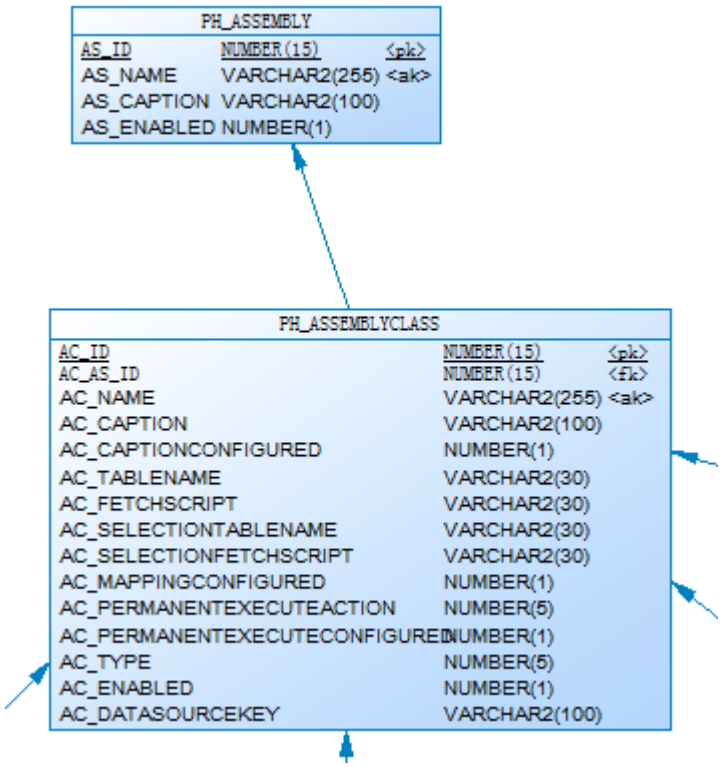
12 业务结构对象模型

12.8 业务数据的特殊处理

12.8.1从业务类中包含主表信息

一对多表结构关系在关系数据库中是最为常见的数据结构：



在一些业务场景中，我们往往需要在一个清单列表中，既能看到业务对象自己所映射的表字段信息，也能看到其关联主表的表字段信息（不一定是全部字段）。这样，要么在表结构设计中添加冗余字段，要么在业务类的设计上采取一些手段，这些都能实现同样的效果。不过，相对来说，在满足系统性能需求的前提下，尽量少采用冗余字段的方法。

12.8.1.1 在主从关联视图的业务对象中 Fetch 出主业务对象

见“11.业务对象生命周期及其状态”的“Fetch 业务对象-从本地获取业务对象-Phenix.Business.BusinessBase<T>提供从 source 业务对象中 Fetch 出另一种类的业务对象的函数”章节。

这种方法，虽然需要编写数据库视图，但有一定的灵活性，可以明确定义需要获取的字段以及过滤条件。

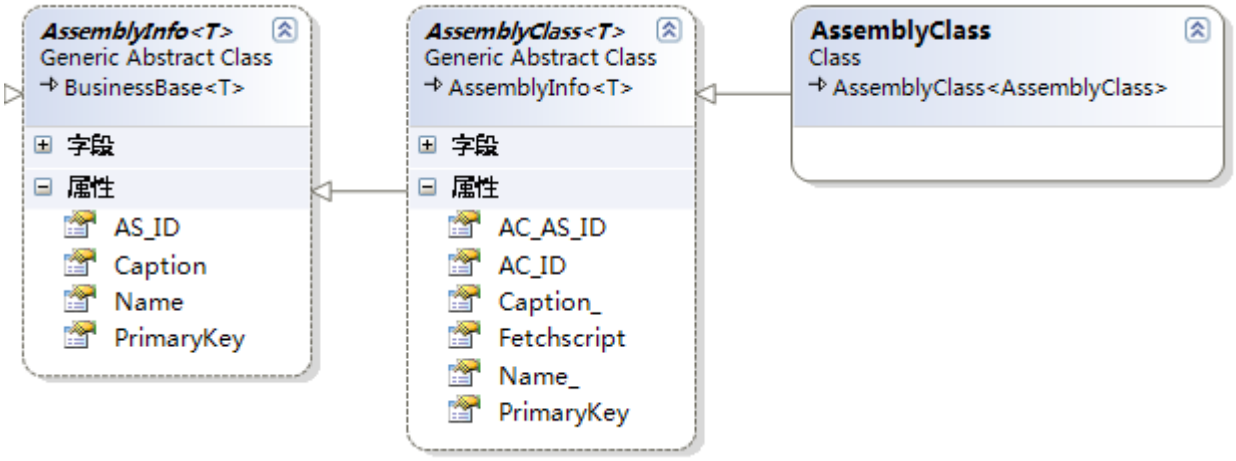
不过，此案例中需要自己编写 Fetch 主业务对象，然后在提交数据的事件函数中，还要自己编写主业务对象的提交代码。这样，虽不失灵活，但一般场景下，如无特殊逻辑处理，就显得太多余了。为此，

