设计习语（Design Idioms）**： 描述与特定编程语言相关的低层模式，技巧，惯用法。
2. **设计模式（Design Patterns）**：主要描述的是“类与相互通信的对象之间的组织关系，包括它们的角色、职责、协作方式等方面。
3. **架构模式（Architecture Patterns）**：描述系统中与基本结构组织关系密切的高层模式，包括子系统划分，职责，以及如何组织它们之间关系的规则。

模板方法

重构关键技法

1. 静态 -> 动态
2. 早绑定 -> 晚绑定
3. 继承 -> 组合
4. 编译时依赖 -> 运行时依赖
5. 紧耦合 -> 松耦合

“组件协作”模式

典型模式：

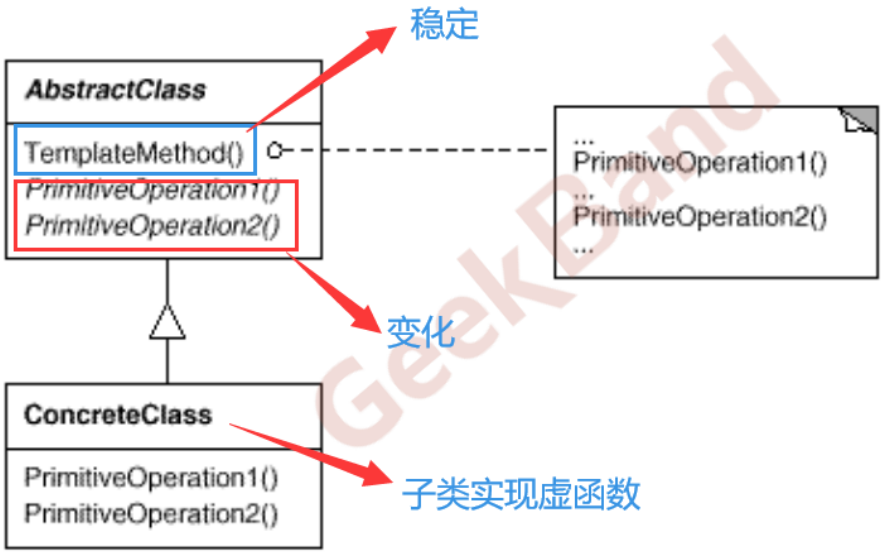
> 1. Template Method
> 2. Strategy
> 3. Observer / Event

定义：定义一个操作中的算法的**骨架（稳定）**，而将一些步骤**延迟（变化）到子类中。**Template Method 使得子类可以不改变（复用）一个算法的结构即可重定义(override 重写)该算法的某些特定步骤。

关键：区分软件中哪些部分是稳定（变化频率小）的？哪些部分是不稳定的？

UML 图

结构（Structure）



要点总结

> 1. Template Method模式是一种非常基础性的设计模式，在面向对象系统中有着大量的应用。它用最简洁的机制（虚函数的多态性）为很多应用程序框架提供了灵活的扩展点，是代码复用方面的基本实现结构。
> 2. 除了可以灵活应对子步骤的变化外，“不要调用我，让我来调用你”的反向控制结构是 Template Method 的典型应用。
> 3. 在具体实现方面，被 Template Method 调用的虚方法可以具有实现，也可以没有任何实现（抽象方法、纯虚方法），但一般推荐将它们设置为 protected 方法

策略模式

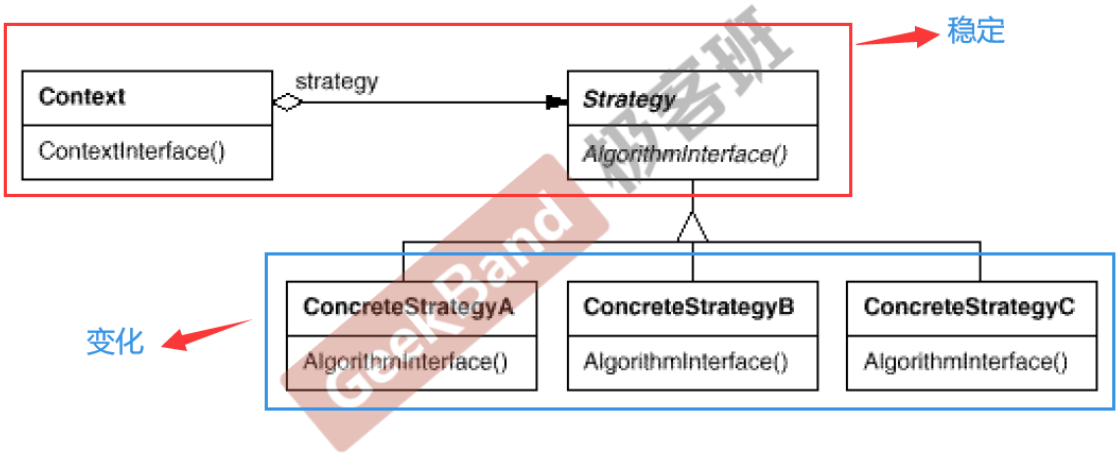
动机

> 1. 在软件构建过程中，某些对象使用的算法可能多种多样，经常改变，如果将这些算法都编码到对象中，将会使对象变得异常复杂；而且有时候支持不使用的算法也是一个性能负担。
> 2. **如何在运行时根据需要透明地更改对象的算法？将算法与对象本身解耦**，从而避免上述问题？

定义：定义一系列算法，把它们一个个封装起来，并且使它们可互相替换（变化）。该模式使得算法可独立于使用它的客户程序(稳定)而变化（扩展，子类化）。

UML 类图

结构（Structure）



要点总结

> 1. Strategy 及其子类为组件提供了一系列可重用的算法，从而可以使得类型在运行时方便地根据需要在各个算法之间进行切换。
> 2. Strategy模式提供了用条件判断语句以外的另一种选择，**消除条件判断语句**，就是在解耦合。***含有许多条件判断语句的代码通常都需要Strategy 模式。***
> 3. 如果 Strategy 对象没有实例变量，那么各个上下文可以共享同一个 Strategy 对象，从而节省对象开销。

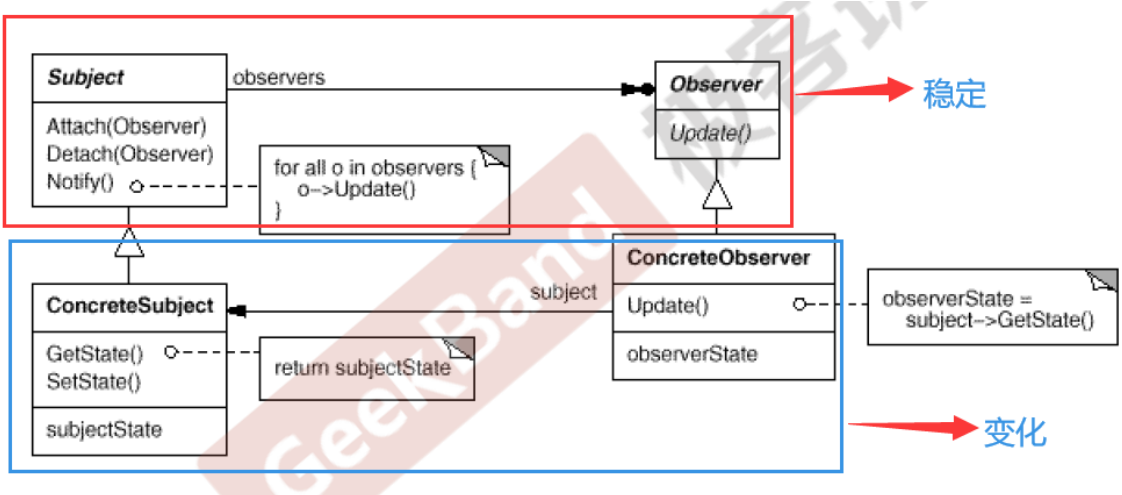
观察者模式

动机

在软件构建过程中，我们需要为某些对象建立一种"通知依赖关系" ——一个对象（目标对象）的状态发生改变，所有的依赖对象（观察者对象）都将得到通知。如果这样的依赖关系过于紧密，将使软件不能很好地抵御变化。

使用面向对象技术，可以将这种依赖关系弱化，并形成一种稳定
的依赖关系。从而实现软件体系结构的松耦合。

UML 类图



要点总结

1. 使用面向对象的抽象，Observer 模式使得我们可以**独立地改变目标与观察者，从而使二者之间的依赖关系达致松耦合。**
2. **目标发送通知时，无需指定观察者，通知（可以携带通知信息作为参数）会自动传播。**
3. **观察者自己决定是否需要订阅通知**，目标对象对此一无所知。
4. Observer 模式是基于事件的 UI 框架中非常常用的设计模式，也是 MVC 模式的一个重要组成部分。

装饰模式

"单一职责"模式

1. 在软件组件的设计中，如果责任划分的不清晰，使用继承得到的结果往往是随着需求的变化，子类急剧膨胀，同时充斥着重复代码，这时候的关键是划清责任。
2. 典型模式
   • Decorator
   • Bridge

