# 面向对象 OO

## 概述

## 面向对象的主要思想

分而治之; 大任务【大项目，复杂需求】

软件：XX系统》子系统》组件【模块】》类

生活：1 班主任管理很多学生 ；组长 【工资】

2军队：军》师》…….

3 集团》分公司》部门

高内聚，低耦合 ：关系密切 放在同一个？ 关系不密切 放在不同的？

类 界面 UI ：NGUI，UGUI

耦合：解耦；关系

生活：分组！张三 李四 投机！ 王五 别扭 性格！

封装变化(细节)：将来通过案例说明 设计模式 实现这个目标！

写多代码 【 1行 2行 3行】、

版本1控制台：获得输入 加法运算 输出

2 页面： 修改 不变 修改

1）稳定 不稳定=变化

2）原则：把痛苦留给自己， 把方便留给别人

分析，设计 实现

实现定义端【插件，包】实现调用端

### 面向对象的设计原则【学oop之前了解 之后掌握】

把名称记住！

**1开-闭原则 （目标） ：对扩展开放，对修改关闭**

**理想化的目标！**

**需求1：写了一段代码【类，组件，系统】**

**game1 类Class1：F1，F2**

**需求2：game2 类F1，F2，F3 怎么办？**

**方法1：打开类Class1，增加一个F3**

**方法2：继承 Class2：Class1{F3}**

**接口，组合，……..**

**有的总是需要 对原来的做一定的修改！**

**移动互联网：手机 手机操作系统**

**Ios之前，微软早就有了 mobile》WP7 8**

**2类的单一职责（一个类的定义）**

**设计，抽象得到的类：功能不要太多，越少越好，**

**功能多，功能属于同一类别的功能**

**学生：上课，听课，考试，讲课【晨讲】争议？**

**老师：讲课，备课！**

一个类有且只有一个改变它的原因【少】 易复用，易维护！

**3面向接口编程而非面向实现 (写一个类时从哪入手)**

先做好一个好对外的接口（公有的方法），实现不是第一步要思考.

先思考做什么，为其他的组件提供什么功能，而不是去先思考功能的实现

**4使用组合而非继承 (复用的最佳实践)**

如果仅仅为了复用优先选择组合复用，而非继承复用。

组合的耦合性对继承低

**5依赖倒置 ( 依赖抽象)**

客户端代码尽量依赖抽象的组件，因为抽象的是稳定的。实现是多变的。

**6里氏替换 (继承后的重写)**

父类出现地方可以被子类替换掉。要保护替换前的原有的工作，在替换后依然保持不变

子类在重写父类方法时，尽量选择扩展重写。

**7接口隔离 （功能拆分）**

尽量定义小而精的接口，少定义大而全的接口

小接口之间功能隔离，实现类需要多个功能时可以选择多实现.或接口之间做继承

**8迪米特法则 (类与类交互的原则)**

不要和陌生人说话【**原则**】

类与类交互时,在满足功能要求的基础上,

传递的数据量越少越好.因为这样可能降低耦合度，便于替换 class1 一个调用 class2

1. 男孩子 伙伴
2. 软件中：方法调用少；参数越少越好 F1 F2（int）（int，string，int）

追求的目标：代码低耦合 好！便于替换

高耦合 两个对象 头 躯干 连接 脖子 复杂 人真人

低耦合 假人：玩具，芭比娃娃 换头！

### 面向对象的三大特性 C# 完全 不完全

1封装: 如何组织类或模块，让封装的类或组件，尽量只负责一个领域的工作.

2继承day2: 复用方式之一，概念形成统一。通过继承可以管理多个概念

3多态day3,4: 一个行为不同的做法。目标一致，实现的方式不同

封装: 如何组织类或模块，让封装的类或组件，尽量只负责一个领域的工作

1. What 组织代码。语句1，语句2》方法 定义方法就是在使用封装

方法，字段》类 定义类就是在使用封装【最常用】

类1，类2》组件 NGUI

组件1，组件2 Easytouch》开发包 系统

1. Why 好处：方便代码的复用和维护【万能答案】 多态，继承

类：

不同的功能封装到不同的类中，出了问题，容易查找。

具体：方便使用者使用！Math. 小车 开！

保护类，实现类的安全性稳定性！私有 共有！小车 车前盖！

1. Who/where/when：程序员 时时刻刻都在用！添加脚本 【类模板】
2. How 所有符合封装定义的做法 都叫封装【都是在使用封装】

## Day01-类与对象 封装

类： 表示一个概念,包含数据(特征)和行为(能力),对象的抽象，没有状态

1. 定义：是一组相似事物的统称 学生【同学们】 一组对象
2. 构成：字段【数据>特征】 方法【>行为，功能】

学生类：id编号 name姓名 AttendClass上课Test考试

1. 语法：

class Student

{

public int id;

public void Test()

{

}

}

4)特点：对象的抽象，没有状态【字段的取值】

多个相似性的具体的对象 抽取共性得到；

Student id？id=？

zs zs.Id=1；

对象：表示一个实例，是类的实现，拥有状态

1. 定义：对象是new类得到的结果

实例化类得到的结果

类的实例化 【一个实例】

1. 语法：类名 对象名=new 类名（）；
2. 特点 ：拥有状态 zs zs.Id=1；

关系：类对象 类是对象的模板；对象是类的实例

得到对象的目的：调用方法或字段！

类与类的四大关系

1泛化: 类与类的继承关系。耦合度最高

继承：1》泛化 多 20岁女孩 子》父 母亲 主角，怪》角色类

2》特化 20岁女孩 父》子 将来的孩子

2实现：实现类与接口之间的关系。 手机 开关 飞接口

3关联: 整体与部分的关系,另一个类作为当前类的成员出现

组合 Person{ id，name，Head，Leg}

4依赖: 类与类的协作关系,另一个类型作为当前类型的方法的

参数或返回值类型出现

Class A{ Test（Course c）}

**案例分析与设计**

1. **简要的分析，设计 了解**
2. **重点：代码实现=OO方法写代码**
3. **用到的技术：OO学到【封装，类：普通的类；特殊的类】**

**【方法：普通的方法；特殊的方法】**

**【3原则：类的单一职责】**

1需求明确：

简单ARPG:

1. 玩家控制主角可以打怪，怪受伤，受伤后可能死亡，死亡后有的可能掉落物品，有的不会掉落
2. 玩家打死怪后可以得到杀怪经验，不同怪经验不同
3. 玩家可以移动，小怪静止，玩家的移动可能通过键盘，也可能通过模拟摇杆控制，玩家在活动过程中有相应的动画播放
4. 主角可以挂机

