Joint NCC & IIS Bull., Comp. Science, 18 (2002), 45–67
© 2002 NCC Publisher

Options management in RescueWare system

M. A. Bulyonkov

A complex programming system may have a large number of parameters controling its configuration and behavior. The editing, programmatic access, packaging and upgrade of these parameters may become a complicated and time-consuming task. We present a component called Dialogic, which stores options as a strictly typed XML structure and automatically generates elaborated dialogs for editing options.

1. Introduction

The paper summarizes the experience gained during the development of the reengineering system RescueWare® and addresses one particular aspect: option management[1]. RescueWare is a large multi-purpose and multi-language system. It may be used for source code analysis and maintenance, for source-to-source transformation (e.g., business rule extraction), and ultimately for legacy system modernization, such as conversion of COBOL CICS-based application to Java and JDBC. The RescueWare system inputs a variety of legacy languages, such as COBOL, Natural, PL/I, DB/2 DDL, BMS, JCL, etc. RescueWare provides a number of tools working on a workstation for automated program analysis and system-wide program navigation.

We consider an option to be a parameter controlling either configuration or the behavior of the system. The nature and purpose of options may vary significantly. For example,

- the list of analysis tools;
- default extensions for COBOL files;
- color of the error highlighting in the editor;
- an option controlling whether an Open dialog must show up on startup or the system should go directly to the most recently opened project;
- a flag indicating whether unused code must be commented out or deleted during business rule extraction; etc.

Options in RescueWare are classified in four categories:

[1]RescueWare® is the registered trademark of Relativity Technologies, Inc.

- global options controlling configuration of the system;
- workspace options describing various characteristics of all objects that constitute the legacy system;
- project options specific to a particular project that is a subsystem of the workspace. Usually a project groups objects corresponding to functionality of a particular legacy system;
- user preferences do not affect core functionality of the system, but only the way the information is shown to the user. Usually, parameters like colors and sizes fall into this category.

A customer may be interested in a subset of available functionality and/or a subset of input languages. That gives rise to the problem of packaging. For instance, if some option is specific to the Natural programming language and this language is not included in the package, the option should not be displayed on any dialog.

In this paper, we present a component called Dialogic, which enables the unified storage, access, display, editing and maintenance of options. The component was originally designed in the context of the RescueWare system, but turned out to have enough generality to be successfully used in a number of other projects as well.

2. Option visualization

Modern integrated development environments (IDE) provide a number of means to ease the graphical user interface (GUI) design. The most common is visual construction of forms, where the developer can place a control on a form and immediately set its properties such as color, text, alignment, etc. Such an approach considerably simplifies GUI development — a lot of actions that otherwise would have to be done programmatically are performed by the system. However, this leads to certain drawbacks:

- Usually visual construction does not support corporative GUI standards, and it is completely up to user's responsibility to set all the properties consistently.
- Complex forms with many controls whose visibility is determined at run-time are hard to edit. Forms with hundreds of controls are practically un-manageable.
- If the layout of a form depends on configuration or run-time values, it must be rearranged programmatically anyway. For example, if some

control is not required in a particular configuration and is made invisible, it still resides on the form and consumes the form loading time, at least.

Corporative standards may vary, of course. Here are some accepted as RescueWare standard:

- *Alignment and sizes.*
  - All controls on the form must be properly aligned.
  - The gap between a control and its label must be the same throughout the application.
  - Heights of controls of the same type, e.g. buttons or line input text boxes, must be the same throughout the application.
- *Keyboard equivalents.*
  - All controls must be reachable through Tab and Shift-Tab keys in a natural order.
  - Any input control must be reachable through hot keys. Hot keys of controls that might be visible simultaneously should differ if possible.
  - "OK" and"Cancel" buttons must have no hot keys at all. "Apply" button must have a hot key.
- *Fonts and colors.* All controls must share the same font with the same size, if only the purpose of the control is not font-specific. Normally it is a standard system font, e.g., MS Sans Serif, but there might be a need to change this font throughout the applications for the purpose of localization.
- *Online help.* Eeach control must have assigned reference to corresponding Help topic.
- *Common behavior.* As an example, suppose that we have a checkbox "Limit Impact Depth" controlling appropriateness of a textbox "Max Depth". Then we have a choice of behavior: either the textbox is disabled when the checkbox is unchecked, or it may be enabled all the time and the checkbox is checked automatically whenever the user modifies the content of the textbox. Both behaviors have their benefits and drawbacks, but we need to choose only one to be used throughout the product.

The task of satisfying all these requirements becomes at least tiresome when the application grows larger and displays a lot of forms. It consumes significant development and quality assurance time, as well as management for supporting strict development discipline. Although all the above requirements might look supplementary since they do not affect the core functionality of the system, they are, in fact, very important because they guarantee common look-and-feel of the system and form the first customer's impression.

Dialogic provides a solution to this problem: all options are organized in a strictly typed data structure and all forms for editing any part of the structure are generated completely automatically satisfying all the above requirements.

3. Data types

An atomic parameter may have one of the following types: *text*, *number*, *set of strings*, *font*, *file name*, *color*, *enumeration*, or *boolean.* Atomic parameters may be composed into groups or lists. We do not claim the set of types is complete, but it covers most of the practical needs. Some atomic types can be narrowed by imposing additional restrictions. For example, we can specify the minimum, the maximum and the step for a number tag.