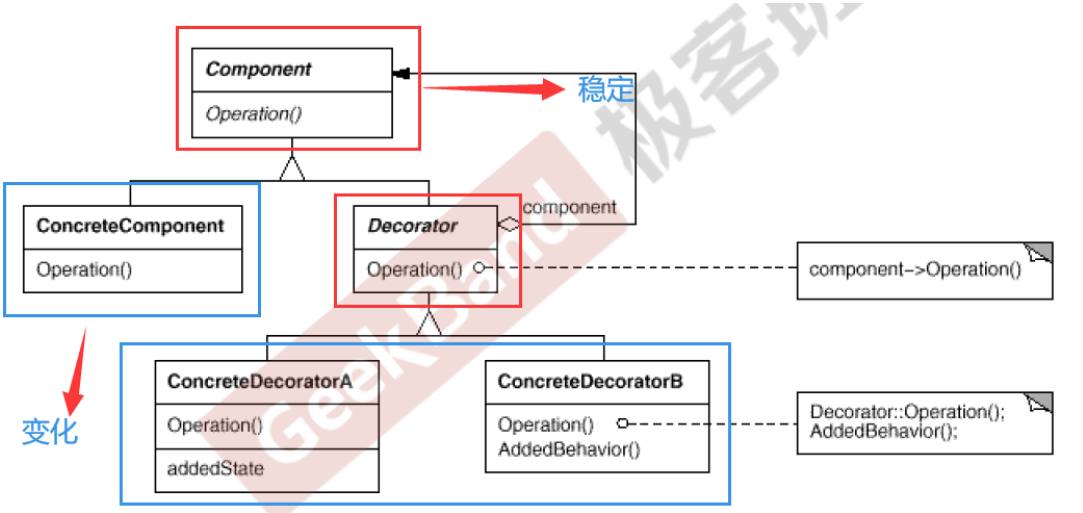
动机

1. 在某些情况下我们可能会"过度地使用继承来扩展对象的功能"，由于继承为类型引入的静态特质，使得这种扩展方式缺乏灵活性；并且随着子类的增多（扩展功能的增多），各种子类的组合（扩展功能的组合）会导致更多子类的膨胀。
2. 如何使"对象功能的扩展"能够根据需要来动态地实现？同时避免"扩展功能的增多"带来的子类膨胀问题？从而使得任何"功能扩展变化"所导致的影响将为最低？

模式定义

动态（组合）地给一个对象增加一些额外的职责。就增加功能而言，Decorator模式比生成子类（继承）更为灵活（消除重复代码 & 减少子类个数）。

UML 类图



要点总结

1. 通过采用组合而非继承的手法， Decorator模式实现了在运行时动态扩展对象功能的能力，而且可以根据需要扩展多个功能。避免了使用继承带来的"灵活性差"和"多子类衍生问题"。
2. Decorator类在接口上表现为 **is-a Component的继承关系**，即Decorator类继承了Component类所具有的接口。但在实现上又表现为 **has-a Component的组合关系**，即Decorator类又使用了另外一个Component类。

> 3. Decorator模式的目的并非解决“多子类衍生的多继承”问题，**Decorator模式应用的要点在于解决“主体类在多个方向上的扩展功能”——是为“装饰”的含义。**

桥模式

动机

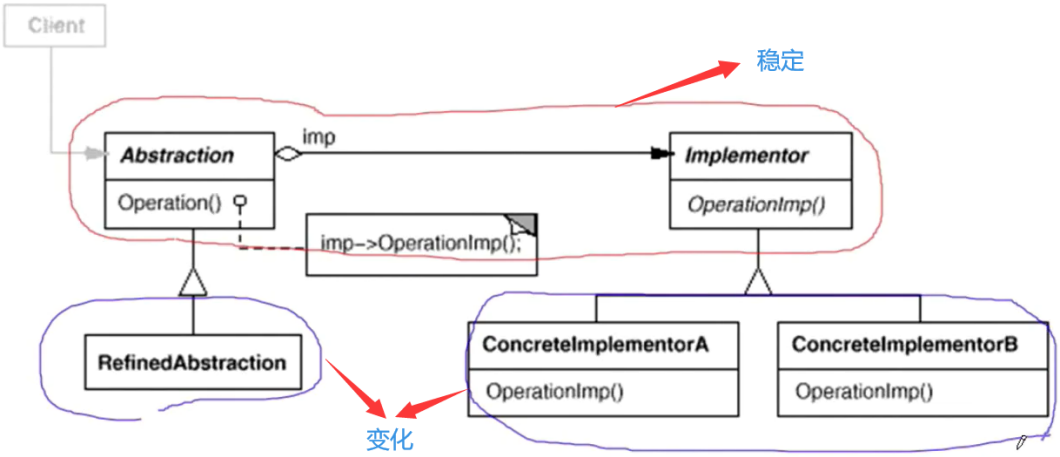
由于某些类型的固有的实现逻辑，使得它们具有两个变化的维度，乃至多个纬度的变化。

如何应对这种“多维度的变化”？如何利用面向对象技术来使得类型可以轻松地沿着两个乃至多个方向变化，而不引入额外的复杂度？

定义

将抽象部分(业务功能)与实现部分(平台实现)分离，使它们都可以独立地变化。

UML 类图



要点总结

> 1. Bridge模式使用“对象间的组合关系”解耦了抽象和实现之间固有的绑定关系，使得抽象和实现可以沿着各自的维度来变化。所谓抽象和实现沿着各自纬度的变化，即“子类化”它们。
> 2. Bridge模式有时候类似于多继承方案，但是多继承方案往往违背单一职责原则（即一个类只有一个变化的原因），复用性比较差。Bridge模式是比多继承方案更好的解决方法。
> 3. Bridge模式的应用一般在“两个非常强的变化维度”，有时一个类也有多于两个的变化维度，这时可以使用Bridge的扩展模式。

工厂方法模式

对象创建模式

1. 通过“对象创建” 模式绕开new，来避免对象创建（new）过程中所导致的紧耦合（依赖具体类），从而支持对象创建的稳定。它是接口抽象之后的第一步工作。
2. **典型模式**

- Factory Method
- Abstract Factory
- Prototype
- Builder

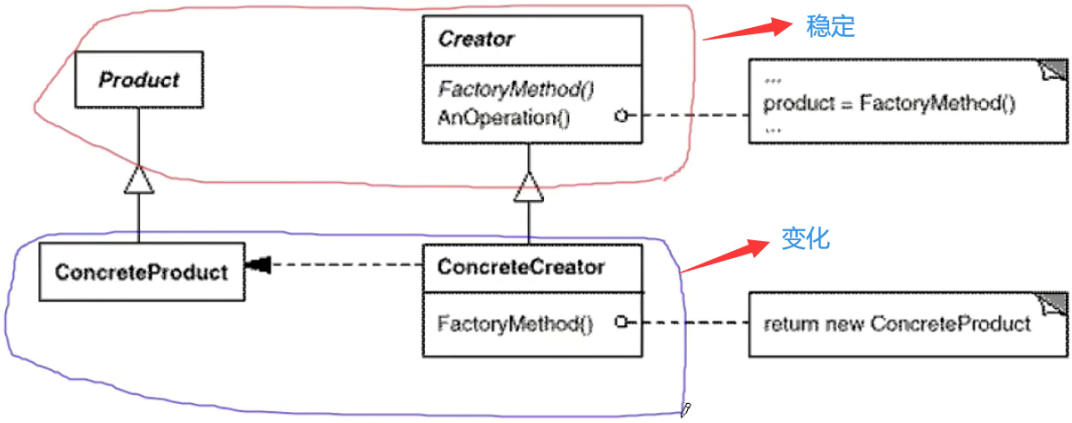
动机

> 1. 在软件系统中，经常面临着创建对象的工作；由于需求的变化，需要创建的对象的具体类型经常变化。
> 2. 如何应对这种变化？**如何绕过常规的对象创建方法(new)，提供一种“封装机制”来避免客户程序和这种“具体对象创建工作”的紧耦合?**

模式定义

定义一个**用于创建对象的接口**，让子类决定实例化哪一个类。**Factory Method使得一个类的实例化延迟（目的：解耦，手段：虚函数）到子类。**

UML 类图



要点总结

> 1. Factory Method模式用于隔离类对象的使用者和具体类型之间的耦合关系。面对一个经常变化的具体类型，紧耦合关系(new)会导致软件的脆弱。
> 2. Factory Method模式通过**面向对象的手法**，将所要创建的具体对象工作**延迟**到子类，从而实现一种扩展（而非更改）的策略，较好地解决了这种紧耦合关系。
> 3. Factory Method模式解决“单个对象”的需求变化。缺点在于要求创建方法

