



应用软件通用设计和开发规范

<div>文件状态：</div> <div><input type="checkbox"/> 草稿</div> <div><input type="checkbox"/> 正式发布</div> <div><input checked="" type="checkbox"/> 正在修改</div>	文件编号：	JUSHASW001
	当前版本：	V0.2
	作 者：	王文鼎
	完成日期：	2018 年 5 月 25 日

编制：

日期：

审核：

日期：

批准：

日期：

版 本 历 史

版本 /状态	作 者	参 与者	完成日期	备注
V0.1	王 文鼎	李 贯涛	2018 年 08 月 15 日	初始版
V0.2	王 文鼎		2018 年 08 月 25 日	初评修正 版

目录

目录.....	3
一、软件项目的设计原则.....	8
■ 1.软件项目	8
■ 2.软件项目设计入口条件.....	8
■ 3.软件项目的设计原则	8
(1)可靠性.....	8
(2)健壮性.....	9
(3)可修改性	9
(4)容易理解	9
(5)程序简便	10
(6)可测试性	10
(7)效率性.....	10
(8)标准化原则.....	10
(9)先进性.....	10
(10)可扩展性	10
(11)安全性	11
二、软件架构设计思想及设计准则.....	12
■ 面向对象软件架构设计原则.....	12
■ 如何对项目的规模进行划分.....	14
■ 如何对不同规模的软件做相应的设计.....	15
■ 基于面向对象设计的六大原则进行设计	15

单一原则	15
开闭原则	17
里氏替换原则	18
依赖倒置原则	20
接口隔离原则	22
迪米特原则	22
■ 基于面向对象软件 23 大设计模式进行设计参考	23
1.单例模式（Singleton Pattern）	23
2.工厂模式	24
3.抽象工厂模式（Abstract Factory Pattern）	25
4.模板方法模式（Template Method Pattern）	26
5.建造者模式（Builder Pattern）	26
6.代理模式（Proxy Pattern）	27
7.原型模式（Prototype Pattern）	28
8.中介者模式	29
9.命令模式	30
10.责任链模式	31
11.装饰模式（Decorator Pattern）	32
12.策略模式（Strategy Pattern）	33
13.适配器模式（Adapter Pattern）	34
14.迭代器模式（Iterator Pattern）	35
15.组合模式（(Composite Pattern)）	35

16.观察者模式 (Observer Pattern)	36
17.门面模式 (Facade Pattern)	38
18.备忘录模式 (Memento Pattern)	38
19.访问者模式 (Visitor Pattern)	41
20.状态模式 (State Pattern)	41
21.解释器模式 (Interpreter Pattern)	42
22.享元模式 (Flyweight Pattern)	43
23.桥梁模式 (Bridge Pattern)	44
三、C#/Java 代码规范	46
■ 代码设计准则	46
重用性	46
可维护性	46
模块化	46
书写注释	47
保持代码风格的一致性	47
命名需要有意义	47
保持可读性	47
■ 命名规范	48
大小写约定	48
通用命名约定	51
程序集和 DLL 的命名	53
命名空间的命名	54

类、结构和接口的命名	55
类成员的命名	57
■ 类设计规范	59
在类和结构之间选择	59
抽象类设计	60
静态类设计	61
接口设计	61
■ 成员设计规范	62
属性设计	62
构造函数设计	63
事件设计	63
字段设计	64
三、软件项目版本号的格式规定	64
■ 对外部版本号的格式规定	64
■ 对内部版本号的格式规定	65
四、软件项目变更版本的原则	66
■ 对外部版本变更的规定	66
初版（V1.0.0）的设置定义：	66
大版本（V1.X.X 到 V2.0.0）的设置定义：	66
升级（V2.0.X 到 V2.1.0）的设置定义：	66
补丁（V2.1.0 到 V2.1.1）的设置定义：	67
■ 对内部版本变更的规定	67

五、交付件的定义	68
六、附录.....	69
■ 软件项目需求说明书模板.....	69
■ 软件规格书模板.....	71
■ 软件概要设计书模板.....	71
■ 软件自测清单模板	73
■ 软件变更清单模板	74

一、软件项目的设计原则

1. 软件项目

软件项目是以达成项目需求为目的，在操作系统之上运行的系统，以实现一系列规定动作之后得到用户想要的结果，有时也需要连接硬件系统并进行特定任务和特定操作。

2. 软件项目设计入口条件

软件项目的设计入口条件是应当有明确的项目需求（或项目需求书）作为输入。

3. 软件项目的设计原则

软件系统在实现项目需求的同时，为了提高项目的持续升级和持续维护，还要兼顾行业内一系列约定的俗成的设计原则，从而保证项目的持续改进及规范化。

(1) 可靠性

软件可靠性是指：

(1) 在规定的条件下，在规定的时间内，软件不引起系统失效的概率；

(2) 在规定的時間周期內，在所述條件下程序執行所要求的功能的能力；

這意味著該軟件在測試運行過程中避免可能發生故障的能力，且一旦發生故障後，具有解脫和排除故障的能力。

(2)健壯性

健壯性又稱魯棒性，是指軟件對於规范要求以外的輸入能夠判斷出這個輸入不符合规范要求，並能有合理的處理方式。軟件健壯性是一個比較模糊的概念，但是卻是非常重要的軟件外部量度標準。軟件設計的健壯與否直接反應了分析設計和編碼人員的水平。

(3)可修改性

要求以科學的方法設計軟件，使之有良好的結構和完備的文檔，系統性能易于調整。

(4)容易理解

軟件的可理解性是其可靠性和可修改性的前提。它並不僅僅是文檔清晰可讀的問題，更要求軟件本身具有簡單明了的結構。這在很大程度上取決於設計者的洞察力和創造性，以及對設計對象掌握得透徹程度，當然它還依賴於設計工具和方法的適當運用。

(5)程序简便

(6)可测试性

可测试性就是设计一个适当的数据集合，用来测试所建立的系统，并保证系统得到全面的检验。

(7)效率性

软件的效率性一般用程序的执行时间和所占用的内存容量来度量。在达到原理要求功能指标的前提下，程序运行所需时间愈短和占用存储容量愈小，则效率愈高。

(8)标准化原则

在软件的结构上实现通用设计，基于业界开放式标准，符合国家和信息产业部的规范。

(9)先进性

满足客户需求，系统性能可靠，易于维护。

(10)可扩展性

软件设计完要留有升级接口和升级空间。对扩展开放，对修改关闭。

(11) 安全性

安全性要求系统能够保持用户信息、操作等多方面的安全要求，同时系统本身也要能够及时修复、处理各种安全漏洞，以提升安全性能。

二、软件架构设计思想及设计准则

面向对象软件架构设计原则

设计面向对象的软件比较困难，而设计可复用的面向对象软件就更加困难。你必须找到相关的对象，以适当的粒度将它们归类，再定义类的接口和继承层次，建立对象之间的基本关系。

你的设计应该对手头的问题有针对性，并且要预见未来的可能出现的变化，对未来的问题和需求也要有足够的通用性。你也希望避免重复性和灵活性的设计，即使不是不可能的至少也是非常困难的。一个设计在最终完成之前常常要被复用好几次，而且每一次都有所修改。

有经验的面向对象设计者的确能够做出良好的设计，而新手则面对众多选择无从下手，总是求助于以前使用过的非面向对象技术。新手需要花费较长的时间领会良好的面向对象设计是怎么回事。有经验的设计者显然知道一些新手不知道的东西，这有时什么呢？

内行的设计者知道：不是解决任何问题都要从头做起。他们更愿意复用以前使用过的解决方案。当找到一个好的解决方案，他们会一遍又一遍地使用。这些经验是他们成为内行的部分原因。因此，你会在许多面向对象系统中看到类和相互通信的对象的重复模式。这些模式解决特定的设计问题，使用面向对象设计更灵活、优雅，最终复用性更好。它们帮助设计者将新的设计建立在以往工作的基础上，复用以往成功的设计方案。一个熟悉这些模式的设计者不需要再去发现它们，而能够立即将它们应用于设计问题中。

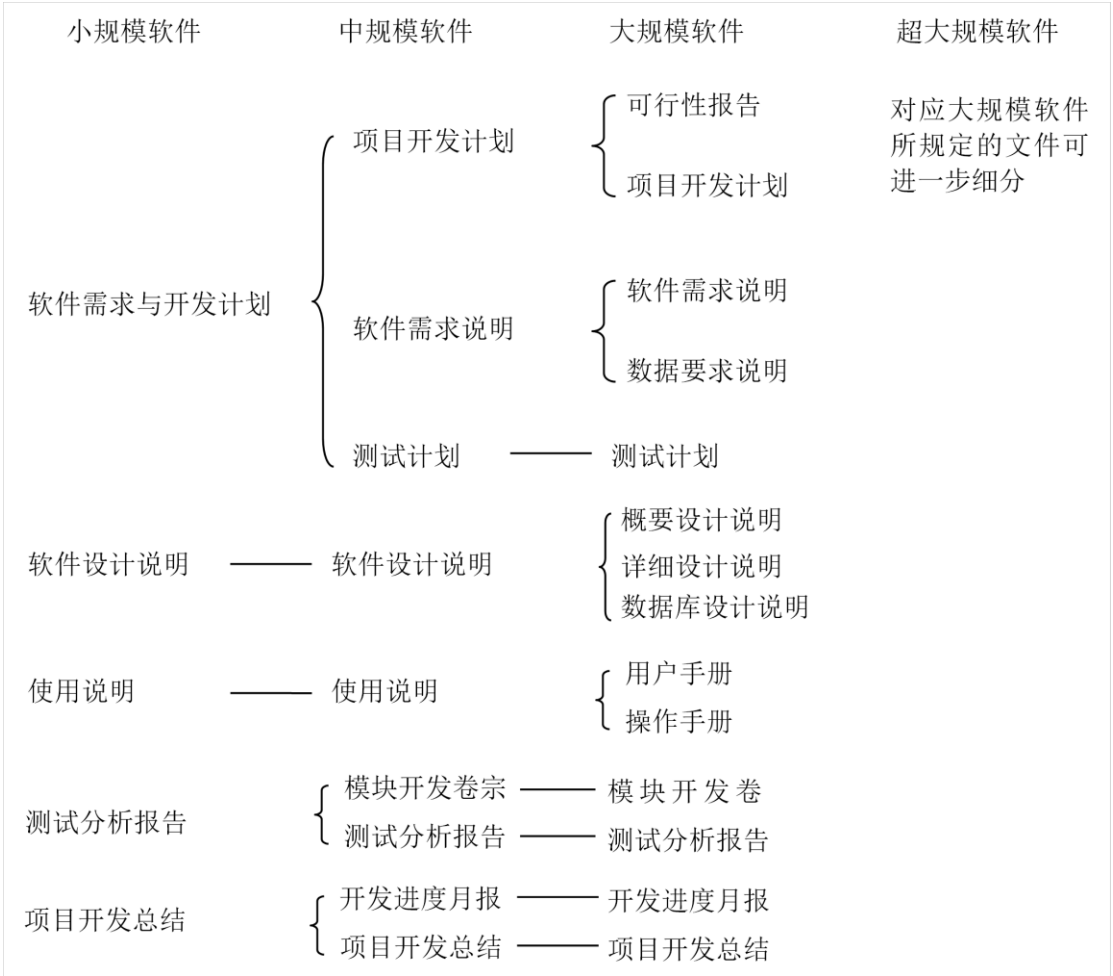
对我们目前软件项目具有指导意义的是：如何对抽象和封装的粒度与可能出现的变化之间的平衡。一切以功能和性能上的可靠性、开发和维护的便捷性来考虑此事。

