ГЛАВА 8

Управление состоянием объектов

Следующая тема, которую я хочу обсудить в реализации инфраструктуры CSLA .NET, — это то, как редактируемые объекты управляют информацией о состоянии. Редактируемые бизнес-объекты содержат наборы однозначной информации о состоянии. Управление этими значениями, по большей части, автоматизировано классами `BusinessBase`, `BusinessListBase` и порталом данных.

Свойства состояния объекта

Все редактируемые бизнес-объекты должны отслеживать, был ли объект только что создан, были ли его данные изменены или он был отмечен для удаления. Используя функциональные возможности правил проверки, объект может также отслеживать свою допустимость. Табл. 8.1 содержит список свойств состояния объектов классов `BusinessBase` и `BusinessListBase`.

Таблица 8.1. Свойства состояния объекта

Свойство	Описание
`IsNew`	Указывает, соответствует ли первично идентифицированное значение объекта в памяти первичному ключу в базе данных. Если соответствия нет, объект новый
`IsSelfDirty`	Указывает, отличаются ли данные объекта в памяти от данных в базе данных. Если отличаются, объект изменен
`IsDirty`	Указывает, был ли изменен сам объект или любой из его дочерних объектов
`IsSelfValid`	Указывает, есть ли у объекта в настоящее время нарушения правил проверки. Если это так, то объект недопустим
`IsValid`	Указывает, допустим ли сам объект и допустимы ли все его дочерние объекты
`IsSavable`	Указывает, может ли объект быть сохранен, комбинируя свойства `IsValid`, `IsDirty`, уровни авторизации и редактирования, а также любой невыполненный асинхронный запуск проверки правил
`IsDeleted`	Указывает, отмечен ли объект для удаления

Интерфейс ITrackStatus

В главе 6, "Реализация бизнес-инфраструктуры", я кратко затронул интерфейс ITrackStatus из пространства имен Csla.Core. Этот интерфейс реализован в классах BusinessBase и BusinessListBase, он позволяет вашему коду получать доступ к значениям состояния объекта, не волнуясь о специфическом типе объекта.

```
public interface ITrackStatus
{
   bool IsValid { get; }
   bool IsSelfValid { get; }
   bool IsDirty { get; }
   bool IsSelfDirty { get; }
   bool IsDeleted { get; }
   bool IsNew { get; }
   bool IsSavable { get; }
}
```

Этот интерфейс используется внутри самой инфраструктуры CSLA .NET, но он доступен также для авторов бизнес-объектов и инфраструктур UI.

Теперь давайте обсудим такие концепции объекта, как "новый", "измененный", "допустимый" и "отмеченный для удаления".

Свойство IsNew

Когда объект "нов", это значит, что он существует в памяти, но не в базе данных или другом постоянном хранилище. Если данные объекта располагаются в базе данных, объект считается "старым". Обычно я рассматриваю это так: если значение первичного ключа в объекте соответствует существующему значению первичного ключа в базе данных, то объект старый, в противном случае он новый.

Значение свойства IsNew хранится в поле _isNew. Когда объект создается впервые, стандартным значением этих свойств будет новый объект.

```
private bool _isNew = true;
```

Свойство IsNew просто предоставляет это значение.

```
[Browsable(false)]
public bool IsNew
{
   get { return _isNew; }
}
```

Свойство отмечено атрибутом Browsable из пространства имен System.ComponentModel. Этот атрибут указывает механизму связывания данных не связывать это свойство автоматически. Без этого атрибута механизм связывания данных автоматически отобразил бы это свойство в таблицах и формах, но обычно это свойство не должно быть отображено. Данный атрибут используется и в других свойствах класса BusinessBase.

Метод MarkOld

Если объект загружен данными из базы данных, поле _isNew устанавливается в состояние false при помощи защищенного метода MarkOld().

```
protected virtual void MarkOld()
{
   _isNew = false;
   MarkClean();
}
```

Обратите внимание на то, что этот процесс устанавливает также объект в состояние "неизменный" (clean), обсуждаемое в этой главе далее, при рассмотрении свойства `IsDirty`. Если данные объекта были только что загружены из базы данных, было бы логично предполагать, что данные объекта соответствуют данным в базе данных и еще не были изменены. Таким образом объект считается "неизменным".