Phenix丶在 Phenix.Core.Mapping.FieldLinkAttribute 中，提供申明托管处理主业务对象的方法，可免去这些繁琐。具体方法，见工程：Phenix.Test.使用指南.12.8.1.1。

12.8.1.2 从业务类继承自主业务类，可自动提交主业务数据

使用本方法，无需编写数据库视图，但要求子业务类是父业务类的从业务类，且包含有关联主业务类的外键字段。

Phenix丶可以通过这种业务类的继承关系、数据的主从关系，自动拼装出与构建数据库视图相类似的 select 语句：




```
/// <summary>
///  程序集类信息
/// </summary>
[Serializable]
public class AssemblyClass : AssemblyClass<AssemblyClass>
{

}

/// <summary>
/// 程序集类信息清单
/// </summary>
[Serializable]
public class AssemblyClassList : Phenix.Business.BusinessListBase<AssemblyClassList, AssemblyClass>
{
}

/// <summary>
/// 程序集类信息
/// </summary>
[Phenix.Core.Mapping.ClassAttribute("PH_ASSEMBLYCLASS",FriendlyName = "程序集类信息"),
System.SerializableAttribute(), System.ComponentModel.DisplayNameAttribute("程序集类信息")]
```

```
  public abstract class AssemblyClass<T> : AssemblyInfo<T> where T : AssemblyClass<T>
  {
    [System.ComponentModel.Browsable(false)]
    [System.ComponentModel.DataAnnotations.Display(AutoGenerateField = false)]
    public override string PrimaryKey
    {
      get { return String.Format("{0}", AC_ID); }
    }

    /// <summary>
    /// AC_ID
    /// </summary>
    public static readonly Phenix.Business.PropertyInfo<long?> AC_IDProperty = RegisterProperty<long?>(c => c.AC_ID);
    [Phenix.Core.Mapping.Field(FriendlyName = "AC_ID", TableName = "PH_ASSEMBLYCLASS", ColumnName = "AC_ID", IsPrimaryKey = true, NeedUpdate = true)]
    private long? _AC_ID;
    /// <summary>
    /// AC_ID
    /// </summary>
    [System.ComponentModel.DisplayName("AC_ID")]
    public long? AC_ID
    {
      get { return GetProperty(AC_IDProperty, _AC_ID); }
      set { SetProperty(AC_IDProperty, ref _AC_ID, value); }
    }

    /// <summary>
    /// AC_AS_ID
    /// </summary>
    public static readonly Phenix.Business.PropertyInfo<long?> AC_AS_IDProperty = RegisterProperty<long?>(c => c.AC_AS_ID);
    [Phenix.Core.Mapping.FieldLink(typeof(Assembly) , "PH_ASSEMBLY", "AS_ID")]
    [Phenix.Core.Mapping.Field(FriendlyName = "AC_AS_ID", TableName = "PH_ASSEMBLYCLASS", ColumnName = "AC_AS_ID", NeedUpdate = true)]
    private long? _AC_AS_ID;
    /// <summary>
    /// AC_AS_ID
    /// </summary>
    [System.ComponentModel.DisplayName("AC_AS_ID")]
    public long? AC_AS_ID
    {
      get { return GetProperty(AC_AS_IDProperty, _AC_AS_ID); }
      set { SetProperty(AC_AS_IDProperty, ref _AC_AS_ID, value); }
    }
```

```
    /// <summary>
    /// AC_NAME
    /// </summary>
    public static readonly Phenix.Business.PropertyInfo<string> Name_Property =
RegisterProperty<string>(c => c.Name_);
    [Phenix.Core.Mapping.Field(FriendlyName = "AC_NAME", Alias = "AC_NAME", TableName =
"PH_ASSEMBLYCLASS", ColumnName = "AC_NAME", NeedUpdate = true, InLookUpColumn = true,
InLookUpColumnDisplay = true)]
    private string _name_;
    /// <summary>
    /// AC_NAME
    /// </summary>
    [System.ComponentModel.DisplayName("AC_NAME")]
    public string Name_
    {
      get { return GetProperty(Name_Property, _name_); }
      set { SetProperty(Name_Property, ref _name_, value); }
    }

    /// <summary>
    /// AC_CAPTION
    /// </summary>
    public static readonly Phenix.Business.PropertyInfo<string> Caption_Property =
RegisterProperty<string>(c => c.Caption_);
    [Phenix.Core.Mapping.Field(FriendlyName = "AC_CAPTION", Alias = "AC_CAPTION", TableName =
"PH_ASSEMBLYCLASS", ColumnName = "AC_CAPTION", NeedUpdate = true)]
    private string _caption_;
    /// <summary>
    /// AC_CAPTION
    /// </summary>
    [System.ComponentModel.DisplayName("AC_CAPTION")]
    public string Caption_
    {
      get { return GetProperty(Caption_Property, _caption_); }
      set { SetProperty(Caption_Property, ref _caption_, value); }
    }

    /// <summary>
    /// AC_FETCHSCRIPT
    /// </summary>
    public static readonly Phenix.Business.PropertyInfo<string> FetchscriptProperty =
RegisterProperty<string>(c => c.Fetchscript);
    [Phenix.Core.Mapping.Field(FriendlyName = "AC_FETCHSCRIPT", Alias = "AC_FETCHSCRIPT", TableName =
"PH_ASSEMBLYCLASS", ColumnName = "AC_FETCHSCRIPT", NeedUpdate = true)]
    private string _fetchscript;
    /// <summary>
```

```
    /// AC_FETCHSCRIPT
    /// </summary>
    [System.ComponentModel.DisplayName("AC_FETCHSCRIPT")]
    public string Fetchscript
    {
      get { return GetProperty(FetchscriptProperty, _fetchscript); }
      set { SetProperty(FetchscriptProperty, ref _fetchscript, value); }
    }
  }
}
```