所以，作为软件的设计者，要注意两个问题：

- 1.要预见未来的可能出现的变化；
- 2.以适当的粒度将代码进行抽象和封装；

因此，我们要衡量以上两点以及后续一系列需要考虑的设计，需要首先对项目的规模进行评估。

如何对项目的规模进行划分



将软件项目按业务量划分为：小规模软件、中规模软件、大规模软件、超大规模软件。

- 例子：
- 小规模软件：色温配对工具、3Dlut 计算器、Downloader...
- 中规模软件：QC、CGA、显示专家、菜单控制软件...
- 大规模软件：RC8610、WEBQA3.0...
- 超大规模软件：巨鲨云

如何对不同规模的软件做相应的设计

将不同的设计思想应用于不同规模的软件，是比较合理和可靠的。

例如，没有必要对一个小规模软件进行太多的抽象和复用性封装，因为它的业务规模比较单一，并且很可能是一个短期的项目，没有必要对其进行过多复杂的设计，虽然对能很好的完成功能，但是做了很多无用的设计，消耗了开发资源。

例如，如果一个中、大、超规模的软件项目，没有较好的设计搭建思路，会导致后期的维护成本上升、后期的扩展性不够，并且甚至可能对功能产生影响。

所以，我们约定：

对定性为小规模软件项目：侧重于功能和性能的实现，而代码的架构需要以简单为主，不需要做抽象至多层(大于三层)的类设计。

对定性为中、大、超大规模软件项目：要学会关注功能逻辑的“分离点”，在可以识别到的“分离点之上进行一至多层的封装”，以达到层级间的解耦和代码复用，提升软件的可维护性。

基于面向对象设计的六大原则进行设计

单一原则

定义：一个合理的类，应该仅有一个引起它变化的原因。

实体与行为应该分为两个类，例如：

```
1.  /// <summary>
2.  /// 苹果实体类
3.  /// </summary>
4.  class Apple
5.  {
6.      /// <summary>
7.      /// 苹果的形状
8.      /// </summary>
9.      public string Shape { get; set; }
10.
11.     /// <summary>
12.     /// 苹果的颜色
13.     /// </summary>
14.     public string Color { get; set; }
15.
16.     /// <summary>
17.     /// 苹果的重量
18.     /// </summary>
19.     public double Weight { get; set; }
20. }
21.
22. /// <summary>
23. /// 吃苹果
24. /// </summary>
25. class EatApple
26. {
27.     /// <summary>
28.     /// 直接吃苹果
29.     /// </summary>
30.     /// <param name="apple">要吃的苹果</param>
31.     /// <returns>吃完后苹果的重量</returns>
32.     public double DirectlyEat(Apple apple) { return apple.Weight / 10; }
33.
34.     /// <summary>
35.     /// 切开吃
36.     /// </summary>
37.     /// <param name="apple">要吃的苹果</param>
38.     /// <returns>吃完苹果的重量</returns>
39.     public double CutAndEat(Apple apple) { return apple.Weight / 5; }
40.
41.     /// <summary>
42.     /// 喂狗吃
43.     /// </summary>
44.     /// <param name="apple">要吃的苹果</param>
```



```
45.    /// <returns>吃完苹果的重量</returns>
46.    public double FeedTheDog(Apple apple) { return apple.Weight; }
47. }
```

Apple 类是对苹果的描述，因此仅仅苹果自身属性的变化（红的，青的）才会引起 **Apple** 类的变化。

EatApple 类是对苹果吃法的描述，因此仅有吃的方式才会引发 **EatApple** 类的变化。

开闭原则

定义：软件中的对象（类、模块、函数等）应该对于扩展是开放的，但是，对于修改是封闭的。

同上，如果 **EatApple** 中 **DirectlyEat** 方法存在优化，应该如下处理：

```
1.  /// <summary>
2.  /// 吃苹果
3.  /// </summary>
4.  class EatApple
5.  {
6.      /// <summary>
7.      /// 直接吃苹果
8.      /// </summary>
9.      /// <param name="apple">要吃的苹果</param>
10.     /// <returns>吃完后苹果的重量</returns>
11.     [Obsolete]
12.     public double DirectlyEat(Apple apple) { return apple.Weight / 10; }
13.
14.     /// <summary>
15.     /// 直接吃苹果(2.0 版)
16.     /// </summary>
17.     /// <param name="apple">要吃的苹果</param>
18.     /// <returns>吃完后苹果的重量</returns>
19.     public double DirectlyEat_v2(Apple apple) { return apple.Weight / 9; }
20. }
```

```
21.    /// <summary>
22.    /// 切开吃
23.    /// </summary>
24.    /// <param name="apple">要吃的苹果</param>
25.    /// <returns>吃完苹果的重量</returns>
26.    public double CutAndEat(Apple apple) { return apple.Weight / 5; }
27.
28.    /// <summary>
29.    /// 喂狗吃
30.    /// </summary>
31.    /// <param name="apple">要吃的苹果</param>
32.    /// <returns>吃完苹果的重量</returns>
33.    public double FeedTheDog(Apple apple) { return apple.Weight; }
34. }
```

里氏替换原则

定义：所有引用基类的地方必须能透明地使用其子类的对象。

同样以 **Apple** 类为例，我们为其抽象出一个基类如下：

```
1.    /// <summary>
2.    /// 苹果实体类
3.    /// </summary>
4.    class Apple : Fruit
5.    {
6.        /// <summary>
7.        /// 是否红富士苹果
8.        /// </summary>
9.        public bool IsFUJI { get; set; }
10.   }
11.
12.   /// <summary>
13.   /// 水果的实体类
14.   /// </summary>
15.   class Fruit
16.   {
17.       /// <summary>
18.       /// 苹果的形状
19.       /// </summary>
20.       public string Shape { get; set; }
21.
22.       /// <summary>
```

```
23.    /// 苹果的颜色
24.    /// </summary>
25.    public string Color { get; set; }
26.
27.    /// <summary>
28.    /// 苹果的重量
29.    /// </summary>
30.    public double Weight { get; set; }
31. }
32.
33. /// <summary>
34. /// 吃苹果
35. /// </summary>
36. class EatApple
37. {
38.    /// <summary>
39.    /// 直接吃苹果
40.    /// </summary>
41.    /// <param name="apple">要吃的苹果</param>
42.    /// <returns>吃完后苹果的重量</returns>
43.    [Obsolete]
44.    public double DirectlyEat(Fruit apple) { return apple.Weight / 10; }
45.
46.    /// <summary>
47.    /// 直接吃苹果(2.0 版)
48.    /// </summary>
49.    /// <param name="apple">要吃的苹果</param>
50.    /// <returns>吃完后苹果的重量</returns>
51.    public double DirectlyEat_v2(Fruit apple) { return apple.Weight / 9; }
52.
53.    /// <summary>
54.    /// 切开吃
55.    /// </summary>
56.    /// <param name="apple">要吃的苹果</param>
57.    /// <returns>吃完苹果的重量</returns>
58.    public double CutAndEat(Fruit apple) { return apple.Weight / 5; }
59.
60.    /// <summary>
61.    /// 喂狗吃
62.    /// </summary>
63.    /// <param name="apple">要吃的苹果</param>
64.    /// <returns>吃完苹果的重量</returns>
65.    public double FeedTheDog(Fruit apple) { return apple.Weight; }
66. }
```

我们 `EatApple` 类中的所有方法的参数类型都是 `Apple` 的父类 `Fruit` 的，在此处所有的 `Fruit` 的派生类（例如 `Apple`）的实例代替父类实体传入方法也是允许的，那么这种设计就是符合里氏原则的。

依赖倒置原则

定义：一种特定的解耦形式，使得高层次的模块不依赖于低层次的模块的实现细节的目的。

依赖倒置原则的几个关键点：

- （1）高层模块不应该依赖低层模块，两者都应该依赖其抽象；
- （2）抽象不应该依赖细节；
- （3）细节应该依赖抽象。

同样以上面的案例，我们拿到一个苹果，准备吃它，那么我们这样调用

```
1. Apple apple = new Apple();  
2. EatApple eatApple = new EatApple();  
3. eatApple.DirectlyEat_v2(apple);
```

如果不想吃苹果了怎么办，如下我们需要进行改动。

```
1. Pear pear= new Pear();  
2. EatPear eatPear = new EatPear();  
3. eatPear.DirectlyEat_v2(pear);
```

如上所示，上层调用方的代码机会需要完全变动。如果我们期望框架的稳定性，添加新的水果和新的吃的方式后不变动框架代码，那么需要为这两个类抽象出 2 个接口：

```
1. interface IEat
2. {
3.     double DirectlyEat_v2(IFruit fruit);
4. }
5. interface IFruit
6. {
7.     string Shape { get; set; }
8.
9.     string Color { get; set; }
10.
11.    double Weight { get; set; }
12. }
```

调用方式将变为

```
1. IFruit fruit= new Pear();
2. IEat eat = new EatPear();
3. eat.DirectlyEat_v2(fruit);
```