We represent option definition in the form of XML files. The benefits of XML files will be discussed later. Each parameter — either atomic or structured — can have attributes controlling the way it is displayed and edited. For example, a number can be displayed either as a textbox, or as a slider, or somehow else. So we can also consider options as an hierarchy of selection pages. Each selection page may contain some elements corresponding to atomic parameters and/or subordinate selection pages. Usually the set of top-level selection pages is displayed as a tab strip, so that the user may select a page by clicking the corresponding tab. Selection could also be controlled by combo-boxes, radio buttons, etc.

3.1. Accessing options

We keep options in XML format. One of the main reasons for using XML for representing options is it is being open and becoming a standard — there is a variety of tools for manipulating a document. Let us compare XML format to other format and tools that were used at different times for storing options in RescueWare.

- XML vs. Databases. Databases allow for elaborate data access. But XML also does. And since XML provides primarily hierarchical navigation, it is probably more suitable for options logical organization. For comparatively small amount of data, which is the case for options, access will probably be even faster. The disk space of options data is an order of magnitude smaller when implemented with XML.

- XML vs. INI files. INI files are easy to manipulate. So are XML files. You may edit them with your favorite text editor. Even if XML is more verbose, in a long run it is preferable, because the structure of XML is typed, and it is more flexible and reliable.
- XML vs. Registry. Registry is conceptually very close to options model, and it is easy to store hierarchical data in registry. But the same is true for XML. Registry is very hard to trace and there is an obvious tendency to leave a lot of hard-to-detect garbage there.

The Dialogic component is implemented as an ActiveX DLL and declares two classes: a global multi-use class `OptionManager` and a multi-use class `DiaLogicOptions`. The main method provided by `OptionManager` is

```
Public Function GetOptions(nm As String, FileName As String) As
DiaLogicOptions
```

The function associates a set of options from the file `FileName` with the same internal name `nm`. If the name `nm` is not yet defined, then the content of the `FileName` is loaded so that the loaded document could be later referred to as `nm`. Otherwise, a reference to the previously loaded object is just returned. This allows various components to share the same options and load them on demand.

Example.

```
Set DLO = DiaLogic.GetOptions("RescueWare", GetRescueDir & _
                      "Data\Options\RWOptions.xml")
```

The class `DiaLogicOptions` has the following methods:

```
Public Function LoadFile(FileName As String) As Boolean
```

loads options definition from XML file and returns `true` if successful.

```
Public Function Document() As DOMDocument
```

gives access to the loaded XML document.

```
Public Function DisplayOptions(Section As String, _
                OwnerForm As Object, _
                Optional StartID As String, _
                Optional DefaultWidth as Single= 6120, _
                Optional DefaultHeight as Single= 5355)
```

displays dialog for the specified `Section`, modally over `OwnerForm`. `StartID`, if specified, is the XML address of the selection page to be initially selected. If `StartID` is omitted, then the selection is performed based on the `VALUE` attributes of `LIST` tags. `StartID` is relative to the node specified by `Section`. `DefaultWidth` and `DefaultHeight` specify the desired width and height of the dialog window. These values will be automatically increased if necessary.

```
Event Saved()
```

is raised when “OK” or “Apply” button is pressed. In both cases the `VALUE` attributes are updated and the XML document is saved back to the disk.

Let us now describe all parameter types supported by DiaLogic.

3.2. Selection lists

A selection list is represented by the `LIST` tag, which contains a number of `CASE` tags, one per each selection page. For example,

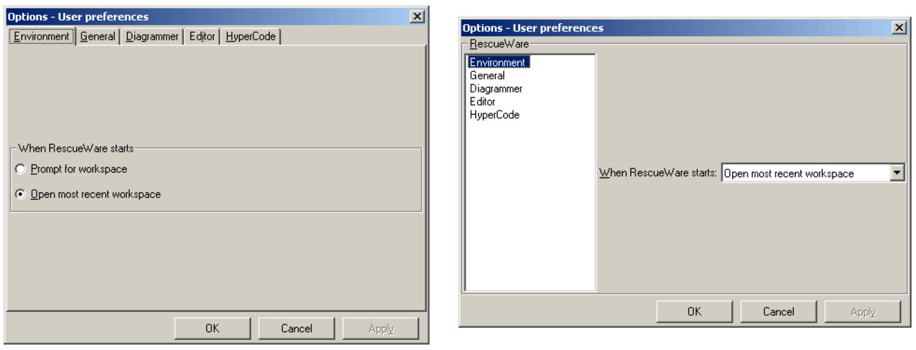
```
<LIST NAME="RW" TYPE="select" CAPTION="RescueWare"
            DISPLAY="page"  VALUE="Env">
    <CASE NAME="Env" CAPTION="Environment">
        <LIST NAME="OnStart"
                CAPTION="When RescueWare starts"
            TYPE="data" DISPLAY="radio" VALUE="Prompt">
            <CASE NAME="Prompt"
                CAPTION="Prompt for project"/>
            <CASE NAME="Open"
                CAPTION="Open most recent project"/>
        </LIST>
    </CASE>
    <CASE NAME="General" CAPTION="General">
    ...
    </CASE>
    <CASE NAME="Diagrammer" CAPTION="Diagrammer">
    ...
    </CASE>
    <CASE NAME="Editor" CAPTION="Editor">
    ...
    </CASE>
    <CASE NAME="HyperCode" CAPTION="HyperCode">
    ...
    </CASE>
</LIST>
```

Here, as well as for any other tag, the attribute `NAME` serves for option identification, while `CAPTION` provides the text to be shown on the screen. The `VALUE` attribute of the `LIST` tag holds the currently selected page that must coincide with the name of some of the `CASE`'s. The attribute `TYPE` may hold values `select` or `data` and distinguishes between the lists designed only for page selection and those whose `VALUE` is essential data.

