3 Addin 工具使用方法

3.1 前提条件

请参考“部署开发环境”章节，成功注册 Addin 工具。

3.2 功能简介

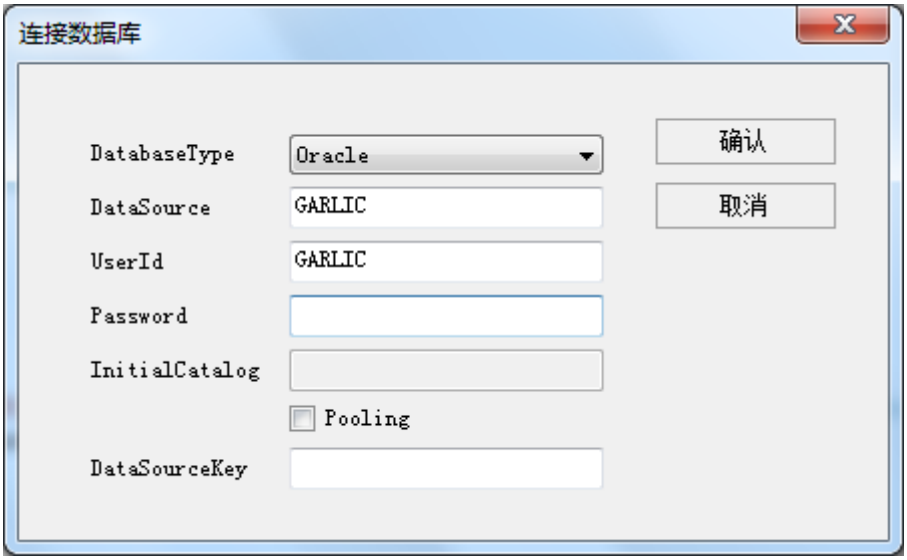
功能	简介
初始化/编辑业务类	可将空类初始化为业务类并添加数据映射关系，且提供所见即所得的手工修订界面。
初始化/编辑查询类	可将空类初始化为查询类并添加数据映射关系，且提供所见即所得的手工修订界面。
初始化指令类	可将空类初始化为指令类。
初始化公共业务规则类	可将空类初始化为公共业务规则类。
构建对象级别规则类	为业务类的构建对象级别规则类，可验证对象的数据是否有效。
构建属性有效性规则类	为业务类的属性构建有效性规则类，可验证赋值的数据是否有效。
构建读/写属性授权规则类	为业务类的属性构建授权规则类，可限制是否允许读取/赋值该属性。
构建执行过程授权规则类	为业务类的过程构建授权规则类，可限制是否允许执行该过程。
添加枚举标签	为枚举打上标签，可在 UI 界面上被绑定作为列表清单使用。
实现对象本地配置	以指定键值，将配置信息持久化到本地的config文件中。
实现对象公共配置	以指定键值，将配置信息持久化到数据库。
实现类本地配置	以类名为键值，将配置信息持久化到本地的 config 文件中。
实现类公共配置	以类名为键值，将配置信息持久化到数据库。
添加窗体插件类	将普通的窗体类继承自 Phenix.Core.Windows.BaseForm，并在本工程中添加其对应的窗体插件类，以作为本窗体插件的主窗体，使得本工程可嵌入到界面框架 UI 插件队列中。
连接数据库	要使用上述功能，必须先连接到一个已构建过配置库的数据库上。

3.3 首次使用/更改数据库

要使用 Addin 工具，必须先连接到一个已构建过配置库的数据库上。

当第一次使用 Addin 工具时，请在任意一个打开的类编辑框里，点击右键选择“Phenix-连接数据库”菜单：

弹出“连接数据库”对话框：



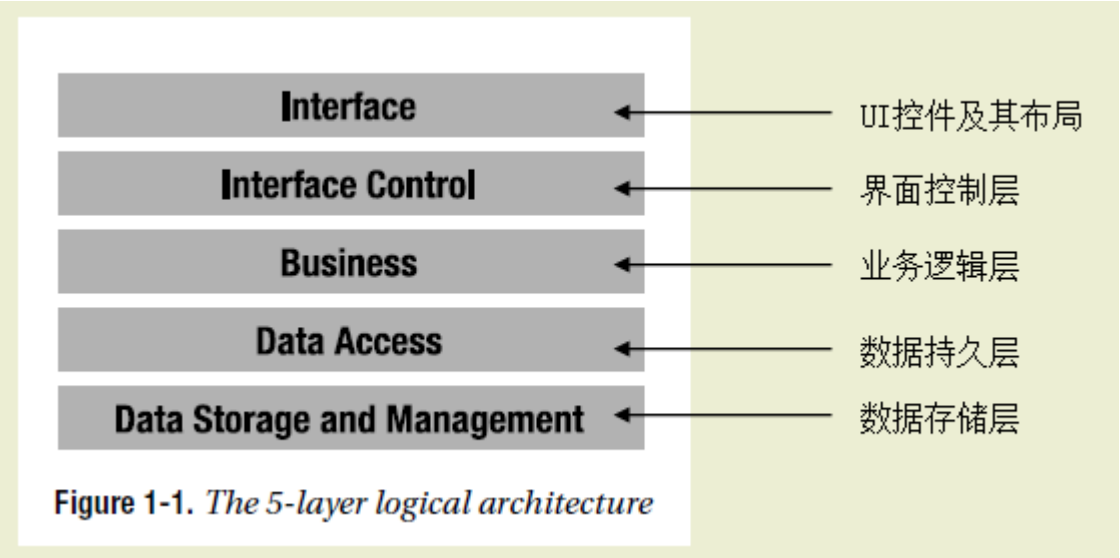
请根据需要以 DBA 权限用户登录数据库，填充好参数后点击“确认”按钮登录。（注意：界面中的“DataSourceKey”输入项，为多数据库应用场景而设置的标记（与 ClassAttribute.DataSourceKey 属性及实体 Fetch()、Save()函数的 DataSourceKey 参数值保持一致），如果不需要则留为空白。）

这样，接下来的任何 Addin 操作都将针对本数据库来执行。下次启动 IDE 时，Addin 会默认使用最近一次的配置参数。

3.4 初始化/编辑业务类

面向对象设计和关系数据设计是应用系统开发中两个不同的过程，业务对象通过关系数据的操作实现业务数据的持久化。而业务对象如何访问关系数据，则涉及到如何将这两套模型粘合在一起的问题。

从逻辑分层架构角度分析，数据持久层是业务对象和关系数据之间的粘合剂，负责业务对象的构建、持久化。为了实现业务对象的构建、持久化，从而产生了对象-关系映射（Object/Relation Mapping，简称 ORM）技术。



上图摘自 CSLA 作者的《Expert C# 2008 Business Objects》。

Phenix 持久层引擎将数据持久层进行了全面的封装，开发者仅需为业务类打上 Mapping 标签就可以实现常规的对象持久化。

下面我们将一步步练习如何构建出一个带 Mapping 代码的业务类。

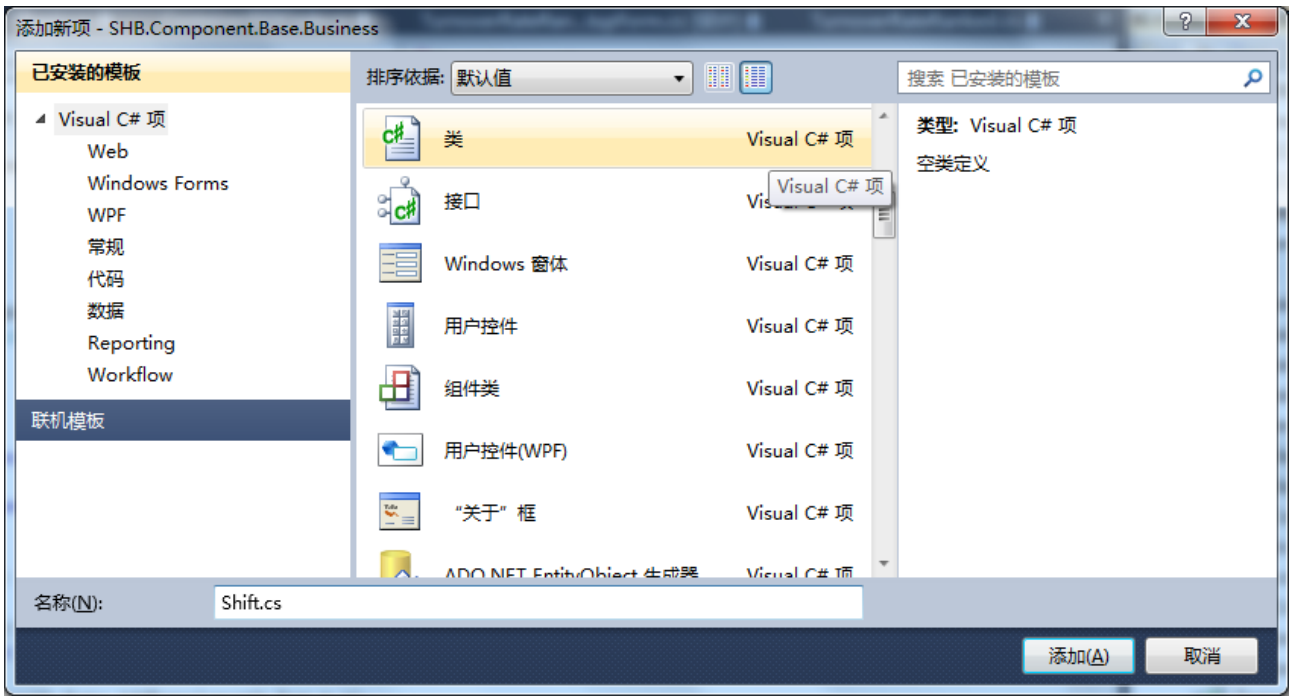
3.4.1 初始化业务类

我们以一个单表为例：

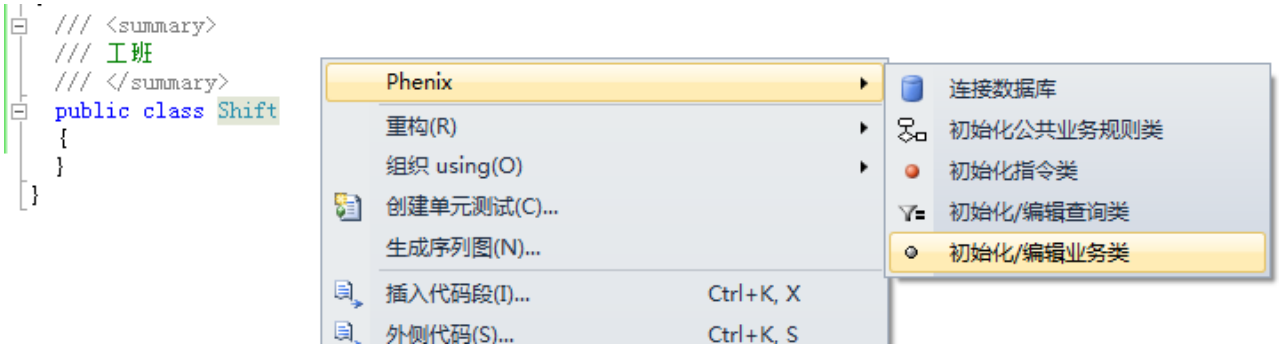
SYS_SHIFT 工班SSF		
SSF_ID	NUMERIC(15)	<pk>
SSF_NAME 名称	VARCHAR(20)	<i>
SSF_CODE 代码(缺省:SSF_NAME拼音码)	VARCHAR(10)	
SSF_REMARK 备注	VARCHAR(100)	
SSF_INPUTER 录入人	VARCHAR(10)	
SSF_INPUTTIME 录入时间	DATE	
SSF_DISABLED 是否禁用	NUMERIC(1)	
Key_1 <pk>		
I_SSF_NAME		

业务类是构建在业务逻辑组件工程中的，一般我们以“.Business”为工程名后缀。

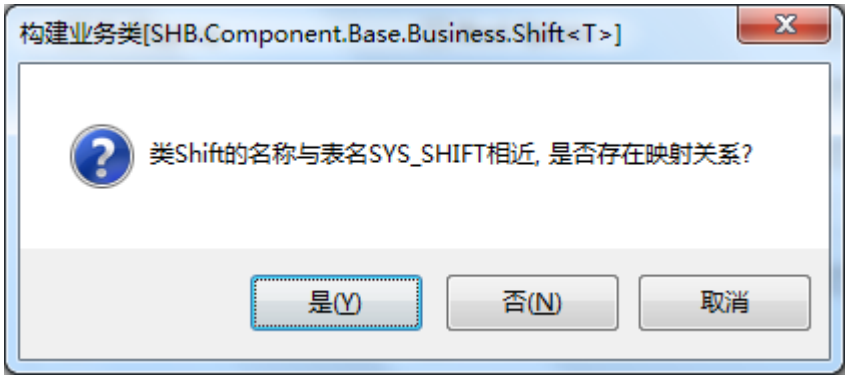
首先，我们在相关的业务逻辑组件工程中建一个空的业务类，类名尽可能和表名一致（剔除表名的前缀、后缀）：



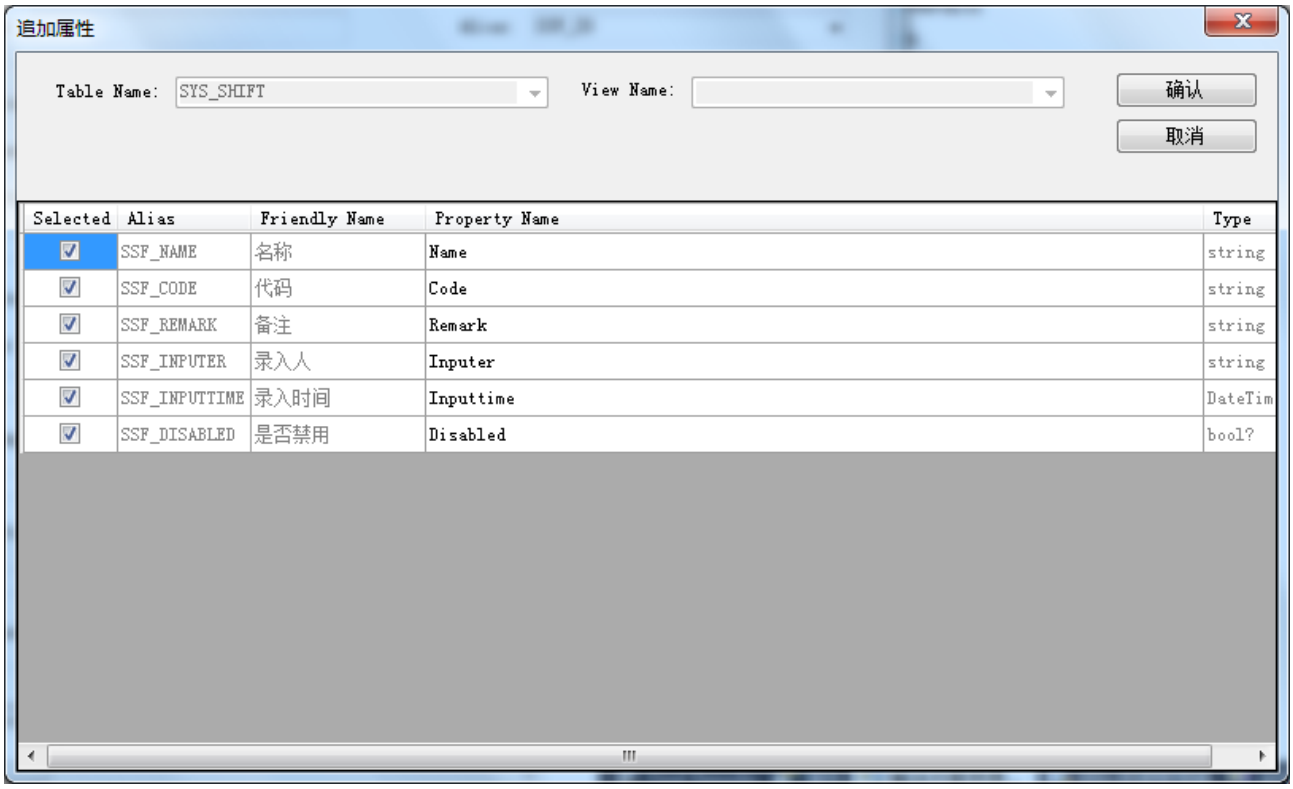
然后，在类编辑界面上用鼠标右键点出浮动菜单，选择：



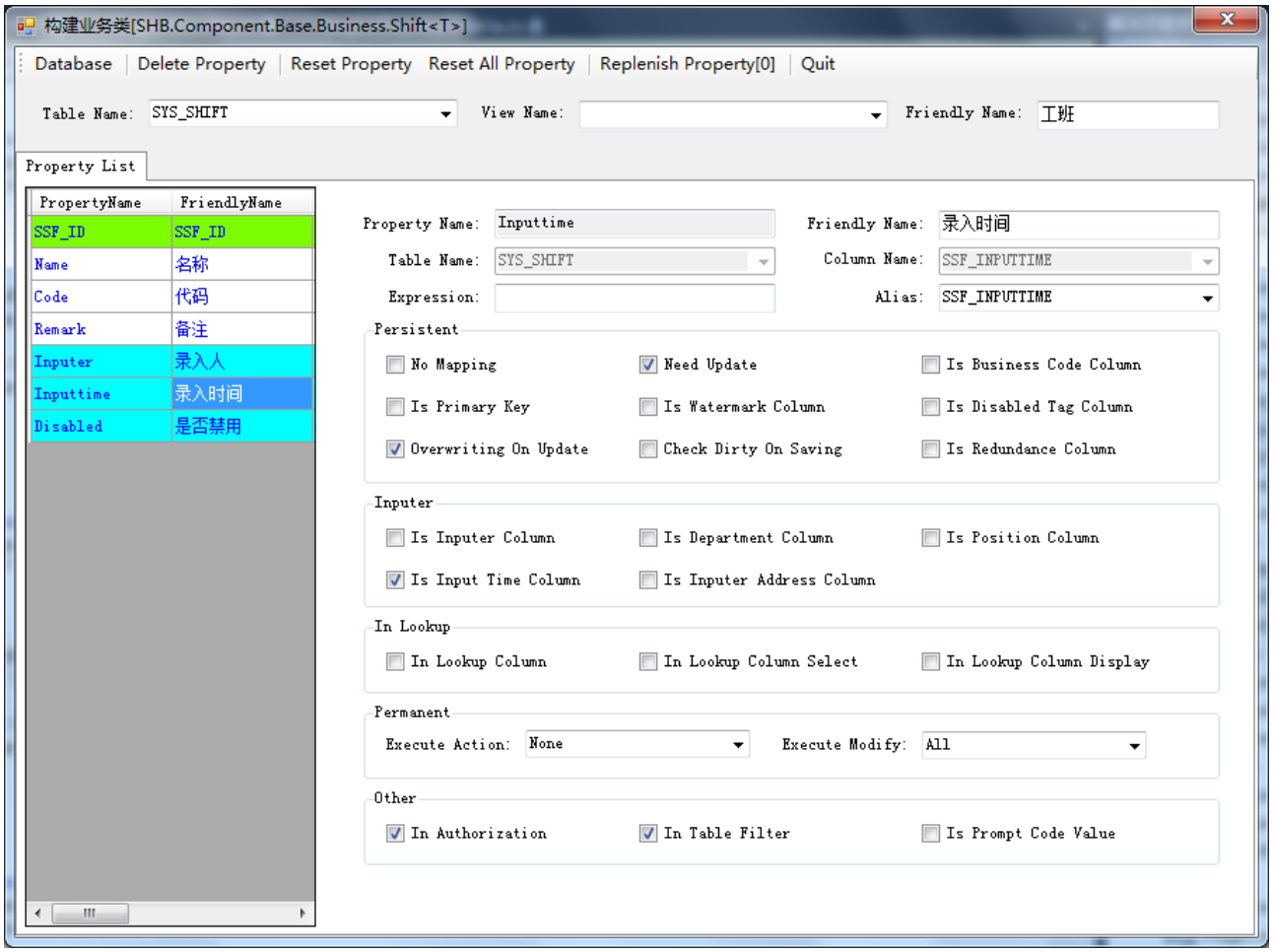
将会弹出提示：



点击“是”，则会根据表“SYS_SHIFT”数据字典信息自动生成业务类代码，并缺省生成主键/外键的映射代码。接下来，你可以对非主键/外键的字段，选择性生成对应的数据属性代码：



如上图，点击清单“selected”项可全选，只有被选中的字段才会生成对应的数据属性代码：



此时，类Shift已生成了代码：

```
  /// <summary>
  /// 工班
  /// </summary>
  [Serializable]
  public class Shift : Shift<Shift>
  {
  }

  /// <summary>
  /// 工班清单
  /// </summary>
  [Serializable]
  public class ShiftList : Phenix.Business.BusinessListBase<ShiftList, Shift>
  {
  }

  /// <summary>
  /// 工班
  /// </summary>
  [Phenix.Core.Mapping.ClassAttribute("SYS_SHIFT", FriendlyName = "工班"),
System.SerializableAttribute(), System.ComponentModel.DisplayNameAttribute("工班")]
  public abstract class Shift<T> : Phenix.Business.BusinessBase<T> where T : Shift<T>
  {
       /// <summary>
        /// SSF_ID
        /// </summary>
        public static readonly Phenix.Business.PropertyInfo<long?> SSF_IDProperty =
RegisterProperty<long?>(c => c.SSF_ID);
        [Phenix.Core.Mapping.FieldUniqueAttribute("PK_SYS_SHIFT")]
        [Phenix.Core.Mapping.Field(FriendlyName = "SSF_ID", TableName = "SYS_SHIFT", ColumnName =
"SSF_ID", IsPrimaryKey = true, NeedUpdate = true)]
        private long? _SSF_ID;
        /// <summary>
        /// SSF_ID
        /// </summary>
        [System.ComponentModel.DisplayName("SSF_ID")]
        public long? SSF_ID
        {
            get { return GetProperty(SSF_IDProperty, _SSF_ID); }
            //set { SetProperty(SSF_IDProperty, ref _SSF_ID, value); }
        }

        [System.ComponentModel.Browsable(false)]
        [System.ComponentModel.DataAnnotations.Display(AutoGenerateField = false)]
```

```
        public override string PrimaryKey
        {
          get { return String.Format("{0}", SSF_ID); }
        }

        /// <summary>
        /// 名称
        /// </summary>
        public static readonly Phenix.Business.PropertyInfo<string> NameProperty =
RegisterProperty<string>(c => c.Name);
        [Phenix.Core.Mapping.FieldUniqueAttribute("I_SSF_NAME")]
        [Phenix.Core.Mapping.FieldRuleAttribute(StringOnImeMode = true, PinyinCodeProperty = "Code")]
        [Phenix.Core.Mapping.Field(FriendlyName = "名称", Alias = "SSF_NAME", TableName = "SYS_SHIFT",
ColumnName = "SSF_NAME", NeedUpdate = true, InLookUpColumn = true, InLookUpColumnDisplay = true)]
        private string _name;
        /// <summary>
        /// 名称
        /// </summary>
        [System.ComponentModel.DisplayName("名称")]
        public string Name
        {
            get { return GetProperty(NameProperty, _name); }
            set { SetProperty(NameProperty, ref _name, value); }
        }

        /// <summary>
        /// 代码
        /// </summary>
        public static readonly Phenix.Business.PropertyInfo<string> CodeProperty =
RegisterProperty<string>(c => c.Code);
        [Phenix.Core.Mapping.FieldRuleAttribute(StringTrim = true, StringUpperCase = true,
StringOnImeMode = false)]
        [Phenix.Core.Mapping.Field(FriendlyName = "代码", Alias = "SSF_CODE", TableName = "SYS_SHIFT",
ColumnName = "SSF_CODE", NeedUpdate = true, InLookUpColumn = true, InLookUpColumnSelect = true)]
        private string _code;
        /// <summary>
        /// 代码
        /// </summary>
        [System.ComponentModel.DisplayName("代码")]
        public string Code
        {
            get { return GetProperty(CodeProperty, _code); }
            set { SetProperty(CodeProperty, ref _code, value); }
        }

        /// <summary>
```

```
        /// 备注
        /// </summary>
        public static readonly Phenix.Business.PropertyInfo<string> RemarkProperty =
RegisterProperty<string>(c => c.Remark);
        [Phenix.Core.Mapping.Field(FriendlyName = "备注", Alias = "SSF_REMARK", TableName = "SYS_SHIFT",
ColumnName = "SSF_REMARK", NeedUpdate = true)]
        private string _remark;
        /// <summary>
        /// 备注
        /// </summary>
        [System.ComponentModel.DisplayName("备注")]
        public string Remark
        {
            get { return GetProperty(RemarkProperty, _remark); }
            set { SetProperty(RemarkProperty, ref _remark, value); }
        }

        /// <summary>
        /// 录入人
        /// </summary>
        public static readonly Phenix.Business.PropertyInfo<string> InputerProperty =
RegisterProperty<string>(c => c.Inputer);
        [Phenix.Core.Mapping.Field(FriendlyName = "录入人", Alias = "SSF_INPUTER", TableName =
"SYS_SHIFT", ColumnName = "SSF_INPUTER", NeedUpdate = true, OverwritingOnUpdate = true, IsInputerColumn
= true)]
        private string _inputer;
        /// <summary>
        /// 录入人
        /// </summary>
        [System.ComponentModel.DisplayName("录入人")]
        public string Inputer
        {
            get { return GetProperty(InputerProperty, _inputer); }
            set { SetProperty(InputerProperty, ref _inputer, value); }
        }

        /// <summary>
        /// 录入时间
        /// </summary>
        public static readonly Phenix.Business.PropertyInfo<DateTime?> InputtimeProperty =
RegisterProperty<DateTime?>(c => c.Inputtime);
        [Phenix.Core.Mapping.Field(FriendlyName = "录入时间", Alias = "SSF_INPUTTIME", TableName =
"SYS_SHIFT", ColumnName = "SSF_INPUTTIME", NeedUpdate = true, OverwritingOnUpdate = true,
IsInputTimeColumn = true)]
        private DateTime? _inputtime;
        /// <summary>
```