我们 **DirectlyEat** 方法已经不再依赖于具体对象，也就是说我把梨换成草莓，换成西瓜，`eat.DirectlyEat_v2(fruit);`方法都不需要变动。但是创建这两个接口的对象依然需要变化。因此我们还需要一个反射创建实例的工厂。如下：

```
1. IFruit fruit= FruitFactory.CreateFruit();
2. IEat eat = EatFactory.CreateEat();
3. eat.DirectlyEat_v2(fruit);
```

```
1. class EatFactory
2. {
3.     public static IEat CreateEat()
4.     {
5.         return Assembly.Load("EatAssembly").AppSetting().CreateInstance("EatType".AppSetting()) as IEat;
6.     }
7. }
8.
```

```
9. class FruitFactory
10. {
11.     public static IFruit CreateFruit()
12.     {
13.         return Assembly.Load("FruitAssembly").AppSetting().CreateInstance(
            "FruitType".AppSetting()) as IFruit;
14.     }
15. }
```

如此，我们将框架内对 **Fruit** 以及 **Eat** 的依赖基本解耦，如果吃什么水果发生改变仅需要配置文件中的关键字（前提是 **Fruit** 模块以及 **Eat** 模块能提供支持），框架代码对 **Fruit** 以及 **Eat** 模块的依赖就完成了倒置，统一对抽象的依赖。

接口隔离原则

定义：类间的依赖关系应该建立在最小的接口上。

迪米特原则

定义：一个对象应该对其他对象有最少的了解。


通俗地讲，一个类应该对自己需要耦合或调用的类知道得最少，类的内部如何实现、如何复杂都与调用者或者依赖者没关系，调用者或者依赖者只需要知道他需要的方法即可。类与类之间的关系越密切，耦合度越大，当一个类发生改变时，对另一个类的影响也越大。

基于面向对象软件 23 大设计模式进行设计参考


1. 单例模式（Singleton Pattern）

定义：Ensure a class has only one instance, and provide a global point of access to it.（确保某一个类只有一个实例，而且自行实例化并向整个系统提供这个实例。）

通用代码：（是线程安全的）




```
public class Singleton {
    private static final Singleton singleton = new Singleton();
    //限制产生多个对象
    private Singleton() {
    }
    //通过该方法获得实例对象
    public static Singleton getSingleton() {
        return singleton;
    }
    //类中其他方法，尽量是 static
    public static void doSomething() {
    }
}
```



使用场景：

- 要求生成唯一序列号的环境；
- 在整个项目中需要一个共享访问点或共享数据，例如一个 Web 页面上的计数器，可以不用把每次刷新都记录到数据库中，使用单例模式保持计数器的值，并确保是线程安全的；
- 创建一个对象需要消耗的资源过多，如要访问 IO 和数据库等资源；
- 需要定义大量的静态常量和静态方法（如工具类）的环境，可以采用单例模式（当然，也可以直接声明为 static 的方式）。

线程不安全实例：



```
public class Singleton {
    private static Singleton singleton = null;
    //限制产生多个对象
    private Singleton() {
    }
    //通过该方法获得实例对象
```

```
public static Singleton getSingleton() {  
    if (singleton == null) {  
        singleton = new Singleton();  
    }  
    return singleton;  
}
```

解决办法:

在 `getSingleton` 方法前加 `synchronized` 关键字, 也可以在 `getSingleton` 方法内增加 `synchronized` 来实现。最优的办法是如通用代码那样写。

2. 工厂模式

定义: Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses. (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。工厂方法使一个类的实例化延迟到其子类。)

`Product` 为抽象产品类负责定义产品的共性, 实现对事物最抽象的定义;

`Creator` 为抽象创建类, 也就是抽象工厂, 具体如何创建产品类是由具体的实现工厂 `ConcreteCreator` 完成的。

具体工厂类代码:

```
public class ConcreteCreator extends Creator {  
    public <T extends Product> T createProduct(Class<T> c) {  
        Product product = null;  
        try {  
            product =  
(Product) Class.forName(c.getName()).newInstance();  
        } catch (Exception e) {  
            // 异常处理  
        }  
        return (T) product;  
    }  
}
```


简单工厂模式：

一个模块仅需要一个工厂类，没有必要把它产生出来，使用静态的方法

多个工厂类：

每个人种(具体的产品类)都对应了一个创建者，每个创建者独立负责创建对应的产品对象，非常符合单一职责原则

代替单例模式：

单例模式的核心要求就是在内存中只有一个对象，通过工厂方法模式也可以只在内存中生产一个对象

延迟初始化：

ProductFactory 负责产品类对象的创建工作，并且通过 prMap 变量产生一个缓存，对需要再次被重用的对象保留

使用场景：jdbc 连接数据库，硬件访问，降低对象的产生和销毁

3.抽象工厂模式（Abstract Factory Pattern）

定义：Provide an interface for creating families of related or dependent objects without specifying their concrete classes.（为创建一组相关或相互依赖的对象提供一个接口，而且无须指定它们的具体类。）

抽象工厂模式通用类图：**抽象工厂模式通用源码类图：****抽象工厂类代码：**

```
public abstract class AbstractCreator {  
    //创建 A 产品家族  
    public abstract AbstractProductA createProductA();  
    //创建 B 产品家族  
    public abstract AbstractProductB createProductB();  
}
```

使用场景：

一个对象族（或是一组没有任何关系的对象）都有相同的约束。

涉及不同操作系统的时候，都可以考虑使用抽象工厂模式

4.模板方法模式（Template Method Pattern）

定义：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.（定义一个操作中的算法的框架，而将一些步骤延迟到子类中。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。）

AbstractClass 叫做**抽象模板**，它的方法分为两类：

- 基本方法

基本方法也叫做基本操作，是由子类实现的方法，并且在模板方法被调用。

- 模板方法

可以有一个或几个，一般是一个具体方法，也就是一个框架，实现对基本方法的调度，完成固定的逻辑。

注意： 为了防止恶意的操作，一般模板方法都加上 **final** 关键字，不允许被覆写。

具体模板：ConcreteClass1 和 ConcreteClass2 属于具体模板，实现父类所定义的一个或多个抽象方法，也就是父类定义的基本方法在子类中得以实现

使用场景：

- 多个子类有公有的方法，并且逻辑基本相同时。

- 重要、复杂的算法，可以把核心算法设计为模板方法，周边的相关细节功能则由各个子类实现。

- 重构时，模板方法模式是一个经常使用的模式，把相同的代码抽取到父类中，然后通过钩子函数（见“模板方法模式的扩展”）约束其行为。

5.建造者模式（Builder Pattern）

定义：Separate the construction of a complex object from its representation so that the same construction process can create different representations.（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。）

- Product 产品类

通常是实现了模板方法模式，也就是有模板方法和基本方法，例子中的 BenzModel 和 BMWModel 就属于产品类。

- Builder 抽象建造者

规范产品的组建，一般是由子类实现。例子中的 CarBuilder 就属于抽象建造者。

- ConcreteBuilder 具体建造者

实现抽象类定义的所有方法，并且返回一个组建好的对象。例子中的 BenzBuilder 和 BMWBuilder 就属于具体建造者。

- Director 导演类

负责安排已有模块的顺序，然后告诉 Builder 开始建造

使用场景：

- 相同的方法，不同的执行顺序，产生不同的事件结果时，可以采用建造者模式。
- 多个部件或零件，都可以装配到一个对象中，但是产生的运行结果又不相同时，则可以使用该模式。
- 产品类非常复杂，或者产品类中的调用顺序不同产生了不同的效能，这个时候使用建造者模式非常合适。

建造者模式与工厂模式的不同：

建造者模式最主要的功能是基本方法的调用顺序安排，这些基本方法已经实现了，顺序不同产生的对象也不同；

工厂方法则重点是创建，创建零件是它的主要职责，组装顺序则不是它关心的。

6.代理模式（Proxy Pattern）

定义：Provide a surrogate or placeholder for another object to control access to it.（为其他对象提供一种代理以控制对这个对象的访问。）

- Subject 抽象主题角色

抽象主题类可以是抽象类也可以是接口，是一个最普通的业务类型定义，无特殊要求。

- RealSubject 具体主题角色

也叫做被委托角色、被代理角色。它才是冤大头，是业务逻辑的具体执行者。

- Proxy 代理主题角色

也叫做委托类、代理类。它负责对真实角色的应用，把所有抽象主题类定义的方法限制委托给真实主题角色实现，并且在真实主题角色处理完毕前后做预处理和善后处理工作。

普通代理和强制代理：

普通代理就是我们想知道代理的存在，也就是类似的 `GamePlayerProxy` 这个类的存在，然后才能访问；

强制代理则是调用者直接调用真实角色，而不用关心代理是否存在，其代理的产生是由真实角色决定的。

普通代理：

在该模式下，调用者只知代理而不用知道真实的角色是谁，屏蔽了真实角色的变更对高层模块的影响，真实的主题角色想怎么修改就怎么修改，对高层次的模块没有任何的影响，只要你实现了接口所对应的方法，该模式非常适合对扩展性要求较高的场合。

强制代理：

强制代理的概念就是要从真实角色查找到代理角色，不允许直接访问真实角色。高层模块只

要调用 `getProxy` 就可以访问真实角色的所有方法，它根本就不需要产生一个代理出来，代理的管理已经由真实角色自己完成。

动态代理：

根据被代理的接口生成所有的方法，也就是说**给定一个接口，动态代理会宣称“我已经实现该接口下的所有方法了”**。

两条独立发展的线路。动态代理实现代理的职责，业务逻辑 `Subject` 实现相关的逻辑功能，两者之间没有必然的相互耦合的关系。通知 `Advice` 从另一个切面切入，最终在高层模块也就是 `Client` 进行耦合，完成逻辑的封装任务。

动态代理调用过程示意图：

动态代理的意图：横切面编程，在不改变我们已有代码结构的情况下增强或控制对象的行为。

首要条件：被代理的类必须要实现一个接口。

7.原型模式（Prototype Pattern）

定义：Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype.（用原型实例指定创建对象的种类，并且通过拷贝这些原型创建新的对象。）

原型模式通用代码：

```
public class PrototypeClass implements Cloneable {
    //覆写父类 Object 方法
    @Override
    public PrototypeClass clone() {
        PrototypeClass prototypeClass = null;
        try {
            prototypeClass =
(PrototypeClass)super.clone();
        } catch (CloneNotSupportedException e) {
            //异常处理
        }
        return prototypeClass;
    }
}
```

