Guide to StyleSheet Writing in NITE NXT

1. Introduction

This document explains how to write xsl stylesheets which specify a layout for an NXT interface. It contains a series of example stylesheets ranging from the simplest interface containing only a label, to interfaces for displaying structured data from real corpora. It also explains how to specify what action should be taken when the end user clicks on buttons or menus in the interface. It is assumed the reader has working knowledge of xml and xsl.

2. Basic Concepts

NITE NXT can be used to prototype user interface designs; to display structured data; or to edit annotations of structured data. It is likely to be most useful to corpora linguists or other researchers who wish to view structured data in a variety of ways for analysis purposes. NXT is a java program which processes an xml meta-data input file and outputs a user interface display. The meta-data file specifies which data and information displays are available for the end user to work with. Data is usually annotated observations from a linguistic corpus, in xml format. Information displays are specified in xsl stylesheets. The end user chooses the data and display from the interface shown in Figure 1.

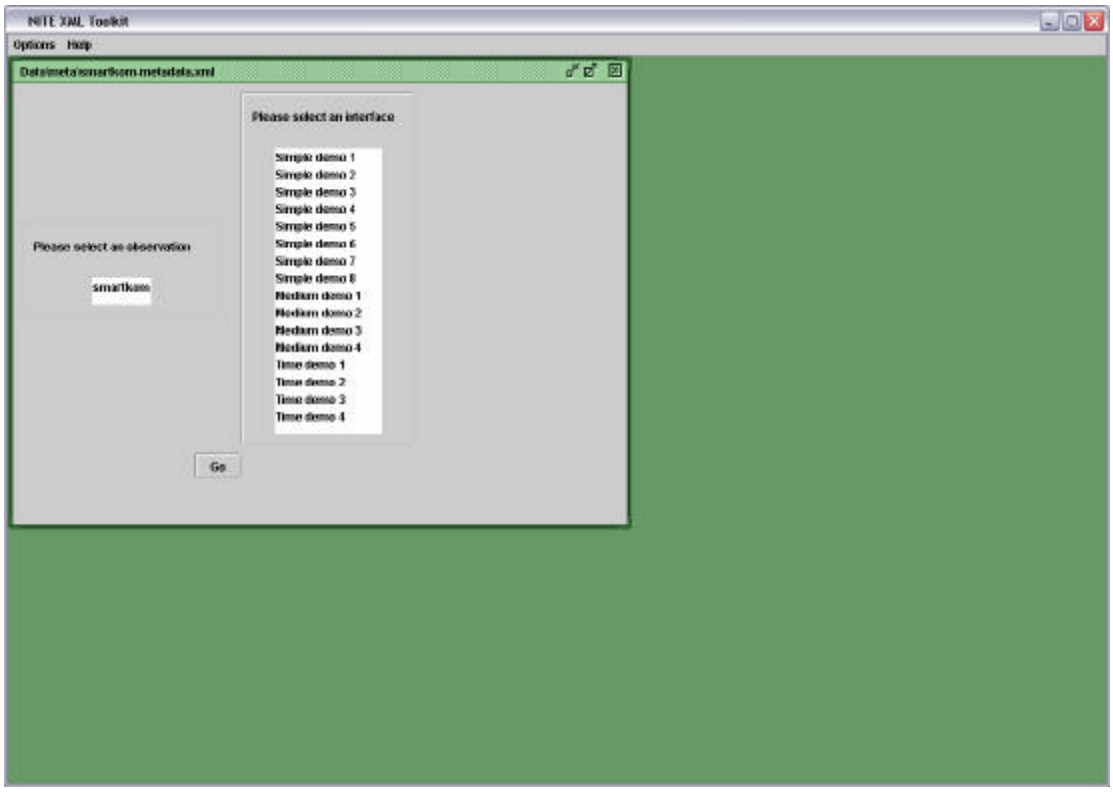


Figure 1 – The NXT main display

When the user presses the “Go” button, the stylesheet processor in NXT transforms the input xml data into an output xml file which specifies the user interface, using the stylesheet selected by the user. This document explains how to write these stylesheets.

Every NITE stylesheet contains a tree of xml which specifies the interface in a declarative way. The tree must begin with <Nite:Root> and end with </Nite:Root>. The start and end of the display specification is delimited by <Nite:Display> and </Nite:Display>. Every display component must be specified by <Nite:DisplayObject>; the “type” attribute specifies which sort of component will be used. A visual index to the display component types can be found in the quick reference guide. The actions which will be taken when the user interacts with the display components are contained in a <Nite:Action> subtree (see section Specifying actions).

3. Simple Information Displays

Simple Demo 1 –InternalFrames, Panels and Labels

The root of the Nite display must be a single **InternalFrame**. This window appears within the main NXT window, as shown in Figure 2. It can be resized and maximized within the main window. All other Nite display components must appear on it. The xsl snippet below shows how to specify an InternalFrame. The ImagePath attribute specifies a path to an image which will be used as an icon in the title bar of the InternalFrame. If this attribute is not supplied, the icon will default to a standard Java icon. The text for the title bar is specified simply as the content of the Nite:DisplayObject; in this case "An Internal Frame".

```
<Nite:DisplayObject type="InternalFrame" ImagePath="Data\Images\smallcherry.gif">
        An Internal Frame
</Nite:DisplayObject>
```

It is often useful to place display objects within a **Panel** because the resulting layout is neater. A Panel placed within the InternalFrame in Figure 2. The Panel has a yellow background colour. The xsl snippet below shows how to specify a Panel.

```
<Nite:DisplayObject type="Panel" background="yellow">
< /Nite:DisplayObject>
```

Another commonly used display component is a **Label**. Labels are generally used to show single non-editable pieces of information, and can be embedded in more complex data structures, such as GridPanes or Trees. Labels can be decorated with images, and the font attributes for the text can also be altered. The xsl code below shows how to create labels. As before, the ImagePath attribute specifies where to find the image which will be used for the icon on this display object (in this case a picture of a cherry, as show in Figure 2). If no image path is specified, the label will be text-only. The FontStyle attribute specifies the font style for the label text. Possible values are "Bold", "Italic" and "Plain". If no FontStyle is specified, it will default to Plain. The FontSize attribute specifies the size of the label text. The attribute value must be an integer . The Font attribute specifies the name of the font to be used. The default is Times New Roman. Note: be careful to specify only fonts which are installed in the end user's computer. The ToolTip attribute is used to set the tooltip text which appears when the end user hovers the mouse over a display object. The text for the label appears in the content of the Nite:DisplayObject, in this case "A test label".

```
<Nite:DisplayObject type="Label" FontStyle ="Italic" Font ="Arial"
FontSize="16" tooltip="Label" ImagePath="Data\Images\cherry.gif">
        A test label
</Nite:DisplayObject>
```

The complete display object tree which specifies the interface in Figure 2 is SimpleDemo1.

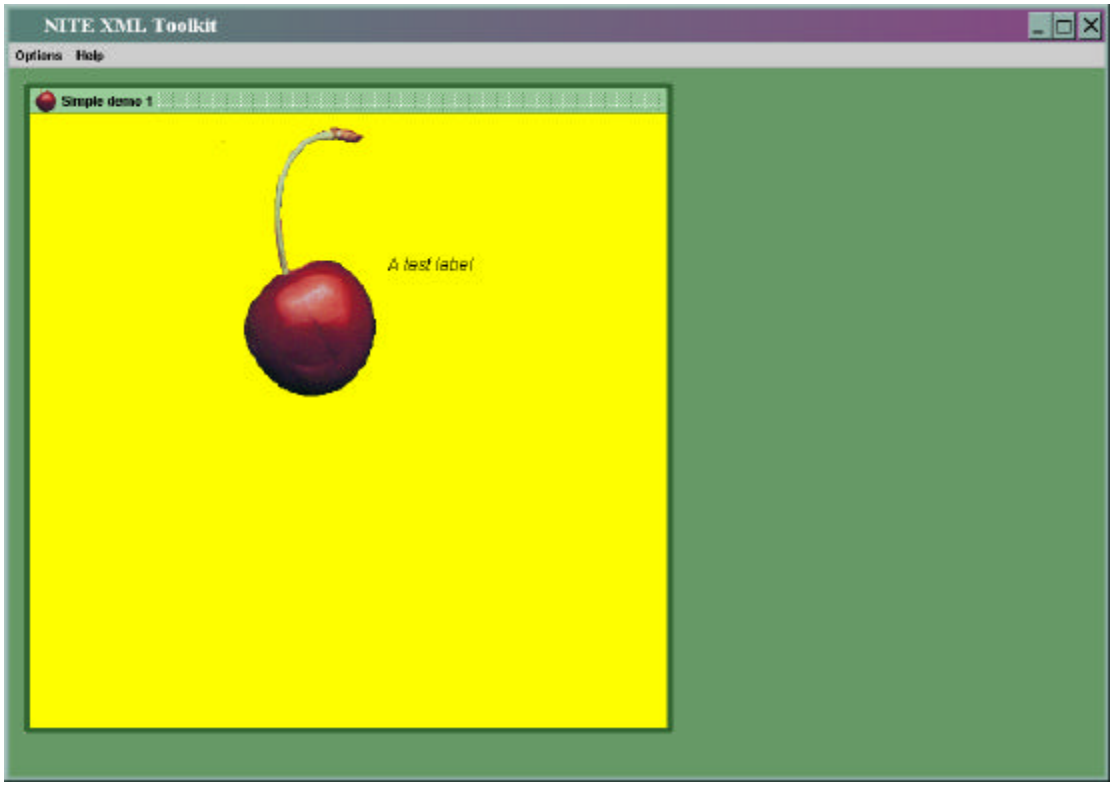


Figure 2- Output of SimpleDemo1.xsl. A Label inside a Panel inside an InternalFrame.