Here and below the `DISPLAY` attribute controls the way information will be presented to the user. Currently we have the following styles for displaying lists:

- `page` — displays tabstrip where cases occupy the client area;
- `combo` — a combo-boxed controlled frame;
- `list` — the list of cases is on the left-hand side;
- `radio` — all selection pages are visible at once and selection is controlled by radio buttons;
- `checkdetailslist` and `detailslist` — the list whose cases are displayed on demand in a separate window.

The list above will be displayed as shown in Figure 1(a).



(a) *(b)*

Figure 1. Paged and radio list (*a*), list view and combo lists (*b*)

Had we changed the `DISPLAY` attributes of tags named `RescueWare` and `OnStart` to `page` and `radio` respectively, the dialog would appear as shown in Figure 1(b).

If the `DISPLAY` attribute is equal to `checkdetailslist` or `detailslist`, the content of a case sub-page is displayed on a separate dialog available via "Details" button. If the attribute equals to `checkdetailslist`, then the list with checkboxes is displayed, and "Select All" and "Unselect All" buttons

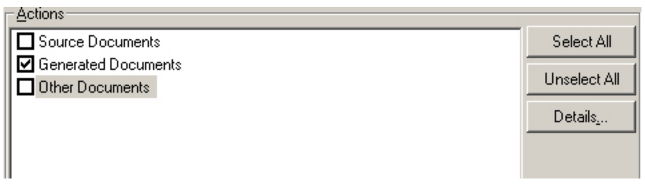


Figure 2. List with separate display of cases

are displayed as well. An example of a list displayed as `checkdetailslist` is shown in Figure 2.

This feature is useful in cases of voluminous and rarely accessed "details", since it significantly decrease loading and makes the dialog more compact.

3.3. Texts

The `TEXT` tag represents a text option. Besides the current value, it may specify the maximal allowed data length. Texts are displayed either as single- or multi-line, if the value of the `DISPLAY` attribute equals `line` and `area`, respectively. For example, suppose that the "General" page of the above list is specified as the following:

```
<CASE NAME="General" CAPTION="General">
  <TEXT NAME="PrjName" CAPTION="Project Name"
      MAXLENGTH="255" DISPLAY="line" VALUE=""/>
  <TEXT NAME="PrjDescr" CAPTION="Project Description"
     MAXLENGTH="255" DISPLAY="area" VALUE=""/>
</CASE>
```

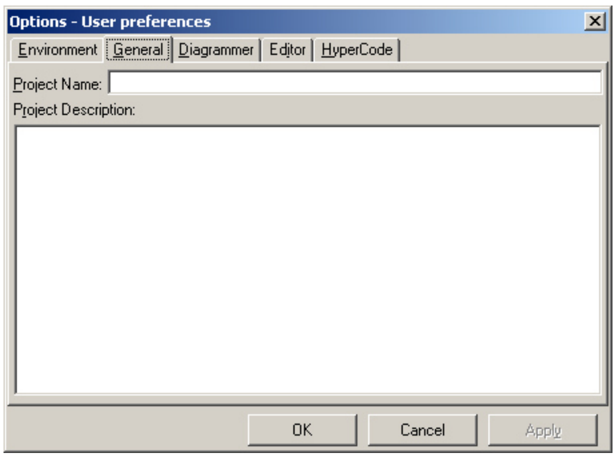
Then the page will be displayed as shown in Figure 3.

Note that we try to occupy the whole space of the page evenly. So if the page contains flexible height options, like `area` text, then their height will be extended to fit the height of the page.

3.4. Colors

The `COLOR` tag represents a color. It is shown as a filled box, on which the user may click to popup the color selection dialog. For example, Figure 4 shows how a couple of such options will be dispayed:

```
<COLOR NAME="Paragraph"
    CAPTION="Paragraph default color"
    VALUE="1552345"/>
<COLOR NAME="Paragraph Selected Color"
```



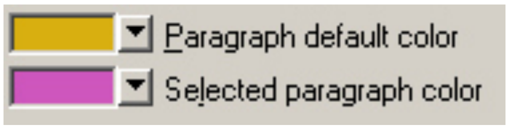
Options - User preferences

Project Name:

Project Description:

Figure 3. Texts

```
CAPTION="Selected paragraph color"
VALUE="12343245"/>
```



Paragraph default color

Selected paragraph color

Figure 4. Colors

The color selection dialog allows for selecting colors from both palette and system colors.

3.5. Fonts

Colors may also be manipulated by the `FONT` tag designed to select the font, fore color and background for texts. For example,

```
<FONT NAME="Fore" CAPTION="Fore Color"
        SAMPLE="MOVE A+ TO . "
        SELECT="fore,back,font"
        FORECOLOR="16777215"
        BACKCOLOR="255" />
```

The `SELECT` attribute specifies what the corresponding dialog may select. The optional `SAMPLE` attribute allows for indicating the text to be shown inside the textbox. If the attribute is omitted, then the current font name and its size are used as a sample. It is also possible to specify BOLD and ITALIC attributes for font characteristics.

