**В первый раз в первый класс или Эффективное использование классов в .NET**

#### Михаил Фленов ([www.flenov.info](http://www.flenov.info))

Когда нужно изучить какой-то язык программирования, то мы покупаем книгу. В книгах нас учат программировать и рассказывают основы языка, но это же далеко не все, есть еще среда разработки, работа с которой – это отдельная, и далеко не простая история. Давайте рассмотрим некоторые секреты Visual Studio и некоторые секреты классов, чтобы убить сразу несколько зайцев и рассмотреть то, что в книгах очень часто опускают.

**Постановка задачи**

Компонент PropertyGrid позволяет отображать свойства класса в виде сетки, как это делает окно Properties в самой среде разработки. Чтобы компонент отобразил свойства нужного вам класса, просто присвойте этот класс свойству SelectedObject. Например, следующая строка заставит отобразить свойства кнопки Button1:

propertyGrid.SelectedObject = Button1;

Давайте представим, что нам нужно использовать PropertyGrid для отображения нужного нам набора значений. Как это сделать? Да очень просто – можно все эти свойства объединить в отдельный класс и использовать его, как любой другой стандартный класс из .NET Framework. Только чтобы свойства выглядели красиво, их нужно правильно оформить при описании класса.

Давайте напишем пример, в котором на главном окне будет только компонент PropertyGrid, где должны отображаться свойства программы потенциального приложений. Все свойства будут объединены в отдельном классе PropertiesClass, это очень удобно, красиво и будет соответствовать всем принципам ООП. Давайте рассмотрим, как правильно и красиво оформить класс, возможно, вы узнаете что-то новое из жизни .NET.

**Создание полей**

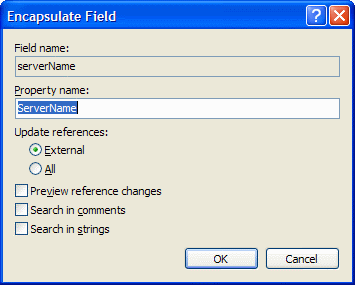
Чем отличается поле (в некоторых источниках можно встретить понятие свойство) от простых переменных? В принципе, это та же переменная, просто она закрыта от внешнего воздействия. Чтобы внешние объекты могли изменять значение поля, нужно создать пару Ментов set и get.

Самый простой способ создать поле - объявить переменную и воспользоваться мастером рефакторинга – Encapsulate Field. Допустим, что нам нужно свойство serverName, для хранения имени сервера. Объявите переменную:

private string serverName;

Имена переменных в .NET принято именовать с маленькой буквы, а вот поля должны иметь то же самое имя, но начинаться с большой буквы. К тому же, переменная должна быть закрыта (private), открытыми (public) будут методы.

Выделите эту строку и выберите меню Refactor | Encapsulate Field.



*Мастер превращения переменной в поле*

В поле Property Name нужно указать имя будущего поля. Как мы уже определились, общепринято, чтобы поля именовались с большой буквы и именно это предлагает нам мастер. Чуть ниже можно выбрать, какие ссылки должны быть обновлены – внешние, или все, которые найдет среда разработки. Если вы только создаете поле, и переменная нигде не использовалась, то не имеет значение, что вы выберете.

В окне мастера есть еще три флага, которые вы можете установить:

- Preview reference changes – его можно установить, чтобы мастер показал вам те места, где он собирается произвести изменения. Если вы не доверяете мастеру, то можно установить этот флаг. На моей практике мастер работает идеально, поэтому я предварительно ничего не просматриваю.

- Search in comment – искать использование ссылок на переменную в комментариях и изменить их на имя поля. Комментарии безопасны и на код не влияют, поэтому использование флага влияет только на эстетичность.

- Search in string – искать использование ссылок на переменную в строках и изменить их на имя поля. Вот это уже опасно и лучше не устанавливать этот флаг, а если устанавливаете, то лучше просматривать изменения. Если у вас где-то есть текстовое сообщение «This is error», и есть переменная error, которую вы превращаете в поле Ugas, то поиск в строках изменит сообщение на «This is Ugas».

В результате работы мастера, строка объявления переменной должна превратиться в следующий код:

private string serverName;

public string ServerName

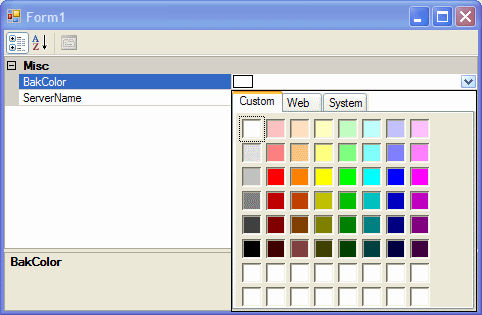
{

get { return serverName; }

set { serverName = value; }

}

Если запустить приложение и посмотреть на сетку свойств, то вы увидите, что наше поле там появилось и его можно изменять. При этом значения полей в классе будут изменяться автоматически в момент их изменения в сетке свойств PropertyGrid. Если поле имеет тип Color, то при его редактировании в сетке будет появляться стандартный выпадающий список цветов.



