Generating a GUI with XML/XSLT

Kristiaan Breuker
Richard Brinkman
Sander Evers
Jeroen van Nieuwenhuizen

April 7, 2000

Chapter 1

Introduction

This document will decribe a way of developing graphical user interfaces by using a universal language. We have developed the language PLUG (which stands for Portable Language for Userinterface Generation). PLUG conforms to the current XML standard. Therefore it is possible to transform a generalised user interface description to program code using an XSLT-transformer.

In chapter 2 we will describe our starting point and specify the goal of our project. In chapter 3 we describe the construction process of PLUG. In chapter 4 we show the design steps that are necessary to translate a general user interface description to a specific development environment. The specifics of the PLUG style sheets are described in chapter 5 for AppBuilder (written by Kristiaan Breuker), chapter 6 for Delphi (written by Richard Brinkman), chapter 7 for Java (written by Sander Evers) and chapter 8 for Tcl/Tk (written by Jeroen van Nieuwenhuizen). The scripts and toolkit are described in chapter 9.1.

We will end with the conclusions in chapter 10.

Chapter 2

Generating GUI code

The idea we started our project with is simple: in the design of an application, separate the GUI from application-specific functionality (which we will simply call *functionality* from now on). Specify the GUI in an independent format, and from this description automatically generate the code for a destination platform: a specific programming language, operating system and widget collection. This way, GUI-dependent code is not intermixed with functionality code, so the choice for a specific platform can be postponed in the development process, or the application can be easily ported to other platforms.

The simplicity of this idea soon vanishes when the following complication comes to mind: the GUI code and the functionality code do not work independently. The functionality code interacts with the user through the GUI, so GUI code and functionality code have to communicate with each other; they have to be linked to each other in some well-defined way. In order to achieve complete independence of the GUI in the functionality code, there should be a fixed interface between them. Extend this line of thought a little further, and we would end up with our own sort of X or Java (in which the interface consists of a virtual machine).

This was not the intention behind our design project, nor did we have enough time for it, so we decided to set our goals differently: from the platform-independent GUI description, we would only generate the static part of the GUI: the code which creates the components and displays them for the first time. The rest of the code — the dynamic part, which is linked to the functionality — should be manually programmed. Both parts are integrated to form the actual GUI.

At this point, we recognized that there exist programming environments such as Visual Café (for Java) or Delphi (for Pascal), which already do exactly the same: generate the static code and let the programmer insert the dynamic code. We did not see this as a problem. In fact, we found it very helpful and made good use of it. Instead of generating the code ourselves, we could generate the files which the programming environment uses to generate code and save a lot of unnecessary work.

Of course, this raises the question if our approach, generating the GUI from a system-independent description, has any added value compared to drawing the GUI in a programming environment. We feel the answer is yes: on several occasions, it is an advantage to have a separate document describing the interface. We illustrate this in section 10.1.

For the description of the GUI, we have used XML. Although this choice was not of our own but of our project coaches, we will explain why it is a good one. XML was designed as a standard for data exchange over the Internet, across various platforms, so it matches the cross-platform nature of our project. For example, one can design the interface on a Mac, do the additional event programming using Linux, and generate code under Windows, without file conversion problems (see also section 10.1). Secondly, we can make use of the XML tools that have already been developed by the Internet community. This way we didn't have to write our own parser and compiler.

Chapter 3

Designing the DTD

3.1 General approach

By means of a Document Type Definition (DTD) one defines a set of constraints under which an XML document is valid. In other words, one defines a formal *language* for a class of XML documents. For the purpose of describing a GUI in a platform-independent way, we defined a language we call PLUG: Portable Language for User-interface Generation. While designing the DTD, we kept two goals in mind:

1. the language should be generic enough to translate to most platforms
2. the language should be expressive enough to describe the most frequently used GUI concepts

The approach we used is the following: we examined four specific platforms for expressing a GUI, and selected the components, attributes and events they had in common (or could be easily simulated using existing concepts). Those concepts were translated into a DTD (defining the syntax of the language), along with a description of their semantics. The four platforms were chosen on a basis of diversity as well as our familiarity with them:

1. Delphi: a programming environment for Pascal to create an application with a GUI for MS Windows
2. Application Builder: a tool to create C++ applications using Motif widgets
3. Java: a platform-independent OO language
4. Tcl/Tk: a platform-independent scripting language

First, GUI *components* (such as text input boxes, buttons, menus) were identified which were similar to all four. We thus found 17 generic components which

could be reasonably mapped to every platform. We put those components in a containment hierarchy, i.e. for every component we defined which components it can contain. To express this in XML, we have mapped each component to an XML element. Its definition and place in the containment hierarchy are specified in the DTD through the `<!ELEMENT>` construct.

Next, for every component a set of *attributes* was identified (such as caption text for a button, number of lines for an input box) using the same criteria. A GUI component attribute maps to an XML attribute and is specified in the DTD through the `<!ATTRIBUTE>` construct. Bearing in mind that all elements had to be translated to programming language concepts, we also introduced a special `id` attribute for all elements, by which they can be referred to in the (event) code.

Finally, we identified a set of *events* for each component, such as click for a button and change for an input box. As mentioned before, we do not generate code for event handling; it has to be programmed manually and integrated with the generated GUI code. Still, we wanted the author of the XML interface description to specify whether or not the application has to respond to certain events. The main reason for this is to indicate our code integration mechanism for which events it has to include manually programmed code (see chapter 4). We accomplish this by mapping each event to a boolean-valued XML attribute. A value of *false* indicates that the application should do nothing with a certain event; *true* indicates that additional event code should be included. To distinguish (GUI) attributes from events in the XML description, we adopted the naming convention that the name of an XML attribute which represents an event has a prefix `on-`, e.g. `onclick` or `onchange`.

3.2 Layout issues

One of the problems in describing an interface in a platform-independent way is defining the layout of components: their position and size. Absolute positioning, i.e. specifying everything in pixel sizes, is not a good option, because pinning down coordinates is far too specific for a general description of an interface: widgets from different platforms have different sizes, font sizes (in pixels) are not known at this stage, widget sizes are sometimes customized by the user of the platform, windows could be resized, etcetera.

Instead, some sort of relative positioning has to be used. To find a solution, we've looked at the layout system which the Java AWT (Abstract Windowing Toolkit) uses. In this system, every container component is associated with a layout manager: an object which lays out every component in the container using a certain policy. Every layout manager has a different policy. For example, BorderLayout can put four components along the four borders of the container, and another one in the middle; GridLayout creates a grid with components of equal size; and FlowLayout lets components flow like text, from left to right, wrapping around the edge.

In the first version of PLUG, the container components `window` and `panel`

have some coarse layout functionality similar to Java layout managers. They have an attribute `layout`, which specifies one of the three possible layout styles: `border`, `horizontal` and `vertical`. Inside a container, you can nest other containers with a different layout style, so a complex layout can be composed with several nested panels. Such a combination of the three layout styles cannot create every possible layout, but it is expressive enough to create a coarse layout sketch of a GUI. In future versions of PLUG, layout options could be refined.

The `border` layout Every component in a container with a `border` layout should specify its position in the container with an attribute `position`. This attribute can take the values `top`, `bottom`, `left`, `right` and `center`. At every position, there can be at most one component, so there can be at most five components in a container with a border layout (of course, one of these components can be another container with more components inside). If there is a component at the position `top` or `bottom`, it sticks to the top or bottom of the container, takes up all the horizontal space, and just the vertical space it needs. If there is a component at the position `left` or `right`, it sticks to the side of the container, takes up all the vertical space, except that needed for components at the top or bottom, and just the horizontal space it needs. If there is a component at the position `center`, it takes up all the remaining space.

The `horizontal` layout In a container with a `horizontal` layout, all components are allocated their needed space, and put next to each other from left to right, in the order in which they appear in the XML document. If the components have a different height, their (vertical) centers are aligned. For containers with this layout, there is another relevant attribute which defines the layout: the `layoutalign` attribute. If it is assigned the value `begin`, the row of components sticks to the left side of the container. With the value `end`, it sticks to the right, and with `center` it stays in the middle.

The `vertical` layout The `vertical` layout is very similar to the `horizontal` layout. Components are put in a column from top to bottom, their horizontal centers are aligned, and `layoutalign` controls whether the column sticks to the top, bottom or middle of the container.

3.3 Fonts and colors

As not all fonts are supported on all platforms, and we didn't want to bother with a difficult font replacement mechanism, we decided to support only three very general types of font-family: serif, sans serif, and monospace (typewriter). It is up to the code generation mechanism or the used GUI library to translate these general types into a specific font-family.

Furthermore, there are three attributes to control font style: bold, italic and underline. Assigning a boolean value to one of these attributes forces the font

to use a specific style (if it can). Otherwise, the font style is undefined and may default to user preferences.

Colors, used as foreground or background of a component, present a similar situation: not all colors are supported everywhere, so we defined a set of (very) basic color names. The code generation mechanism or the GUI library maps this name to a value. If no color is specified, the eventual color is undefined and may default to user preferences.

Both font and color options leave much room for refinement in future versions of PLUG. We hope there will emerge widely used platform-independent standards for specifying font and color.

Chapter 4

Transforming the description using XSLT

To transform a PLUG document into platform-specific files (either actual code or interface description files for a programming environment), we use XSLT 1.0[W3Cb], a W3C recommendation of November 1999. XSLT is an XML-compliant language for describing transformations from one XML document into another. After the source XML document and the XSLT document are parsed, the transformation is performed by an *XSL processor*. This program interprets the transformation rules described in the XSLT document, applies them to the source XML document and hereby creates an output document. Although this document is usually also an XML document, it can as well be an unformatted plain text document. As you will soon notice, we rely heavily on this feature in our project.

4.1 Graphical notation

In order to show how we designed the processes of document transformation, we will first introduce a graphical notation to depict those processes.

- A rectangle with a small fold in it represents a file.
- An ellipse represents a transformation process. This process may have several inputs, which are represented by incoming arrows (pointing from files to the process), and several outputs, represented by outgoing arrows (pointing from the process to files).
- A small stick man represents a developer. A developer produces files; this is indicated from by an arrow from a developer to a file.

A three-letter label in the description of a file indicates the format of a file:

- (XSL) indicates an XSLT document

- (XML) indicates an XML document (which is not also an XSLT document)
- (TXT) indicates a non-XML text document: either a file (containing GUI data) which is used by a specific programming environment, or code in a specific programming language.
- (BIN) indicates a binary, executable program (e.g. a .exe file for Windows, or a .class file for Java)

With font and line style, we indicate how specific a certain file is:

- Italic font indicates that a file is platform-specific. If the entire process is applied to produce GUI code for a different platform, files with italic font have to be changed. Normal font means they don't have to be changed.
- Dashed line style indicates that a file in application-specific. If the entire process is applied to produce GUI code for a different application, files with dashed line style have to be changed. Solid line style means they don't have to be changed.

4.2 High-level design

In chapter 2 we have given a general description of what our transformation process should accomplish: from a platform-independent description (describing the static part of the GUI) and some manually created platform-specific code (describing the dynamic part of the GUI), it has to generate GUI code for a specific platform. This is expressed, using our graphical notation, in fig. 4.1.

To generate a GUI for a specific platform, the developer delivers two documents: a PLUG (XML) document (which can be used again to generate the same GUI on another platform), and a document containing platform-specific event code for that GUI. The format of the latter document is not yet defined. It definitely contains code, which would imply a TXT format, but it also has to be structured in another way, because the developer has to tell the transforming process which code applies to which event. Therefore, we have put a question mark after the format.

These two documents are the input to the transformation process, which we will refine in the next section. For now, this process just "does the magic" and delivers a text document. (In most actual cases, it will probably deliver multiple documents, but we will also ignore that for now.) This text document is either a GUI description file for a specific programming environment like Delphi or code in a programming language like Java. In the former case, the document is processed by the programming environment and will eventually result in compiled code; in the latter, it is simply processed by a compiler.