2需求调研：ARPG 在多款ARPG研究 共性，优点，缺点，【策划没有的功能】

角色系统，成长系统，技能系统，动画系统，

运动系统，任务系统，背包系统，结算系统

3分析思路: **技巧+开发经验【项目经验】**

1. 识别对象: 技巧【从需求文字描述中找名词，确定适合做对象的名词】

1）玩家，主角，怪，物品，经验【字段】，键盘，模拟摇杆，动画

**开发经验【项目经验】 节点【树枝，树叶】**

2）最后的对象：主角，怪,物品,输入控制 ,动画

1. 分配职责：针对每个对象找共性【特征+行为】

【技巧：如何找共性的行为：找动词！】

1主角：收集经验，升级,捡物品, 主角状态(经验值，攻击力，攻击速度,生命，魔法)，

2马达:移动, 移动速度

3怪 ：受伤，掉落物品，死亡，小怪状态(贡献经验值,生命，防御……)

4物品：种类，名称，描述，图片，

5操作方式：控制移动,控制攻击

6动画系统：播放,取消,

7动画事件处理：处理方法

1. 建立交互:找关系 4大关系！ 找调用关系【开发经验】

**移动: 按下移动按钮🡪调用马达的移动方法🡪动画系统播移动动画**

Framework

**打怪:** 按下技能按钮🡪调用技能系统释放技能🡪调用动画系统播动画🡪处理动画事件

🡪调用小怪的受伤🡪可能调用死亡🡪调用主角的收集经验🡪升级

**4设计=深入分析：**

**步骤1：发现潜在的数据类型，潜在的字段，潜在的方法**

**同时运用设计原则**

**确定最终的设计结果【对象，字段，方法】**

**步骤2：确定对象的数据类型【C#：枚举，结构，类，接口，委托】**

**步骤3：确定方法的类别【普通方法，抽象方法，虚方法，重写方法，隐藏方法..】**

**用的技术：技巧+开发经验**

**设计的成果：系统的类图【类1图，类2图，接口类图】**

类设计【**系统的类图 ：文字描述；图形描述【多】！】**:

角色系统 1.0

1主角状态：PlayerStatus

数据：经验(Exp，MaxExp) 生命(HP,MaxHP) 魔法(SP,MaxSP) 伤害(Damage)

防御（Defence）攻击速度(attackSpeed) 攻击距离(attackDistance)

行为：收集经验CollectExp，升级 LevelUp 受击OnDamage 死亡Dead

2小怪状态：MonsterStatus

数据：生命(HP,MaxHP) ，魔法(SP,MaxSP) ，伤害(Damage) 防御（Defence）

攻击速度(attackSpeed) 攻击距离(attackDistance) 贡献经验值(GiveExp)

行为：受击(OnDamage) 死亡(Dead)

运动系统

3角色马达：CharacterMotor

数据：移动速度 moveSpeed,转向速度 rotationSpeed ,动画系统(chAnim),

角色控制器(chController)

行为：移动（Move），转向(LookAtTarget)

4角色输入控制：CharacterInputController （EasyTouch插件）

数据：马达(chMotor)

行为：摇杆移动执行的方法(JoystickMove)，

摇杆停止时执行的方法(JoystickMoveEnd)

动画系统

5角色动画系统：CharacterAnimation

数据：动画组件（anim）

行为：播放动画(PlayAnimation)

6 动画事件行为：AnimationEventBehaviour

数据：动画组件*（*anim）

行为: 攻击时使用（OnAttack） 撤销动画播放（OnCancelAnim）

背包系统

7游戏装备：Equipment

数据：种类（category），名称(name)，描述(description)，图片(icon)，

5实现 ：根据类图 实现代码

步骤1：【根据类图】实现类的主要结构【框架】

字段的类型暂时随意；方法的返回类型参数随意

步骤2：【根据开发经验-至少做过一次！跟着老师做！举一反三】实现类的细节：

确定字段的准确的类型，访问性

确定方法的返回类型参数，访问性

实现方法体

大部分类中包含两部分信息：数据与行为

在类的语法上，类中包含五种成员：字段，方法，【属性，索引器，事件】

在语义上，类中的字段可以表示数据，

其余成员都是用于表示行为的，方法是行为表示的典型代表

一个类的设计遵循类的单一职责的原则

## Day02-继承

### 为什么需要继承

为了保留原有的功能，通过继承可以复用，不用每次都从头开始

为了能够对类进行层次性的管理

### 继承的优点

1. 复用代码的一种方式
2. 统一概念，概念复用
3. 以层次化的方式管理类

### 继承的缺点

耦合度高：父类的改变直接影响到所有的子类，而不需要通知子类

建议：

继承层次不要太深,三层即可

尽量选择抽象类来继承

### 继承的语法

写法：class A: B 表示A类继承B类，A类称为子类（派生类）,B类称为父类（基类，超类）

父类中所定义除私有成员外都继承给子类

### 继承中的构造方法

构造方法不会继承给子类，但是在创建子类对象时，自动调用父类的构造方法，且父类构造方法先执行，子类构造方法后执行.

当子类采用无参构造方法创建对象时，默认调用父类的无参构造方法，如果父类没有无参构造方法，则报编译错误，解决方法有两个: 1.为父类添加无参构造方法, 2.在子类的构造方法中用base关键字指明要调用父类的哪一个有参构造方法

Public Dog() : base(“ssss”)

{

}

//Q:1.为什么需要构造方法？

//A: 只要类型需要创建对象，都需要构造方法，因为构造方法是创建对象的唯一通道.

//Q:2.为什么要重载构造方法？

//A:构造方法在创建对象的同时，并为对象成员做初始化，通过传递的参数为成员赋值

//Q:3.构造方法之间可不可以互相调用

//A:可以,本类构造方法互相调用通过this关键字，

子类调用父类的构造方法通过base关键字

哪种情况适合用继承组织类的关系

1. 两个或更多个类从概念上是一致的
2. 从数据和行为上是一致的
3. 会不会这几个类型统一处理

## Day03-多态开闭原则

### 怎么理解多态

多态：多种形态，对象的多种形态，行为的多种形态

体现为，子类对象可以通过父类型引用，行为的多态体现为方法的重写，隐藏，重载

继承将相关概念的共性进行抽象，并提供了一种复用的方式。

多态在共性的基础上，体现类型及行为的个性化。一个行为有多个不同的实现

### 方法隐藏

解决父类的方法在子类不适合的问题。但是父类的方法不能重写（不是virtual,abstract,override）,此时可能选择方法隐藏，子类新定义一个方法，与父类方法同签名，子类型用新定义的方法隐藏掉父类继承的旧方法。

方法隐藏时语法建议在方法前加new关键字

### 虚方法

什么是虚方法：

语法：用vritual关键修饰的已经实现的方法，即是虚方法

语义：虚方法表示一个可以在子类中重写的方法

### 方法重写

方法重写解决父类所定义方法在子类中不适用（虚方法），或父类没有实现(抽象方法)，这样的方法子类可以重写，重写是为了满足子类对该方法的不同需求.