原型模式实际上就是实现 **Cloneable** 接口，重写 **clone()** 方法。

使用原型模式的优点：

- 性能优良

原型模式是在内存二进制的拷贝，要比直接 **new** 一个对象性能好很多，特别是要在一个循环体内产生大量的对象时，原型模式可以更好地体现其优点。

- 逃避构造函数的约束

这既是它的优点也是缺点，直接在内存中拷贝，构造函数是不会执行的（参见 13.4 节）。

使用场景：

- 资源优化场景

类初始化需要消化非常多的资源，这个资源包括数据、硬件资源等。

- 性能和安全要求的场景

通过 **new** 产生一个对象需要非常繁琐的数据准备或访问权限，则可以使用原型模式。

- 一个对象多个修改者的场景

一个对象需要提供给其他对象访问，而且各个调用者可能都需要修改其值时，可以考虑使用原型模式拷贝多个对象供调用者使用。

浅拷贝和深拷贝：

浅拷贝：**Object** 类提供的方法 **clone** 只是拷贝本对象，其对象 **内部的数组、引用对象**等都不拷贝，还是指向原生对象的内部元素地址，这种拷贝就叫做 **浅拷贝**，其他的原始类型比如 **int**、**long**、**char**、**string**（当做是原始类型）等都会被拷贝。

注意： 使用原型模式时，引用的成员变量必须满足两个条件才不会被拷贝：一是 **类的成员变量**，而不是方法内变量；二是必须是一个 **可变的引用对象**，而不是一个原始类型或不可变对象。

深拷贝：对私有的类变量进行独立的拷贝

如：`thing.arrayList = (ArrayList<String>)this.arrayList.clone();`

8. 中介者模式

定义：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.（用一个中介对象封装一系列的对象交互，中介者使各对象不需要显示地相互作用，从而使其耦合松散，而且可以独立地改变它们之间的交互。）

- **Mediator** 抽象中介者角色

抽象中介者角色定义统一的接口，用于各同事角色之间的通信。

- **Concrete Mediator** 具体中介者角色

具体中介者角色通过协调各同事角色实现协作行为，因此它必须依赖于各个同事角色。

- **Colleague** 同事角色

每一个同事角色都知道中介者角色，而且与其他的同事角色通信的时候，一定要通过中介者角色协作。每个同事类的行为分为两种：一种是同事本身的行为，比如改变对象本身的状态，处理自己的行为等，这种行为叫做自发行为（**Self-Method**），与其他的同事类或中介者没有任何的依赖；第二种是必须依赖中介者才能完成的行为，叫做依赖方法（**Dep-Method**）。

通用抽象中介者代码：

```
public abstract class Mediator {  
    //定义同事类  
    protected ConcreteColleague1 c1;  
    protected ConcreteColleague2 c2;  
    //通过 getter/setter 方法把同事类注入进来  
    public ConcreteColleague1 getC1() {  
        return c1;  
    }  
    public void setC1(ConcreteColleague1 c1) {  
        this.c1 = c1;  
    }  
    public ConcreteColleague2 getC2() {  
        return c2;  
    }  
    public void setC2(ConcreteColleague2 c2) {  
        this.c2 = c2;  
    }  
    //中介者模式的业务逻辑  
    public abstract void doSomething1();  
    public abstract void doSomething2();  
}
```

ps: 使用同事类注入而不使用抽象注入的原因是因为抽象类中不具有每个同事类必须要完成的方法。即每个同事类中的方法各不相同。

问: 为什么同事类要使用构造函数注入中介者，而中介者使用 **getter/setter** 方式注入同事类呢？

这是因为同事类必须有中介者，而中介者却可以只有部分同事类。

使用场景:

中介者模式适用于多个对象之间紧密耦合的情况，紧密耦合的标准是：在类图中出现了蛛网状结构，即每个类都与其他类有直接的联系。

9. 命令模式

定义: Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations. (将一个请求封装成一个对象，从而让你使用不同的请求把客户端参数化，对请求排队或者记录请求日志，可以提供命令的撤销和恢复功能。)

- **Receive** 接收者角色

该角色就是干活的角色，命令传递到这里是应该被执行的，具体到我们上面的例子中就是 Group 的三个实现类（需求组，美工组，代码组）。

- **Command** 命令角色

需要执行的所有命令都在这里声明。

- **Invoker** 调用者角色

接收到命令，并执行命令。在例子中，我（项目经理）就是这个角色。


使用场景：

认为是命令的地方就可以采用命令模式，例如，在 GUI 开发中，一个按钮的点击是一个命令，可以采用命令模式；模拟 DOS 命令的时候，当然也要采用命令模式；触发一反馈机制的处理等。

10. 责任链模式

定义：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.（使多个对象都有机会处理请求，从而避免了请求的发送者和接受者之间的耦合关系。将这些对象连成一条链，并沿着这条链传递该请求，直到有对象处理它为止。）

抽象处理者的代码：

```

public abstract class Handler {
    private Handler nextHandler;
    //每个处理者都必须对请求做出处理
    public final Response handleMessage(Request request) {
        Response response = null;
        //判断是否是自己的处理级别

        if(this.getHandlerLevel().equals(request.getRequestLevel())){
            response = this.echo(request);
        }else{ //不属于自己的处理级别
            //判断是否有下一个处理者
            if(this.nextHandler != null){
                response =
this.nextHandler.handleMessage(request);
            }else{
                //没有适当的处理者，业务自行处理
            }
        }
    }
}
```

```
    }  
    return response;  
}  
//设置下一个处理者是谁  
public void setNext(Handler _handler) {  
    this.nextHandler = _handler;  
}  
//每个处理者都有一个处理级别  
protected abstract Level getHandlerLevel();  
//每个处理者都必须实现处理任务  
protected abstract Response echo(Request request);  
}
```

抽象的处理者实现三个职责：

- 一是定义一个请求的处理方法 `handleMessage`，唯一对外开放的方法；
- 二是定义一个链的编排方法 `setNext`，设置下一个处理者；
- 三是定义了具体的请求者必须实现的两个方法：定义自己能够处理的级别 `getHandlerLevel` 和具体的处理任务 `echo`。

注意事项：

链中节点数量需要控制，避免出现超长链的情况，一般的做法是在 `Handler` 中设置一个最大节点数量，在 `setNext` 方法中判断是否已经是超过其阈值，超过则不允许该链建立，避免无意识地破坏系统性能。

11.装饰模式（Decorator Pattern）

定义：Attach additional responsibilities to an object dynamically keeping the same interface. Decorators provide a flexible alternative to subclassing for extending functionality.（动态地给一个对象添加一些额外的职责。就增加功能来说，装饰模式相比生成子类更为灵活。）

● Component 抽象构件

`Component` 是一个接口或者是抽象类，就是定义我们最核心的对象，也就是最原始的对象，如上面的成绩单。

注意：在装饰模式中，必然有一个最基本、最核心、最原始的接口或抽象类充当 `Component` 抽象构件。

● ConcreteComponent 具体构件

`ConcreteComponent` 是最核心、最原始、最基本的接口或抽象类的实现，你要装饰的就是它。

● Decorator 装饰角色

一般是一个抽象类，做什么用呢？实现接口或者抽象方法，它里面可不一定有抽象的方法呀，

在它的属性里必然有一个 `private` 变量指向 `Component` 抽象构件。

- 具体装饰角色

`ConcreteDecoratorA` 和 `ConcreteDecoratorB` 是两个具体的装饰类，你要把你最核心的、最原始的、最基本的东西装饰成其他东西，上面的例子就是把一个比较平庸的成绩单装饰成家长认可的成绩单。

使用场景：

- 需要扩展一个类的功能，或给一个类增加附加功能。
- 需要动态地给一个对象增加功能，这些功能可以再动态地撤销。
- 需要为一批的兄弟类进行改装或加装功能，当然是首选装饰模式。

12.策略模式（Strategy Pattern）

定义：Define a family of algorithms,encapsulate each one,and make them interchangeable.
(定义一组算法，将每个算法都封装起来，并且使它们之间可以互换。)

- **Context** 封装角色

它也叫做上下文角色，起承上启下封装作用，屏蔽高层模块对策略、算法的直接访问，封装可能存在的变化。

- **Strategy** 抽象策略角色

策略、算法家族的抽象，通常为接口，定义每个策略或算法必须具有的方法和属性。各位看官可能要问了，类图中的 `AlgorithmInterface` 是什么意思，嘿嘿，`algorithm` 是“运算法则”的意思，结合起来意思就明白了吧。

- **ConcreteStrategy** 具体策略角色（多个）

实现抽象策略中的操作，该类含有具体的算法。

使用场景：

- 多个类只有在算法或行为上稍有不同的场景。
- 算法需要自由切换的场景。
- 需要屏蔽算法规则的场景。

注意事项：具体策略数量超过 4 个，则需要考虑使用混合模式

策略模式扩展：策略枚举



```
public enum Calculator {  
    //加法运算  
    ADD("+") {  
        public int exec(int a,int b){  
            return a+b;  
        }  
    },  
    //减法运算  
    SUB("-") {  
        public int exec(int a,int b){  
            return a-b;  
        }  
    }  
}
```

```
        public int exec(int a, int b) {
            return a - b;
        }
    };
    String value = "";
    //定义成员值类型
    private Calculator(String _value) {
        this.value = _value;
    }
    //获得枚举成员的值
    public String getValue() {
        return this.value;
    }
    //声明一个抽象函数
    public abstract int exec(int a, int b);
}
```

定义：

- 它是一个枚举。
- 它是一个浓缩了的策略模式的枚举。

注意：

受枚举类型的限制，每个枚举项都是 **public**、**final**、**static** 的，扩展性受到了一定的约束，因此在系统开发中，策略枚举一般担当不经常发生变化的角色。

致命缺陷：

所有的策略都需要暴露出去，由客户端决定使用哪一个策略。

13. 适配器模式（Adapter Pattern）

定义：Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.（将一个类的接口变换成客户端所期待的另一种接口，从而使原本因接口不匹配而无法在一起工作的两个类能够在一起工作。）

类适配器：

- Target 目标角色

该角色定义把其他类转换为何种接口，也就是我们的期望接口，例子中的 **IUserInfo** 接口就是目标角色。

- Adaptee 源角色

你想把谁转换成目标角色，这个“谁”就是源角色，它是已经存在的、运行良好的类或对象，

经过适配器角色的包装，它会成为一个崭新、靓丽的角色。

- **Adapter 适配器角色**

适配器模式的核心角色，其他两个角色都是已经存在的角色，而适配器角色是需要新建立的，它的职责非常简单：把源角色转换为目标角色，怎么转换？通过继承或是类关联的方式。

- **使用场景：**

你有动机修改一个已经投产中的接口时，适配器模式可能是最适合你的模式。比如系统扩展了，需要使用一个已有或新建立的类，但这个类又不符合系统的接口，怎么办？使用适配器模式，这也是我们例子中提到的。