这种业务类的设计方法与普通业务类的区别，在于它从主业务类（泛型类）上继承，在属性定义上，需有主表关联的外键字段及其映射关系（不管它是否被自身业务对象用到），并且，如果数据库中没有构建对应的物理外键（也就是在逻辑上具备了主外键的关联关系，但是由业务逻辑层自行控制业务数据的完整性）的话，还需要在这个外键映射字段上显式标记上 Phenix.Core.Mapping.FieldLinkAttribute（否则可以不必标记，因为在初始化业务类的时候 Phenix 会根据数据库的数据字典为它补上的）。

以下是 Fetch 上述业务对象时 Phenix 自动拼装出的 select 语句：

```
<Command Text="select  AC_ID,AC_AS_ID,AC_NAME,AC_CAPTION,AC_FETCHSCRIPT,AS_ID,AS_NAME,AS_CAPTION from PH_ASSEMBLYCLASS,PH_ASSEMBLY where AC_AS_ID=AS_ID" />
```

如果需要自动提交主业务的数据，在外键映射字段的 Phenix.Core.Mapping.FieldLinkAttribute 标记上，可以申明主业务数据的业务类：

```
    /// <summary>
    /// AC_AS_ID
    /// </summary>
    public static readonly Phenix.Business.PropertyInfo<long?> AC_AS_IDProperty = RegisterProperty<long?>(c => c.AC_AS_ID);
    [Phenix.Core.Mapping.FieldLink(typeof(Assembly) , "PH_ASSEMBLY", "AS_ID")]
    [Phenix.Core.Mapping.Field(FriendlyName = "AC_AS_ID", TableName = "PH_ASSEMBLYCLASS", ColumnName = "AC_AS_ID", NeedUpdate = true)]
    private long? _AC_AS_ID;
    /// <summary>
    /// AC_AS_ID
    /// </summary>
    [System.ComponentModel.DisplayName("AC_AS_ID")]
    public long? AC_AS_ID
    {
      get { return GetProperty(AC_AS_IDProperty, _AC_AS_ID); }
      set { SetProperty(AC_AS_IDProperty, ref _AC_AS_ID, value); }
```