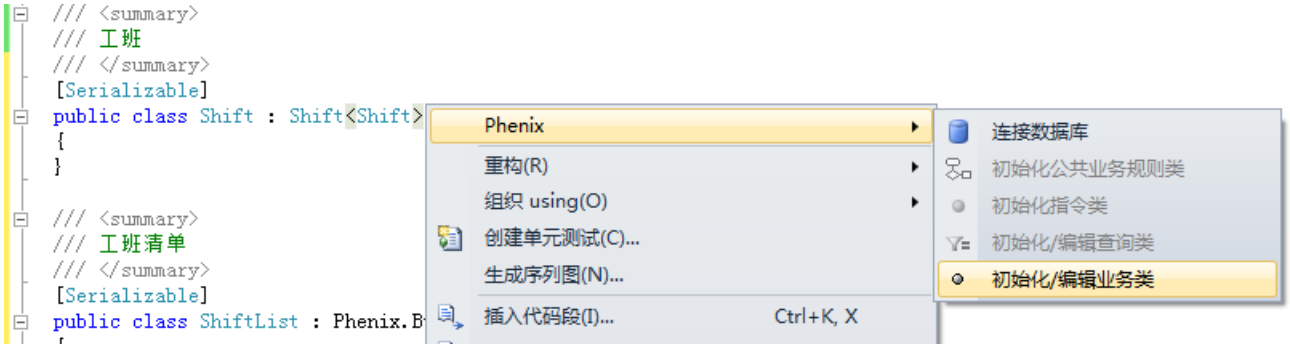
```
        /// 录入时间
        /// </summary>
        [System.ComponentModel.DisplayName("录入时间")]
        public DateTime? Inputtime
        {
            get { return GetProperty(InputtimeProperty, _inputtime); }
            set { SetProperty(InputtimeProperty, ref _inputtime, value); }
        }
```

3.4.2 编辑业务类

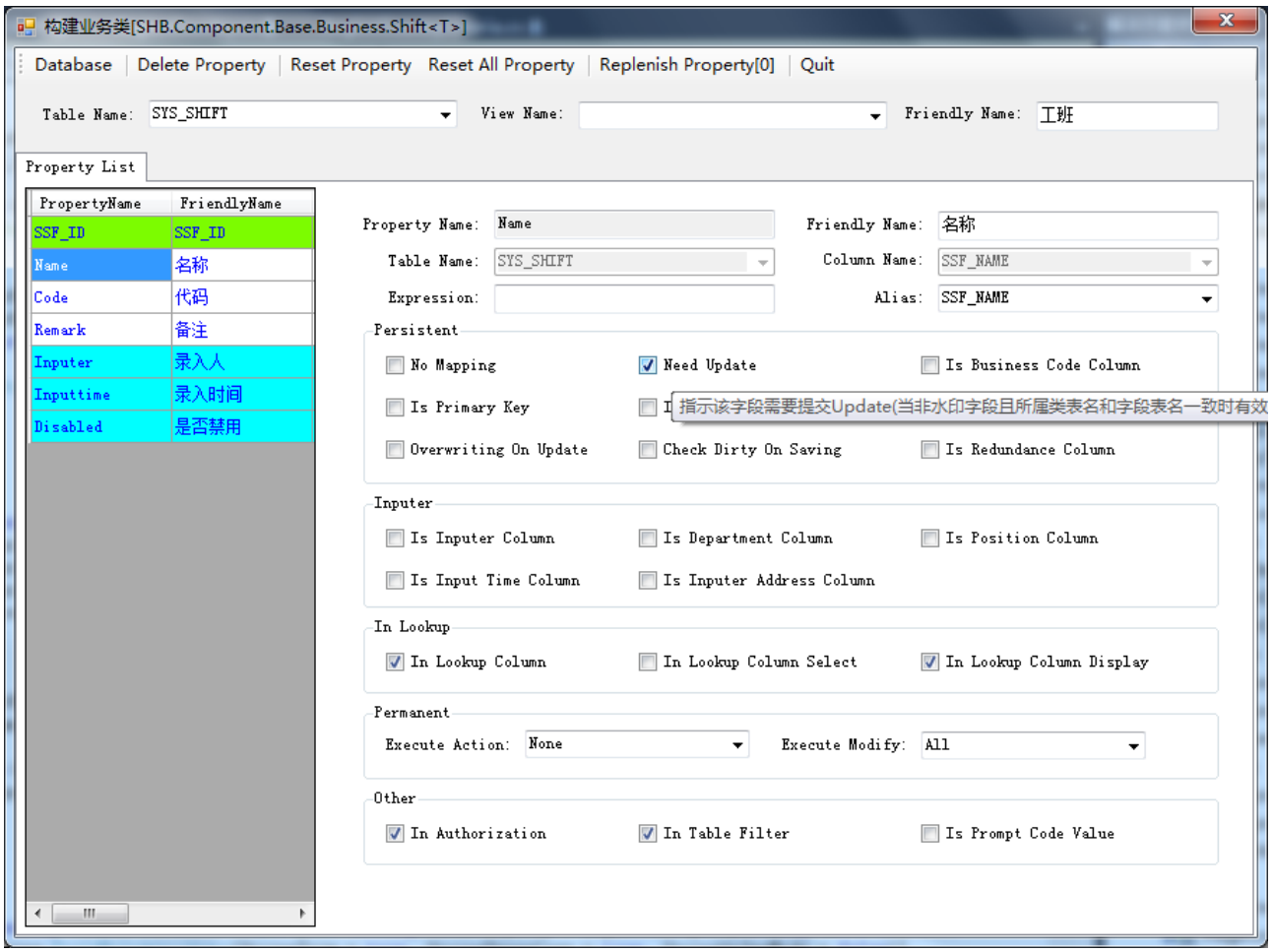
一般情况下，我们无需再对 Mapping 细节进行修改。但是，在一些特殊的业务逻辑下，比如不希望某个字段发生改变后被提交到数据库，这时，我们可以直接去修改相关的代码（见下例黄底代码被删除）：

```
        [Phenix.Core.Mapping.FieldUniqueAttribute("I_SSF_NAME")]
        [Phenix.Core.Mapping.FieldRuleAttribute(StringOnImeMode = true, PinyinCodeProperty = "Code")]
        [Phenix.Core.Mapping.Field(FriendlyName = "名称", Alias = "SSF_NAME", TableName = "SYS_SHIFT",
ColumnName = "SSF_NAME", NeedUpdate = true, InLookUpColumn = true, InLookUpColumnDisplay = true)]
        private string _name;
```