- **注意事项：**

详细设计阶段不要考虑使用适配器模式，使用主要场景为扩展应用中。

- **对象适配器：**

- **对象适配器和类适配器的区别：**

类适配器是类间继承，对象适配器是对象的合成关系，也可以说是类的关联关系，这是两者的根本区别。（实际项目中对象适配器使用到的场景相对比较多）。

14. 迭代器模式（Iterator Pattern）

定义：Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.（它提供一种方法访问一个容器对象中各个元素，而又不需暴露该对象的内部细节。）

- **Iterator 抽象迭代器**

抽象迭代器负责定义访问和遍历元素的接口，而且基本上是有固定的 3 个方法：**first()**获得第一个元素，**next()**访问下一个元素，**isDone()**是否已经访问到底部（Java 叫做 **hasNext()**方法）。

- **ConcreteIterator 具体迭代器**

具体迭代器角色要实现迭代器接口，完成容器元素的遍历。

- **Aggregate 抽象容器**

容器角色负责提供创建具体迭代器角色的接口，必然提供一个类似 **createIterator()** 这样的方法，在 Java 中一般是 **iterator()** 方法。

- **Concrete Aggregate 具体容器**

具体容器实现容器接口定义的方法，创建出容纳迭代器的对象。

ps: 迭代器模式已经被淘汰，java 中已经把迭代器运用到各个聚集类（collection）中了，使用 java 自带的迭代器就已经满足我们的需求了。

15. 组合模式（Composite Pattern）

定义：Compose objects into tree structures to represent part-whole hierarchies. Composite

lets clients treat individual objects and compositions of objects uniformly. (将对象组合成树形结构以表示“部分-整体”的层次结构,使得用户对单个对象和组合对象的使用具有一致性。)

- **Component** 抽象构件角色

定义参加组合对象的共有方法和属性,可以定义一些默认的行为或属性,比如我们例子中的 `getInfo` 就封装到了抽象类中。



- **Leaf** 叶子构件

叶子对象,其下再也没有其他的分支,也就是遍历的最小单位。

- **Composite** 树枝构件

树枝对象,它的作用是组合树枝节点和叶子节点形成一个树形结构。

树枝构件的通用代码:

```
public class Composite extends Component {  
    //构件容器  
    private ArrayList<Component> componentArrayList = new  
ArrayList<Component>();  
    //增加一个叶子构件或树枝构件  
    public void add(Component component) {  
        this.componentArrayList.add(component);  
    }  
    //删除一个叶子构件或树枝构件  
    public void remove(Component component) {  
this.componentArrayList.remove(component);  
    }  
    //获得分支下的所有叶子构件和树枝构件  
    public ArrayList<Component> getChildren() {  
        return this.componentArrayList;  
    }  
}  

```

使用场景:

- 维护和展示部分-整体关系的场景,如树形菜单、文件和文件夹管理。
- 从一个整体中能够独立出部分模块或功能的场景。

注意:

只要是树形结构,就考虑使用组合模式。

16. 观察者模式 (Observer Pattern)

定义: Define a one-to-many dependency between objects so that when one object changes

state,all its dependents are notified and updated automatically.（定义对象间一种一对多的依赖关系，使得每当一个对象改变状态，则所有依赖于它的对象都会得到通知并被自动更新。）

- **Subject** 被观察者

定义被观察者必须实现的职责，它必须能够动态地增加、取消观察者。它一般是抽象类或者是实现类，仅仅完成作为被观察者必须实现的职责：管理观察者并通知观察者。

- **Observer** 观察者

观察者接收到消息后，即进行 **update**（更新方法）操作，对接收到的信息进行处理。

- **ConcreteSubject** 具体的被观察者

定义被观察者自己的业务逻辑，同时定义对哪些事件进行通知。

- **ConcreteObserver** 具体的观察者

每个观察在接收到消息后的处理反应是不同，各个观察者有自己的处理逻辑。

被观察者通用代码：

```

public abstract class Subject {
    //定义一个观察者数组
    private Vector<Observer> obsVector = new
Vector<Observer>();
    //增加一个观察者
    public void addObserver(Observer o) {
        this.obsVector.add(o);
    }
    //删除一个观察者
    public void delObserver(Observer o) {
        this.obsVector.remove(o);
    }
    //通知所有观察者
    public void notifyObservers() {
        for(Observer o:this.obsVector) {
            o.update();
        }
    }
}

```

使用场景：

- 关联行为场景。需要注意的是，关联行为是可拆分的，而不是“组合”关系。
- 事件多级触发场景。
- 跨系统的消息交换场景，如消息队列的处理机制。

注意：

- 广播链的问题

在一个观察者模式中最多出现一个对象既是观察者也是被观察者，也就是说消息最多转发一次（传递两次）。

- 异步处理问题

观察者比较多，而且处理时间比较长，采用异步处理来考虑线程安全和队列的问题。

17.门面模式（Facade Pattern）

定义：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.（要求一个子系统的外部与其内部的通信必须通过一个统一的对象进行。门面模式提供一个高层次的接口，使得子系统更易于使用。）

- Facade 门面角色

客户端可以调用这个方法。此角色知晓子系统的所有功能和责任。一般情况下，本角色会将所有从客户端发来的请求委派到相应的子系统去，也就说该角色没有实际的业务逻辑，只是一个委托类。

- subsystem 子系统角色

可以同时有一个或者多个子系统。每一个子系统都不是一个单独的类，而是一个类的集合。子系统并不知道门面的存在。对于子系统而言，门面仅仅是另外一个客户端而已。

使用场景：

- 为一个复杂的模块或子系统提供一个供外界访问的接口
- 子系统相对独立——外界对子系统的访问只要黑箱操作即可
- 预防低水平人员带来的风险扩散

注意：

- 一个子系统可以有多个门面
- 门面不参与子系统内的业务逻辑

18.备忘录模式（Memento Pattern）

定义：Without violating encapsulation,capture and externalize an object's internal state so that the object can be restored to this state later.（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。这样以后就可将该对象恢复到原先保存的状态。）

- Originator 发起人角色

记录当前时刻的内部状态，负责定义哪些属于备份范围的状态，负责创建和恢复备忘录数据。

- Memento 备忘录角色（简单的 javabean）

负责存储 Originator 发起人对象的内部状态，在需要的时候提供发起人需要的内部状态。

- **Caretaker** 备忘录管理员角色（简单的 javabean）

对备忘录进行管理、保存和提供备忘录。

使用场景：

- 需要保存和恢复数据的相关状态场景。
- 提供一个可回滚（rollback）的操作。
- 需要监控的副本场景中。
- 数据库连接的事务管理就是用的备忘录模式。

注意：

- 备忘录的生命期
- 备忘录的性能

不要在频繁建立备份的场景中使用备忘录模式（比如一个 for 循环中）。


clone 方式备忘录：

- 发起人角色融合了发起人角色和备忘录角色，具有双重功效

多状态的备忘录模式

- 增加了一个 BeanUtils 类，其中 backupProp 是把发起人的所有属性值转换到 HashMap 中，方便备忘录角色存储。restoreProp 方法则是把 HashMap 中的值返回到发起人角色中。

BeanUtil 工具类代码：

```

public class BeanUtils {
    //把 bean 的所有属性及数值放入到 Hashmap 中
    public static HashMap<String, Object> backupProp(Object
bean) {
        HashMap<String, Object> result = new
HashMap<String, Object>();
        try {
            //获得 Bean 描述
            BeanInfo
beanInfo=Introspector.getBeanInfo(bean.getClass());
            //获得属性描述
            PropertyDescriptor[]
descriptors=beanInfo.getPropertyDescriptors();
            //遍历所有属性
            for(PropertyDescriptor des:descriptors){
                //属性名称
                String fieldName = des.getName();
                //读取属性的方法
```

```
Method getter =
des.getReadMethod();

//读取属性值
Object
fieldValue=getter.invoke(bean, new Object[] {});
    if(!fieldName.equalsIgnoreCase("class")) {
        result.put(fieldName, fieldValue);
    }
}
} catch (Exception e) {
    //异常处理
}
return result;
}
//把 HashMap 的值返回到 bean 中
public static void restoreProp(Object
bean, HashMap<String, Object> propMap) {
    try {
        //获得 Bean 描述
        BeanInfo beanInfo =
Introspector.getBeanInfo(bean.getClass());
        //获得属性描述
        PropertyDescriptor[] descriptors =
beanInfo.getPropertyDescriptors();
        //遍历所有属性
        for(PropertyDescriptor des:descriptors) {
            //属性名称
            String fieldName = des.getName();
            //如果有这个属性
            if(propMap.containsKey(fieldName)) {
                //写属性的方法
                Method setter = des.getWriteMethod();
                setter.invoke(bean, new
Object[] {propMap.get(fieldName)});
            }
        }
    } catch (Exception e) {
        //异常处理
        System.out.println("shit");
        e.printStackTrace();
    }
}
}
```



多备份的备忘录：略

封装得更好一点：保证只能对发起人可读

● 建立一个空接口 **IMemento**——什么方法属性都没有的接口，然后在发起人 **Originator** 类中建立一个内置类（也叫做类中类）**Memento** 实现 **IMemento** 接口，同时也实现自己的业务逻辑。

19.访问者模式（Visitor Pattern）

定义：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.（封装一些作用于某种数据结构中的各元素的操作，它可以在不改变数据结构的前提下定义作用于这些元素的新的操作。）

● **Visitor**——抽象访问者

抽象类或者接口，声明访问者可以访问哪些元素，具体到程序中就是 **visit** 方法的参数定义哪些对象是可以被访问的。

● **ConcreteVisitor**——具体访问者

它影响访问者访问到一个类后该怎么干，要做什么事情。

● **Element**——抽象元素

接口或者抽象类，声明接受哪一类访问者访问，程序上是通过 **accept** 方法中的参数来定义的。

● **ConcreteElement**——具体元素

实现 **accept** 方法，通常是 **visitor.visit(this)**，基本上都形成了一种模式了。

● **ObjectStruture**——结构对象

元素产生者，一般容纳在多个不同类、不同接口的容器，如 **List**、**Set**、**Map** 等，在项目中，一般很少抽象出这个角色。

使用场景：

● 一个对象结构包含很多类对象，它们有不同的接口，而你想对这些对象实施一些依赖于其具体类的操作，也就说是用迭代器模式已经不能胜任的情景。

● 需要对一个对象结构中的对象进行很多不同并且不相关的操作，而你想避免让这些操作“污染”这些对象的类。

20.状态模式（State Pattern）

定义：Allow an object to alter its behavior when its internal state changes.The object will

appear to change its class. (当一个对象内在状态改变时允许其改变行为, 这个对象看起来像改变了其类。)

