Библиотека Windows Forms, демонстрация

Юрий Литвинов
yurii.litvinov@gmail.com

6

Запускаем студию, New Project, Templates -> Visual C# -> Windows -> Windows Forms Application. Создаётся новый проект, открывается редактор форм с пустой формой. Показываем структуру проекта: под файлом Form1.cs скрываются ещё три файла — Form1.Designer.cs, Form1.resx, и настоящий Form1.cs. Form1.Designer.cs содержит сгенерированный дизайнером форм код формы, и его нельзя редактировать вручную. Каждый раз, когда вы меняете что-то на форме в редакторе, этот класс перегенерится. Form1.resx содержит ресурсы, используемые на форме, такие как, например, строки, показываемые пользователю. По-хорошему, в коде не должно быть захардкоженых строковых констант и прочей информации, которая может быть зависимой от локали, все такие штуки выносятся в ресурсные файлы, и тогда перевод программы на другой язык сводится просто к замене ресурсов. Мы пока не будем этим заморачиваться. Form1.cs — это то место, куда, собственно, надо писать код.

Обратите внимание, файлы Form1.Designer.cs и Form1.cs содержат объявление одного и того же класса Form1, объявленного как partial. Собственно, ключевое слово partial говорит компилятору, что в файле содержится только часть объявления класса, и следует поискать ещё файлы, где класс доопределён. partial-ами можно делать любые классы, не только относящиеся к пользовательскому интерфейсу, должны выполняться следующие требования — все куски класса должны называться одинаково, быть помеченными partial, быть в одном неймспейсе, иметь неконфликтующие модификаторы видимости. Обычно partial-ы, тем не менее, используются только для сгенерённого кода — тут они позволяют не заботиться о сохранении пользовательских правок, как в Яве, а просто перегенерить весь файл, когда нужно. Все пользовательские правки будут делаться в другом файле, хотя там и определяется тот же самый класс. Для рукописных классов это неудобно, потому что требуется прыгать по разным файлам, чтобы редактировать класс (плохо с точки зрения, собственно, инкапсуляции, у нас код, делающий одно дело, разбросан по разным местам). После компиляции класс получается один, точно так же, как если бы он был описан в одном файле.

Посмотрим на сгенерённый редактором код Form1.Designer.cs. Сначала идёт объявление поля components, штуки, которая будет содержать в себе всякие компоненты на форме. Дальше перегруженный метод Dispose, который чем-то напоминает деструктор в C++ и служит для освобождения ресурсов, выделенных объекту. Здесь он просто вызывает Dispose у всех компонентов. Дальше идёт директива #region — это просто штука для удобства отображения исходников, код, заключённый в #region/#endregion, можно свернуть в редакторе студии (и по умолчанию он будет показываться свёрнутым). Вместо кода будет

показываться строка, которая написана после #region. В обычном коде так тоже можно писать, и иногда (например, в процессе рефакторинга, когда вам надо быстро рассортировать кучу методов по файлу) бывает полезно. Использовать его в уже готовом коде не рекомендуют — он, как ни странно, ухудшает читаемость, а не улучшает её, скрывая, быть может, интересные куски кода, и заставляя кликать на плюсики. Если у вас такой большой файл, что хочется сделать регионы, имеет смысл разбить его на более мелкие классы и вынести в отдельные файлы.

Дальше идёт метод InitializeComponent, самый интересный на самом деле для нас, поскольку содержит в себе код создания и инициализации всех контролов на форме. Сюда можно смотреть, если требуется сделать что-то вручную в рукописном коде, но вы не знаете как. Менять что-либо здесь бессмысленно. Пока у нас форма пустая, тут ничего особо интересного. this.SuspendLayout(); говорит, что пока не надо пытаться автоматически раскладывать элементы по форме (поскольку они сейчас будут создаваться), в конце вызовется this.ResumeLayout(false);, который скажет библиотеке, что вот теперь правила автоматического размещения элементов (лейаут) можно применять. Между этими вызовами находится код, заполняющий проперти объекта-формы. Теперь посмотрим на Form1.cs и увидим, что единственное содержательное, что там написано, это вызов InitializeComponent в конструкторе.