Simple Demo 2 –Tabbed Panes

SimpleDemo 2 demonstrates how to use a **TabbedPane**. The output of this stylesheet is shown in Figure 3. This container is useful. when displaying several information displays, only one of which is required to be active at any given time. A snippet of xsl for specifying a TabbedPane is shown below:

```
<Nite:DisplayObject type="TabbedPane">
</Nite:DisplayObject>
```

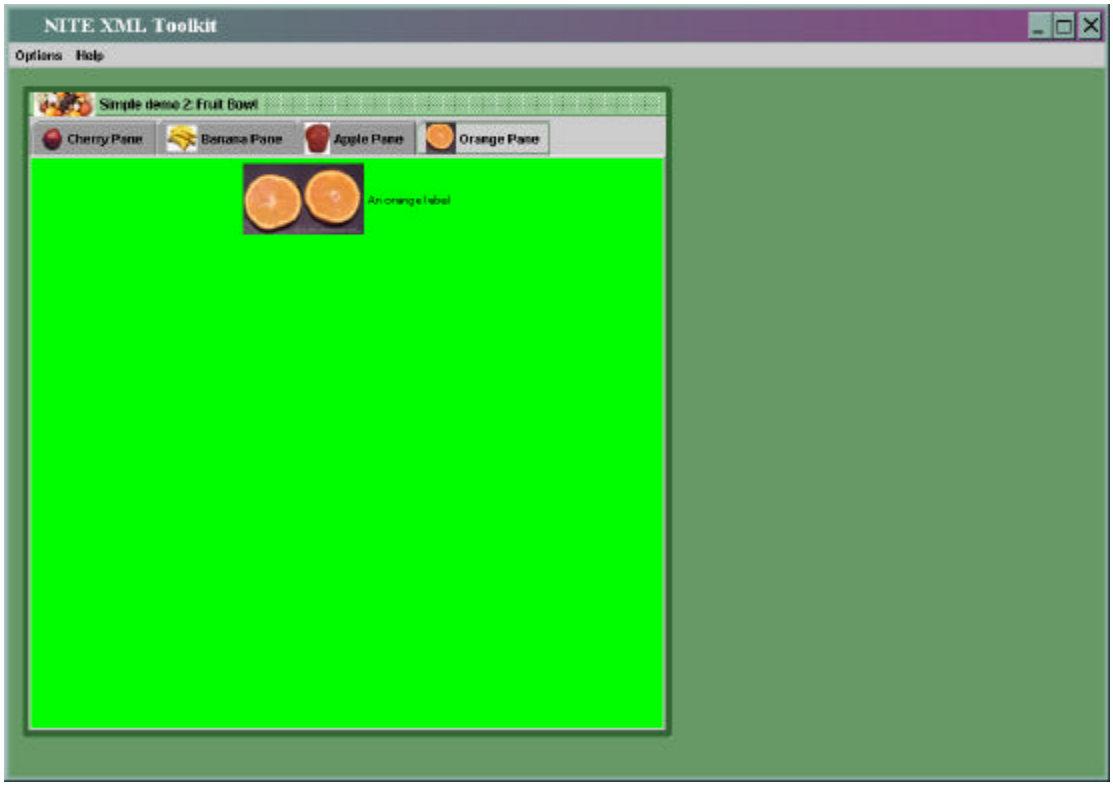


Figure 3 Simple Demo 2: A tabbed with four tabs displaying different pieces of information

There is no limit to the number of tabs in the TabbedPane. The only valid children for TabbedPanes in the Nite:DisplayObjects tree are Panels, with one Panel for every tab. The text and image at the top of the tab are specified by the content of the Panel display object and its ImagePath:

```
<Nite:DisplayObject type="Panel" background="red"
ImagePath="Data\Images\smallbanana.jpg">
        Banana Pane
</Nite:DisplayObject>
```

Simple Demo 3 - SplitPanes

Simple Demo 3 demonstrates how to create a **SplitPane**. This type of display object can be used for displaying two separate information displays where it is useful for the end user to be able to adjust the prominence of one set of data. As shown in Figure 4, the two information displays in a SplitPane are separated by a grey dividing line. The user can drag the dividing line to make one of the displays larger or smaller. Xsl code for specifying a SplitPane is shown below:

```
<Nite:DisplayObject type="SplitPane" split ="horizontal">

</Nite:DisplayObject>
```

The split attribute specifies whether the dividing line of the SplitPane should be horizontal or vertical. The only valid children for a SplitPane in the Nite:DisplayObject tree are Panels, and a SplitPane can contain only two panels.

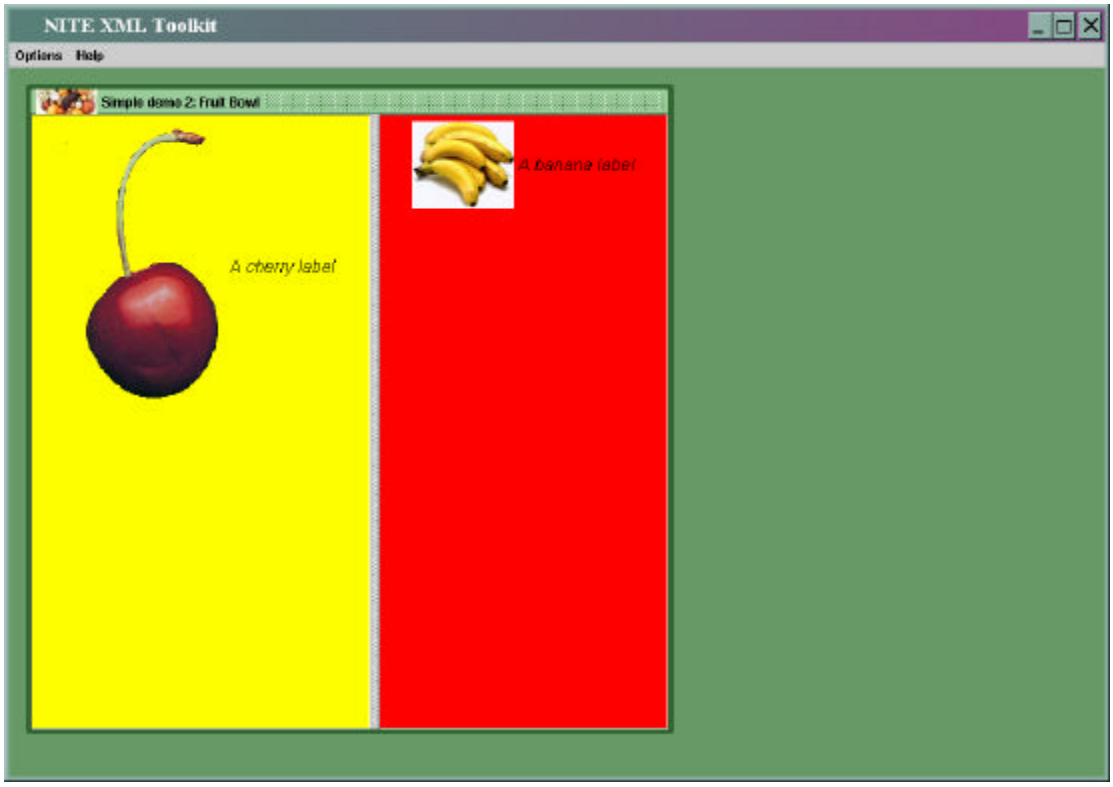


Figure 4 - Simple DisplayDemo 3. An internal frame containing a split pane. Each split contains a pane with a label.

Simple Demo 4 -ScrollPanes

Simple Demo 4 demonstrates the use of a **ScrollPane**. ScrollPanes are useful for information displays which contain more information than can be visible on the screen at one time. When placed on a ScrollPane, a display object appears with scroll bars along the right and bottom if necessary. Figure 5 shows a series of panels on a ScrollPane. The InternalFrame has been resized by the end user so that it is necessary for the ScrollPane to show its scroll bars at the right and bottom. A snippet of Xsl for specifying a ScrollPane is below:

```
<Nite:DisplayObject type="ScrollPane">
</Nite:DisplayObject>
```

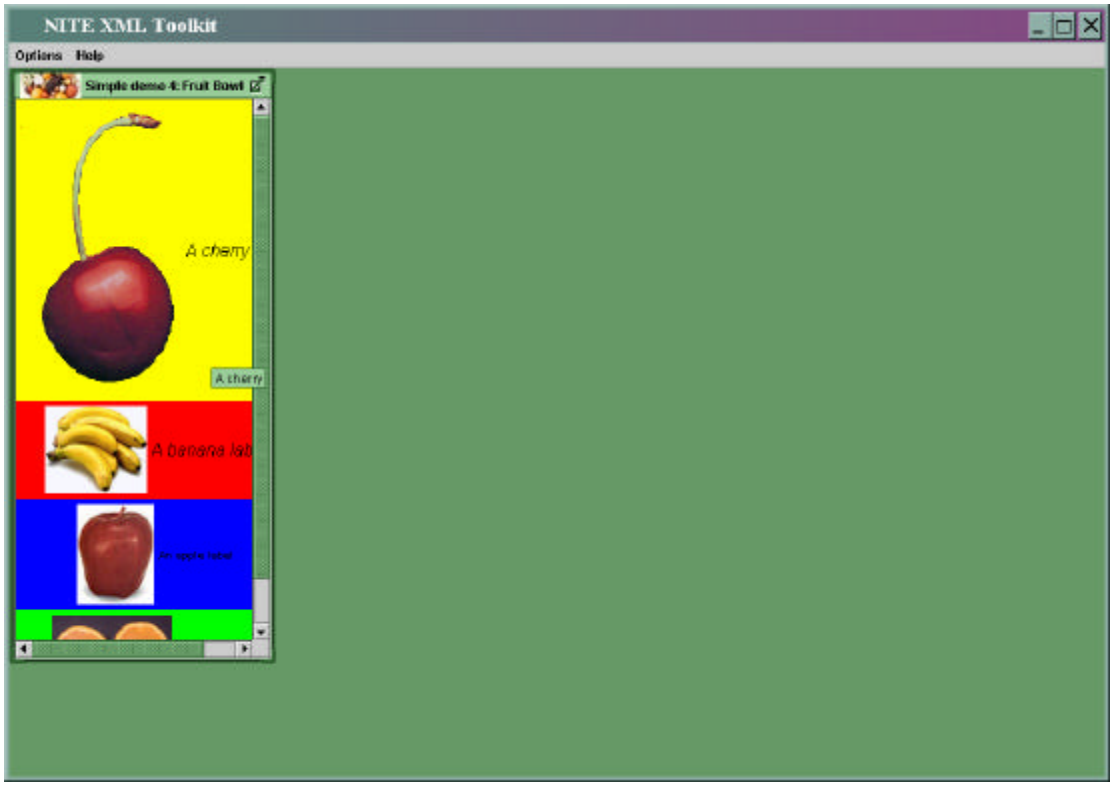


Figure 5 - . SimpleDemo4. Four Panels vertically arranged in a ScrollPane. In this case the user has resized the Internal Frame which contains the ScrollPane, which has forced the scroll bars to become visible so that all the contents of the ScrollPane can be reached

Simple Demo 5 –Lists

Simple Demo 5 illustrates the use of **Lists**. Lists are used to show vertical lists of related pieces of information. In this example, the lists are simple, non-editable, non-interactive displays (see Specifying to find out how to handle user events on Lists). Lists can contain only Labels which specify single pieces of information. Figure 6 shows the output of Simple Demo 5 which specifies two lists on a SplitPane. Example xsl code for specifying lists is shown below:

```
<Nite:DisplayObject type="List">
</Nite:DisplayObject>
```