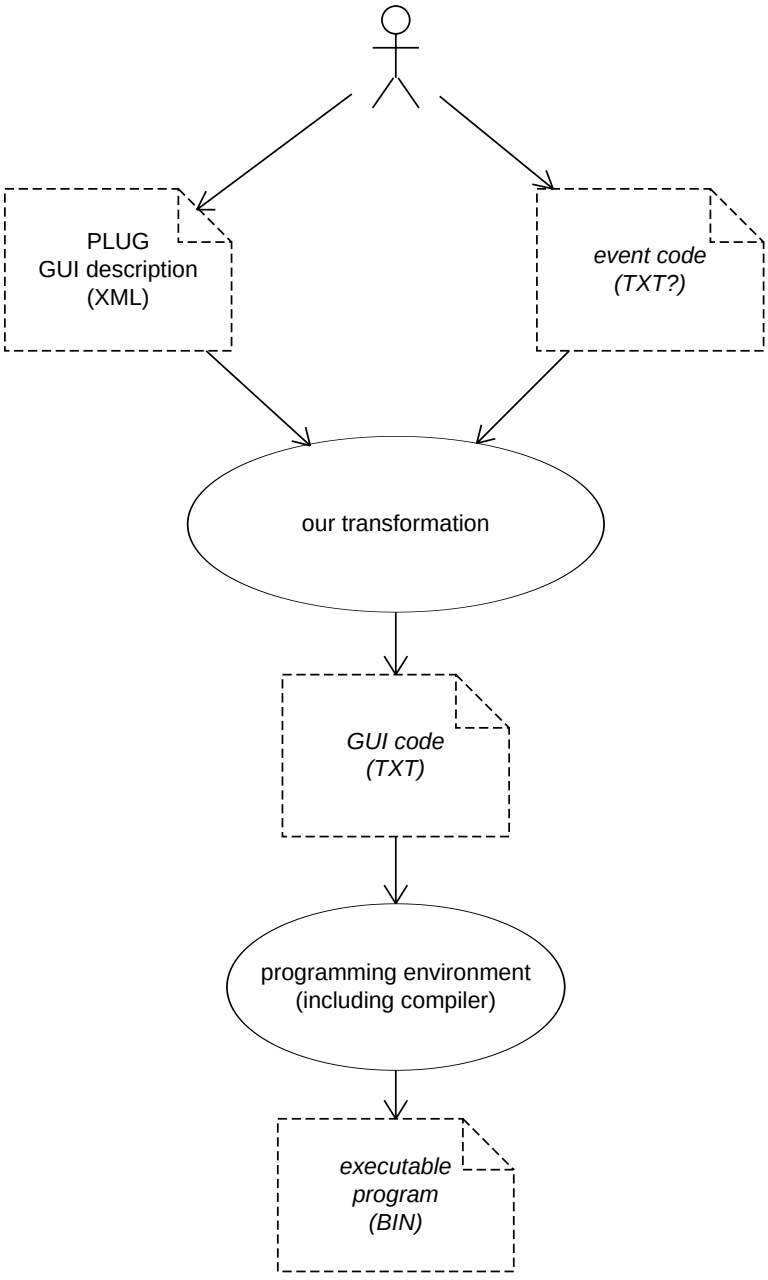


Figure 4.1: High-level design

4.3 Refining the design

The initial design is now refined (and a modified a little). We have already mentioned the reason for the first change: the document containing the developer's event code has to be structured for further processing. As we intend to handle (most) processing with XSLT, it is logical that this document be in XML format. In this way, tags surrounding a text node with event code contain the information required to link this piece of code to a specific component and event type.

It is not desirable to write the complete XML document from scratch. Typing in the correct XML header and tags is a tedious and time-consuming job. It is easy to generate the XML framework for this document from the PLUG description (remember from section 3.1 that this description indicates for which events custom code should be added), so the developer needs only insert code in the specific programming language.

This modification is shown in the upper part of fig. 4.2: taking the PLUG description as an input, a framework (still platform-independent) is created and delivered back to the developer. The developer fills in the platform-specific code to produce the document called "event code".

Next, a refinement is made: all the static GUI code is derived apart from the event code, and put into a separate document. This document also has to contain some meta-information defining where to insert the event code. This would imply XML format, but we've added another question mark. The reason is that it is not immediately clear how to implement the following process - merging the static code and the event code - with two XML documents for input. Of course, we will return to this problem in the following section.

4.4 Implementation using XSLT

The basic building block for our implementation, which is shown in fig. 4.3, is the XSL processor. We have used LotusXSL from IBM Alphaworks, a Java-based XSL processor. To access it from the bash command line, we used a small wrapper class called XSL Converter. It takes three arguments: an input XML filename, an XSLT filename, and an output filename.

The "create event code framework" process is implemented by running XSL Converter with the PLUG document as input, and an XSLT document which transforms it into a code framework like this:

```
<codeframe>
  ...
  <code id="okbutton" event="onclick">
    <!-- You can enter your code here -->
  </code>
  ...
</codeframe>
```

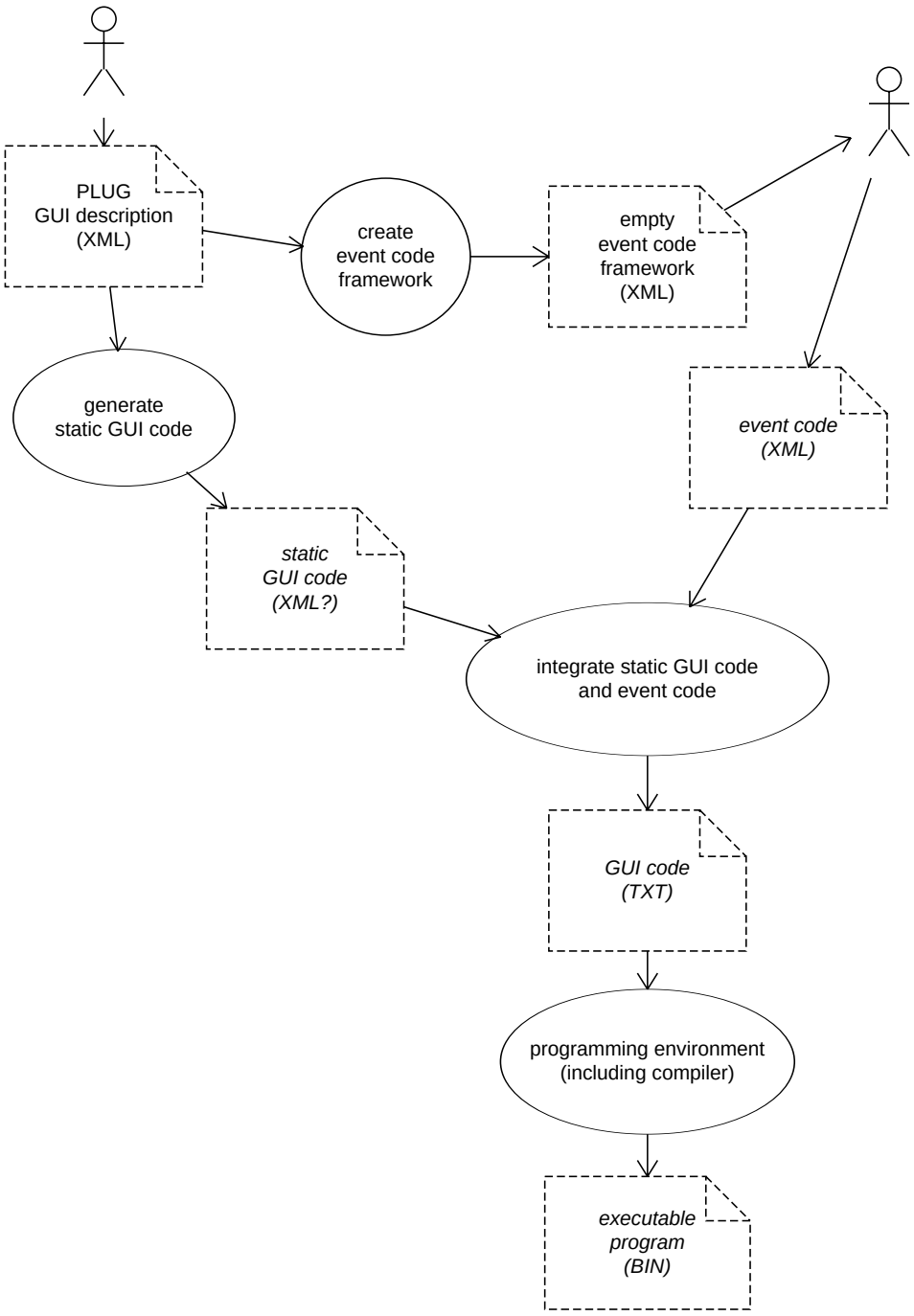


Figure 4.2: Refining the design

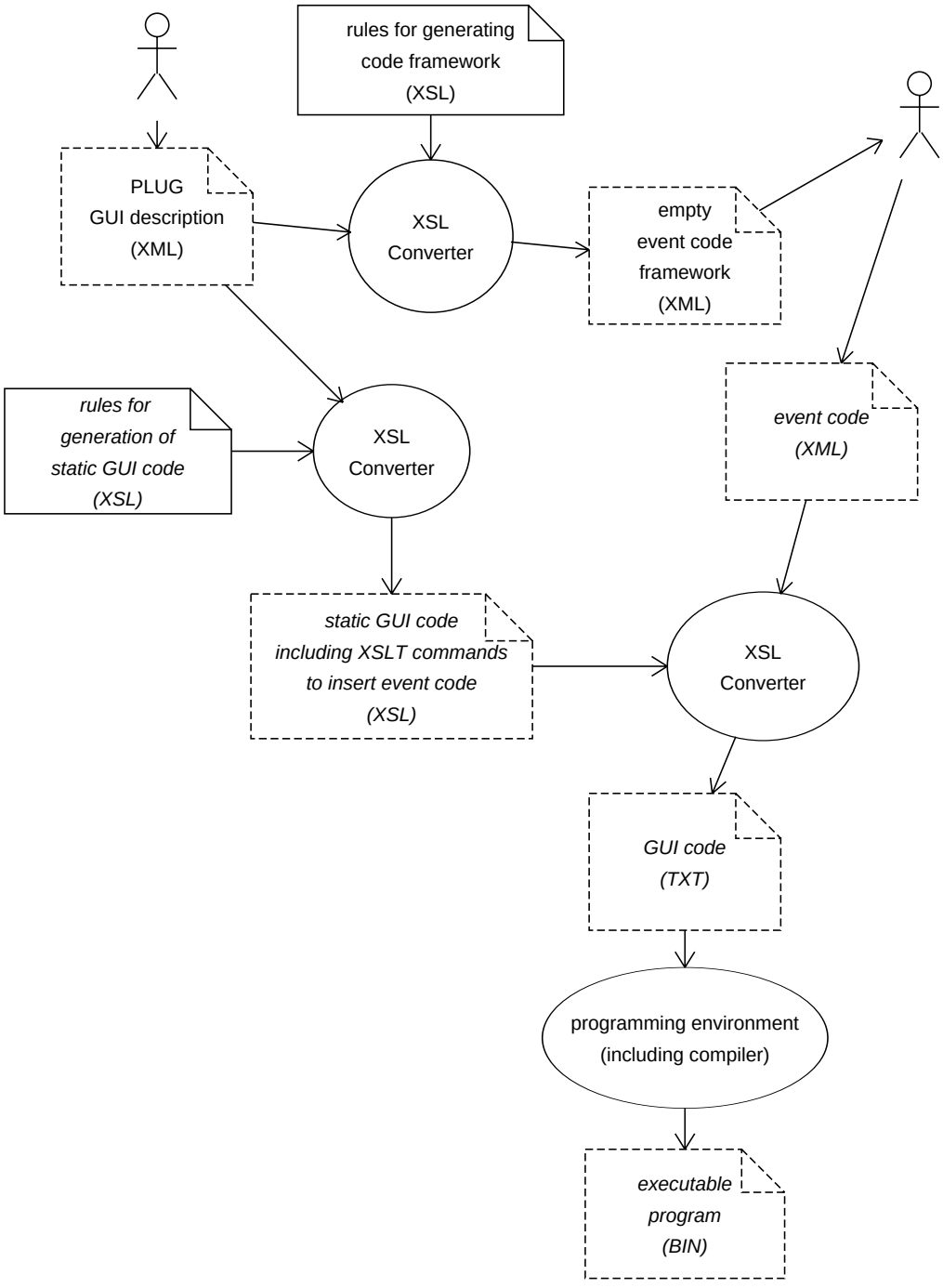


Figure 4.3: Implementation using XSLT

The rules in this XSLT document tell the XSL processor to create a root tag named `<codeframe>` and, for every occurrence of a true-valued event attribute in the input document, create a `<code>` tag with matching id and event name attributes. Note that this XSLT document is platform-independent and application-independent, so it appears with solid line style and normal font.