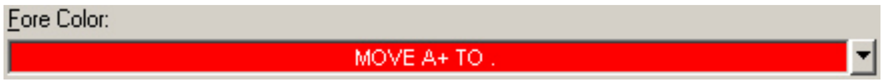


Figure 5. Fonts

3.6. Booleans

Boolean options are represented by the BOOL tag that can hold either `true` or `false` values and is displayed as a check-box. Since a boolean option represents the user's choice, it is natural that it could control a sub-page, which becomes enabled only if the check-box is checked. For example,

```
<BOOL NAME="Colorize" CAPTION="Colorize" VALUE="false">
   <COLOR NAME="Color" CAPTION="Fore Color" VALUE="312345"/>
</BOOL>
```

will be displayed as shown in Figure 6.

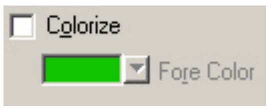


Figure 6. Booleans

3.7. Numbers

The NUMBER tag specifies numeric options. MIN, MAX, and STEP attributes limit the range of allowed values. Numbers are displayed and manipulated in three different ways depending on the value of the DISPLAY attribute:

- `text` — an ordinary textbox (the STEP attribute is ignored).
- `updown` — an ordinary text accompanied by buddy up-down control for incremental modification.
- `slider` — the slider control.

For example,

```
<NUMBER NAME="TabStep" CAPTION="Tabulation step"
      VALUE = "4" MIN="2" MAX="8" STEP="1" DISPLAY="slider"/>
```

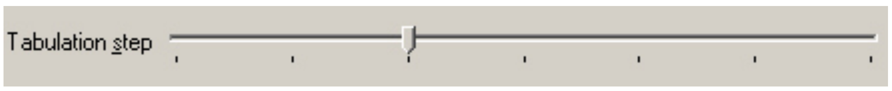
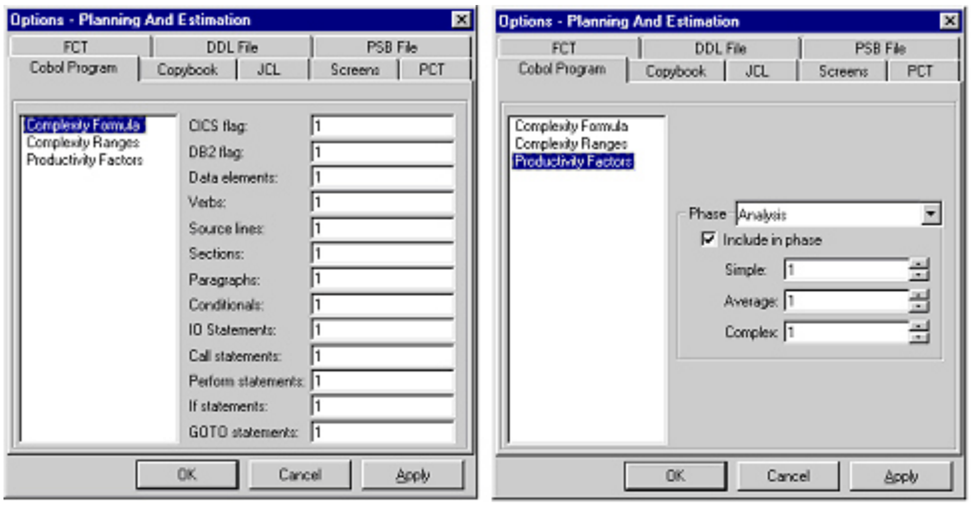


Figure 7. Number as a slider

will be displayed as shown in Figure 7

The usage of `NUMBER` tags, as well as various selection styles, is illustrated in Figure 8.



CICS flag:	1
DB2 flag:	1
Data elements:	1
Verbs:	1
Source lines:	1
Sections:	1
Paragraphs:	1
Conditionals:	1
IO Statements:	1
Call statements:	1
Perform statements:	1
If statements:	1
GOTO statements:	1

Phase: Analysis

[✓] Include in phase

Simple:	1
Average:	1
Complex:	1

Figure 8. Numbers

3.8. File names

The `FILE` tag supports manipulation with file names. It is similar to the `TEXT` tag, but has special prompts when a file name is entered, and an additional button to call up a standard file selection dialog. In order to restrict selection to directories only, one must set the `TYPE` attribute to `dir`. For example,

```
<FILE NAME="Target directory"
      CAPTION="Target directory"
      VALUE="C:\Relativity.Inc\Samples"/>
```

is displayed as shown in Figure 9.

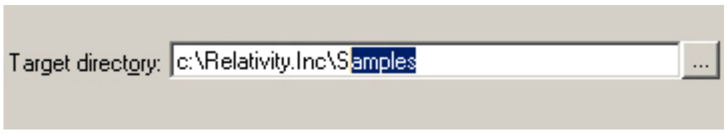

Target directory: c:\Relativity.Inc\Samples

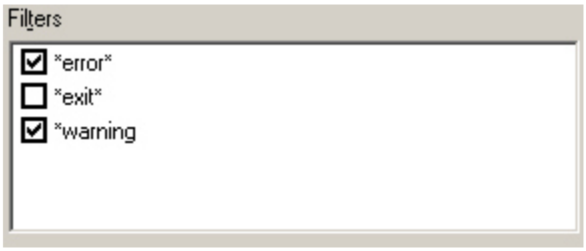
Figure 9. Files and Directories

3.9. Strings

Quite often an option is a set of strings. This capability is provided by the `STRINGS` tag. If the `STYLE` attribute equals to `check`, then a check box appears for extra selection. Each string is represented by a `STRING` tag with two attributes: `VALUE` for the string itself and `CHECKED` to indicate whether the string is selected. For example,

```
<STRINGS NAME="Filter" CAPTION="Filters" STYLE="check">
      <STRING VALUE="*exit*" CHECKED="false"/>
      <STRING VALUE="*error*" CHECKED="true"/>
      <STRING VALUE="*warning*" CHECKED="true"/>
</STRINGS>
```

is displayed as shown in Figure 10.