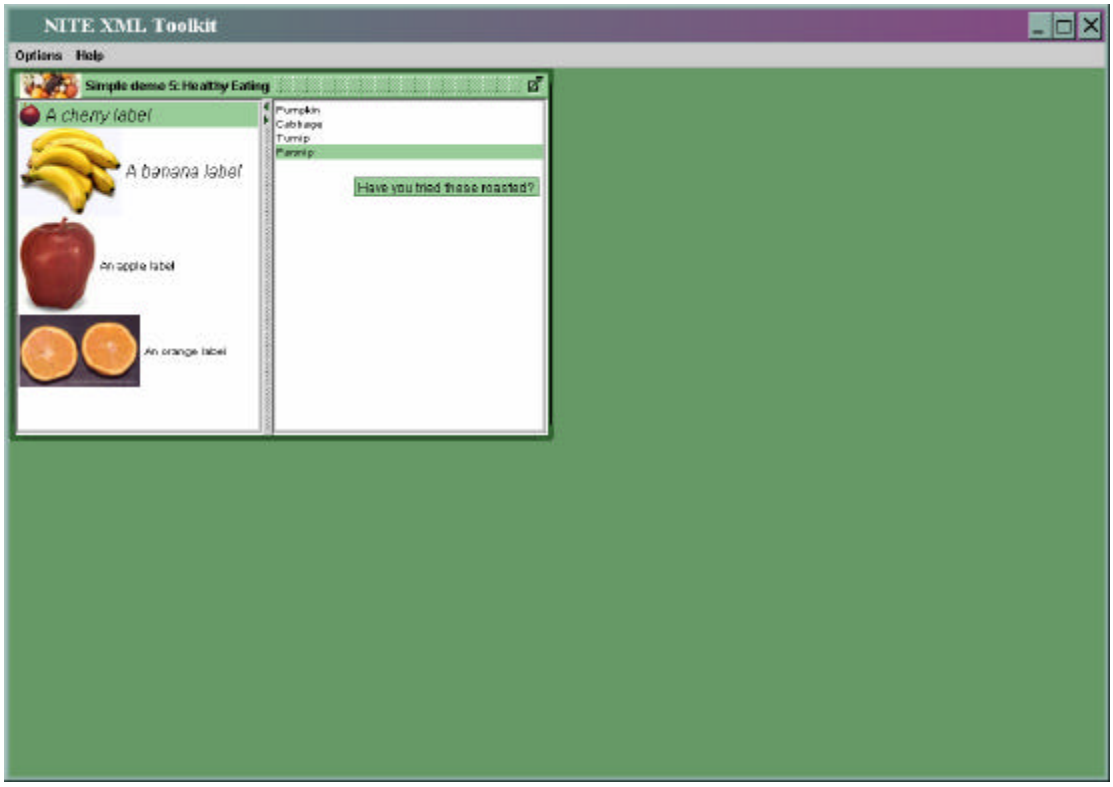


Figure 6 - SimpleDemo5. A SplitPane containing two lists. One list has labels with images, the other has labels with plain text.

Simple Demo 6 - TextAreas

Simple Demo 6 illustrates the use of **TextArea**s. TextAreas are used to display documents. A TextArea can contain any number of **FontStyles** and **TextElements**. FontStyles specify how text should be displayed, while TextElements specify the text itself along with the Font Style which should be used to display it. Figure 7 shows an example TextArea (also with a label for decoration). The document in the TextArea has 4 different FontStyles: large green bold text; blue text; red text and normal black text. It has several TextElements, e.g. “Wisdom from the man who brought you the Cat in the hat”; “I like”; “nonsense”; “, it wakes up the brain cells.....”. Whenever a change is style is required, a new TextElement is specified. The xsl which specifies the TextArea in Simple Demo 5 is shown below:

```
<Nite:DisplayObject type="TextArea">

        <Nite:DisplayObject type="FontStyle" name="bluestyle"
        textcolour="Blue" FontSize="14" />

        <Nite:DisplayObject type="FontStyle" name="greenstyle"
        textcolour="Green" font="Roman" FontStyle="Bold"
        FontSize="20" />

        <Nite:DisplayObject type="FontStyle" name="redstyle"
        textcolour="Red" font="Roman" FontStyle="Regular"
        FontSize="12" />

        <Nite:DisplayObject type="FontStyle" name="normalstyle"
        textcolour="Black" font="Roman" FontStyle="Regular"
        FontSize="12" />

        <Nite:DisplayObject type="TextElement" style="greenstyle">

                Wisdom from the man who brought you
                  the Cat in the Hat
```

```
            <xsl:text ></xsl:text >
        </Nite:DisplayObject>
        <Nite:DisplayObject type="TextElement"
            style ="normalstyle">I
            like</Nite:DisplayObject>
        <Nite:DisplayObject type="TextElement"
            style ="bluestyle">nonsense</Nite:DisplayO
            bject>
        <Nite:DisplayObject type="TextElement"
            style ="normalstyle">, it wakes up the
            brain cells. Fantasy is a necessary
            ingredient in living. It's a way of looking
            at life through the wrong end of the
            telescope. And that enables you
            to</Nite:DisplayObject>
        <Nite:DisplayObject type="TextElement"
            style ="bluestyle">laugh</Nite:DisplayObject
            >
      <Nite:DisplayObject type="TextElement"
            style ="normalstyle">
          at life's realities.
            <xsl:text ></xsl:text >
        </Nite:DisplayObject>
        <Nite:DisplayObject type="TextElement"
            style ="redstyle">Dr
            Seuss</Nite:DisplayObject>
    </Nite:DisplayObject>
</Nite:DisplayObject>
```

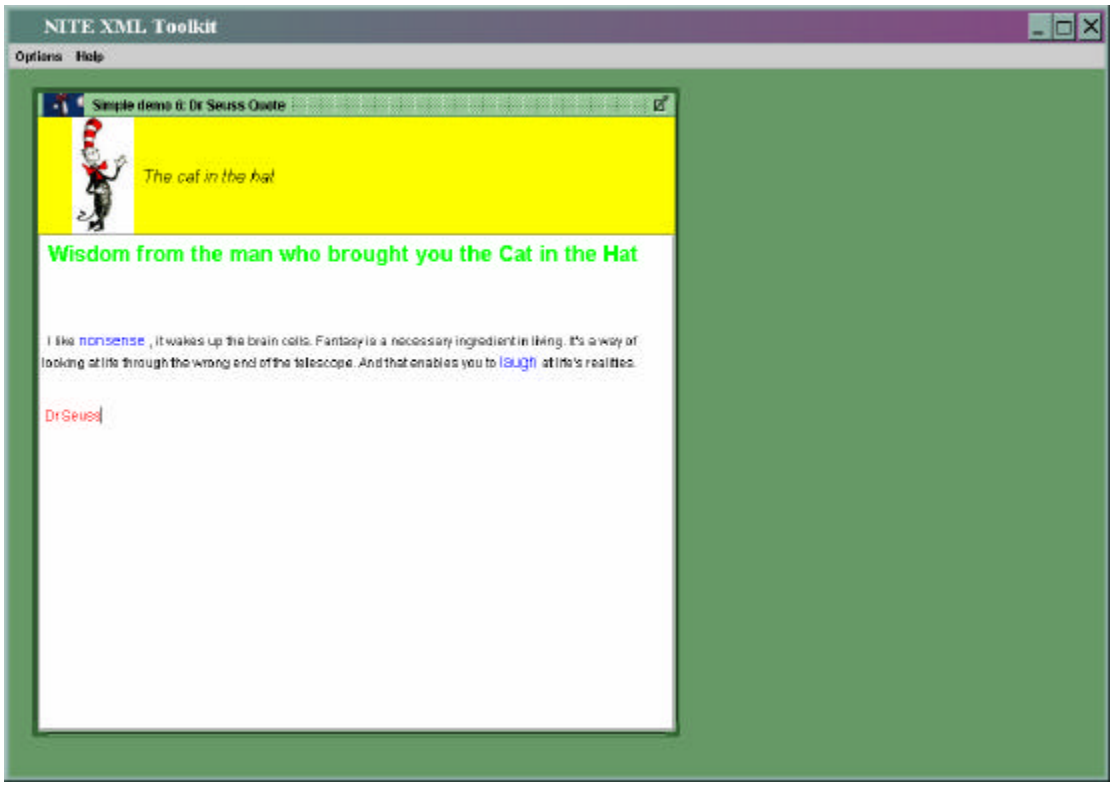


Figure 7 - An internal frame displaying a panel which contains a label and a text area.

Simple Demo 7 –GridPane

GridPanes are useful for displaying information in a structured tabular format. SimpleDemo7 illustrates how to use it for a simple example; for more examples using real corpora data see Medium Example 1 –GridPanel for parts of speech and translation. The output of the stylesheet is shown in Figure 8.

Xsl code for specifying a GridPanel is show below.

```
<Nite:DisplayObject type="GridPanel" Border="true" Columns="4" background="white">
</Nite:DisplayObject>
```

The Border attribute specifies whether each entry in the GridPanel is surrounded by a border or not. The Background attribute specifies a background colour for the GridPanel (default is grey). The Columns attribute specifies how many columns are in the GridPanel. In this case there are four. The only legal child type for GridPanes in the Nite:DisplayObjects tree is GridPanelEntry.

Example code for specifying a GridPanelEntry is shown below:

```
<Nite:DisplayObject type="GridPanelEntry" ColSpan="1" RowSpan="1">
</Nite:DisplayObject>
```

The ColSpan attribute specifies how many of the GridPanel's columns should be used up by this GridPanelEntry. The RowSpan attribute does the same for the rows. The default value for both attributes is 1 i.e. GridPanelElements usually take up one cell of the grid. Occasionally it is desirable for elements to span more than one column, as in the star fruit picture in Figure 8. This was accomplished by setting ColSpan to 2.