The "generate static GUI" process is also implemented by processing the PLUG document with an XSLT document. However, the transformation rules of this document are not as easy to describe. They make up the core of our entire code generation process, and are quite different for each platform (hence the italic font in fig. 4.3). For every platform we examined (see section 3.1), we have created such an XSLT document; they are discussed one by one in the next chapters.

The last process to be implemented is the integration of the custom edited event code into the static GUI code. With a little "abuse" of the XSLT mechanism, we also managed to do this using XSL Converter. Although it consists mainly of plain text, we make the output document of the previous process technically *an XSLT document*, with XSLT commands at the places where the custom code is to be inserted. These commands tell the XSL processor to insert code within a specific tag from the input XML document, i.e. the event code from the developer.

4.5 Dealing with multiple output files

The implementation described above would work fine if only one output text file is needed. In practice, all the platforms we examined use more files for the definition of an interface. Typically, there is one file for the whole application, and one file for each window. Because it is not (yet) possible to generate multiple output documents from one input document with XSLT, we have made the decision to split up our input document accordingly: one XML file for the application, and one for each window.

Because the application output file typically tells something about the windows in the application, the application input file must contain references to the window input files. These references cannot be dereferenced during XSLT processing, because XSL Converter can only read from one input file. Therefore, we chose to *include* the window files in the application XML document using the `<!ENTITY ... SYSTEM "...">` mechanism; this way, dereferencing happens during the XML parsing, prior to XSLT processing.

This approach has a drawback: since the window files are included in the middle of the application XML document, they must not begin with an `<?xml>` heading and `<!DOCTYPE>` declaration. Consequently, they are not complete XML documents by themselves. Because we do want to use them as such, we simply concatenate the standard heading and the window description when needed.

All this is shown in fig. 4.4. The processes labeled "transform window" represent (instances of) the entire transformation process as discussed above

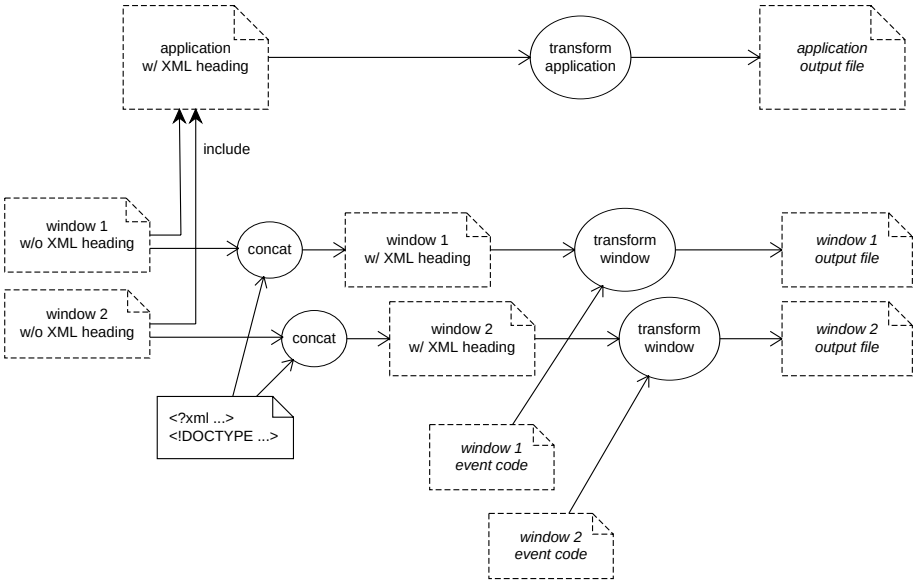


Figure 4.4: Multiple input and output files

(indicated in fig. 4.1 as "our transformation"). Since our PLUG language does not define events on the application itself, the application input file does not have to be integrated with event code, so the process labeled "transform application" does not need an extra input. It is implemented with one XSL Converter process. This process requires another platform-specific XSLT document, which transforms the PLUG description directly into the application output file.

Chapter 5

PLUG for AppBuilder

In this chapter I will describe how a PLUG description of an application is transformed to a number of documents from which CDE's Application Builder (AppBuilder for short) can create a working program.

First I will give a short description of AppBuilder and describe which translations are to be made when creating AppBuilder input documents. I assume that the reader is familiar with CDE and AppBuilder.

Then I will discuss some design problems and AppBuilder shortcomings I came across when creating the XSL documents necessary for the translation of PLUG XML documents to AppBuilder input documents. I will also describe the specific decisions I had to make while implementing PLUG for AppBuilder.

Finally I will point out some improvements that could me made and why they should be made.

5.1 AppBuilder

5.1.1 Introduction

Application Builder (AppBuilder for short) is a program which can be used to create simple applications for the Common Desktop Environment.

Windows can be designed using a graphical interface and code can be added later on, by defining events for the different components used in a window.

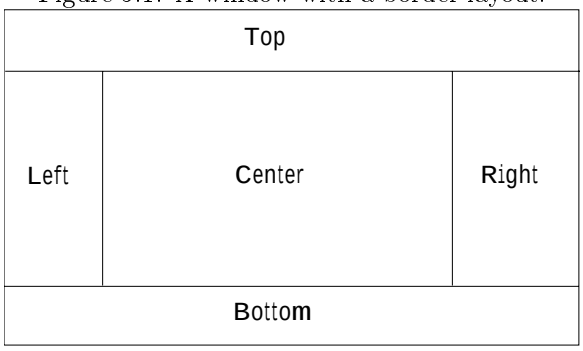
AppBuilder generates the necessary Motif C++ code and makefiles to make a working CDE application.

5.1.2 AppBuilder Input Files

AppBuilder projects consist of two types of files: a *.bip* project file, which contains general project information, and a number of *.bil* module files, which describe each of the project's windows.

The PLUG XML files (an application file and some window files) can be mapped one to one to AppBuilder input files (a project file and some module

Figure 5.1: A window with a border layout.



files). The resulting files can be read by AppBuilder to generate C++ code and make a working application.

5.2 Design issues

5.2.1 AppBuilder Disadvantages

Because AppBuilder is designed to be used by both experienced and non-experienced users, much of the functionality of CDE cannot be implemented using only AppBuilder. A number of attributes from our application and window documents could not be implemented. In some cases I found a workaround, but in most cases these functionalities are not supported by this version of the AppBuilder XSL files (see also section 5.3 on Future Developments).

5.2.2 Layout Issues

We have defined three layout styles, i.e. horizontal, vertical and border layout. I did not have much trouble implementing these layout styles, because although AppBuilder can work with a pixel layout in which all components have an absolute position, it is also possible to define positions relative to other components. This means components are attached to the edge of a window or to another component.

In a horizontal and vertical layout only the first element in the row is attached to the top resp. left edge of the window. The following elements are attached to their preceding elements. All elements are vertically resp. horizontally centered in the window.

The border layout is slightly more complicated. The border layout consists of five areas, which are top, bottom, left, right and center (see figure 5.1). The top element is attached to the top, left and right of the window and the bottom element is attached to the bottom, left and right of the window. The left and right element are attached to the left resp. right edge of the window and to the top and bottom elements, if such elements exist, otherwise to the edge of the

window where no element is defined. The center element is attached to the top, bottom, left and right elements, if such elements exist, otherwise to the edge of the window.

I have tried to minimize the number of panels used, which makes the XSL file a bit complicated, but the output *.bil* file more readable and more efficient. Extra panels are only placed when

- elements are placed on a window when no panel is defined. AppBuilder allows components only to be placed on a control panel, so when no panel is defined, a panel has to be inserted.
- elements are placed in a border layout when no panel is defined. When, for example, an element (which is not a panel) is placed in the top region of a border layout, this element is placed onto a new panel which covers the top region of the border layout.
- a horizontal or vertical layout is defined and the `layoutalign` attribute is set to 'middle'. An extra panel is placed on the existing panel to make it possible to center all the contained elements.
- a popup menu is assigned to an element other than a panel. Because only panels can heva popup menus, an extra panel has to be inserted. This panel will have the same dimensions of the element that is placed on it, so it looks like the element contains the popup menu.

When extra panels are inserted, I use the following naming convension:

- When an extra horizontal or vertical layout panel is inserted, this new panel is called "`@id`_middle".
- When a panel for the top, bottom, left, right or center region of a border layout are inserted, this new panel is called "`@id`_top", "`@id`_bottom", "`@id`_left", "`@id`_right" or "`@id`_center", according to the position of the panel.
- When a popup panel is inserted, this new panel is called "`@id`_popup".

5.2.3 Implementing Common Attributes

Here is an overview of how I have implemented the common attributes and which decisions I had to make to implement them. For element-specific attributes see section 5.2.4 on PLUG Elements Mappings and for the implementation of events see section 5.2.5 on Event Implementation.

- `id`
  The `id` attribute is mapped onto the name attribute all AppBuilder components have. In some cases, for example `panels` or checkable `menuitems`, the actual AppBuilder name can differ from the given `id` in the windows's XML file.

- colors
  In our window DTD ten colors are defined (black, white, blue, green, red, yellow, gray, magenta, cyan and brown). AppBuilder defines colors for components by naming the colors according to our definition, so I had no problem implementing the attributes `foregroundcolor` and `backgroundcolor`.
  In fact, defining colors in a window XML file is more flexible than assigning colors to components in AppBuilder, because for example within PLUG it is possible to color menu items individually, whereas AppBuilder can only color whole menus.

- `state`
  Because all components in AppBuilder have an active property, the `state` attribute is easily implemented by inserting an `active` property which is "true" when "enabled" and "false" when "disabled".

- fonts
  Because AppBuilder has no support for fonts, I have left out the attributes `fontname`, `fontitalic`, `fontbold` and `fontunderline`. I could leave these out, because without these attributes, text is still readable.

- `position` (see also section 5.2.2 on Layout Issues)
  When non-panel elements have a `position` attribute, an extra panel is inserted to place the element on. When a panel has a `position` attribute, the attachments of the panels are set accordingly to it's `position`.

- `popup`
  In AppBuilder, a popup menu can only be assigned to a panel, so a new panel is inserted when this attribute is set on a non-panel element on which the element is placed (see also section 5.2.2 on Layout Issues).

5.2.4 PLUG Element Mappings

In this section I will discuss how I implemented element-specific attributes. I will do this by a per-element description of the attributes of all of the PLUG elements. For implementation of the events, see section 5.2.5 on Event Implementation and for implementation of the common attributes see section 5.2.3 on Implementing Common Attributes.

- **application**
  The `id` attribute is used for the name of the AppBuilder project. The `guiversion` attribute is used to check if the correct version of PLUG is used. The `text` attribute is not used by AppBuilder.

- **window**
  For a window a normal base-window is used.
  The window's `id` attribute is both used for the module's filename and the module's name. A window also has a `guiversion` attribute to check for

the correct version of PLUG. The `text` attribute is used as the window's label property. When `mainwindow` is set to "true", this window is the first window to show up when the application is started and when the mainwindow is closed, the application is shut down. There can only be one main window. Because AppBuilder has no support for windows with an `image` as background, I omitted this attribute. The `layout` and `layoutalign` attributes are implemented according to our specifications (see section 3.2 on Layout styles). When `resizable` is set to "true", the window can be resized, otherwise is cannot be resized. The `visible` attribute is used to set the window's initial visible state by setting the window's `visible` property.