Дальше посмотрим на Program.cs. Там используется класс System.Windows.Forms.Application, который позволяет задать глобальные настройки исполнения приложения на WinForms, и содержит тот самый цикл обработки событий, про который шла речь на прошлой паре. EnableVisualStyles даёт возможность использовать стиль операционной системы для отображения окна и контролов на нём, впрочем, это работает только на XP. SetCompatibleTextRenderingDefault задаёт параметры отображения текста на некоторых контролах, и опять-таки, для современных приложений не имеет значения и его лучше не трогать. Метод Run запускает цикл обработки событий и показывает ту форму, которую ему передали.

Теперь собственно как добавлять новые контролы. Кстати, касательно терминологии: в продуктах Майкрософт элементы пользовательского интерфейса принято называть контролами, в некоторых других платформах (например, Qt) — виджетами. Это одно и то же. Так вот, выбираем пункт меню View -> Toolbox, открывается Toolbox (если он ещё не был открыт), там открываем палитру Common Controls, вытягиваем на форму кнопочку. Сохраняемся, смотрим в Form1.Designer.cs, видим там код создания и инициализации кнопочки. Собственно, тот же самый код можно написать и вручную, прямо в рукописном коде, никакой магии тут нет.

Вернёмся обратно в дизайнер форм. Открываем окно свойств, если оно ещё не открыто. Иначе View -> Properties Window. Выделяем кнопочку. Самые интересные свойства тут Text (то, что пишется на кнопочке), Name (имя переменной, по которой можно получить доступ к кнопочке из кода). Ещё полезны свойства Visible (показывать/не показывать кнопку на форме), Enabled (кнопочка активна/неактивна), BackColor (цвет кнопки), ForeColor (цвет текста на кнопке), Tag (просто любые данные, которые можно хранить в кнопке, это просто поле типа object, и бывает очень полезно, если мы создаём кнопки динамически, чтобы, например, их различать), TabIndex (задаёт порядок, в котором контролы будут посещены при обходе формы с клавиатуры клавишей Tab), TabStop (будет ли контрол вообще посещён при обходе табом). Ещё есть несколько свойств, относящихся к размещению кнопки на форме и поведению при ресайзе, но про это чуть потом, сначала

как делать так, чтобы при нажатии на кнопку что-нибудь происходило. Для этого применяется отдельная вкладка в окне свойств:

Вообще, сгенерировать обработчик клика на кнопку можно и просто даблкликом на кнопке, но кнопка, как и любой другой контрол, поддерживает ещё кучу всяких событий, каждому из которых тоже можно задать обработчик. Весь список и можно увидеть на вкладке Events. В общем, делаем двойной клик на кнопке, и оказываемся в рукописной части кода класса Form1, в сгенерённом методе-обработчике события клика:

```
private void button1_Click(object sender, EventArgs e)
{

}
```

Сигнатура должна быть знакома по предыдущему занятию. В Form1.Designer.cs появился соответствующий кусок кода:

```
this.button1.Click += new System.EventHandler(this.button1_Click);
```

Опять-таки, что тут делается, понятно по предыдущему занятию — у кнопки есть событие Click, на которое подписывается обработчик `button1_Click`. Теперь всё-таки сделаем, чтобы что-то происходило, добавив в этот обработчик код, меняющий, скажем, заголовок окна:

```
private void button1_Click(object sender, EventArgs e)
{
    this.Text = "Ololo";
}
```

`button1_Click`, хоть и сгенерённое, но всё же плохое имя для контрола, лучше обработчики начинать с On, и следовать общему для всего кода стайлгайду. Просто меняем его имя в Form1.cs на OnButton1Click, жмём Ctrl-. (или кликаем на маленький красный прямоугольник снизу), выбираем нужное действие, он поменяет имя везде, в том числе, и в сгенерённом коде. Точно так же можно и с другими событиями, например, очень легко сделать, чтобы кнопка меняла цвет при наведении на неё мышки. В дизайнере выбираем кнопку, находим её событие MouseEnter во вкладке Events, даблклик по пустому полю с именем метода, пишем обработчик, чтобы получилось как-то так:

```
private void button1_MouseEnter(object sender, EventArgs e)
{
    button1.BackColor = Color.Red;
}
```

Дальше находим событие MouseLeave и делаем для него обработчик таким:

```
private void button1_MouseLeave(object sender, EventArgs e)
{
    button1.BackColor = SystemColors.Control;
}
```

Запускаем приложение, проверяем, что всё работает.

