APPENDIX F

The Views.Form class

The Views.Form class provides an alternative to the Visual Studio design tool for developing graphical user interfaces. This appendix provides a specification for the XML notation used by the class constructor to lay out the elements of a form and describes the properties and methods exported by the class.

F.1 Creating Forms with Views

The `Form` class in the `Views` namespace provides a simple way to create sophisticated graphical user interfaces (GUIs). On Windows systems, it is possible to create GUIs by using classes in the `System.Windows.Forms` namespace. Writing the code to use these classes is possible but tedious and error-prone. Professional programmers would normally use the Microsoft Visual Studio development environment to generate much of the code automatically.

Views offers an alternative approach which is designed to be easy to use. The GUIs created with Views support only a subset (albeit a rich subset) of the full repertoire of controls possible with Windows forms. Interaction between the calling program and the controls used on a Windows form is normally implemented using events. The interface provided by the `Views.Form` class is simplified so that programs cannot achieve the same effects as a full Windows program. If the calling program needs more sophisticated interaction with the control, it is possible but only by implementing its own event handlers and abjuring the simplified methods for interaction provided by the `Views.Form` class.

Views is intended to be easy to use, to require a small memory footprint on the computer, and is being ported to different platforms. Views is available for download in both compiled and source code versions from the official website. Its URL is

```
http://www.cs.up.ac.za/csharp
```

On Windows systems where you type a command into a command window (or MS-DOS window) to invoke the C# compiler, you need to use one of the following commands to compile a program that uses Views:

```
csc file1.cs file2.cs ...
```

or

```
csc /r:Views.dll file1.cs file2.cs ...
```

where your C# source code files are named `file1.cs`, `file2.cs`, etc.

The first, and simpler, command will work if the Views namespace has been installed and registered by the system administrator. The second version must be used otherwise. It requires that a copy of the file `Views.dll` be located in the same folder as your source code files. (It is a relatively small file and therefore you are unlikely to run out of disk space if you keep several copies of the file, one for each of the programs you are working on.)

F.2 Syntax of specifications for Views.Form

The `Views.Form` class constructor checks the argument string to verify that it is a valid specification for laying out the controls on a Windows form. The verification includes checking that the tags are nested appropriately, that the tags are provided with all the necessary attributes, that the tags are only provided with attributes supported by Views, and that attribute values are reasonable.

The tags may be nested according to the rules shown in Table F.1. The first two rules say that a form may be constructed in two ways.

form:	`<Form>` *controlGroup* `</Form>`
|	`<Form>` *positionList* `</Form>`
controlGroup:	`<vertical>` *controlList* `</vertical>`
|	`<horizontal>` *controlList* `</horizontal>`
|	`<Panel>` *positionList* `</Panel>`
controlList:	{ *control* }
positionList:	{ *positionedControl* }
positionedControl:	`<position>` *control* `</position>`
textItemList:	{ `<item>` *text* `</item>` }
control:	*controlGroup*
|	`<Button/>`
|	`<CheckBox/>`
|	`<CheckedListBox>` *textItemList* `</CheckedListBox>`
|	`<DomainUpDown>` *textItemList* `</DomainUpDown>`
|	`<GroupBox>` *radioButtonList* `</GroupBox>`
|	`<Label/>`
|	`<ListBox/>`
|	`<OpenFileDialog/>`
|	`<PictureBox/>`
|	`<ProgressBar/>`
|	`<SaveFileDialog/>`
|	`<TextBox/>`
|	`<TrackBar/>`
radioButtonList:	{ `<RadioButton/>` }

Table F.1 Syntax of the Views XML Specification

1. The first way is by writing the tag `<Form>` followed by a *controlGroup* followed by the closing tag `</Form>`.
2. The second way is by writing the tag `<Form>` followed by a *positionList* followed by `</Form>`.

In turn, a *controlGroup* is defined by the three rules which come later in the table, and a *positionList* by another rule later in the table.

If we look at the rule for *positionList* we see that it is defined as { *positionedControl* }. The curly braces are used to indicate that the material within the braces may be repeated

indefinitely often. That is, a *positionList* consists of zero or more *positionedControl* constructions, one after the other.