- **button**
  For a button a push-button is used.
  The `text` attribute is used for the button's `label`. Because AppBuilder only supports text or an image to be set as a label, I didn't implement the `image` attribute.

- **statictext**
  For a statictext a string label is used.
  A statictext contains **line**s, which are set as the label's `label` property, individual lines seperated by newline characters (`\n`).

- **input**
  For a one-line input a text-field is used, for a multiline input a text-pane is used.
  I had to split this element up into two different elements: a one-line element and a multiline element. When the `numberoflines` attribute is set to "1" (the default value), a text-field is inserted, otherwise a text-pane in inserted. When `charsonline` is set, the `width` of the text-field/-pane will be set to this value. The `word-wrap` property for a text-pane is set to what the `wordwrap` attribute is set to. Scrollbars will be added to a text-pane according to the `scrollbar` attribute.

- **menubar**
  For a menubar a menu-bar container is used.
  This container contains **menu-item**s.

- **menuitem**
  For a menuitem a item-for-menu is used (or when in a menubar item-for-menubar).
  The `label` of the item-for-menu is set to whatever the `text` attribute is. For the same reason as with the **button**, the `image` attribute is not implemented.
  I had to find a workaround when implementing checkable menuitems, because AppBuilder has no support for them. When the `checkable` attribute is set to "true", not one, but two item-for-menu's are inserted, one with name "`@id`_on" and with an "x" prepended to the `label` and one with

name "`@id_off`". When the `checked` attribute is is set to "true", `@id_on` is visible, otherwise `@id_off` is visible. Some code is added to implement 'check' behaviour. The code for the menuitem's events is added to both inserted menuitems.

- **menuseparator**
  For a menuseparator a item-for-menu is used with label-type set to separator.
  AppBuilder supports multiple line-styles for separators, but because PLUG doesn't support this feature, I just used the standard etched-in `line-style`.

- **menu**
  For a menu an item-for-menu (or when in a menubar item-for-menubar) and a menu are used.
  For a submenu a item-for-menu has to be inserted first. The `id` attribute is given to this menu item and a submenu with name "`@id_menu`" is inserted. The `menu` property of the menu item is set to "`@id_menu`". The `text` and `image` attributes are used in the same way as with the **menuitem**.
  The `tearoff` attribute of the menu is used for the inserted submenu.

- **checkbox**
  For a checkbox a nonexclusive choice is used.
  Because a choice exists of a number of choices, each with it's own name, a choice with only one item-for-choice is inserted. The `id` property is used for the choice's name and the item-for-choice's name is `@id_item`. The choice's `choice-type` property is set to "nonexclusive", which will create a checkbox group.
  The `text` attribute is used for the item-for-choice's `label` property. The `checked` attribute is used for the item-for-choice's `selected` property.

- **radiogroup**
  For a radiogroup an exclusive choice is used.
  The `text` attribute is used as the choice's `label` property. The choice's `choice-type` property is set to "exclusive", to create a radio group.

- **radioitem**
  For a radioitem a item-for-choice is used.
  The `text` attribute is used as the item-for-choice's `label` property. The item-for-choice's `selected` property is set to the radioitem's `checked` property. Only one radioitem in a radiogroup can be 'checked'.

- **popupmenu**
  For a popupmenu a menu is used.
  The `text` attribute is not used. The `tear-off` property of the menu is set to the popupmenu's `tearoff` attribute.

- **list**
  For a list a list is used.

Table 5.1: Panel border-frames

`@bevel`	`@border`	`border-frame`
none	none	flat
none	sunken	etched-in
none	raised	etched-out
sunken	-	shadow-in
raised	-	shadow-out

If `numberoflines` is specified, the `num-rows` property of the list is set to `@numberoflines`, otherwise it is omitted. When the `selectmode` attribute is set to "single", the `selection-mode` of the list is set to "browse" and when `selectmode` is "multiple", `selection-mode` is set to "browse-multiple". I have chosen for the browse selection modes, because I think it makes the resulting list easier to use. With the browse selection modes, a user can press the mouse button on an item and then use the mouse to scroll through the list. When using the non-browse selection modes, a user can only select items by directly clicking on them.
The list items are stored as **line**s within PLUG. AppBuilder requires all items to have a name, so the contained lines are implemented as separate item-for-list elements, each with a name "`@id`_item$n$", where n is replaced by the `position()` of the line within the list. Because items are stored as plain lines, there is no way to have certain items already selected, so the `select` property for all item-for-list elements is set to false. The label of each item is set to the `text()` within the line tag.

- **panel**
  For a panel a container with `container-type` relative is used.
  Because panels in AppBuilder don't have both a bevel and a border property, the `bevel` and `border` attribute are mapped onto the `border-frame` property as described in table 5.1.
  The `layout` and `layoutalign` attributes are implemented according to our specifications (see section 3.2 on Layout styles). Layout panels are inserted when necessary.

- **image**
  For an image a label with label-type image is used.
  In the XML document a source name is specified without the *.gif* extension. The generate script for AppBuilder takes care of the conversion from the GIF format to the XPM format. A label is inserted with a `label-type` "image" and the `label` property set to the specified `source`, with the ".gif" extension appended to it, which in necessary because the image converter creates filenames ending with *.gif.xpm*.

- **progressbar**
  For a progressbar a read-only scale is used.

A scale is inserted and the `read-only` property of the scale is set to "true". The `orientation`, `min-value`, `max-value`, `initial-value` and `show-value` properties of the scale are set to `orientation`, `min`, `max`, `initvalue` and `showvalue` attributes, respectively. When `orientation` is "horizontal", the `direction` property is set to "left-to-right" and when `orientation` is "vertical", `direction` is "bottom-to-top". These are the most natural ways of filling the progressbar. `Increment` is set to 1 and `decimal-points` is set to 0.

- **scale**
  For a scale a scale is used.
  A scale is inserted and the `read-only` property of the scale is set to "false". The `orientation`, `min`, `max`, `initvalue` and `showvalue` attributes and the `direction` and `decimal-points` properties are used in the same way as with the **progressbar**. The `increment` property of the scale is set accordingly to the `incrementvalue` attribute.

5.2.5 Implementing Events

In this section I will describe all events that can be specified in a PLUG document. Events in AppBuilder are defined as connnections between components or from a component to code. There are predefined connections to change the state or text of components, but because PLUG doesn't support those, all has to be done in plain C++ code, so all connections are from an element with action-type 'execute-code'.

Note: When using the quote (") character, you should escape it like this: \".

- `oncreate`, `ondestroy`
  Each of the PLUG elements can have an `oncreate` and an `ondestroy` event. The connection's `when` property is set to "after-create" and "destroy" respectively.

- `onshow`, `onhide`
  A number of elements can have an `onshow` and an `onhide` event. For a show event the connection's `when` property is set to "popup" for a **popupmenu** and to "show" for the rest of the elements. For a hide event the connection's `when` property is set to "hide".

- `onclick`
  A number of elements can have an `onclick` event. For a **checkbox** the `when` property is set to "toggle", for a **menu** to "popup" and for the rest of the elements to "activate".

- `onchange`
  A number of elements can have an `onchange` event. For a **scale** the `when` property is set to "value-changed", for the rest of the elements to "text-changed".

An exception is the **radiogroup** element. For each of the radiogroups' **radioitem**s a connection is made from that radio item with the `when` property set to "activate". This is because there is no such thing as an onchange event for a radio group in AppBuilder. A disadvantage is that the code is executed twice, once for the item being deselected and once for the item being selected, so if the code may only be activated once, be sure to test if the code should be activated, for example by checking the state of the calling item.

- `onselect`
  A **list** can have an `onselect` event. The connections's `when` property is set to "item-selected".

5.3 Future Developments

5.3.1 Produce C++ Code

Because AppBuilder has certain shortcomings (see section 5.2.1), it may be a better idea to produce native Motif C++ code instead of AppBuilder files. This may result in more complicated XSL files, but it will produce much more flexible and efficient code.

In this way, it will be possible to work around the drawbacks of AppBuilder. It would, for example, be possible to create native checkable menu items instead of creating two text menuitems (see 5.2.4 on menu items).

AppBuilder also produces a lot of 'garbage' along with useful program code. This could be implemented much more efficient, making the application only contain code it really uses.

5.3.2 Some Element Improvements

The implementation of some of the (attributes of) elements could be improved.

- It is not possible to have both an image and text on a button or menu item. Now only text is shown, but when only an image is specified, the specified image could be displayed, something which isn't done at this time.

- Both the progressbar and the scale have a property `decimal-points`. This is used to display number with a decimal fraction. The last `decimal-points` digits of the value of the progressbar/scale are displayed as decimal fraction and only integer values can be assigned to a progressbar/scale.
  To support non-integer values a conversion to integer values with a `decimal-points` value could be implemented.

- Popup menus have an attribute `text`, which is to be used the menu's title. This attribute isn't used right now, but could be used in the future.

5.4 Conclusions

With this version of PLUG for AppBuilder, it is possible to generate code for applications which are described using the PLUG XML standard. I had some trouble implementing certain elements and attributes, but I managed to solve those problems to make working version of PLUG for AppBuilder.

As I stated in section 5.3, there are a number of improvements that can be made, such as more efficient code and the implementation of some more attributes, but that may be done in a next version of PLUG for AppBuilder.

Chapter 6

PLUG for Delphi

First I will introduce the specific design decisions that had to be made for the translation from PLUG to a working Delphi 5.0 application. The design decisions are mentioned and an explanation for the choices is given. I assume that the reader is familiair with programming in Delphi.

Finally I will give a summary of improvements that can be made together with some explanation why I think they should be implemented.

6.1 Design Decisions

6.1.1 Layout issues

Unfortunately it is impossible to do all the layout in the Delphi style-sheets. Delphi works with coordinates to position the components. There are no commands to place component `A` beside component `B`. To do that you have to calculate the coordinates by setting `B.Left` to `A.Left+A.Width`. The problem is that the Delphi style-sheet does not know the `Width` of a component, because it is dependent on various aspects, such as fontsize, screen resolution and some other Windows settings. So it impossible to do the whole layout in style-sheets. Although the style-sheets have this drawback, they are not completely useless. In fact the most layout issues are done in the style-sheets.

For each component the following layout properties are set:

Anchors In Delphi each component[1] has an `Anchor` property which is a set of `TAnchorKind = (akTop, akLeft, akRight, akBottom)`. If for instance `akBottom` is in the set `Anchors` than the space between the bottom of the component and the bottom of the window or the panel in which it resides is constant. It is possible to set both `akTop` and `akBottom`. Then the component will grow if the window or panel grows.

[1] Delphi makes difference between a component and a control. I will call them both components from now on.

Tags All components have a `Tag` property. The `Tag` property is not used in Delphi. I will use this property to set the layout (border, horizontal or vertical) and, in the case of horizontal or vertical layout, the layoutalign (begin, middle or end). LayoutUnit (see section 6.1.1.1) will use this value to do the actual layout.

The most significant nibble (= 4 bits) can have the following values:

- 1 = Horizontal layout
- 2 = Vertical layout
- 3 = Border layout
- 4 = For extra inserted panels[2]

The least significant nibble is used for panels and forms with a horizontal or vertical layout and can have the following values:

- 0 = Begin align
- 1 = Middle align
- 2 = End align

Delphi "feature" Delphi likes to make a mess of the order in which components appear[3]. The order in which the components appear in the DFM file[4] will not necessarely be the same as the order in which Delphi places the components on the desktop. All components without a `TabOrder` property[5] are created first and then the other components are created (in the order you have specified). I have worked around this problem setting the `Tag` property of Labels and Images (the only two components without `TabOrder`) to the value that the `TabOrder` property would have had if it existed. In the LayoutUnit (see below) the original order is restored.

