ISSN 2078–5097. Вісн. Львів. ун-ту. Сер. прикл. матем. та інф. 2018. Вип. 26. С. 145–154
Visnyk of the Lviv University. Series Appl. Math. and Informatics. Issue 26. P. 145–154

УДК 004.432.2, 004.422.83

ВИКОРИСТАННЯ МЕХАНІЗМУ ПОДІЙ C# .NET ДЛЯ СТВОРЕННЯ КОМПОНЕНТА WINDOWS FORMS

Сергій Ярошко[1], **Світлана Ярошко**[2]

[1]*Львівський національний університет імені Івана Франка,*
вул. Університетська, 1, Львів, 79000, e-mail: kafprog@lnu.edu.ua
[2]*Національний університет "Львівська політехніка",*
вул. Степана Бандери, 12, Львів, 79013, e-mail: sm.jaroshko@gmail.com

Детально описано процес створення нового компонента користувача мовою C# для платформи Windows Forms .NET, що інкапсулює групу залежних перемикачів – компонента *RadioGroup*. Він спрощує створення, налаштування такої групи та взаємодію з нею. Значну увагу приділено засобам підтримки середовища програмування етапу проектування для нового компонента: категоризації властивостей, підказкам, редакторам властивостей, смарт-тегам тощо. Для реалізації перелічених вимог суттєву роль відіграє використання *подій* – одного з основних механізмів об'єктно-орієнтованого програмування мовою C#. Для створення *RadioGroup* опрацьовано наявні події багатьох компонентів платформи, а також спроектовано та використано події самого компонента і його складових частин.

Ключові слова: подія програмного компонента, компонент користувача Windows Forms, RadioGroup, смарт-тег, дизайнер компонента.

1. ВСТУП

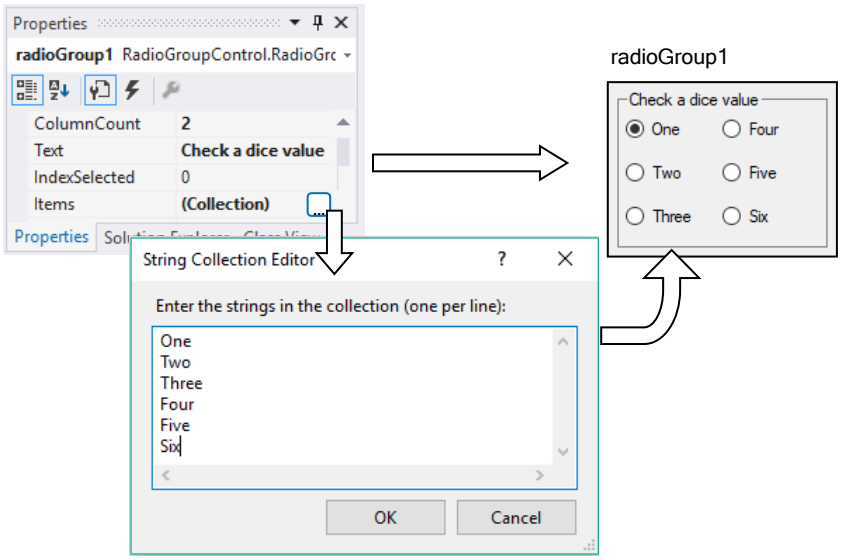
Одним з поширених елементів інтерфейсу користувача Windows-застосунків є група залежних перемикачів або кнопок. Зазвичай так запитують про вибір однієї з декількох альтернатив. Користувач може позначити тільки одну кнопку з групи і так повідомити програму про свій вибір. Для програмування групи залежних перемикачів мовою C# з використанням бібліотеки Microsoft Windows Forms створюють потрібну кількість екземплярів класу *RadioButton* і поміщають їх у деякий контейнер: вікно програми або компонент *Panel* чи *GroupBox*. Щоб з'ясувати, яку з кнопок вибрав користувач, перевіряють властивість *Checked* кожної з них, або задають опрацювання події *CheckedChanged* (чи *Click*) кожної з них. Загалом доводиться виконувати багато рутинної роботи. Її стає в рази більше, якщо в одному вікні потрібно розташувати декілька незалежних наборів альтернатив, текст програми у цьому випадку втрачає ясність і структурованість. Доцільно було б інкапсулювати описані дії у класі, що описує новий візуальний елемент керування.

