应用软件通用设计和开发规范

文件状态：	文件编号：	JUSHASW001
[ ] 草稿	当前版本：	V0.1
[ ] 正式发布	作　　者：	王文鼎
[√] 正在修改	完成日期：	2018 年 5 月 20 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	2016 年 08 月 15 日	初始版

目录

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）在规定的时间周期内，在所述条件下程序执行所要求的功能的能力；

这意味着该软件在测试运行过程中避免可能发生故障的能力，且一旦发生故障后，具有解脱和排除故障的能力。

(2)健壮性

健壮性又称鲁棒性，是指软件对于规范要求以外的输入能够判断出这个输入不符合规范要求，并能有合理的处理方式。软件健壮性是一个比较模糊的概念，但是却是非常重要的软件外部量度标准。软件设计的健壮与否直接反应了分析设计和编码人员的水平。

(3)可修改性

要求以科学的方法设计软件，使之有良好的结构和完备的文档，系统性能易于调整。

(4)容易理解

软件的可理解性是其可靠性和可修改性的前提。它并不仅仅是文档清晰可读的问题，更要求软件本身具有简单明了的结构。这在很大程度上取决于设计者的洞察力和创造性，以及对设计对象掌握得透彻程度，当然它还依赖于设计工具和方法的适当运用。

(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、C#/Java 设计准则及代码规范

命名准则

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	`namespace System.Security { ... }`
类	Pascal	`public class StreamReader { ... }`

标识符	大小写	示例
接口	Pascal	`public interface IEnumerable { ... }`

方法	Pascal	`public class Object {`
		`  public virtual string ToString();`
		`}`

属性	Pascal	`public class String {`
		`  public int Length { get; }`
		`}`

事件	Pascal	`public class Process {`
		`  public event EventHandler Exited;`
		`}`

字段	Pascal/ Camel	`public class MessageQueue {`
		`  public static readonly TimeSpan`
		`  InfiniteTimeout;`
		`}`
		`public struct UInt32 {`
		`  public const Min = 0;`
		`}`

枚举值	Pascal	`public enum FileMode {`
		`  Append,`
		`  ...`
		`}`

标识符	大小写	示例

参数	Camel	`public class Convert {`
		`public static int ToInt32(string value);`
		`}`

（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
IList<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。

类设计准则

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> : IRCSOperat
    ion<TResult>
2.      where TCommand : IRCSCommand
3.      where TDevice : IRCSDevice<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(IRCSCommandReply 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。

成员设计准则

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)readyonly 不应该用于可变类型（例如大部分数组和集合）。

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("EatAssemply".AppSetting()).CreateInstance("E
    atType".AppSetting()) as IEat;
6.      }
7.  }
8.
9.  class FruitFactory
10. {
11.     public static IFruit CreateFruit()
12.     {
13.         return Assembly.Load("FruitAssemply".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。

编码时注意事项

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。

三、软件项目版本号的格式规定

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。

对版本号的格式规定

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

四、软件项目变更版本的原则

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。

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

软件需求书

软件规格书

软件概要设计书

软件自测清单

软件变更清单