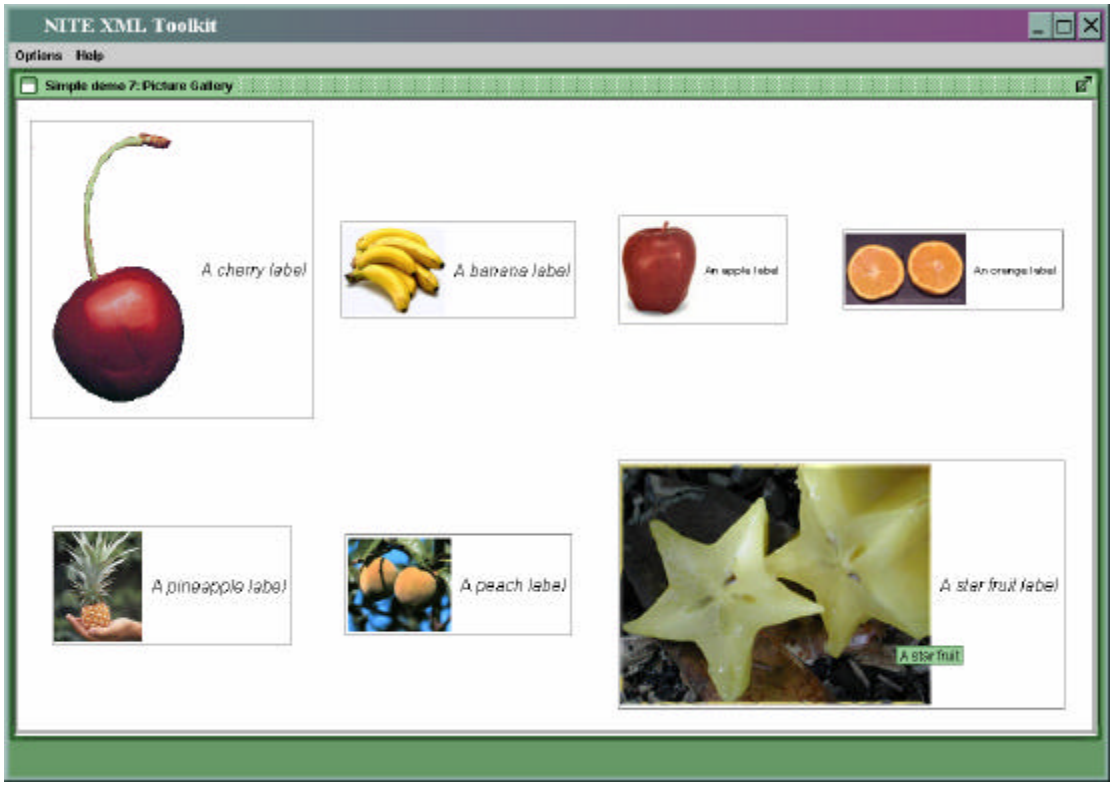


Figure 8 -SimpleDemo7.xsl. An Internal Frame containing a GridPane with 4 columns and 2 rows. Each GridPaneEntry contains a label. Every GridPaneEntry takes up one cell, apart from the star fruit label entry. It spans 2 columns because it is a big picture.

Simple Demo 8 – Trees

Hierarchical structured data can be represented in a **Tree**. A simple example of this can be found in Simple Demo 8, while more complicated examples using real corpus data can be found in Medium Example 3 – Structured Data in Trees. The output of the SimpleDemo8 stylesheet is shown in Figure 9.

A Tree is specified thus:

```
<Nite:DisplayObject type="Tree">

</Nite:DisplayObject>
```

The only legal children for Trees in the Nite:DisplayObject tree are TreeNodes. TreeNodes can contain TreeNodes (if they are branch nodes) or Labels, TimedLabels or GridPanels (if they are leaf nodes). This is an example of a branch node:

```
<Nite:DisplayObject type="TreeNode">
        Creatures
</Nite:DisplayObject>
```

This is an example of a leaf node:

```
<Nite:DisplayObject type="TreeNode">
        <Nite:DisplayObject type="Label" textcolour="red"
        Font="Arial" FontSize="16" tooltip="A cute panda"
        ImagePath="Data\Images\pandacloseup.jpg">
        Red panda
        </Nite:DisplayObject>
</Nite:DisplayObject>
```

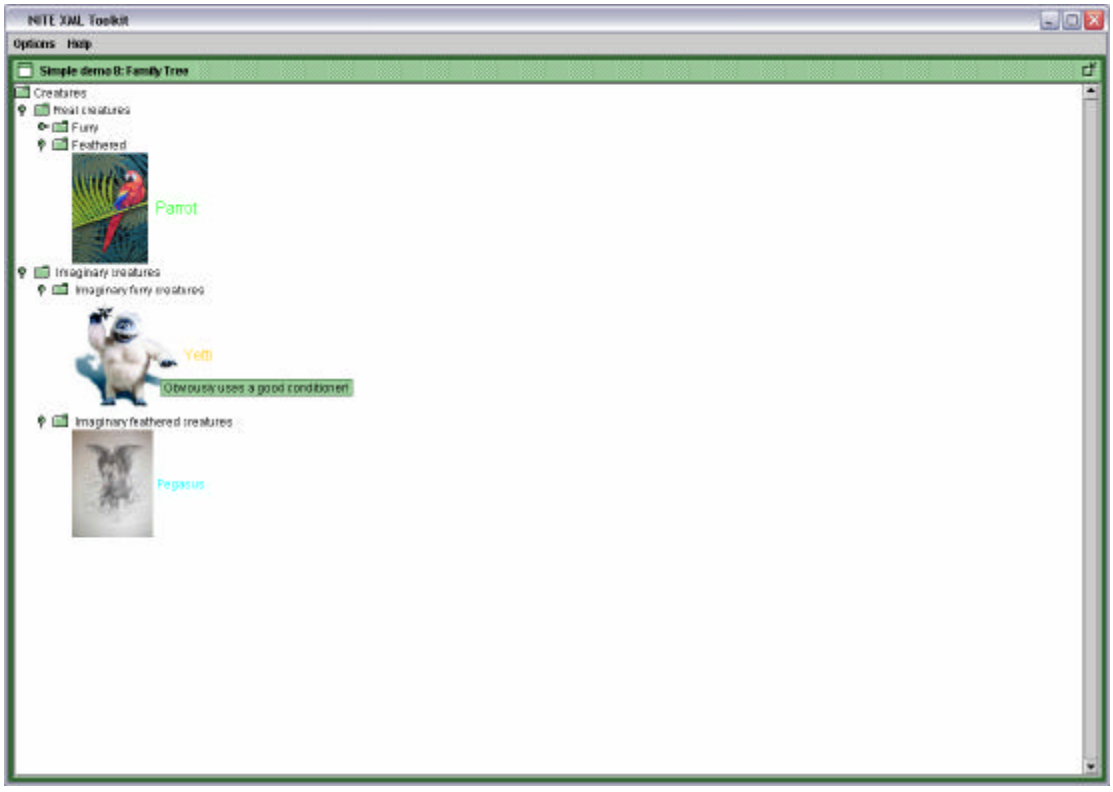


Figure 9 - SimpleDemo8.xsl A Tree on a ScrollPane on an Internal Frame. The tree contains pictorial Labels as leaf nodes.

4. Examples using real corpora

The following example stylesheets are more complex than the Simple Demos because they use data from real corpora: Smartkom and the Maptask.

Medium Example 1 –GridPanel for parts of speech and translation

Medium Example 1 shows how to use a GridPanel used to display data from the Smarktkom corpus (see output in Figure 10). The Smartkom data contains original German transcription from an audio track, part of speech for every German word, and an English translation for every German sentence. The ColSpan attribute in GridPanelEntry can be used to ensure that the German words and corresponding part of speech are aligned, with the English translation for a whole sentence underneath all of the words and parts of speech. This is accomplished using the following xsl template:

```
<xsl:template match="utterance">
    <xsl:param name ="count" select="count(word[@type='W'])" />
  - <Nite:DisplayObject type="GridPanel" Columns="{$count}" Border="true">
    - <xsl:for-each select ="word">
      - <xsl:if test ="@type = 'W'">
        - <Nite:DisplayObject type="GridPanelEntry" RowSpan="1"
            ColSpan="1" position="fill">
          - <Nite:DisplayObject type="Label" FontStyle ="Arial"
              FontSize ="14">
            <xsl:value-of select="." />
          </Nite:DisplayObject>
        </Nite:DisplayObject>
      </xsl:if>
    </xsl:for-each>
    - <xsl:for-each select ="word">
      - <xsl:if test ="@pos != ''">
        - <Nite:DisplayObject type="GridPanelEntry" RowSpan="1"
            ColSpan="1" position="fill">
          - <Nite:DisplayObject type="Label" FontStyle ="Arial"
              FontSize ="12" textcolour ="red">
            <xsl:value-of select="@pos" />
          </Nite:DisplayObject>
        </Nite:DisplayObject>
      </xsl:if>
    </xsl:for-each>
    - <Nite:DisplayObject type="GridPanelEntry" RowSpan="1"
        ColSpan="{$count}" position="fill">
      - <Nite:DisplayObject type="Label" FontStyle ="Arial" FontSize ="14"
          textcolour ="blue">
        <xsl:value-of select="@translation" />
      </Nite:DisplayObject>
    </Nite:DisplayObject>
  </Nite:DisplayObject>
```

```
</xsl:template>
```

First of all, the template specifies a GridPanel with a column for every German word in an utterance. This is achieved by counting how many matches there are for Words of type “W” and setting the Columns attribute to this value.

There are then two loops to print out the German words and the corresponding parts of speech.

The first loop iterates over all matches for “Word” of type “W” (which in this coding scheme represents non-punctuation words). A label is created for every word, and each label is placed on a GridPanelEntry which takes up one cell of the GridPanel. The “position” attribute of the GridPanelEntry is set to “fill”, which means that the contents of the entry should stretch horizontally to take up all the available space in that cell. Another possible value of this attribute is “center” which means that the content of the entry is placed in the middle of cell, with blank space around it.

The second loop iterates over all matches of Word of type which have a POS tag. It creates a label to display the part of speech tag in red inside a GridPanelEntry of size one GridPanel cell. This has the effect of placing the appropriate part of speech tag directly underneath the matching word, as every word has a POS tag in the data.

Next a label with blue text colour is created for the English translation of the German utterance. This is placed in a GridPanelEntry which has a ColSpan attribute set to the number of columns in the GridPanel i.e. the label will span the whole length of the utterance.

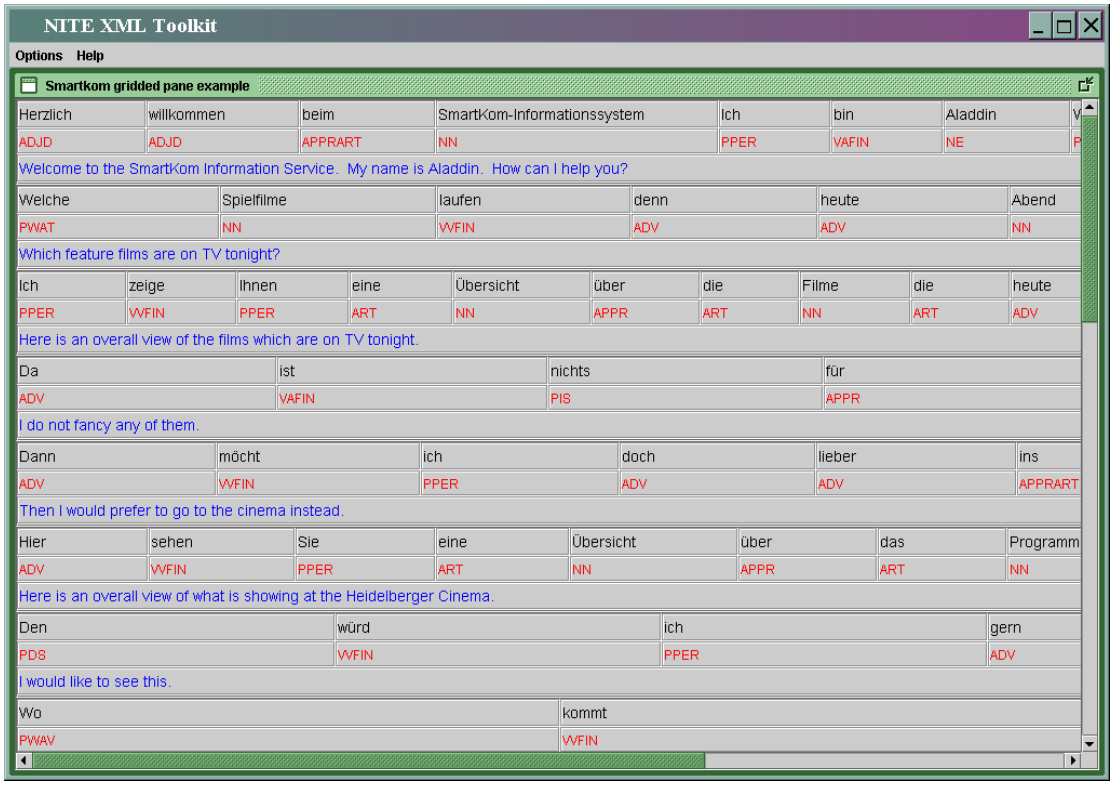


Figure 10 - MediumExample1.xsl. A ScrollPane containing a series of GridPanels displaying data from the SmartKom Example. for the words, parts of speech and translation.