三种方法可以重写

abstract 方法在子类必须重写，除非子类也是抽象类

virtual 方法在子类可以重写，父类方法的做法与子类不同

override方法，已经重写过的方法，在子类还可以继续重写，除非被标识为sealed

方法重写时必须在方法前加override关键字

Sealed(密封)的作用

1. 用在类的定义上，指示当前类不能做父类,也就是任何类都不可继承当前类
2. 用在重写的成员，指示当前类的子类，不能再次重写该成员

### 方法隐藏原理

方法隐藏是子类的方法表加入一个新项，新项中表示的方法签名与继承来的方法签名一致，但是在方法表中不能出现相同签名的方法项，所以必须要隐藏掉一个。子类用新项隐藏旧项。方法隐藏是在方法表加新项

### 方法重写原理

父类的方法在子类重写后，是子类型将方法表中相应方法签名对应的方法所在的地址重写了，重写后，原方法签名项关联到新方法的地址。当引用调用该方法时，访问新地址中的方法。

所以方法重写后，不管是通过父类还是子类型的引用，调用方法时，都调用对象真实类型中定义的方法

如：A是父类，B是子类，A中Show方法在子类中重写

A obj = new B(); //A类型引用指向B类型的对象

obj.Show();

此时调用的是B类型定义的Show方法.

### 动态绑定(晚期绑定)与静态绑定(早期绑定)

绑定：类型与关联的方法的调用关系，通俗讲就是一个类型能够调用哪些方法

静态绑定：是指调用关系是在运行之前确定的，即编译期间

动态绑定：是指调用关系是在运行期间确定的。

静态绑定因为在编译期确定，不占用运行时间，所以调用速度比动态绑定要快

动态绑定因为在运行期确定，占用运行时间，但是更灵活。动态绑定比静态绑定灵活性好.

方法重写是动态绑定

方法隐藏是静态绑定

### 抽象类

语法：

用abstract修饰类即为抽象类.

抽象类不能创建对象

抽象类中可能包含抽象成员(方法，属性)

语义：

抽象类表示一个抽象概念(数据和行为整体).

表达概念中拥有的共性行为数据.

抽象只表示做什么，拥有什么数据，但往往不表达具体做法，不表达数据具体取值

用于做基类，统一所有子类，管理子类.

什么时候适合用抽象类

1. 当你需要表示一个抽象概念时。
2. 当你不希望类创建对象的时候
3. 当你有行为，但是不需要实现的时候
4. 当你有多个概念，需要一致的管理时
5. 当你有一些行为，在做法上有多种可能时，但又不希望客户了解具体做法。

### 抽象方法

语法：

用abstract修饰并且没有实现的方法.只有方法声明，没有实现

抽象方法只能出现在抽象类中

抽象方法在本类中不实现，实现推迟到子类中，子类必须重写实现

语义：

抽象方法表达抽象行为,只关注本身，不关注行为实现

抽象方法描述做什么，不描述怎么做

抽象方法一个行为的抽象

## Day04-接口

### 什么是接口

使用关键字interface 创建的数据类型。接口名建议用”I”开头，其后单词首字母大写

接口是抽象的，接口是一组行为的抽象。

接口只表达“Can Do”,不表达“How To”

接口只关注行为，不关注数据，且不关注行为的实现，实现由实现类完成

接口是规范，定义一组对外的行为规范，要求它的实现类必须遵循

### 接口的作用

扩展一个已有类的行为

规范不同类型的行为,达到了不同类型在行为上是一致的

### 接口的语法特点

接口中不能包含字段，可以包含：方法，属性，索引器，事件

接口中的所有成员不能加任何访问修饰符，全部默认公有

接口中的所有成员不能有实现，全部默认抽象的

实现类实现接口用“：”与继承相同

实现类实现可以实现多个接口，且每个接口中所有的成员必须都实现

接口中的成员在实现类中以public的方式实现（除显式实现）

接口的引用可以指向实现类的对象

### 接口的使用

接口与接口之间可继承，且可以多继承

类与类是单继承，类与接口是多实现，接口与接口是多继承

结构（struct）可以实现接口，但不能继承

### 接口的显式实现

作用：

1. 解决接口中的成员对实现类不适用的问题
2. 解决多接口实现时的二义性

做法：

在实现的成员前加接口名，并且不能加任何访问修饰符,默认为private

显式实现成员只能通过接口类型的引用调用。

Void InterFace1.Fun()

{}

### Framework常用接口

IComparable 比较接口（比较器），使类型支持比大小的功能

IComparer 比较接口（比较器），提供比较的方法，常用于排序比较

IEnumerable 枚举器，使类型支持简单迭代(foreach)

IEnumerator 枚举器, 支持MoveNext ,自己可以控制迭代的节奏

## Day05-委托与事件

### 什么是委托

委托是方法的代理，委托的是方法，当调用委托时就是调用了这个方法.

委托是一类行为的抽象。是方法的引用，是一种数据类型

### 委托的使用

定义委托

delegate 返回值 方法签名

delegate void Handler();

创建实例

Handler handler = new Handler(Fun);

调用委托

handler();

Public static void Fun()

{

}

### 委托的分类

单播委托：一个委托对象只关联一个方法

多播执行：一个委托对象关联多个方法

### 委托的作用

1. 将方法做为参数传递，可以将一个方法的执行代码注入到另一个方法中
2. 实现回调，实现的方式比接口更灵活
3. 实现任意方法的异步调用。
4. 事件实现的基础

|  |  |
| --- | --- |
| 何时使用委托而不使用接口 |  |

委托和接口都允许类设计器分离类型声明和实现。任何 类或 结构都能继承和实现给定的 接口。 可以为任何类上的方法创建 委托，前提是该方法符合委托的方法签名。 接口引用或委托可由不了解实现该接口或委托方法的类的对象使用。既然存在这些相似性，那么类设计器何时应使用委托，何时又该使用接口呢？

在以下情况下，请使用委托：

* 当使用事件设计模式时。
* 当封装静态方法可取时。
* 当调用方不需要访问实现该方法的对象中的其他属性、方法或接口时。
* 需要方便的组合。
* 当类可能需要该方法的多个实现时。

在以下情况下，请使用接口：

* 当存在一组可能被调用的相关方法时。
* 当类只需要方法的单个实现时。
* 当使用接口的类想要将该接口强制转换为其他接口或类类型时。
* 当正在实现的方法链接到类的类型或标识时：例如比较方法。

#### 实例化委托多种写法

**委托的使用步骤：三步** 1：定义2：实例化委托（多种写法） 3：调用委托

**方式一、二：基本传统标准的写法,适合于: 委托已有的方法**

Handler handler = new Handler(Fun);

Handler handler = Fun;

**方式三：匿名方法,适合于:功能简单少量代码可以完成，且该功能不需要在其它地方复用.**

handler = delegate(int a)

{

for (int i = 0; i < a; i++)

Console.WriteLine("HAHA");

};

**方式四：Lambda 表达式,适合于:功能代码超简单，一行代码可以完成**

handler = (p) => Console.WriteLine(p + "HAHA");

**练习：实例化委托以下的委托， 分别使用4种方法来实例化完成**

Public void Handler();

Public int Handler(int a);

Public int Handler(Person p) //从一个对象中选一个属性 Person id或age

Public Person Handler(List< Person > list,int id)//从一个集合中选择编号=id的那个对象

### 委托实现方法的异步调用

同步调用方法：排队调用，前一个方法执行时，后一个方法等待它的结束后才能启动

异步调用方法：不排队调用，前一个方法如果异步调用，后一个方法不必等待它的结束就可启动，异步调用的方法是创建了一个新线程来工作的，与后一个方法所在不同的线程，各自独立，互不影响