Метод `MarkNew`

Есть также соответствующий метод `MarkNew()`.

```
protected virtual void MarkNew()
{
   _isNew = true;
   _isDeleted = false;
   MarkDirty();
}
```

Обычно эти методы автоматически вызывает портал данных через интерфейс `IDataPortalTarget`, но, будучи защищенными, они доступны также для автора бизнес-объекта. Это позволяет бизнес-объекту изменять свое состояние, чтобы справляться с теми редкими случаями, когда стандартное поведение не является подходящим.

Знание того, нов объект или стар, помогает реализации портала данных, как описано в главе 15, "Персистентность и портал данных". Свойство `IsNew` контролирует выбор, следует ли вставить данные в базу данных или модифицировать их.

Иногда свойство `IsNew` может быть полезным и для разработчика UI. Некоторые режимы UI могут отличаться у нового и существующего объектов. Способность редактировать данные первичного ключа объекта является хорошим примером; она зачастую присуща только до момента сохранения данных в базе данных. Когда объект становится старым, первичный ключ фиксируется.

Свойство `IsSelfDirty`

Объект считается измененным, когда значения в его полях не соответствуют значениям в базе данных. Если значения в полях объекта соответствуют значениям в базе данных, объект не является измененным. Практически невозможно знать всегда, соответствуют ли значения объекта значениям в базе данных, поэтому представленная здесь реализация работает, скорее, как "достоверное предположение". Реализация полагается на разработчика бизнес-логики, способного указать, когда объект изменяется и становится, таким образом, измененным.

Подобная концепция — объект считается измененным, если он или любой из его дочерних объектов были изменены. Эта концепция отражена свойством `IsDirty`, которое отличается от свойства `IsSelfDirty`, отражающего состояние только данного конкретного объекта (и не учитывает его дочерние объекты).

Вот текущее состояние значения, содержащегося в поле.

```
private bool _isDirty = true;
```

Затем значение предоставляется как свойство.

```
[Browsable(false)]
public virtual bool IsSelfDirty
{
   get { return _isDirty; }
}
```

Обратите внимание на то, что это свойство отмечено как `virtual`. Это важно, потому что иногда бизнес-объект становится измененным не просто потому, что изменились его данные. В данном случае разработчик бизнес-логики должен будет переопределить свойство `IsSelfDirty`, чтобы обеспечить более сложную реализацию.

На заметку. Хотя изменение режима свойства `IsSelfDirty` является редким случаем, эта возможность была затребована многими участниками сообщества; таким образом, свойство было сделано виртуальным.

Значением по умолчанию для свойства `IsSelfDirty` является `true`, поскольку значения поля нового объекта не будут соответствовать значениям в базе данных.

Метод `MarkClean`

Если впоследствии значения объекта загружаются из базы данных, то значение поля `_isDirty` изменяется на `false` при вызове метода `MarkOld()`, поскольку метод `MarkOld()` вызывает метод `MarkClean()`.

```
[EditorBrowsable(EditorBrowsableState.Advanced)]
protected void MarkClean()
{
   _isDirty = false;
   if (_fieldManager != null)
      FieldManager.MarkClean();
   OnUnknownPropertyChanged();
}
```

Этот метод не только устанавливает значение поля `_isDirty` в состояние `false`, но и вызывает метод `MarkClean()` класса `FieldManager`, чтобы отметить все содержащиеся им поля как неизменные. Как только объект и его поля отмечаются как неизменные, этот метод вызывает метод `OnUnknownPropertyChanged()`, реализованный в классе `Csla.Core.BindableBase`, чтобы передать событие `PropertyChanged` для всех свойств объекта. Это уведомляет механизм связывания данных об изменении объекта, в результате WPF и Windows Forms смогут обновить представление для пользователя. Более подробная информация о классе `BindableBase` и связывании данных приведена в главе 10, “Связывание данных”.

Метод `MarkDirty`

Есть также соответствующий метод `MarkDirty()`. Этот метод вызывают на разных этапах существования объекта, включая любое изменение значения свойства или вызов метода `MarkNew()`. Когда значение свойства изменяется, для этого свойства передается специфическое событие `PropertyChanged`.