Medium Example 2 – Indentation in TextAreas

Medium Example 2 shows how to use TextAreas to display structured data using indentation. The output of this stylesheet is shown in Figure 11. Utterances in the Smartkom data are annotated with dialogue games. Games can be nested within other games., so it is often useful to emphasise this structure in the display. This can be achieved by setting tabs in the TextArea so that content within games is indented (this can also be achieved using Trees).

The templates which specify the tab settings for the TextArea are shown below.

```
<xsl:template match="game">
    <xsl:param name ="tabDepth" select="0" />
  - <Nite:DisplayObject type="TextElement" tabstop="{$tabDepth}"
        style ="gamestyle">
      <xsl:value-of select="@type" />
      <xsl:text >game:</xsl:text >
      <xsl:text ></xsl:text >
    </Nite:DisplayObject>
  - <xsl:apply-templates>
      <xsl:with-param name ="tabDepth" select="$tabDepth + 1" />
    </xsl:apply-templates>
</xsl:template>
```

The game template matches to dialogue game annotations in the xml data. It creates a TextElement to display the type of the dialogue game in a FontStyle for displaying games, and a tab stop attribute value of 0. The tab stop attribute specifies the level of indentation for the TextElement. Top level games should not be indented, and so have a tab depth of 0. The tab depth is incremented and passed as a parameter for other matching templates.

A fragment of the matching utterance template is shown below. It uses the tab depth parameter to set up a TextElement to display the speaker of the utterance at the correct indentation – one tab after the game that the utterance belongs to.

```
<xsl:template match="utterance">
    <xsl:param name ="tabDepth" />
  <Nite:DisplayObject type="TextElement" tabstop="{$tabDepth}"
      style ="movestyle">
    <xsl:value-of select="@who" />
  </Nite:DisplayObject>
```

....

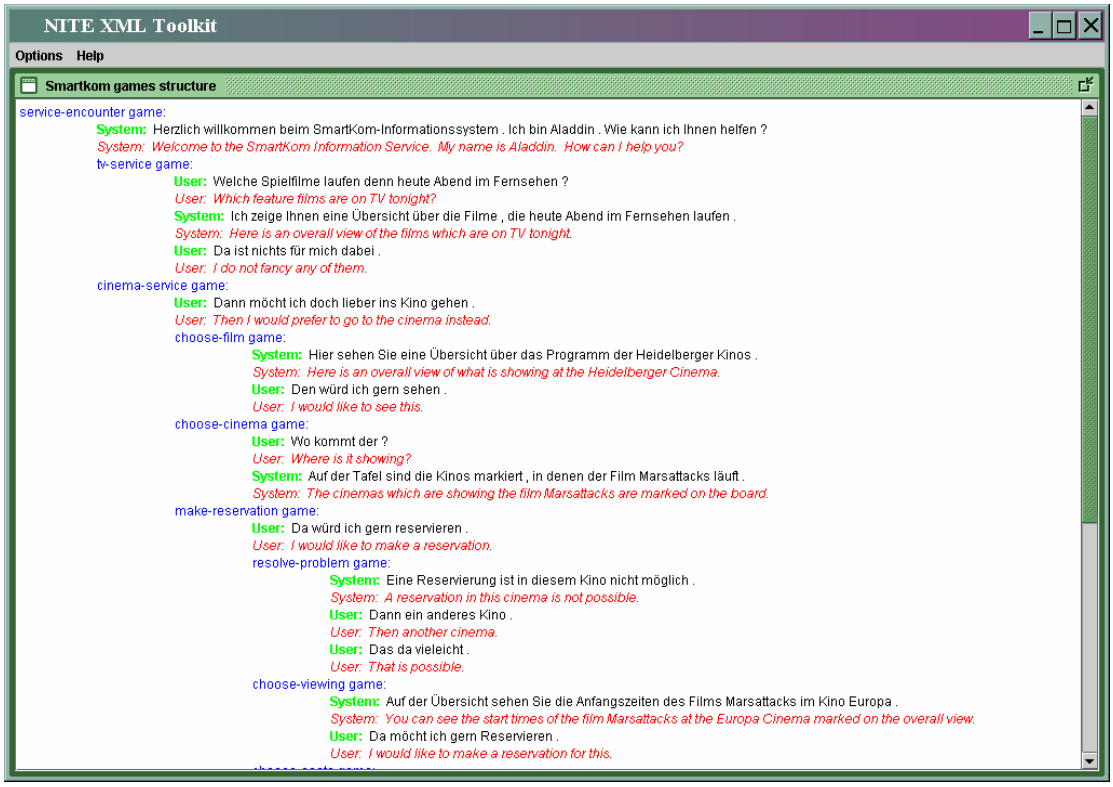


Figure 11 - . MediumExample2.xsl.The InternalFrame contains a TextArea which displays the structure of the smartkom data using indentations. The translation of each utterance is displayed under the data in red italic style.

Medium Example 3 – Structured Data in Trees

Trees can also be used to display hierarchical structured data. Medium Example 3 shows how to display the dialogue game structure of the Smartkom data in a Tree. The output of this stylesheet is show in Figure 12. The Tree uses custom icons. This is specified as follows:

```
<Nite:DisplayObject type="Tree" ExpandedImage="minus.gif" CollapsedImage="plus.gif"
OpenImage="" ClosedImage="">
```

The expanded and collapsed icons are used to indicate whether tree nodes are hiding more nodes within them, or whether they are fully expanded. In this case, no open or closed images are specified.

Recursive templates are used to populate the Tree so that the structure of the Tree mirrors the structure of the dialogue games in the xml data. The game template recursively creates a TreeNode for every dialogue game labeled with the game type. The utterance template creates leaf nodes in the tree with Labels displaying words in the utterance.

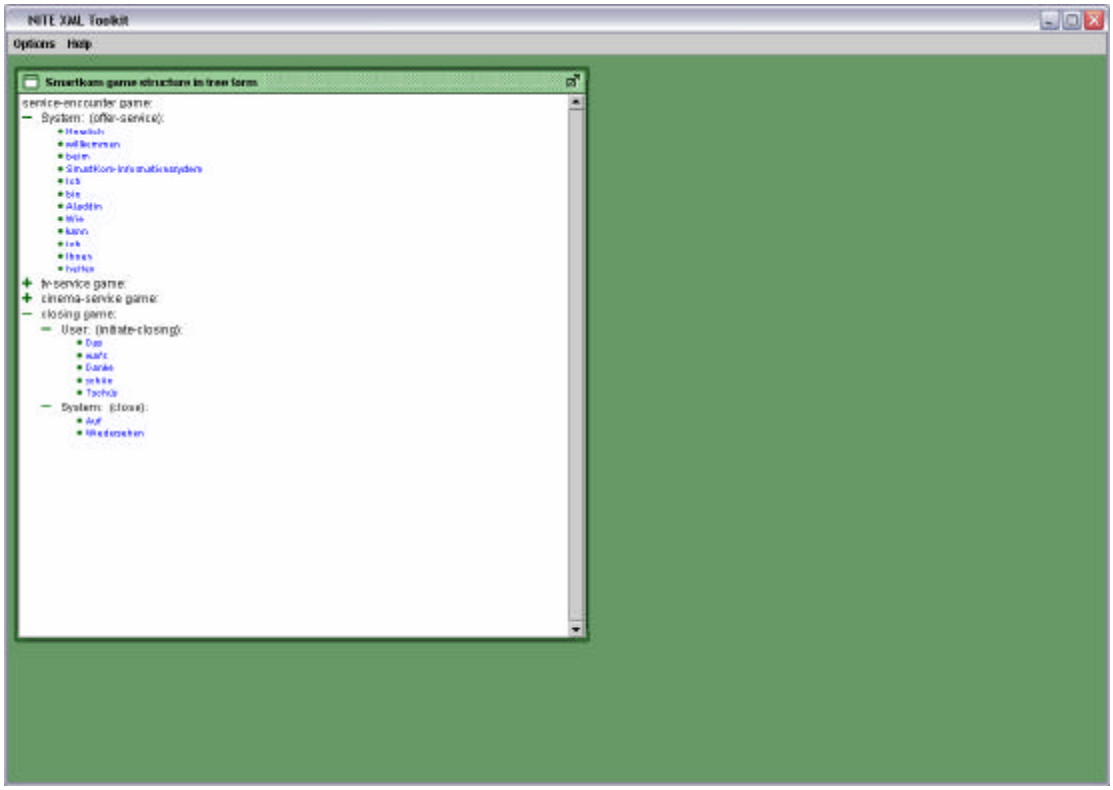


Figure 12 - A Tree on an Internal Frame. The tree represents the structure of the smartkom dialogue data. The words uttered by system and user are leaf nodes. The icons for the tree have been customised.

Medium Example 4 – Trees containing Grid Panes

Medium Example 4 shows how to show structured data in a way which gives an overview of the structure of the data as well as detailed information (see Figure 13). Trees are used to illustrate the structure of the dialogue moves in the Smartkom data while GridPanes are used to show details of each utterance, including parts of speech tags and a translation. This is achieved in a similar way to the previous example (Medium Example 3 – Structured Data in Trees), except that in the utterance template, the leaf TreeNodes contain GridPanes. The GridPanes are specified in the same way as Medium Example 1 –GridPanel for parts of speech and translation.