也可以在类编辑界面上用鼠标右键点出浮动菜单，选择：



在对话框上进行修改：



在“Property List”清单中选择好数据属性，将“Persistent”栏“Need Update”的勾选框点去勾选即可。

3.4.3 与数据字典保持一致

当系统需求发生变更时，可能在系统设计上会对某些表的表字段进行增删改，此时业务类也可能要做相应的修改。Phenix丶能为你实现：

1，追加字段到业务类上；

2，删除业务类上的数据属性及其关联字段；

3，对一些关键的映射关系进行提醒；

4，重置字段的映射信息；

Phenix丶无法替你做的是：

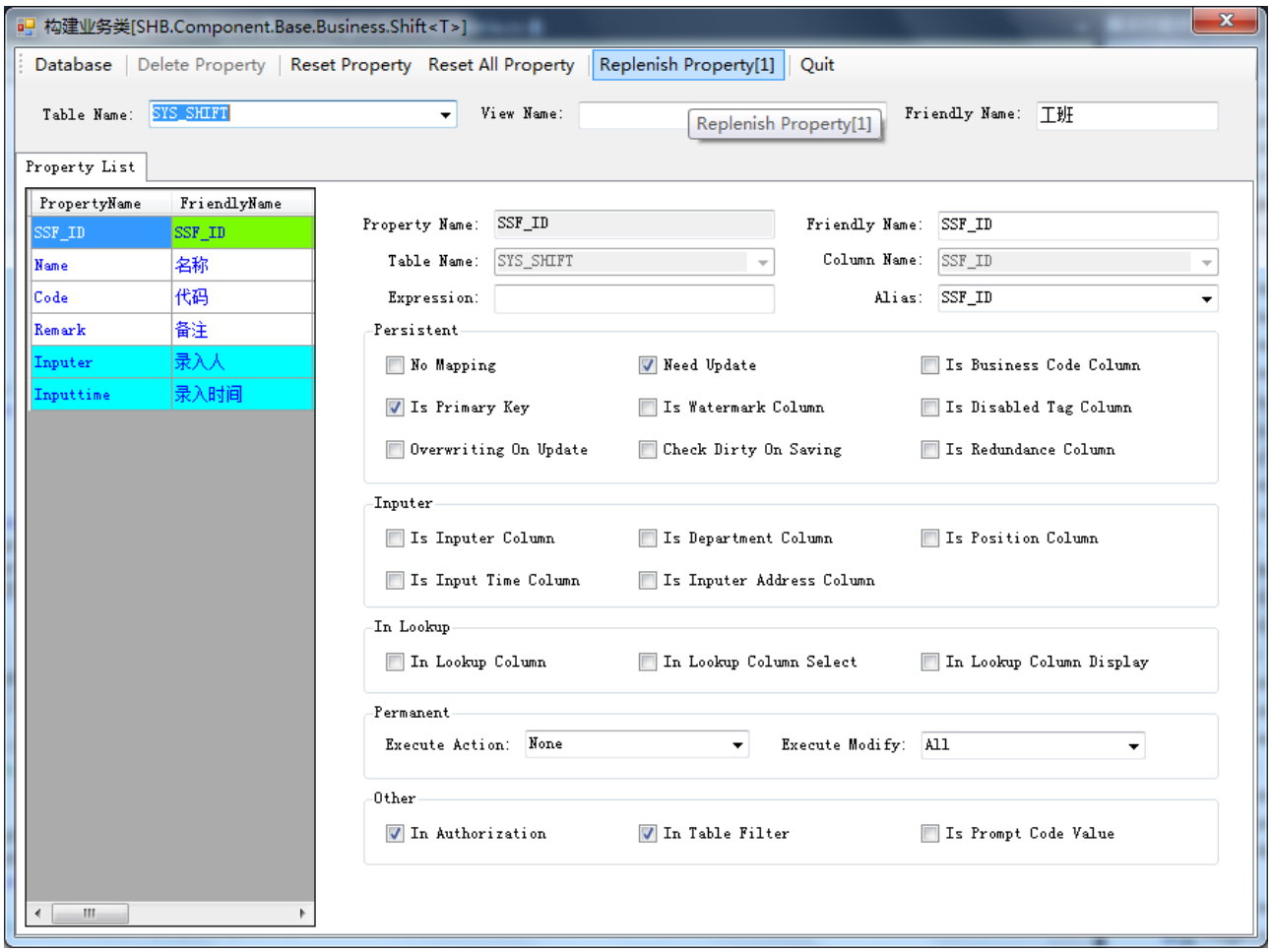
1， 修改业务类上数据属性及其关联字段的数据类型；

2， 业务类上数据属性及其关联字段的名称，自动随着表的数据字典改变而改变；

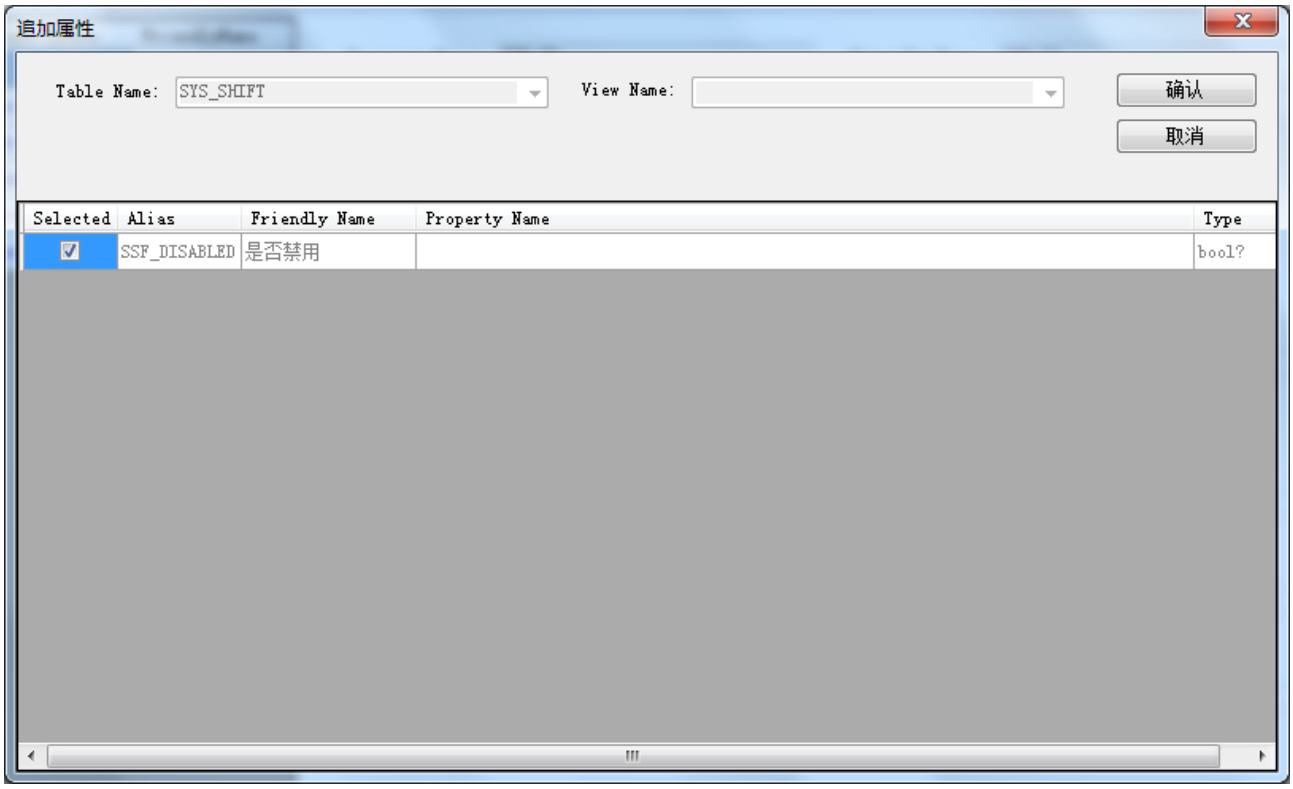
3.4.3.1 追加字段到业务类上

当数据库表中新增字段而希望本业务类也添加相应的数据属性，比如表“SYS_SHIFT”新增了

字段"SSF_DISABLED"，则：

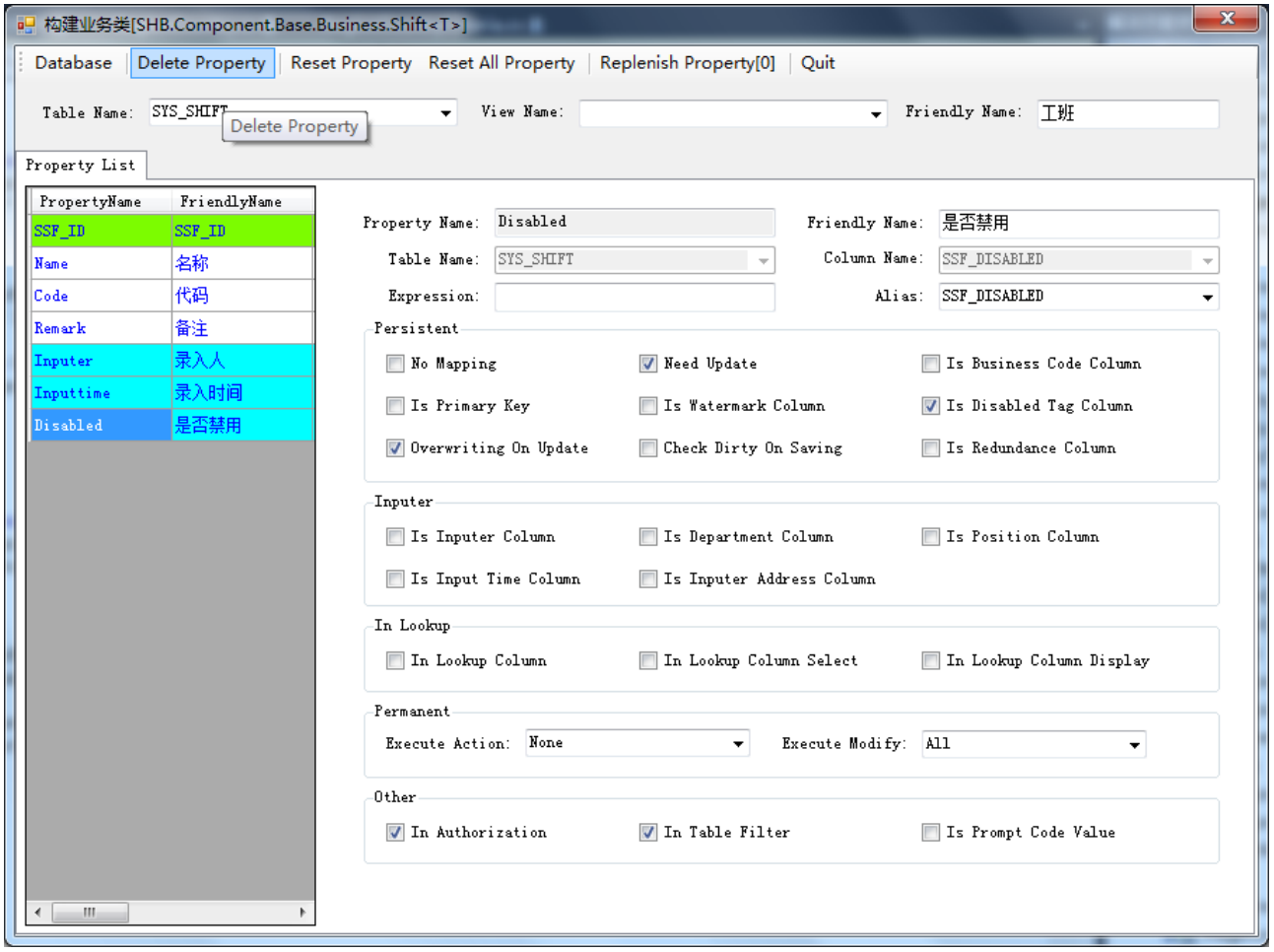


“Replenish Property”功能按钮附带的中括号内会显示业务类缺失了多少数据属性，点击它可追加数据属性：



3.4.3.2 删除业务类上的数据属性及其关联的字段

在“Property List”清单中选择好数据属性，点击“Delete Property”功能按钮即可：

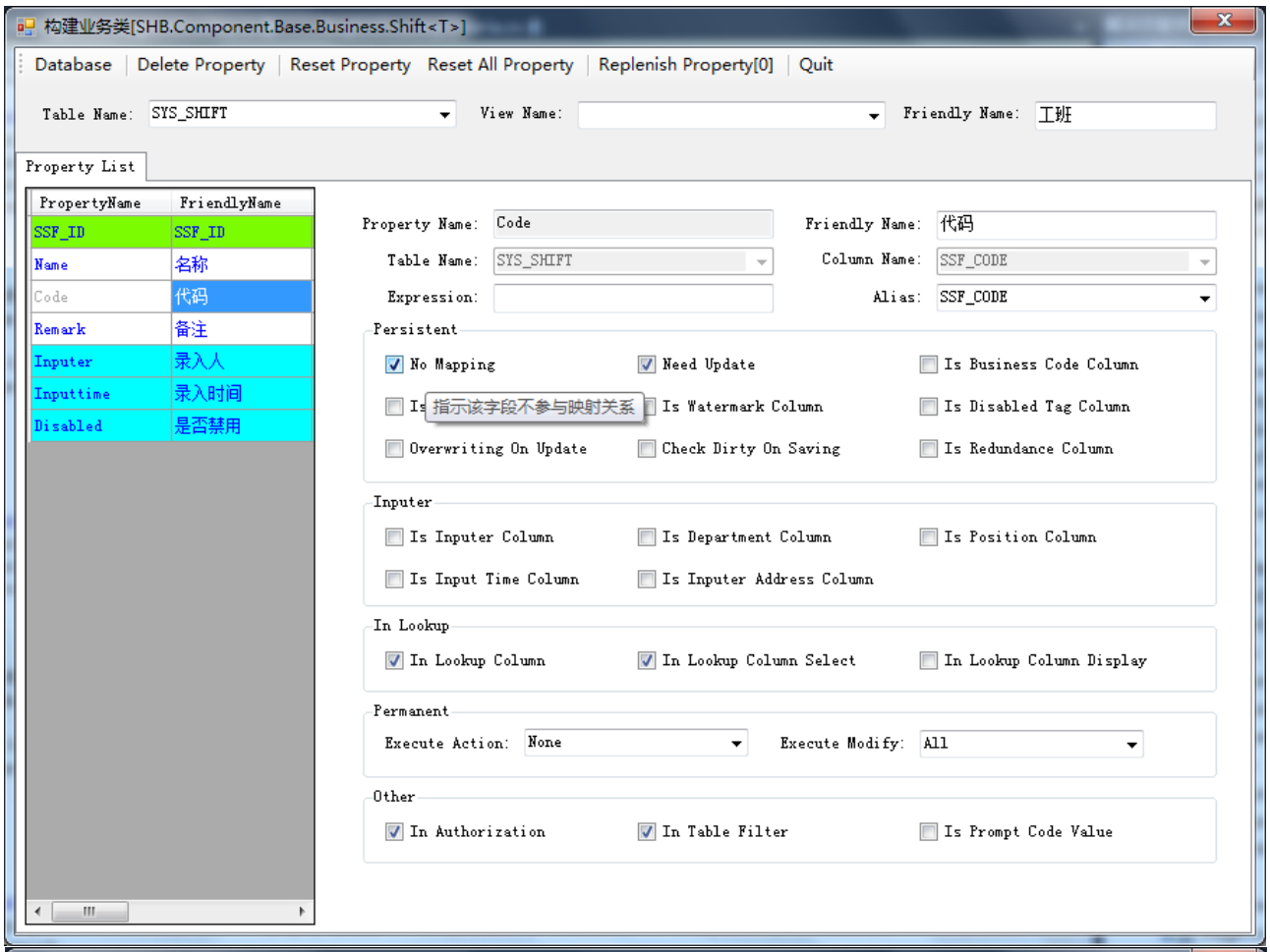


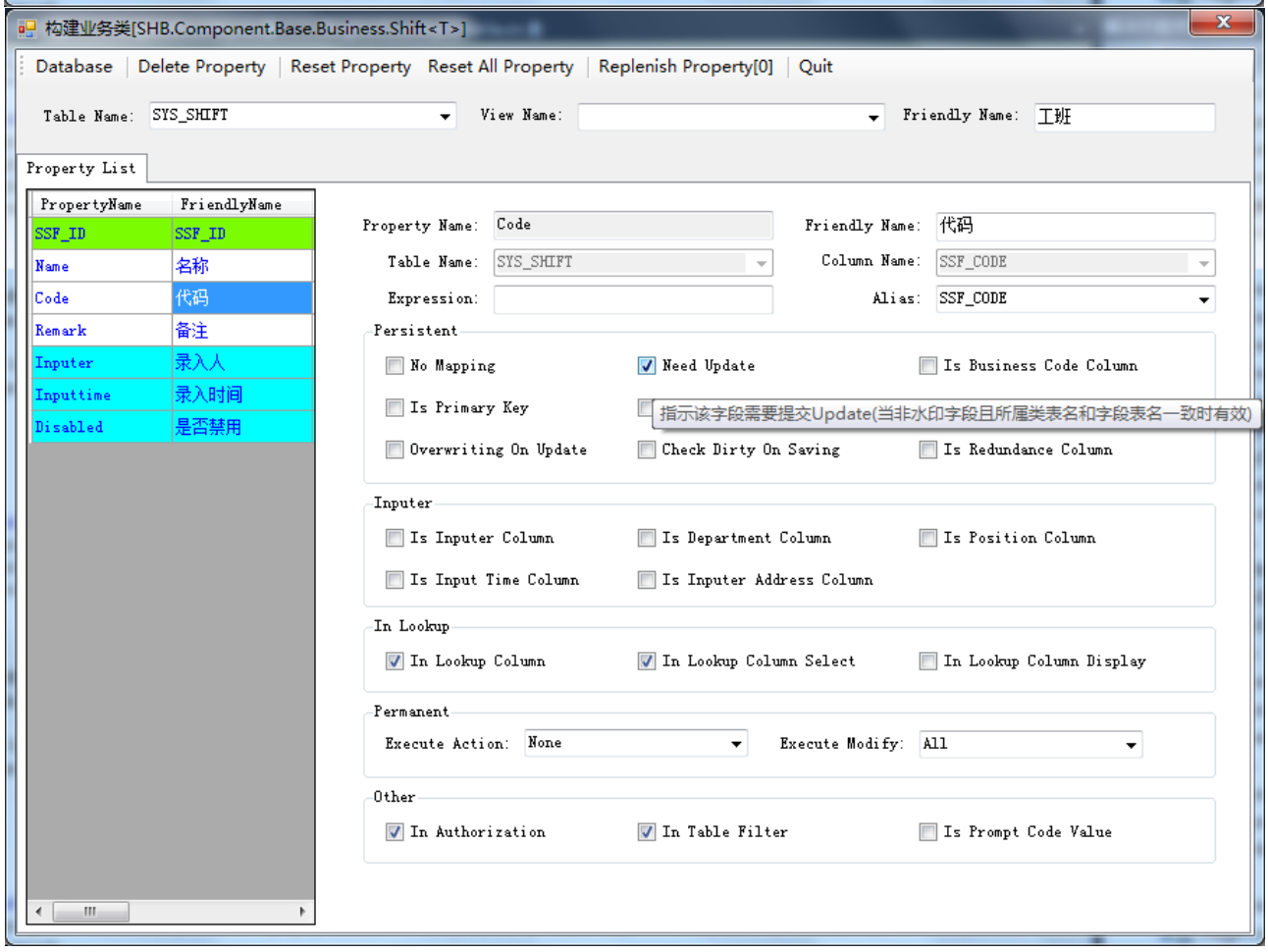
注意：主键数据属性及其关联的字段是不允许删除的。

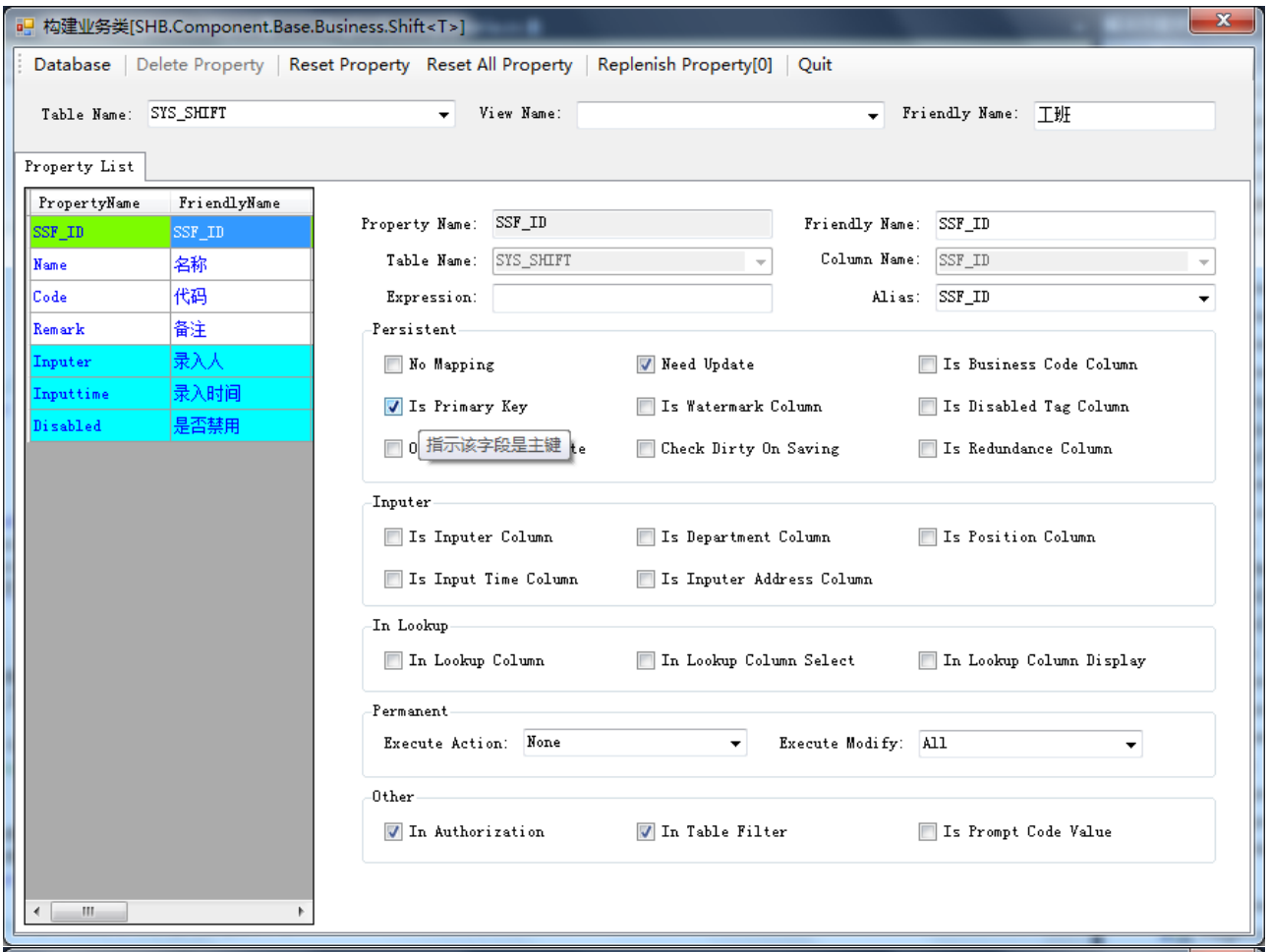
3.4.3.3 对一些关键的映射关系进行提醒

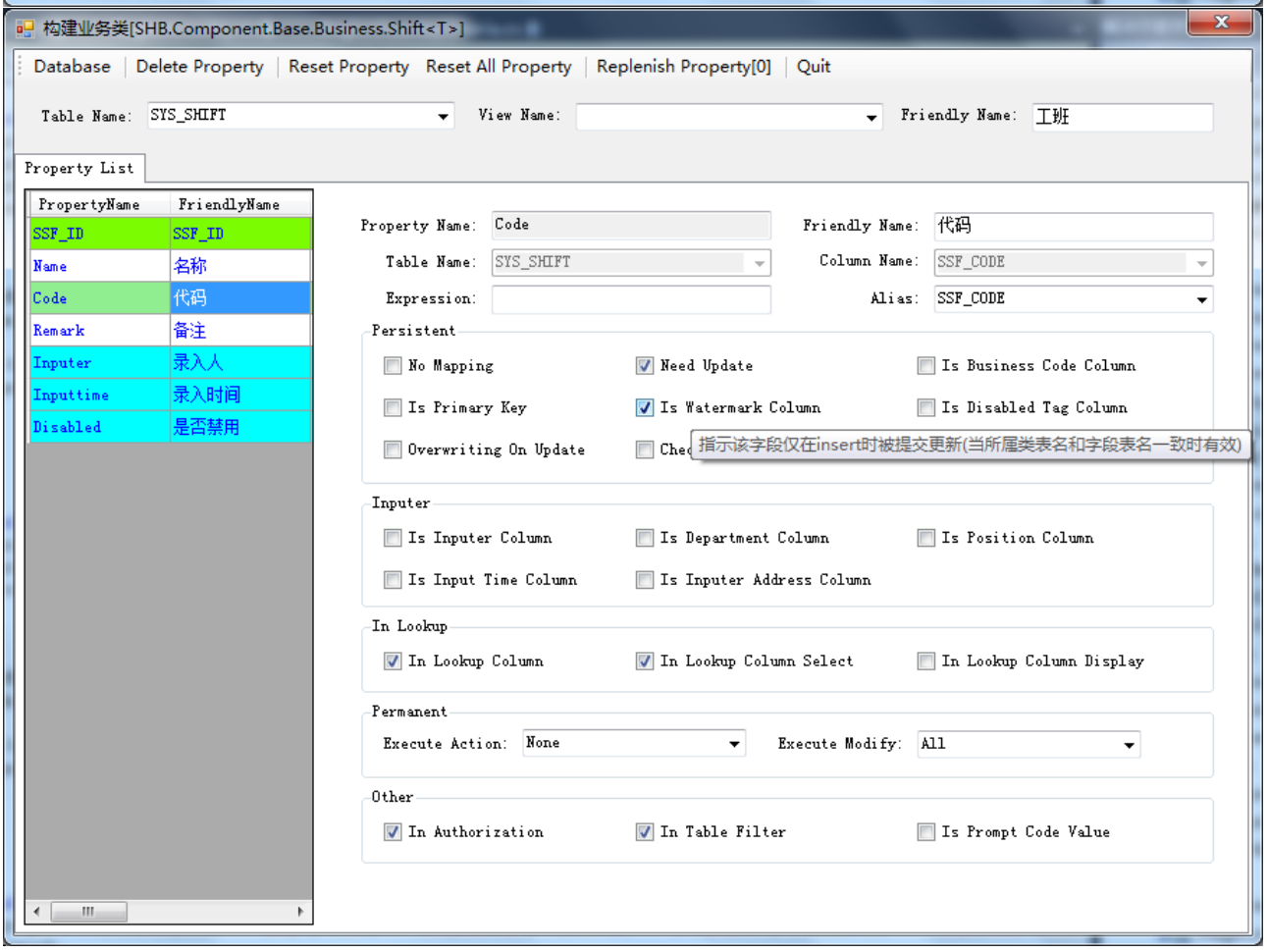
“Property List”清单上记录的底色和字体颜色，是对下面的映射关系进行提醒：

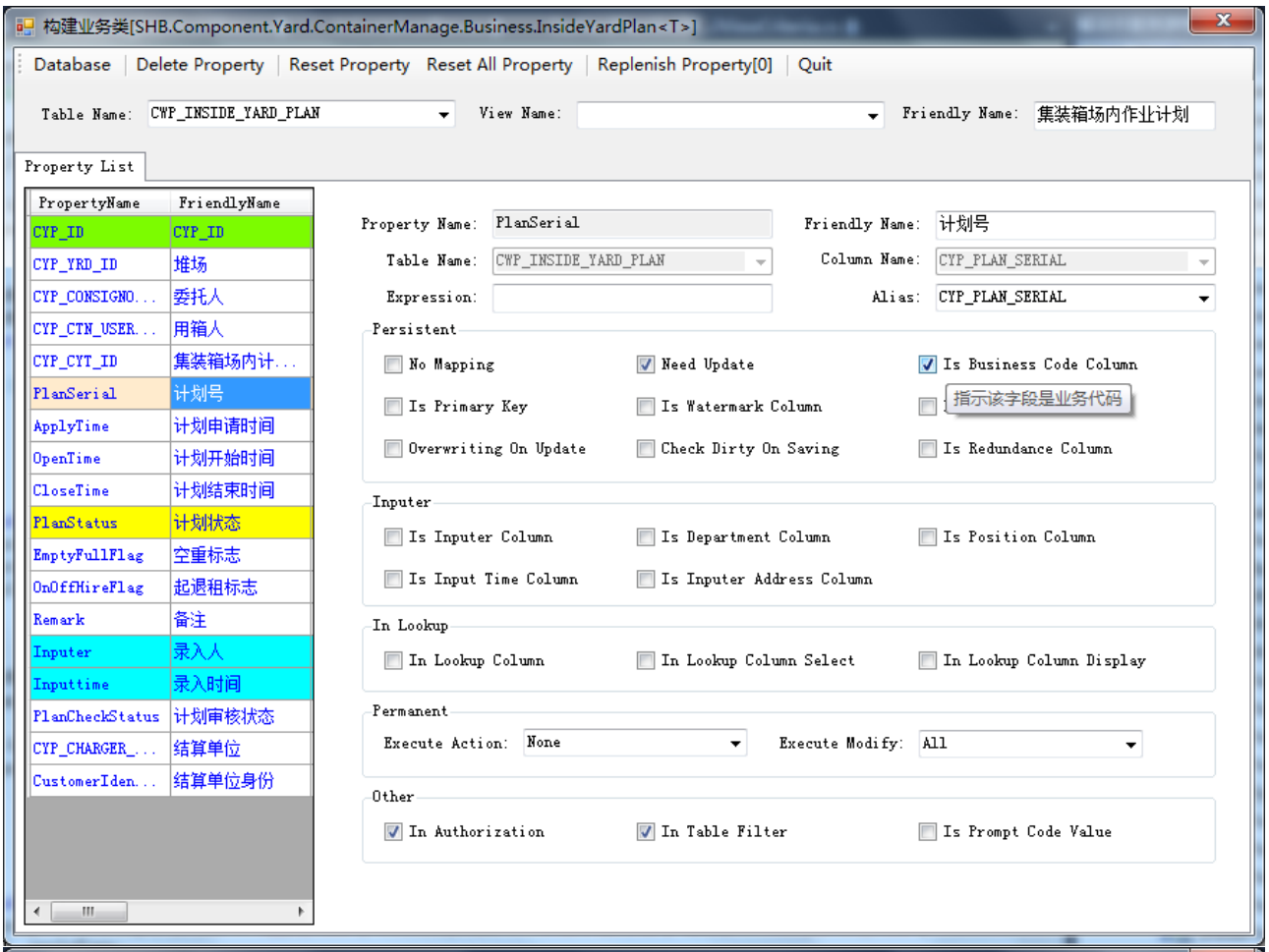
映射关系	字体颜色	底色
No Mapping	Color.LightGray	
Need Update && Is Reserved Column	Color.BurlyWood	
Need Update && Not Nullable	Color.Blue	
Need Update && Nullable	Color.Black	
Not Need Update	Color.Gray	
Is Primary Key		Color.LawnGreen
Is Watermark Column		Color.LightGreen
Is Business Code Column		Color.BlanchedAlmond
Overwriting On Update		Color.Aqua
Check Dirty On Saving		Color.Yellow
找不到数据库字段		Color.Red

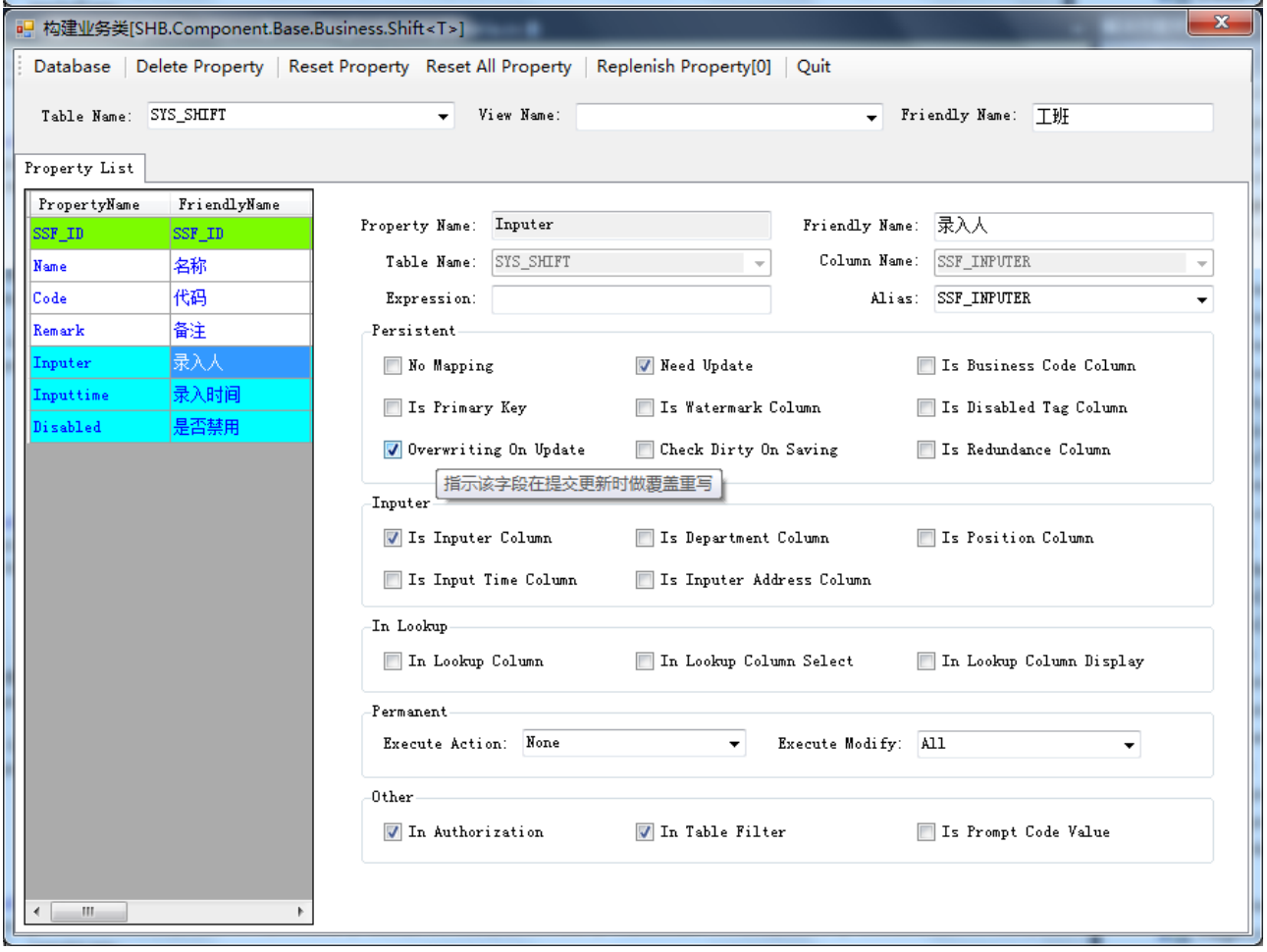

构建业务类[SHB.Component.Base.Business.Shift<T>]
Database
Delete Property
Reset Property
Reset All Property
Replenish Property[0]
Quit
Table Name: SYS_SHIFT
View Name:
Friendly Name: 工班
Property List
PropertyName
FriendlyName
SSF_ID
SSF_ID
Name
名称
Code
代码
Remark
备注
Inputer
录入人
Inputtime
录入时间
Disabled
是否禁用
Property Name: Code
Friendly Name: 代码
Table Name: SYS_SHIFT
Column Name: SSF_CODE
Expression:
Alias: SSF_CODE
Persistent
No Mapping
Need Update
Is Business Code Column
Is Primary Key
Is Watermark Column
Is Disabled Tag Column
Overwriting On Update
Check Dirty On Saving
Is Redundance Column
指示该字段不参与映射关系
Inputer
Is Inputer Column
Is Department Column
Is Position Column
Is Input Time Column
Is Inputer Address Column
In Lookup
In Lookup Column
In Lookup Column Select
In Lookup Column Display
Permanent
Execute Action: None
Execute Modify: All
Other
In Authorization
In Table Filter
Is Prompt Code Value

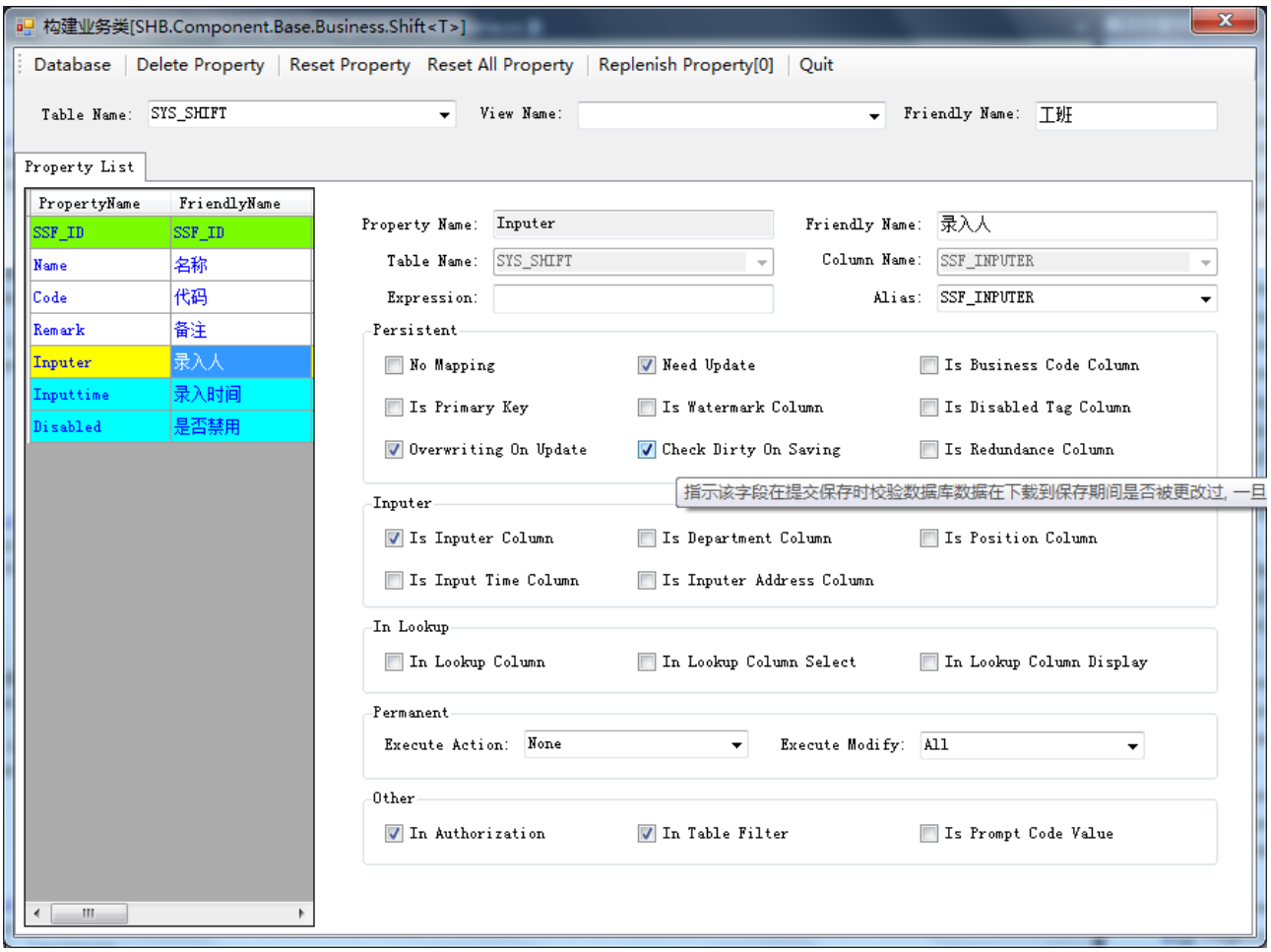

构建业务类[SHB.Component.Base.Business.Shift<T>]
Database
Delete Property
Reset Property
Reset All Property
Replenish Property[0]
Quit
Table Name: SYS_SHIFT
View Name:
Friendly Name: 工班
Property List
PropertyName
FriendlyName
SSF_ID
SSF_ID
Name
名称
Code
代码
Remark
备注
Inputer
录入人
Inputtime
录入时间
Disabled
是否禁用
Property Name: Code
Friendly Name: 代码
Table Name: SYS_SHIFT
Column Name: SSF_CODE
Expression:
Alias: SSF_CODE
Persistent
No Mapping
Need Update
Is Business Code Column
Is Primary Key
Overwriting On Update
Check Dirty On Saving
Is Redundance Column
指示该字段需要提交Update(当非水印字段且所属类表名和字段表名一致时有效)
Inputer
Is Inputer Column
Is Department Column
Is Position Column
Is Input Time Column
Is Inputer Address Column
In Lookup
In Lookup Column
In Lookup Column Select
In Lookup Column Display
Permanent
Execute Action: None
Execute Modify: All
Other
In Authorization
In Table Filter
Is Prompt Code Value