Метод `MarkDirty()` может быть вызван и в других случаях, когда значение определенного свойства *не* изменено, событие `PropertyChanged` должно быть передано для всех свойств объекта. Так механизм связывания данных будет уведомлен об изменении, если *какое-нибудь* свойство объекта привязано к элементу управления UI.

Задача, таким образом, в том, чтобы гарантировать передачу по крайней мере одного события `PropertyChanged` при любом изменении состояния объекта. Если некое свойство изменено, то событие `PropertyChanged` должно быть передано *для этого свойства*. Но если нет никакого способа указать, какое именно свойство изменено (как при сохранении объекта в базе данных), то нет никакой реальной возможности передать событие `PropertyChanged` для каждого свойства.

Реализация этого потребует нескольких перегрузок метода `MarkDirty()`.

```
protected void MarkDirty()
{
   MarkDirty(false);
}

[EditorBrowsable(EditorBrowsableState.Advanced)]
protected void MarkDirty(bool suppressEvent)
{
   _isDirty = true;
   if (!suppressEvent)
      OnUnknownPropertyChanged();
}
```

Первая перегруженная версия может быть вызвана разработчиком бизнес-логики, если он хочет вручную отметить объект как измененный. Она предназначена для использования, когда измениться может неизвестное свойство.

Метод `PropertyHasChanged`

Вот вторая перегруженная версия, вызываемая методом `PropertyHasChanged()`.

```
protected virtual void PropertyHasChanged(string propertyName)
{
   MarkDirty(true);
   var propertyNames = ValidationRules.CheckRules(propertyName);
   if (ApplicationContext.PropertyChangedMode ==
                          ApplicationContext.PropertyChangedModes.Windows)
      OnPropertyChanged(propertyName);
   else
      foreach (var name in propertyNames)
         OnPropertyChanged(name);
}
```

Метод `PropertyHasChanged()` вызывают методы `SetProperty()`, обсуждаемые в главе 7, “Объявление свойств”, для указания на изменение определенного свойства. Обратите внимание на то, что в данном случае все правила проверки для свойства будут проверены (более подробная информация по этой теме приведена в главе 11, “Бизнес-правила и правила проверки”).

Совет. Этот метод объявлен как `virtual`, позволяя вам добавлять дополнительные этапы в процесс, если нужно. Кроме того, это значит, что при необходимости вы можете переопределить режим для реализации отслеживания изменений на уровне поля.

Передача события `PropertyChanged` для некоего измененного свойства, отмечает объект как измененный. Это не так просто, как хотелось бы, поскольку код должен вести себя по-другому, когда используется WPF, в отличие от любой другой

технологии UI, такой как Windows Forms или Web Forms. С другой стороны, хотя нет никакого способа автоматически обнаружить, используется ли данный объект WPF, есть возможность переключения конфигурации, которую разработчик бизнес-логики может установить так, чтобы указать, как этот метод должен себя вести.

Настройка параметра PropertyChangedMode

Параметр `Csla.ApplicationContext.PropertyChangedMode` может быть настроен при помощи файла конфигурации приложения или в коде.

В коде разработчик UI обычно устанавливает значение, выполняя один раз при запуске приложения следующую строку кода.

```
Csla.ApplicationContext.PropertyChangedMode =
   Csla.ApplicationContext.PropertyChangedModes.Xaml;
```

Это должно быть сделано только в приложении WPF, поскольку значение по умолчанию, `Windows`, является правильным выбором для любого приложения не WPF.

Значение может быть также установлено в файле приложения `app.config`, при добавлении следующего элемента в элемент `<appSettings>`.

```
<add key=»CslaPropertyChangedMode» value=»Xaml» />
```

Так или иначе, результат заключается в том, что события `PropertyChanged` передаются, как требуется WPF, а не как нужно для Windows Forms или Web Forms.

Передача событий для WPF

Когда событие `PropertyChanged` перехватывается механизмом связывания данных WPF, в UI обновляются только те элементы управления, которые привязаны к *данному конкретному свойству*. Это, конечно, имеет смысл, но немного сложно.