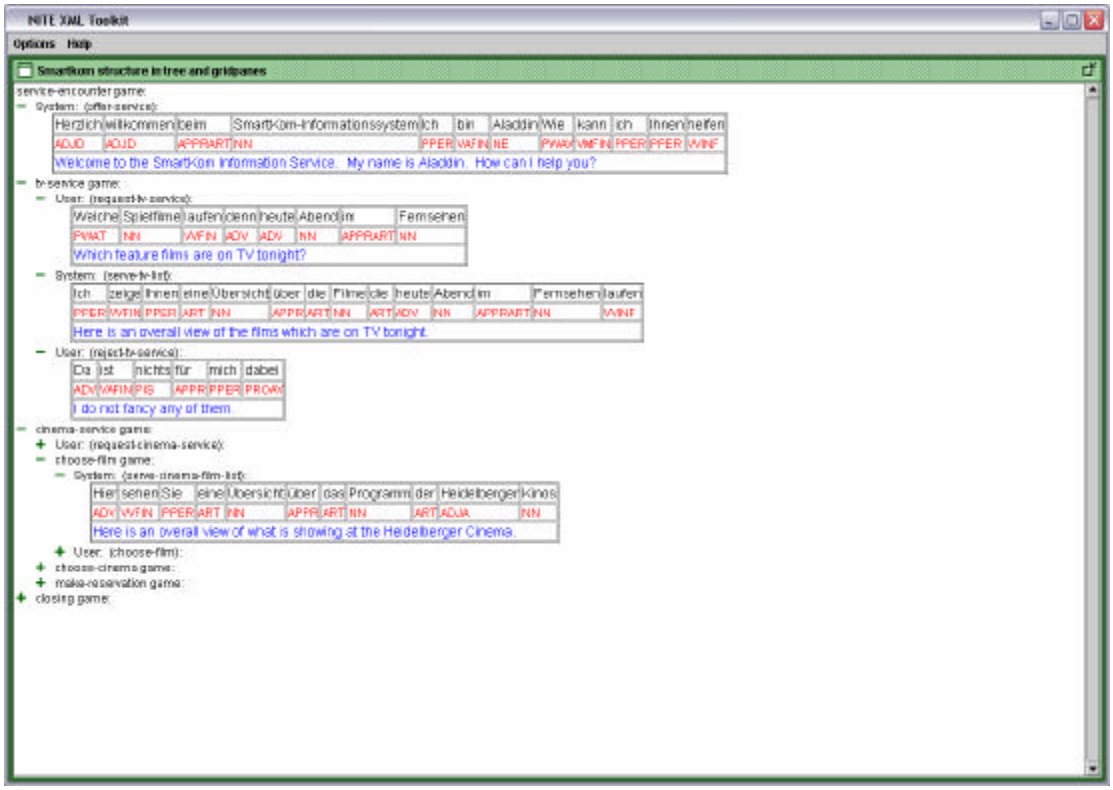


Figure 13 - A Tree containing GridPanels as leaf nodes. This allows both a view onto the structure of the dialogue as well as detailed information each utterance, including parts of speech.

5. Using the Time Highlighting Feature

NITE NXT enables the end user to synchronize a video or audio track with a textual transcription . When the user clicks on the “Synchronize to text” checkbox on the video or audio window (see Figure 14), text in the information display which matches the current time on the video will be highlighted. The green highlighter moves through the information display as the video advances. To achieve this, the stylesheet must specify start and end times for display objects which should be synchronised with the clock. The next sections illustrate how this can be done using the Smartkom and Maptask corpora.

Timing with the Smartkom Data

The Smartkom data is annotated for time at an utterance level. This means that sentences rather than individual words have associated timing information.

In Time Example 1, the data is displayed in a GridPane, using similar code to Medium Example 1 –GridPanel for parts of speech and translation. However, the labels which display the German words are a special type of label – **TimedLabels**. TimedLabels work in the same way as Labels except they have additional attributes: **start** and **end**, the start and end times of the time scope of this display object. Each TimedLabel within a GridPanel for an utterance uses the start and end time of the utterance so that they are all highlighted simultaneously. Neither the parts of speech tags nor the translation use TimedLabels because these are additional pieces of annotation information which are not aligned to the video signal. Note that the

GridPane and the GridPaneEntries do not contain timing information; this is delegated to the TimedLabels contained within them.

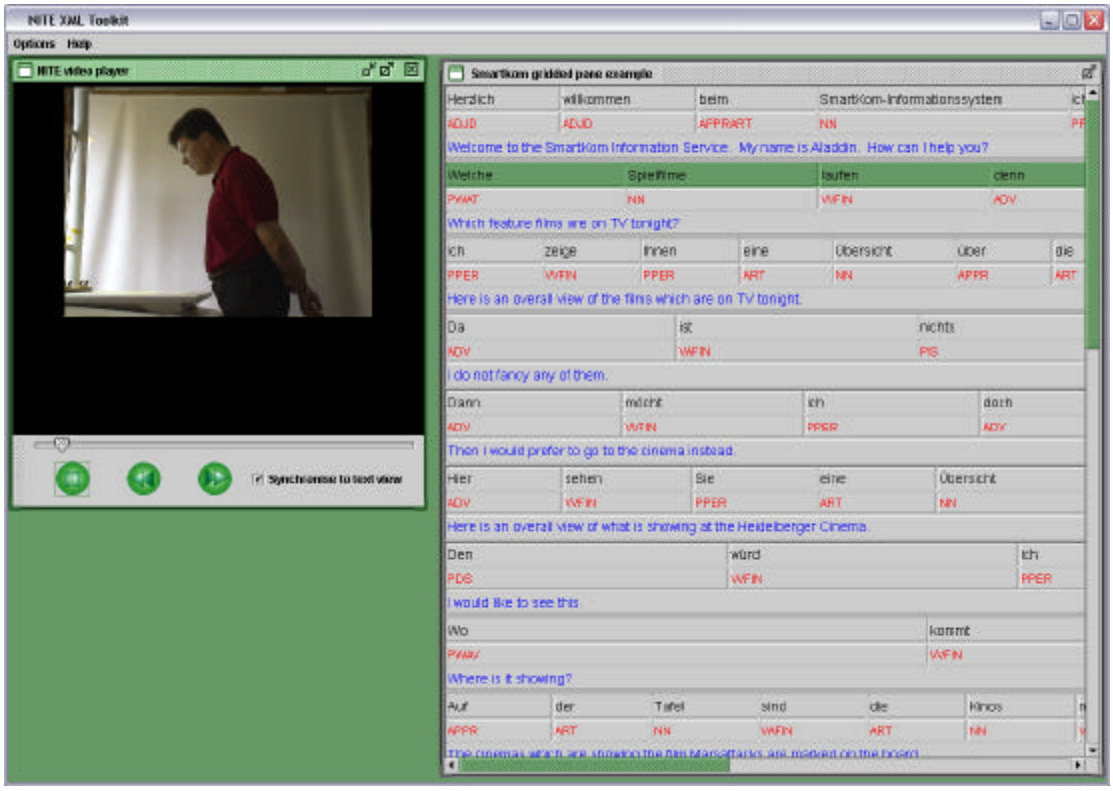


Figure 14 - TimeExample1.xsl. A GridPane showing smartkom data. The utterance currently spoken on the video is highlighted in green in the GridPane.

Figure 15 shows how timing information can be displayed on Trees using TimeExample2. Leaf nodes within time scope are highlighted in green. The tree nodes collapse automatically to show child nodes within time scope. This is accomplished simply by altering the Medium Example 3 – Structured Data in Trees stylesheet so that is uses TimedLabels rather than Labels as leaf nodes.

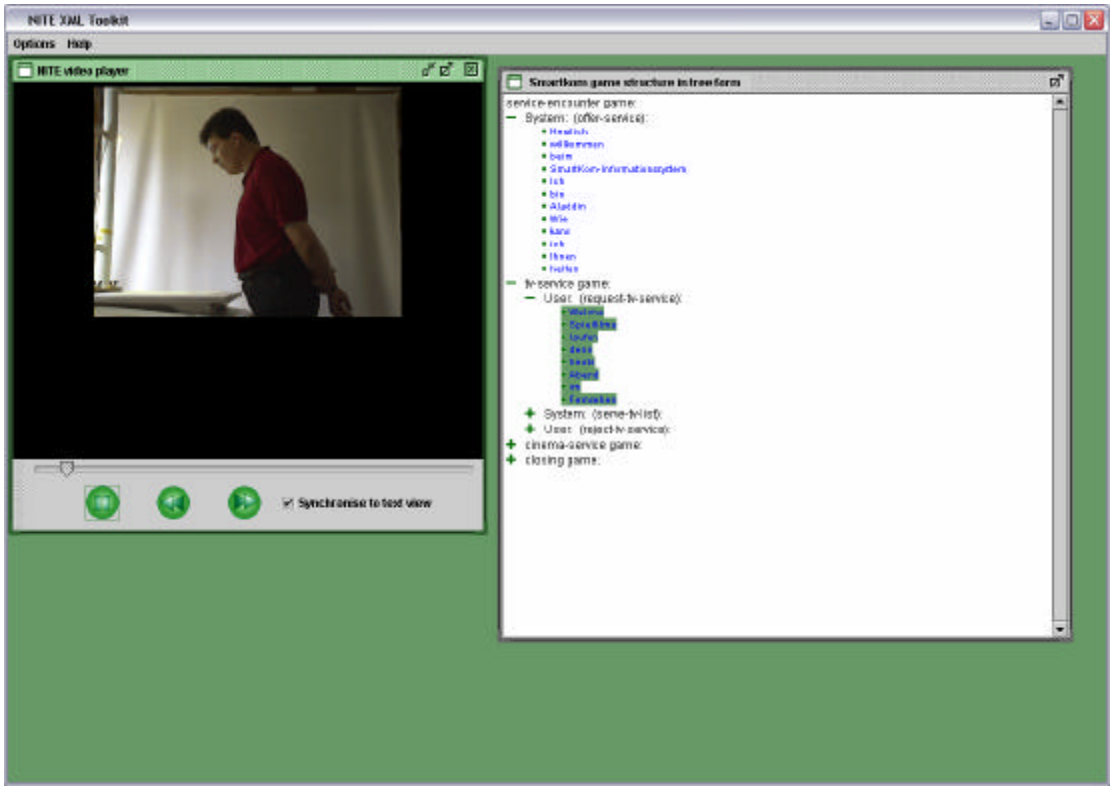


Figure 15 - TimeExample2.xsl. The smartkom data represented in tree form with the utterance currently spoken on the video highlighted in green.

TimeExample3 illustrates how the Medium Example 4 – Trees containing Grid Panes stylesheet can be adapted to denote time, as shown in Figure 16. Once again, this is accomplished by changing the Labels within the GridPanelEntries to TimedLabels.