- **State**——抽象状态角色

接口或抽象类, 负责对象状态定义, 并且封装环境角色以实现状态切换。

- **ConcreteState**——具体状态角色

每一个具体状态必须完成两个职责: 本状态的行为管理以及趋向状态处理, 通俗地说, 就是本状态下要做的事情, 以及本状态如何过渡到其他状态。

- **Context**——环境角色

定义客户端需要的接口, 并且负责具体状态的切换。

使用场景:

- 行为随状态改变而改变的场景

这也是状态模式的根本出发点, 例如权限设计, 人员的状态不同即使执行相同的行为结果也会不同, 在这种情况下需要考虑使用状态模式。

- 条件、分支判断语句的替代者

注意:

状态模式适用于当某个对象在它的状态发生改变时, 它的行为也随着发生比较大的变化, 也就是说在行为受状态约束的情况下可以使用状态模式, 而且使用时对象的状态最好不要超过 5 个。

21. 解释器模式 (Interpreter Pattern)

定义: Given a language, define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language. (给定一门语言, 定义它的文法的一种表示, 并定义一个解释器, 该解释器使用该表示来解释语言中的句子。)

- **AbstractExpression**——抽象解释器

具体的解释任务由各个实现类完成, 具体的解释器分别由 **TerminalExpression** 和 **Non-terminalExpression** 完成。

- **TerminalExpression**——终结符表达式

实现与文法中的元素相关联的解释操作, 通常一个解释器模式中只有一个终结符表达式, 但有多实例, 对应不同的终结符。具体到我们例子就是 **VarExpression** 类, 表达式中的每个终结符都在栈中产生了一个 **VarExpression** 对象。

- **NonterminalExpression**——非终结符表达式

文法中的每条规则对应于一个非终结符表达式, 具体到我们的例子就是加减法规则分别对应到 **AddExpression** 和 **SubExpression** 两个类。非终结符表达式根据逻辑的复杂程度而增加, 原则上每个文法规则都对应一个非终结符表达式。

- **Context**——环境角色

具体到我们的例子中是采用 `HashMap` 代替。

使用场景：

- 重复发生的问题可以使用解释器模式
- 一个简单语法需要解释的场景

注意：

尽量不要在重要的模块中使用解释器模式，否则维护会是一个很大的问题。在项目中可以使用 `shell`、`JRuby`、`Groovy` 等脚本语言来代替解释器模式，弥补 `Java` 编译型语言的不足。

22. 享元模式（Flyweight Pattern）

定义：Use sharing to support large numbers of fine-grained objects efficiently.（使用共享对象可有效地支持大量的细粒度的对象。）

对象的信息分为两个部分：内部状态（`intrinsic`）与外部状态（`extrinsic`）。

- 内部状态

内部状态是对象可共享出来的信息，存储在享元对象内部并且不会随环境改变而改变。

- 外部状态

外部状态是对象得以依赖的一个标记，是随环境改变而改变的、不可以共享的状态。

- `Flyweight`——抽象享元角色

它简单地讲就是一个产品的抽象类，同时定义出对象的外部状态和内部状态的接口或实现。

- `ConcreteFlyweight`——具体享元角色

具体的一个产品类，实现抽象角色定义的业务。该角色中需要注意的是内部状态处理应该与环境无关，不应该出现一个操作改变了内部状态，同时修改了外部状态，这是绝对不允许的。

- `unsharedConcreteFlyweight`——不可共享的享元角色

不存在外部状态或者安全要求（如线程安全）不能够使用共享技术的对象，该对象一般不会出现现在享元工厂中。

- `FlyweightFactory`——享元工厂

职责非常简单，就是构造一个池容器，同时提供从池中获得对象的方法。

享元工厂的代码：

```
public class FlyweightFactory {  
    //定义一个池容器  
    private static HashMap<String, Flyweight> pool= new  
HashMap<String, Flyweight>();  
    //享元工厂  
    public static Flyweight getFlyweight(String Extrinsic){  
        //需要返回的对象  
        Flyweight flyweight = null;  
        //在池中没有该对象
```

```
        if(pool.containsKey(Extrinsic)){
            flyweight = pool.get(Extrinsic);
        }else{
            //根据外部状态创建享元对象
            flyweight = new
ConcreteFlyweight1(Extrinsic);
            //放置到池中
            pool.put(Extrinsic, flyweight);
        }
        return flyweight;
    }
}
```

使用场景：

- 系统中存在大量的相似对象。
- 细粒度的对象都具备较接近的外部状态，而且内部状态与环境无关，也就是说对象没有特定身份。
- 需要缓冲池的场景。

注意：

- 享元模式是线程不安全的，只有依靠经验，在需要的地方考虑一下线程安全，在大部分场景下不用考虑。对象池中的享元对象尽量多，多到足够满足为止。
- 性能安全：外部状态最好以 java 的基本类型作为标志，如 String, int, 可以提高效率。

23.桥梁模式（Bridge Pattern）

定义：Decouple an abstraction from its implementation so that the two can vary independently.（将抽象和实现解耦，使得两者可以独立地变化。）

● Abstraction——抽象化角色

它的主要职责是定义出该角色的行为，同时保存一个对实现化角色的引用，该角色一般是抽象类。

● Implementor——实现化角色

它是接口或者抽象类，定义角色必需的行为和属性。

● RefinedAbstraction——修正抽象化角色

它引用实现化角色对抽象化角色进行修正。

● ConcreteImplementor——具体实现化角色

它实现接口或抽象类定义的方法和属性。

使用场景：

- 不希望或不适用使用继承的场景

- 接口或抽象类不稳定的场景
- 重用性要求较高的场景

注意：

发现类的继承有 **N** 层时，可以考虑使用桥梁模式。桥梁模式主要考虑如何拆分抽象和实现。

三、C#/Java 代码规范

代码设计准则

重用性

软件工程师的一个目标就是通过重复使用代码来避免编写新的代码。因为重新使用已有的代码可以降低成本、增加代码的可靠性并提高它们的一致性。

可维护性

一段风格良好的代码肯定具有很好的可读性。如果代码的维护者不明白你所书写的代码的含义，那么代码的维护就会成为一件十分困难而耗时的的工作，所以在你编写代码的时候，一定要时刻的考虑到你所书写的代码别人是不是能够真正理解。

模块化

封装和信息隐藏。如果一个模块的代码过长或者过于复杂，一定要考虑你的模块是不是需要重新组织一下，或者将一个模块分解成多个模块。

不要对使用你代码的用户抱有太多的幻想。要让代码保护自己不被错误使用。

书写注释

注释可以帮助别人更好的理解你的代码，尽管这一条经常被软件开发人员忽视，但是对于代码的作者，使用者或者维护人员来说，代码注释在提高代码的可读性和可维护性方面是十分必要的。

保持代码风格的一致性

无论是在文件级别，模块级别，还是工程级别，请保持代码风格的一致性。

命名需要有意义

变量、函数、类和名字空间的命名要使用有意义的英文单词，且命名不要过长，简明易懂为佳，尽量不要使用缩写，更不要使用汉语拼音。

保持可读性

代码长度不要超过屏幕，如果超过了，需要考虑在合适的地方断行，保持代码的易读性。

一个方法在 1920*1080 的分辨率 100%放大的情况下，内容不要超过一个屏幕，如果存在超过的情况可以考虑把部分内容重构为另外方法。

命名规范

本章旨在指导开发人员在代码命名方面更具有一致性，提升代码可读性。

大小写约定

Pascal: 用于除参数名称与局部变量的所有标识符。第一个字母必须大写，并且后面的连接词第一个字母也要大写。例如：**PropertyDescriptor**、**HtmlTag**。在特殊情况由两个字母首字母缩写词在其中两个字母都大写，例如 **IOStream**。

Camel: 仅用于参数命名以及局部变量。名称中第一个单词首字母小写。例如：**propertyDescriptor**、**ioStream**、**htmlTag**。

注意：像“Open_Windows_Text”、“cmd_CheckStuats”在命名中间加下划线连字符这样的不可取。

表 1.1 描述了不同类型的标识符的大小写规则。注意为了大小写规则，将关闭格式的复合词视为单个单词。（详见表 1.2）

表 1.1(标识符的大小写规则)

标识符	大小写	示例
命名空间	Pascal	<code>namespace System.Security { ... }</code>
类	Pascal	<code>public class StreamReader { ... }</code>
接口	Pascal	<code>public interface IEnumerable { ... }</code>

标识符	大小写	示例
方法	Pascal	<pre>public class Object { public virtual string ToString(); }</pre>
属性	Pascal	<pre>public class String { public int Length { get; } }</pre>
事件	Pascal	<pre>public class Process { public event EventHandler Exited; }</pre>
字段	Pascal/ Camel	<pre>public class MessageQueue { public static readonly TimeSpan InfiniteTimeout; } public struct UInt32 { public const Min = 0; }</pre>
枚举值	Pascal	<pre>public enum FileMode { Append, ... }</pre>

标识符	大小写	示例
参数	Camel	<pre>public class Convert { public static int ToInt32(string value); }</pre>

（表 1.1）

表 1.2(关闭格式的复合词)

Pascal	Camel	错误
BitFlag	bitFlag	Bitflag
Callback	callback	CallBack
Canceled	canceled	Cancelled
DoNot	doNot	Don't
Email	email	EMail
Endpoint	endpoint	EndPoint
FileName	fileName	Filename
Gridline	gridline	GridLine
Hashtable	hashtable	HashTable
Id	id	ID
Indexes	indexes	Indices
LogOff	logOff	LogOut
LogOn	logOn	LogIn
Metadata	metadata	MetaData, metaData

Pascal	Camel	错误
Multipanel	multipanel	MultiPanel
Multiview	multiview	MultiView
Namespace	namespace	NameSpace
Ok	ok	OK
Pi	pi	PI
Placeholder	placeholder	PlaceHolder
SignIn	signIn	SignOn
SignOut	signOut	SignOff
UserName	userName	Username
WhiteSpace	whiteSpace	Whitespace
Writable	writable	Writeable