Framework中的任何方法都可以异步调用。前提是通过委托引用该方法，使用BeginInvoke开启异步调用模式。

步骤：

1. 为需要异步调用的方法定义一个相应的委托
2. 创建该委托的引用指向需要异步调用的方法
3. 使用委托类型的BeginInvoke方法开始异步调用,
   1. BeginInvoke中的参数IAsyncCallback 表示异步调用的方法结束时执行的回调方法，往往用回调方法来处理异步的结果
   2. BeginInvoke中的参数object 表示在回调方法中需要用到的数据，该数据被封装在IAsyncResult的AsyncState属性中。
4. 如方法有返回值，则需要调用EndInvoke取得返回值

参考文档**使用委托进行异步编程**

<http://127.0.0.1:47873/help/1-2780/ms.help?method=page&id=38A345CA-6963-4436-9608-5C9DEFEF9C64&topicversion=100&topiclocale=ZH-CN&SQM=1&product=VS&productVersion=100&locale=ZH-CN>

使用 IAsyncResult 调用异步方法

### 什么是事件

事件语义上的特点:偶发，有影响，事件发生后，其他对象做出响应

事件模型是一个天然的通知模型

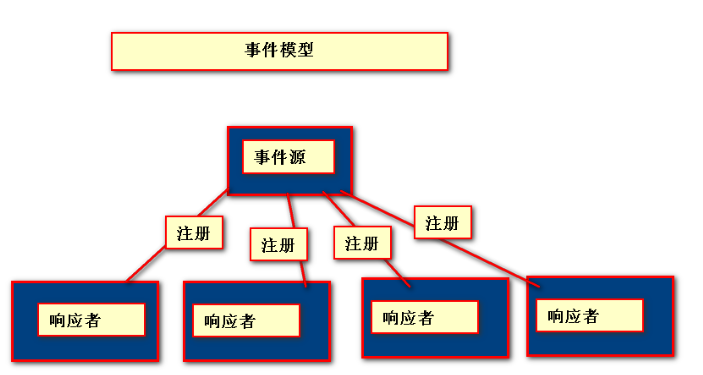
事件模型是一个天然观察者模式的实现

### 事件的使用

定义事件：public event 委托类型 事件名;

注册事件：事件源对象.事件 += 响应方法

触发事件：事件在类内部触发,触发代码同委托调用



# 类库 Framework

## Day01-泛型与集合

### 泛型

#### 什么是泛型

泛型将类型参数的概念引入 .NET Framework，类型参数使得设计如下类和方法成为可能：这些类和方法将一个或多个类型的指定推迟到客户端代码声明并实例化该类或方法的时候。

例如，通过使用泛型类型参数 T，您可以编写其他客户端代码能够使用的单个类，而不致引入运行时强制转换或装箱操作的风险或成本

泛型就是将类型做参数

#### 泛型的优点

* 1. 类型安全，避免强制类型转换
  2. 性能较好，避免装箱操作
  3. 将一个算法基于类型的不同而复用。
  4. 避免代码膨胀

#### 泛型约束

在定义泛型类时，可以对客户端代码能够在实例化类时用于类型参数(泛型)的类型种类施加限制。如果客户端代码尝试使用某个约束所不允许的类型来实例化类，则会产生编译时错误。这些限制称为约束。约束是使用 where 上下文关键字指定的。 下表列出了六种类型的约束：

|  |  |
| --- | --- |
| **约束** | **说明** |
| T：struct | 类型参数必须是值类型。可以指定除 [Nullable](http://127.0.0.1:47873/help/1-4496/ms.help?method=page&id=T%3ASYSTEM.NULLABLE&product=VS&productVersion=100&topicVersion=100&locale=ZH-CN&topicLocale=ZH-CN) 以外的任何值类型。 有关更多信息，请参见 [使用可以为 null 的类型（C# 编程指南）](http://127.0.0.1:47873/help/1-4496/ms.help?method=page&id=0BACBE72-CE15-4B14-83E1-9C14E6380C28&product=VS&productVersion=100&topicVersion=100&locale=ZH-CN&topicLocale=ZH-CN)。 |
| T：class | 类型参数必须是引用类型；这一点也适用于任何类、接口、委托或数组类型。 |
| T：new() | 类型参数必须具有无参数的公共构造函数。当与其他约束一起使用时， new() 约束必须最后指定。 |
| T：<基类名> | 类型参数必须是指定的基类或派生自指定的基类。 |
| T：<接口名称> | 类型参数必须是指定的接口或实现指定的接口。可以指定多个接口约束。约束接口也可以是泛型的。 |
| T：U | 为 T 提供的类型参数必须是为 U 提供的参数或派生自为 U 提供的参数。这称为裸类型约束。 |

#### 泛型的应用

泛型类，泛型方法，泛型接口，泛型委托，泛型集合

一句话讲，需要数据类型的地方就可以用泛型

常用泛型设计一些通用(基于不同类型的复用)的算法。

### 集合-数据结构

#### 列表：List<T> ArrayList

特点：有序，内存空间连续，读取速度快，大小可变，通过位置索引取元素.

缺点：输入，删除的速度慢

类库中已定义的列表类 ：System.Collections.ArrayList 非泛型

System.Collections.Generic.List<T> 泛型

支持的功能：添加 Add，删除 Remove RemoveAt RemoveRange，

插入 Insert ，查找 Contains，索引 this[index]，统计数量 Count，

清空 Clear

List新增的方法：Find FindAll FindIndex FindLastIndex RemoveAll

扩展方法：Sum Max Min Aveage OrderBy OrderByDescending Any All ……..

#### 字典: Dictionary<TKey,TValue>,Hashtable

特点：键值对的集合，通过键查找值，查找方便，无序

支持的操作：增加Add，删除 Remove ,清空 Clear ，检查存在ContainsKey ContainsValue

属性: 个数 Count, 键集合 Keys,值 Values

## Day02-06实战案例-ARPG 技能系统设计与实现

1. 通用工具类
   1. 数组助手类: CollectionHelper
      1. 排序:升序，降序
      2. 查找:查找单个，查找全部
      3. 选择：选取数组中对象的某些成员形成一个独立的数组
   2. 对象池: GameObjectPool //PoolManager
      1. 容器（用于存放缓存的对象）
      2. 添加
      3. 删除
      4. 取得
   3. Transform助手类: TransformHelper
      1. 自定义FindChild
      2. 转向

## Day07 程序集与反射

### 什么是程序集

程序集是 .NET Framework 应用程序的构造块；程序集构成了部署、版本控制、重复使用、激活范围控制和安全权限的基本单元。程序集是为协同工作而生成的类型和资源的集合，这些类型和资源构成了一个逻辑功能单元。程序集向公共语言运行时提供了解类型实现所需要的信息。对于运行时，类型不存在于程序集上下文之外。