Раніше широко вживаною була бібліотека Borland Visual Component Library, написана мовою Object Pascal. У її складі є компонент *TRadioGroup*, що "вміє" створювати колекцію кнопок-перемикачів відповідно до заданої користувачем колекції назв, обрамляти її рамкою з підписом, розташовувати кнопки рівномірно по своїй поверхні, і, навіть, у декілька стовпчиків. Компонент також відстежує стан своїх кнопок і повідомляє користувачеві номер вибраної, сигналізує про зміни цього номера. Ми маємо на меті створити схожий компонент для середовища .Net засобами мови програмування C#.

Бібліотека Windows Forms містить декілька класів, наслідуванням від яких можна створювати нові компоненти, а середовище програмування Microsoft Visual Studio надає відповідні шаблони проектів. Процес створення нового елемента керування описаний в літературі, наприклад, у [1, 2], і на перший погляд видається простим. Проте створення компонента, що містить колекцію елементів керування, узгоджену з колекцією рядків-назв несподівано стикається з низкою труднощів. Ми опишемо як їх подолати.

2. ПРОЕКТ КОМПОНЕНТА *RadioGroup*

Щоб створити новий компонент Windows Forms, наслідують від одного з класів бібліотеки: від готового елемента керування, якщо хочуть удосконалити його поведінку, від *Control*, якщо новий елемент має унікальний зовнішній вигляд, від *Component* для створення компонента, який не має зовнішнього вигляду тощо. Ми будуватимемо новий елемент керування з декількох готових компонентів, тому візьмемо за основу клас *UserControl*.



Вигляд компонента *radioGroup1* після редагування його властивостей

Припустимо, що користувач захотів створити групу з шести перемикачів, розташованих у два стовпчики. Тоді група мала б виглядати так, як зображено на рис.

Перелічимо складові частини та властивості класу *RadioGroup*:

- компонент *GroupBox* – зображає рамку навколо групи і назву у розриві рамки, займає всю поверхню елемента керування;
- компонент *TableLayoutPanel* – зручний інструмент для розташування перемикачів у комірках прямокутної таблиці, автоматично масштабує таблицю, якщо змінити розмір елемента керування, займає всю клієнтську частину *GroupBox*;

- колекція *names* – задана користувачем послідовність назв кнопок;
- колекція *buttons* – колекція залежних кнопок, створена елементом керування відповідно до *names*;
- властивість *Text* – надає доступ до відповідної властивості компонента *GroupBox*;
- властивість *Items* – надає доступ до колекції імен;
- властивість *ColumnCount* – дає змогу встановлювати кількість стовпчиків групи;
- властивість *IndexSelected* – повідомляє, задає номер позначеного перемикача.

Події та додаткові властивості опишемо згодом.

Для створення нового елемента керування використаємо шаблон проекту Windows Forms Control Library. На етапі проектування компонента ми зможемо виконати не надто багато роботи: налаштувати початковий розмір компонента, додати до нього екземпляри *GroupBox* та *TableLayoutPanel*, налаштувати деякі їхні властивості. Наприклад, доцільно ще на етапі проектування налаштувати колекцію стилів стовпчиків компонента *TableLayoutPanel*. Зрозуміло, що кількість стовпців у групі потрібно обмежити, наприклад, числом 8 – ширшу панель з кнопками буде важко охопити поглядом. Тому на етапі проектування створимо колекцію з восьми пропорційних стилів стовпців *TableLayoutPanel*, а значення властивості *ColumnCount* задамо рівною 1 – тепер видимим буде тільки один стовпець, змінити кількість стовпців можна буде простою зміною значення *ColumnCount*.

Решту роботи зі створення елемента керування *RadioGroup* можна виконати тільки програмно.

```
public partial class RadioGroup: UserControl
{
   private StringCollection names = new StringCollection();
   private List<RadioButton> buttons = new List<RadioButton>();

   public StringCollection Items
   {
       get { return this.names; }
       set { this.names = value; UpdateButtons(); }
   }
   ...
}
```

Тут для зберігання колекції імен використано контейнер *StringCollection*, як у подібних стандартних компонентів, наприклад, *ListBox*. Отже, властивість *Items* можна буде редагувати за допомогою стандартного редактора. Для колекції кнопок підійде будь-який послідовний контейнер змінного розміру, наприклад, *List<RadioButton>*.