6.1.1.1 LayoutUnit.pas

Of course only setting the `Tag` values of panels and forms will do nothing. You have to read the `Tag` values at run-time to decide which layout have to be used for the panel or the form. LayoutUnit.pas, which is automatically added to the Delphi project file (.DPR) and each unit (.PAS), exports three functions:

[2] Extra panels are inserted if the border-layout is used. Every component in a borderlayout is placed in its own panel (with the correct `Align` property).

[3] The people at Borland will call that a feature ...

[4] Delphi separates the layout and the functionality in different files. A DFM-file contains the description of the graphical user interface (which components are used and what is there position on the screen). A PAS-file contains the functionality.

[5] The `TabOrder` property specifies in which order the components are selected when the user presses the TAB-key.

1. `function MinWidth(WinControl: TWinControl): Integer;`
   This function returns the minimal width needed to display the WinControl (= panel or form).

2. `function MinHeight(WinControl: TWinControl): Integer;`
   This function returns the minimal height needed to display the WinControl (= panel or form).

3. `procedure LayoutWinControl(WinControl: TWinControl);`
   This procedure will do all the layout of the WinControl. It will automatically make recursive calls to itself to do the layout of the panels inside the WinControl. So you do not have to call `LayoutWinControl` for each panel. Only for the main panel you have to call this procedure. By the way you do not have to call this procedure yourself, because it is automatically called in the constructor of the window.

6.1.2 Design desicions made for each tag that is defined in PLUG

6.1.2.1 Application

The `id`-attribute is used for the name of the project-file. The value cannot be a Delphi reserved word such as begin, function, ... The `text`-attribute is used for the title that will appear in the Windows taskbar. The `guiversion`-attribute is checked to ensure that only PLUG 1.0 files are being parsed, because future versions of PLUG will have features the current parser has not.

6.1.2.2 Button

All the common options (see section 6.1.3) and the onclick-event (see section 6.1.4) are implemented for the button. The `button`-tag will be mapped on a `TButton` for a button without image and `TBitBtn` for a button with an image. The `text`-attribute is used for setting the button's caption (the text on the button). When an `image`-attribute is given the `Glyph` property will be loaded with the GIF-image.

6.1.2.3 Statictext

All the common options (see section 6.1.3) are implemented for the statictext. The statictext-tag will be mapped on a TLabel if zero or one line-tag is placed inside the statictext-tag. A TEdit is used if more line-tags are inserted. The TEdit will look like a TLabel (no Tabstops, no BorderStyle, buttonface color and not enabled). The height of the TMemo is calculated in the stylesheet. Note that the stylesheet will not be able to calculate the correct height if you use a different font-size.

6.1.2.4 Input

All the common options (see section 6.1.3) and the onchange-event (see section 6.1.4) are implemented for the input. The `input`-tag will be mapped on a `TEdit` if the `numberoflines`-attribute is less than 2 and a `TMemo` otherwise. The height of the `TMemo` is calculated in the stylesheet. Note that the stylesheet will not be able to calculate the correct height if you use a different font size. The `numberofchars`-attribute is not used because there is no restriction of the number of characters for both a `TEdit` and a `TMemo`. The `wordwrap`-attribute and the `scrollbar`-attribute are only used for a `TMemo`.

6.1.2.5 Menubar

All the common options (see section 6.1.3) are implemented for the menubar. The `menubar`-tag will be mapped on a `TMainMenu`.

6.1.2.6 Menuitem

All the common options (see section 6.1.3) are implemented for the menuitem. The `menuitem`-tag will be mapped on a `TMenuItem`. When the `image`-attribute is given then the `Bitmap` property is loaded with the GIF-image. The `checkable`-attribute is being ignored because in Delphi all menuitems are checkable. The `checked`-attribute is used to set the initial state of the `Checked` property. To (re)set the checkmark at run-time you have to change the `Checked` property in the onclick event handler. This will not be done automatically.

6.1.2.7 Menuseparator

The `menuseparator`-tag will be mapped on a `TMenuItem` with `Caption`='-'.

6.1.2.8 Menu

All the common options (see section 6.1.3) are implemented for the menu. The `menu`-tag will be mapped on a `TMenuItem`. The `text`-attribute is used for the caption of the menu. The `image`-attribute will be used to set the `Bitmap` property. The `tearoff`-attribute will be ignored since this feature is not common in Windows.

6.1.2.9 Checkbox

All the common options (see section 6.1.3) are implemented for the checkbox. The `checkbox`-tag will be mapped on a `TCheckbox`. The `text`-attribute is used for the caption of the checkbox. The `checked`-attribute is used for the initial value of the `Checked` property.

6.1.2.10 Radiogroup

All the common options (see section 6.1.3) are implemented for the radiogroup. The `radiogroup`-tag will be mapped on a `TRadioGroup`. The `text`-attribute is used for the `Caption` property of the radiogroup.

6.1.2.11 Radioitem

All the common options (see section 6.1.3) are implemented for the radioitem. The `radioitem`-tag will not be mapped on a TRadioItem. Instead, the `text`-attributes of all radioitems belonging to a radiogroup are used to set the value of the `Items` property of the radiogroup.

6.1.2.12 Popupmenu

All the common options (see section 6.1.3) are implemented for the popupmenu. The `popupmenu`-tag will be mapped on `TPopupMenu`. The `text`-attribute will be ignored because in Windows popupmenus do not have titles.

6.1.2.13 List

All the common options (see section 6.1.3) are implemented for the list. The `list`-tag will be mapped on a `TListBox`. The `numberoflines`-attribute is used to guess the height of the listbox. Only with the normal font the height of the listbox will be correct. The `selectmode`-attribute is used to set the `MultiSelect` property.

6.1.2.14 Panel

All the common options (see section 6.1.3) are implemented for the panel. The `panel`-tag will be mapped on a `TPanel`. The `Autosize` property will be set to `True`, although this is not stricty necessary. There seems to be a Delphi bug regarding the `Autosize` property. If you change sizes at run-time, the `Autosize` feature does not seem to work correctly. So in the LayoutUnit (see section 6.1.1.1) the `Autosize` property of panels will first be set to `False` (to make it possible to change the size) and later put it back to `True`. This is how it works. The `BevelInner` and `BevelOuter` properties are set according to the `bevel`- and `border`-attribute. If the parents `layout`-attribute is set to "border" than the `position`-attribute will be used to set the `Align` property. The `Tag` property gets a value corresponding to his own `layout`- and `layoutalign`-attribute (see section 8.1.3)

6.1.2.15 Image

All the common options (see section 6.1.3) are implemented for the images. The `image`-tag will be mapped on a `TImage`. The `source`-attribute is used in the constructor of the window in which the image resides to read the GIF image from file with the following construction:

Table 6.1: Translation of colors

PLUG	Delphi
black	clBlack
white	clWhite
green	clGreen
red	clRed
yellow	clYellow
gray	clGray
magenta	clPurple
cyan	clAqua
blue	clBlue

```
Image_Id.Picture.LoadFromFile(Image_Source);
```

6.1.2.16 Progressbar

All the common options (see section 6.1.3) are implemented for the progressbar. The `progressbar`-tag will be mapped on a `TProgressbar`. The `min`- and `max`-attributes are used to set the `Min` and `Max` properties. The `initvalue`-attribute is used to set the `Position` property. The `orientation`-attribute is used to set the `Orientation` property. The `showvalue`-attribute is ignored because a `TProgressbar` is not able to display a value in the progressbar.

6.1.2.17 Scale

All the common options (see section 6.1.3) are implemented for the scale. The `scale`-tag will be mapped on a `TTrackBar`. The `min`- and `max`-attribute are used to set the `Min` and `Max` properties. The `initvalue`-attribute is used to set the `Position` property. The `incrementvalue`-attribute is used to set the `Frequency` property. The `orientation`-attribute is used to set the `Orientation` property. The `showvalue`-attribute is being ignored because `TProgressbar` is not able to display a value in the progressbar.

6.1.3 Common options

For each component the following properties are set:

Enabled Enabled will be set to True if the state-attribute is equal to "enabled" and to False if the state-attribute is equal to "disabled".

Color For each component having a Color property the backgroundcolor-attribute is translated to the Windows colors (see table 6.1).

Font For each component having a `Font` property the following properties are set:

- `Font.Color` is the translation of the foregroundcolor-attribute (see table 6.1).
- `Font.Name` is set to "Times New Roman" if fontname="serif", "MS Sans Serif" if fontname="sansserif", "Courier" if fontname="monospace" or left unchanged if no fontname is specified
- `Font.Style` is set according to the fontbold- fontitalic- and fontunderline-attributes

Popup For each component having a `Popup` property, this property is set to the popup-attribute (if present).

6.1.4 Events

Events are used to insert Delphi source code from the (generated) codeframe into the units. Each event is linked to exactly one event handler (and each event handler is linked to exactly one event). The event handler looks like:

```
procedure button1Click(Sender: TObject);
```

The Sender is always the component that caused the event. Note that only the headers are generated automatically. In the codeframe you have to include begin and end;. This to make it possible to declare constants, types and variables. A typical codeframe could look like:

```
<codeframe>
  ...
  <code id="button1" event="onclick">
    var I: Integer;
    begin
      //Do something with the variable I
    end;
  </code>
  ...
</codeframe>
```

oncreate For windows (or forms as they are called in Delphi) oncreate is handled as a normal event (thus setting OnCreate to some method). Note that the oncreate event handler is different from the forms constructor. All other component does not have an OnCreate event. For each component with oncreate="true" a subroutine in the forms constructor will be added with the same functionality as other events.

Table 6.2: Event naming

Delphi component	PLUG events	Delphi events
Form	onshow onhide	OnShow OnHide
Button	onclick	OnClick
Edit	onchange	OnChange
Memo	onchange	OnChange
MenuItem	onclick	OnClick
Menu	onclick	OnClick
RadioGroup	onchange	OnClick
PopupMenu	onshow	OnPopup
List	onselect	OnClick
Trackbar	onchange	OnChange

ondestroy Ondestroy events works like oncreate events.

Other events Not all events specified in PLUG are named equally in Delphi. Take a look at table 6.2 to see the differences.

6.2 Possible Improvements

6.2.1 More efficient generation of code

The Delphi source code is not as efficient as source code written by an experienced programmer. Sometimes panels are added to force layout issues whether it is necessary or not. In a border layout for each component a panel is added. But this is only neccessary for components without an `Align` property. So maybe you could check first if a component has an `Align` property before inserting a new Panel.

6.2.2 Calculation of coordinates in the stylesheets

In this version of PLUG for Delphi all coordinates are calculated in a separate LayoutUnit. The reason for this was that only at run-time the precise sizes of the components are known. This is not exactly true. It should be possible to calculate the width of a component as a function of: the component itself, the font family and the fontsize used, the number of characters. I think that such a function could exist, although not easy to calculate. With fixed width fonts (with fontname "Monospace") this function will be easier to calculate than with proportional fonts. But in the latter case you can approximate the sizes by saying that the size is at most the number of character times the maximal characterwidth (or -height). The big advantage of this method is that the layout is already correct if loaded in the Delphi environment. Now all components are placed on top of each other (all components have Top en Left set to zero).

6.3 Conclusions

PLUG for Delphi version 1.0 serves quite well in generating Delphi Source code, although it is possible to generate more efficient code. The layout can better be done inside a stylesheet to see the correct layout when you load the project in the Delphi environment. Maybe this will be fixed in the next version of PLUG.

Chapter 7

PLUG for Java/Swing

This chapter discusses the mapping of a PLUG description to Java code using Swing GUI components. First, it describes a systematic way of setting up a GUI. Then, the mapping from PLUG elements and attributes to Java components is explained. Finally, suggestions for improvement are given and some conclusions are drawn.

7.1 Setting up a GUI in Java/Swing

In Java, a GUI component is an object, usually an instance of a class from the Swing package (Swing is a part of the Java Foundation Classes). The attributes of a component are part of the state of the object and are accessed through `get()` and `set()` methods on the object. Some components, like `JPanel`, can also contain other components. Those components are added at runtime by means of an `add()` method on the container component.