抽象工厂

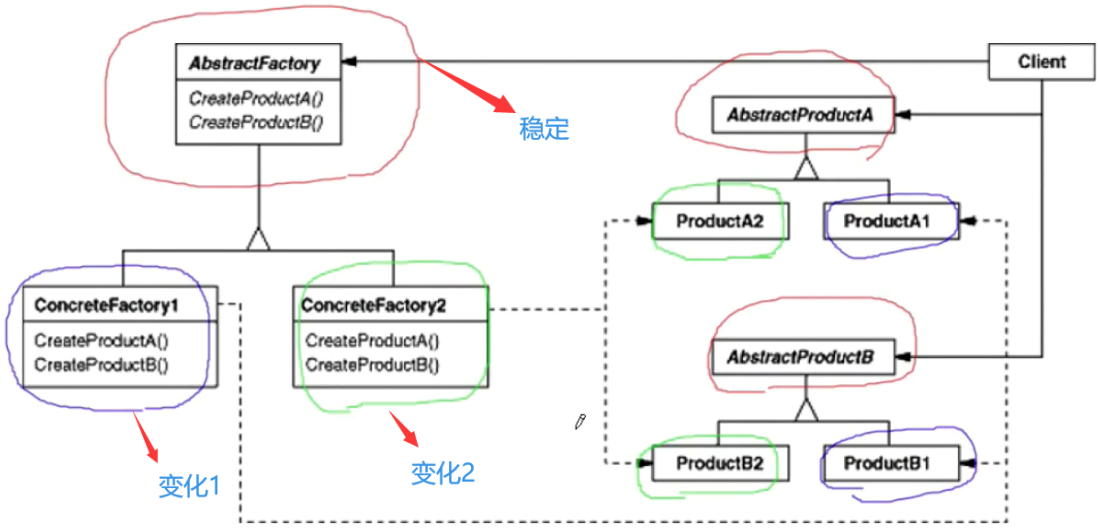
动机

1. 在软件系统中，经常面临着“一系列相互依赖的对象”的创建工作；同时，由于需求的变化，往往存在更多系列对象的创建工作。
2. 如何应对这种变化？如何绕过常规的对象创建方法(new)，提供一种“封装机制”来避免客户程序和这种“多系列具体对象创建工作”的紧耦合?

定义

提供一个接口，让该接口负责创建一系列“相关或者相互依赖的对象”，无需指定它们具体的类。

UML 类图



要点总结

> 1. 如果没有应对“多系列对象构建”的需求变化，则没有必要使用Abstract Factory模式，这时候使用简单的工厂完全可以。
> 2. “系列对象”指的是在某一特定系列下的对象之间有相互依赖、或作用的关系。不同系列的对象之间不能相互依赖。
> 3. Abstract Factory模式主要在于应对“新系列”的需求变动。其缺点在于难以应对“新对象”的需求变动。

原型模式

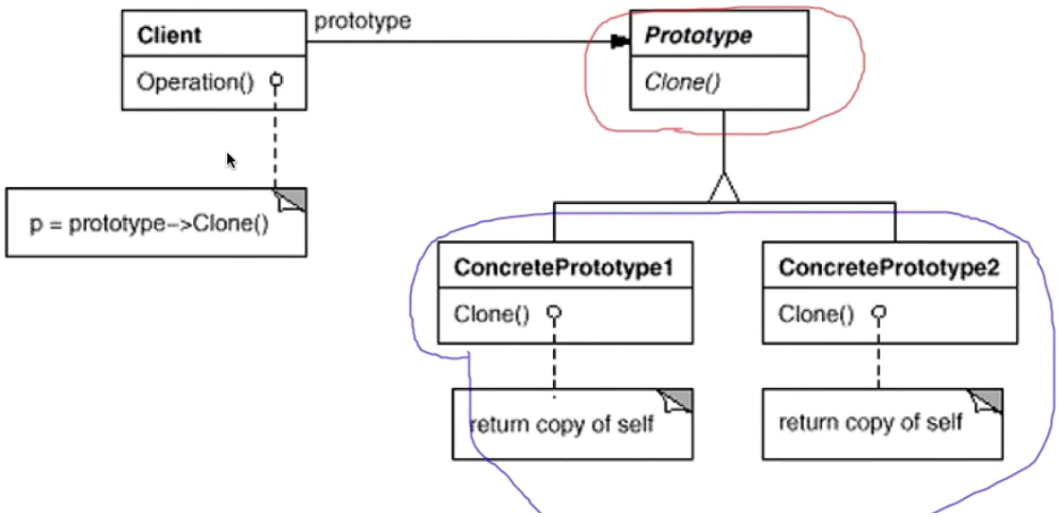
动机

1. 在软件系统中，经常面临着**“某些结构复杂的对象”的创建工作**；由于需求的变化，这些对象经常面临着剧烈的变化，但是它们却拥有比较稳定一致的接口。
2. 如何应对这种变化?如何向“客户程序（使用这些对象的程序）”隔离出“这些易变对象”，从而使得“依赖这些易变对象的客户程序”不随着需求改变而改变?

定义

使用原型实例指定创建对象的种类，然后通过**拷贝（深拷贝）**这些原型来创建新的对象。

UML 类图



要点总结

> 1. Prototype模式同样用于隔离类对象的使用者和具体类型(易变类)之间的耦合关系，它同样要求这些“易变类”拥有“稳定的接口”。
> 2. Prototype模式对于“如何创建易变类的实体对象”采用“原型克隆’的方法来做，它使得我们可以非常灵活地动态创建“拥有某些稳定接口'”的新对象——所需工作仅仅是注册一个新类的对象(即原型),然后在任何需要的地方Clone。
> 3. Prototype模式中的Clone方法可以利用某些框架中的序列化来实现深拷贝。

构建器模式

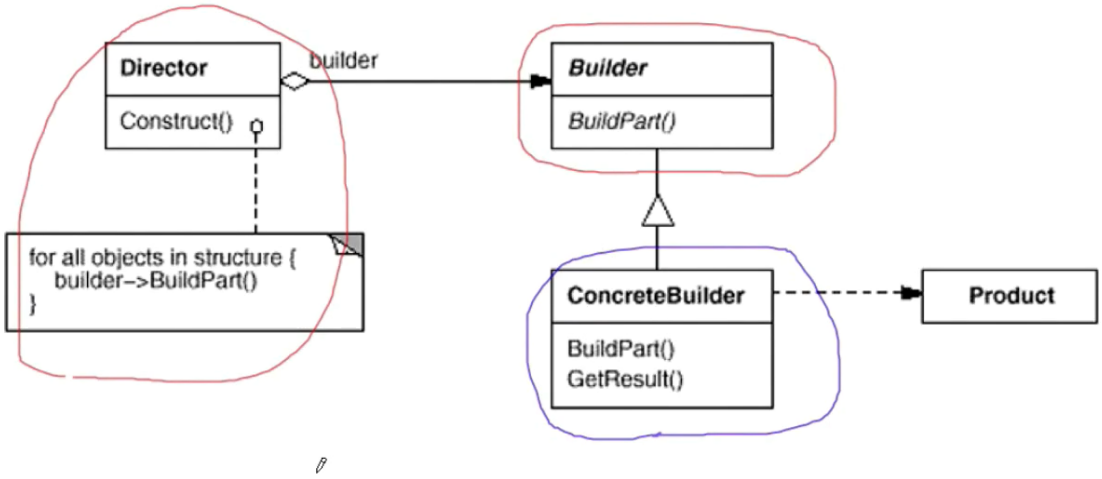
动机

1. 在软件系统中，有时候面临着“一个复杂对象”的创建工作，其通常由各个部分的子对象用一定的算法构成；由于需求的变化，这个复杂对象的各个部分经常面临着剧烈的变化，但是将它们组合在一起的算法却相对稳定。
2. 如何应对这种变化？如何提供一种“封装机制”来隔离出“复杂对象的各个部分”的变化，从而保持系统中的“稳定构建算法”不随着需求改变而改变？

定义

将一个复杂对象的构建与其表示相分离，使得同样的构建过程(稳定)可以创建不同的表示(变化)。

UML 类图



要点总结

> 1. Builder 模式主要用于“分步骤构建一个复杂的对象”。在这其中“分步骤”是一个稳定的算法，而复杂对象的各个部分则经常变化。
> 2. 变化点在哪里，封装哪里—— Builder模式主要在于应对“复杂对象各个部分”的频繁需求变动。其缺点在于难以应对“分步骤构建算法”的需求变动。
> 3. 在Builder模式中，要注意不同语言中构造器内调用虚函数的差别（C++ vs. C#）。

单件模式

对象性能模式

面向对象很好地解决了“抽象”的问题，但是必不可免地要付出一定的代价。对于通常情况来讲,面向对象的成本大都可以忽略不计。但是某些情况,面向对象所带来的成本必须谨慎处理。

典型模式:

- Singleton
- Flyweight

动机

1. 在软件系统中，经常有这样一些特殊的类，必须保证它们在系统中只存在一个实例，才能确保它们的逻辑正确性、以及良好的效率。
2. 如何绕过常规的构造器，提供一种机制来保证一个类只有一个实例?
3. 这应该是类设计者的责任，而不是使用者的责任。