При изменении свойства происходит запуск проверки правил и бизнес-правил, связанных с этим свойством (см. подробности в главе 11, “Бизнес-правила и правила проверки”). Это приводит также к запуску проверки правил, связанных с *зависимыми свойствами*. Это означает, что изменение одного свойства может запустить проверку правил для нескольких свойств.

Когда проверка правила терпит неудачу, UI отобразит пользователю нечто, означающее недопустимость значения. Механизм связывания данных WPF узнает о необходимости изменить представление элемента управления, потому что он перехватывает событие `PropertyChanged` для свойства, к которому привязан элемент управления.

Если для измененного свойства передано только событие `PropertyChanged`, то любые нарушения правил проверки для *зависимых свойств* будут проигнорированы механизмом связывания данных и не будут видимы пользователю.

Обратите внимание на то, что вызов метода `CheckRules()` возвратит массив, содержащий имена всех свойств, для которых выполняются бизнес-правила или правила проверки. Это позволяет методу `PropertyHasChanged()` передавать события `PropertyChanged` для всех этих свойств, а не только для свойства, которое фактически изменено. В результате нарушенные правила будут отражены в UI, даже для тех свойств, которые фактически не изменены.

Передача событий для Windows Forms

В Windows Forms связывание данных работает по-другому. Когда событие `PropertyChanged` перехватывается механизмом связывания данных Windows Forms, в UI обновляются все элементы управления, связанные с этим бизнес-объектом. Любое одиночное событие `PropertyChanged` обновит все элементы управления в UI. Это значит, что чем меньше будет событий `PropertyChanged`, тем лучше, поскольку каждое приводит к обновлению UI.

Для других технологий, таких как Web Forms, событие PropertyChanged не используется механизмом связывания данных вообще. Но все же важно передавать это событие, поскольку специальный код UI зачастую прослушивает это событие, чтобы узнать об изменении этого объекта. Поскольку большая часть этого специального кода UI была написана до WPF, он предполагает благоприятное поведение для Windows Forms, а не для WPF.

Таким образом, стандартное поведение подразумевает передачу только одного события PropertyChanged для свойства, которое фактически изменено. Это справедливо даже в случае, когда в результате изменения свои проверки выполняет несколько свойств.

Финальный результат заключается в том, что проверка правил свойств осуществляется при установке свойства IsDirty в состояние true и передаче соответствующего события PropertyChanged.

Свойство IsDirty

Вы уже видели, как каждый отдельный бизнес-объект управляется со своим свойством IsSelfDirty. Когда объект является родительским, все немного усложняется. Это связано с тем, что родительский объект считается измененным, если изменились его поля или любой из его дочерних объектов.

Это важно потому, что свойство IsDirty используется порталом данных для оптимизации модификации объектов в базе данных. Если объект не был изменен, нет никакого смысла модифицировать базу данных значениями, которые в ней уже есть. Но давайте рассмотрим случай, когда родительский объект SalesOrder содержит список дочерних объектов LineItem. Даже если пользователи не изменяют сам объект SalesOrder, они изменяют объекты LineItem, граф объекта которого, в целом, следует полагать измененным. Когда код UI вызывает метод _salesOrder.Save(), вполне разумно ожидать, что измененный объект LineItem (дочерний объекта SalesOrder) будет сохранен.

Свойство IsDirty облегчает коду портала данных определение того, был ли граф объекта изменен, поскольку он объединяет значение свойства IsSelfDirty объекта со значениями свойств IsDirty всех дочерних объектов.

```
[Browsable(false)]
public virtual bool IsDirty
{
   get { return IsSelfDirty || (_fieldManager != null &&
                                          FieldManager.IsDirty()); }
}
```

Как обсуждалось в главе 7, "Объявление свойств", класс FieldManager используется для определения того, были ли изменены дочерние объекты данного объекта.

На заметку. Если вы не будете использовать управляемые вспомогательные поля для хранения ссылок дочерних объектов, то переопределите этот метод так, чтобы проверять состояние свойства IsDirty ваших дочерних объектов. Я рекомендую всегда использовать для дочерних ссылок управляемые вспомогательные поля, таким образом вам не придется волноваться об этой проблеме.