Note that the rule which says that a *control* can be a *controlGroup* is a rule that permits nesting of `<vertical>...</vertical>` groups inside `<vertical>...</vertical>` groups, and so on.

The attributes which can be provided for each tag are detailed in the following sections of this appendix where the tags are listed with some explanations.

Note that capitalization of the tag names in Table F.1, such as `RadioButton`, matches that of the name of the corresponding class in the `System.Windows.Forms` namespace. The full name of the `RadioButton` class is therefore

```
System.Windows.Forms.RadioButton
```

and exactly this capitalization must be used in a C# program that refers to the class.

For your convenience, the `Views.Form` class accepts any capitalization of the tag names and the attribute names. However, it would be good practice for you to adopt the same capitalization as the Windows class names because you will be less likely to make mistakes when you subsequently use those classes.

F.3 Grouping tags

The tags listed in Table F.2 are those which enclose groups of other tags. The `<vertical>`, `<horizontal>` and `<Panel>` tags are used for generic control groups (called *controlGroup* in Table F.1) whereas the `<Form>` and `<GroupBox>` are used in more restricted circumstances.

An asterisk alongside an attribute (usually the `Name` attribute) indicates that the attribute must be specified. Explanations of the code letters used for the attribute values in the table appear in Section F.5.

Grouping construct	Attributes	Description

`<Form>`
`...`
`</Form>`
	`Text=S`
	`ForeColor=C`
	`BackColor=C`
		The outermost pair of tags needed to begin and end a complete specification. The enclosed contents must be a vertical list, or a horizontal list, or a panel, or a single control, or a list of `<position>` ... `</position>` controls

`<vertical>`
`...`
`</vertical>`
	`Width=M`
	`Height=M`
	`ForeColor=C`
	`BackColor=C`
		Display the enclosed constructs in a vertical list

`<horizontal>`
`...`
`</horizontal>`
	`Width=M`
	`Height=M`
	`ForeColor=C`
	`BackColor=C`
		Display the enclosed constructs in a horizontal list

`<Panel>`
`...`
`</Panel>`
	`*Name=S`
	`Width=M`
	`Height=M`
	`ForeColor=C`
	`BackColor=C`
		The enclosed contents must be a list of `<position>` ... `</position>` controls, each of which places a control at a precise location

`<GroupBox>`
`...`
`</GroupBox>`
	`*Name=S`
	`Text=S`
	`Width=M`
	`Height=M`
	`ForeColor=C`
	`BackColor=C`
		The enclosed contents must be `<RadioButton>` controls

Table F.2 Views.Form grouping constructs

F.4 Control tags

Table F.3 lists all the basic controls supported by the Views.Form class. The table lists all the attributes appropriate for each control *except* for `halign` and `valign`. The `halign` attribute may be used with any open tag which is immediately nested inside a `<vertical>` ... `</vertical>` group, while a `valign` tag may be used with an open tag immediately nested inside a `<horizontal>` ... `</horizontal>` group.

An asterisk alongside an attribute (usually the `Name` attribute) indicates that the attribute *must* be specified. Explanations of the code letters used for the attribute values in the table appear in Section F.5.

Views.Form control	Attributes	Description

`<Button/>`	`*Name=S`
	`Text=S`
	`Image=F`
	`Width=M`
	`Height=M`
	`ForeColor=C`
	`BackColor=C`
	`Font=FNT`
		Creates a push button.
		The button can be labelled with a string (taken from the `Text` attribute) or with a picture (where the name of the file containing the picture is taken from the `Image` attribute) or both.
		The size of the button defaults to something large enough to hold the label, either text or an image.
		Clicking the button causes `GetControl` to return with the name of the control

`<CheckBox/>`	`*Name=S`
	`Text=S`
	`Width=M`
	`Height=M`
	`ForeColor=C`
	`BackColor=C`
	`Font=FNT`
		Creates a small square which the user can click to add or remove a check mark.
		The `GetValue` method can be used to retrieve the status of a check box

`<CheckedListBox>`
`...`
`</CheckedListBox>`
	`*Name=S`
	`Text=S`
	`Width=M`
	`Height=M`
	`ForeColor=C`
	`BackColor=C`
	`Font=FNT`
		Creates a pull-down list of check boxes.
		The `GetValue` method can be used to retrieve the status of each check box in the list