### 程序集的内容

通常，静态程序集可能由以下四个元素组成：

* 程序集清单，包含程序集元数据。
* 类型元数据。
* 实现这些类型的 Microsoft 中间语言 (MSIL) 代码。
* 资源集。

### 什么是反射

反射提供了描述程序集、模块和类型的对象（ Type 类型）。 可以使用反射动态创建类型的实例，将类型绑定到现有对象，或从现有对象获取类型并调用其方法或访问其字段和属性。如果代码中使用了特性，可以利用反射来访问它们。有关更多信息，请参见 利用特性扩展元数据。

### 反射的作用

1. 了解类型信息
2. 动态创建类型实例
3. 动态访问实例的成员

### 反射开发基本流程

1. 得到数据类型
2. 动态创建对象
3. 查看类型信息（了解本身信息,成员信息）

### 反射常用类

取得数据类型Type

如何得到Type

方式一：Type.GetType(“类型全名”);

适合于类型的名称已知

方式二：obj.GetType();

适合于类型名未知，类型未知，存在已有对象

方式三：typeof(类型)

适合于已知类型

方式四：Assembly.Load(“XXX”).GetType(“名字”);

适合于类型在另一个程序集中

Type类常用Get系列方法 Is系列属性

System.Reflection 命名空间

MethodInfo(方法)

重要方法: Invoke

PropertyInfo(属性)

重要方法：SetValue GetValue

FieldInfo(字段)

重要方法：SetValue GetValue

ConstructInfo(构造方法)

重要方法：Invoke

动态创建对象

Activator.CreateInstance(string 程序集名称,string 类型全名).Unwarp()

Activator.CreateInstance(Type type);

Assembly assembly = Assembly.Load(程序集);

assembly.CreateInstance(Type);

//找到有参构造方法，动态调用构造方法

type.GetConstructor(typeof(string)).Invoke()

# 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。这样，你就能一次又一次地使用该方案而不必做重复劳动

### 创建型-关注对象创建的问题

#### SINGLETON（单件，单例） 关键词：唯一

##### 问题：

如何保证一个类只创建一个对象，且该对象全局共享

##### 意图：

保证一个类仅有一个实例，并提供一个访问它的全局访问点。

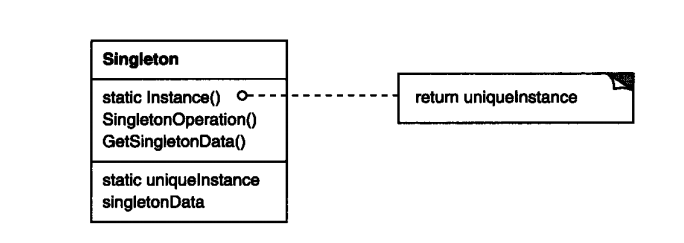
##### 适用性:

• 当类只能有一个实例而且客户可以从一个众所周知的访问点访问它时。

• 当这个唯一实例应该是通过子类化可扩展的，并且客户应该无需更改代码就能使用一个

扩展的实例时。

##### 结构:



#### SimpleFactory 简单工厂 关键字：多选一

##### 问题：

客户在使用对象时，面临在多个类型中选择一个来创建对象，具体对象的类型可能有变化

##### 意图：

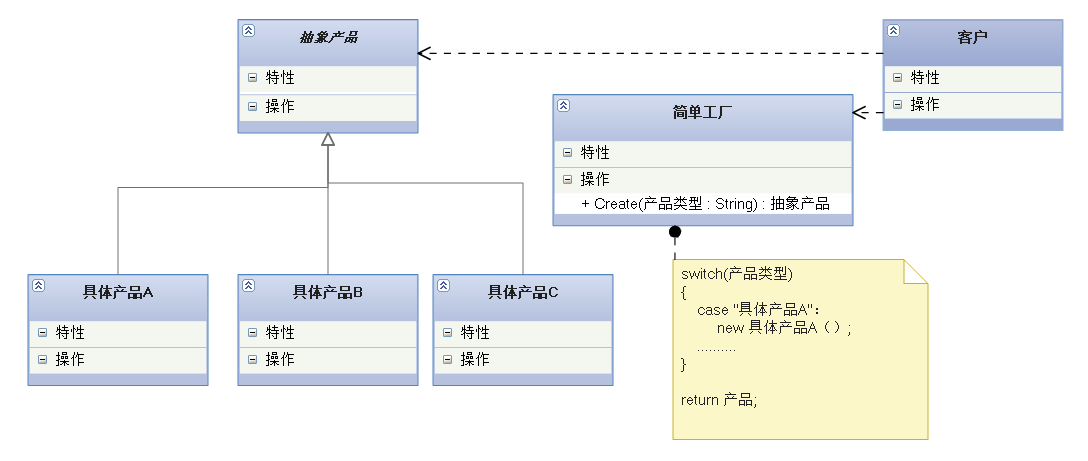
定义一个全局的工厂类，负责类型的选择及对象的创建初始化，从而实现将创建与表示分离

##### 适用性：

• 当一个类不知道它所必须创建的对象的类的时候。

• 当类将创建对象的职责委托给一个全局创建点，客户不需要关心具体类型，需要对象的时候，找全局创建点即可.

##### 结构：



#### Abstract Factory 抽象工厂 关键词：系列

##### 问题：

多个类型中以系列化的方式创建对象

##### 意图：

提供一个创建一系列相关或相互依赖对象的接口，而无需指定它们具体的类

##### 适用性：

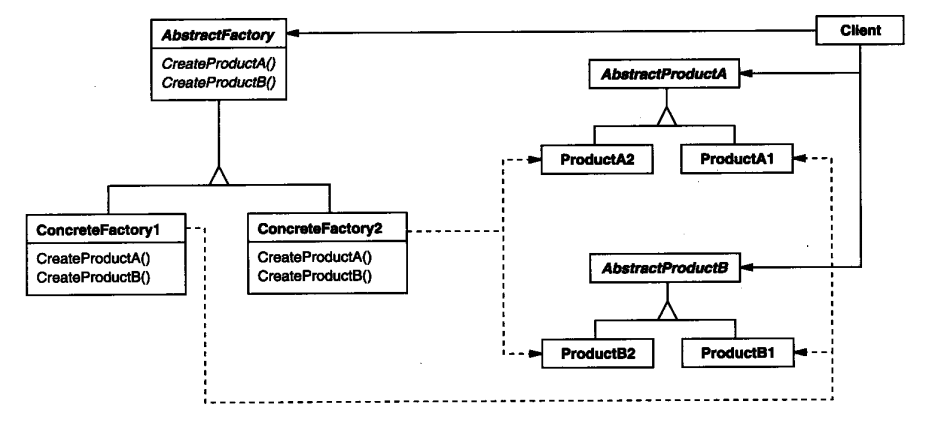
• 一个系统要独立于它的产品的创建、组合和表示时。(创建与表示分离)

• 一个系统要由多个产品系列中的一个来配置时。(多系列中选一系列)

• 当你要强调一系列相关的产品对象的设计以便进行联合使用时。(同系列中多种产品)