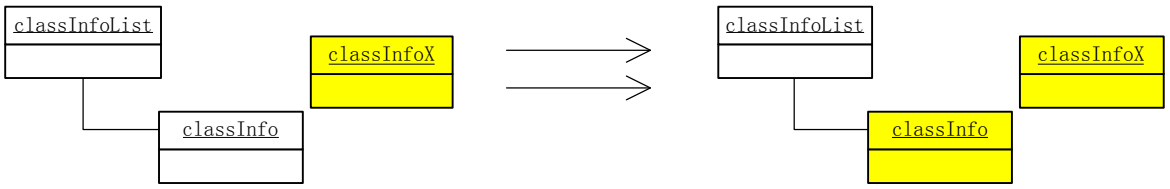
```
}
```

代码见工程：Phenix.Test.使用指南.12.8.1.2。

12.8.2业务对象的“附体”

在一些应用场景里，当前操作的业务对象是被各种对象引用（比如被 BindingSource 绑定）的，但此时又希望在引用不变的情况下处理另外一个同类型的业务对象 X。这时，如果我们把当前对象的业务数据和状态数据都替换为 X 的，就等同于操作 X 对象了，也就是说这两个物理对象成为了同一个业务对象，原来的业务对象在内存里消失了。

为了实现这个功能，必须将对象 X 的业务数据等信息覆盖到这个当前操作的对象里，类似于“附体”的行为。从数据结构上来讲，就是变更了它与表记录的对应关系；从业务逻辑上来讲，它和 source 对象拥有了相同的业务数据。



比如：

```
classInfoList.classInfo.ReplaceFrom(classInfoX);
classInfoList.classInfo.Save(); // == classInfoX.Save()
```

当前操作的对象调用 ReplaceFrom()时，除了将 source 对象的状态信息、所有业务数据都被带过来外，还会将 source 业务对象的 OldFieldValues、Details、Links 一并 Clone 过来。所以，提交它就等于提交了 source 对象。

由于 ReplaceFrom()函数是虚拟函数，业务类可覆写它，添加上自己需置换的内容：

```
/// <summary>
/// 置换为与source相同内容的对象
/// </summary>
public virtual void ReplaceFrom(T source)
```

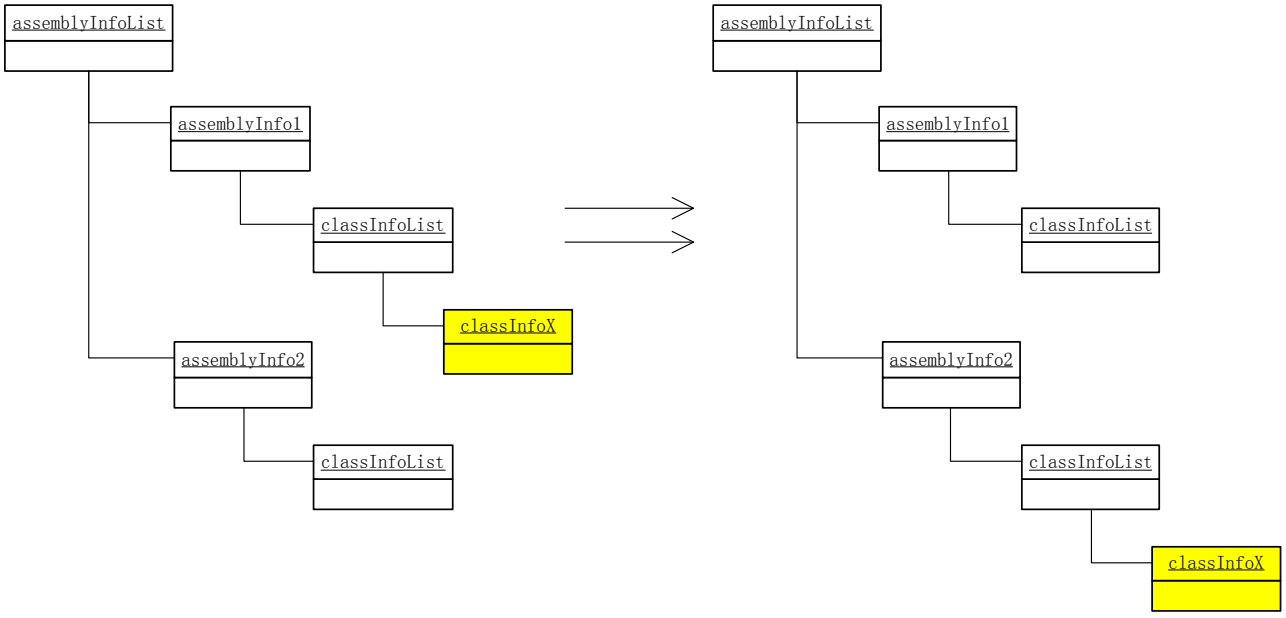
12.8.3业务对象换“东家”