3. ВЗАЄМОДІЯ КОЛЕКЦІЇ ІМЕН І КОЛЕКЦІЇ КНОПОК

Щоб узгодити вміст цих колекцій, достатньо виконати відповідний програмний код. Для додавання кнопок:

```
for (int i = buttons.Count; i < names.Count; ++i )
{
    // Створити нову кнопку.
    RadioButton radioButton = new RadioButton();
    // Задати її властивості та обробітник події.
    radioButton.AutoSize = true;
    radioButton.TabStop = true;
    radioButton.Click += button_Click;
    radioButton.Text = names[i];
    // Додати кнопку до колекції.
    buttons.Add(radioButton);
}
```

Для вилучення кнопок:

```
// Обов’язково від’єднати обробітник події перед вилученням кнопки!
for (int i = names.Count; i < buttons.Count; ++i)
    buttons[i].Click -= button_Click;
buttons.RemoveRange(names.Count, buttons.Count - names.Count);
```

Проте таке рішення працюватиме не завжди, адже в класі *StringCollection* визначено низку методів для зміни вмісту колекції. Жоден з них не впливатиме на колекцію кнопок. Як вирішити цю проблему? Об’єкти *names* і *buttons* не “знають” про існування один одного, тому не можуть обмінюватися повідомленнями. Загалом вони і не повинні знати. У сучасних програмах об’єкт сигналізує про зміну свого стану, ініціюючи *подію* (*event*), а залежний від нього об’єкт підписує на подію метод опрацювання, що виконається автоматично, як тільки станеться подія. Застосуємо цей підхід для організації взаємодії двох колекцій. Для цього нам доведеться оголосити підклас *StringCollection*, оголосити в ньому спеціальну подію та перевизначити всі методи, що змінюють вміст колекції.

Для того, щоб оголосити власну подію, потрібно виконати декілька кроків. Найперше оголошують клас для опису інформації про подію – підклас *EventArgs*. Далі оголошують тип делегата події. Зазвичай він має два параметри: перший, *Object*, вказує на джерело події, другий передає інформацію про подію (і має оголошений на попередньому кроці тип). Наступний крок – оголосити саму подію (за допомогою ключового слова *event*) в термінах типу делегата. Нарешті, потрібно визначити метод-диспетчер події. Його завдання – перевірити, чи підписався хтось на подію, і якщо так, то викликати опрацювання події, передавши у виклику джерело і аргументи події. Щоб ініціювати подію в довільному місці програми, викликають метод-диспетчер.

Тепер колекція імен компонента *RadioGroup* матиме новий тип. Наведемо фрагмент з оновленого оголошення класу.

```
public class NameCollection : StringCollection
{
    // Тип делегата події (тут ItemsChangedEventArgs – тип аргумента події).
    public delegate
        void ItemsChangedEventHandler(object sender, ItemsChangedEventArgs e);
    // Подія
    public event ItemsChangedEventHandler ItemsChanged;
    // Диспетчер події
    private void OnItemsChanged(ItemsChangedEventArgs e)
    {
```

```
        if (ItemsChanged != null) ItemsChanged(this, e);
    }
    // Один з методів, що ініціює подію
    public new void Add(string item)
    {
        base.Add(item);
        OnItemsChanged(new ItemsChangedEventArgs(true));
    }
    ...
}
// Оновлена колекція імен
private NameCollection names = new NameCollection();

// Конструктор компонента задає метод опрацювання події зміни колекції імен
public RadioGroup()
{
    InitializeComponent();
    names.ItemsChanged += names_ItemsChanged;
}
```

Тепер, завдяки події, компонент *RadioGroup* дізнаватиметься про кожну зміну колекції імен і викликатиме метод *names_ItemsChanged* для внесення поправок у колекцію кнопок.

4. ПОДІЇ КОМПОНЕНТА *RadioGroup*

У попередньому розділі ми оголосили “внутрішню” подію компонента й організували з її допомогою взаємодію його незалежних частин. Настав час оголосити події, за допомогою яких компонент взаємодіятиме з зовнішніми програмами.

Головною подією групи буде зміна номера позначеного перемикача – подія *IndexChanged*. Вона може трапитись з двох причин: або користувач клацнув мишею на непозначеному перемикачі, або програміст безпосередньо змінив значення властивості *IndexSelected*. Ця подія не потребує спеціальних аргументів, бо вичерпну інформацію про неї можна отримати через властивість *IndexSelected* – достатньо самого настання події. Тому для оголошення використаємо стандартний тип делегата *EventHandler*. Диспетчер події також виглядає прогнозовано.

```
[DefaultEvent("IndexChanged")]
public partial class RadioGroup: UserControl
{
    public event EventHandler IndexChanged;
    private void OnIndexChanged(EventArgs e)
    {
        if (IndexChanged != null) IndexChanged(this, e);
    }
...
```

До класу компонента застосовано атрибут *DefaultEventAttribute*. Він повідомляє середовищу програмування, що *IndexChanged* є подією за замовчуванням, тому її потрібно автоматично вибирати на сторінці властивостей. Варто зазначити, що важко переоцінити роль атрибутів у налаштуванні підтримки компонентів середовищем програмування, проте це питання виходить за межі статті.