So, in order to completely set up a GUI component (including its children), the following steps can be carried out:

1. Declare a variable of the class which the component belongs to
2. Create a new instance of the class, and assign it to the variable
3. Set the attributes of the object using `set()` methods
4. For each of its children: carry out steps 1 through 4 for it, then add it to the parent component

And that is (almost) exactly what happens in the Java programs we generate. For each component on a window, a setup procedure is defined named *component-id*`_setup`, where *component-id* is replaced with the value of the `id` attribute of the component in question. This procedure creates the component, sets its attributes, and (if it is a container object) calls the setup procedures of its children. A difference with the steps described above is that the declaration

of the variable happens outside of the setup procedure, so it has a broader scope (I will show later on that this is useful in event handling).

I have not mentioned yet in which classes these setup procedures are defined. Defining a setup method for a component in a class to which the component in question belongs is not a good option, because it would require a new subclass for every single component. Since we regard windows as main units in our design, it seems a good choice to create a subclass of `JFrame` for each window, and in that subclass define all the procedures which together set up the GUI. This way, the only classes that have to be generated for an application are one main class and one for each window, just as we described earlier.

Let's focus on a `JFrame` subclass. When it is instantiated, first an empty `JFrame` instance is created — a window with no contents on it, nothing in the title bar, and some default attribute values. This window is not yet visible. Then, the constructor of the subclass is called. This is in fact the setup procedure for our window, except that the window is already created when it's called. It sets some attributes (like the window title) using `set()` methods inherited from its superclass, then calls the setup procedures of its children (remember, these are defined in the same class), and finally adds those children components to itself.

This last step, adding children to `JFrame`, is a little different from adding children to, say, `JPanel`. This is because `JFrame` *is* not an actual container component, but *has* a container component: its content pane, which is an ordinary `JPanel`. Components that appear on the window, like buttons and input boxes, are actually on this content pane. So I create a new `JPanel`, add the window children components to this panel, and then set the content pane to be this new panel. The only difference is the menu bar: this component does not belong on the content pane. It is separately set by means of a method `setJMenuBar()` in `JFrame`.

As every window in the XML description maps to its own class in Java, the class can be named after the window's `id` attribute. Every class is instantiated exactly once. To be able to refer to a window from other windows, I have used the Singleton design pattern from [Gam95]: a `static` class variable named `singleton` is defined, containing a reference to the one instance of the class. Its value can be obtained using the `static getSingleton()` method.

Fig. 7.1 shows the general framework of a Java class file (defining a window) generated by PLUG. Words in *italic* font are not actual Java code, but a description of code, or an indication that code between brackets is optional or repeated.

7.2 From XML to Java via JML

The XML document describes components with attributes, and how they are contained in each other. This XML document is mapped to a Java class described by the framework above, by the XSLT document. It would be nice if the containment structure in the XML document is the same as the containment structure of the Java Swing classes; this would simplify the description of the

```
class window-id extends JFrame {

   static window-id singleton;
   public static window-id getSingleton()
      { return singleton; };
   static { singleton = new window-id(); }

   public window-id() {
      setTitle("window-title");
      set the other attributes in a similar way
      setup_menubar-id();
      setJMenuBar(menubar-id);
      setup_contentpane-id();
      setContentPane(contentpane-id);
   }

and for every component on the window:
[
   private component-class component-id;
   private void setup_component-id() {
      component-id = new component-class();
      component-id.setText("component-text");
       (and/or other attributes)
      if the component is a container, for each child component:
      [
      setup_child-component-id();
      component-id.add(child-component-id);
      ]
   }   // end setup-procedure
]

}  // end class
```

Figure 7.1: Java class framework for a window

mapping (the XSLT rules).

Unfortunately, the PLUG structure is not exactly the same. For example, in PLUG components are contained directly in the window, while in Java they are on a panel, and this panel is part of the window. Another example: in PLUG, a component optionally has a scrollbar. In Java, such a component is put on a special scroll panel (a `JScrollPane`). The result of this is that an XSLT document which transforms PLUG directly into Java code would contain a lot of extra if...then rules and would be difficult to read.

To avoid this, I have decided to perform the transfromation in two steps: first from PLUG to a new XML language I call JML; then from JML to Java code (or actually, as described in section 4.4, an XSLT document containing a lot of Java code). The idea is that all changes of nomenclature and structure are performed in the first step, so the second step can be a very direct mapping to code. The JML language is therefore designed to be as close as possible to Java code:

1. Every XML element represents a Swing component like `JTextField`, `JPanel`, `JScrollPane`, etc.
2. The containment structure of elements in JML is the same as the containment structure of components in the Java code
3. Attribute (string) values in JML are the same as the Swing attribute values

7.3 Specific mappings

This section describes specific mappings from PLUG to Java, the encountered problems and their solutions. These mappings are mostly performed in the first transformation step. Only interesting mappings are covered; trivial ones as "a `button` maps to a `JButton`, and its `text` attribute maps to a Java `Text` attribute" are left out. If you are interested in those, refer to the XSLT documents themselves (the document that describes the first transformation step is fairly well readable).

7.3.1 Components and component-specific attributes

Most PLUG components have a direct counterpart in Swing. Some components had to be simulated using other Swing components, and especially these are discussed here.

The `window` component is split up into two Swing components: a `JFrame` (carrying the window's id) and a `JPanel` with the id `contentpane`. All of the window's children are put inside the panel, except for the menubar and the popup menus. All attributes, except for `layout`, apply to the `JFrame`.

The `input` component is simply mapped to a `JTextField` if the attribute `numberoflines` has a value of 1. If not, it is mapped to a `JTextArea` with a `JScrollPane` around it (with the id of the `input` concatenated with `_scrollpane`).

The attribute value of `scrollbar` is mapped to the attributes `HorizontalScrollBarPolicy` and `VerticalScrollBarPolicy`. Even if the `input` has no scrollbars specified, the `JTextArea` is still placed on a `JScrollPane` (without scrollbars). If it would not be, it would have the (undesired) behaviour of adjusting its visible size to the number of lines typed in it. A `list` is mapped to a `JList` with a `JScrollPane` around it, for the same reason.

In Swing, there is no component which displays uneditable multi-line text. Therefore, a `statictext` is mapped to a `JPanel` with a vertical layout, which contains multiple `JLabel` components. Attributes like `state`, `position` and `backgroundcolor` apply to the `JPanel`; font attributes and `foregroundcolor` apply to every `JLabel`.

A `menuitem` is mapped to either a `JMenuItem` or a `JCheckableMenuItem`, depending on the value of its `checkable` attribute.

A `radiogroup` is mapped to a `radiopanel` in JML. This component does not exist in Java; in the second transformation it is translated into a `JPanel` with several `JRadioButtons` on it, which are grouped into a `ButtonGroup`. If the `radiogroup` would have been transformed into a `JPanel` in the first transformation, I would have lost the information that the panel's contents had to be grouped into a `ButtonGroup`.

A `popupmenu` is mapped to a `JPopupMenu`. This component has no fixed place on the window, so it is not part of the component containment structure on the contentpane. Its setup procedure is called by the setup procedure (i.e. the constructor) of the `JFrame` it belongs to, but it is not added to any component. Components with a `popup` attribute get assigned a `PopupListener`, a subclass of `MouseAdapter`, which tells the JPopupMenu in question to show itself at the coordinates of the mouse click. The constructor of `PopupListener` gets a reference to the `JPopupMenu` as an argument, so it is important that the menu is already created at the time the event listener is created. Of the entire window, popup menus are therefore the first components to be set up.

7.3.2 Layout

Since the PLUG layout types were based on Java LayoutManagers, the mapping between them is pretty straightforward. A panel with `layout="border"` is managed by a `BorderLayout`, a panel with `layout="horizontal"` by a `BoxLayout` with horizontal orientation, and a panel with `layout="vertical"` by a `BoxLayout` with a vertical orientation. When components are added to a panel with a `BorderLayout`, an extra argument to the `add()` method is derived from the value of the `position` attribute. The `layoutalignment` attribute, specific to `horizontal` and `vertical` layout, results in adding invisible "glue" components to the panel; either after the other components, before them, or both.

Unfortunately, there is one difference between the `BoxLayout` and PLUG's `horizontal` and `vertical` layouts. The latter two preserve the components' original sizes, while a horizontal `BoxLayout` tries to resize the component's height (same goes for vertical and width) to its `MaximumSize` value. It would be best to write our own LayoutManager that does not have this behaviour, but to

reduce work I used a little workaround: I subclassed the component classes for which this behaviour was really undesired (`JTextField` and `JScrollPane`) and overrided their `getMaximumSize()` method to return their `PreferredSize`.

7.3.3 Fonts

There are several PLUG attributes that make up the appearance of a font: `fontname`, `fontsize`, `fontbold`, `fontitalic` and `fontunderline`. In Java, a font has the relevant attributes `Name`, `Style`, and `Size`. The Style attribute is an integer indicating whether the font is bold, italic, both, or plain (underline is not supported). These font attributes can only be set during construction of the object; once created, a `Font` object cannot be changed.

This would not be a problem if the appearance of the font was entirely determined by the PLUG description: for every component, a new `Font` object would be created, immediately taking into account all the attribute values. This is not the case. Font attributes in PLUG can be left out, meaning that platform defaults should be used; platform defaults which are not known when the XSLT transformation is performed. So, when some of a font attributes should use the platform default value, and some are overridden, I have to read the font attributes at runtime, override some of them with our own values, and create a new font object with this set of attributes.

The way it is implemented at the moment, these font changes happen one by one for each attribute, meaning that new objects are created unnecessarily often. This makes the generated Java code a little inefficient. On the other hand, the XSLT code is shorter.

7.3.4 Events

Events in Java are handled by objects known as event listeners: they subscribe themselves to event-producing objects, which notify them when an event takes place. For each event defined in PLUG, such an event listener class is defined using Java's *inner class* construct; an object of this class is created, and subscribed to the component which produces the event. In the event listener class (in the body of the method which handles the specific event), the custom code is inserted.

This way, event code has direct access to all the window's variables (even private ones). Since every GUI component is referenced in a variable of our `JFrame` subclass, easy access to each component is provided. In the custom code, attributes can be set and get just like this:

```
<code id="button1" event="onclick">
   input1.setText("You have just pushed a button.").
</code>
```

The only events treated differently are `oncreate` and `ondestroy`, since creation and destruction of a component are not events in Java. Creation code could

be added inside a constructor, but this would require subclassing a component to add a new constructor. Instead, the custom code is placed directly after a component's attributes and children are set up. Destruction code could be added in a `finalize()` method of a new subclass, but I found that in practice, objects are not even finalized when the program ends. I did not find a satisfactory way to force this destruction, so the `ondestroy` event is currently *not supported* in PLUG for Java.

7.4 Possible improvements

7.4.1 Move JML to a higher level of abstraction

Presently, every Swing component is mapped to a JML element, e.g. a `JButton` corresponds to a `<jbutton>`. For every JML element, there is a separate piece of code in the XSLT document that generates Java source code from JML. These pieces of code are very much alike in structure, but (mainly) class names and attribute names are different. Therefore, it may be well possible to use one general XSLT template for all components, parametrized with class and attribute names. The JML code would then describe components on a higher level of abstraction, and look something like this:

```
<component class="JButton" id="button1">
   <attribute name="Text" type="string" value="Push me!"/>
   <attribute name="Enabled" type="boolean" value="false"/>
</component>
```

A minor drawback with this approach is that semantic checks (on the JML document) like "every button must have a text attribute" cannot be performed anymore by the XML parser. But, since every JML document is generated automatically, it is less likely to contain errors than the user-created PLUG document.

7.4.2 Custom import and declaration statements

In practice, I found out that it is very desirable, if not indispensable, to add custom import statements (at the start of the .java file) and declarations (at the start of the class body). It would be easy to extend PLUG, to make this possible.