Filters

☑ *error*

☐ *exit*

☑ *warning

Figure 10. Strings

The user can add a new string by pressing Insert, delete the selected string by pressing Delete, or edit the selected string by pressing F2. The user may also click the right button and access the same actions from the popup menu. As usual, the user may left-click on the selected item to start editing.

The `ORDER` attribute controls list sorting: `none` — for no sorting, `asc` — for ascending, and `desc` — for descending. If the string list has been modified, the user may click F5 to restore the sorting.

3.10. Groups

The `GROUP` tag serves primarily for display purposes: it shows a number of elements on a common frame. For example,

```
<GROUP NAME="Controls" CAPTION="Controls attributes">
        <FONT NAME="LabelFont" CAPTION="Labels"
              FONTNAME="MS Sans Serif"
              SIZE="8"
              BOLD="false"
              BACKCOLOR="-2147483633"
              SAMPLE = "Label1"
              SELECT="font,fore,back"/>
        <FONT NAME="TextBoxFont" CAPTION="TextBoxes"
              FONTNAME="MS Sans Serif"
              SIZE="8"
              BOLD="false"
              SAMPLE = "Text1"
              SELECT="font,fore,back" />
        <FONT NAME="ComboBoxFont" CAPTION="Comboboxes"
              FONTNAME="MS Sans Serif"
              SIZE="8"
              BOLD="false"
              SAMPLE = "Combo1"
              SELECT="font,fore,back" />
</GROUP>
```

is displayed as shown in Figure 11.

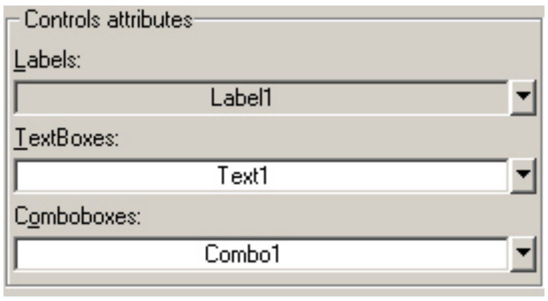


Figure 11. Groups

A `GROUP` element may have the `DISPLAY` attribute that is set to `frame` by default. If the value of the attribute equals `none`, the dialog will not have the surrounding frame. The `PLACE` attribute controls the placement of grouped options: `vert` (default) for vertical placement and `horz` — for horizontal. For example,

```
<GROUP NAME="Connections" CAPTION="Connections"
      DISPLAY="none" PLACE="horz">
  <BOOL NAME="ShowGoto" CAPTION="Show Goto" VALUE="false">
    <COLOR NAME="GotoColor" CAPTION="Goto Color"
           VALUE="16777023"/>
  </BOOL>
  <BOOL NAME="ShowPerform" CAPTION="Show Performs's"
        VALUE="true">
    <COLOR NAME="PerformColor" CAPTION="Perform Color"
            VALUE="16728063"/>
  </BOOL>
</GROUP>
```

will be displayed as shown in Figure 12.

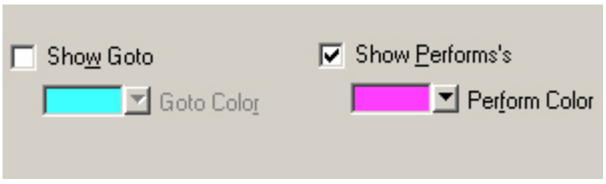


Figure 12. Horizontal groups

3.11. Sections (overall structure)

One file can contain descriptions of several options dialogs called sections. The root tag for the whole document is `OPTIONS` containing `CASE` options, one for each section. For example,

```
<OPTIONS>
  <CASE NAME="RescueWare" CAPTION="RescueWare">
    <LIST NAME="RW" CAPTION="RescueWare"
      DISPLAY="page" VALUE="Editor">
        ...
    </LIST>
  </CASE>
  <CASE NAME="Planning" CAPTION="Planning And Estimation">
    <LIST NAME="Source Type" CAPTION="Source Type"
        DISPLAY="page" VALUE="COBOL">
        ...
      </LIST>
    </CASE>
    <CASE NAME="Methods" CAPTION="Methods">
      <LIST NAME="TYPE" CAPTION="Method Type"
          DISPLAY="page" VALUE="Verify">
        ...
      </LIST>
    </CASE>
  <OPTIONS>
```

4. Advanced display control

In this section, we consider special means for display beautifying. Some elements in the options structure are used for the only purpose of decorating the displayed forms. Special attributes make the resulting dialogs more compact and convenient to use. Finally, we describe how the user-defined behavior can extend Dialogic.