所谓业务对象换“东家”，从数据结构上来讲，就是变更它的外键值为新的主表记录；从业务逻

辑、业务结构上来讲，就是更换它的 Owner 属性值为新的主业务对象。

12.8.3.1 Move To

如果操作的业务对象存在于如下的业务结构中，希望从老东家 assembyInfo1 换成新东家 assembyInfo2：



可以：

```
assemblyInfo1.classInfoList.Remove(classInfoX);
assemblyInfo2.classInfoList.Add(classInfoX);
assemblyInfoList.Save();
```

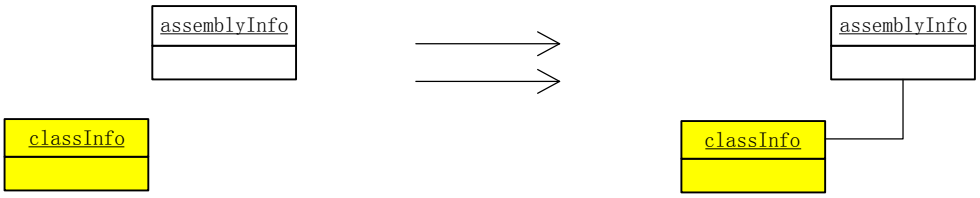
也可以：

```
classInfoX.MoveTo(assemblyInfo2.classInfoList);
assemblyInfoList.Save();
```

这两套代码实现的效果是一样的。

12.8.3.2 Link To

如果操作的是独立的“东家”对象，即 assemblyInfo 不存在于上述的业务结构中：



则可以采取：

```
classInfo.LinkTo(assemblyInfo);
classInfo.Save();
```

12.8.3.3 持久化

上述几种处理方式，持久化效果都是一样的：PH_AssemblyClass（程序集类）表记录的 AC_AS_ID（所属程序集表记录外键）从老东家 assembyInfo1 的 ID 值 update 成新东家 assembyInfo2 的 ID 值。

12.8.4Deleted As Disabled

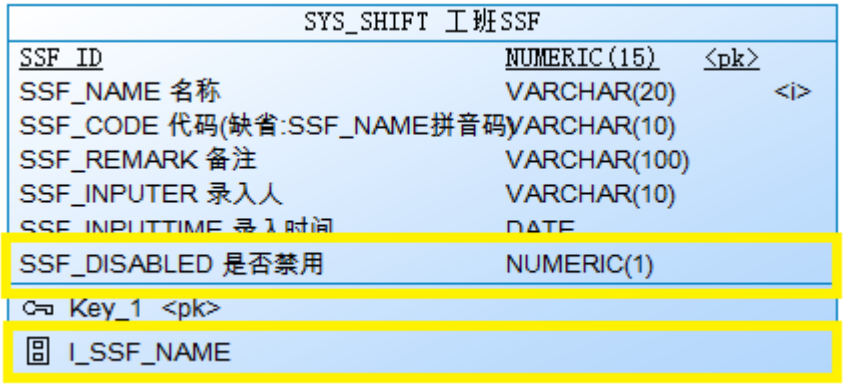
在应用系统中一些关键记录虽然已经不再被使用(即 Disabled)，但是不允许彻底从数据库中删除，而要在正常业务操作中过滤掉这些被 Disabled 的记录，Phenix 持久层引擎为此做了特殊处理。

12.8.4.1 数据结构上的设计要求

首先，需要在表结构上：

- 构建一个“禁用字段”，用它标记“当前记录是否被禁用”的开关值，以区别于正常记录；
- 至少构建一个唯一键索引，用它来触发恢复 Disabled 记录的操作；

举例如下：



当删除这种含有禁用字段的业务对象并持久化的时候，实际没有真正删除它对应的记录，而是将这条记录上的禁用字段赋值为 Phenix.Core.Mapping.CodingStandards.DefaultDisabledTrueValue 内容。