`<DomainUpDown>`
`...`
`</DomainUpDown>`
	`*Name=S`
	`Text=S`
	`Width=M`
	`Height=M`
	`ForeColor=C`
	`BackColor=C`
	`Font=FNT`
		Creates a pull-down list from which a single item in the list can be selected as the current value.
		The `GetValue` method can be used to retrieve the index of the currently selected item in the list

`<GroupBox>`
`...`
`</GroupBox>`
	`*Name=S`
	`Text=S`
	`Width=M`
	`Height=M`
	`ForeColor=C`
	`BackColor=C`
	`Font=FNT`
		Displays a rectangular box used to hold a group of radio buttons. Only one radio button at a time can be selected – if the user clicks on one, it is enabled and another one becomes disabled.
		The `GetText` method can be used to retrieve the label of which radio button in the group is currently selected

`<Label/>`	`Name=S`
	`Text=S`
	`Width=M`
	`Height=M`
	`ForeColor=C`
	`BackColor=C`
	`Font=FNT`
		Displays a string

Table F.3 Views.Form controls

Views.Form control	Attributes	Description

`<ListBox/>`	`*Name=S`
	`Text=S`
	`Width=M`
	`Height=M`
	`ForeColor=C`
	`BackColor=C`
	`Font=FNT`
		Creates a rectangular box which can be used for input or output of many lines of text.
		Currently selected text in a `ListBox` can be retrieved by the `GetText` method; the `PutText` method appends new text to the `ListBox` contents

`<OpenFileDialog/>`	`*Name=S`
	`Text=S`
	`Width=M`
	`Height=M`
	`ForeColor=C`
	`BackColor=C`
	`Font=FNT`
		Creates a button which, if pressed, causes a new window to open where an existing file can be selected by navigating through the file system.
		Clicking the button and selecting a file causes `GetControl` to return with the name of the control; the name of the selected file can be retrieved by the `GetText` method

`<Panel>`
`...`
`</Panel>`
	`Name=S`
	`Text=S`
	`*Width=M`
	`*Height=M`
	`ForeColor=C`
	`BackColor=C`
	`Font=FNT`
		Creates a rectangular region, called a panel, within which controls may be individually placed. The panel may be scrollable horizontally or vertically, as neeeded to view all the controls.
		The controls inside the panel are placed at specific coordinates using `<position> ... </position>` tags

`<PictureBox/>`	`Name=S`
	`Image=F`
	`Width=M`
	`Height=M`
		Displays a graphics image.
		If the `Image` attribute is left undefined, a grey rectangle will be displayed. If `Width` and `Height` are omitted, they default to the size of the image held in the file

`<ProgressBar/>`	`*Name=S`
	`Value=D`
	`Minimum=D`
	`Maximum=D`
	`Width=M`
	`Height=M`
	`ForeColor=C`
	`BackColor=C`
		Creates a horizontal bar where the shaded part on the left is used to indicate how much of a task has been completed.
		Note: Views provides a default value for `Minimum` of 0 and a default for `Maximum` of 100.
		The amount of progress displayed by the control can be changed by calling the `SetValue` method

`<RadioButton/>`	`*Name=S`
	`Text=S`
	`Checked=D`
	`Width=M`
	`Height=M`
	`ForeColor=C`
	`BackColor=C`
	`Font=FNT`
		Creates a round button which becomes checked when clicked. A list of radio buttons is enclosed by `GroupBox` tags.
		The checked state of the button can be determined by calling the `GetValue` method. Alternatively, the name of which button in the group is checked can be obtained by calling the `GetText` method on the `GroupBox` control

Table F.3 Views.Form controls (continued)

Views.Form control	Attributes	Description

`<SaveFileDialog/>`	`*Name=S`
	`Text=S`
	`Width=M`
	`Height=M`
	`ForeColor=C`
	`BackColor=C`
	`Font=FNT`
		Creates a button which if pressed causes a new window to open where either an existing file to be overwritten can be selected by navigating the file system or a new filename can be entered. Clicking the button and selecting a file causes `GetControl` to return with the name of the control; the name of the selected file can be retrieved by the `GetText` method