构建业务类[SHB.Component.Base.Business.Shift<T>]
Database
Delete Property
Reset Property
Reset All Property
Replenish Property[0]
Quit
Table Name: SYS_SHIFT
View Name:
Friendly Name: 工班
Property List
PropertyName FriendlyName
SSF_ID SSF_ID
Name 名称
Code 代码
Remark 备注
Inputer 录入人
Inputtime 录入时间
Disabled 是否禁用
Property Name: SSF_ID
Friendly Name: SSF_ID
Table Name: SYS_SHIFT
Column Name: SSF_ID
Expression:
Alias: SSF_ID
Persistent
No Mapping
Need Update
Is Business Code Column
Is Primary Key
Is Watermark Column
Is Disabled Tag Column
指示该字段是主键
Check Dirty On Saving
Is Redundance Column
Inputer
Is Inputer Column
Is Department Column
Is Position Column
Is Input Time Column
Is Inputer Address Column
In Lookup
In Lookup Column
In Lookup Column Select
In Lookup Column Display
Permanent
Execute Action: None
Execute Modify: All
Other
In Authorization
In Table Filter
Is Prompt Code Value


构建业务类[SHB.Component.Base.Business.Shift<T>]
Database
Delete Property
Reset Property
Reset All Property
Replenish Property[0]
Quit
Table Name: SYS_SHIFT
View Name:
Friendly Name: 工班
Property List
PropertyName FriendlyName
SSF_ID SSF_ID
Name 名称
Code 代码
Remark 备注
Inputer 录入人
Inputtime 录入时间
Disabled 是否禁用
Property Name: Code
Friendly Name: 代码
Table Name: SYS_SHIFT
Column Name: SSF_CODE
Expression:
Alias: SSF_CODE
Persistent
No Mapping
Need Update
Is Business Code Column
Is Primary Key
Is Watermark Column
Is Disabled Tag Column
Overwriting On Update
指示该字段仅在insert时被提交更新(当所属类表名和字段表名一致时有效)
Inputer
Is Inputer Column
Is Department Column
Is Position Column
Is Input Time Column
Is Inputer Address Column
In Lookup
In Lookup Column
In Lookup Column Select
In Lookup Column Display
Permanent
Execute Action: None
Execute Modify: All
Other
In Authorization
In Table Filter
Is Prompt Code Value

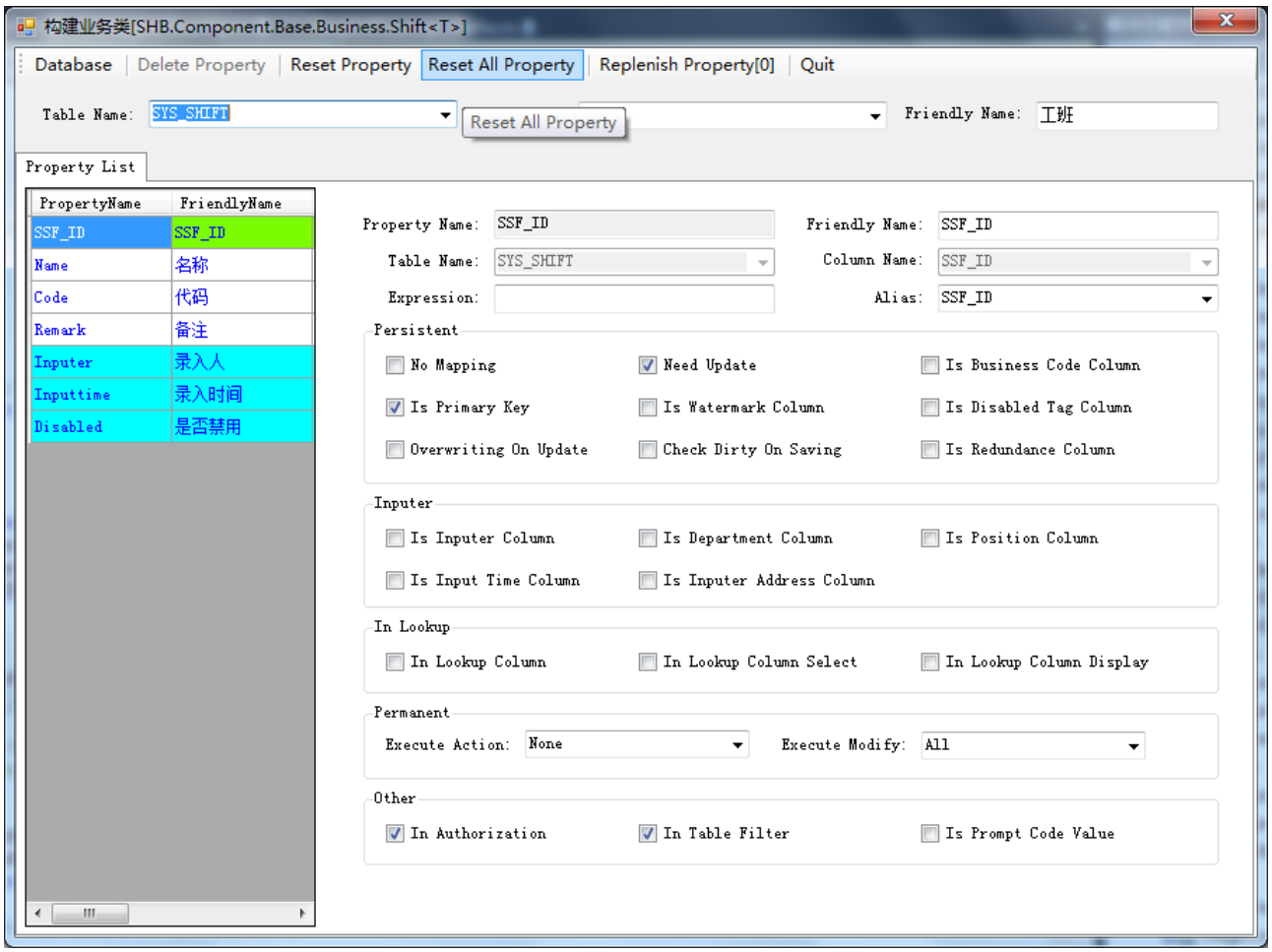

构建业务类[SHB.Component.Yard.ContainerManage.Business.InsideYardPlan<T>]
Database
Delete Property
Reset Property
Reset All Property
Replenish Property[0]
Quit
Table Name: CWP_INSIDE_YARD_PLAN
View Name:
Friendly Name: 集装箱场内作业计划
Property List
PropertyName FriendlyName
CYP_ID CYP_ID
CYP_YRD_ID 堆场
CYP_CONSIGNO... 委托人
CYP_CTN_USER... 用箱人
CYP_CYT_ID 集装箱场内计...
PlanSerial 计划号
ApplyTime 计划申请时间
OpenTime 计划开始时间
CloseTime 计划结束时间
PlanStatus 计划状态
EmptyFullFlag 空重标志
OnOffHireFlag 起退租标志
Remark 备注
Inputer 录入人
Inputtime 录入时间
PlanCheckStatus 计划审核状态
CYP_CHARGER_... 结算单位
CustomerIden... 结算单位身份
Property Name: PlanSerial
Friendly Name: 计划号
Table Name: CWP_INSIDE_YARD_PLAN
Column Name: CYP_PLAN_SERIAL
Expression:
Alias: CYP_PLAN_SERIAL
Persistent
No Mapping
Need Update
Is Business Code Column
Is Primary Key
Is Watermark Column
指示该字段是业务代码
Overwriting On Update
Check Dirty On Saving
Is Redundance Column
Inputer
Is Inputer Column
Is Department Column
Is Position Column
Is Input Time Column
Is Inputer Address Column
In Lookup
In Lookup Column
In Lookup Column Select
In Lookup Column Display
Permanent
Execute Action: None
Execute Modify: All
Other
In Authorization
In Table Filter
Is Prompt Code Value


构建业务类[SHB.Component.Base.Business.Shift<T>]
Database
Delete Property
Reset Property
Reset All Property
Replenish Property[0]
Quit
Table Name: SYS_SHIFT
View Name:
Friendly Name: 工班
Property List
PropertyName FriendlyName
SSF_ID SSF_ID
Name 名称
Code 代码
Remark 备注
Inputer 录入人
Inputtime 录入时间
Disabled 是否禁用
Property Name: Inputer
Friendly Name: 录入人
Table Name: SYS_SHIFT
Column Name: SSF_INPUTER
Expression:
Alias: SSF_INPUTER
Persistent
No Mapping
Need Update
Is Business Code Column
Is Primary Key
Is Watermark Column
Is Disabled Tag Column
Overwriting On Update
Check Dirty On Saving
Is Redundance Column
指示该字段在提交更新时做覆盖重写
Inputer
Is Inputer Column
Is Department Column
Is Position Column
Is Input Time Column
Is Inputer Address Column
In Lookup
In Lookup Column
In Lookup Column Select
In Lookup Column Display
Permanent
Execute Action: None
Execute Modify: All
Other
In Authorization
In Table Filter
Is Prompt Code Value



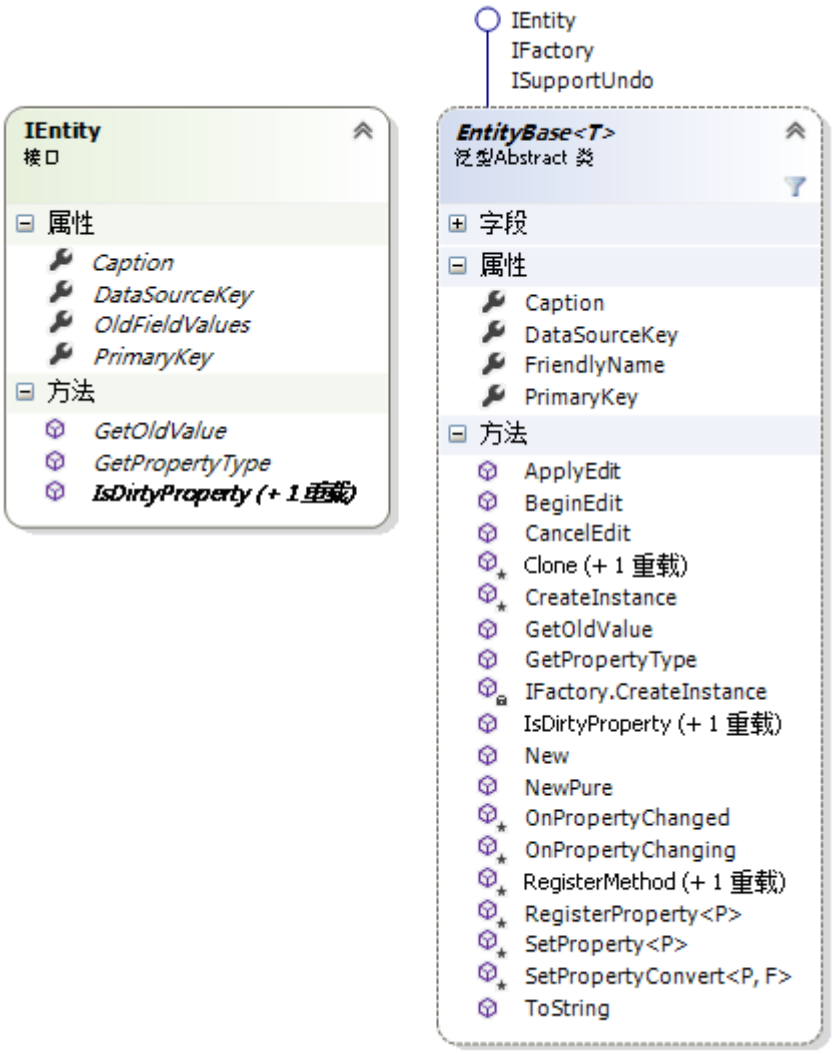
3.4.3.4 重置字段的映射信息

Phenix丶提供对指定数据属性、全部属性映射信息的重置功能，即恢复到缺省值：



3.5 生成轻量级的 Entity 类

在特殊业务场景下，需要处理大批量的数据，而且并不用绑定到界面，也无需使用到 CSLA 的回滚机制，仅仅是读取遍历而已。此时，CSLA 的业务类就显得有点重，我们需要能操作一种轻量级的 Entity 对象，并且是可以快速 Fetch 到内存的。

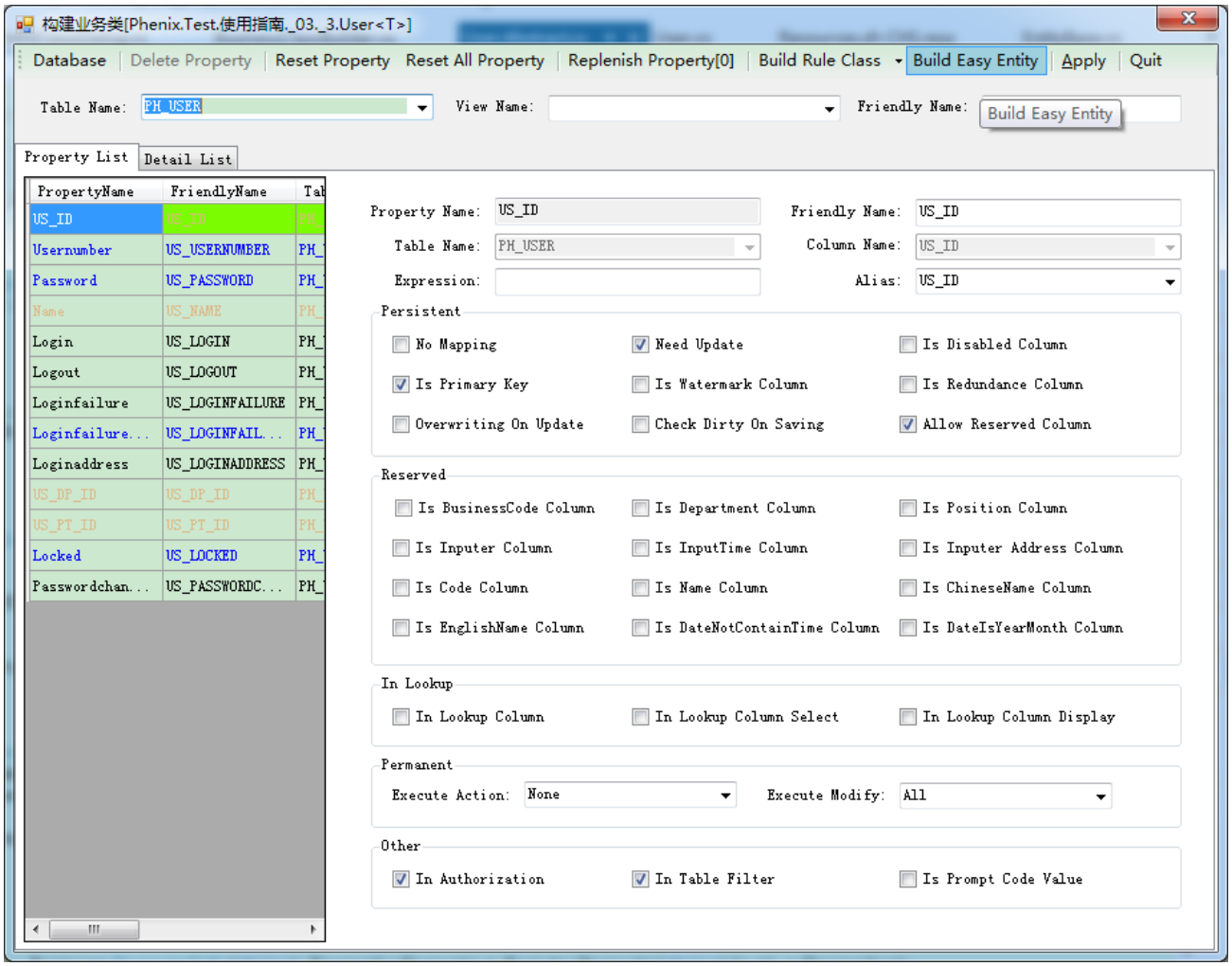


上图所示，就是 Phenix丶在 Phenix.Core.Data 命名空间中提供的轻量级 Entity 基类。

其特点是：

- 可重载的友好名、主键属性；
- 具备与业务类一样配套的查询条件和方法，比如 ICriteria、CriteriaExpression、Expression<Func<TBusiness, bool>>等；
- 默认下仅能够读取数据，但也可以自行调节为处理增删改的提交更新，方法是在属性的 setter 里调用 SetProperty()函数；
- 含有 BeginEdit()、CancelEdit()、ApplyEdit()函数，提供简单的一级回滚机制，不会实现 CSLA 的 N 级回滚机制；
- 含有 NewPure()、New()静态方法；

如果要自动生成轻量级 Entity 类代码的话，可通过“编辑业务类”界面上的“Build Easy Entity”功能按钮：



自动生成的代码摘录如下：

```
/// <summary>
/// 用户
/// </summary>
[Serializable]
[System.ComponentModel.DisplayName("用户")]
[Phenix.Core.Mapping.Class("PH_USER", FriendlyName = "用户")]
public class UserEasy : EntityBase<UserEasy>
{
  private UserEasy()
  {
    //禁止添加代码
  }

  /// <summary>
  /// 构建实体
  /// </summary>
  protected override object CreateInstance()
```

```
    {
      return new UserEasy();
    }

    /// <summary>
    /// 标签
    /// 缺省为唯一键值
    /// 用于提示信息等
    /// </summary>
    [System.ComponentModel.Browsable(false)]
    [System.ComponentModel.DataAnnotations.Display(AutoGenerateField = false)]
    public override string Caption
    {
      get { return base.Caption; }
    }

    /// <summary>
    /// 主键值
    /// </summary>
    [System.ComponentModel.Browsable(false)]
    [System.ComponentModel.DataAnnotations.Display(AutoGenerateField = false)]
    public override string PrimaryKey
    {
      get { return US_ID.ToString(); }
    }

    /// <summary>
    /// US_ID
    /// </summary>
    public static readonly Phenix.Core.Mapping.PropertyInfo<long?> US_IDProperty =
RegisterProperty<long?>(c => c.US_ID, "US_ID");
    /// <summary>
    /// US_ID
    /// </summary>
    [Phenix.Core.Mapping.Field(FriendlyName = "US_ID", TableName = "PH_USER", ColumnName = "US_ID",
IsPrimaryKey = true, NeedUpdate = true)]
    private long? _US_ID;
    /// <summary>
    /// US_ID
    /// </summary>
    [System.ComponentModel.DisplayName("US_ID")]
    public long? US_ID
    {
      get { return _US_ID; }
    }
```

```
    /// <summary>
    /// US_USERNUMBER
    /// </summary>
    public static readonly Phenix.Core.Mapping.PropertyInfo<string> UsernumberProperty =
RegisterProperty<string>(c => c.Usernumber, "US_USERNUMBER");
    /// <summary>
    /// US_USERNUMBER
    /// </summary>
    [Phenix.Core.Mapping.Field(FriendlyName = "US_USERNUMBER", Alias = "US_USERNUMBER", TableName =
"PH_USER", ColumnName = "US_USERNUMBER", NeedUpdate = true)]
    private string _usernumber;
    /// <summary>
    /// US_USERNUMBER
    /// </summary>
    [System.ComponentModel.DisplayName("US_USERNUMBER")]
    public string Usernumber
    {
      get { return _usernumber; }
    }



  /// <summary>
  ///用户集合
  /// </summary>
  [Serializable]
  public class UserEasyList : EntityListBase<UserEasyList, UserEasy>
  {
    private UserEasyList()
    {
      //禁止添加代码
    }

    /// <summary>
    /// 构建实体
    /// </summary>
    protected override object CreateInstance()
    {
      return new UserEasyList();
    }

    /// <summary>
    /// 标签
    /// 缺省为 TEntity 上的 ClassAttribute.FriendlyName
    /// 用于提示信息等
    /// </summary>
```

```
    [System.ComponentModel.Browsable(false)]
    [System.ComponentModel.DataAnnotations.Display(AutoGenerateField = false)]
    public override string Caption
    {
      get { return base.Caption; }
    }
  }
```

Fetch 方法举例如下：

```
IList<UserEasy> userEasies = Phenix.Business.Security.UserPrincipal.User.Identity.FetchList<UserEasy>();
```

或：

```
UserEasyList userEasyList = Phenix.Business.Security.UserPrincipal.User.Identity.FetchList<UserEasyList, UserEasy>();
```

复杂的查询，可通过参数传入查询条件。

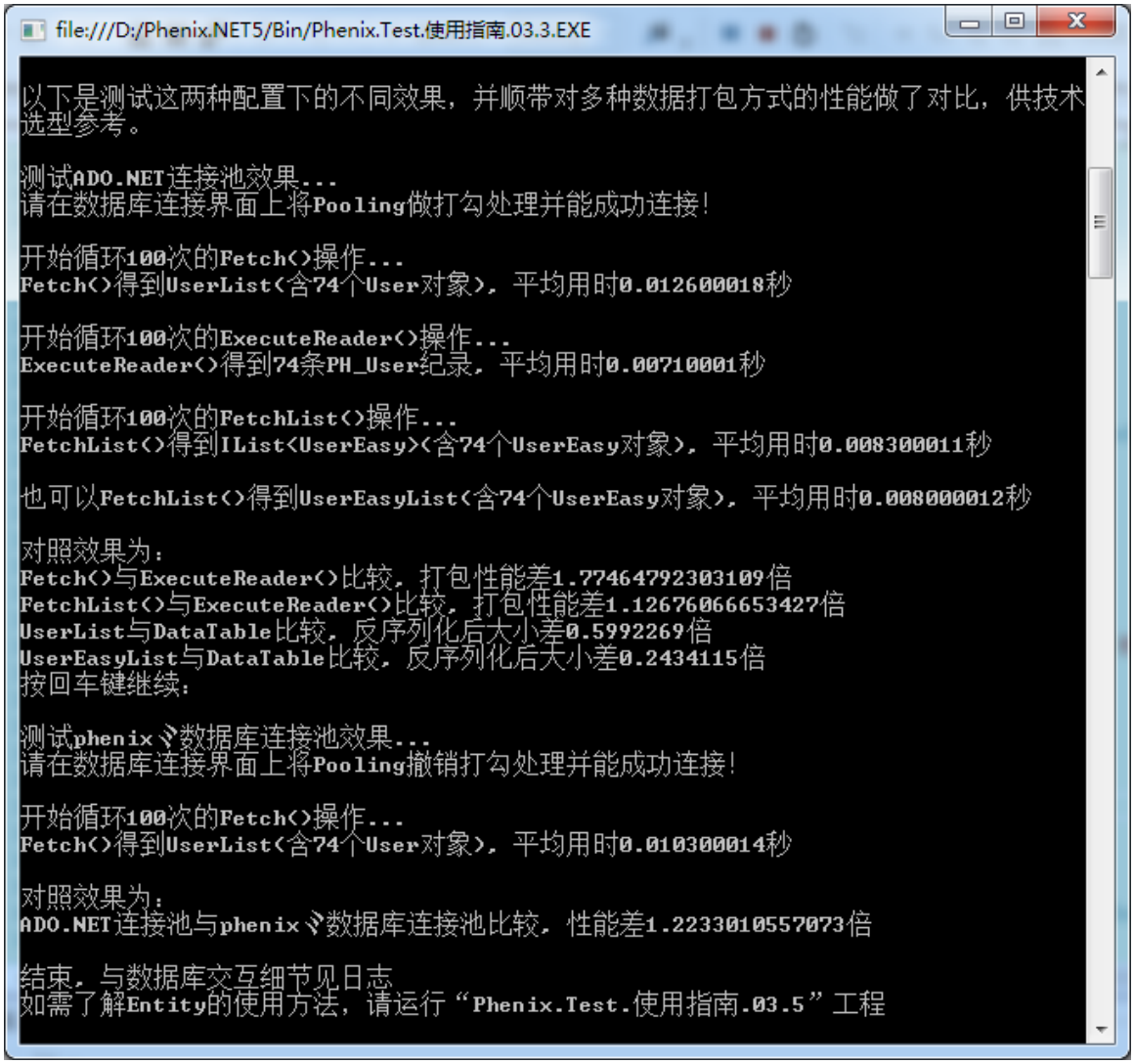
另外，为了编写代码风格的统一，也可以操作 Phenix.Business 程序集里继承下来的类，直接操作它们的 Fetch()功能：


```
UserEasyList userEasyList = UserEasyList.Fetch();
IList<UserEasy> userEasies = UserEasy.FetchList ();
```