4.1. Icons

The `CASE` tag allows for the `ICON` attribute. Depending on how the case is displayed, the icon is placed either on a list element, or a tab, or a combo item. In order to enable icons, the `DialogicOptions` class is extended with the property `DefaultIconDir` that specifies the root path for the icon location. So, if the property is set as

```
ProjectOptions.DefaultIconDir = GetRescueDir,
```

then the case

```
<CASE NAME="COBOL" ICON="Model\Repository\Icons\Cobol.gif">
```

will be displayed with the icon that is used for Cobol throughout the RescueWare system.

4.2. Tooltips

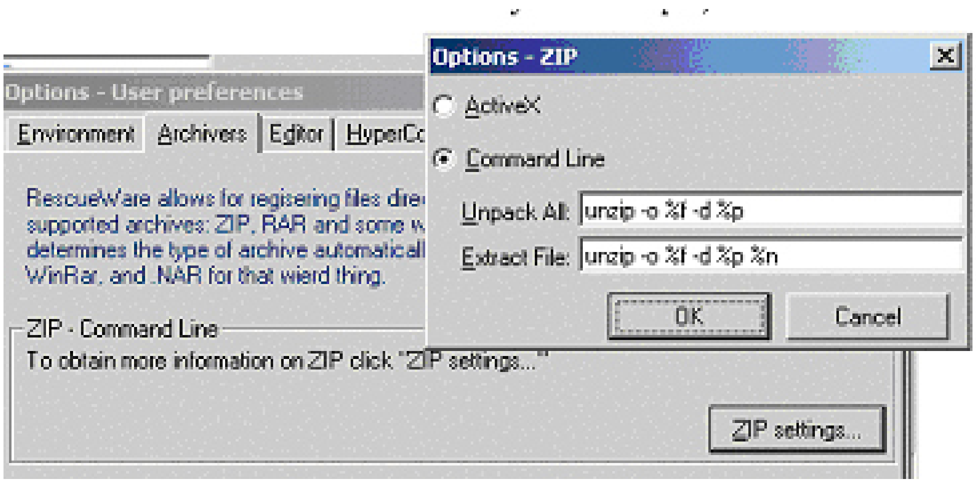
Practically all tags can be decorated with the `TOOLTIP` attribute. If the attribute is specified, the standard Windows tool tip will be displayed for the corresponding displayed element.

4.3. Labels

The dialogs constructed by Dialogic may be decorated with arbitrary texts. The tag `LABEL` has a single attribute `CAPTION`. The Dialogic attempts to reserve the minimal number of lines for displaying the text.

4.4. Details

In a situation when a part of options takes a lot of space but is not frequently accessed, it is reasonable to separate access to this part via a separate dialog. The `DETAILS` attribute permits that for `BOOL`, `LIST` and `GROUP` tags. If the `DETAILS` attribute is specified, the element is displayed as a frame with a button in the right-bottom corner. The caption of the button equals to the value of the `DETAILS` attribute. The caption of the frame starts with the caption of the element and may be extended: for `BOOL` with `Off` or `On` depending on the value of the element, and for `LIST`, whose attribute TYPE is equal to `data`, with the caption of the selected case.



RescueWare allows for regisering files dire
supported archives: ZIP, RAR and some w
determines the type of archive automaticall
WinRar, and .NAR for that wierd thing.

To obtain more information on ZIP click "ZIP settings..."

ActiveX [✓ Command Line]

Unpack All: unzip -o %f -d %p

Extract File: unzip -o %f -d %p %n

Figure 13. Separated details

4.5. Availability

Sometimes it is necessary to restrict the user's access to an option by either disabling its modification or hiding it completely. The attributes `ENABLE` and `VISIBLE` serve this purpose.

4.6. User-defined rendering

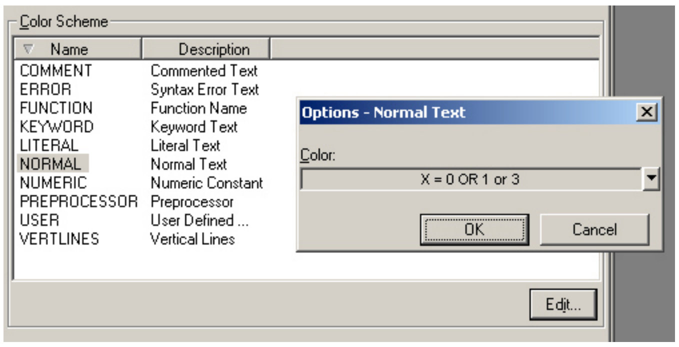
Despite the variety of display options, some elements require a special treatment, for example, to have lists with several columns, or specially edited texts, etc. This becomes crucial when the content depends on dynamically accessed data like HyperCode or Workspace. The idea is to attach a user-defined class to the element, so that it will perform all the rendering and event processing. However, for technical reasons, we cannot properly support an arbitrary ActiveX control and have limited ourselves to a predefined set of controls.