(表 1.2)

通用命名约定

- 1) 命名应当选择易读的标识符。例如：名为的属性
HorizontalAlignment 可读性比 AlignmentHorizontal。
- 2) 命名的易于理解的重要性比命名的简介、易读要高。例如
CanScrollHorizontally 优于 ScrollableX。
- 3) 使用类型名称在语义上是有意义的名称，而不是特定于语言的关键字。例如，GetLength 是比 GetInt 更好的名称。

- 4)不使用匈牙利语表示法，例如：`int intMax = 0;`。
- 5)避免广泛使用“x”以及与关键字冲突的标识符。
- 6)不缩写用作标识符名称的一部分。例如，使用 **GetWindow** 而非 **GetWin**。
- 7)除非必要不使用并不被广泛接受的任何首字母缩写词，即时它的确是缩写词，例如，使用 **Realtime** 而非 **RT**。
- 8)避免使用硬编码，例：`for (i = 1 To 7)`。而是使用变量，如 `For (i = 1 To NUM_DAYS_IN_WEEK)` 以便于维护和理解。
- 9)命名中需要规避的关键字如表 1.3

表 1.3 (关键字检索表)

C#	Visual Basic	C++	CLR
sbyte	SByte	char	SByte
byte	Byte	unsigned char	Byte
short	Short	short	Int16
ushort	UInt16	unsigned short	UInt16
int	Integer	int	Int32
uint	UInt32	unsigned int	UInt32
long	Long	__int64	Int64
ulong	UInt64	unsigned __int64	UInt64
float	Single	float	Single

C#	Visual Basic	C++	CLR
double	Double	double	Double
bool	Boolean	bool	Boolean
char	Char	wchar_t	Char
string	String	String	String
object	Object	Object	Object

(表 1.3)

- 10)对于替换已有 API 的新 API 命名规范有如下几条：
- (1)原则上不允许删除旧的或者弃用的 API。
 - (2)创建的现有 API 的新版本时需要使用类似于旧 API 的名称，这有助于突出显示 API 之间的关系。
 - (3)添加一个后缀而不是前缀以指示现有 API 的新版本，这将方便发现查找新 API。因为大多数浏览器和 Intellisense 按字母顺序显示标识符。
 - (4)尽量使用版本号作为后缀来标识新的 API 以方便区分 API 的不同的版本。

程序集和 DLL 的命名

程序集是部署和托管的代码程序的标识的单元，它通常与 DLL 是进行一对一映射，但也可以跨一个或多个文件。

程序集命名规范一般是按照 `<Company>.(<Product>|<Technology>).<Component>.dll` 的格式，其中 `<Component>` 包含一个或多个以点分隔的子句，例如 `JUSHA.Data.MySql.dll`、`JUSHA.WEBQA.Client.WpfForms.dll`。

对于不同的版本问题，在 `dll` 的编译时，加入版本信息。

命名空间的命名

命名空间的命名遵循以下基础规则：

`<Company>.(<Product>|<Technology>)[.<Feature>][.<Subnamespace>]`。

例如：`JUSHA.WEBQA.Communication`。

具体规则说明如下：

- 1)前缀需要表明公司标识，通常应当为 `JUSHA`。
- 2)第二个名称应当是独立于版本，独立于硬件平台的稳定的产品名称。
- 3)尽量遵循 `Pascal` 命名规则，除非公司的商标，产品名称等不可分割的关键字，例如 `JUSHA`。
- 4)按照实际情况，尽量使用复词，例如 `System.Collections` 而不是 `System.Collection`，品牌名称和首字母缩写词例外。
- 5)命名过程中避免泛型名称例如 `Element`，`Node`，`Log`，和 `Message`，因为这样做将有非常高概率导致方案的类型名称冲突。

类、结构和接口的命名

类、结构和接口的命名同样以 **Pascal** 命名规则为准则，同时应当遵循以下规则：

1) 尽量以基类作为派生类的结尾以提高可读性，清楚地说明继承关系。例如 **ArgumentOutOfRangeException**，这是一种类型的 **Exception**，和 **SerializableAttribute**，这是一种类型的 **Attribute**。这条规则不应当盲从，需要合理判断使用场景，例如 **Button** 类是一种类型的 **Control** 事件，尽管 **Control** 未出现在其名称。

2) 所有的接口应当以 **I** 作为接口的前缀。例如，**IComponent**（描述性名词），**ICustomAttributeProvider**（名词短语），和 **IPersistable**（形容词）是适当的接口名称。与其他类型名称，避免缩写。

3) 确保的名称不同仅通过 **I** 前缀的接口名称定义其中该类是对此接口的唯一实现。例如接口 **ICommand** 与类 **Command**，确保类 **Command** 是对接口 **ICommand** 的唯一实现。

4) 在描述泛型参数名称时，如果任何描述性的名称不会增加其易读性的情况下，优先考虑 **T** 作为单字母名称，否则尽量使用描述性的名称。例如：

```
1. public int IComparer<T> { ... }
2. public interface ISessionChannel<TSession> where TSession : ISession{
3.     TSession Session { get; }
4. }
```

5) 常见类型的派生或实现尽量遵守表 1.4。

表 1.4（常见类型的派生或实现）

基类型	派生/实现类型准则
System.Attribute	✓ 自定义特性类的名称以 “Attribute”为后缀。
System.Delegate	✓ 事件中使用的委托名称以 “EventHandler”为后缀。 ✓ 事件处理程序的委托名称以 “Callback”为后缀。 X 不要使用“Delegate”作为后 缀。
System.EventArgs	✓ 使用"EventArgs"作为后缀。
IEnumerable ICollection IList IEnumerable<T> ICollection<T>	✓ 使用"Collection"作为后缀。

基类型	派生/实现类型准则
ICollection<T>	
System.IO.Stream	✓ 使用"Stream"作为后缀。
CodeAccessPermission IPermission	✓ 使用"Permission"作为后缀。

(表 1.4)

类成员的命名

1)方法命名

由于方法的执行操作的行为，设计准则要求方法名称是谓词或谓词短语，遵循这一准则选项可用于将方法名称与属性和类型名称区分开来。例如：

```
1. public class String {
2.     public int CompareTo(...);
3.     public string[] Split(...);
4.     public string Trim();
5. }
```

2)属性命名

A、用名词、形容词名词、名词短语命名属性，避免在存在 GetXX

方法的同时声明 **XX** 属性，例如存在方法 `string GetTextWriter(int value) { ... }`的同时声明属性 `string TextWriter { get {...} set {...} }`。

B、对 **bool** 类型属性命名时使用 **Is**, **Can**, **Has** 作为前缀，以增强可读性。（变量 **bool** 也同样试用）

C、有些情况下，允许直接将类名作为属性名，例如：

```
1. public class Color {...}
2. public class Control {
3.     public Color Color { get;set ; }
4. }
```

3)事件的命名

A、应当谓词或谓词短语来命名事件，例如 **Clicked**, **Painting**, **DroppedDown**, 依次类推。同时应当考虑到描述相同事件发生的时态是处于之前还是之后，例如 **Closing** 与 **Closed**。不要使用"**Before**"或"**After**"前缀或 **postfixes** 以指示预和后期事件。

B、作为事件的委托定义时需要加上"**EventHandler**"后缀。例如：

```
1. public delegate void ClickedEventHandler(object sender, ClickedEventArgs e);
```

C、通常情况下，对于可以在派生类中重写的事件，应在基类上提供一个受保护的方法（称为 **Onxxx**，方便派生类调用事件）。此方法只应具有事件参数 **e**，因为发送方总是类型的实例（**sender** 总是 **this**）。例如：

```
1. public event EventHandler<bool> Click;
2.
3. protected virtual void OnClick(bool isMouseLeftClick)
4. {
```

```
5.     Click?.Invoke(this, isMouseLeftClick);  
6. }
```

4)变量的命名

A、变量的命名应当使用名词、形容词名词、名词短语。

B、对局部变量的命名应当遵守 **Camel** 规范。

C、对全局变量的命名应当在遵守 **Camel** 的同时以 “_” 作为首字母用来对局部变量与全局变量的区分。

|类设计规范

在设计类的时候，首先应当确保每个类定义完善的一组相关成员，而不仅仅是随机的不相关的功能集合。

在类和结构之间选择

Struct 的选择应当具有所有以下的特征：

- 1)它逻辑上表示的单个值，类似于基元类型 (**int**, **double** 等。)
- 2)它具有实例大小小于 **16** 字节。
- 3)不可变。
- 4)它不需要频繁进行装箱。

其他场景尽量选用 **Class**。

抽象类设计

1)推荐在抽象类中定义为受保护或私有构造函数。（私有构造函数可以用于限制对抽象类的具体实现）

2)推荐将抽象类视作对接口的补充，为一些公用方法提供实现。

下面是一个抽象类的一个使用范例：

```
1. public abstract class BaseOperation<TResult, TCommand, TDevice> : IRCSEOperation<TResult>
2.     where TCommand : IRCSCCommand
3.     where TDevice : IRCSEDevice<TCommand>
4. {
5.     protected TDevice TargetDevice{get;set;}
6.     protected int CurrentCommandIndex{get;set;} = 0;
7.     protected List<TCommand> CommandQueue{get;set;}
8.     public BaseOperation(TDevice device, int interval = 10, int timeout = 0)
9.     {
10.         .....
11.     }
12.
13.     protected virtual void OnOperationComplete(TResult reply)
14.     {
15.         .....
16.     }
17.
18.     public int Interval { get; private set; }
19.     public int Timeout { get; private set; }
20.
21.     public virtual TResult Excute()
22.     {
23.         .....
24.     }
25.
26.     public virtual void ExcuteAsync()
27.     {
28.         .....
29.     }
30.     protected abstract TResult AnalysisReply(IRCSCCommandReply reply);
31. }
```

静态类设计

1)静态类应当谨慎使用，尽量可用作支持框架的面向对象的核心的类。

2)不应当把静态类视作杂项的存储桶。

3)不应声明或重写中的静态类的实例成员。

4) 可以使用无法实例化的抽象类代替静态类来实现类似静态类的功能。例如：

```
1. public abstract Class ClassA{  
2.     private ClassA();  
3.  
4.     public static void Test()  
5.     {  
6.  
7.     }  
8. }
```