Fetch()到本地的 Entity 对象，经编辑后，是可以被提交更新到数据库的。方法是调用 Phenix.Business.Security.UserPrincipal.User 提供的 InsertList()、UpdateList()、DeleteList()系列函数。使用方法，可参见“Phenix.Test.使用指南.03.5”测试工程。

当前几种数据的检索方法的性能对照，可通过“Phenix.Test.使用指南.03.5”测试工程来验证。下

图为测试结果：

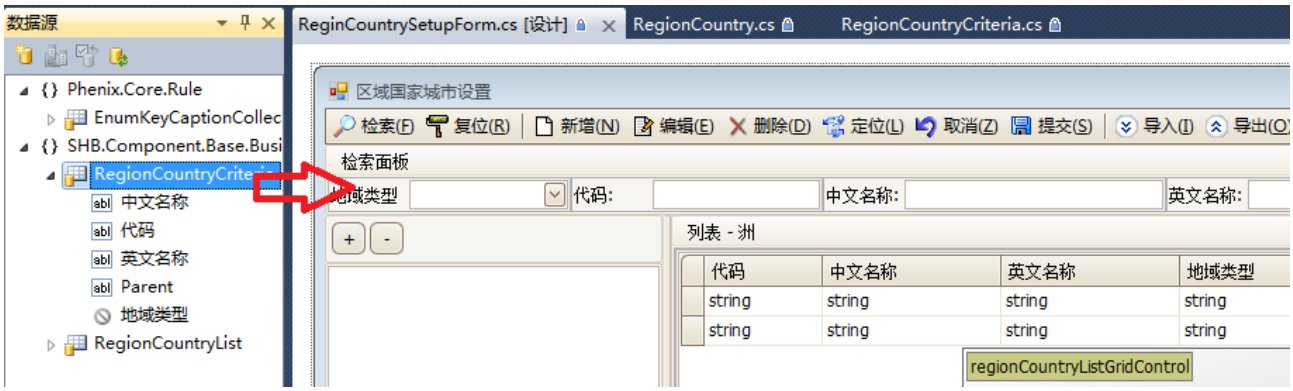


3.6 初始化/编辑查询类

Phenix丶为业务类/集合的检索提供了多种接口方式，查询类是其中之一。

使用查询类，主要是为了UI界面的设计开发，可以实现和业务类一致的数据绑定设计方法，减少界面控制层的代码量，达到快速设计开发的目的。

以下图片显示的是，在检索面板内的标签、输入框，它们的数据绑定都可以通过将对应的数据源拖放及其引导界面一步步操作来实现：



上述案例中 RegionCountryList 数据的检索，即可以通过 Phenix.Windows.BarManager 自动完成，也可以通过编写代码手工来实现，例如：

```
RegionCountryList regionCountryList = RegionCountryList.Fetch(new RegionCountryCriteria()
{ RegionType = RegionType.City });
```

查询类的编码方式和业务类的编码方式区别不大：

```
/// <summary>
/// 查询运输作业地点
/// </summary>
[System.SerializableAttribute(), System.ComponentModel.DisplayNameAttribute("查询运输作业地点")]
public class TransportationLocationCriteria : CriteriaBase
{
    /// <summary>
    /// 城市
    /// </summary>
    [Phenix.Core.Mapping.CriteriaField(Operate = Phenix.Core.Mapping.CriteriaOperate.Equal,
Logical = Phenix.Core.Mapping.CriteriaLogical.And,
      FriendlyName = "城市", SourceName = "SYS_TRANSPORTATION_LOCATION", Alias = "STL_CITY_SRC_ID",
TableName = "SYS_TRANSPORTATION_LOCATION", ColumnName = "STL_CITY_SRC_ID")]
    private long? _citySrcId;
    /// <summary>
    /// 城市
    /// </summary>
    [System.ComponentModel.DisplayName("城市")]
    public long? CitySrcId
    {
        get { return _citySrcId; }
        set { _citySrcId = value; }
    }
```

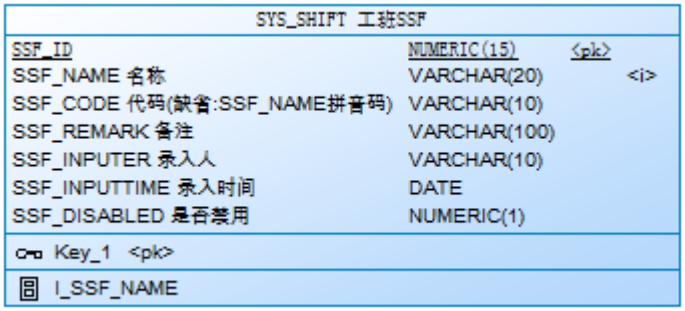
```
        /// <summary>
        /// 中文名称
        /// </summary>
        [Phenix.Core.Mapping.CriteriaField(Operate =
Phenix.Core.Mapping.CriteriaOperate.LikeIgnoreCase, Logical = Phenix.Core.Mapping.CriteriaLogical.And,
          FriendlyName = "中文名称", SourceName = "SYS_TRANSPORTATION_LOCATION", Alias = "STL_CHNNAME",
TableName = "SYS_TRANSPORTATION_LOCATION", ColumnName = "STL_CHNNAME")]
        private string _chnnameLike;
        /// <summary>
        /// 中文名称
        /// </summary>
        [System.ComponentModel.DisplayName("中文名称")]
        public string ChnnameLike
        {
            get
            {
                return string.IsNullOrEmpty(_chnnameLike) ? string.Empty : _chnnameLike.Substring(1,
_chnnameLike.Length - 2);
            }
            set
            {
                if (string.IsNullOrEmpty(value))
                    _chnnameLike = string.Empty;
                else
                    _chnnameLike = "%" + value + "%";
            }
        }

        /// <summary>
        /// 代码
        /// </summary>
        [Phenix.Core.Mapping.CriteriaField(Operate =
Phenix.Core.Mapping.CriteriaOperate.LikeIgnoreCase, Logical = Phenix.Core.Mapping.CriteriaLogical.And,
          FriendlyName = "代码", SourceName = "SYS_TRANSPORTATION_LOCATION", Alias = "STL_CODE",
TableName = "SYS_TRANSPORTATION_LOCATION", ColumnName = "STL_CODE")]
        private string _codeLike;
        /// <summary>
        /// 代码
        /// </summary>
        [System.ComponentModel.DisplayName("代码")]
        public string CodeLike
        {
            get
            {
                return string.IsNullOrEmpty(_codeLike) ? string.Empty : _codeLike.Substring(1,
_codeLike.Length - 2);
```

```
            }
            set
            {
                if (string.IsNullOrEmpty(value))
                    _codeLike = string.Empty;
                else
                    _codeLike = "%" + value + "%";
            }
        }
    }
```

下面我们将一步步练习如何构建出一个带 Mapping 代码的查询类。

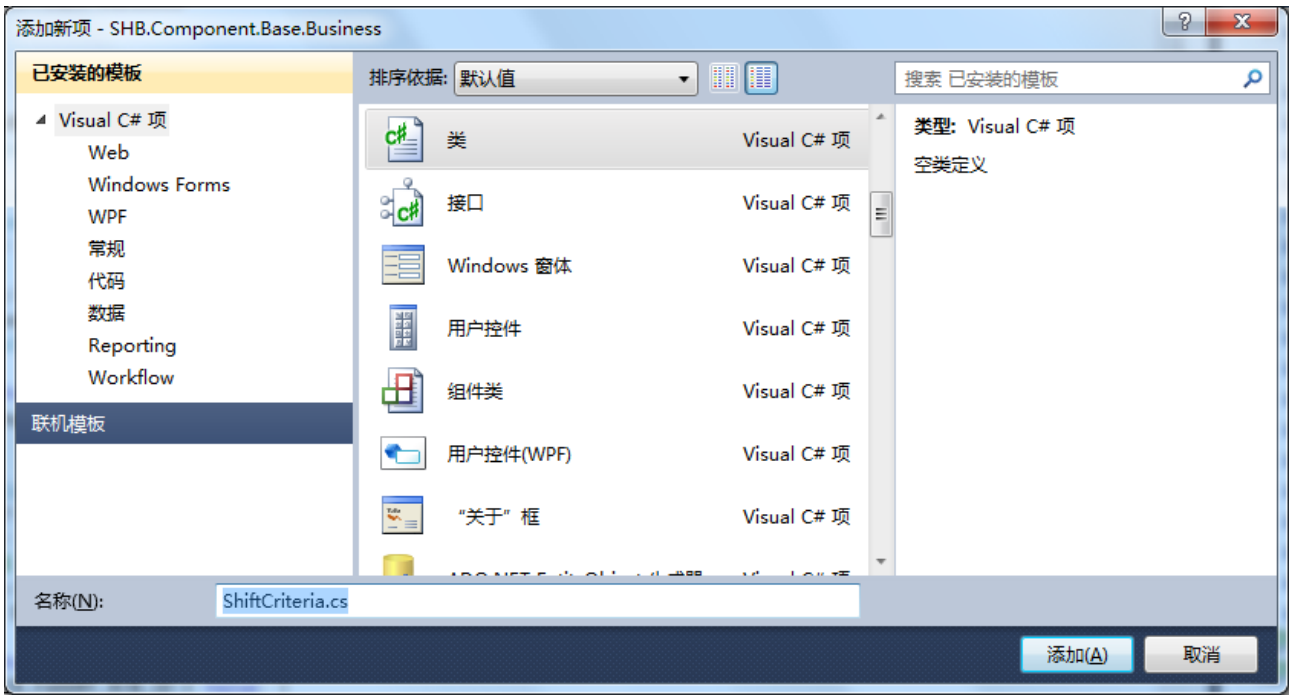
3.6.1 初始化查询类

我们以一个单表为例：



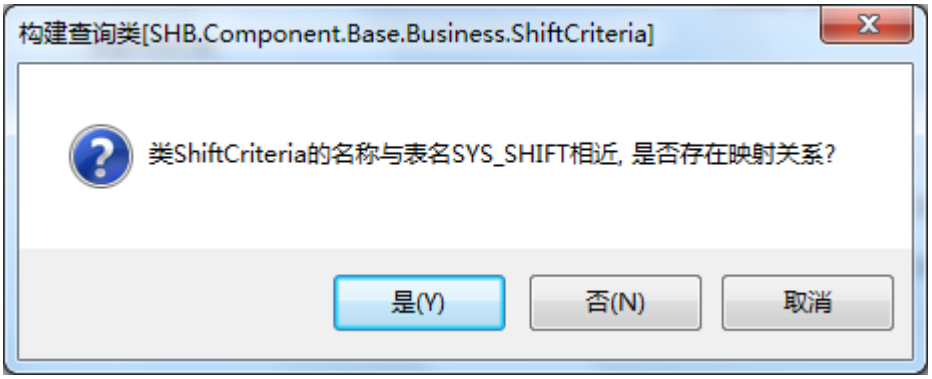
查询类是构建在业务逻辑组件工程中的，一般我们以“.Business”为工程名后缀。

首先，我们在相关的业务逻辑组件工程中建一个空的查询类，一般我们以所查询的主表名（剔除表名的前缀、后缀）+“Criteria”为类名：

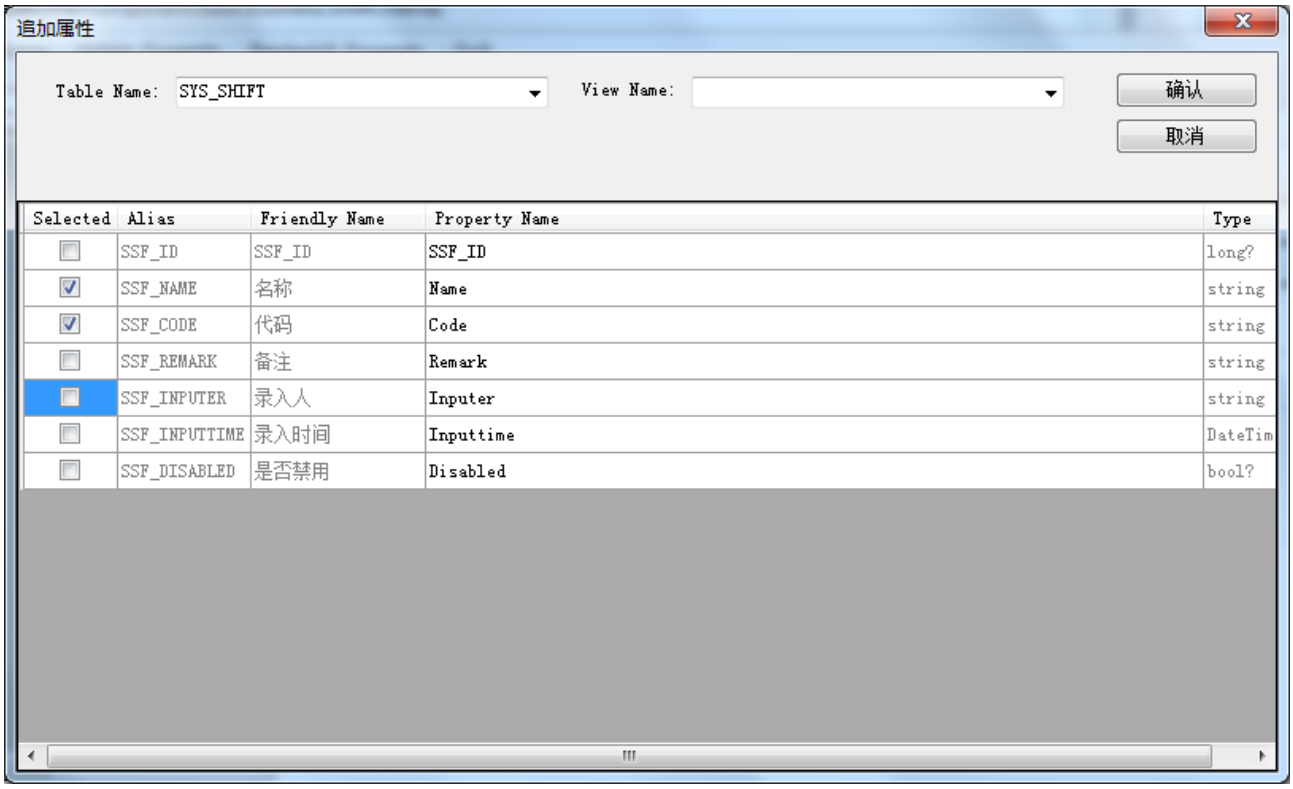


然后，在类编辑界面上用鼠标右键点出浮动菜单，选择：

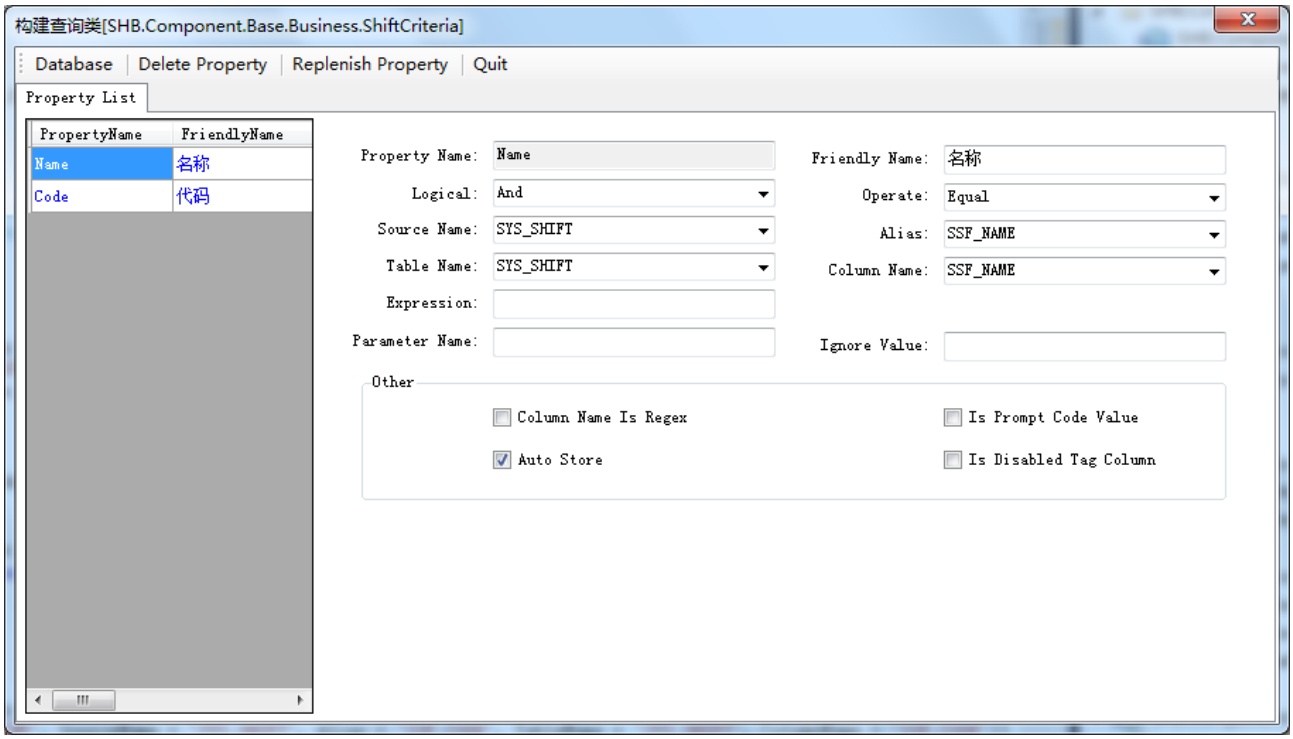
将会弹出提示：



点击“是”，则会根据表“SYS_SHIFT”数据字典信息自动生成查询类代码。

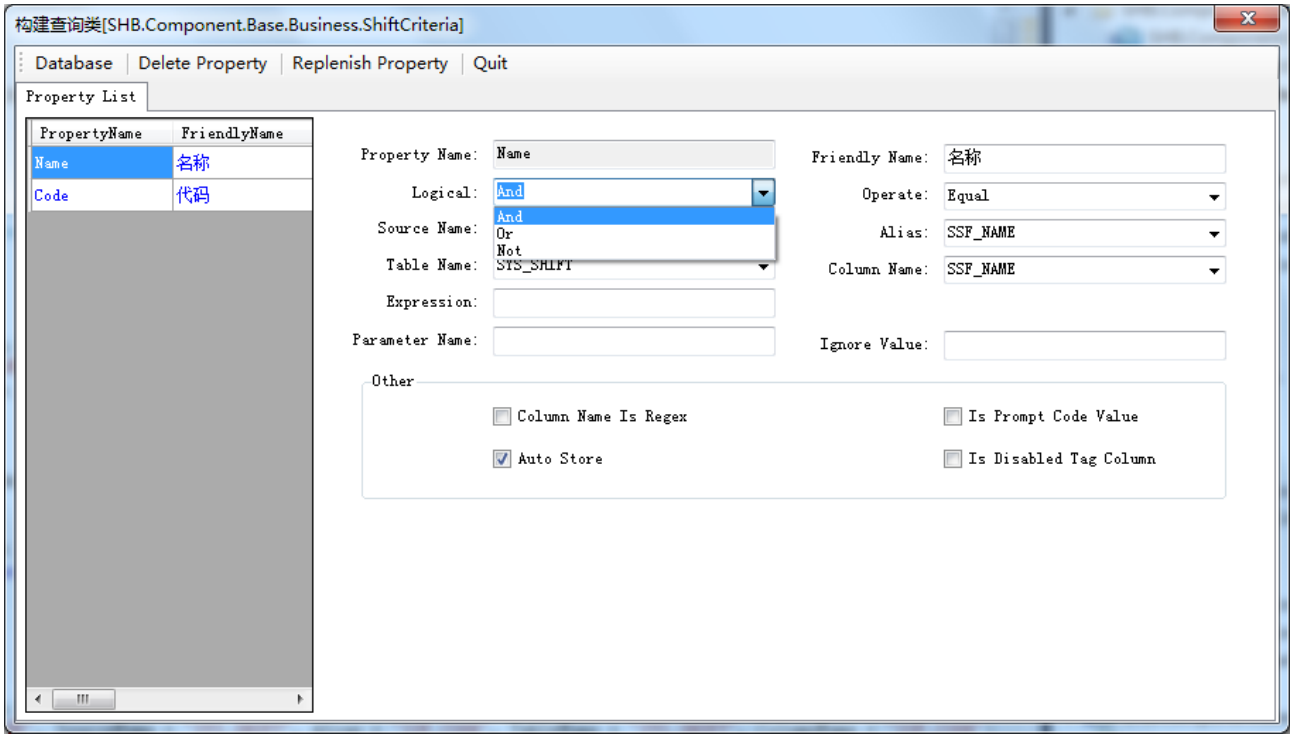


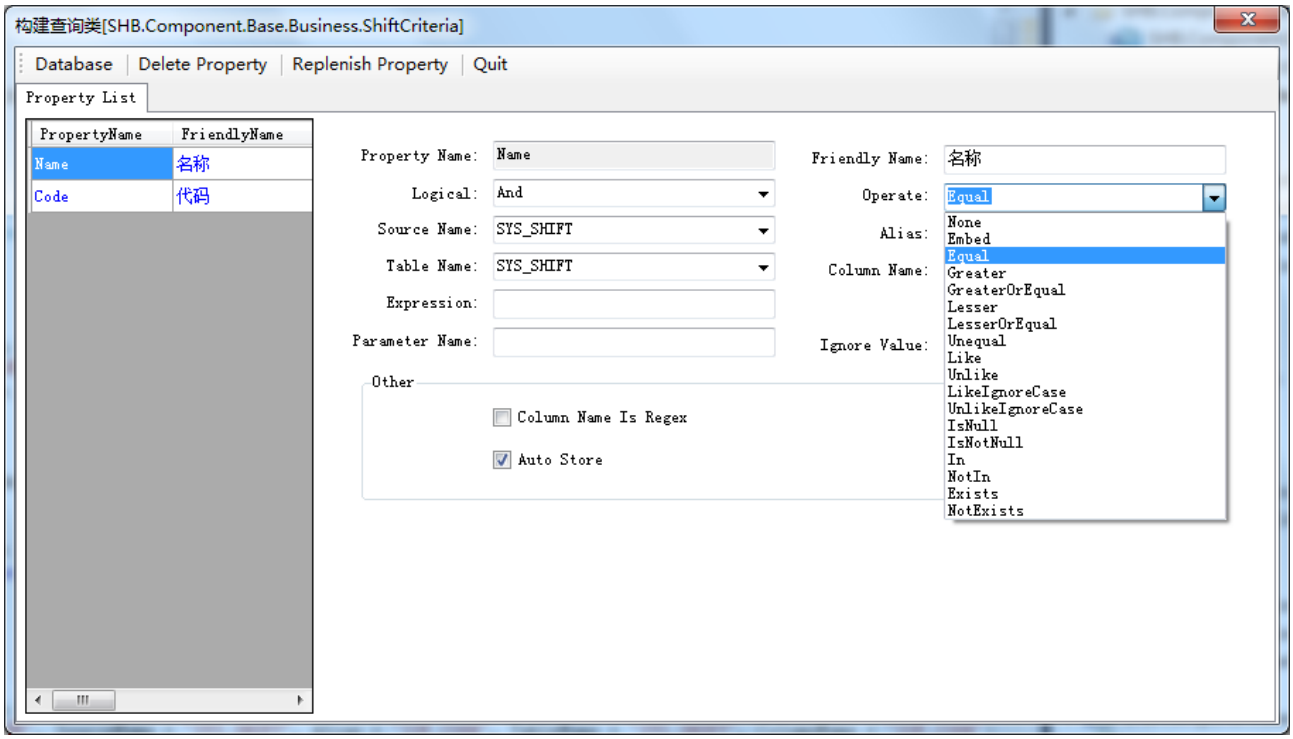
如上图，点击清单“selected”项可全选，只有被选中的字段才会生成对应的数据属性代码：