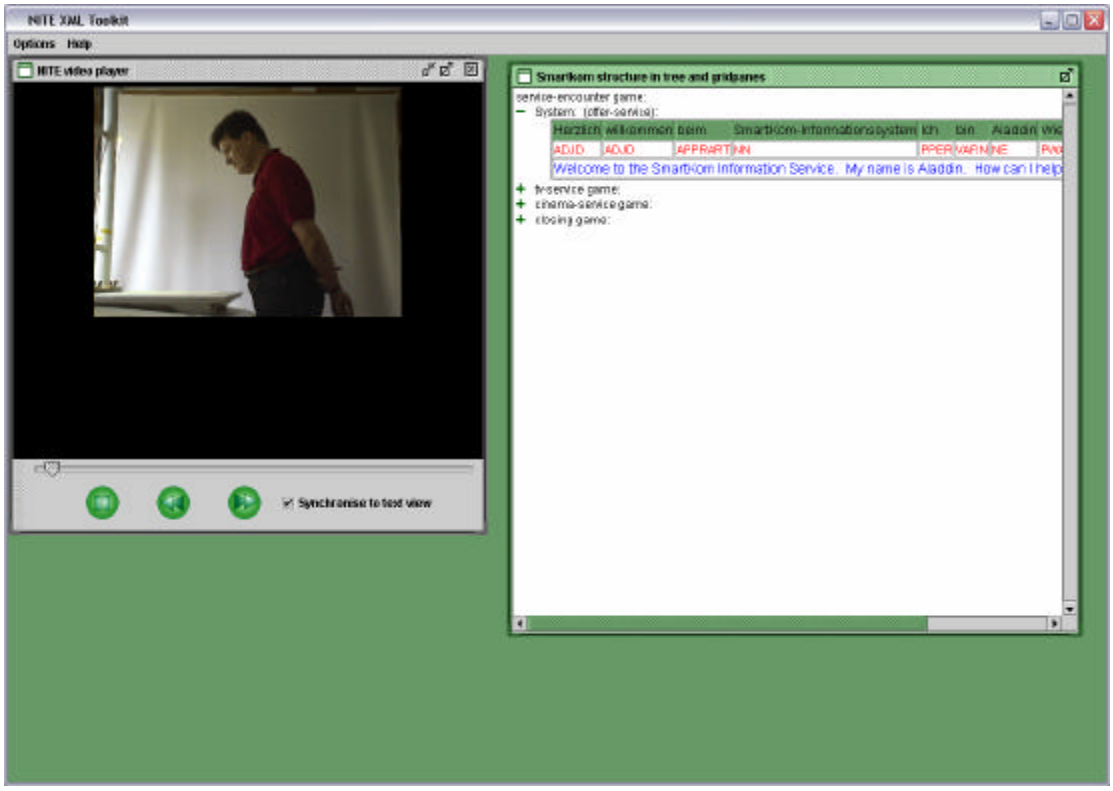


Figure 16 - TimeExample3.xsl. The smartkom data represented in GridPanes as TreeNodes. The currently spoken utterance is highlighted in the appropriate top row of the GridPane element in green.

TimeExample4 shows how to specify that TextElements within a TextArea should be highlighted when in time scope. This is achieved by adapting the stylesheet Medium Example 2 – Indentation in TextAreas so that the TextElements specify **start** and **end** attributes representing when the elements come in and out of time scope.

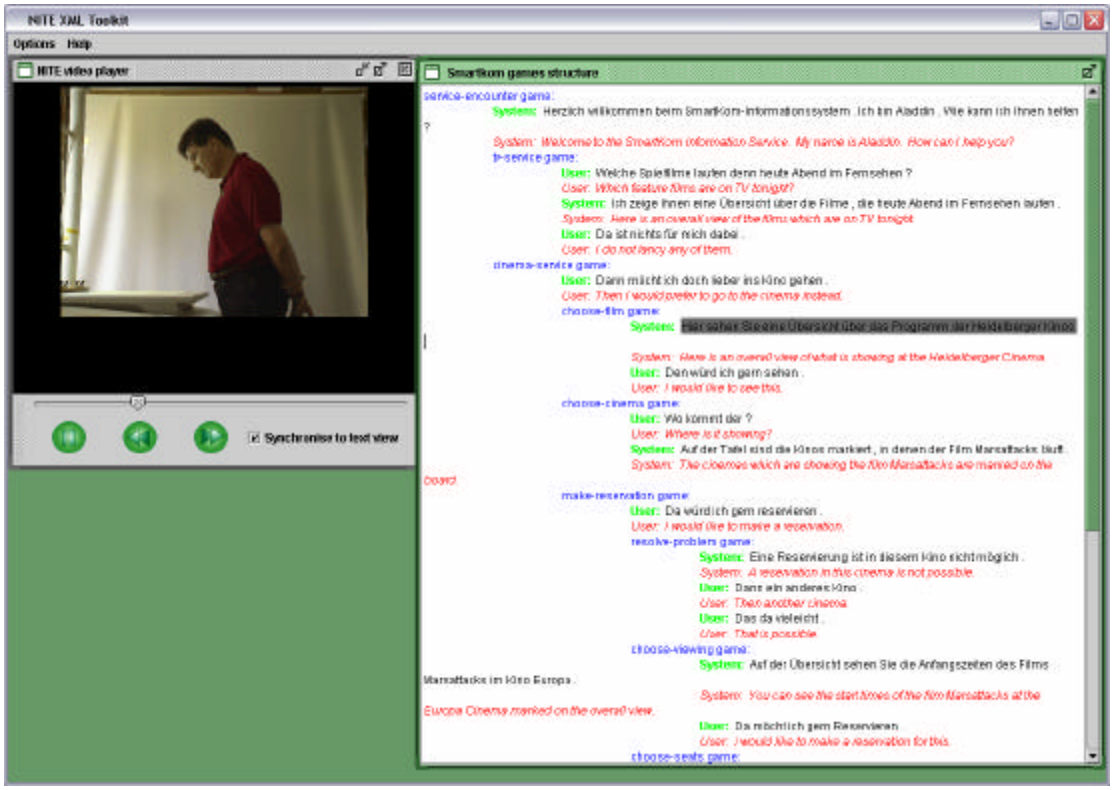


Figure 17 - TimeExample4.xsl. The smartkom data is displayed as indented text on a TextArea. The TextElements are time aligned, so that the transcription is highlighted when the corresponding bit on the video is reached.

Timing with the Map Task Data

The MapTask corpus has more detailed timing information than the Smartkom data. Each word in the Maptask is a timed unit with start and end times specified. This makes it possible to sychnronize the transcription display to the audio/video signal more accurately, as shown in Figure 18, Figure 19 and Figure 20. Stylesheets MapTaskTimeExample1, MapTaskTimeExample2 and MapTaskTimeExample3 illustrate how to achieve this.

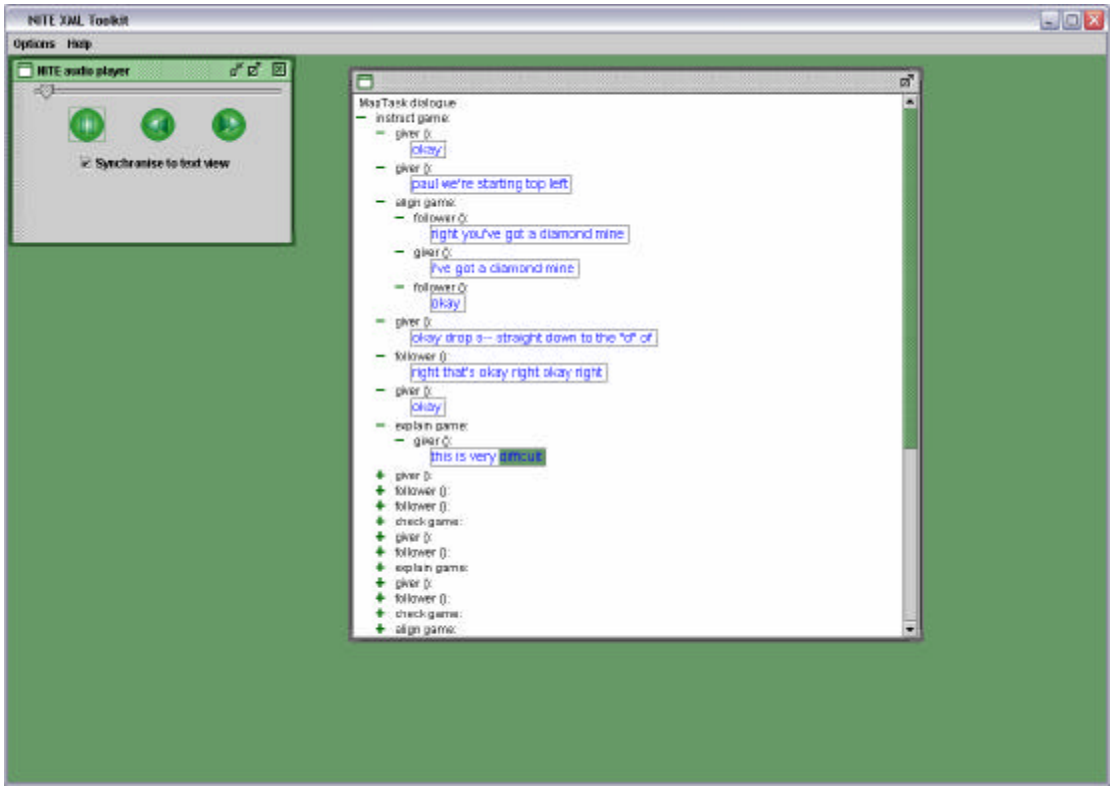


Figure 18 - MapTaskTimeExample1.xsl. A tree containing GridPanels to show the map task dialogue structure. The GridPanels are used to arrange the timed units horizontally next to each other to make them easier to read. The green highlight denotes the transcription

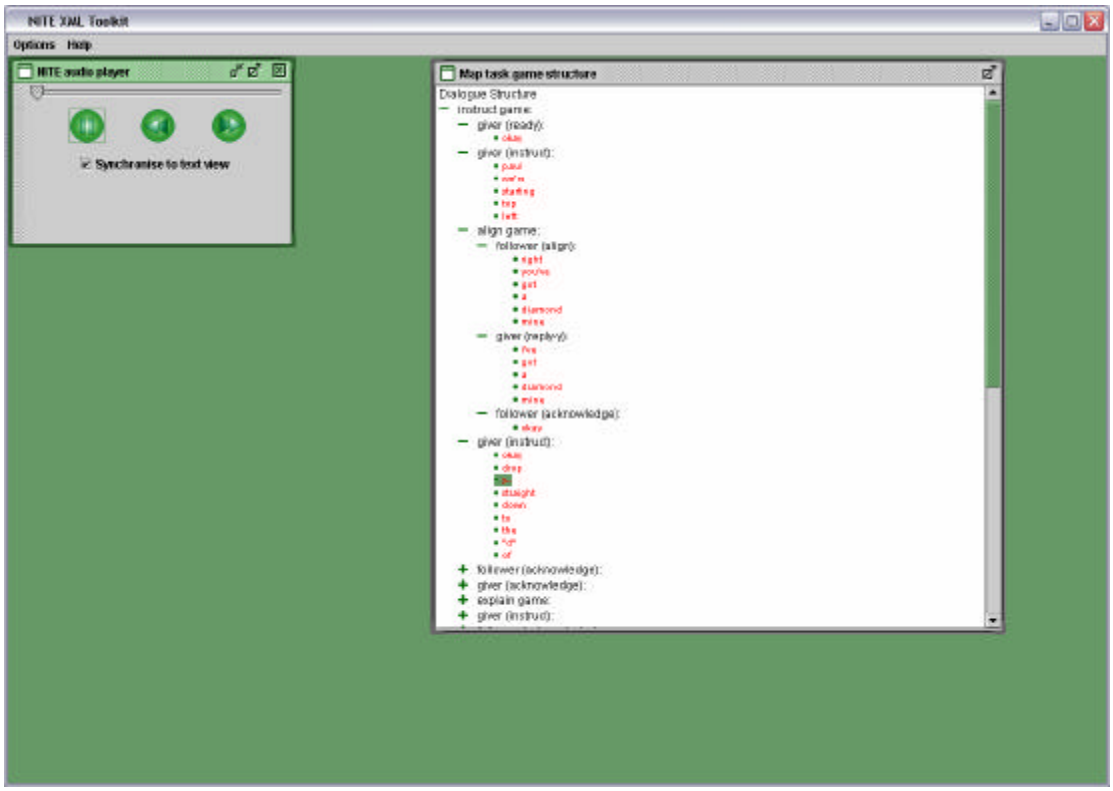


Figure 19 - MapTaskTimeExample2.xsl A Tree showing the map task games and moves. The green highlighted leaf node is the transcript of the currently spoken word on the audio track.