Теперь можно поговорить о взаимном расположении контролов на форме. То, что мы сейчас сделали, никуда не годится — достаточно сжать форму, чтобы понять, почему: кнопка окажется вне отображаемой зоны формы, и по ней будет не кликнуть. Нормальные пользовательские интерфейсы так делать не должны, все элементы управления всегда должны быть доступны, и при этом должны адекватно выглядеть при разных разрешениях экрана. Сама винда вроде как до сих пор имеет окошки системных настроек, которые на нетбуке не лезут на экран, так что даже по кнопке Ok не кликнуть. Для этого используется, во-первых, свойство MinimumSize формы, во-вторых, средства задания положения контролов на форме.

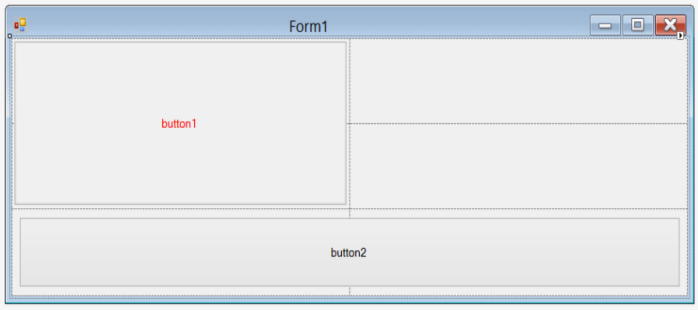
Первое такое средство — свойство Anchor контрола. По умолчанию кнопка привязана к левому и верхнему краям формы, так что при изменении размера формы сохранит положение относительно начала координат. Можно привязать кнопку к другим краям формы, если привязать, скажем, к левому и правому, то левая граница кнопки будет привязана к левому краю, а правая — к правому, так что кнопка будет растягиваться при ресайзе. Если не привязать ни к чему, то кнопка будет стремиться сохранить пропорции расстояний до краёв. Используется всё это в разных ситуациях, например, если у вас есть рабочая область, кнопки слева и кнопки справа, то рабочую область лучше сделать растягивающейся (привязав ко всем краям формы сразу), кнопки слева привязать к левому краю, а кнопки справа — к правому.

Второе полезное свойство — Dock, оно позволяет заставить контрол заполнять либо всего контрола-родителя, либо какую-то его часть (верхнюю, левую и т.д.). С ним можно употреблять свойство Margins, задающее отступ в пикселах от границы контрола до границы контрола-родителя (то есть формирующее такую пустую рамку вокруг контрола). Margin-ы можно задавать по отдельности сверху, снизу, справа и слева, и на самом деле это инструмент, который позволяет точно задать положение контрола относительно родителя так, что его будут уважать средства автоматического размещения контрола на форме (лейауты). Поэтому используется даже чаще, чем кажется, хотя полезен и сам по себе, например, принято, чтобы между контролами всегда был какой-то зазор.

Собственно, последний способ управлять положением контролов на форме — это лейаут. Лейауты можно понимать как алгоритмы размещения контролов. Лейаут спрашивает у контрола, сколько ему места надо обязательно, желательно, максимально, знает свой текущий размер, и из этого вычисляет оптимальные размеры контролов по тому алгоритму, который реализует. Например, FlowLayout просто последовательно выкладывает контролы на форму, не заморачиваясь с размерами и масштабированием. TableLayout представляет форму в виде сетки, где каждый контрол занимает одну или несколько ячеек. При масштабировании ячейки масштабируются пропорционально, и контролы, если им сказано это делать, масштабируются вместе с ними. Лейауты в WinForms используются относительно редко, потому что анкоры — достаточно выразительный и более простой способ делать масштабируемые формы, но в других библиотеках (например, WPF или Qt) лейауты являются одним из главных средств задания положения элементов. В WinForms лейауты могут быть весьма полезны, если мы динамически создаём контролы во время работы программы, тогда лейаут может положить нам контрол куда надо, какой бы размер ни имела форма.

Рассмотрим подробнее TableLayout. Найти его можно на палитре Toolbox, меню Containers. Выкладываем его на форму, сразу делаем Dock Fill, чтобы заполнить всего

родителя (иначе лейаут сам не будет ресайзиться, и толку от него будет немного), задаём ему количество рядов и колонок, и кидаем в одну из получившихся клеток нашу кнопку. У кнопки появляются дополнительные атрибуты Cell, Column, ColumnSpan, Row, RowSpan. ColumnSpan и RowSpan управляют тем, сколько соседних ячеек занимает контрол, так что с их помощью можно делать, например, так:



У самого лейаута тоже есть свойства, наиболее полезные — Rows и Columns, которые позволяют задать количество и относительные размеры строчек и столбцов сетки.