3.6.2 编辑查询类

通常，你可以直接修改源代码，也可以通过右键点出图形化界面，快速修改 Mapping 细节。





3.7 初始化指令类

CSLA 所提供的指令类，是对面向对象设计的一个生动诠释。

指令类借助CSLA 自身的移动对象机制，将跨物理域等底层技术细节封装于无形之中，为开发者提供了简单、灵活的设计接口。

指令类暴露给调用方（消费者）的接口，仅限于其构造函数及其参数、属性、以及标准的 Execute()

函数。其属性所封装的字段可以为任何类型的对象，只要能序列化就可以跨物理域传递；而其内部业务逻辑代码，或者说执行代码（override DoExecute()函数，甚至是直接 override Csla.CommandBase<T>的 DataPortal_Execute()），则因为运行在服务端，而可以与数据库或者其他服务端的资源进行交互，甚至执行一个复杂的服务端工作流，等等。

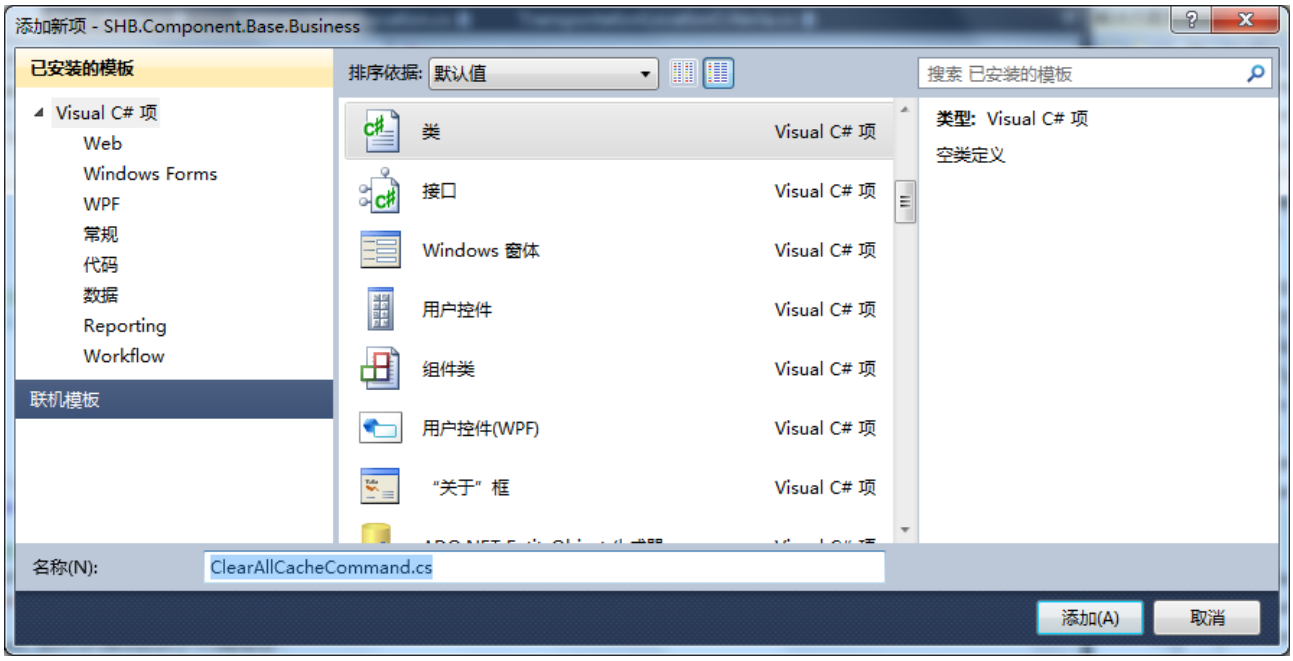
消费者代码案例：

```
new ClearAllCacheCommand().Execute();
```

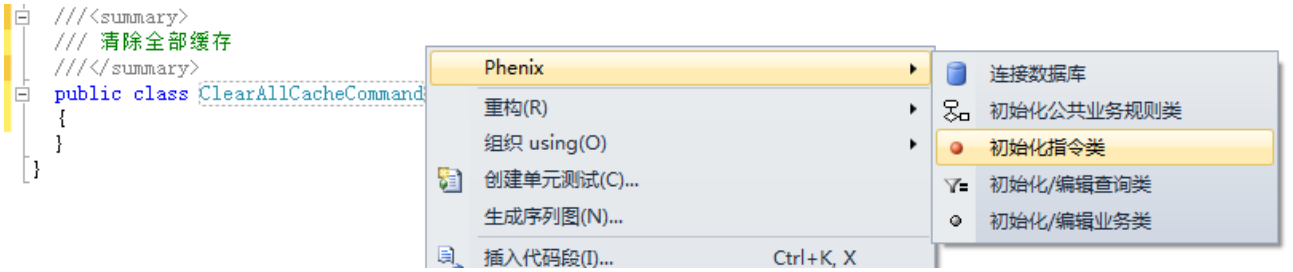
下面我们一步步来构建指令类：

指令类是构建在业务逻辑组件工程中的，一般我们以“.Business”为工程名后缀。

首先，我们在相关的业务逻辑组件工程中建一个空的指令类，一般我们以这个指令类的具体含义+“Command”为类名：



然后，在类编辑界面上用鼠标右键点出浮动菜单，选择：



自动生成指令类框架代码：

```
/// <summary>
/// 清除全部缓存
/// </summary>
public class ClearAllCacheCommand : Phenix.Business.CommandBase<ClearAllCacheCommand>
{
  /// <summary>
  /// 清除全部缓存
  /// </summary>
  public ClearAllCacheCommand()
  {
  }

  /// <summary>
  /// 处理执行指令(在运行持久层的程序域里被调用)
  /// 参数可替换为 DbConnection connection 用于执行非事务过程
  /// </summary>
  protected override void DoExecute(System.Data.Common.DbTransaction transaction)
  {

  }
}
```

这样，我们可以在 DoExecute()函数中写上业务逻辑/持久化代码(这里演示了如何通过 Phenix 持久层引擎提供的底层接口直接操作数据库）：

```
  protected override void DoExecute(System.Data.Common.DbTransaction transaction)
  {
    //使用框架提供的数据库连接与事务控制
    using (DbCommand command = DbCommandHelper.CreateSqlCommand(transaction,
      "update PH_CacheAction set CA_ActionTime = sysdate"))
    {
      DbCommandHelper.ExecuteNonQuery(command);
    }
  }
```

3.8 初始化公共业务规则类

“对象设计不是关于标准化数据的，它是关于标准化行为的。”（摘自《Expert C# 2008 BusinessObjects》）。也就是说，对象设计的目标是保证给定的行为（业务规则或业务逻辑）在对象模型中只存在一次，我们应该以公共业务规则类的形式将业务规则编写为标准化、规范化的代码供不同的业务对象使用，业务对象仅需将这些公共业务规则类注册到自身的业务规则库 BusinessRules 中即可拥

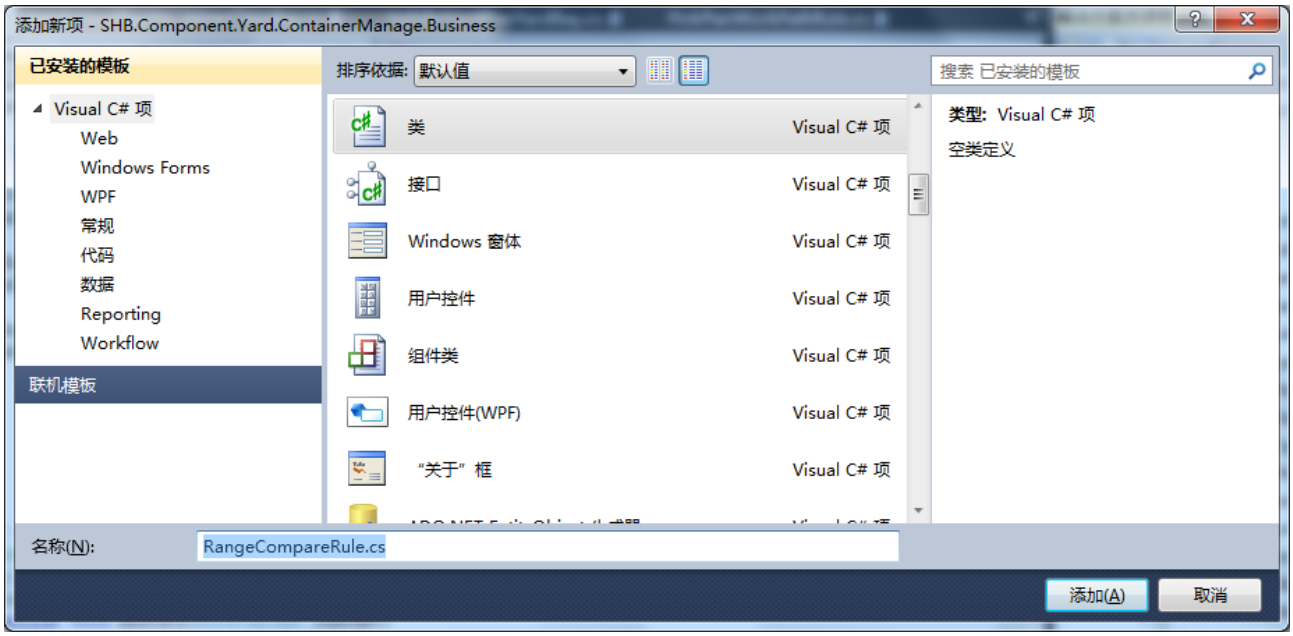
有它们相应的业务规则，剩下的工作就交给 Phenix丶平台自动去处理：

```
/// <summary>
/// 注册业务规则
/// </summary>
protected override void AddBusinessRules()
{
  BusinessRules.AddRule(new RangeCompareRule(minValueProperty, maxValueProperty);
  base.AddBusinessRules();
}
```

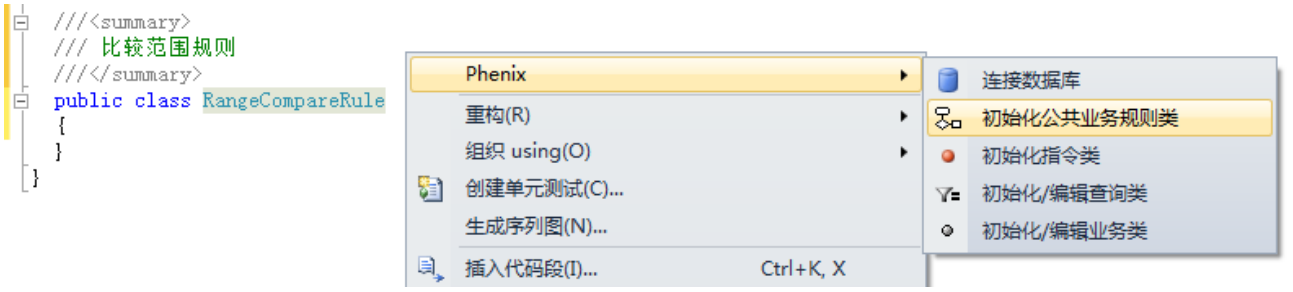
下面我们以 Phenix丶提供的 Phenix.Business.Rules.RangeCompareRule<T>（比较范围规则）作为案例，来演示如何构建公共业务规则类：

公共业务规则类是构建在业务规则组件工程中的，一般我们以“.Rule”为工程名后缀。

首先，我们在相关的业务规则组件工程中建一个空的公共业务规则类，一般我们以这个公共业务规则类的具体含义+“Rule”为类名：



然后，在类编辑界面上用鼠标右键点出浮动菜单，选择：



自动生成公共业务规则类框架代码：

```
  ///<summary>
  /// 比较范围规则
  ///</summary>
  public class RangeCompareRule : Phenix.Business.Rules.CommonBusinessRule
  {
    /// <summary>
    /// 比较范围规则
    /// </summary>
    public RangeCompareRule()
    {
    }

    /// <summary>
    /// 执行
    /// </summary>
    protected override void Execute(Csla.Rules.RuleContext context)
    {

    }
```

这样，我们可以在 Execute()函数中写上业务逻辑代码：

```
  ///<summary>
  /// 比较范围规则
  ///</summary>
  public class RangeCompareRule<T> : Phenix.Business.Rules.CommonBusinessRule
    where T : IComparable
  {
    ///<summary>
    /// 比较范围规则
    ///</summary>
    ///<param name="primaryProperty">主属性信息</param>
    ///<param name="minValuePropertyInfo">最小值属性信息</param>
    ///<param name="maxValuePropertyInfo">最大值属性信息</param>
    public RangeCompareRule(IPropertyInfo primaryProperty, IPropertyInfo minValuePropertyInfo,
IPropertyInfo maxValuePropertyInfo)
      : base(primaryProperty)
    {
      if (primaryProperty != minValuePropertyInfo)
        AffectedProperties.Add(minValuePropertyInfo);
```

```
      if (primaryProperty != maxValuePropertyInfo)
        AffectedProperties.Add(maxValuePropertyInfo);

      InputProperties = new List<Csla.Core.IPropertyInfo> { minValuePropertyInfo,
maxValuePropertyInfo };

      _minValuePropertyInfo = minValuePropertyInfo;
      _maxValuePropertyInfo = maxValuePropertyInfo;
    }

    #region 属性

    private readonly IPropertyInfo _minValuePropertyInfo;
    /// <summary>
    /// 最小值属性信息
    /// </summary>
    public IPropertyInfo MinValuePropertyInfo
    {
      get { return _minValuePropertyInfo; }
    }

    private readonly IPropertyInfo _maxValuePropertyInfo;
    /// <summary>
    /// 最大值属性信息
    /// </summary>
    public IPropertyInfo MaxValuePropertyInfo
    {
      get { return _maxValuePropertyInfo; }
    }

    #endregion

    #region 方法

    ///<summary>
    /// 执行
    ///</summary>
    protected override void Execute(Csla.Rules.RuleContext context)
    {
      IDataErrorInfo errorInfo = context.Target as IDataErrorInfo;
      if (errorInfo != null)
        if (object.ReferenceEquals(PrimaryProperty, MinValuePropertyInfo))
        {
          if (!String.IsNullOrEmpty(errorInfo[MaxValuePropertyInfo.Name]))
            return;
        }
```

```
        else if (object.ReferenceEquals(PrimaryProperty, MaxValuePropertyInfo))
        {
          if (!String.IsNullOrEmpty(errorInfo[MinValuePropertyInfo.Name]))
            return;
        }

      object minValueObject = context.InputPropertyValues[MinValuePropertyInfo];
      object maxValueObject = context.InputPropertyValues[MaxValuePropertyInfo];
      if (object.Equals(minValueObject, null) || object.Equals(maxValueObject, null))
        return;
      T minValue = (T)Phenix.Core.Reflection.Utilities.ChangeType(minValueObject, typeof(T));
      T maxValue = (T)Phenix.Core.Reflection.Utilities.ChangeType(maxValueObject, typeof(T));
      if (object.Equals(minValue, null) || object.Equals(maxValue, null))
        return;
      int result = minValue.CompareTo(maxValue);
      if (result > 0)
      {
        string message = string.Format(Phenix.Business.Properties.Resources.RangeCompareRule,
          MinValuePropertyInfo.FriendlyName, minValue, MaxValuePropertyInfo.FriendlyName, maxValue);
        context.Results.Add(new Csla.Rules.RuleResult(RuleName, PrimaryProperty, message) { Severity =
Severity });
      }
    }

    #endregion
  }
```

上述案例所述功能已在 Phenix 的 Phenix.Business.Rules.RangeCompareRule<T>中实现，在具体的应用系统开发当中仅需拿来注册使用即可。

3.9 构建对象级别/属性有效性规则类

业务对象需要保证被写入的数据是正确的、完整的、有效的，才能进一步对这些数据进行处理：

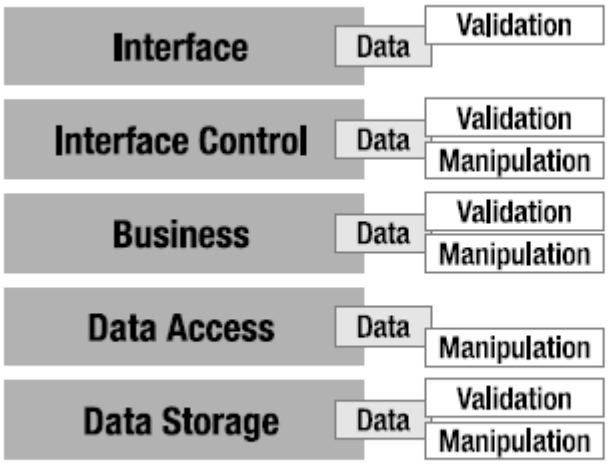


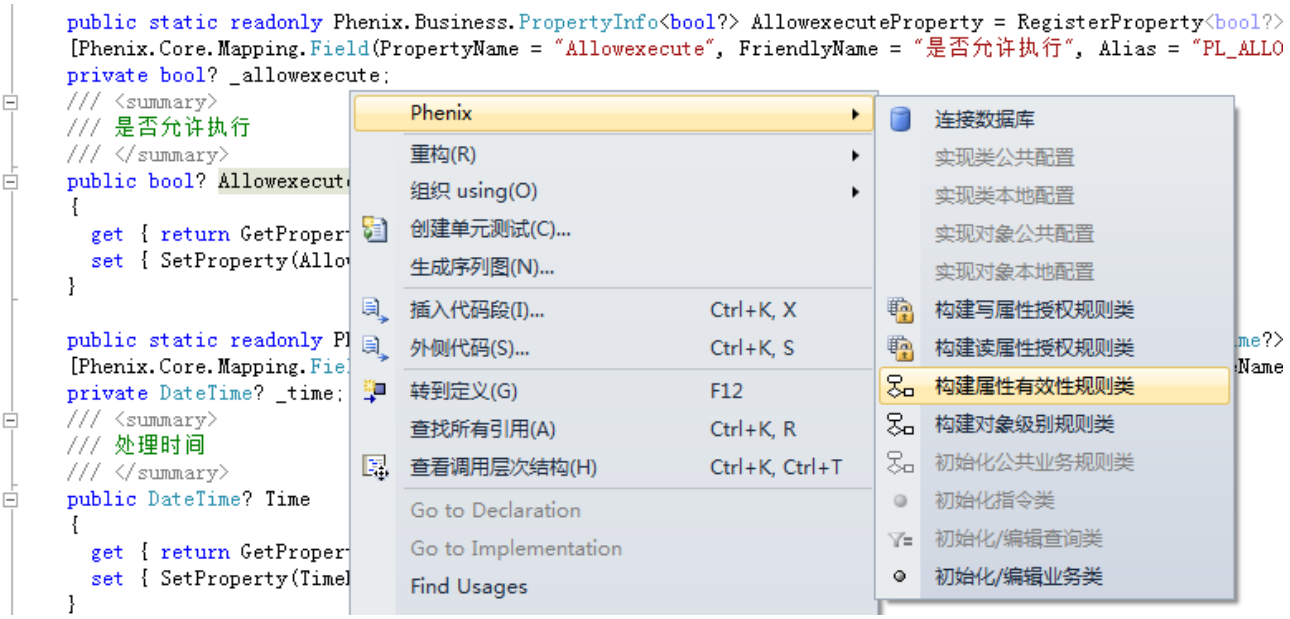
Figure 1-9. *Common locations for business logic in applications*

上图摘自 CSLA 作者的《Expert C# 2008 Business Objects》。

这些对写入数据进行有效性验证的逻辑代码，需要统一编写成有效性规则类，注册到业务对象的业务规则库 BusinessRules 里，它就拥有了相应的数据有效性的验证功能：

```
/// <summary>
/// 注册业务规则
/// </summary>
protected override void AddBusinessRules()
{
  BusinessRules.AddRule(new ProcessLockAllowexecuteEditValidationRule());
  base.AddBusinessRules();

}
```

我们以 ProcessLock 业务对象的 Allowexecute 属性添加“不允许为空”有效性验证功能为例，演示如何构建有效性规则类，这只要在该属性上点击右键点出浮动菜单，选择：



自动生成有效性规则类框架代码：

```
/// <summary>
/// "是否允许执行"有效性规则
/// </summary>
public class ProcessLockAllowexecuteEditValidationRule : Phenix.Business.Rules.EditValidationRule
{
  /// <summary>
  /// 初始化
  /// </summary>
  public ProcessLockAllowexecuteEditValidationRule()
    : base(ProcessLock.AllowexecuteProperty) { }

  #region Register

  /// <summary>
  /// 注册业务规则
  /// </summary>
  public static void Register()
  {
    ProcessLock.BusinessRuleRegistering += new
Phenix.Business.Core.BusinessRuleRegisteringEventHandler(ProcessLock_BusinessRuleRegistering);
  }

  private static void ProcessLock_BusinessRuleRegistering(Csla.Rules.BusinessRules businessRules)
  {
    businessRules.AddRule(new ProcessLockAllowexecuteEditValidationRule());
  }
```

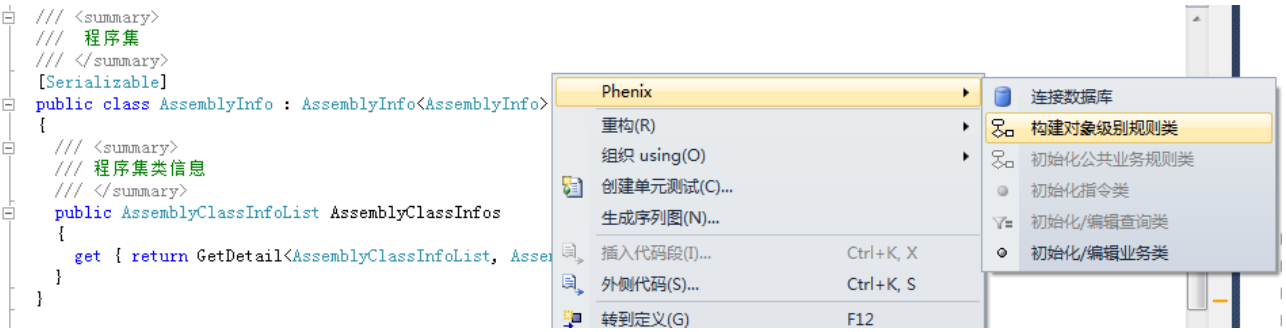
```
    #endregion

    /// <summary>
    /// 执行
    /// </summary>
    protected override void Execute(Csla.Rules.RuleContext context)
    {
    }
  }
```

这样，我们可以在 Execute()函数中写上相应的有效性验证代码即可：

```
    protected override void Execute(Csla.Rules.RuleContext context)
    {
      var value = this.ReadProperty(context.Target, PrimaryProperty);
      if (value == null || string.IsNullOrWhiteSpace(value.ToString()))
      {
        var message = string.Format("{0}不允许为空！", PrimaryProperty.FriendlyName);
        context.Results.Add(new RuleResult(RuleName, PrimaryProperty, message) { Severity =
RuleSeverity.Error });
      }
    }
```

上述案例仅仅是为了演示，Phenix 已结合了数据字典的有效性验证信息、微软 Entity Framwork 的 System.ComponentModel.DataAnnotations，将这些“不允许为空”等普遍性的有效性校验功能做了集成管理。在具体的应用系统开发当中，如果有超出数据字典有效性验证之外的验证需求，只要给数据属性打上 ValidationAttribute 标签即可。

对于对象级别的数据有效性规则，比如在提交的时候要检验其子对象集合不允许为空（子表记录不允许为空），那么我们必须构建一个对象级别的规则类，在该业务类上点击右键点出浮动菜单，选择：



