DOI https://doi.org/10.35219/rce206705325

Dynamic creation of visual components in C# language

Lupasc Adrian
alupasc@ugal.ro
Dunarea de Jos University of Galati, Romania

Visual programming is a way of designing an application by directly exploiting a set of graphical elements specific to the used environment. An application is visual when it has a suggestive graphical interface that provides the user with certain tools through which various functions are accessed (updating a database, for example). The development of such an application consists not only in placing the required controls on the form, but also in describing their associated functionality. The visual components of the application can be processed in the Designer module by a simple drag and drop or can be created dynamically during the running of the application. In this regard, this paper discusses the way in which various visual components of an application developed in C# can be created and added to a form in a dynamic manner. Such a method brings increased flexibility to the developed application, while providing dynamism in designing the tasks to be implemented. Moreover, the methods associated to the implemented template class can be called for any application the programmer wants to develop.

Keywords: class, instance, static data field, static method, static constructor.

JelClassification: C99, I29, J15, L63, L86

1. Introduction

C# was created by Microsoft as a development tool for .NET architecture and combines facilities tested over time with cutting-edge innovations. The language provides an effective way to write programs for a professional and modern development environment.

Often, the most difficult problem with learning a programming language is that no element is isolated. The components of a programming language work together and this relationship makes it difficult to present an aspect of C# language without involving others.

In this sense, C# has been characterized as a component-oriented language because it contains complete support for the development of software components. For example, C# has features that directly implement elements that make up components, such as properties, methods, and events.

Developing a visual application in C# involves building at least one form, the objects corresponding to the controls contained in it, setting the properties for each object (position, size, text, colors) as well as their functionality (source code to be executed which is associated with the defined events) and adding the objects within the control tree associated with the form.

When the controls related to the graphical interface of an application are static (they do not depend on the result of running the program at a certain step), adding them at the time of designing through the drag and drop method is recommended and most appropriate. However, in many situations, some controls may need to be added to the form during the execution of the application, and their number depends, for example, on the result of a query on a database table. In this case, the visual components of the application can no longer be processed in Designer mode, but a programmable solution is needed to add these components in real time to the graphical interface.

2. Template class with methods for dynamic creation of visual components

In this section, we'll describe how to create objects for the *CheckBox*, *RadioButton*, *GroupBox*, Button, *TextBox*, and *ListBox* classes. Also, the manner in which dynamic functionality is associated through events related to each control will be described.

Thus, a static class has been created (*createdNewControlClass*.cs) that includes static methods by which objects related to the classes described above are created (Figure 1). On each call of a method, an object is created (a visual component in this case) which is then attached to a form related to the graphical interface of the application.

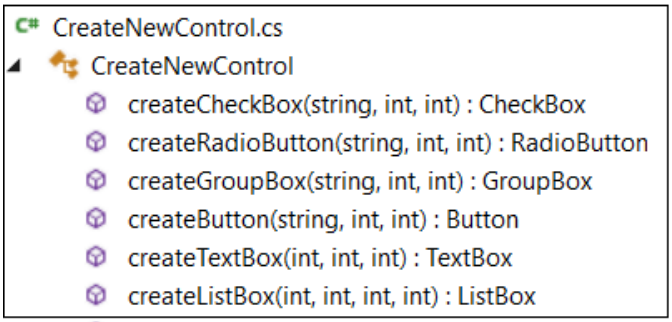

Figure 1 – Methods for dynamic creation of visual components

CreateCheckBox, createRadioButton, createGroupBox and createButton methods

The source code associated with the *createCheckBox* and createRadioButton methods is shown in Figure 2, while the source code for the *createdGroupBox* and *createButton* methods is shown in Figure 3.

```
public static CheckBox createCheckBox(string checkBoxText, int leftPosition, int topPosition)
{
    CheckBox ch = new CheckBox();
    ch.Text = checkBoxText;
    ch.AutoSize = true;
    ch.Font = new System.Drawing.Font("Microsoft Sans Serif", 12F);
    ch.Location = new System.Drawing.Point(leftPosition, topPosition);
    return ch;
}
```

```
public static RadioButton createRadioButton(string radioButtonText, int leftPosition, int topPosition)
{
    RadioButton rb = new RadioButton();
    rb.Text = radioButtonText;
    rb.AutoSize = true;
    rb.Font = new System.Drawing.Font("Microsoft Sans Serif", 12F);
    rb.Location = new System.Drawing.Point(leftPosition, topPosition);
    return rb;
}
```