• 当你提供一个产品类库，而只想显示它们的接口而不是实现时。(客户端依赖抽象)

##### 结构：



#### Builder 建造器，生成器 关键词：组装

##### 变化:

复杂对象：对象中的成员可能是其它类型的对象

复杂对象的结构是稳定的，但是组成结构的具体对象是变化的

##### 问题：

复杂对象对于子部件构建的过程

##### 意图：

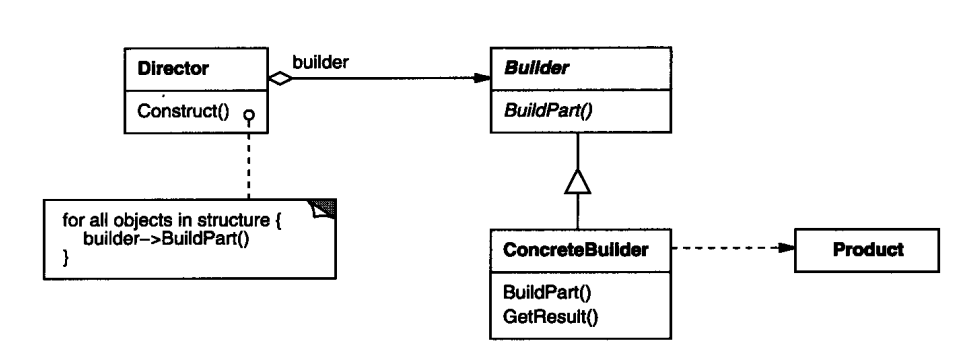
将一个复杂对象的构建与它的表示分离，使得同样的构建过程可以创建不同的表示。

##### 适用性：

• 当创建复杂对象的算法应该独立于该对象的组成部分以及它们的装配方式时。

• 当构造过程必须允许被构造的对象有不同的表示时

##### 结构：



### 结构型-关注类或对象的结构问题，组织类与类的关系

#### Facade (外观,门面) 关键词：向导

##### 问题：

客户代码访问业务代码时，需要使用多个业务对象才能实现，此时不应该让客户代码来组织访问流程，应将访问流程的组织交由外观类来实现，提供给客户一个简单的访问接口即可.

##### 意图：

为子系统中的一组接口提供一个一致的界面， F a c a d e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。

##### 适用性：

• 当你要为一个复杂子系统提供一个简单接口时。子系统往往因为不断演化而变得越来越

复杂。大多数模式使用时都会产生更多更小的类。这使得子系统更具可重用性，也更容

易对子系统进行定制，但这也给那些不需要定制子系统的用户带来一些使用上的困难。

F a c a d e可以提供一个简单的缺省视图，这一视图对大多数用户来说已经足够，而那些需

要更多的可定制性的用户可以越过 f a c a d e层。

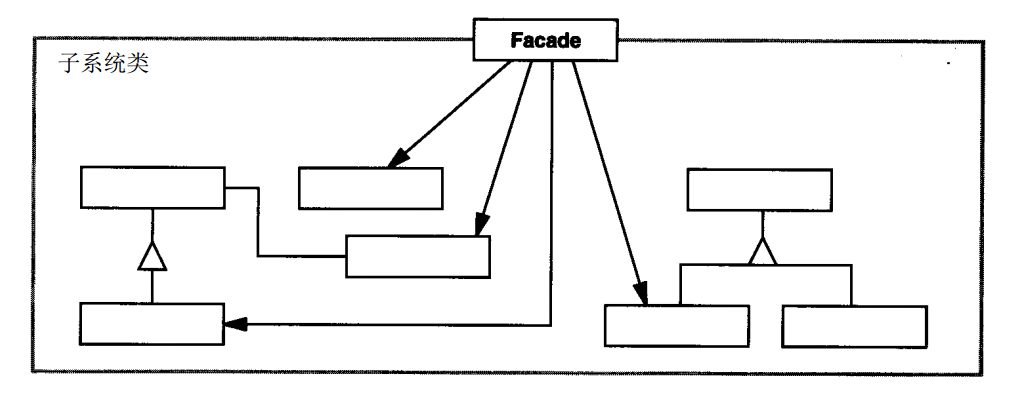
• 客户程序与抽象类的实现部分之间存在着很大的依赖性。引入 f a c a d e 将这个子系统与客户以及其他的子系统分离，可以提高子系统的独立性和可移植性。