5)静态类可以用来作为存放拓展方法的集中类。

6)静态类可以作为存放常用方法的工具类。

接口设计

CLR 不支持多重继承（即，CLR 类不能从多个基类继承），但它允许实现除了从基类继承的一个或多个接口的类型。因此，接口通常用于实现多重继承的效果。例如，**IDisposable** 是一个接口，用于类型来支持 **Disposability** 独立于他们想要参与任何其他继承的层次结构。

1)如果一种业务存在多种实现的场景，推荐使用接口规范输入输出规则。（例如数据库可能是 **SQLite**, **MySql**...但是对表的业务操作是相同的，此时应当用接口来描述业务。）

2)如果两个模块间互相调用，推荐使用接口，翻转两个模块的依赖，使模块依赖接口实现解耦。

3)如果一个接口中声明的方法很多，但其实现类中存在很多的冗余方法，推荐将接口进行更高层次的抽象，避免冗余，使接口尽量简洁。

4)尽量避免使用标记接口（不包含任何成员的接口），如有存在这个需求，更推荐使用特性。

5)不允许将成员添加到已发布的接口内，这将破坏接口的实现。为了避免版本控制的问题，应当创建一个新的接口。

|成员设计规范

属性设计

1)如果调用方法不允许操作属性，那么此属性应该只有 **Get** 方法。

2)**setter** 的可访问性不应该高于 **getter**。

3)在不会导致安全漏洞的情况下为属性设置默认值，这将会避免很多 **NullReferenceException** 导致的异常。

4)**setter** 应当是简单的操作，不应该具有任何前置条件，如果存在发生异常的情况，建议重新设计为一个方法（索引器除外）。

构造函数设计

1)理想情况下，构造函数是默认的，如果的确需要在创建是对成员的默认值进行赋值，请尽量简单。（简单的构造函数是指：具有极小的数字的参数和所有参数都是基元或者枚举的构造函数，简单构造函数能提高框架的可用性）。

2)尽量避免在构造函数内执行大量工作而导致类的实例化需要大量时间。

3)避免在构造函数内调用基类的虚拟成员。

4)如果构造函数需要设置为静态的，那么不允许其被公开。使用 **private static** 代替 **public static**（在公开静态构造函数内执行操作可能会导致意外的行为）。

事件设计

1)推荐尽量使用 [System.EventHandler<TEventArgs>](#) 作为事件处理程序的委托，而不是选择新建委托。

2) 推荐尽量从 [EventArgs](#) 派生出自定义类来作为事件的自变量，除非确定该事件不需要传入任何数据。（例如 **EventArgs Click (sender s,object e)**）

3)引发事件尽量使用受保护的虚方法，这样可以允许在派生类中引发该事件（按照约定，该虚方法以 **On** 开头，结构、密封类或者静态内中的非静态事件除外）。

4)引发非静态事件者不应当将 **null** 作为 **sender**，相反，引发静

态事件应当将 `null` 作为 `sender`。

字段设计

1)全局变量建议仅是 `private`，如果需要 `public` 或者 `protect` 尝试用属性替代之。

2)全局变量命名应当以下划线“`_`”作为开头。

3)常量字段用于用不更改的常量。

4)`readonly` 不应该用于可变类型（例如大部分数组和集合）。

三、软件项目版本号的规定

对软件项目的持续升级和开发是个动态的过程，如果有需求的改变，则会触发项目的开发内容改变。

对外部版本号的规定

所以，为了标识不同的开发内容和交付内容，软件项目设置了版本号对软件进行管理，版本号规则设置如下：

版本号的设置如下：**VXX.YY.ZZ**

V 表示：版本号标识（Version 首字母）

XX 表示：大版本

YY 表示：升级

ZZ 表示：补丁

默认情况下：每个版次的初始数字是 0，预留两位数字标识每个

层级的版本，如果是两位数字首位为 0 时，0 默认隐藏；

如当版本号为：V01.01.00 可以写成 V1.1.0

对内部版本号的格式规定

对与内部版本，就是在开发阶段未正式受控前，为了表征未提交测试或多次测试不通过时的不同提测代码进行区分，特地开放第四位有效数字作为内部版本区分。

版本号的设置如下：VXX.YY.ZZ.WW

如当版本号为：V1.1.0.2 -----表示版本 V1.1.0 的内部第二个版本。

注意：在软件正式受控后，内部版本号被舍去，内部版本号对客户不可见。

四、软件项目变更版本的原则

对外部版本变更的规定

所以对软件项目的变更结果依据软件版本号进行如下划分：

初版、大版本、升级、补丁；

初版（V1.0.0）的设置定义：

当项目立项的首次提测并通过测试，软件在公司层面得到受控，被客户可见时，软件的版本设置应划分为初版，其版本号为 V1.0.0。

大版本（V1.X.X 到 V2.0.0）的设置定义：

当软件项目的变更影响范围

1. 当软件的对外属性大部分都显著高于原版本时；
2. 当软件的整体架构已经不能复用或有重大调整时；
3. 软件对客户的 UI 全面改变或操作习惯颠覆性调整时；

软件的版本可以作为大版本进位。

升级（V2.0.X 到 V2.1.0）的设置定义：

当软件项目的变更影响范围

1. 变更的对外属性是不只是对软件性能的改进，而是增减软件的功能时；
2. 变更的改动影响范围超出模块内或方法内，并且横跨多个

模块时；

3. 软件的变更影响足以达到“升级”的影响范围时；
一律视为是升级变更。

补丁（V2.1.0 到 V2.1.1）的设置定义：

当软件项目的变更影响范围

1. 变更的对外属性是只停留在对软件性能的改进，没有增加软件的功能时；
2. 变更的改动影响范围只是停留在模块内或方法内时；
3. 软件的变更影响不足以达到“升级”的影响范围时；
一律视为是补丁变更。

对内部版本变更的规定

对与内部版本，就是在开发阶段未正式受控前，为了表征未提交测试或多次测试不通过时的不同提测代码进行区分，特地开放第四位有效数字作为内部版本区分。

版本号的设置如下：VXX.YY.ZZ.WW

如当版本号为：V1.1.0.2 变为 V1.1.0.3 时，表示内部的第三个提测版本，或者内部开发了第三次有不同的升级或不同的补丁的变更，为了以示区分而在开始时设置的内部开发版本。

五、交付件的定义

当软件项目将要受控时，进入研发测试流程时，要有明确的交付件作为下个阶段的输入。

交付件包括：

- 1.软件规格书
- 2.软件概要设计书
- 3.软件代码工程
- 4.软件安装包
- 5.软件使用手册

六、附录

软件项目需求说明书模板

软件需求说明书

1 引言

1.1 编写目的：阐明编写需求说明书的目的，指明读者对象。

1.2 项目背景：应包括

- 项目的委托单位、开发单位和主管部门；
- 该软件系统与其他系统的关系。

1.3 定义：列出文档中所用到的专门术语的定义和缩写词的原文。

1.4 参考资料：可包括

- 项目经核准的计划任务书、合同或上级机关的批文
- 文档所引用的资料、规范等
- 列出这些资料的作者、标题、编号、发表日期、出版单位或

资料来源

2 任务概述

2.1 目标

2.2 运行环境

2.3 条件与限制

3 数据描述

3.1 静态数据

3.2 动态数据：包括输入数据和输出数据。

3.3 数据库描述：给出使用数据库的名称和类型。

3.4 数据词典

3.5 数据采集

4 功能需求

4.1 功能划分

4.2 功能描述

5 性能需求

5.1 数据精确度

5.2 时间特性：如响应时间、更新处理时间、数据转换与传输时间、运行时间等。

5.3 适应性：在操作方式、运行环境、与其他软件的接口以及开发计划等发生变化时，应具有适应能力。

6 运行需求

6.1 用户界面：如屏幕格式、报表格式、菜单格式、输入输出时间等。

6.2 硬件接口

6.3 软件接口

6.4 故障处理

7 其他需求

如可使用性、安全保密、可维护性、可移植性等。

软件规格书模板

软件概要设计书模板

概要设计说明书

1 引言

1.1 写目的：阐明编写概要设计说明书的目的，指明读者对象。

1.2 项目背景：应包括

- 项目的委托单位、开发单位和主管部门
- 该软件系统与其他系统的关系。

1.3 定义：列出本文档中所用到的专门术语的定义和缩写词的愿意。

1.4 参考资料：

- 列出这些资料的作者、标题、编号、发表日期、出版单位或资料来源

- 项目经核准的计划任务书、合同或上级机关的批文；项目开发计划；需求规格说明书；测试计划（初稿）；用户操作手册

- 文档所引用的资料、采用的标准或规范。

2 任务概述

2.1 目标

2.2 需求概述

2.3 条件与限制

3 总体设计

3.2 总体结构和模块外部设计

3.3 功能分配：表明各项功能与程序结构的关系。

4 接口设计

4.1 外部接口：包括用户界面、软件接口与硬件接口。

4.2 内部接口：模块之间的接口。

5 数据结构设计

6 逻辑结构设计

所有文档的统一封面格式如下页所示。

7 物理结构设计

8 数据结构与程序的关系

9 运行设计

9.1 运行模块的组合

9.2 运行控制

9.3 运行时间

10 出错处理设计

10.1 出错输出信息

10.2 出错处理对策：如设置后备、性能降级、恢复及再启动等。

11 安全保密设计

12 维护设计

说明为方便维护工作的设施，如维护模块等。

软件自测清单模板

自检清单									
基本功能									
型号:		RC8610 1.0.0				日期: 2018.05. 17			
测试环境		win10 64 位/win10 32 位/win7 64 位/win7 32 位 中文							
序号	项目		具体操作	重 复 性	预 期 现 象	结 果	自 检 人 员	备 注	
1	安装包 与文档		安装包	1、安装包能正常安装 2、已安装过的应覆盖安装 3、已安装新版本，再安装老版本，提示				刘善翼	
			卸载	1、主程序的卸载，范围是否全面 2、守护进程的卸载					
			通过杀毒软件						
			服务设为防火墙白名单						
	文档		规格书与软件界面和功能匹配					李贯涛	

功能未开放清单			
序号	功能项	说明	备注
1	无		

2			
3			

Bug 修复清单									
序号	Bug 编号	Bug 描述	修复情况			影响范围	自检结果	自检人员	备注
1									
2									
3									
4									

软件变更清单模板

变更功能项					
序号	功能项	变更前	变更后	影响范围	备注
1					
2					
3					