Продемонструємо також, коли виникає подія *IndexChanged*: коли клацають на кнопці чи змінюють властивість.

```
// опрацювання клацання на кнопці-перемикачі
private void button_Click(object sender, EventArgs e)
{   // з'ясуємо номер кнопки, на якій клацнули
    int indexClicked = buttons.IndexOf(sender as RadioButton);
    // якщо номер позначеної змінився
    if (indexClicked != indexSelected)
    {   // запам'ятаємо нове значення
        indexSelected = indexClicked;
        // ІНІЦІЮЄМО ПОДІЮ
        this.OnIndexChanged(new EventArgs());
    }
    // транспортує подію миші від вкладеного компонента
    this.OnClick(e);
}
// властивість, що відображає номер позначеної кнопки
public int IndexSelected
{   get { return this.indexSelected; }
    set
    {   if (this.indexSelected != value)
        {   // запам'ятаємо нове значення
            this.indexSelected = value;
            // змінимо стан кнопки
            this.buttons[indexSelected].Checked = true;
            // ІНІЦІЮЄМО ПОДІЮ
            this.OnIndexChanged(new EventArgs());
    }   }
}
```

Дуже поширеними для візуальних елементів керування є події миші: *Click*() та *DoubleClick*(). Всю поверхню групи *RadioGroup* займають вкладені компоненти, тому події миші стаються для них, а не для групи. Щоб повідомити батьківський компонент про клацання, достатньо в методі опрацювання вкладеного компонента викликати диспетчер події батьківського. У наведеному вище прикладі в тілі методу *button_Click* таку роботу виконує виклик *OnClick*(*e*). Подібно подія подвійного клацання на *TableLayoutPanel* транспортується до батьківського компонента: у методі її опрацювання викликаємо *OnDoubleClick*(*e*).

5. ПОДІЇ ДИЗАЙНЕРА

Ми присвятили достатньо уваги властивості *Items*, вона чудово працює на етапі виконання: кожна зміна колекції імен одразу відображається на колекцію кнопок. Проте на етапі проектування компонент *RadioGroup* поводиться по-іншому. Виявляється, дизайнер форми оминає метод *set* властивості та безпосередньо змінює значення поля властивості. Тому на етапі проектування зміна *Items* не призводить до перебудови кнопок. Щоб виправити цю помилку, доведеться докласти додаткових зусиль.

Про кожну зміну компонента на етапі проектування можна довідатися за допомогою сервісу *ComponentChangeService* [2, 3]. Кожного разу, коли під час проектування форми ми змінюємо значення властивості якогось компонента, виникає подія *ComponentChanged*. Аби у відповідний момент оновити колекцію кнопок, потрібно задати опрацювання цієї події. Певним сюрпризом є те, що подія стається для зміни кожної властивості, а не лише *Items*, і не лише потрібного нам компонента. Щоб уникнути зайвих багаторазових перебудов, доведеться розпізнавати тип компонента, для якого сталася подія, та ім'я зміненої властивості.

Доведеться дати відповідь на ще одне запитання: де розташувати відповідний програмний код? Загромаджувати оголошення *RadioGroup* такими технічними подробицями було б помилкою, тому напишемо окремий клас – дизайнер компонента. Він також допоможе нам створити список смарт-тегів, щоб поліпшити підтримку компонента на етапі проектування.

```
public class RadioGroupDesigner : System.Windows.Forms.Design.ControlDesigner
{
    private IComponentChangeService changeService;
    public override void Initialize(IComponent component)
    {   base.Initialize(component);
        // отримаємо посилання на сервіс
        changeService = (IComponentChangeService)
            GetService(typeof(IComponentChangeService));
        if (changeService != null)
        { changeService.ComponentChanged += ComponentChanged; }
    }
    protected override void Dispose(bool disposing)
    {   if (changeService != null)
        {   // обов'язково звільнити ресурс!
            changeService.ComponentChanged -= ComponentChanged; }
        base.Dispose(disposing);
    }
    // МЕТОД ОПРАЦЮВАННЯ ЗМІНИ КОМПОНЕНТА
    private void ComponentChanged(object sender, ComponentChangedEventArgs e)
    {   IComponent comp = (IComponent)e.Component;
        // розпізнавання типу компонента
        if (comp.Site.Component.GetType() == typeof(RadioGroup))
        {   // розпізнавання імені властивості
            if (e.Member.Name == "Items")
            { ((RadioGroup)comp.Site.Component).UpdateButtons(); }
        }
    }
}
```