定义

保证一个类仅有一个实例，并提供一个该实例的全局访问点。

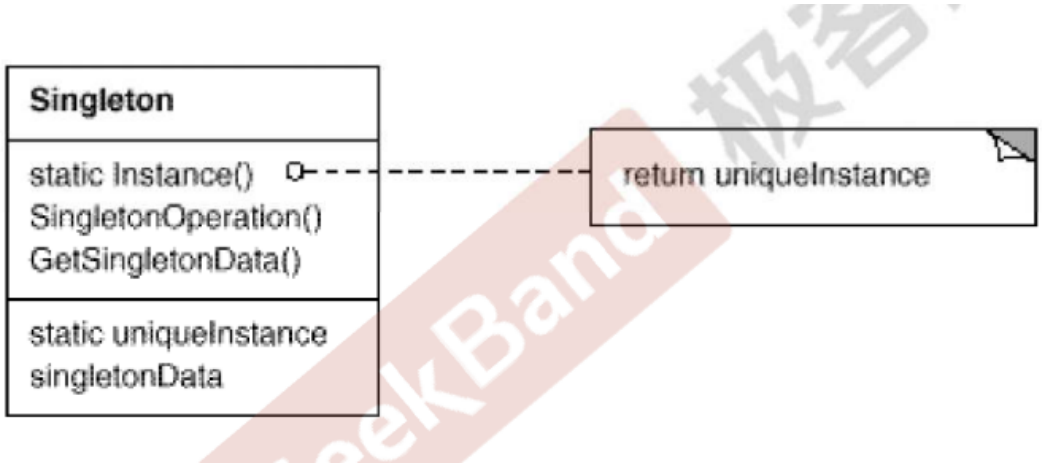
代码

```
#include <atomic>
#include <memory>
#include <mutex>
using namespace std;

class Singleton
{
```

```
private:
    Singleton();
    Singleton(const Singleton &other); // 拷贝构造函数设置为私有 or =delete

    static std::atomic<Singleton *> m_instance;
    static std::mutex m_mutex;

public:
    static Singleton *getInstance();
    static Singleton *m_instance;
};

Singleton *Singleton::m_instance = nullptr; // 静态成员初始化

//线程非安全版本
Singleton *Singleton::getInstance()
{
    if (m_instance == nullptr)
    {
        m_instance = new Singleton(); // 保证只创建一个对象
    }
    return m_instance;
}

//线程安全版本，但锁的代价过高
Singleton *Singleton::getInstance()
{
    Lock lock; // 加锁
    if (m_instance == nullptr)
    {
        m_instance = new Singleton();
    }
    return m_instance;
}

// 很著名的一种实现方式----双检查锁  实际上不正确
// 双检查锁，但由于内存读写reorder不安全
Singleton *Singleton::getInstance()
{
    // 只有指针为空的情况下才需要加锁 降低代价
    if (m_instance == nullptr)
    {
        Lock lock;
        if (m_instance == nullptr) // 必须判断 双检查
        {
            m_instance = new Singleton();
            /**      理想执行顺序
             * 1. 分配内存
             * 2. 调用构造函数
             * 3. 返回指针
             *
             *      reorder 之后可能顺序
             * 1. 分配内存
             * 2. 返回指针
             * 3. 调用构造函数
             *
             * 假设线程 A 执行到 2. 时但是还未执行 3;
```

```
                * 线程 B 执行该段代码，发现了指针非空并返回，但是此时对象并没有构造，导致未定义
行为。
                */
            }
        }
        return m_instance;
}

// C++ 11版本之后的跨平台实现 (volatile)
std::atomic<Singleton *> Singleton::m_instance;
std::mutex Singleton::m_mutex;

Singleton *Singleton::getInstance()
{
    Singleton *tmp = m_instance.load(std::memory_order_relaxed);
    std::atomic_thread_fence(std::memory_order_acquire); //获取内存fence
    if (tmp == nullptr)
    {
        std::lock_guard<std::mutex> lock(m_mutex);
        tmp = m_instance.load(std::memory_order_relaxed);
        if (tmp == nullptr)
        {
            tmp = new Singleton;
            std::atomic_thread_fence(std::memory_order_release); //释放内存fence
            m_instance.store(tmp, std::memory_order_relaxed);
        }
    }
    return tmp;
}
```

UML 类图



要点总结

1. Singleton模式中的实例构造器可以设置为protected以允许子类派生。
2. Singleton模式一般不要支持拷贝构造函数和Clone接口，因为这有可能导致多个对象实例，与Singleton模式的初衷违背。
3. 如何实现多线程环境下安全的Singleton？注意对双检查锁的正确实现。

享元模式

动机

1. 在软件系统采用纯粹对象方案的问题在于大量细粒度的对象会很快充斥在系统中，从而带来很高的运行时代价——主要指内存需求方面的代价。
2. 如何在避免大量细粒度对象问题的同时，让外部客户程序仍然能够透明地使用面向对象的方式来进行操作?

定义

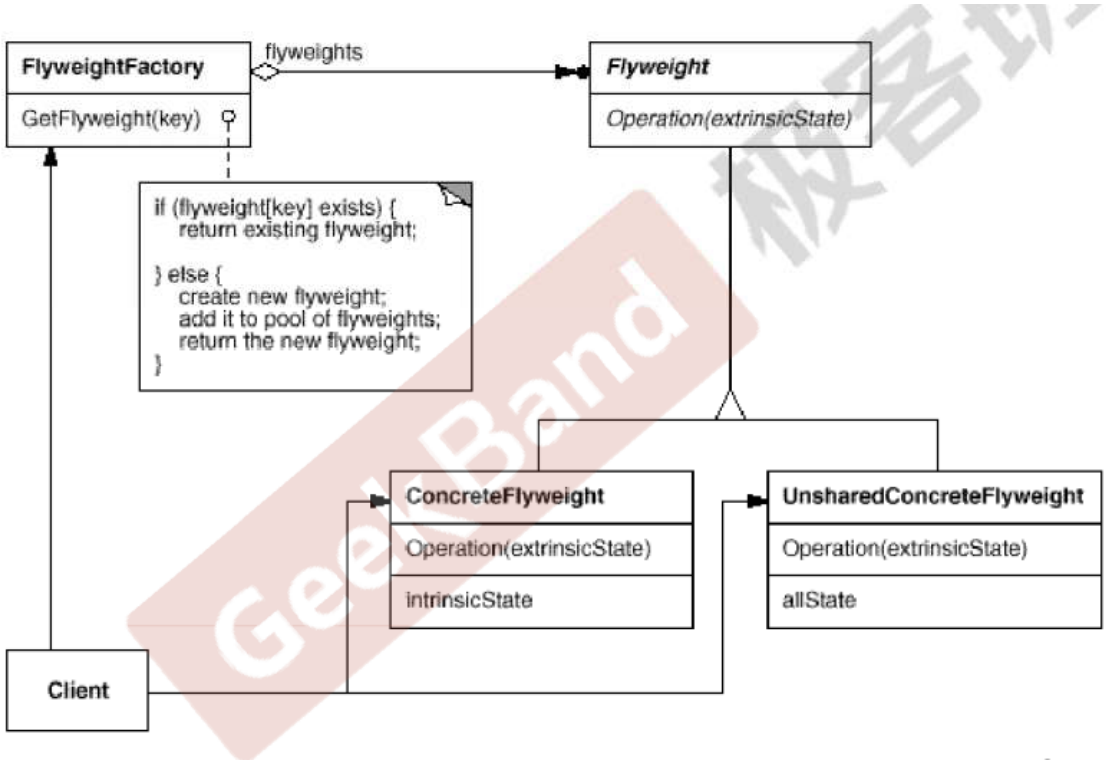
运用**共享技术**有效地支持大量细粒度的对象。

代码实现

```
#include <string>
#include <map>
using namespace std;

class Font
{
private:
    //unique object key
    string key;

    //object state
    //....

public:
    Font(const string &key)
    {
        //...
    }
};

class FontFactory
{
private:
    map<string, Font *> fontPool;

public:
    Font *GetFont(const string &key)
    {
        auto item = fontPool.find(key);

        if (item != fontPool.end())
        {
            return fontPool[key];
        }
        else
        {
            Font *font = new Font(key);
            fontPool[key] = font;
            return font;
        }
    }
```

```
    void clear()
    {
        //...
    }
};
```

UML 类图



要点总结

> 1. 面向对象很好地解决了抽象性的问题，但是作为一个运行在机器中的程序实体我们需要考虑对象的代价问题。Flyweight主要解决面向对象的**代价问题**，一般 *不触及面向对象的抽象性问题*。
> 2. Flyweight采用对象共享的做法来降低系统中对象的个数，从而降低细粒度对象给系统带来的内存压力。在具体实现方面，要注意对象状态的处理。
> 3. 对象的数量太大从而导致对象内存开销加大——什么样的数量才算大？这需要我们**仔细的根据具体应用情况进行评估，而不能凭空臆断。**