Figure 2 – The source code associated with the createCheckBox and createRadioButton methods

```
public static GroupBox createGroupBox(string groupBoxText, int leftPosition, int topPosition)
{
    GroupBox gb = new GroupBox();
    gb.Text = groupBoxText;
    gb.AutoSize = true;
    gb.Font = new System.Drawing.Font("Microsoft Sans Serif", 12F);
    gb.Location = new System.Drawing.Point(leftPosition, topPosition);
    return gb;
}
```

```
public static Button createButton(string buttonText, int leftPosition, int topPosition)
{
    Button btn = new Button();
    btn.Text = buttonText;
    btn.AutoSize = true;
    btn.Font = new System.Drawing.Font("Microsoft Sans Serif", 12F);
    btn.Location = new System.Drawing.Point(leftPosition, topPosition);
    return btn;
}
```

Figure 3 – The source code for the createdGroupBox and createButton methods

As can be seen in Figures 2 and 3, the definition of the four methods is similar: all methods are static (cannot be accessed by class objects), are public, have three formal parameters (a string type and two *int* type) and when called returns an object of the class for which it was created (*CheckBox*, *RadioButton*, *GroupBox* or *Button*). The string parameter of the methods picks up the name the control will receive when it is added to a visual component of the application (for example, to a form). The two integer type formal parameters (*leftPosition*, *topPosition*) will retrieve the coordinates of the upper left corner of the control (within the component to which they are added). Each method sets the main properties associated with the control that will be created (location, color, font and name).

CreateTextBox and createListBox methods

Figure 4 includes the source code associated with the *createTextBox* and *createListBox* methods. Unlike the methods described in Figures 2 and 3, the *createTextBox* method includes three formal int type parameters that set the top left corner of the control (within the component to which it is added) while the width parameter sets the length of the control. In addition to the three related parameters and the *createTextBox* method, *createListBox* method also includes a formal int type parameter that sets the height of the control when it is added to a parent component.

```
public static TextBox createTextBox(int leftPosition, int topPosition, int width)
{
    TextBox tb = new TextBox();
    tb.AutoSize = true;
    tb.Width = width;
    tb.Font = new System.Drawing.Font("Microsoft Sans Serif", 12F);
    tb.Location = new System.Drawing.Point(leftPosition, topPosition);
    return tb;
}
```

```
public static ListBox createListBox(int leftPosition, int topPosition, int width, int height)
{
    ListBox lb = new ListBox();
    lb.Height = height;
    lb.Width = width;
    lb.Font = new System.Drawing.Font("Microsoft Sans Serif", 12F);
    lb.Location = new System.Drawing.Point(leftPosition, topPosition);
    return lb;
}
```

Figure 4 – The source code associated with the createTextBox and createListBox methods

3. Adding objects related to dynamically created visual components on a form

In order to argue the usefulness of dynamically generating controls, let us suppose we want to add on a form all the study programs running at the chosen faculty and in the user-selected field, in order to allocate the number of places open for contest (both tax-free and paid) for the admission activity taking place at a university (Figure 5).

If the study programs are added (Figure 5) in Designer mode, when the application is being designed, the programmer could add controls (of the CheckBox type, for example) for all study programs managed in the database at that time. But what will happen if, at the next admission, at the level of a faculty, other study programs are added? Will these appear on the graphical interface? In the context of the design method of the graphical interface described above, the answer is no. Basically, when controls are "coupled" to the data dynamics of a database, their recommended design solution is the programmable one, that is, while running the application.

Figure 5 – Graphic interface for allocating locations to a domain

Thus, the CheckBox controls in Figure 5 and the *TextBox* controls corresponding to the number of places (tax free and paid) were created dynamically to provide, at any time during the execution of the application, the updated form of data included in the database. Thus, the code sequence that allows for the dynamic creation of *CheckBox* controls corresponding to the study programs is shown in Figure 6, while the code sequence that allows dynamic generation of *TextBox* controls for the number of tax-free / paid places is shown in Figure 7.

```
foreach(DataRow dr1 in ds.Tables[2].Rows)
{
    if(dr1.ItemArray.GetValue(2).ToString()==codFacultate)
    {
        CheckBox ch = new CheckBox();
        ch.Location = new Point(10, pozitiech);
        ch.Text = dr1.ItemArray.GetValue(1).ToString();
        ch.AutoSize = true;
        ch.Font = new Font("Microsoft SANS MS", 20f);
        ch.Click += ch_Click;
        this.tabPageAlocare.Controls.Add(ch);
        myListCB.Add(ch);
        pozitiech += 35;
```