До класу компонента дизайнер приєднують за допомогою атрибута:

```
[Designer(typeof(RadioGroupDesigner))]
public partial class RadioGroup: UserControl
{   ...
```

Вікно смарт-тегів – зручний інструмент етапу проектування. Воно схоже до контекстного меню, бо містить список вибраних пунктів. Цими пунктами можуть бути команди, редактори властивостей, написи тощо. Тому вікно смарт-тегів діє як міні вікно *Properties* і допомагає швидко налаштувати найважливіші властивості компонента. Для його створення перевизначають властивість *ActionLists* дизайнера так, щоб вона повертала колекцію команд вікна смарт-тегів. Побудову такої колекції інкапсулюють у класі, похідному від *DesignerActionList*.

Розглянемо детальніше, як за допомогою смарт-тега взаємодіяти з властивістю компонента *RadioGroup*, наприклад, з *IndexSelected*. Нащадок *DesignerActionList* огортає своєю властивістю відповідну властивість компонента, причому визначає для неї метод *set* особливо.

```
public class RadioGroupActionList : System.ComponentModel.Design.DesignerActionList
{   private RadioGroup linkedControl;
```

```
...
    public string IndexSelected
    {   get { return linkedControl.IndexSelected; }
        set { (TypeDescriptor.GetProperties(linkedControl)["IndexSelected"]).
                  SetValue(linkedControl, value); }
    } ...
}
```

Бачимо, що прочитати значення огорнутої властивості можна звичайним звертанням до неї, а запис нового значення у неї виконують за допомогою методу *PropertyDescriptor.SetValue*(). Такий спосіб гарантує виникнення у компонента події *PropertyChanged*, яка поінформує всі частини середовища програмування про внесені користувачем зміни.

6. ДОДАТКОВІ ВЛАСТИВОСТІ

Ми вже описали більшість властивостей компонента RadioGroup. Поза увагою залишилися найпростіші. Властивість Text огортає однойменну властивість вкладеного компонента *GroupBox*:

```
public override string Text
{
    get { return theGrpBox.Text; }
    set { theGrpBox.Text = value; }
}
```

Властивість *ColumnCount* цілком нова, тому використовує власне поле для зберігання значення. Його зміна спричиняє перебудову *TableLayoutPanel*.

```
private int columnCount = 1;
public int ColumnCount
{   get { return columnCount; }
    set { if (value < 1) value = 1; else if (value > 8) value = 8;
          if (columnCount != value)
          { columnCount = value; UpdateLayout(); }
        }
}
```

Ще дві властивості надають додаткові можливості керування вмістом групи: *FlowDirection* задає порядок заповнення таблиці кнопками, *Sorted* визначає, чи впорядковувати список імен за алфавітом. Їхнє влаштування схоже до *ColumnCount*, тому наводити їх повністю ми не будемо.

Повний текст компонента *RadioGroup*, дизайнера *RadioGroupDesigner*, класу для створення смарт-тегів *RadioGroupActionList* та тестової аплікації, а також опис інших аспектів створення компонента наведено у [4].

7. ВИСНОВКИ

Механізм подій мови програмування C# – це потужний інструмент для організації взаємодії різних частин програми без жорсткого прив’язування їх одне до одного. Ми описали різні випадки використання подій. По-перше, за допомогою опрацювання внутрішньої події складова частина компонента реагує на зміну стану іншої частини. По-друге, компонент описує власні події, щоб користувач компонента зміг дізнатися про важливі зміни і відреагувати на них. По-третє, компонент активно

взаємодіє з середовищем програмування та зі стандартними компонентами шляхом опрацювання їхніх подій.