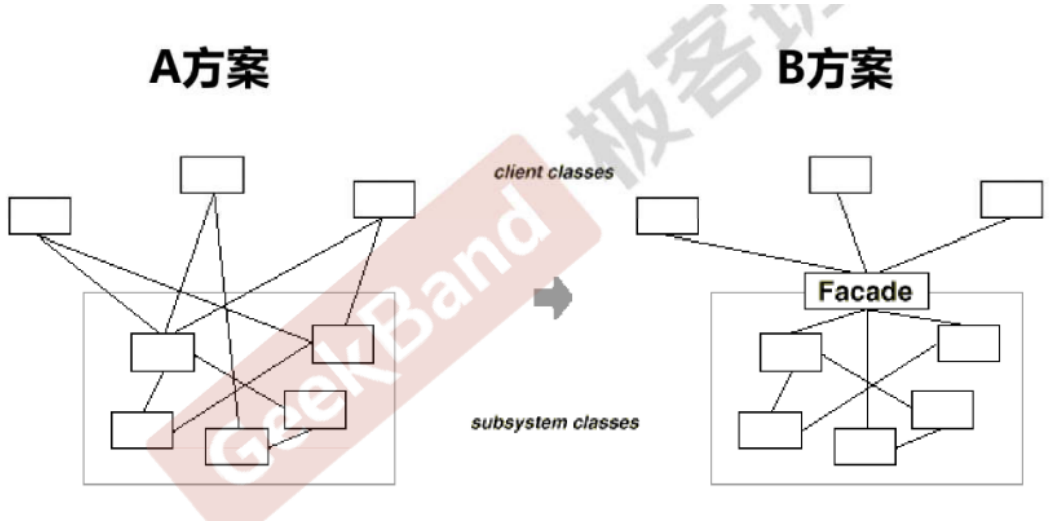
门面模式

接口隔离模式

> 1. 在组件构建过程中，某些接口之间直接的依赖常常会带来许多问题、甚至根本无法实现。采用添加一层间接（稳定）接口，来隔离本来互相紧密关联的接口是一种常见的解决方案。
> 2. 典型模式
>   1. 门面模式
>   2. 代理模式

> 3. 适配器模式
> 4. 中介者模式

系统间耦合的复杂度



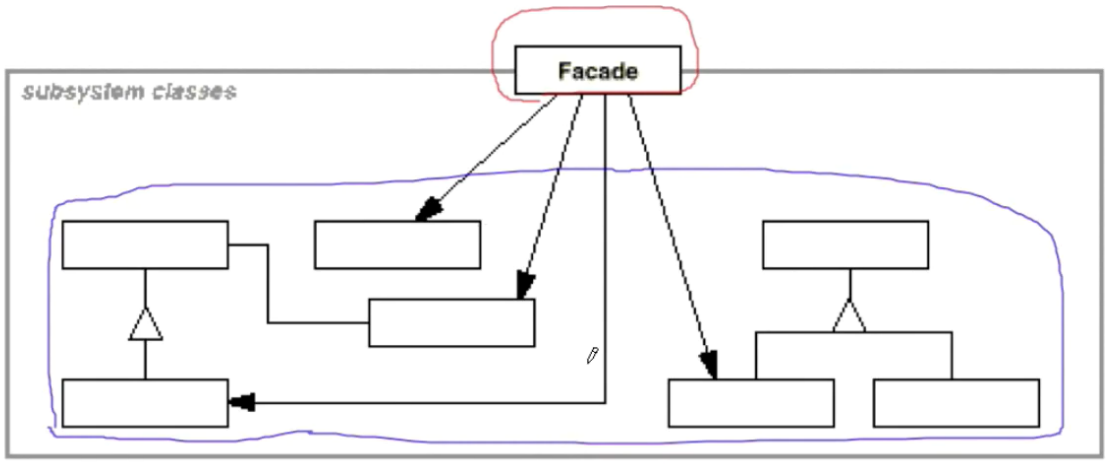
动机

1. 上述A方案的问题在于组件的客户和组件中各种复杂的子系统有了过多的耦合，随着外部客户程序和各子系统的演化，这种过多的耦合面临很多变化的挑战。
2. 如何简化外部客户程序和系统间的交互接口？如何将外部客户程序的演化和内部子系统的变化之间的依赖相互解耦？

定义

为子系统中的一组接口提供一个 **一致（稳定）**的界面，Facade 模式定义了一个高层接口，这个接口使得这一子系统更加 **容易使用（复用）**。

UML 类图



要点总结

> 1. 从客户程序的角度来看。Facade模式简化了整个组件系统的接口，对于组件内部与外部客户程序来说i达到了一种""解耦"的效果——内部子系统的任何变化不会影响到 Facade 接口的变化。
> 2. Facade设计模式更注重从 **架构的层次去看整个系统**，而不是单个类的层次。Facade很多时候更是一种架构设计模式。
> 3. Facade 设计模式并非一个集装箱，可以任意地放进任何多个对象。Facade模式中组件的内部应该是，"相互耦合关系比较大的一系列组件"，而不是一个简单的功能集合。

代理模式

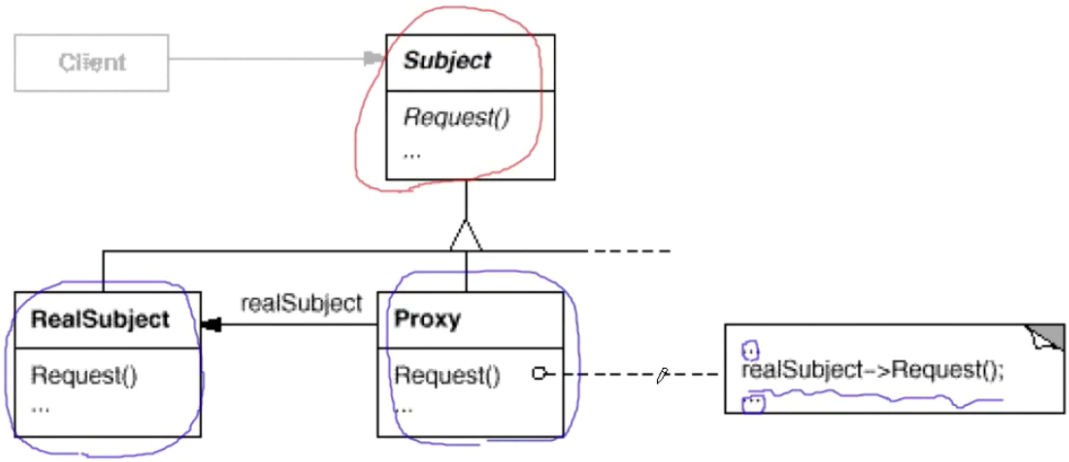
动机

1. 在面向对象系统中，有些对象由于某种原因(比如对象创建的开销很大，或者某些操作需要安全控制，或者需要进程外的访问等)，直接访问会给使用者、或者系统结构带来很多麻烦。
2. 如何在不失去透明操作对象的同时来管理/控制这些对象特有的复杂性？增加一层间接层是软件开发中常见的解决方式。

定义

为其他对象提供一种代理以控制（隔离，使用接口）对这个对象的访问。

UML 类图



要点总结

> 1. "增加一层间接层"是软件系统中对许多复杂问题的一种常见解决方法。在面向对象系统中直接使用某些对象会带来很多问题，作为间接层的 proxy 对象便是解决这一问题的常用手段。
> 2. 具体 proxy 设计模式的实现方法、实现粒度都相差很大,有些可能对单个对象做细粒度的控制，如 copy-on-write 技术,有些可能对组件模块提供抽象代理层，在架构层次对对象做 proxy。
> 3. Proxy并不一定要求保持接口完整的一致性，只要能够实现间接控制，有时候损及一些透明性是可以接受的。

适配器模式

动机

1. 在软件系统中，由于应用环境的变化，常常需要将"一些现存的对象"放在新的环境中应用，但是新环境要求的接口是这些现存对象所不满足的。
2. 如何应对这种"迁移的变化"？如何既能利用现有对象的良好实现，同时又能满足新的应用环境所要求的接口?

身边的适配器

转接头、电源适配器等

定义

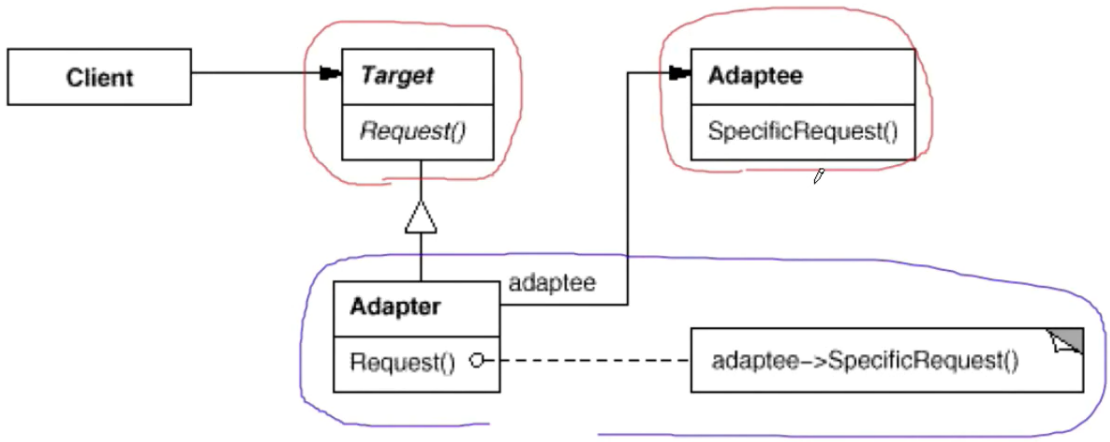
将一个类的接口转换成客户希望的另一个接口。Adapter 模式使得原本由于接口不兼容而不能一起的那些类可以一起工作。