*Свойство типа Color и выпадающее окно для выбора стандартных значений*

**Атрибуты**

Обратите внимание, что все поля в сетке свойств попадают в раздел Misc. К тому же, ни одного из них нет подсказок. Как сделать и то и другое? Очень просто, нужно использовать атрибуты, но для этого нужно еще подключить пространство имен System.ComponentModel.

Атрибуты устанавливаются в квадратных скобках прямо перед описанием поля. Именно поля, а не переменной или чего-то другого. В этих квадратных скобках атрибут устанавливается в виде:

ИмяАтрибута("Значение")

Категория свойства задается с помощью атрибута Category. Например, следующий код помещает свойство ServerName в раздел «Подключение»:

private string serverName;

[Category("Подключение")]

public string ServerName

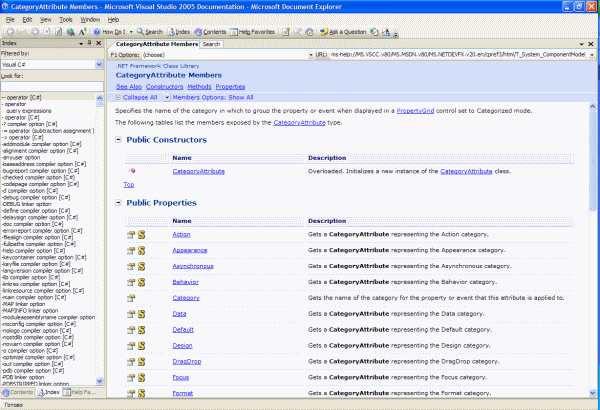
{

get { return serverName; }

set { serverName = value; }

}

А что если установить курсор на имени атрибута Category и вызвать контекстную подсказку? Оказывается, что это на самом деле объект, а не какой-то оператор языка. В объектно-ориентированном языке все должно быть объектами и в данном случае все правильно. Когда мы объявляем атрибут, то на самом деле создаем экземпляр класса. За атрибут Category отвечает класс CategoryAttribute.



*Помощь по атрибуту Category*

Давайте кратко пробежимся, по некоторым, наиболее интересным атрибутам:

- Description – через этот атрибут можно задать описание. В сетке PropertyGrid при выделении поля это описание будет появляться внизу компонента;

- DisplayName – отображаемое имя. Имена полей должны быть в коде на английском и не могут содержать пробелов. А что же делать нам русским, у которых пользователи не очень одаренные (программисты в крупных и даже маленьких организациях нас поймут) и пугаются английских слов? В таких случаях, вы можете указать дружественное имя с помощью данного атрибута. В коде доступ к полю будет происходить через английское имя, а в сетке PropertyGrid будет русское имя;

- DefaultValue – значение по умолчанию. Это всего лишь атрибут и на реальное значение он никак не влияет, но иногда может оказаться очень полезным.

Существуют и другие атрибуты, и в их поиске всегда поможет MSDN, но мне пока хватало рассмотренных четырех.

**Программирование атрибутов**

Мы рассмотрели атрибуты и увидели, что они удобны, если использовать компонент PropertyGrid. Но ведь этот компонент нужен далеко не всегда, а вот атрибутам применение можно найти и в других местах, например, для локализации или для сохранения свойств объекта в файле.

В листинге 1 показан пример, как можно получить доступ к атрибутам. Я постарался написать код максимально универсально, чтобы вы могли использовать его для различных нужд с минимальными изменениями. Главное, что показывает код – поиск нужного атрибута, а дальше уже все зависит от вашей техники.

Листинг 1. Код обработки атрибутов всех свойств

// создаем экземпляр класса PropertiesClass

PropertiesClass tempClass = new PropertiesClass();

// получаем список свойств (полей) класса

PropertyInfo[] pi = tempClass.GetType().GetProperties();

// цикл перебора всех полей

for (int i=0; i<pi.Length; i++)

{

// цикл просмотра атрибутов внетри текущего поля

foreach (Attribute attr in Attribute.GetCustomAttributes(pi[i]))

{

// если текущий атрибут – категория, то выводим его

if (attr.GetType() == typeof(CategoryAttribute))

MessageBox.Show(pi[i].Name+" находиться в категории "+

((CategoryAttribute)attr).Category.ToString());

}

}

В самом начале создается экземпляр класса PropertiesClass, свойства которого мы будем просматривать в поисках атрибутов. В данном примере, мы просматриваем все свойства, поэтому получаем список всех свойств и перебираем его. У свойств есть метод GetCustomAttributes, с помощью которого можно получить список всех атрибутов.