自动生成有效性规则类框架代码：

```
  /// <summary>
```

```
  /// "程序集"业务规则
  /// </summary>
  public class AssemblyInfoObjectRule : Phenix.Business.Rules.ObjectRule
  {
    /// <summary>
    /// 初始化
    /// </summary>
    public AssemblyInfoObjectRule()
      : base() { }

    #region Register

    /// <summary>
    /// 注册业务规则
    /// </summary>
    public static void Register()
    {
      AssemblyInfo.BusinessRuleRegistering += new
Phenix.Business.Core.BusinessRuleRegisteringEventHandler(AssemblyInfo_BusinessRuleRegistering);
    }

    private static void AssemblyInfo_BusinessRuleRegistering(Csla.Rules.BusinessRules businessRules)
    {
      businessRules.AddRule(new AssemblyInfoObjectRule());
    }

    #endregion

    /// <summary>
    /// 执行
    /// </summary>
    protected override void Execute(Csla.Rules.RuleContext context)
    {

    }
  }
```

这样，我们可以在 Execute()函数中写上相应的有效性验证代码即可：

```
    /// <summary>
    /// 执行
    /// </summary>
    protected override void Execute(Csla.Rules.RuleContext context)
    {
      AssemblyInfo target = (AssemblyInfo)context.Target;
      if (target.AssemblyClassInfos.Count == 0)
```

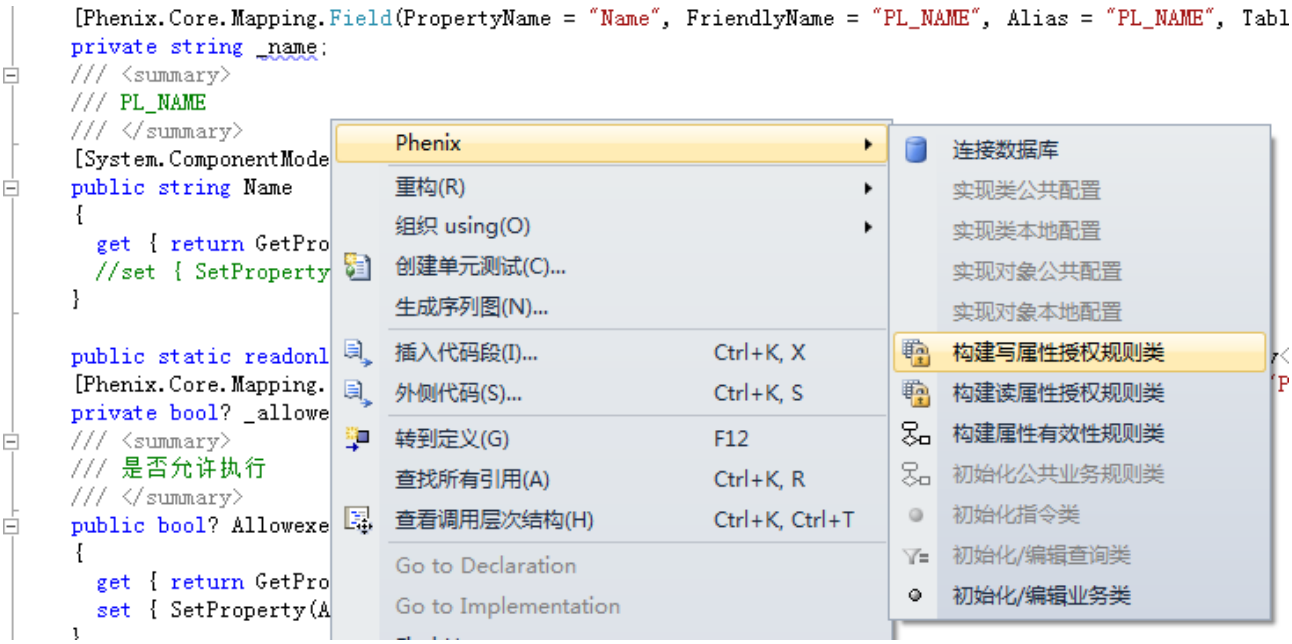
```
      {
        var message = string.Format("{0}的{1}不允许为空！", target.Caption,
AssemblyClassInfoList.FriendlyName);
        context.Results.Add(new RuleResult(RuleName, null, message) { Severity = RuleSeverity.Error });
      }
    }
```

3.10 构建读/写属性授权规则类

业务对象在（通过公共业务规则类、有效性规则类）验证写入数据之前还要先过授权这个坎。说到数据授权，首先我们想到的是验证当前用户是否有权读取、写入数据，这个功能在 Phenix 平台上是完全可配置化的，无需编写一行代码，不过更为复杂的数据授权其实还包含着业务规则，比如有这样的业务场景：“过程锁记录在新增提交后名称是不允许修改的”，如果我们将这样的验证代码写成有效性规则类的话，那只能在 Name 属性被写入后才会抛出异常或警示，这势必造成交互界面不是很友好，为什么不能在这之前就禁止掉 Name 属性的写入呢？为了防止用户误改名称，只要控制住 Name 输入框的 ReadOnly 属性就能达到我们的目的。为此，我们必须通过写属性授权规则类来实现它，然后将这个规则类注册到过程锁对象的规则库 AuthorizationRules 里：

```
    /// <summary>
    /// 注册授权规则
    /// </summary>
    protected override void AddAuthorizationRules()
    {
      AuthorizationRules.AddRule(new ProcessLockNameWriteAuthorizationRule());
      base.AddAuthorizationRules();
    }
```

这只要在 Name 属性上点击右键点出浮动菜单，选择：



自动生成写属性授权规则类框架代码：

```
/// <summary>
/// 写"名称"授权规则
/// </summary>
public class ProcessLockNameWriteAuthorizationRule : Phenix.Business.Rules.WriteAuthorizationRule
{
  /// <summary>
  /// 初始化
  /// </summary>
  public ProcessLockNameWriteAuthorizationRule()
    : base(ProcessLock.NameProperty) { }

  #region Register

  /// <summary>
  /// 注册授权规则
  /// </summary>
  public static void Register()
  {
    ProcessLock.AuthorizationRuleRegistering += new
Phenix.Business.Core.AuthorizationRuleRegisteringEventHandler(ProcessLock_AuthorizationRuleRegisterin
g);
  }

  private static void
ProcessLock_AuthorizationRuleRegistering(Phenix.Business.Rules.AuthorizationRules authorizationRules)
  {
```

```
      authorizationRules.AddRule(new ProcessLockNameWriteAuthorizationRule());
    }

    #endregion

    /// <summary>
    /// 执行
    /// </summary>
    protected override void Execute(Phenix.Business.Rules.AuthorizationContext context)
    {
      context.HasPermission = true;
    }
  }
}
```

这样，我们可以在 Execute()函数中写上业务逻辑代码：

```
    protected override void Execute(Phenix.Business.Rules.AuthorizationContext context)
    {
      context.HasPermission = ((IBusiness)context.Target).IsNew;
    }
```

上述案例仅仅是为了演示，Phenix 对于“XXX 记录在新增提交后 N 属性是不允许修改的”这种业务规则，只要在这个属性上打上标签即可（见下例黄底代码），无需编写规则类：

```
    /// <summary>
    /// PL_NAME
    /// </summary>
    public static readonly Phenix.Business.PropertyInfo<string> NameProperty =
RegisterProperty<string>(c => c.Name);
    [Phenix.Core.Mapping.Field(PropertyName = "Name", FriendlyName = "PL_NAME", Alias = "PL_NAME",
TableName = "PH_PROCESSLOCK", ColumnName = "PL_NAME", NeedUpdate = true, IsWatermarkColumn = true)]
    private string _name;
    /// <summary>
    /// PL_NAME
    /// </summary>
    [System.ComponentModel.DisplayName("PL_NAME")]
    public string Name
    {
      get { return GetProperty(NameProperty, _name); }
      set { SetProperty(NameProperty, ref _name, value); }
    }
```

3.11 构建执行过程授权规则类

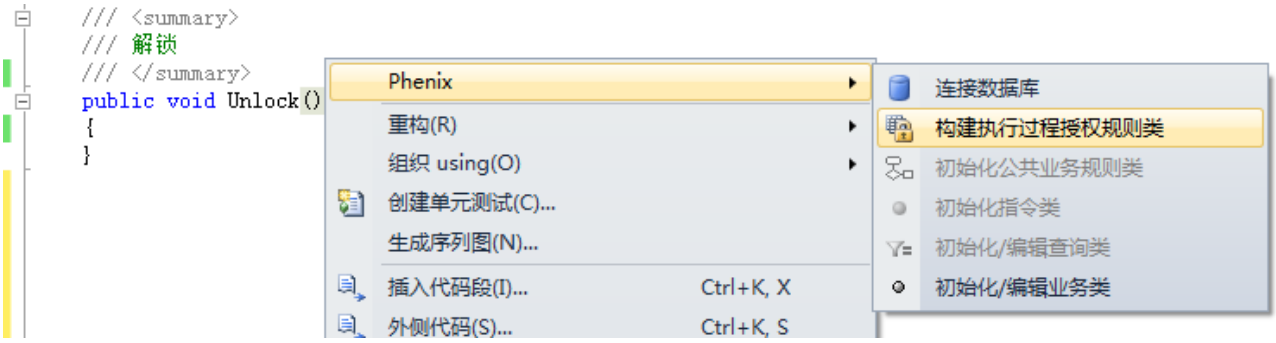
业务对象公共调用接口函数是可以受到权限控制的，这只要为这个函数打上标签并调用CanExecuteMethod()函数（见下例黄底代码）即可：

```
/// <summary>
/// 解锁
/// </summary>
public static readonly Phenix.Business.MethodInfo UnlockMethod = RegisterMethod(c => c.Unlock());
/// <summary>
/// 解锁
/// </summary>
[Phenix.Core.Mapping.Method(FriendlyName = "解锁")]
public void Unlock()
{
  CanExecuteMethod(UnlockMethod, true);
  //以下代码为具体的业务逻辑
}
```

与属性的授权相类似，函数的授权也不仅仅是权限控制这么简单，更为复杂的执行授权其实还包含着业务规则。比如有这样的业务场景：“过程锁只能由加锁者自行解锁”，那么，我们必须通过执行过程授权规则类来实现它，然后将这个规则类注册到过程锁对象的规则库 AuthorizationRules 里：

```
/// <summary>
/// 注册授权规则
/// </summary>
protected override void AddAuthorizationRules()
{
  AuthorizationRules.AddRule(new ProcessLockUnlockExecuteAuthorizationRule());
  base.AddAuthorizationRules();
}
```

执行过程授权规则类的构建，是在 Unlock()函数上点击右键点出浮动菜单，选择：



自动生成执行过程授权规则类框架代码：

```
  /// <summary>
  /// 执行解锁授权规则
  /// </summary>
  public class ProcessLockUnlockExecuteAuthorizationRule :
Phenix.Business.Rules.ExecuteAuthorizationRule
  {
    /// <summary>
    /// 初始化
    /// </summary>
    public ProcessLockUnlockExecuteAuthorizationRule()
      : base(ProcessLock.UnlockMethod) { }

    #region Register

    /// <summary>
    /// 注册授权规则
    /// </summary>
    public static void Register()
    {
       ProcessLock.AuthorizationRuleRegistering += new
Phenix.Business.Core.AuthorizationRuleRegisteringEventHandler(ProcessLock_AuthorizationRuleRegisterin
g);
    }

    private static void
ProcessLock_AuthorizationRuleRegistering(Phenix.Business.Rules.AuthorizationRules authorizationRules)
    {
      authorizationRules.AddRule(new ProcessLockUnlockExecuteAuthorizationRule());
    }

    #endregion

    /// <summary>
    /// 执行
    /// </summary>
    protected override void Execute(Phenix.Business.Rules.AuthorizationContext context)
    {
      context.HasPermission = true;
    }
  }
```

这样，我们可以在 Execute()函数中写上业务逻辑代码：

```
protected override void Execute(Phenix.Business.Rules.AuthorizationContext context)
{

  context.HasPermission =
    Phenix.Business.Security.UserPrincipal.User.Identity.UserNumber ==
((ProcessLock)context.Target).Usernumber;
}
```

3.12 添加枚举标签

3.12.1 为什么用枚举

在设计应用系统时，有两种场景需要用到枚举：

1， 业务对象的状态：状态是对象（通过其属性和过程）执行某项活动或等待某个事件时的条件，是 OOA 动态模型中最关键的输出物之一，被深深地纠缠在业务逻辑之中；

2， 业务对象的类型：是对业务对象的归类，往往在 new 业务对象的时候被设定，如果必须被固化在业务逻辑中的话，则需要用到枚举（产品化软件因倾向于可配置化，而更多地采用代码库+配置库的形式，但这样会使得设计复杂度急剧放大，所以为了将设计复杂度限制在可控范围内，不能迷信全面、绝对的可配置化）；

相对字符串等形式，使用枚举的好处是：

- 编码表达直观：非枚举方式一般使用缩写字符串，编码者需要培训后才能接手工作，开发、维护成本高；
- 值比较和存储的效率高：非枚举方式因为使用了字符串，虽然是缩写，但相对来说效率较低；
- 不容易出错：非枚举方式一般使用缩写字符串，存在错漏、大小写等人为失误问题；
- 可利用编译器在设计期检查出 bug：非枚举方式类似于脚本，只能在系统运行当中发现到不正常（非普通的系统报错，往往是软故障）的状况下才会去检查错误，跟踪查错难度较大，测试成本高；
- 一处存储（.DLL）、一处维护（.csprog）、多处引用，便于 IDE 检索：非枚举方式需要在数据库中开代码表维护，一旦记录被破坏，将影响系统运行，系统稳定性和安全性都很差；
- 升级方便，仅需替换 DLL 即可：非枚举方式还要同时升级数据库中的代码表，一旦遗漏将暴露升级风险；

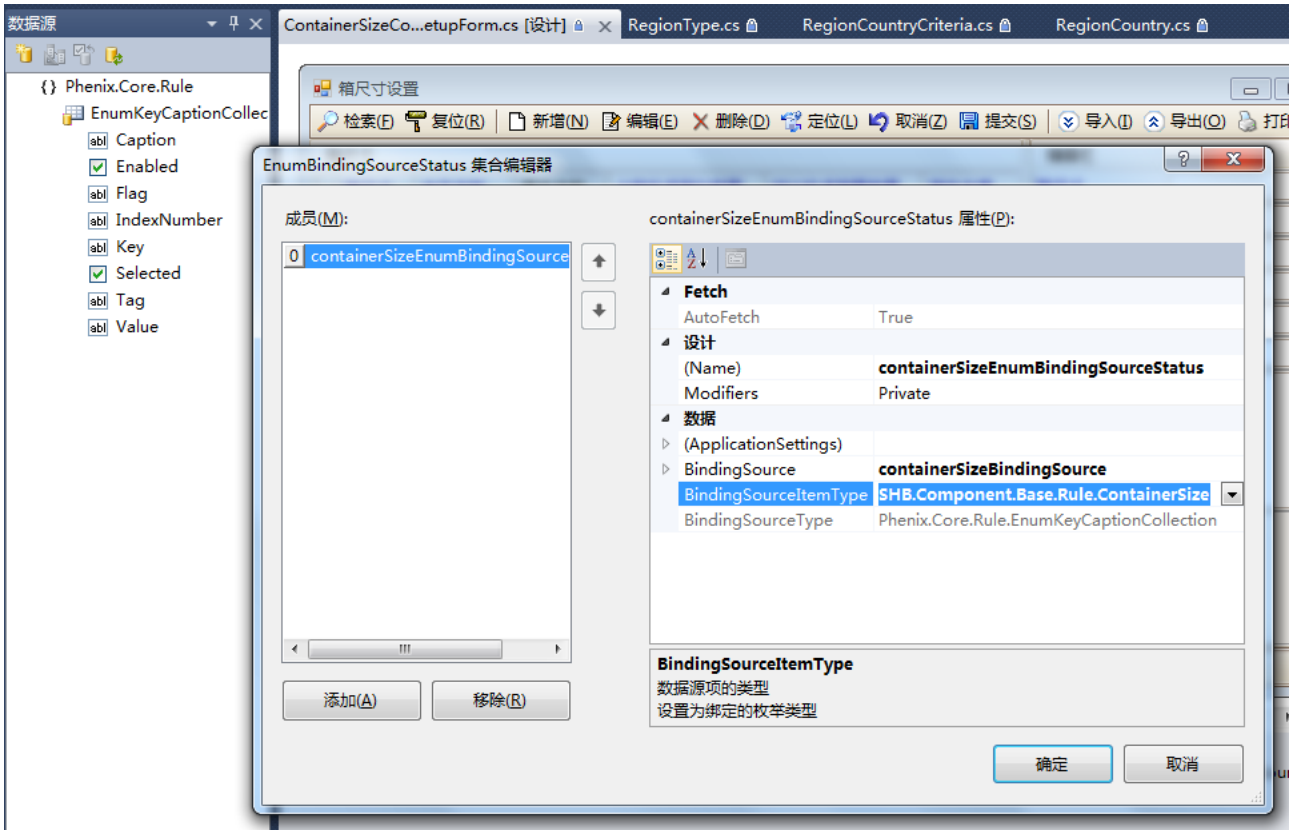
使用枚举需注意的事项：

- 如果要增加枚举项，请用新的序列值，特别是系统上线后；
- 不允许随意交换枚举项的序列值，特别是系统上线后；
- 编写存储过程时，因为枚举是采取整型方式存储的，所以如果直接用整型来做判断条件的话，表达会不够清晰，也容易出错，往往还要做注释，此时，请使用公共常量（变量名和枚举名保持一致，变量值和枚举值保持一致）来包装枚举整型，再用在判断语句中；

3.12.2 为什么要为枚举打上标签

为枚举打上标签，主要是可以将枚举项封装为普通的数据集合（承载枚举数据的队列载体为：Phenix.Core.Rule.EnumKeyCaptionCollection），为在 UI 界面的设计开发上提供与业务类一致的数据绑定设计方法，从而减少界面控制层的代码量，达到快速设计开发的目的。

枚举数据集合往往被用在业务对象枚举属性的选择下拉清单及赋值、Caption 显示上。下图示意的是如何通过 Phenix.Windows.BarManager 的配置功能，将 Phenix.Core.Rule.EnumKeyCaptionCollection 的 BindingSource 与具体的枚举类型关联起来，实现枚举数据集合的自动填充：

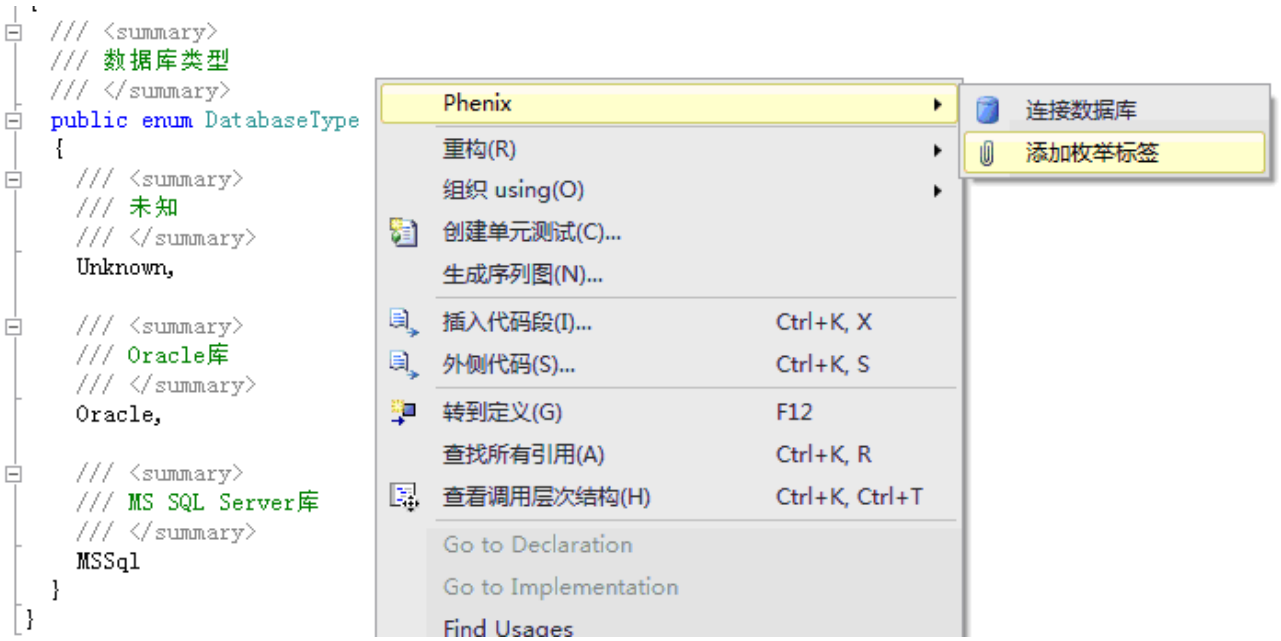


如果需要手工填充的话，举例如下：

```
this.readLevelEnumKeyCaptionBindingSource.DataSource =
EnumKeyCaptionCollection.Fetch<ReadLevel>();
```

3.12.3 如何为枚举打上标签

在枚举编辑界面上用鼠标右键点出浮动菜单，选择：



自动生成枚举标签代码：

```
  /// <summary>
  /// 数据库类型
  /// </summary>
  [Phenix.Core.Operate.KeyCaptionAttribute(FriendlyName = "数据库类型"),
System.SerializableAttribute()]
  public enum DatabaseType
  {
    /// <summary>
    /// 未知
    /// </summary>
    [Phenix.Core.Rule.EnumCaptionAttribute("未知")]
    Unknown,

    /// <summary>
    /// Oracle库
    /// </summary>
    [Phenix.Core.Rule.EnumCaptionAttribute("Oracle库")]
    Oracle,

    /// <summary>
    /// MS SQL Server库
```

```
    /// </summary>
    [Phenix.Core.Rule.EnumCaptionAttribute("MS SQL Server库")]
    MSSql
  }
```

3.13 可配置化对象

业务组件作为业务系统原子级的组件，其可配置化特性影响到整个业务系统的可配置化能力，而对象的可配置化是业务组件基本的功能外需求。

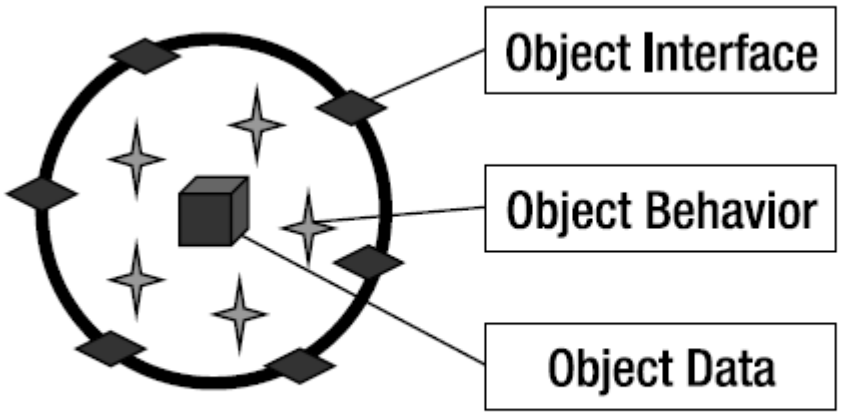


Figure 1-14. *A business object composed of state, implementation, and interface*

上图摘自 CSLA 作者的《Expert C# 2008 Business Objects》。

对象的可配置化，就是在不通过硬编码的前提下，可以动态地改变其实际运行当中的业务逻辑。

我们知道，业务逻辑是通过算法来实现的，而算法的基本逻辑结构有：顺序结构、条件分支结构、循环结构。同样，在任何一个程序设计语言里，也仅包含着 5 种最基本的算法语句：输入语句、输出语句、赋值语句、条件语句、循环语句，如“天有五行，水火金木土，分时化育，以成万物”一般，繁衍出千变万化的软件世界。

对于对象来说，其 Object Interface 包含着输入语句、输出语句，而 Object Behavior 则包含着赋值语句、条件语句、循环语句，它们的关键核心是表达式（赋值表达式、条件表达式）及其参数。对象的可配置化，就是考虑如何在系统的运行期，通过改变这些参数来干预业务逻辑。也就是将原本是表达式里的常量参数改写成变量，这些变量的值被持久化并通过配置界面来修改，以影响到业务数据的结果或者业务逻辑的走向。

表达式参数值有两种持久化方法：一种是存放在本地的 config 文件里，其输入、输出接口

由 Phenix.Core.AppSettings 静态类提供；一种是集中存放在数据库的指定表中，其输入、输出接口由 Phenix.Core.Dictionary.Configuration 静态类提供。