Хотя свойство IsSelfDirty предоставляет состояние одного объекта, свойство IsDirty зачастую полезнее, поскольку оно предоставляет состояние не только объекта, но и всех его дочерних объектов.

Свойство IsSelfValid

Объект считается допустимым, если в настоящее время у него нет нарушенных правил. Управление бизнес-правилами осуществляет пространство имен `Csla.Validation`, описанное в главе 11, "Бизнес-правила и правила проверки". Свойство `IsSelfValid` просто предоставляет флаг, указывающий, нарушает ли объект в настоящее время правила.

Есть также свойство `IsValid`, указывающее, допустим ли текущий объект *и все его дочерние объекты*.

Вот код свойства `IsSelfValid`, возвращающий значение только для этого объекта, не считая его дочерних объектов.

```
[Browsable(false)]
public virtual bool IsSelfValid
{
   get { return ValidationRules.IsValid; }
}
```

Подобно свойству `IsSelfDirty`, это свойство отмечено атрибутом `Browsable`, поэтому по умолчанию механизм связывания данных игнорирует это свойство.

Нет никаких методов, позволяющих непосредственно контролировать допустимость объекта. Допустимость объекта контролируется подсистемой правил проверки и бизнес-правил, обсуждаемой в главе 11, "Бизнес-правила и правила проверки". Однако этот метод объявлен как `virtual`, поэтому при желании вполне возможно заменить или дополнить способ, которым инфраструктура CSLA .NET обеспечивает концепцию допустимости.

Свойство IsValid

В то время как свойство `IsSelfValid` указывает допустимость определенного объекта, истинная допустимость объекта должна также учитывать допустимость любых дочерних объектов. Даже если сам объект допустим, но содержит недопустимые дочерние объекты, граф объекта в целом следует считать недопустимым.

Свойство `IsValid` комбинирует результат свойства `IsSelfValid` с результатами свойств `IsValid` всех содержащихся в нем дочерних объектов.

```
[Browsable(false)]
public virtual bool IsValid
{
   get
   {
      return IsSelfValid && (_fieldManager == null ||
                             FieldManager.IsValid());
   }
}
```

Для определения недопустимости любых дочерних объектов здесь также используется класс `FieldManager`. В результате, бизнес-объект считается допустимым только тогда, когда он сам и все его дочерние объекты допустимы.

Свойство IsSavable

Сохранять объект в базе данных следует только тогда, когда он допустим, его данные изменились, нет никаких невыполненных асинхронно правил проверки и текущий пользователь имеет право модифицировать объект.

Свойство `IsValid` указывает, допустим ли объект, а свойство `IsValidating` — есть ли какие-либо невыполненные асинхронно правила проверки. Свойство `IsDirty` указывает, изменились ли данные объекта. Подсистема правил авторизации обсуждается в главе 12, "Аутентификация и авторизация".

Свойство `IsSavable` — это простое вспомогательное свойство, объединяющее эти концепции в единое свойство.

```
[Browsable(false)]
public virtual bool IsSavable
{
   get
   {
      bool auth;
      if (IsDeleted)
         auth = Csla.Security.AuthorizationRules.CanDeleteObject(
                                                          this.GetType());
      else if (IsNew)
         auth = Csla.Security.AuthorizationRules.CanCreateObject(
                                                          this.GetType());
      else
         auth = Csla.Security.AuthorizationRules.CanEditObject
                                                        (this.GetType());
      return (auth && IsDirty && IsValid && !ValidationRules.
IsValidating);
   }
}
```

Код авторизации интересен тем, что, принимая решение о типе авторизации, он полагается на состояние объекта. Например, если свойство `IsDeleted` возвращает значение `true`, объект предназначен для удаления, и, таким образом, осуществляется проверка на право удалять.

С учетом наличия у пользователя прав, код удостоверяется, что объект был изменен, допустим и что нет никакого невыполненного асинхронно правила проверки.

Главная задача этого свойства в том, чтобы позволить разработчику UI сделать доступной или недоступной кнопку Save (Сохранить) (или подобный элемент UI), обеспечивая сохранение объекта, только если он может быть сохранен. Например, оно используется элементом управления `CslaDataProvider` для автоматического разрешения и запрещения элементов управления в WPF, как обсуждается в главе 19, "Пользовательский интерфейс Windows Presentation Foundation".