Перебирая атрибуты мы в цикле проверяем тип текущего, и если это категория, то выводим значение, которое можно увидеть в ((CategoryAttribute)attr).Category.

**Метаданные**

Метод GetType(), который мы использовали, для получения атрибутов присутствует у всех типов данных и позволяет получить метаданные этого типа. Например, в случае с объектом (а ведь в .NET все объекты), мы можем узнать, практически все, что только можно, нужно и может пригодиться. Кое-что мы уже успели увидеть, но это только капля в море из доступных метаданных.

В листинге 2 мы приготовили небольшой, но очень интересный код, который позволяет сохранить в XML файле значения всех полей. Обратите внимание, что значение цвета обрабатывается индивидуально, а именно, оно приводиться к числу RGB и уже это число сохраняется. Это логично, ведь цвет – это сложный объект и если его сохранять простым приведением к строке ToString(), то загрузить результат в будущем будет проблематично.

Листинг 2. Сохранение значений всех полей класса в XML

// для записи будет использоваться класс XmlWriter

XmlWriter xmlOut = XmlWriter.Create(filename);

xmlOut.WriteStartElement("ИмяРаздела");

// получаем список всех полей текущего класса

PropertyInfo[] pi = this.GetType().GetProperties();

// цикл сохранения всех полей

for (int i = 0; i < pi.Length; i++)

{

//цвет сохраняем отдельно в виде числа

if (pi[i].PropertyType == typeof(Color))

{

Color c = (Color)(pi[i].GetValue(this, null));

xmlOut.WriteElementString(pi[i].Name, c.ToArgb().ToString());

}

else

xmlOut.WriteElementString(pi[i].Name, pi[i].GetValue(this, null).ToString());

}

xmlOut.WriteEndElement();

xmlOut.Flush();

xmlOut.Close();

В листинге 3 показан код, который загружает значения свойств из XML фала и устанавливает их текущему объекту. Код написан максимально универсально и вы можете использовать его во своих примерах, без каких либо изменений. Единственное, что нужно учесть, что вы обрабатываете все, используемые в классе типы данных.

Листинг 3. Загрузка значений полей из файла

// для чтения будет использоваться класс XmlReader

XmlReader xmlIn = XmlReader.Create(filename);

xmlIn.MoveToContent();

string lastElementName = "";

// цикл чтения файла

while (xmlIn.Read())

{

switch (xmlIn.NodeType)

{

// если найден новый тег

case XmlNodeType.Element:

lastElementName = xmlIn.Name;

break;

// если найден конец текущего тега

case XmlNodeType.EndElement:

if (xmlIn.Name == "MenuProperties")

return;

break;

// получено значение тега

case XmlNodeType.Text:

PropertyInfo pi = this.GetType().GetProperty(lastElementName);

// приводим значение тега к типу данных поля

if (pi.PropertyType == typeof(string))

pi.SetValue(this, xmlIn.Value, null);

else if (pi.PropertyType == typeof(int))

pi.SetValue(this, int.Parse(xmlIn.Value), null);

else if (pi.PropertyType == typeof(bool))

pi.SetValue(this, bool.Parse(xmlIn.Value), null);

else if (pi.PropertyType == typeof(Color))

pi.SetValue(this, Color.FromArgb(int.Parse(xmlIn.Value)), null);

break;

}

}

xmlIn.Close();

**Совместный доступ**

Работая над большим проектом, может возникнуть необходимость одновременного изменения одного и того же файла сразу несколькими сотрудниками. Как поступить в этом случае? До сих пор эту проблему решали использованием утилит контроля версий, например, Microsoft Source Safe. Но подобные утилиты действуют по принципу, когда один файл может редактировать только один человек. Значит, кто первый забрал файл для редактирования, тот и бог, а остальные будут ждать освобождения.

Дабы не приходилось долго ждать, рекомендуется создавать максимально небольшие классы и каждый из них располагать в отдельном файле. В этом случае, программисты в команде будут блокировать на редактирование небольшие файлы и вероятность одновременного доступа к одному и тому же блоку кода уменьшается.

В книге Макконелла «Совершенный код» есть рекомендация создавать классы, которые будут содержать 7, максимум 10 методов. В этом есть здравый смысл, потому что большие классы сложнее воспринимать и анализировать. Но если один класс будет реализовывать слишком маленькую часть общей задачи, то придется создавать много вложений и глубокие наследования, а это не даст преимуществ с точки зрения производительности и читабельности кода.