下面，我们从具体应用的案例上，来分别介绍这两种配置的用法。

3.13.1 类的本地配置

以客户端程序连接 WCF 服务的端口号为例，这样的配置参数应该存放在客户端本地：

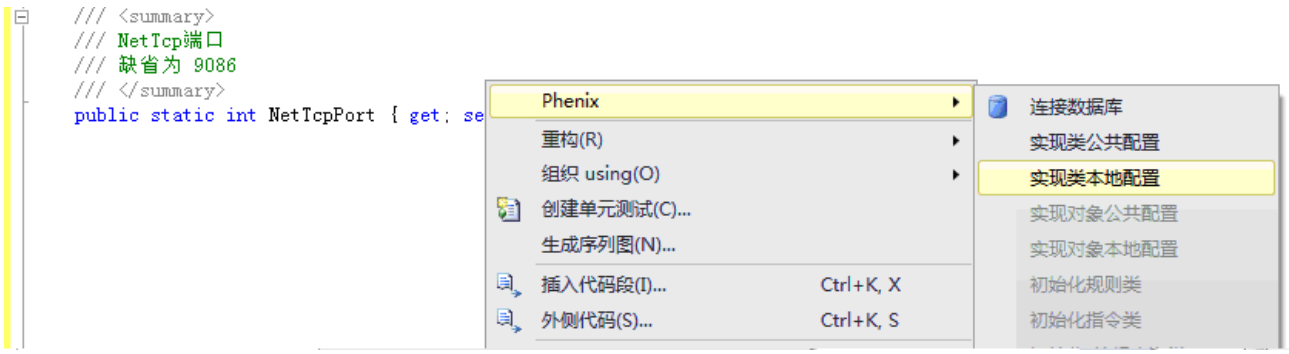
```
private static int? _netTcpPort;
/// <summary>
/// NetTcp端口
/// 缺省为 9086
/// </summary>
public static int NetTcpPort
{
  get { return AppSettings.GetProperty(ref _netTcpPort, 9086); }
  set { AppSettings.SetProperty(ref _netTcpPort, value); }
}
```

我们从头演示。

先写一个静态属性：

```
/// <summary>
/// NetTcp端口
/// 缺省为 9086
/// </summary>
public static int NetTcpPort { get; set; }
```

然后，在属性上用鼠标右键点出浮动菜单，选择：



生成代码：

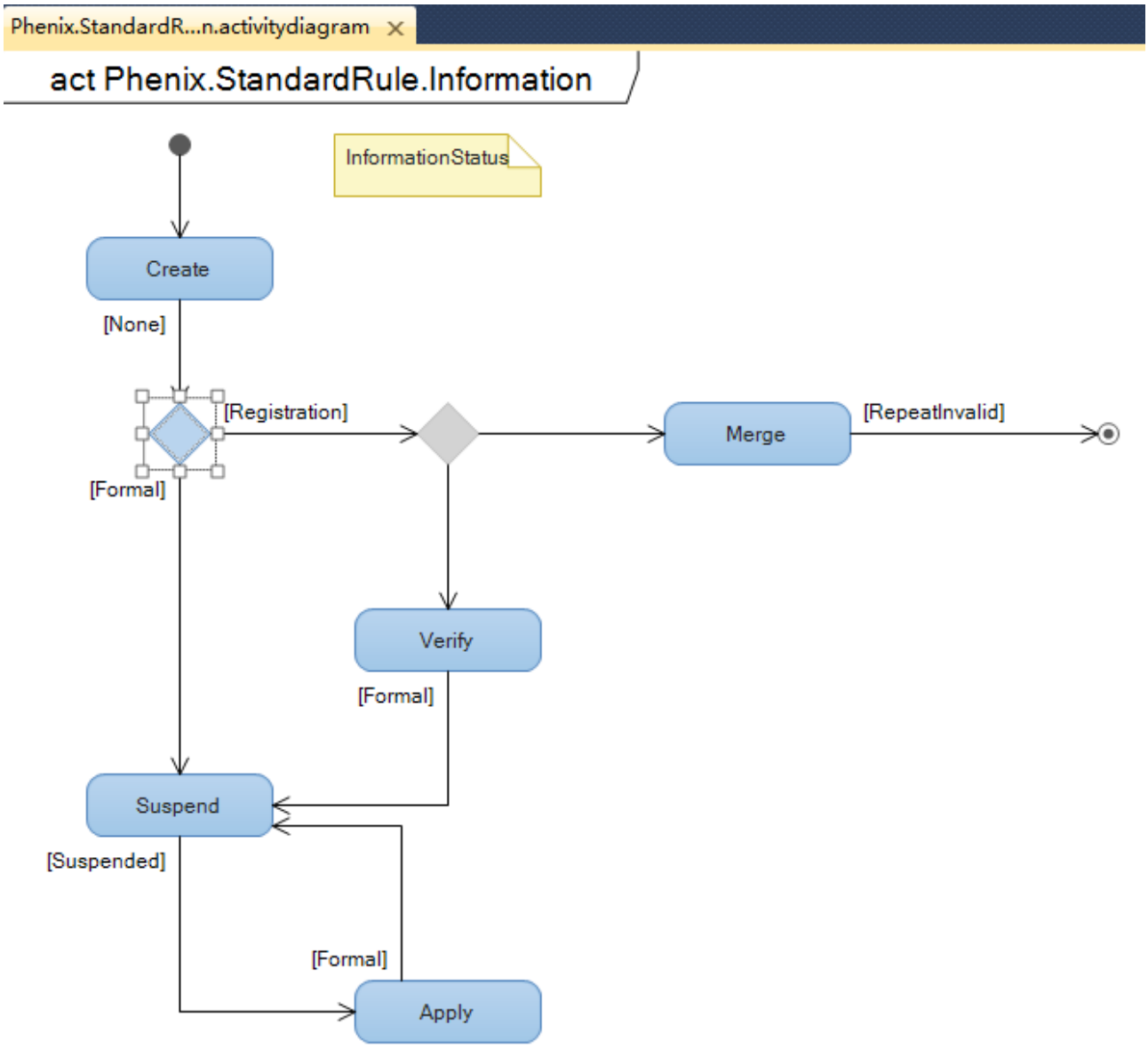
```
/// <summary>
/// NetTcp端口
/// 缺省为 9086
/// </summary>
private static int? _netTcpPort;
/// <summary>
/// NetTcp端口缺省为 9086
/// </summary>
public static int NetTcpPort
{
  get { return Phenix.Core.AppSettings.GetProperty(ref _netTcpPort, PleaseInputDefaultValue); }
  set { Phenix.Core.AppSettings.SetProperty(ref _netTcpPort, value); }
}
```

最后将缺省值 9086 填写上去就完成了。

3.13.2 类的公共配置

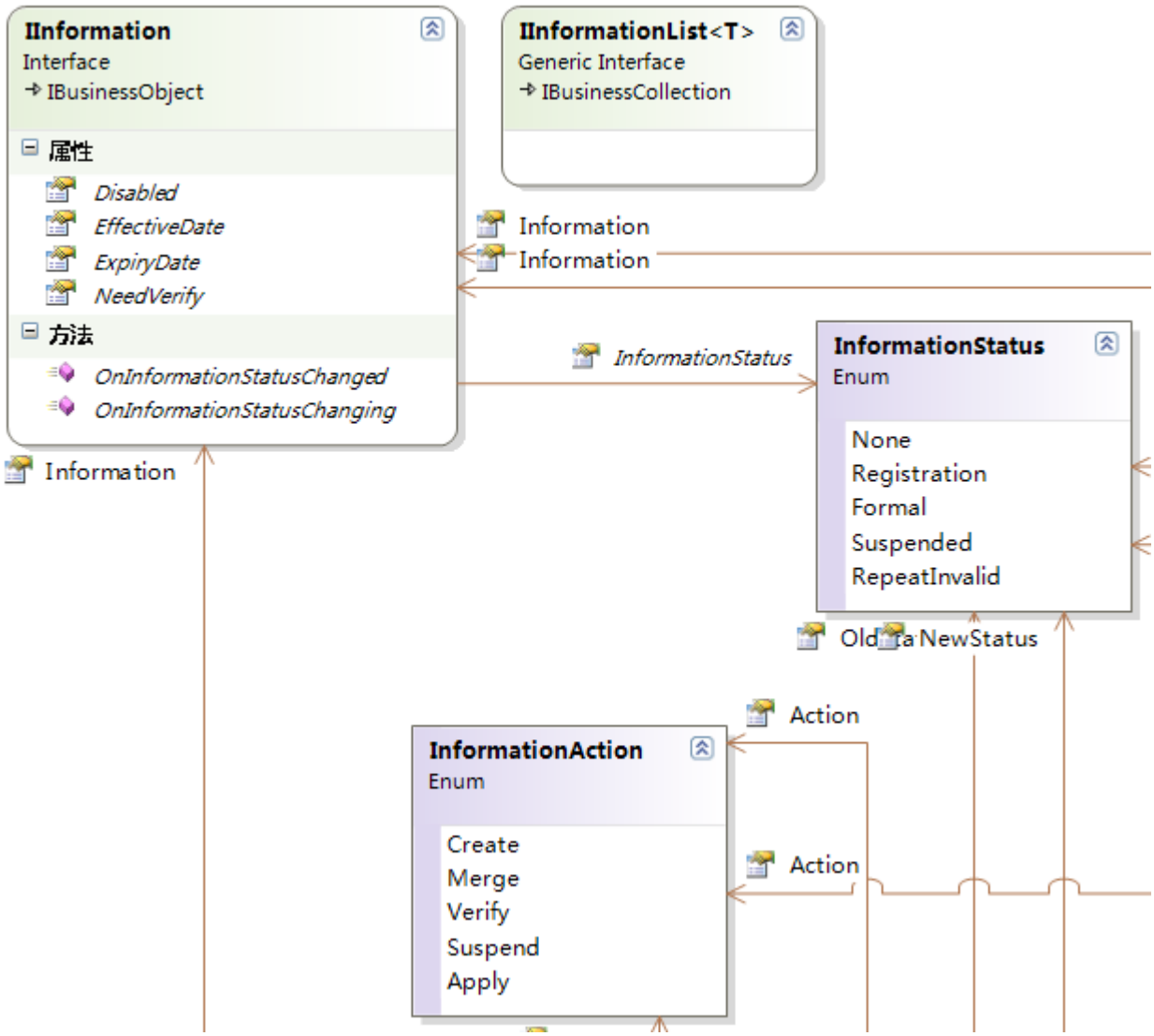
以标准化的资料处理机制为例。

信息系统中常用的功能是对资料的处理，Phenix丶为此提供了一套标准化的资料处理机制，由组件 Phenix.StandardRule.Information 负责实现，其中有对资料状态的控制：



上图为资料的活动图，其动作和状态分别对应着 InformationAction 和 InformationStatus 这两个枚举，决定了资料状态的变更。

此处，我们重点关注上图中选择框圈住的一个决策节点：NeedVerify。Information 组件根据这个决策节点的值来选择是否走审核流程，而这个决策节点的值则交给业务系统中具体的 Information 类来提供。为此，Phenix 提供了 IInformation 接口来限定这些 Information 类（使用 Information 组件的 Information 类必须实现 IInformation 接口）：



以 CustomerInfo 业务类为例，如果硬编代码的话，是这样的：

```
private static bool _needVerify = true;
/// <summary>
/// 是否需要审核
/// </summary>
public bool NeedVerify
{
    get { return _needVerify; }
}
```

其实可以：

生成代码：

```
    /// <summary>
    /// 是否需要审核
    /// </summary>
    private static bool? _needVerify;
    /// <summary>
    /// 是否需要审核
    /// </summary>
    [Phenix.Core.Mapping.PropertyAttribute(FriendlyName = "是否需要审核")]
    public bool NeedVerify
    {
      get { return Phenix.Core.Dictionary.Configuration.GetProperty(ref _needVerify,
PleaseInputDefaultValue); }
      set { Phenix.Core.Dictionary.Configuration.SetProperty(ref _needVerify, value); }
    }
```

最后将缺省值 true 填写上去就完成了。

3.13.3 对象的本地配置和公共配置

所谓对象的配置，是指 Phenix 允许对象以任何数据作为 key 值来持久化配置内容，而不是单一的配置值。

配置值的持久化方式同样也是分为本地的和公共的，编程接口和类的配置差异并不大。

下面以对象的公共配置为例，对 CustomerInfo 业务类的 NeedVerify 配置值进行扩展，比如我们可以通过将操作数据的用户所担任的岗位名称作为 key 值来持久化 NeedVerify 配置值，这样，审核流程的可有可无将受到当前用户岗位的影响。

生成代码：

```
        /// <summary>
        /// 是否需要审核
        /// </summary>
        private string NeedVerifyKey
        {
          get { return PleaseInputKeyValue; }
        }
        [NonSerialized]
        [Csla.NotUndoable]
        private bool? _needVerify;
        /// <summary>
        /// 是否需要审核
        /// </summary>
        [Phenix.Core.Mapping.PropertyAttribute(FriendlyName = "是否需要审核")]
        public bool NeedVerify
        {
          get { return Phenix.Core.Dictionary.Configuration.GetProperty(NeedVerifyKey, ref _needVerify,
PleaseInputDefaultValue); }
          set { Phenix.Core.Dictionary.Configuration.SetProperty(NeedVerifyKey, ref _needVerify, value); }
        }
```

最终代码是：

```
        /// <summary>
        /// 是否需要审核
        /// </summary>
        private string NeedVerifyKey
        {
          get
          {
            if (Phenix.Business.Security.UserPrincipal.User == null ||
```

```
            Phenix.Business.Security.UserPrincipal.User.Identity.Position == null)
            return null;
          return Phenix.Business.Security.UserPrincipal.User.Identity.Position.Name;
        }
      }
      [NonSerialized]
      [Csla.NotUndoable]
      private bool? _needVerify;
      /// <summary>
      /// 是否需要审核
      /// </summary>
      public bool NeedVerify
      {
        get { return Phenix.Core.Dictionary.Configuration.GetProperty(NeedVerifyKey, ref _needVerify,
true); }
        set { Phenix.Core.Dictionary.Configuration.SetProperty(NeedVerifyKey, ref _needVerify, value); }
      }
```

注意，当 Key 值为 null 时，其效果是和类的配置一样的。

总之，业务逻辑中被固化的是审核流程受到“什么”影响，而“什么”的值因配置化而可以动态地被改变。

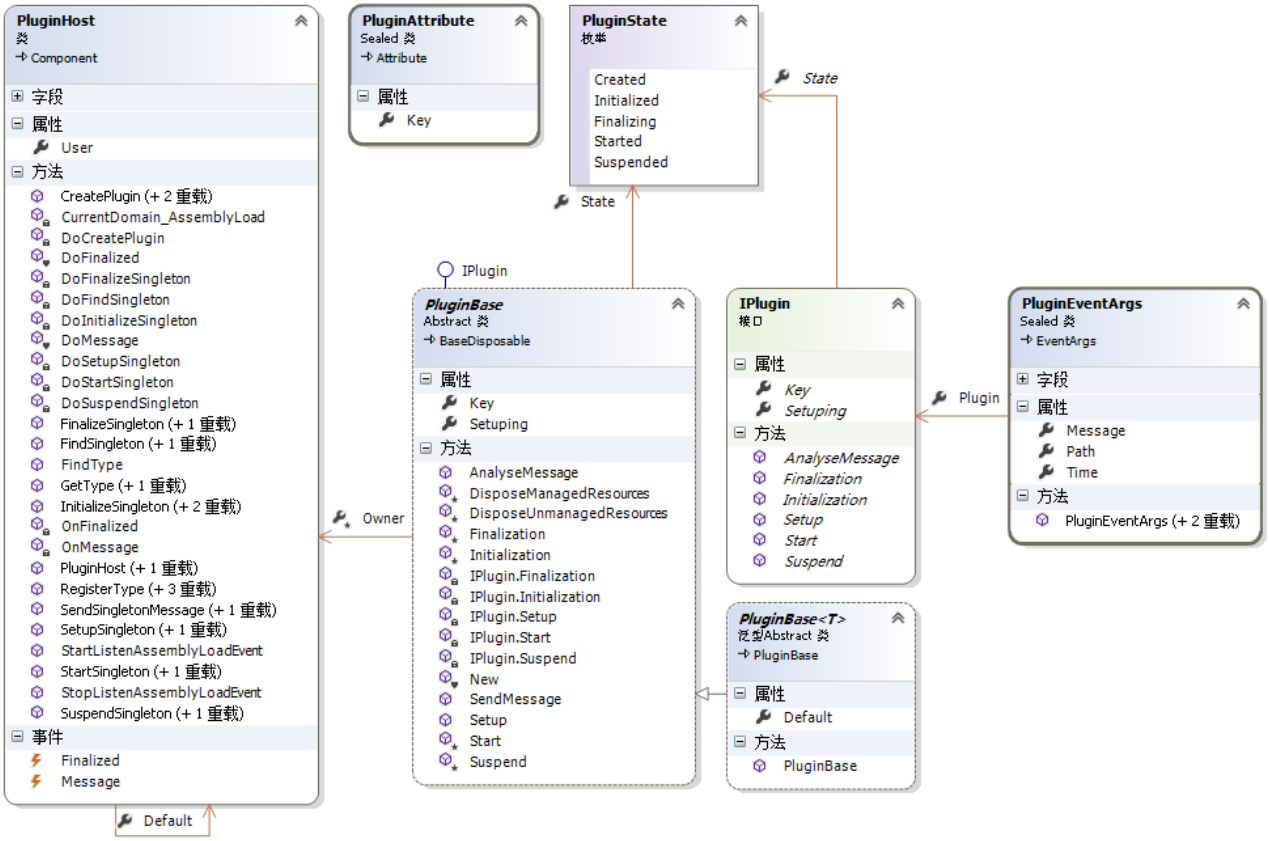
综上所述，利用对象的可配置化方法，我们可以实现简单的流程化配置功能。所谓“简单”，指的是我们只要事先设定了流程的所有节点和判断条件，就像铺设好的火车轨道和道岔一样，那么，我们在应用系统中开发出这每个判断条件参数值的配置界面（类比扳道工具），即可达到流程化的配置功能。

3.14 窗体插件类

插件是一种遵循一定规范的应用程序接口编写出来的程序或组件。

Phenix丫为编写插件提供了一整套的规范和管理插件的机制，核心是由插件基类（PluginBase）和插件容器（PluginHost）组成。应用系统通过 PluginHost 对象操作继承自 PluginBase 的插件对象，而插件对象可以将自身的消息通过 PluginHost 回馈给应用系统。

这些插件从物理位置和应用功能上可分为窗体插件和服务插件。但不管如何使用，都将在以下图的技术框架之上进行搭建：



3.14.1 窗体插件

窗体插件是插件在系统展现层的应用，从而规范了窗体组件的开发，以及使得系统升级成为很便利的一件事情，而启动一个插件则仅需执行一行代码：

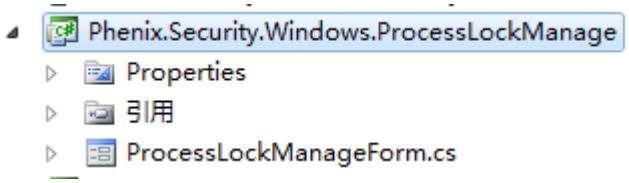
```
PluginHost.Default.SendSingletonMessage(插件程序集名称, this);
```

在实际开发中，黄底红字部分应被替换为你希望加载的插件程序集名称，比如以下示例里：

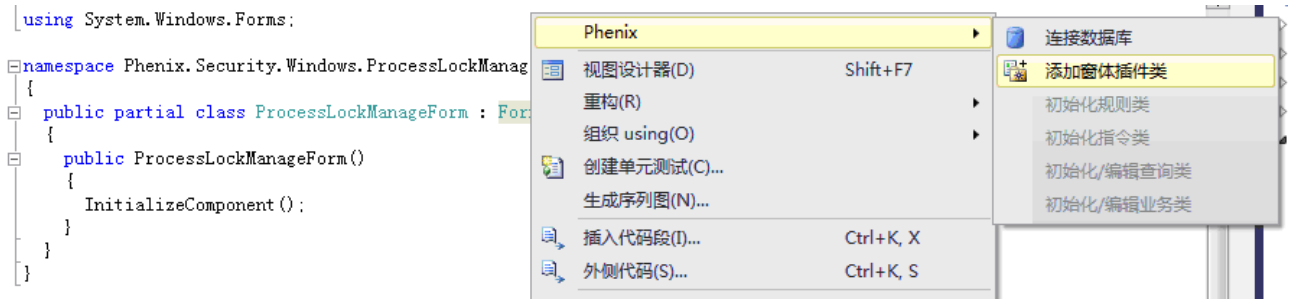
```
PluginHost.Default.SendSingletonMessage("Phenix.Security.Windows.ProcessLockManage", null);
```

3.14.2 演示如何快速搭建一个窗体插件

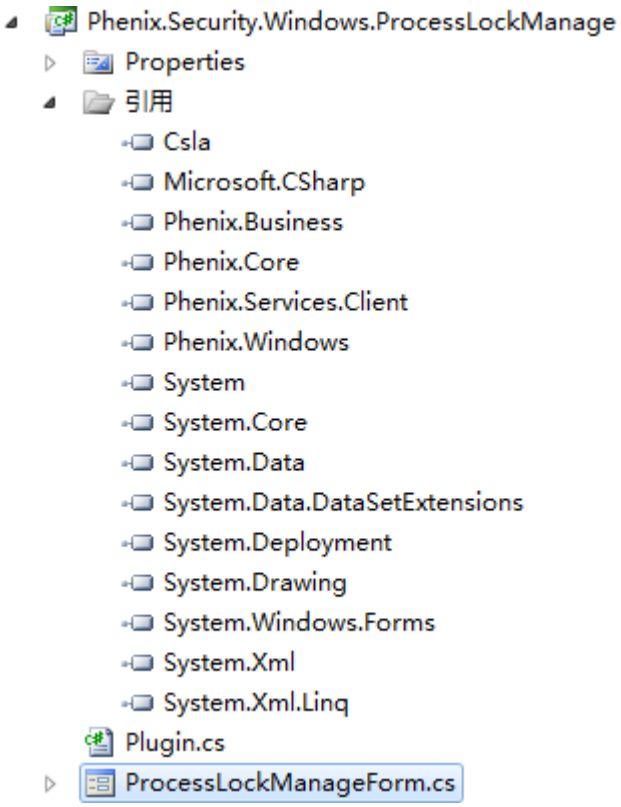
首先构建一个界面组件工程，在工程里建一个主窗体：



打开主窗体的编辑界面，在窗体类上用鼠标右键点出浮动菜单，选择：



会自动将普通的窗体类继承自 Phenix.Core.Windows.BaseForm，并在工程中添加了对应的窗体插件类：



该窗体插件类是这样的：

```
public class Plugin : PluginBase<Plugin>
{
  private BaseForm _mainForm;

  /// <summary>
  /// 分析消息
```

```
    /// 由 PluginHost 调用
    /// </summary>
    /// <param name="message">消息</param>
    /// <returns>按需返回</returns>
    public override object AnalyseMessage(object message)
    {
      BaseForm ownerForm = message as BaseForm;
      if (ownerForm != null && ownerForm.IsMdiContainer)
      {
        _mainForm = BaseForm.ExecuteMdi<ProcessLockManageForm>(ownerForm);
        return _mainForm;
      }
      return BaseForm.ExecuteDialog<ProcessLockManageForm>(message);
    }
  }

  /// <summary>
  /// 可变更程序集输出类型以用于调试
  /// </summary>
  static class Program
  {
    [STAThread]
    static void Main()
    {
      Application.EnableVisualStyles();
      Application.SetCompatibleTextRenderingDefault(false);

      //设为调试状态
      Phenix.Core.AppConfig.Debugging = true;
      //模拟登陆
      Phenix.Business.Security.UserPrincipal.User =
        Phenix.Business.Security.UserPrincipal.CreateAuthenticated(
          UserIdentity.AdminId, UserIdentity.AdminUserName, UserIdentity.AdminUserNumber,
          String.Empty, new long?(), new long?());
      Phenix.Services.Client.Library.Registration.RegisterEmbeddedWorker(false);
      //模拟启动插件
      PluginHost.Default.SendSingletonMessage("Phenix.Security.Windows.ProcessLockManage", null);
    }
```