代码

典型：STL 中 stack 和 queue 严格叫做适配器，默认利用 deque 实现。

```
stack<int, vector<int>> stk;

queue<int, vector<int>> q;
```

UML 类图



要点总结

> 1. Adapter模式主要应用于“希望复用一些现存的类，但是接口又与复用环境要求不一致的情况”,，在**遗留代码复用、类库迁移等方面**非常有用。
> 2. GoF 23 定义了两种Adapter模式的实现结构：对象适配器和类适配器。但**8类适配器采用" 多继承”的实现方式，一般不推荐使用**。对象适配器采用“对象组合”的方式,更符合松耦合精神。
> 3. Adapter模式可以实现的非常灵活，不必拘泥于 Gof 23 中定义的两种结构。例如完全可以将Adapter模式中的 ”现存对象” 作为新的接口方法参数，来达到适配的目的。

中介者模式

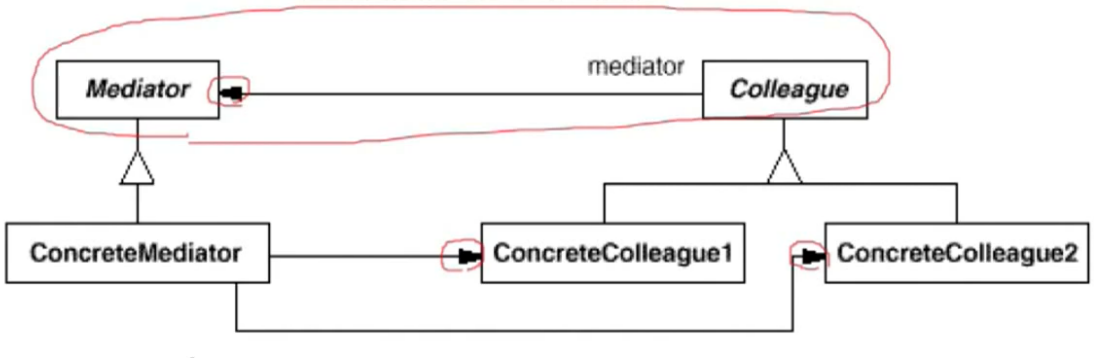
动机

1. 在软件构建过程中，经常会出现多个对象互相关联交互的情况，对象之间常常会维持一种复杂的引用关系，如果遇到一-些需求的更改，这种直接的引用关系将面临不断的变化。
2. 在这种情况下，我们可使用一一个“中介对象”来管理对象间的关联关系，避免相互交互的对象之间的紧耦合引用关系，从而更好地抵御变化。

定义

用一个中介对象来 **封装(封装变化)** 一系列的对象交互。中介者使各对象不需要显式的相互 **引用(编译时依赖→运行时依赖)**，从而使其耦合松散(管理变化)，而且可以独立地改变它们之间的交互。

结构



要点总结

> 1. 将多个对象间复杂的关联关系解耦，Mediator 模式将多个对象间的控制逻辑进行集中管理，变 “多个对象互相关联” 为 “多个对象和一个中介者关联”， 简化了系统的维护，抵御了可能的变化。
> 2. 随着控制逻辑的复杂化,，Mediator 具体对象的实现可能相当复杂。这时候可以对Mediator对象进行分解处理。
> 3. Facade模式是解耦系统间(单向)的对象关联关系；Mediator 模式是解耦系统内各个对象之间(双向)的关联关系。

状态模式

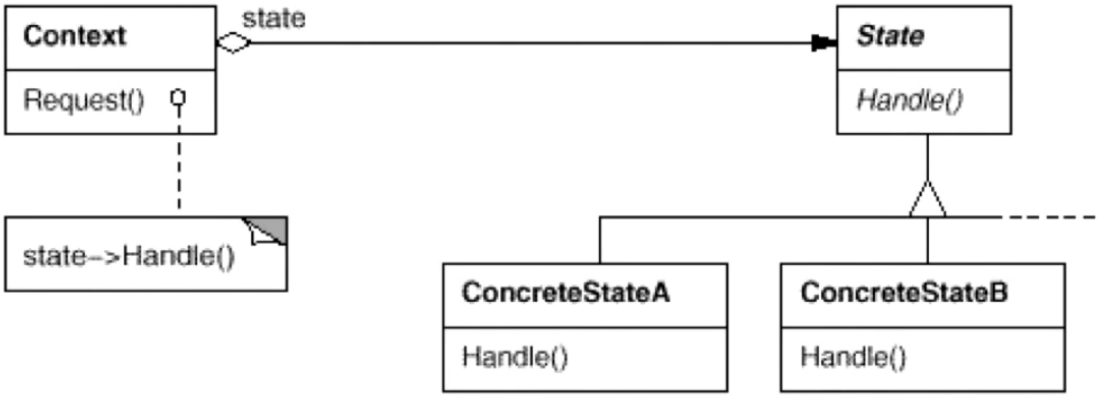
动机

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

定义

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

UML 类图



要点总结

> 1. State 模式将所有与一个特定状态相关的行为都放入一个 State 的子类对象中，在对象状态切换时，切换相应的对象；但同时维持 State 的接口，这样实现了具体操作与状态转换之间的解耦。
> 2. 为不同的状态引入不同的对象使得状态转换变得更加明确，而且可以保证不会出现状态不一致的情况,因为转换是原子性的一即要么彻底转换过来，要么不转换。
> 3. 如果 State 对象没有实例变量，那么各个上下文可以共享同一个 State 对象,从而节省对象开销。

备忘录

动机

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

定义

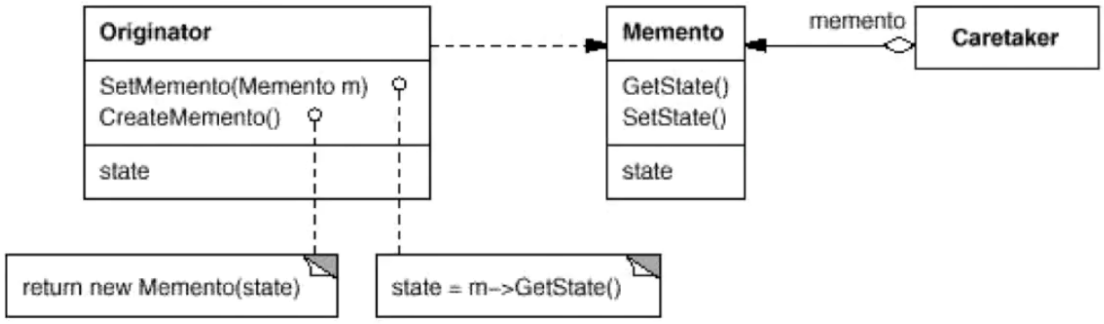
在不破坏封装性的前提下，捕获一个对象的内部状态，并在该对象之外保存这个状态。这样以后就可以将该对象恢复到原先保存的状态。

代码示意

```
#include <string>
using std::string;

class Memento
{
    string state;
    //..    状态
public:
    Memento(const string &s) : state(s) {}

    string getState() const
    {
        return state;
    }
```

```
    void setState(const string &s)
    {
        state = s;
    }
};

class Originator
{
    string state;
    //.... 状态
public:
    Originator() {}
    Memento createMomento()
    {
        Memento m(state); // 创建一个状态 并返回对象（保存当前状态）
        return m;
    }
    void setMomento(const Memento &m)
    {
        state = m.getState();
    }
};

int main()
{
    Originator orginator;

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

    //... 改变orginator状态

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

    return 0;
}
```

UML 类图



要点总结

> 1. 备忘录(Memento)存储原发器(Originator)对象的内部状态，在需要时恢复原发器状态。
> 2. Memento 模式的 **核心是信息隐藏**，即 Originator 需要向外接隐藏信息，保持其封装性。但同时又需要将状态保持到外界(Memento)。
> 3. 由于现代语言运行时(如C#、Java等)都具有相当的 **对象序列化** 支持，因此往往采用效率较高、又较容易正确实现的序列化方案来实现Memento模式。