Свойство `IsDeleted`

Инфраструктура CSLA .NET поддерживает как отложенное, так и немедленное удаление объекта. При *немедленном* подходе данные объекта немедленно удаляются из базы данных, без предварительной загрузки объекта в память. Для этого нужно заранее знать значение (значения) первичного ключа объекта, как обсуждается в главах 4, "Объектные стереотипы CSLA .NET", и 5, "Шаблоны объектов CSLA .NET".

При *отложенном* подходе объект должен быть загружен в память. Теперь пользователь может просматривать данные объекта и манипулировать ими, а затем принять решение об удалении, отметив объект как удаленный. Объект не удаляется немедленно, это происходит лишь при его сохранении в базе данных. Таким образом, вместо вставки или модификации данных объекта происходит их удаление из базы данных.

Данный подход особенно полезен для дочерних объектов коллекции. В этом случае пользователь может добавлять и модифицировать одни дочерние объекты и в то же время удалять другие. Все операции вставки, модификации и удаления происходят в пакете, при сохранении коллекции в базе данных.

Информация о том, отмечен ли объект для удаления, отслеживается полем `_isDeleted` и предоставляется свойством `IsDeleted`.

```
private bool _isDeleted;

[Browsable(false)]
public bool IsDeleted
{
   get { return _isDeleted; }
}
```

Подобно другим свойствам состояния, это не может быть связанным элементом данных.

Метод MarkDeleted

Как и у свойства `IsSelfDirty`, здесь есть защищенный метод, позволяющий, при необходимости, отметить объект для удаления.

```
protected void MarkDeleted()
{
   _isDeleted = true;
   MarkDirty();
}
```

Безусловно, пометка объекта для удаления является еще одним способом изменения его данных. Таким образом, вызов метода `MarkDirty()` означает, что состояние объекта было изменено.

Методы Delete и DeleteChild

Вызов метода `MarkDeleted()` осуществляется из методов `Delete()` и `DeleteChild()`. Метод `Delete()` используется для пометки родительского объекта при отложенном удалении, в то время как вызов метода `DeleteChild()` родительским объектом (таким, как коллекция) отмечает для отложенного удаления дочерний объект.

```
public void Delete()
{
   if (this.IsChild)
      throw new NotSupportedException(Resources.ChildDeleteException);

   MarkDeleted();
}

internal void DeleteChild()
{
   if (!this.IsChild)
      throw new NotSupportedException(Resources.NoDeleteRootException);

   MarkDeleted();
}
```

Оба метода делают то же самое: вызывают метод `MarkDelete()`. Но метод `Delete()` имеет область видимости `public` и может быть вызван, только если объект *не дочерний* (эта тема обсуждается в главе 9, "Отношения между родительскими и дочерними объектами"). Метод `DeleteChild()`, наоборот, может быть вызван, только если объект *дочерний*. Поскольку он предназначен для использования классом `BusinessListBase`, он имеет область видимости `internal`.

На настоящий момент у вас должно быть хорошее понимание различных значений состояния объекта, используемых классами `BusinessBase` и `BusinessListBase`.

Заключение

В этой главе я продолжил обсуждение реализации инфраструктуры CSLA .NET. Здесь рассматривались значения состояния объекта, поддерживаемые редактируемыми объектами и позволяющие остальной части инфраструктуры CSLA .NET, бизнес-коду и коду UI взаимодействовать с вашими объектами стандартизированным способом.

Все редактируемые объекты поддерживают следующие значения состояния.

- `IsSelfDirty`.
- `IsDirty`.
- `IsSelfValid`.
- `IsValid`.
- `IsNew`.
- `IsSavable`.
- `IsDeleted`.

Некоторые из этих значений важны для отношений между родительскими и дочерними объектами, обсуждаемыми в главе 9, "Отношения между родительскими и дочерними объектами". Другие полагаются на проверку правильности и авторизацию, как обсуждается в главах 11, "Бизнес-правила и правила проверки", и 12, "Аутентификация и авторизация". Но большинство из них используется порталом данных, как обсуждается в главе 15, "Персистентность и портал данных".