• 当你需要构建一个层次结构的子系统时，使用 f a c a d e模式定义子系统中每层的入口点。

如果子系统之间是相互依赖的，你可以让它们仅通过 f a c a d e 进行通讯，从而简化了它们

之间的依赖关系。

##### 结构



#### ADAPTER（适配器） 关键词：转接

##### 问题：

两个功能一致的类，但是由于接口的不同不能直接复用，加入适配器做接口转换

##### 意图：

将一个类的接口转换成客户希望的另外一个接口。 A d a p t e r模式使得原本由于接口不兼容而不能一起工作的那些类可以一起工作

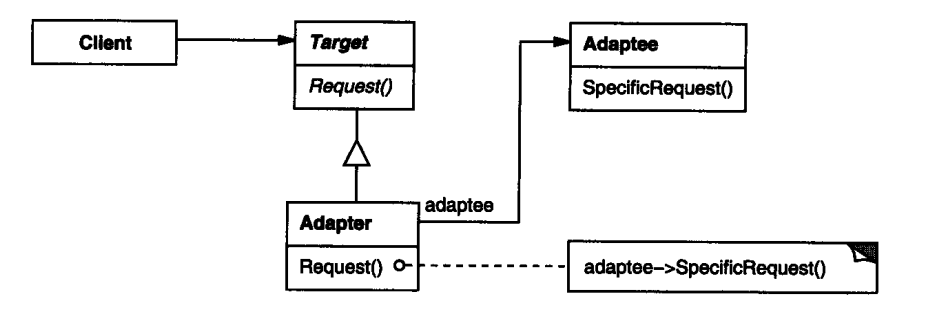
##### 适用性：

• 你想使用一个已经存在的类，而它的接口不符合你的需求。

• 你想创建一个可以复用的类，该类可以与其他不相关的类或不可预见的类（即那些接口可能不一定兼容的类）协同工作。

• （仅适用于对象 A d a p t e r）你想使用一些已经存在的子类，但是不可能对每一个都进行子类化以匹配它们的接口。对象适配器可以适配它的父类接口。

##### 结构：



#### BRIDGE（桥接）关键词:连接两端的变化

##### 问题：

主体类有变化产生不同的派生，但是主体类成员也有变化，在组合关系上可能会形成排列组合，所以将两个变化分别独立，再用关联建立主体与成员的关系，这个关联即是桥

##### 意图：

将抽象部分(主体的成员)与它的实现部分分离，使它们都可以独立地变化。

##### 适用性：

• 你不希望在抽象和它的实现部分之间有一个固定的绑定关系。 例如这种情况可能是因为，

在程序运行时刻实现部分应可以被选择或者切换。

• 类的抽象以及它的实现都应该可以通过生成子类的方法加以扩充。这时 B r i d g e模式使你可以对不同的抽象接口和实现部分进行组合，并分别对它们进行扩充。

• 对一个抽象的实现部分的修改应对客户不产生影响，即客户的代码不必重新编译。

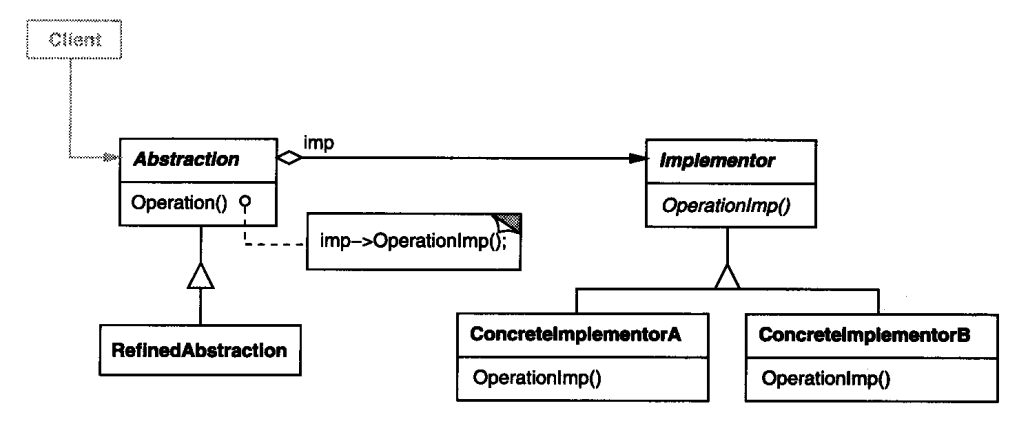
• 正如在意图一节的第一个类图中所示的那样，有许多类要生成。这样一种类层次结构说

明你必须将一个对象分解成两个部分。 R u m b a u g h称这种类层次结构为“嵌套的泛化”

（nested generalizations） 。

• 你想在多个对象间共享实现（可能使用引用计数） ，但同时要求客户并不知道这一点。

##### 结构：



#### COMPOSITE（组合）关键词：树

##### 问题：

将对象组织成树形结构

##### 意图：

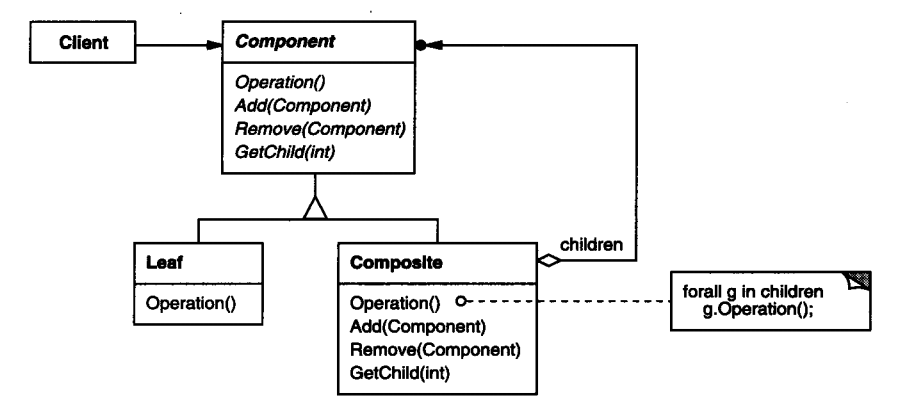
将对象组合成树形结构以表示“部分 -整体”的层次结构。 C o m p o s i t e使得用户对单个对象和组合对象的使用具有一致性。

##### 适用性：

• 你想表示对象的部分 -整体层次结构。

• 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。

##### 结构：



#### DECORATOR （装饰）关键词：包装

##### 问题：

对一个对象动态扩展原有的行为能力,不断包装，不断的扩展

##### 意图：

动态地给一个对象添加一些额外的职责。就增加功能来说， D e c o r a t o r模式相比生成子类更为灵活。

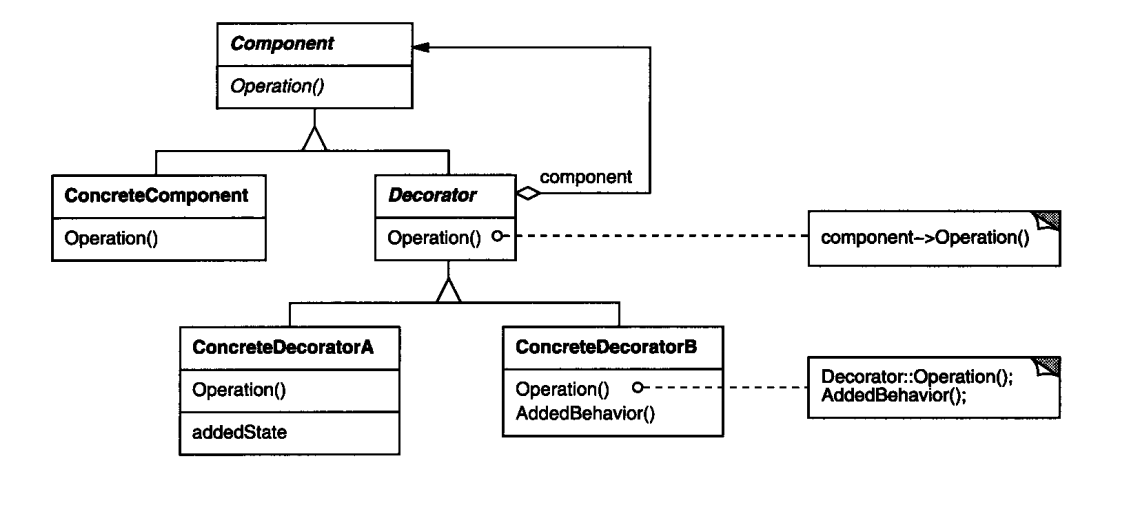
##### 适用性：

• 在不影响其他对象的情况下，以动态、透明的方式给单个对象添加职责。

• 处理那些可以撤消的职责。

• 当不能采用生成子类的方法进行扩充时。一种情况是，可能有大量独立的扩展，为支持每一种组合将产生大量的子类，使得子类数目呈爆炸性增长。另一种情况可能是因为类定义被隐藏，或类定义不能用于生成子类。

##### 结构：



#### PROXY（代理）关键词（幌子）

问题：

对一个对象进行访问控制的一个原因是为了只有在我们确实需要这个对象时才对它进行创建和初始化，在创建初始化之前，由一个小对象表示这个大对象

##### 意图:

为其他对象提供一种代理以控制对这个对象的访问。

##### 适用性:

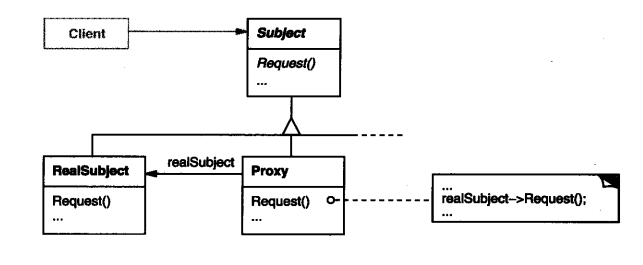
在需要用比较通用和复杂的对象指针代替简单的指针的时候，使用Proxy模式。下面是一些可以使用Proxy模式常见情况：