`<TextBox/>`	`*Name=S`
	`Text=S`
	`Width=M`
	`Height=M`
	`ForeColor=C`
	`BackColor=C`
	`Font=FNT`
		Creates a rectangular text box which can be used for input or output of a short text item. The current text in a text box can be retrieved by the `GetText` method and can be changed by the `PutText` method

`<TrackBar/>`	`*Name=S`
	`Value=D`
	`Minimum=D`
	`Maximum=D`
	`Width=M`
	`Height=M`
	`ForeColor=C`
	`BackColor=C`
		Creates a slider control where the user can drag a marker backwards or forwards with the mouse. Note: Views provides a default value for `Minimum` of 0 and a default for `Maximum` of 100. Moving the slider causes `GetControl` to return with the name of the control, and the new value represented by the slider can be read by the `GetValue` method

Table F.3 Views.Form controls (continued)

F.5 Attribute values

The nature of each attribute value is indicated in Tables F.2 and F.3 by a code letter. The meanings of the letters are as follows:

`A`	an alignment (see below)
`S`	a text string written enclosed in single quotes or double quotes or as an unbroken sequence of letters, digits, hyphens and periods (full stops)
`M`	a size measure (see below)
`C`	the name of a colour
`F`	the name of a file (actually a path to a file)
`D`	a decimal value
`FNT`	a font specification (see below)

Font specification. The font for the text displayed in a control can be selected using the Font attribute. Views provides a simple notation for specifying fonts. The specification is composed of four parts: the name of the font family, the weight of the font, the slant of the font and the size of the font. The abbreviations for the first three components that are accepted by Views are follows:

Family			

`sansserif`
`sans`
`sf`
	SansSerif	`roman`
		`rm`
			Roman

`monospace`
`courier`
`tt`
`teletype`
	Monospaced		

Weight			
`medium` `md`	*Medium*	`bold` `bf`	*Bold Face*
Style			

`upright`
`up`
	Upright	`italic`
		`it`
		`emphasis`
		`em`
			Italic

The name *Roman* (and its synonym *rm*) refers to a generic serif font, usually Times Roman; *SansSerif* (and its synonyms *sans* and *sf*) usually refers to Arial or Helvetica; *Monospace* (and its synonyms *Courier*, *tt* or *teletype*) refers to a generic monospaced font such as *Courier.*

The name *Bold* (and its synonym *bf*) gives a bold font; a regular unemboldened font may be obtained by using the name *Medium* (or its synonym *md*).

The name *Italic* (or its synonyms *it, Emphasis* or *em*) gives a slanted font style; a normal upright font style may be obtained by using the name *Upright* (or its synonym *up*).

The size of the font may be obtained by writing the size in points as a decimal number; that number may optionally contain a fractional part.

The descriptors may be combined in any order. Some examples are shown in the following table:

Font description	**Sample**
`Bold24`	**Hello there**
`ItalicSans16`	*Hello there*
`Courier9.5`	`Hello there`

Size measures. The height or width of a control can be expressed as simply a decimal number, in which case the unit of size defaults to points. If desired, a different unit of measurement may be supplied immediately following the number. The recognized units are listed in Table F.4.

Name	Meaning
in	Inches
cm	Centimetres
mm	Millimetres
pt	Points (the default)
pc	Picas

Table F.4 Measurement units

There are 72 points per inch. Views assumes that the screen resolution is one pixel per point, i.e. 72 pixels per inch.

Some examples of attribute settings are:

```
Width=64
Height=2.5cm
Image='C:\Temp\MyPhotos\picture1.jpg'
Name=Start
Text='Select the input file'
```

Alignment settings. The `valign` and `halign` attributes may be assigned the following values:.

`valign:`	`Top` (the default), `Middle`, `Bottom`
`halign:`	`Left` (the default), `Centre` (or `Center`), `Right`

F.6 Views.Form methods

The methods supported by the `Views.Form` class are listed in Table F.5.

Note that because the Windows implementation of `Views.Form` inherits from the `System.Windows.Forms.Form` class, you may also call any public method of that parent class when you compile your C# program on a Windows system.