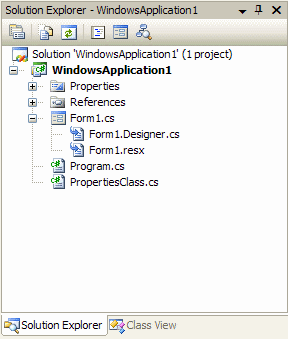
**Разделяй и властвуй**

В Visual Studio и .NET придумали очень элегантное решение совместного редактирования - разбиение большого класса на несколько маленьких файлов, т.е. реализация одного класса может быть распределена на несколько файлов. Эта особенность уже эксплуатируется средой разработки Visual Studio в визуальных формах.

Создайте приложение с визуальным окном. Посмотрите, из чего состоит визуальная форма? А ведь она состоит из двух файлов, например, если форма имеет имя Form1, то ее реализация будет в файлах Form1.cs и Form1.Designer.cs. В первом находятся методы и обработчики событий, а во втором среда разработки помещает весь код, который отвечает за создания визуальной формы, которую вы создаете в дизайнере.

Как реализуется разделение на файлы? В каждом из этих файлов перед объявлением класса стоит ключевое слово (модификатор) partial:

partial class Form1



*Визуальные формы в Visual Studio 2005 состоят из двух файлов*

Теперь, во время сборки проекта, компилятор будет искать все классы, в которых используется одно и то же имя и при этом в объявлении есть слово partial. Если такого слова не будет, то возникнут проблемы, особенно, если два класса с одним именем находятся в одном пространстве имен. Проблемы не слишком большие, но проект не будет собран, а результатом будет сообщение:

*Missing partial modifier on declaration of type 'WindowsApplication1.Form1'; another partial declaration of this type exists*

*Отсутствует модификатор partial в объявлении типа 'WindowsApplication1.Form1'; существует другое разделяемое объявление данного класса*

Теперь, один программист может редактировать файл Form1.Designer.cs, создавая визуальную форму, а другой программист может создавать обработчики событий и их реализации в файле Form1.cs. Можно пойти дальше, и создать еще один файл, в котором будут находиться объявления переменных, и методы, не являющиеся обработчиками событий. Чтобы сделать это, добавьте в проект файл, и назовите его, например, Form1.Extended.cs. Среда разработки создаст этот файл и в нем будет заготовка класса Form1. Добавьте к этой заготовке модификатор partial, и этот класс станет частью уже существующей формы. Вы сможете писать код внутри класса этого файла, и обращаться к методам и переменным из одноименного класса из файлов Form1.cs и Form1.Designer.cs, как будто это одно целое пространство имен.

**Регионы**

В Visual Studio есть еще одна интересная и удобная возможность – объединять блок кода в регион и сворачивать (прятать) этот регион внутри редактора кода. Функции, комментарии уже давно можно прятать. Если они уже отлажены, то код можно свернуть, дабы он не мозолил глаза. Точно так же можно прятать и по несколько функций, если объединить их в один блок.

Регион начинается со слова #region, после которого идет имя региона, а заканчивается #endregion.

#region ИмяРегиона

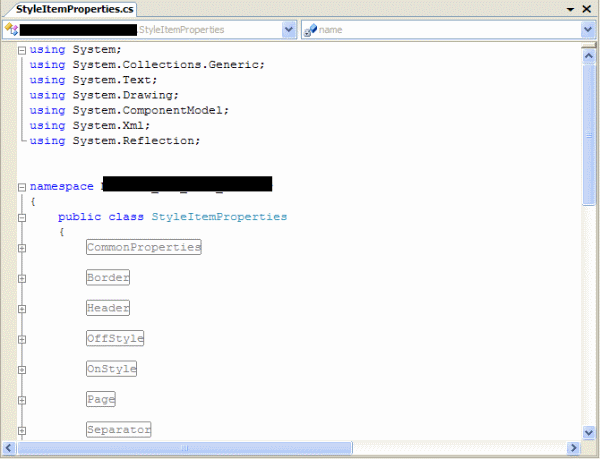
функция 1

функция 2

функция 3

#endregion ИмяРегиона

В своих проектах, я всегда объединяю методы и переменные, схожие по значению в отдельные регионы. Теперь, свернув все, исходный файл в редакторе кода становиться максимально компактным, а поиск нужного для редактирования метода отнимает минимум времени. Разворачивать код – быстрее, чем пролистывать громадный файл.



*Серые слова в прямоугольниках – это имена свернутых регионов*

**Итого**

За долгую карьеру программиста мне приходилось работать с разными языками программирования и с различными средами разработки. Первая версия .NET вызвала у меня отвращения, потому что являлась грубой подделкой на Java. Сейчас, это уже полноценный язык и в сочетании с Visual Studio .NET показывает настоящую мощь. Не могу сказать, что по возможностям и мощи .NET догнал своего основного конкурента – Java, но это уже не сырая недоработка, а вполне рабочее решение.