以下为 Phenix丶在 Phenix.Core.Mapping.CodingStandards 类中约定的表字段命名、表字段值使用规则：

属性	说明	备注
DefaultDisabledColumnName	缺省“禁用标识”字段名	缺省值：字段名后缀为“_DISABLED”
DefaultDisabledTrueValue	缺省“禁用”字段值	缺省值：字段值为“1”
DefaultDisabledFalseValue	缺省“可用”字段值	缺省值：字段值为“0”

并且，表字段类型必须定义为 NUMERIC(1)（注：可自动映射为布尔型业务类字段和属性）。

当表字段是以上述格式命名的，则这个字段必定是 Disabled 标记字段(除非 AllowReservedColumn = false)，并强制纳入到 Phenix丶的“Deleted As Disabled”服务的管理范围内。

12.8.4.2 业务结构上的设计要求

要实现上述功能，业务类里必须定义包含有 Phenix.Core.Mapping.FieldAttribute 标记 IsDisabledColumn = true 的字段：

```
        /// <summary>
        /// 是否禁用
        /// </summary>
        public static readonly Phenix.Business.PropertyInfo<bool?> DisabledProperty =
RegisterProperty<bool?>(c => c.Disabled,false);
        [Phenix.Core.Mapping.Field(FriendlyName = "是否禁用", Alias = "SSF_DISABLED", TableName =
"SYS_SHIFT", ColumnName = "SSF_DISABLED", NeedUpdate = true, OverwritingOnUpdate = true, IsDisabledColumn
= true)]
        private bool? _disabled;
        /// <summary>
        /// 是否禁用
        /// </summary>
        [System.ComponentModel.DisplayName("是否禁用")]
        public bool? Disabled
        {
            get { return GetProperty(DisabledProperty, _disabled); }
            set { SetProperty(DisabledProperty, ref _disabled, value); }
        }
```

当 ColumnName 符合 Phenix.Core.Mapping.CodingStandards.DefaultDisabledColumnName 规范时必定是禁用字段（除非 AllowReservedColumn = false）。

要判断业务对象是否是可禁用的，从 Phenix.Business.BusinessBase<T>的下述静态属性值可以知道：

属性	说明	备注
DeletedAsDisabled	删除即禁用	当包含禁用字段(FieldAttribute.IsDisabledColumn = true)且存在唯一键时为 true;

12.8.4.3 Fetch 被 Disabled 的记录

缺省情况下，Fetch 出来的业务对象清单里是不包含 Disabled 记录的，但在有些业务场景下会需要浏览它们，比如在检索历史记录的时候，记录里一些字段的关联代码有可能已被 Disabled，此时不应该屏蔽掉这些被 Disabled 的代码表记录。

12.8.4.3.1 查询类

如果 Fetch 的参数是使用查询类的话，可以在查询类里添加与禁用字段对应的查询项：

```
/// <summary>
/// 工班过滤条件
/// </summary>
[Serializable]
public class ShiftCriteria : Phenix.Business.CriteriaBase
{
   [CriteriaField(Operate = CriteriaOperate.Equal, Logical = CriteriaLogical.And, FriendlyName = "是否禁用", ColumnName = "SSF_DISABLED", IsDisabledColumn = true)]
    private bool? _disabled;
    /// <summary>
    /// 是否禁用
    /// </summary>
    public bool? Disabled
    {
        get { return _disabled; }
        set { _disabled = value; }
    }
}
```

Fetch 时可以这样调用：

```
ShiftList.Fetch(new ShiftCriteria() { Disabled == true; });
```

12.8.4.3.2 条件表达式

如果 Fetch 的参数是使用条件表达式的话，可以在条件表达式里添加与禁用字段对应的查询项：

```
/// <summary>
/// 工班清单
/// </summary>
[Serializable]
public class ShiftList : Phenix.Business.BusinessListBase<ShiftList, Shift>
{
  /// <summary>
  /// 检索全部的工班
  /// </summary>
  protected ShiftList FetchAll()
  {
    return ShiftList.Fetch(Shift.DisabledProperty == true);
  }
}
```