Practically any element may have a specified attribute `CLASS`. The value of this attribute must be a registered ActiveX class with an appropriate interface. If the element has both `CLASS` and `DETAILS` attributes specified, it will be displayed according to the `CLASS` and accompanied with an additional button to allow fancy editing. The interface of the class must define the following methods:

- `ControlType` — determines the type of control that will represent the element. Presently it can be equal to `"list"`, `"text"`, `"area"`, `"combo"`, and `"bool"` corresponding to the ListView, TextBox, RichText, ImageCombo, and CheckBox, respectively. The set of these controls may be easily extended on the developers demand. The `ControlType` may differ from the type of displayed element. For example, you may display a number as a list (of binary digits) if you wish.
- `MinWidth`, `MinHeight` — determines the minimum width and height required for the control.
- `Flexible` — determines whether the height of the control may be changed while formatting.
- `LoadControl(aParentForm, aDispayedObject, aOptionElement)` — the action to be performed when the control is placed on the dialog. The dialog form is passed through the `aParentForm` argument, `aDisplayedObject` is the reference to the already loaded control of the type specified by `ControlType`, and `aOptionElement` is the reference to the element to be rendered.
- `Render` — the method is called when the dialog needs to (re)display the element.
- `ProcessEvent(EventName,arg)` — the method specifies reaction to the loaded control. `EventName` may be equal to `"DblClick"`, `"ColumnClick"`, etc., and `arg` is an additional argument for the event.
- `Edit` — the method is called when the element has specified `DETAILS`.

- `Terminate` — any additional actions to be performed when the dialog is unloaded.

For example, the dialog that assigns colors to the source language keywords may look like it is shown in Figure 14.



Color Scheme

Name	Description
COMMENT	Commented Text
ERROR	Syntax Error Text
FUNCTION	Function Name
KEYWORD	Keyword Text
LITERAL	Literal Text
NORMAL	Normal Text
NUMERIC	Numeric Constant
PREPROCESSOR	Preprocessor
USER	User Defined ...
VERTLINES	Vertical Lines

Options - Normal Text

Color: X = 0 OR 1 or 3

Figure 14. User-defined rendering

The method

```
Public Function DisplaySubOptions(xn As IXMLDOMElement)
        As Boolean
```

of the object corresponding to the dialog form may be used in user-defined classes to edit some of the options in the standard Dialogic way. The dialog form object is passed to the user class via the LoadControl method.

5. Generating dialogs

The dialog construction algorithm takes the structure to be displayed and the desired sizes — width and height — of the resulting dialog. The desired sizes may be extended if they are not sufficient for displaying the content. The algorithm is implemented as a two-phase recursive procedure. At the first phase, we evaluate minimum sizes for all displayed elements. For example, for a selection list whose `DISPLAY` attribute is equal to `LIST`, the minimum width is equal to the sum of the maximum of all captions of cases and the maximum width of all cases extended with appropriate border widths. The second phase of the formatting algorithm extends, if necessary, the minimal sizes, loads the corresponding controls, and places them in proper positions on the dialog form.

There is a number of tricky cases in that evaluation. One of them concerns formatting of tab strips and labels. We cannot know the height of space occupied by tabs themselves without knowing the actual width of the control. Since at the first phase of the formatting procedure only the desired width is known, the resulting minimum width might be greater than necessary, because the width of the whole dialog may be extended.

Yet another interesting problem is the automatic assignment of hot keys. The main requirement here can be formulated as follows: hot keys must be different for all controls — visible and enabled — available at the same time. For example, it is not necessary for the hot keys of controls located on different pages to differ.

Let us state the problem formally. Suppose we have a rooted tree. For each node in the tree, the set of admissible colors is specified. It is necessary to choose a color for each node from the set of admissible colors so that none of the node predecessors or siblings has the same color.

The tree is derived from the options hierarchy in such a way that two options can be available at the same time if and only if they are either siblings or one is a predecessor of another. The transformation of the options hierarchy is rather straightforward: all we need to do is to flatten some lists, like radio buttons lists, and Booleans since they do not imply visual hierarchy. Each node in the tree has the set of admissible symbols that is basically the set of characters in the element's caption.

We do not know exactly the problem's complexity, but we suspect that it is NP-complete, since it is a sort of the graph coloring problem. We use a heuristic algorithm that traverses the structure of displayed options top-down and assigns hot keys on each level independently, keeping the set of occupied hot keys for each branch. In fact, the requirement of absence of duplicate hot keys is not completely strict, and the implemented algorithm attempts to minimize the number of coincidences. The implementation of the algorithm has a number of enhancements: whenever possible it selects the first symbol in a word and a letter rather than a digit or a special symbol.

The straightforward top-down method has a significant advantage of preserving the hot keys on the top level and thus keeping the users' habits. Had we used a global method, any alteration of options on the lower level, like changing an element's caption, might have lead to a completely new assignment of hot keys to all options.

6. Processing options

6.1. Upgrade

The problem of upgrade becomes much more simple due to the usage of a strictly typed hierarchy. Suppose the user installs a new version of the product and the set of options of the previous version is different from the new one. The option upgrade procedure takes the template file from the new version and copies the corresponding values from the existing file. So, all obsolete options do not appear in the upgrade file and new options have default values. Normally, the correspondence between the new and old options is induced structurally. However, it may happen that the new version was reorganized so that some of the options were moved to another place in the hierarchy. It is desirable to keep the user-defined values of moved options. For this purpose each element may have the ID attribute unique throughout the whole file. The upgrade procedure exploits this attribute for establishing the correspondence between the options prior to moving down the hierarchy. The ID attribute is also very useful for direct programmatic access to options.

The upgrade procedure is applied to the workspace and project options as soon as they are opened. The user options are upgraded when the user logs into the system. There is no reason to upgrade global options because the user cannot modify them.

Since the option template is available for each type of options, it becomes easy to add the possibility of restoring system defaults uniformly. For this purpose any element may be decorated with the attribute `RESTORABLE`. If the value of this attribute equals "true", then any generated dialog for the element will have the button "Restore" whose function is copying the values from the corresponding sub-tree of the template into the edited option. For example,

```
<CASE ID="BLT.Java" NAME="Java"
    CAPTION="Generate Java" RESTORABLE="true">
```

means that all options for Java generation may be restored by a single click.