ЛІТЕРАТУРА

1. *Уотсон К.* Microsoft Visual C#. Базовый курс / К. Уотсон, К. Нейгел, Я. Х. Педерсен, Дж. Д. Рид, М. Скиннер, Э. Уайт. – Пер. с англ. – Москва: “Диалектика”, 2009. – 1212 с.
2. *MacDonald M.* Pro .Net 2.0 Windows Forms and Custom Controls in C# / M. MacDonald. – Apress, 2006. – 1040 p.
3. *Агуров П.* C#. Разработка компонентов в MS Visual Studio 2005/2008 / П.В. Агуров. – Санкт-Петербург: БХВ-Петербург, 2008. – 480 с.
4. *Ярошко Сергій* Construction and Design-Time Support of the RadioGroup User Control [Електронний ресурс] / С. Ярошко // CodeProject. – 2017. – Режим доступу: http://www.codeproject.com/Articles/1204564/Construction-and-Design-Time-Support-of-the-RadioG

Стаття: надійшла до редколегії 12.09.2018
доопрацьовано 18.10.2018
прийнята до друку 31.10.2018

USAGE OF THE EVENT MECHANISM OF C# .NET IN CREATION OF WINDOWS FORMS COMPONENT

Sergii Yaroshko[1], **Svitlana Yaroshko**[2]

[1]*Ivan Franko National University of Lviv,*
Universytetska Str., 1, Lviv, 79000, e-mail: kafprog@lnu.edu.ua
[2]*National University “Lviv Polytechnic”,*
Stepana Bandery Str, 12, Lviv, 79013, e-mail: sm.jaroshko@gmail.com

Let us suppose we want to ask a user of our application to make a choice from some alternatives. This task is quite simple: we can put several radio buttons into the application window and recognize programmatically which of them the user has selected. If we want to have several separate groups of *RadioButton* controls, the most common method is to group them in some container controls, such as *Panel* or *GroupBox*. It works well at first time, but after a couple iterations with container and buttons, we will desire to have something more useful, for example, a control containing a collection of radio buttons. It would be similar to the *TRadioGroup* component from Visual Component Library that was very popular in earlier time.

The article describes in details the creation process of a new user-defined control by C# for the Windows Forms framework – the *RadioGroup* control. It is paid attention to the IDE tools of design time support of the *RadioGroup* control, such as categories and definitions of properties and events, property editors, smart-tags and so on. To achieve the goal we used in many cases the *events* – one of the most important mechanisms of object-oriented programming in C#. While the *RadioGroup* control were created, we have handled existing events of many components of the framework, have created and used several events of the control and of its parts.

The control will display a group of radio buttons bounded by border with a caption text. The key property of the *RadioGroup* control will be *Items* – a collection of strings representing the buttons names. The control creates radio buttons automatically, according to the *Items* content, like a *ListBox* control does, arranges the buttons uniformly over the own surface in one or more columns. The *RadioGroup* control notifies a program about the button index that the user has selected.

At first glance, there are no obstacles to achieve this goal. We can construct a user control with a *GroupBox* (to provide the control with a *Text* and a border), a *TableLayoutPanel* (to arrange

the buttons), an indexed collection of names (a *StringCollection*) and an indexed collection of buttons (a *List<RadioButton>*). We met a first problem at this point. The *StringCollection* provides a lot of modifying methods, such as *Add*(*aString*), *Clear*(), *RemoveAt*(*position*) and so on. But execution of the *Items.Add*("*Name*") do not cause any changes in the buttons collection. Also difficulties occur when we try to ensure that our control behaves properly at design time and the *RadioGroup* interacts well with Visual Studio.

We have to do some work to fix the problem with the *Item* property. Every changes of the names collection must reflect on the buttons collection. The name collection should signal about changes to the parent control. For this purpose we can define a new class that derives from *StringCollection*, extend it with an event and redefine every add-remove methods to signal the event.

The *IndexChanged* event is a specific event of the RadioGroup control. It is also the default event. It raises when a user clicks on an unchecked button or sets a new value to the *IndexSelected* property. Mouse events occur with the enclosed controls of the *RadioGroup*: with a button or a table layout panel. Thus we have to take care to transfer the events to the parent control. So the *button_Click*() method calls the *RadioGroup.OnClick*() and the *theTableLtPnl_DoubleClick*() method calls the *RadioGroup. OnDoubleClick*().

To teach our control to build buttons at design time we can write a special designer for our control and make it responsible for assigning the handler to the *IcomponentChange-Service.ComponentChanged* event. This event raises every time any property of any control of the application has changed while the application is being designed, so we have to recognize the type of control and the name of its changed property.

Key words: event of a component, user-defined control, Windows Forms, RadioGroup, smart-tag, component designer.