组合模式

“数据结构”模式

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

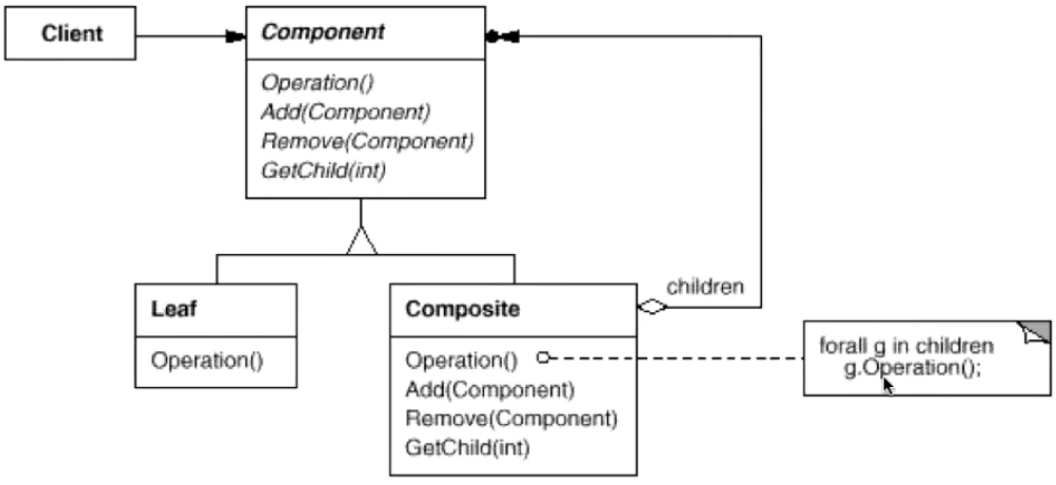
动机

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

定义

将对象组合成 **树形结构** 以表示“部分整体”的层次结构。Composite 使得用户 **对单个对象和组合对象的使用具有一致性(稳定)。**

UML 类图



要点总结

> 1. Composite模式采用树形结构来实现普遍存在的对象容器，从而 **将"一对多"的关系转化为"一对一"的关系**，使得客户代码可以一致地(复用)处理对象和对象容器，无需关心处理的是单个的对象，还是组合的对象容器。
> 2. 将"客户代码与复杂的对象容器结构"解耦是Composite的核心思想，解耦之后，客户代码将与纯粹的抽象接口——而非对象容器的内部实现结构——发生依赖，从而更能 "应对变化"。
> 3. Composite模式在具体实现中,可以让父对象中的子对象反向追溯；如果父对象有频繁的遍历需求，可使用缓存技巧来改善效率。

迭代器模式

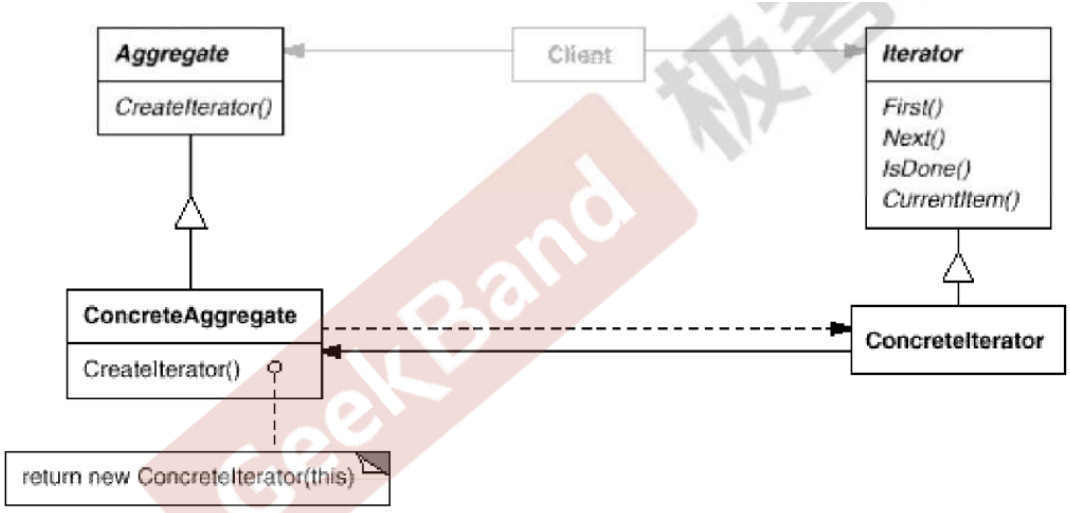
动机

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

定义

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

UML 类图



要点总结

> 1. 迭代抽象：访问一个聚合对象的内容而无需暴露它的内部表示。
> 2. 迭代多态：为遍历不同的集合结构提供一个统一的接口，从而支持同样的算法在不同的集合结构上进行操作。
> 3. 迭代器的健壮性考虑：遍历的同时更改迭代器所在的集合结构，会导致问题。

职责链模式

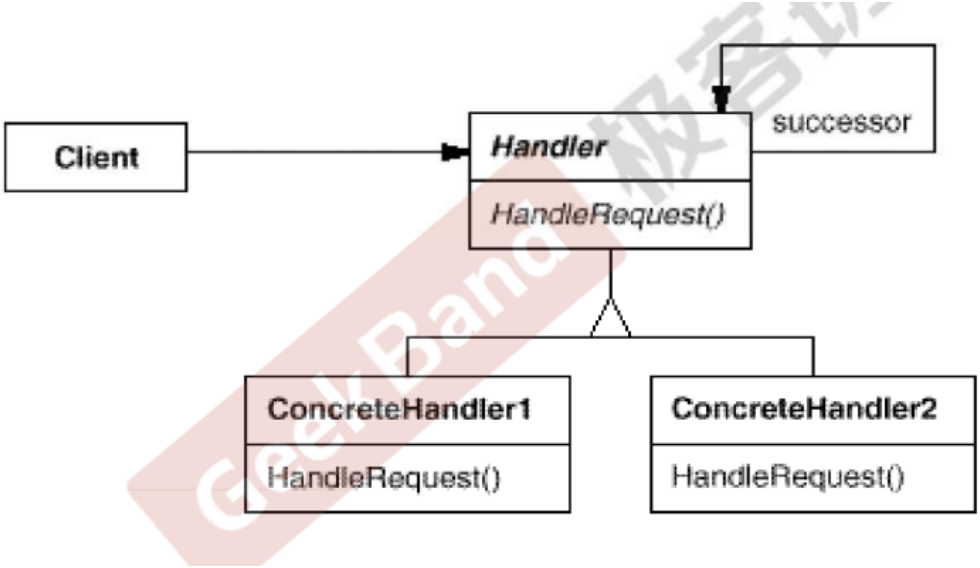
动机

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

定义

使多个对象都有机会处理请求，从而避免请求的发送者和接收者之间的耦合关系。**将这些对象连成一条链，并沿着这条链传递请求，直到有一个对象处理它为止。**

UML 类图



要点总结

> 1. Chain of Responsibility 模式的应用场合在于"一个请求可能有多个接受者，但是最后真正的接受者只有一一个”，这时候请求发送接受者的遇合有可能出现“变化脆弱”的症状，职责链的目的就是蒋兰者解耦，从而更好地应对变化。
> 2. 应用了Chain of Responsibility 模式后，对象的职责分派将更具灵活性。我们可以在运行时动态添加修改请求的处理职责。
> 3. 如果请求传递到职责链的末尾仍得不到处理，应该有一个合理的缺省机制。这也是每一个接受对象的责任，而不是发出请求的对象的责任。

命令模式

“行为变化”模式

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

动机

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

定义

将一个请求(行为)封装为一个对象，从而使你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。

UML类图

要点总结

> 1. Command 模式的根本目的在于将“行为请求者”与“行为实现者”解耦，在面向对象语言中，常见的实现手段是“将行为抽象为对象”。
> 2. 实现 Command 接口的具体命令对象 ConcreteCommand 有时候根据需要可能会保存一些额外的状态信息。**通过使用 Composite 模式，可以将多个“命令”封装为一个“复合命令” MacroCommand。**
> 3. Command 模式与 C+ + 中的函数对象有些类似。但两者定义行为接口的规范有所区别：Command以面向对象中的"接口-实现”来定义行为接口规范，更严格,但有性能损失；C++函数对象以函数签名来定义行为接口规范，更灵活，性能更高。