6.2. Packaging

Essentially, packaging allows for specifying a subset of options. The whole set of options may be considered as a multidimensional space. There are two such dimensions in RescueWare: lingual and functional. In order to support packaging, each element in the hierarchy of options may have the attribute `PACKAGE`. The lingual dimension indicates what source languages a particular option is relevant to. For example,

```
<CASE ID="Callie" NAME="Callie" CAPTION="Callie"
      PACKAGE="COB,NAT;">
```

says that control flow analysis tool Callie works for COBOL and Natural languages only. The functional dimension reflects availability of some functionality in the package. For example,

```
<CASE NAME="GenerationOptions"
      CAPTION="Generation" PACKAGE=";LT">
```

says that the generation options are relevant only if the customer has purchased the legacy transition (`LT`) functionality of the system.

Having the `PACKAGE` attribute properly set, the task of option packaging consists in removing (sub-trees of) all elements that do not fit the current package specification. Note that no modification of the source code or redesign of dialogs is necessary.

6.3. Consistency with the repository model

Some options are directly related to the repository model. For example, the user may specify the color of boxes for `SCREEN` objects on diagrams, or the file extensions for legacy objects of the COBOL type. The relationship is made explicit by assigning additional semantics to the `CAPTION` attribute: if the value of the attribute is equal to `"!ENTITY"`, the packaging procedure looks up into the repository model and substitutes the value with the caption of the corresponding repository entity. Also the icon attribute is automatically added. For example, the element

```
<CASE NAME="COBOL" CAPTION="!ENTITY">
```

will be transformed to

```
<CASE NAME="COBOL" CAPTION="Cobol Program"
        ICON="Model\Repository\Icons\cobol.gif">
```

So there is no need to support consistency manually whenever changes are made to the repository model.

This was just an example of how a uniform representation of options allows for automation of their maintenance.

7. Conclusion and future work

We have described an approach to parameters management in a complex system. The key points of this approach are uniform representation of the hierarchy of options and automatic generation of dialogs for option editing. The generation method is data-driven in the sense that all formatting and behavior of the resulting dialog is based primarily on the data structure. Still it produces the state of the art dialogs satisfying all corporative standards.

One of the directions of further development is to make formatting procedure more automatic. For example, the choice of the best way to display a number might be based on the set of values: if the set is small enough, then the slider representation is the best choice, otherwise textbox will be more appropriate. Similarly, a combobox representation is not well-suited for lists that have both empty and non-empty cases — in this case a radio button representation would be more appropriate.

8. Related works

The overall description of the RescueWare system can be found in [1].

The implementation of Dialogic is based on Micorsoft ActiveX [2] technology and XML markup language [3].

The idea of automatic derivation of dialogs from data types is far from new, being common for database management systems. However, usually the generated forms are rather trivial: either a grid for editing the whole table, or one plain form for each record type [4].

As an example of elaborate dialog construction, we can mention [5] where dialogs are constructed from a subset of Algol 68 types.

The XForms [6] bridge the gap between HTML forms and data representation in XML. However, the resulting forms are still quite simple because of poor expressiveness of HTML forms.

Microsoft announced XDocs [7], which will generate complex forms based on XML data definition, to appear in the middle of 2003.

Various approaches to adaptive construction of graphical user interfaces are discussed in [9–11, 8].

References

[1] Automated Software Re-engineering / Ed. by A. N. Terekhov, A. A. Terekhov. — Sankt-Petersburg, 2000. (in Russian).

[2] Brown R., Baron W. Chadwick W.D. Designing Solutions with COM+ Technologies. — Microsoft Press, 2000.

[3] Extensible markup language (xml). — `http://www.w3.org/XML/`.

[4] Dobson R. Programming Microsoft Access 2000. — Microsoft Press, 1999.

[5] Bulyonkov M. A., Bulyonkova A. A. Mapping object-oriented data base scheme into DBBB DMS file structure // Problems of Theoretical and Experimental Programming. — Novosibirsk, 1993. — P. 147–157 (in Russian).

[6] Xforms — the next generation of web forms. — `http://www.w3.org/MarkUp/Forms/`.

[7] Microsoft “xdocs”. — `http://www.microsoft.com/office/xdocs/default.asp`.

[8] Eisenstein J., Puerta A. R. Adaptation in automated user-interface design. Intelligent user interfaces // Proc. of IUI'2000. — N.-Y.: ACM Press, 2000. — P. 74–81 (available at `citeseer.nj.nec.com/318525.html`).

[9] Grundy J., Hosking J. Developing adaptable user interfaces for component-based systems // Interacting with Computers. — 2002. — Vol. 14, N 3. — P. 175–194 (available at `citeseer.nj.nec.com/article/grundy00developing.html`).

[10] Lecolinet E. XXL: A dual approach for building user interfaces // Proc. of ACM Sympos. on User Interface Software and Technology, Seattle, USA, Nov. 1996. — P. 99–108 (available at `citeseer.nj.nec.com/lecolinet96xxl.html`).

[11] Schlungbaum E. Individual user interfaces and model-based user interface software tools // Proc. of IUI'1997. — N.-Y.: ACM Press, 1997. — P. 229–232 (available at `citeseer.nj.nec.com/schlungbaum96individual.html`).