Views.Form method	Description
`Form( string spec, ...)`	It is the constructor: a Windows Form with the controls defined by the XML specification is created and displayed. The first argument may be either the XML specification, provided as a string constant, or it may be the name of a file which contains the specification. The specification may contain {0}, {1} ... patterns, and these refer either to optional arguments which follow the XML specification, or to elements of an optional second argument which is an array of objects. If the `{i}` pattern is used, the `ToString` method of the *i*-th optional argument is invoked and the resulting string is substituted for the `{i}` pattern in the specification string
`void CloseGUI()`	Terminates the execution thread which waits for the user to click on the form and releases other system resources. It is important that this method be invoked when the form no longer needs to be displayed
`string GetControl()`	Waits for the user to perform an action on the form (e.g. clicking a button) and then returns the name of the control that was clicked
`string GetText( string name )`	Returns a text value that is associated with the control whose name is given. If the control is a `TextBox` or `ListBox`, that text has been entered by the user. If the control is a `OpenFileDialog` or `SaveFileDialog`, the text is the name of a file
`int GetValue( string name )`	Returns an integer value associated with the control whose name is given. For a `TrackBar` or `ProgressBar` control, this integer denotes the current position of the marker. For a `CheckBox` control, a zero or one result indicates whether the box is currently unchecked or checked, respectively. For a `DomainUpDown` control, the result is the index of the currently selected item in the list (the first item is numbered 0)
`int GetValue( string name, int index)`	For a `CheckedListBox` control, the result is the status of the check box at position *index* in the list, where 1 means checked and 0 means unchecked. For other controls, the result is the same as would be returned by `GetValue(name)`
`void PutText( string name, string cval )`	Sets the `Text` attribute of the control whose name is specified. It can be used to display text in a `TextBox` or `ListBox` control
`void PutValue( string name, int v )`	Sets an integer value associated with the control whose name is specified. This method is used to adjust the state of a `ProgressBar` or to set the state of a `CheckBox` control

Table F.5 Views.Form methods

Views.Form method	Description
`void PutImage( string name, string f );`	Replaces the image displayed in a control which has an `Image` attribute with a new image obtained from the file whose name is supplied as `f`
`void PutImage( string name, Image im );`	The same as above except that the image to be used is an instance of the `Image` class
`(instance of Views.Form)` `[string name]`	This is an indexer operation which returns the control whose name is provided. For the Windows implementation of Views, the control is an instance of a class in the `System.Windows.Forms` namespace

Table F.5 Views.Form methods (continued)

F.7 Recommended coding style

To maintain reasonable resemblance to the code used with Windows forms created with Microsoft Visual Studio, the following basic code pattern is recommended.

```
string f = @"<Form> ... </Form>";

try {
  Views.Form form = new Views.Form(f);
  for (;;) {
    string name = form.GetControl();
    if (name == null) break;
    ActionPerformed(name);
  }
  form.CloseGUI();
} catch( Exception e ) {
  Console.WriteLine("Error while using Views.Form:\n\n{0}",
    e.Message);
}
```

The `try` statement will intercept all errors from the statement where the form is created through to the statement where the form is closed.

The `ActionPerformed` method could have a structure like the following, though simplifications for especially simple forms and some special cases may be appropriate:

```
void ActionPerformed( string name ) {
  switch(name) {
  case "name1":
    ...
    break;
  case "name2":
    ...
    break;
  case ...        // as many cases as needed
  }
}
```

F.8 Use of the indexer operation

If an instance of the `Views.Form` class is created, for example like this

```
Views.Form f = new Views.Form( @"<Form>
      <vertical>
         <Label Name=Label1 Text='Enter Your Name: '/>
         <TextBox Name=Box1 Width=150/>
      </vertical>
   </Form>";
```

then the displayed form contains instances of controls. Access to an individual control may be obtained by using the indexer operation, and that access may be used to achieve run-time effects.

For example, assuming the above example of a form, the program that created the form instance may execute these statements

```
System.Windows.Forms.TextBox tb = f["Box1"];
System.Windows.Forms.Label lab = f["Label1"];
lab.BackColor = Color.Red;
lab.ForeColor = Color.Yellow;
f.Invalidate();  // force form to be redrawn with new colours
```

to change the font used for the `TextBox` control and the colours on the `Label` control.

Notes:

- The colour names are also defined in the `System.Drawing` namespace.
- The `Invalidate` method is inherited by `Views.Form` from its parent class, `System.Windows.Form`.