访问器模式

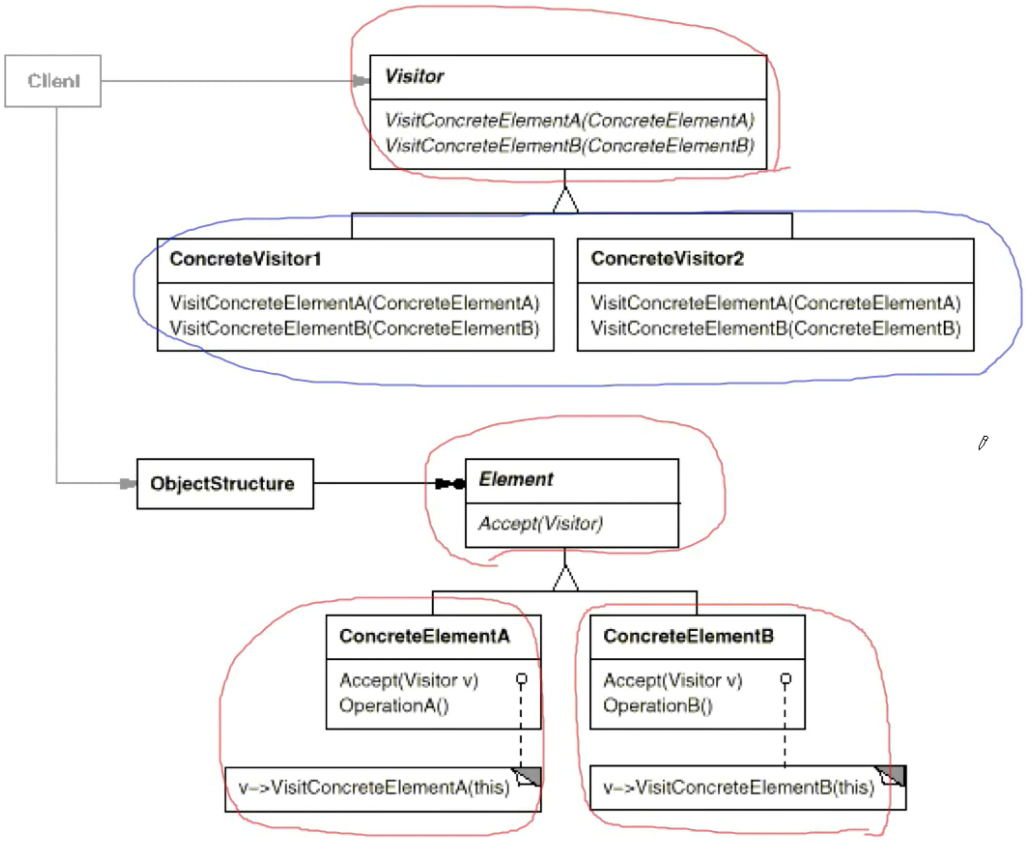
动机

1. 在软件构建过程中，由于需求的改变,某些类层次结构中常常需要增加新的行为(方法) , 如果直接在基类中做这样的更改,将会给子类带来很繁重的变更负担,甚至破坏原有设计。
2. 如何在不更改类层次结构的前提下,在运行时根据需要透明地为类层次结构_上的各个类动态添加新的操作,从而避免上述问题?

定义

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

UML 类图



要点总结

> 1. Visit模式通过所谓双重分发(double disath)来实现在不更改(不添加新的操作——编译时) Element 类层次结构前提下，在运行时透明地为类层次结构上的各个类动态添加新的操作（支持变化）。
> 2. 所谓双重分发即 Visitor 模式中间包括了两个多态分发（注意其中的多态机制）：第一个为 accept 方法的多态辨析；第二个为 visitElement 方法的多态辨析。
> 3. Visitor 模式的最大确定在于扩展类层次结构（增添新的 Element 子类），会导致 Visitor 类的改变。因此 Visitor 模式适用于 "Element 类层次结构稳定，为其中的操作却经常面临频繁改动"。

解析器模式

"领域规则"模式

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

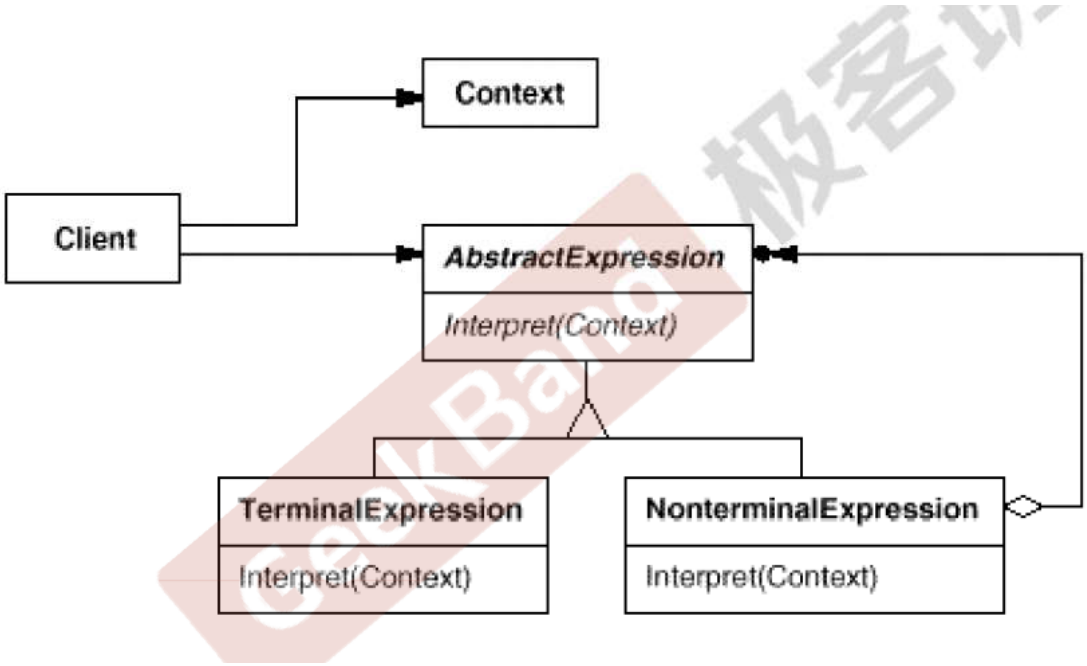
动机

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

定义

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

UML 类图



要点总结

> 1. Interpreter模式的应用场合是 Interpreter 模式应用中的难点，只有满足"业务规则频繁变化，且类似的结构不断重复出现，并且容易抽象为语法规则的问题"才适合使用 Interpreter 模式。
> 2. 使用 Interpreter 模式来表示文法规则，从而可以使用面向对象技巧来方便地"扩展"文法。
> 3. Interpreter 模式**比较适合简单的文法表示**，*对于**复杂的文法**表示 Interperter 模式会产生比较大的类层次结构，需要求助于语法分析生成器这样的标准工具。*

设计模式总结

一个目标

管理变化，提高复用

两种手段

分解 和 **抽象**

八大原则

1. **依赖倒置原则（DIP）**： 高层模块(稳定)不应该依赖于低层模块(变化)，二者都应该依赖于抽象(稳定)； 抽象(稳定)不应该依赖于实现细节(变化)，实现细节应该依赖于抽象(稳定)。
2. **开放封闭原则（OCP）**：对扩展开放，对更改封闭。 类模块应该是可扩展的，但是不可修改。
3. **单一职责原则（SRP）**： 一个类应该仅有一个引起它变化的原因。变化的方向隐含着类的责任。
4. **Liskov 替换原则（LSP）**： 子类必须能够替换它们的基类(IS-A)。继承表达类型抽象。
5. **接口隔离原则（ISP）**： 不应该强迫客户程序依赖它们不用的方法。 接口应该小而完备。

6. **优先使用对象组合，而不是类继承**：类继承通常为“白箱复用”，对象组合通常为“黑箱复用”。 继承在某种程度上破坏了封装性，子类父类耦合度高。而对象组合则只要求被组合的对象具有好定义的接口，耦合度低。
7. **封装变化点**： 使用封装来创建对象之间的分界层，让设计者可以在分界层的一侧进行修改，而不会对另一侧产生不良的影响，从而实现层次间的松耦合。
8. **针对接口编程，而不是针对实现编程**： 不将变量类型声明为某个特定的具体类，而是声明为某个接口。 客户程序无需获知对象的具体类型，只需要知道对象所具有的接口。 减少系统中各部分的依赖关系，从而实现“高内聚、松耦合”的类型设计方案。

重构技法

1. 静态 -> 动态
2. 早绑定 -> 晚绑定
3. 继承 -> 组合
4. 编译时依赖 -> 运行时依赖
5. 紧耦合 -> 松耦合

从封装变化角度对模式分类

- 组件协作：
  - Template Method
  - Strategy
  - Observer / Event
- 单一职责：
  - Decorator
  - Bridge
- 对象创建:
  - Factory Method
  - Abstract Factory
  - Prototype
  - Builder
- 对象性能：
  - Singleton
  - Flyweight
- 接口隔离:
  - Façade
  - Proxy
  - Mediator
  - Adapter
- 状态变化：
  - Memento
  - State
- 数据结构：
  - Composite
  - Iterator
  - Chain of Resposibility
- 行为变化：
  - Command
  - Visitor
- 领域问题：
  - Interpreter

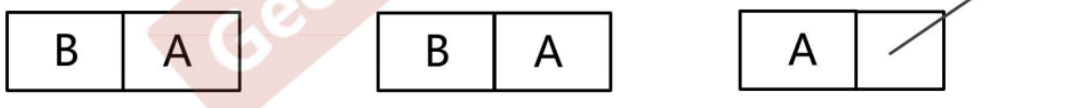
6

C++ 对象模型

```
class A: B {
    //...
};
```

```
class A {
   B b;
   //...
};
```

```
class A {
   B* pb;
   //...
};
```



7

第 3 中结构在设计模式中最常见。

关注变化点和稳定点

什么时候不用模式

1. 代码可读性很差时
2. 需求理解还很浅时
3. 变化没有显现时
4. 不是系统的关键依赖点
5. 项目没有复用价值时
6. 项目将要发布时

经验之谈

1. 不要为模式而模式
2. 关注**抽象类&接口**
3. 理清变化点和稳定点
4. 审视依赖关系
5. 要有 Framework 和 Application 的区隔思维
6. 良好的设计是演化的结果

设计模式成长之路

1. “手中无剑,心中无剑”：见模式而不知
2. "手中有剑,心中无剑”：可以识别模式，作为应用开发人员使用模式
3. “手中有剑,心中有剑”：作为框架开发人员为应用设计某些模式
4. “手中无剑,心中有剑”：忘掉模式，只有原则