* 远程代理（Remote Proxy）为一个对象在不同的地址空间提供局部代表
* 虚代理（Virtual Proxy）根据需要创建开销很大的对象。
* 保护代理（Protection Proxy ）控制对原始对象的访问。保护代理用于对象应该有不同

的访问权限的时候。

* 智能指引(Smart Reference)取代了简单的指针，它在访问对象时执行一些附加操作。
* 当第一次引用一个持久对象时，将它装入内存。
* 在访问一个实际对象前，检查是否已经锁定了它，以确保其他对象不能改变它。

##### 结构:



### 行为型-关注对象交互的问题

#### TEMPLATE METHOD( 模板方法) 关键词（骨架）

##### 问题：

对于某功能实现的大流程已经确定，但是每一个步骤可能有不同的实现。在父类中定义流程（骨架）子类负责每个步骤的具体实现

##### 意图：

定义一个操作中的算法的骨架，而将一些步骤延迟到子类中。 TemplateMethod使得子类可以不改变一个算法的结构即可重定义该算法的某些特定步骤。

##### 适用性：

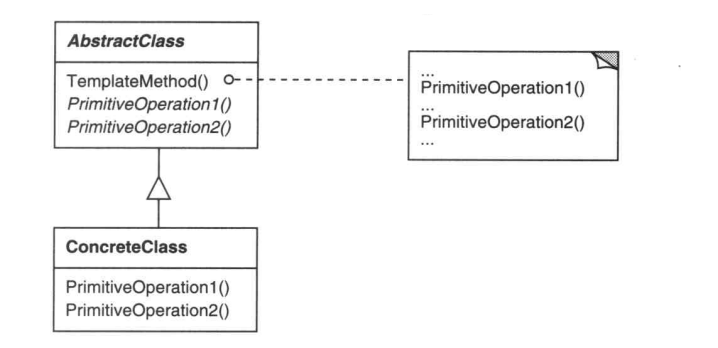
• 一次性实现一个算法的不变的部分，并将可变的行为留给子类来实现。

• 各子类中公共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。这是O p d y k e和J o h n s o n所描述过的“重分解以一般化”的一个很好的例子 [ O J 9 3 ] 。首先识别

现有代码中的不同之处，并且将不同之处分离为新的操作。最后，用一个调用这些新的操作的模板方法来替换这些不同的代码。

• 控制子类扩展。模板方法只在特定点调用“ h o o k”操作（参见效果一节） ，这样就只允许在这些点进行扩展。

##### 结构：



#### STRATEGY( 策略) 关键词(切换算法)

##### 问题：

算法是多种实现，但是实际使用中，往往选择其中之一，且可以在运行时切换不同的算法，看上去就像动态改变一个方法的执行

##### 意图：

定义一系列的算法 ,把它们一个个封装起来 ,并形成一个一致的抽象,并且使它们可相互替换。本模式使得算法可独立于使用它的客户而变化。

##### 适用性：

• 许多相关的类仅仅是行为有异。“策略”提供了一种用多个行为中的一个行为来配置一

个类的方法。

• 需要使用一个算法的不同变体。例如，你可能会定义一些反映不同的空间 / 时间权衡的

算法。当这些变体实现为一个算法的类层次时 [ H O 8 7 ] ,可以使用策略模式。

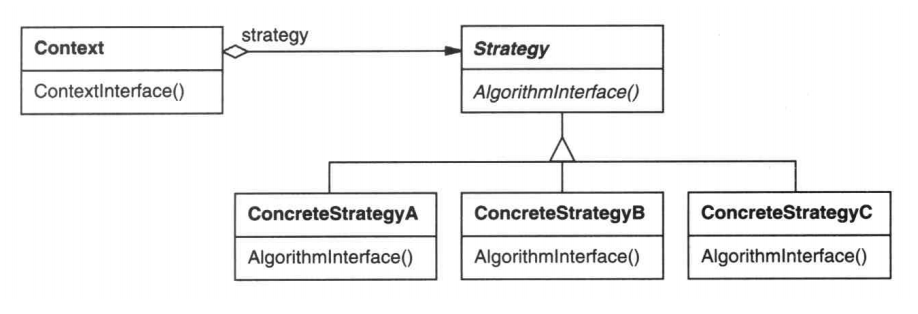
• 算法使用客户不应该知道的数据。可使用策略模式以避免暴露复杂的、与算法相关的数

据结构。

• 一个类定义了多种行为 , 并且这些行为在这个类的操作中以多个条件语句的形式出现。

将相关的条件分支移入它们各自的 S t r a t e g y类中以代替这些条件语句。

##### 结构：



#### STATE（状态）关键词：状态决定行为

##### 问题：

对象状态的改变，从而带来行为上的变化。

##### 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

##### 适用性：

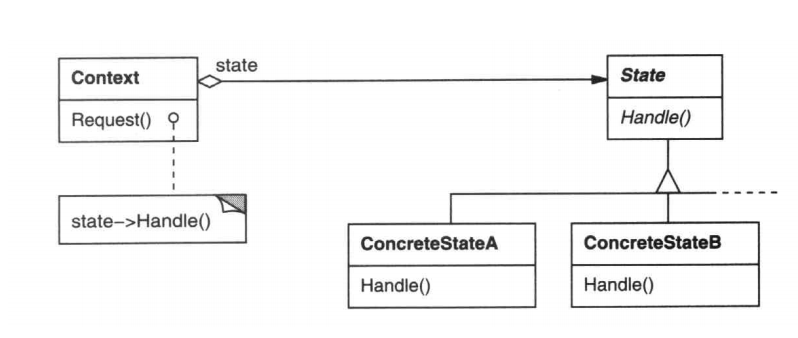
• 一个对象的行为取决于它的状态 , 并且它必须在运行时刻根据状态改变它的行为。

• 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。这个状

态通常用一个或多个枚举常量表示。通常 , 有多个操作包含这一相同的条件结构。 S t a t e模式将每一个条件分支放入一个独立的类中。这使得你可以根据对象自身的情况将对

象的状态作为一个对象，这一对象可以不依赖于其他对象而独立变化

##### 结构：



#### OBSERVER（观察者）

##### 问题：

一个对象的改变，及时的将通知广播给其他关注它的对象，(一对多的通知)

##### 意图：

定义对象间的一种一对多的依赖关系 ,当一个对象的状态发生改变时 , 所有依赖于它的对象都得到通知并被自动更新

##### 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方面。将这二者封装在独立的对象中以使它们可以各自独立地改变和复用。

• 当对一个对象的改变需要同时改变其它对象 , 而不知道具体有多少对象有待改变。

• 当一个对象必须通知其它对象，而它又不能假定其它对象是谁。换言之 , 你不希望这些对象是紧密耦合的。

##### 结构：