7.4.3 Specific PLUG layout managers

As mentioned in section 7.3.2, I used a workaround to make the existing `BoxLayout` manager work as desired. It would be better to write a layout manager which exactly implements the desired behaviour for horizontal and vertical layout. It could have the orientation and alignment parameters as constructor arguments.

7.4.4 Enhancing the `scale` implementation

The PLUG `scale` element is currently mapped to a simple JSlider, which cannot display the numerical value. With some work, it could be mapped to a JPanel with a JSlider and a JLabel on it, and an event listener which updates the label to the value of the slider.

7.5 Conclusions

It is possible and not too complicated to transfrom a PLUG document into a Java/Swing GUI. It is profitable to perform this transformation in two steps. Therefore, two XSLT documents have been constructed which implement the transformation.

There is room for several improvements, which could be made in a next version of PLUG for Java/Swing.

Chapter 8

PLUG for Tcl/Tk

First I will introduce the specific design decisions that had to be made for the translation from PLUG to a working Tcl/Tk 8.0 application. The design decisions are mentioned and an explanation for the choices is given. Simple examples are used to illustrate the ideas. Knowledge of is assumed.

Finally I will give a summary of improvements that can be made together with some explanation why I think they should be implemented.

8.1 Design Decisions

8.1.1 Common attributes.

Fonts The mappings from the font attribute to font-names are the following: serif becomes Helvetica, sansserif becomes Arial and monospace becomes Courier. This choices are arbitrary and better choices are possible. The choice has been made to use the default font of the Tcl/Tk installation when a font is not found. This means that I do not have to check whether a font exists, when it does not Tcl/Tk automatically uses the default font of the installation. When no font attributes are specified Helvetica 10 points is used when present.

state The value of this attribute is given to the -state option of the Tk-widget. When the Tk-widget does not have this option it is ignored. The other possibility, not packing them, isn't as good as ignoring in most cases[1]. Not packing them means that when they are shown later they can occur on other positions as intended by the interface designer (due to the working of the pack command.). So showing them seems less harmful as not showing them.

[1] Of course there are situation where not packing is preferable. To prevend packing in this case the interface must be edited by hand but we advise against this.

8.1.2 Creation of Tcl/Tk code files

The `tcltk.xsl` style-sheet is used by the generate program to create a style-sheet (`window.xsl`, where `window` is the name of the window) for each window and a Tcl/Tk code file `name.tcl` consisting of inclusions of `window.tcl` files, when the name of the application is `name` (see example 1). The `window.tcl` files are created by the integrate script with use of the `window.xsl` style-sheets. This gives a modular construction to the Tcl/Tk code, everything about a window is in one file and the program can be started with `name.tcl`.

Example 1 *A simple example of an application.tcl and two window.tcl files (details not shown).*

Application.tcl:

```
source window1.tcl
source window2.tcl
```

window1.tcl:

```
button .a
pack .a
```

window2.tcl:

```
button .b
pack .b
```

8.1.3 Layout issues

To preserve the name of widgets the choice was made to generate always all frame-widgets that are necessary for a correct function of a layout style. When this isn't done the possibility exist that by adding a element to a panel or window may cause a already existing button to change it's name (For example .middle.button1 becomes .middle.left.button1). This possible name change has a negative impact on the reusibility of code any good. So the choose was made to focus on the reusibility of code instead of efficiency[2]. In the remainder of the text I use the ˜ to refer to the widget the element is in.

8.1.3.1 Border-layout

To be able to use the borderlayout with only the pack command some extra panels have to be added to a window of panel with the border layout. The reason is that the pack does not have a center option. The following panels are created and packed to simulate a borderlayout:

1. A top frame (˜.top). This frame should be on the top of it's parent so no side option is given (the default is top) to pack. An element in it should use the whole width so the fill option of the pack command is set to x to

[2]It seems that this choice isn't necessary see section 8.3.2 on page 53 about possible use of variables.

let it occupy the whole width. An element in this frame should have the fill option set to x and expand set to yes. It will then occupy the whole width even when the window or panel is resized.

2. A bottom frame (~.bottom). This frame should be on the bottom so the side option of pack is set to bottom. It should also occupy the whole width so the fill option is also set to x. An element in this frame should have the fill option set to x, the expand option to yes and the side option set to bottom.

3. A middle frame (~.middle). This frame contains the center region which should occupy the remaining space, so the expand option is set to yes and the fill option is set to both. This frame does not contain any elements except the following frames.

4. A middle.left frame (~.middle.left). This frame should use the remaining space in the vertical direction so fill is set to both[3] and the side option is set to left because it should be on the left. An element in this frame should have set the expand option to yes, the fill option set to both and the side option to left.

5. A middle.center frame (~.middle.center). This frame should use the remaining space in the vertical direction so fill is set to both[3] and the side option is set to right because it should be on the left. An element in this frame should have the expand option set to yes and the fill option set to both, it should expand on a resize in both directions.

6. A middle.right frame (~.middle.right). This frame should occupy the remaining space so fill is set to both and expand is set to yes. An element in this frame should have the expand option set to yes, the fill option set to both and the side option set to right.

8.1.3.2 Vertical-layout

To be able to implement the vertical layout it is necessary to put a frame called ~.vpanel on the panel or window with the vertical layout. The panel is as big as the elements in it. When the layoutalign attribute has the value begin then it's packed at the top. When the layoutalign attribute has the value middle it's placed on the left (automatically centered by Tcl/Tk). When the layoutalign attribute has the value true it's placed at the bottom. All elements that are placed in the ~.vpanel should be placed at the top. Because the ~.vpanel sticks to the bottom, left or top the elements in it are at the right position.

8.1.3.3 Horizontal-layout

To be able to implement this it is necessary to put a frame called ~.hpanel on the panel or window with the vertical layout. The panel is as big as the elements

[3]Should be y, but both have the same effect because expand is set to no.

in it. When the layoutalign attribute has the value begin then it's packed at the left. When the layoutalign attribute has the value middle it's placed on the top (automatically centered by Tcl/Tk). When the layoutalign attribute has the value true it's placed at the right. All elements that are placed in the ~.hpanel should be placed at the left. Because the ~.hpanel sticks to the left, top or right the elements in it are at the right position.

8.1.4 Events

oncreate Tcl/Tk doesn't have a oncreate event but it's easy to implement this if you look when a oncreate should occur: when a element is created(=packed). So I just call the oncreate event just after the pack command which puts (creates) the element at its place. And the oncreate event is handled[4].

Example 2 *A simple oncreate example.*

```
button .vpanel.button1 -text "Ok"
pack .vpanel.button1 -side top
button1oncreate
```

ondestroy Tcl/Tk has a ondestroy event called Destroy. The only thing I have to do is to bind the code to the destroy procedure which has the name element-id ++ ondestroy. This means that a button with id="button1" will be bind to button1ondestroy.

onchange (Radiogroup) This event does not exist in Tcl/Tk. It is simulated by binding a onclick to the radioitems. This indicates that a onchange event may have occured when a radioitem is clicked. To accomplish this an onchange, containg the value of onchange, and a proc variable, containing the name of the omchange procedure, if any, are passed to the radioitem template.

onchange (Input) There is no onchange event in Tcl/Tk. For this reason I use the Onleave event to simulate the onchange event. This indicates that when a onchange event (in fact a Onleave event) occurs in the Tcl/Tk code there may be a change. The code of the user should check whether a change has occured or not.

onselect There is no onselect even in Tcl/Tk. For this reason I bind a left mouse button click to the window. To select an item you have to click the mouse-button which means that there is also a mouse-click event when a should occur. For this reason it's logical to use the mouse click event to simulate a onselect event. The onselect event has now the meaning that a selection or deselection may have occured. The user code should handle this.

[4]Simple but effective.

onclick The onclick event is standard binded to the -command option. This means that given the name of the onclick procedure to the -command option will give the desired behavior.

onclick (Menubutton) The menubutton already has a standard onclick event: opening the associated window. To give the possibility to use an onclick event I bind an extra onclick event to the menubutton. When the menu is mapped to a cascading entry the name of the onclick procedure is given to the -command option of the menu add command of the parent menu.

onshow The onshow event is binded to the Map event of Tcl/Tk.

8.1.5 Input

The input tag which logical would be mapped to the text-widget is first placed on a panel and then mapped to the text-widget. The reason for this is that I might want to use scrollbars which in Tcl/Tk are not part of the text widget but are "stand-alone". When the value of the scrollbar attribute is something else as none then appropriate scrollbar-widgets must be created and connected to the text-widget. When these scrollbars are not placed on a panel with the text-widget, problems will arise when the window is resized.

The charsonline and numberoflines attributes are ignored[5] in case of a borderlayout when other elements demand that the horizontal or vertical space should be bigger as specified by these attributes. This is necessary to conform to the borderlayout standard.

8.1.6 Radiogroup

There is no radiogroup in Tcl/Tk. Radioitems are grouped together by a common variable. Therefore there will be created a frame for the Radiogroup. This frames contains a frame which will contain the radioitems. The command variable is passed by the radiogroup template to the radioitems template.

8.1.7 Progressbar

In Tcl/Tk there is no widget to map a progressbar to. To simulate a progressbar code is generated to simulate one with the help of coloring a canvas-widget.

8.1.8 Window

The visible attribute of the window tag has no function when the mainwindow attribute has the value true. On some windowing systems not ignoring it may cause problems to ever see it and Tcl/Tk is portable to many windowing systems, which means I also must generated portable Tcl/Tk code.

[5] In run time due to the fact that the expand option may be yes and the fill option may be x, y or both

The window may contain frames to implement the behavior of the desired layout (see section 8.1.3 on page 44).

8.2 Implementation of tags and their attributes

8.2.1 Common Options

backgroundcolor and foregroundcolor These attributes are used to set the -bg and -fg options of the Tk-widgets. The value of the attribute is copied which means that in fact PLUG for Tcl/Tk can use undocumented colors as specified in rgb.txt[6].

Fonts Font attributes are set with the -font { fontname size otheroptions } of the Tk-widgets. The default it is set to -font {Helvetica 10}.

popup A popup menu should be showed when the element(s) it belongs to is/are right clicked. So I use the bind statement for the element to execute the tk_popup command with the name of the popup menu and the mouse position as parameters.

state When a Tk-widget has a -state option the value of the state attribute is used as value for this option.

8.2.2 Application

The text attribute of the application is ignored. The reason for ignoring the text attribute is simple the name of the main window is used for the naming of the task-bar and program-list entries.

8.2.3 Button

A button tag will be mapped to the button-widget.

image To be able to use an image it should be created first. There for there the image will be created before the creation of the button. The image is given to a variable called @id ++ img. After this the button is created with the -image option set to $@id ++ img.

Example 3 *A simple example illustrating the code generated to display a image on a button.*

```
set button1img [image create photo -file image.gif ]
button .hpanel.button1 -image $button1img
pack .hpanel.button1 -side left
```

[6] You should not use them if you want your application to be portable to different GUI-platforms.

8.2.4 Checkbox

A checkbox tag is mapped to the checkbutton-widget.

checked The checkbutton doesn't have a option to set the checked option direct. But by Tk a variable is associated with the checkbutton-widget. So by setting this variable to 1 if checked is true and 0 is checked is false it's possible to implement this feature.

8.2.5 Image

The image will be displayed using a canvas-widget. The first thing that has to be done is the creation of an image. After that a canvas widget is created with the -width and -height options set to the width and height of the image. When it is resized the canvas will change size the image won't. The image stays in the left upper corner. The third step is packing the canvas to its "parent". After that a the create image command of the canvas is called with the image position in the left upper-corner.

8.2.6 Input

charsonline This option is used to specify the -width option of the text-widget.

numberoflines This option is used to specify the -height option of the text-widget.

8.2.7 List

The list tag will be mapped to a listbox-widget.

numberoflines The numberoflines attribute specifies the number of lines that should be shown, but this attribute is ignored[5] in case of a borderlayout when other elements demand that the vertical space should be bigger. It sets the -height option of the listbox-widget.

selectmode This attribute is copied to the -selectmode option of the listbox-widget.