Figure 6– Dynamic creation of CheckBox controls corresponding to the study programs

```
        TextBox tbb = new TextBox();
        tbb.Location = new Point(900, pozitietbb);
        tbb.Size = new Size(50,Font.Height+10);
        foreach(DataRow drr in ds.Tables[3].Rows)
        {
            if (drr.ItemArray.GetValue(0).ToString() == dr1.ItemArray.GetValue(0).ToString() && drr.ItemArray.GetValue(1).ToString() ==
                den_an && drr.ItemArray.GetValue(2).ToString().CompareTo("0")==0 && Convert.ToInt32(drr.ItemArray.GetValue(5))==1 )
                    tbb.Text = drr.ItemArray.GetValue(3).ToString();
        }

        tbb.ReadOnly = true;
        this.tabPageAlocare.Controls.Add(tbb);
        tbBuget.Add(tbb);

        TextBox tbt = new TextBox();
        tbt.Location = new Point(1000, pozitietbt);
        tbt.Size = new Size(50, Font.Height + 10);
        foreach (DataRow drr in ds.Tables[3].Rows)
        {
            if (drr.ItemArray.GetValue(0).ToString() == dr1.ItemArray.GetValue(0).ToString() && drr.ItemArray.GetValue(1).ToString() ==
                den_an && drr.ItemArray.GetValue(2).ToString() == "1")
                    tbt.Text = drr.ItemArray.GetValue(3).ToString();
        }

        tbt.ReadOnly = true;
        this.tabPageAlocare.Controls.Add(tbt);
        tbTaxa.Add(tbt);

        pozitietbb += 35;
        pozitietbt += 35;
    }
}
```

Figure 7– Dynamic generation of TextBox controls corresponding to the number of tuition-free / paid places

In Figure 6, *ds.Tables[2]* refers to the content of the Study Program table whose data was included in the *DataSet* object with the name ds. In this respect, through the repetitive structure, all the study programs belonging to the faculty, whose code is the same as the one stored in the variable *codeFaculty* (the faculty code selected by the user on the graphical interface). In this way, for each dr1 object (corresponding to each study program) a *CheckBox* control is created and dynamically added to the form (in a *TabPage* object).

After creating and adding to the form a CheckBox object corresponding to a study program, two other TextBox controls are created (also dynamically) to retrieve the number of places in the database for both the Tax-free and Paid financing method. These controls are also added to the *TabPage* control.

All created controls (both *CheckBox* and *TextBox*) are then added to Generic type lists (highly-styled lists of objects that can be accessed through an index and that provide methods for searching, sorting, or manipulating items thereof). The three lists are shown in Figure 8, and the objects were then added to them to be later manipulated (for example, when the user selects another faculty, the study programs must be removed from the form and added to the currently selected one).

```
List<CheckBox> myListCB = new List<CheckBox>();
List<TextBox> tbBuget = new List<TextBox>();
List<TextBox> tbTaxa = new List<TextBox>();
```

Figure 8– List<T> type lists for manipulating objects dynamically created

4. Conclusions and final remarks

This paper proposes an interactive solution to dynamically add visual components during the execution of the application. In this way, a way to create objects related to CheckBox, RadioButton, GroupBox, Button, TextBox, and ListBox was presented and the way in which dynamic functions can be associated through the events of each control was described. Also, it was presented the way in which objects related to the dynamically created visual components are being added on the form by means of a practical example, involving the allocation of places for the admission contest taking place annually at the level of a university. Summarizing, it can be considered that the implementation of the aspects described in this paper can create an increased

dynamics in the development of computer applications, which are characterized by an increased dynamics of the data managed and manipulated during their execution.

References

1. *Albahari Joseph, Albahari Ben (2017), C# 7.0 in a Nutshell: The Definitive Reference, O'Reilly Media, Inc, 2017.*
2. *Emmisberger Patrick (2013), Dynamic test generation with static fields and initializers (Bachelor's Thesis).*
3. *Kendal Simon (2011), Object Oriented Programming using C#, Ventus Publishing ApS.*
4. *Kurt Nørmark (2010), Object-oriented Programming in C# (http://www.cs.aau.dk/~normark/oop-csharp/html/notes/theme-index.html).*
5. *Mulchrone Kieran (2010), An Introduction to Object Oriented Programming with C#*
6. *Purdum Jack (2012), Beginning Object Oriented Programming with C#, Wrox, 2012.*
7. *Rahman Mohammad (2012), Expert C# 5.0: With the .NET 4.5 Framework, Apress Publications.*
8. *Xua Jin, Liub Guoqiang (2013), Analysis and Application of Class and Interface in C# Language, Trans Tech Publications, Switzerland, pp.1929-1932.*