8.2.8 Menu

A menu is implemented by creating a menubutton when the menu is direct on the menubar. After this a menu is associated with the menubutton. When it's not direct on the menubar a cascading menu entry is added to the parent menu or popupmenu.

image The image is handled in a similar fashion as the image attribute of button (page: 48).

tearoff The value of the tearoff attribute is passed to the -tearoff option of the menu.

text The value of the text attribute is passed to the -text option of the menu.

8.2.9 Menubar

There is no menubar in Tcl/Tk. The standard in Tcl/Tk is to use a frame for this. I conform to this standard.

8.2.10 Menuitem

The menuitem is mapped to a button or checkbutton when it's direct on the menubar. Otherwise it will be created with the `add command` or `add checkbutton` command. The choice between button and checkbutton is based on the value of the checkable attribute. The same goes up for the choice between add command and add checkbutton.

checked This attribute indicates whether the checkbutton should be checked or not. This is implemented by setting the associated variable of the menuitem to 1, if the value of checked is true, or 0, if the value of checked is false. The setting of the associated variable is done after the packing of the element.

image The image is handled in a similar fashion as the image attribute of button (page: 48).

text The value of the text attribute is passed to the -label option in case of a item not direct on the menubar and to the -text option when the menuitem is direct on the menubar.

8.2.11 Menuseparator

The creation of a menuseparator is straight forward. I just do a call to the `add separator` command of the menu the menuseparator is in.

8.2.12 Panel

A panel is in Tcl/Tk a frame-widget and therefore a panel is mapped to a frame-widget. This frame-widget may contain other frames for implementing the layout (see section 8.1.3 on page 44). A frame with no components isn't visible in Tcl/Tk.

bevel This attribute is mapped to the -relief option of the panel-widget.

border The panel-widget doesn't have a possibility to set the border to raised, sunken or flat. There for the choice is made to simulate this by giving a flat border the borderwidth of 0 (-bd 0) and the other borders a bd of 2 (-bd 2).

8.2.13 Popupmenu

A popupmenu is created with the help of a popupmenu.

tearoff The tearoff attribute is used to set the -tearoff option of the menu-widget.

text The text attribute, which is meant to set the title of the popupmenu, is mapped to a the -label option of an `add command` command. No value is passed to the -command option. This makes it can serve as a 'title bar' of the popup menu.

8.2.14 Progressbar

In Tcl/Tk there is no widget to map a progressbar to. Therefore it is coded out. First a procedure for updating the value of the progressbar is created. After that a procedure for creating the progressbar is created. The a call to the creation procedure is inserted. After that a call to the updating procedure to set the initial value is inserted. For the exact coded I refer to the stylesheet which uses a slightly changed version of the code for the gauge found in *Effective Tcl/Tk programming*[HM98]

8.2.15 Radiogroup

text To set the title with the value of this attribute a label widget is created on the radiogroup frame with the -text option set to text.

8.2.16 Radioitem

The radioitem tag is mapped to a radiobutton-widget. The -variable option of the radiobutton is set to the variable variable passed by the parent radiogroup. The -value option is set to the id of the radioitem. When the passed onchange variable has the value is true the Button-1 event of Tcl/Tk will be binded to the procedure passed by the proc variable.

checked When this attribute has the value is true a call will be made to the invoke command of the radiobutton-widget.

text The value of this attribute is passed to the -text option.

8.2.17 Scale

The scale tag is mapped to the scale widget. The -variable option is set to the @id ++ var which makes it possible to change the value of the scale by chaning the variable with the name @id ++ var.

incrementvalue Passed to the -resolution option.

initvalue To set the initvalue of the scale the associated variable (@id ++ var) is set to the value of initvalue.

max Passed to the -to option.

min Passed to the -from option.

orientation Passed to the -orient option.

showvalue Passed to the -showvalue option.

8.2.18 Statictext

The statictext is mapped to the label-widget. The -text option is set to the line in the line tags.

8.2.19 Window

The mainwindow is mapped to the standard window generated by wish on startup. Other windows are included in a procedure and in this procedure created with the toplevel command. The child windows can then be created by a call to the appropriate procedure. This is standard done in the code when the visible attribute of the child window is true.

8.3 Possible Improvements

8.3.1 More efficient generation of code.

As mentioned before (see section 8.1.3 on page 44) will the Tcl/Tk style-sheet always make all frames for the layout, even when there are not necessary, to preserve the full name of an object when elements are added to the interface. It seems that this is not always necessary to create all frames in the case of the border-layout. For example when a panel with the border-layout only contains a element in the top region it is not necessary to make the left, right, center and bottom frames. This because the name of the element in the top region does not change when a bottom, left, right or center item is added. The only reason why this is not implemented yet is the little time that was left for the project.

It is not difficult but it will take a lot of time (from my perspective, with a fast approaching deadline) to distinguish between all possible cases.

In fact it seems that for the top and bottom elements of the border-layout no panels are ever needed, but I have not researched this good enough, it depends on the features of the Tk-widgets, to be absolutely sure.

8.3.2 Using Variables

The Tcl/Tk-code is not always as optimal as possible. This because of the choice to try to preserve names of the widgets when new elements are added to the window. The use of variables could make the Tcl/Tk-code much more efficient and readable. It also hides the interface code from the event-code programmer.

To use variables I only need a unique variable that contains the full path of the widget. The id attribute of the element, which is unique, could be used for this (see example 4). With this variable it becomes possible to make the code more efficient (frames are left out when they are not needed.) without having to change the event code when elements are added because the variables are used.

Example 4 *example of the use of variables to make the code more efficient and more readable.*

Code without variables:

```
button .printdialog.vpanel.button1 -text "Ok"
pack .printdialog.vpanel.button1 -side left
```

Code with variables:

```
set button1 .printdialog.vpanel.button1
button $button1 -text "OK"
pack $button1 -side left
```

8.3.3 Using Procedures

When the variables as introduced in section 8.3.2 are used it becomes possible to use procedures for the creation of the elements in a window. This makes the code much more readable. A simple example (see example 5) will show the general idea. The procedures could be placed in a special file which always is included in the `application.tcl` file.

Example 5 *Simple example of the use of procedures.*

```
proc createbutton { elementname text side } {
            button $elementname -text $text
            pack $elementname -side $side
}
set button1 .printdialog.vpanel.button1
createbutton $button1 "Ok" left
set button2 .printdialog.vpanel.button2
createbutton $button2 "Cancel" left
```

8.3.4 Reducing the lines of code

It might be possible to reduce the lines of code in the style-sheet by splitting it up in different style-sheets and including them in the main style-sheet. This would make the style-sheet easier to read and much easier to maintain.

8.3.5 Improving Comments and Indents

The layout of the generate code is not ideal on this moment. The comments could be clearer and a better indentation could make the code more readable.

8.4 Conclusions

The generation of Tcl/Tk code with PLUG is possible. The generation of code is straightforward for mast tags. Only for tags wherefor there are in Tcl/Tk no standard widgets causes some minor problems, which were solveable by coding these widgets out.

The Tcl/Tk-code that's generated from PLUG could be made more efficient. The focus of the project was however more on reusibility of event code when the interface is changed than on efficient code generation. As mentioned before it seems possible to preserve the reusibility of event code when the code is made more efficient with the help of procedures and variables. But to implement this in a structured way the entire style-sheet must be rewritten.

Chapter 9

Tools: scripts and a graphical alternative

9.1 Use of shell scripts

We make an extensive use of shell scripts to achieve the implementation showed in figure 4.4. It is only tested for the bash shell under Unix, but probably more shells will work correct too. Even under other operating systems such as Windows it is possible to run the bash shell (with Cygwin).

There is one central script where the whole story starts. This script is called "generate" and is located in the users bin/plug directory (or another directory which is added to the path). This generate script is not language dependent. It will do some basic stuff that is needed for every language:

1. It starts by making links in the destination directory to the DTD's used in PLUG.
2. Then the already existing codeframe are backed up.
3. Then the application XML-file is parsed with the (general) application2codeframe.xsl style-sheet. This will make another script containing commands to make a codeframe for each window in the application. After this script is called it is removed.
4. After the new (empty) codeframes are generated, they are compared with the backed up codeframes. With help of diff3 a new codeframe is made, containing all the already entered code from the old codeframe and the new code-tags.
5. At last the language-specific generate script is invoked. This script is located in the language directory under bin/plug (i.e. bin/plug/delphi/generate).

Because the language-specific generate script is language-specific it is impossible to describe it in such a detail as in the case of the general scripts. But the common thing of all language-specific generate scripts is that they have to create the static gui-code (see figure 4.3) and a script named "integrate", which is responsible for the integration of the code from the codeframe.

9.2 Graphical alternative

For the point-and-click-generation it is also possible to use the tool RICK (Rapid Interface Construction Kit). This toolkit offers the same kind of functionality as the scripts described in section 9.1. It is a development environment for PLUG. You can load a project and display the windows it contains and edit the corresponding codeframe. It is very easy to use. You only have to specify the correct target and to click on the "generate" and "integrate" buttons.

Chapter 10

Conclusions

10.1 Applications

The main benefit of PLUG is that it describes a graphical user interface without a commitment to a specific implementation, and that several different implementations can be quickly derived from it without a lot of manual coding. We think that there are several practical occasions where this benefit comes to its full potential:

- Some applications, such as internet-based client-server applications, have implementations on multiple platforms. For users, it is easy if the user interfaces look the same on different platforms. PLUG makes it easy to achieve this. Changes in the interface which do not involve changes in functionality (i.e. moving a button, rearranging menus) can be instantaneously applied to all implementations without modifying a single line of code.

- In the practice of Software Engineering, it is very important to have documents which precisely determine the design of a system, without mentioning any implementation details. Several engineers who are working on the project need to rely on such documents to make their own design or implementation decisions. An XML description of an interface can function as such a design document.

- In a large project, the person designing the interface is often not the person programming it. With PLUG, a cognitive ergonomist (for example) can define the interface outside of a specific programming environment.

- Tree-like information structures such as PLUG menus can be generated or modified by other applications that use XML technology. For example, a menu describing a product hierarchy can be automatically generated from a product information database using XML and XSLT.

10.2 Future work

10.2.1 Makefiles

In the future it is necessary to make the generate script more efficient with the help of makefiles. The current version of the generate script of PLUG generates all files when invoked even when this is not necessary. Using makefiles can help to reduce this by only regenerating files for which the source has changed.

10.2.2 Import statements

For a better integration of the program code and the interface it seems like a good idea to put some kind of input statement in the code-frames. This allows the programmer to specify some source code files which are needed by the program-code in the code-frame. The integration of pure program code and interface code will become easier this way, because the programmer does not have to edit the generated files by hand anymore to include the include statements.

Bibliography

[app] *Common Desktop Environment: Application Builder User's Guide.* http://www.tru64unix.compaq.com/faqs/publications/base_doc/DOCUMENTATION/V50_HTML/APPBLDR/TITLE.HTM.

[CH99] Alex Ceponkus and Faraz Hoodbhoy. *Applied XML, a Toolkit for Programmers.* Robert Ispen, 1999.

[Gam95] Erich Gamma. *Design Patterns, Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.

[HM98] Mark Harrison and Michael McLennan. *Effective Tcl/Tk Programming, Writing Better Programms with Tcl and Tk.* Addison-Wesley, 1998.

[jav] *Java 2 Platform, Standard Edition v1.2.2 API Specification.* http://java.sun.com/products/jdk/1.2/docs/api/index.html.

[MST95] Aaron Marcus, Nick Smilonich, and Lynne Thompson. *The Cross-GUI Handbook, For Multiplatform User interface design.* Addison-Wesley, 1995.

[tcl] *Tcl 7.6 / Tk 4.2 Manual.* http://velociraptor.mni.fh-giessen.de/TclTk/tcltk-man-html.

[W3Ca] W3C. *XML Path Language (XPath) Version 1.0.* http://www.w3.org/TR/xpath.

[W3Cb] W3C. *XSL Transformations (XSLT) Version 1.0.* http://www.w3.org/TR/xslt.