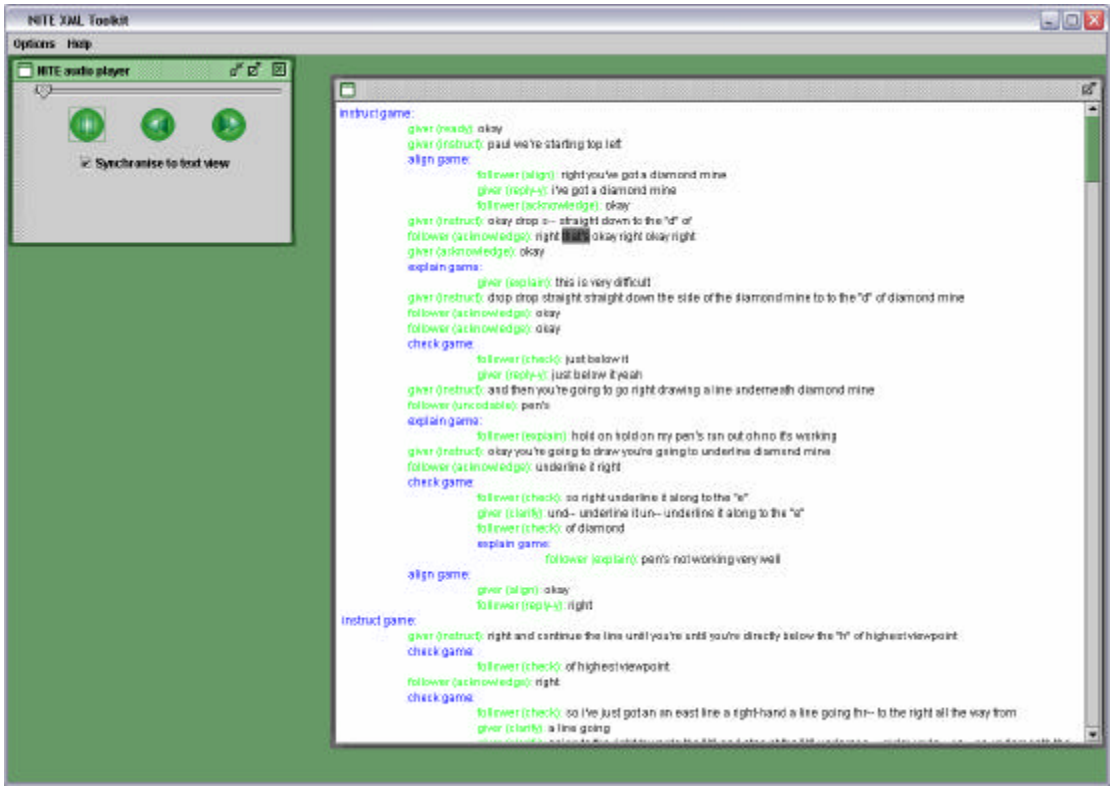


Figure 20 - MapTaskTimeExample3.xsl A TextArea using indentations to show the structure of the dialogue. The Text Elements are aligned to time, so the grey highlight denotes that the audio track has reached an utterance corresponding to that word in the transcription

6. Specifying actions

Introduction

In addition to specifying the layout of NXT interfaces which allow a user to view the data, it is possible to specify edits or actions which the user can perform on the NXT interface. These allow the user to edit the underlying xml data. At the current time, we recommend that stylesheets with action specifications should be used only with small datasets as performance problems inherent in the stylesheet processing approach make action execution very slow. At present, stylesheets with actions can be used on small data sets, or as a prototyping tool. We suggest that users who work with large datasets should use the NITE display objects library to write java programs for displaying and editing data. The java classes for handling editing actions on the XML are used by NXT to create an interface from the xsl specification, and so this document also serves as a useful introduction to the java functionality.

There are five actions which can be performed on the underlying xml data: change textual content of an element; change attribute value in an element; add a child to an element; add a sibling to an element; and delete an element. These actions can be performed on either simple (JDOM) or standoff (NOM) xml files. Examples of each of these actions in the simple xml case will be discussed in the following sections.

There are many ways in which a user could indicate which action should be performed on each element. The main assumption in the user interface design is that the user must select an element by clicking on a representation of it on the user interface with the left mouse button. Once an element is selected the user can invoke

an action by right clicking with the mouse or pressing a key. Some actions can be performed on the underlying data immediately. Others require further input from the user such as a new value for an attribute, or new textual content. The NITE display library provides various basic means of eliciting this information from the user, by pop-up menus and dialogue boxes. We intend to extend this library in the future.

Changing textual content in an element

The simplest example of editing xml from the interface is in ActionDemo1.xsl. Note that many of the examples for actions use the animals.xml data file for the sake of simplicity. The stylesheet specifies an interface which displays each of the animals from the data in a list with the textual content of each animal element in brackets after the animal name. When the user selects a list entry and presses the "change element content" button (see Figure 21), a dialogue pops up inviting the user to type in a new value of the textual content of the element. Once the user presses "Ok", the editing action is executed, the underlying xml is changed, and the whole interface specification is re-processed and redisplayed.

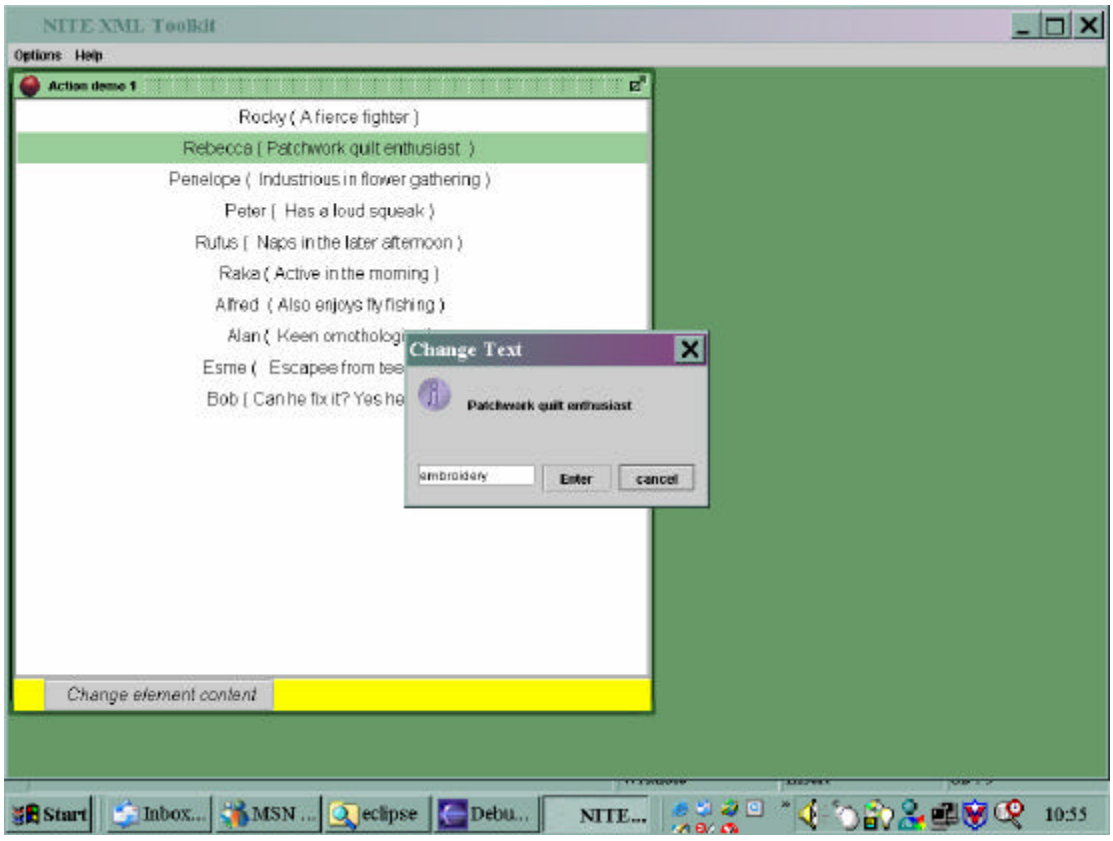


Figure 21 ActionDemo1. Change textual content

This is achieved as follows. Firstly, the action must be specified in the xsl file after the <nite:Root> element, and before the <nite:DisplayObject>s. In this example the action specification looks like this:

```
- <nite:Root>
   - <!-- definition of user input actions -->
- <nite:Actions>
```

```
- <!-- The action which causes the change textual content option pane to appear -->
<nite:Action id="Action1" description="Change textual content for xml element"
    dialoguebox="ChangeTextualContentOptionPane" source="List1" />
</nite:Actions>
```

The only action for this display has the id “Action1”. The id field is required to specify which action should be associated with which display object. The “description” tag is used to explain what the action is intended for. The “dialoguebox” tag specifies which method of getting user input to complete the action will be used. In this case, as the user has to type in new text for the textual content of an element, the ChangeTextualContentOptionPane is used. This display object from the library produces the dialogue box shown in Figure 21. The “source” tag refers to the ID of a display component in which the user will have selected the element for editing. In this case, it refers to List1.

Having specified what should happen when Action1 is executed, we also need to specify how Action1 should be triggered. This is done by specifying an action reference within the display object which should trigger the action to be executed. In this example, the action happens when the button with the id of “Button1” is pressed, as shown below. The nite:ActionReference specifies an actionID which is “Action1”, the ID of the action we set up above.

```
- <nite:DisplayObject id="Button1" type="Button" FontStyle="Italic" Font="Arial" FontSize="16"
    tooltip="Button">
  - <!--
   specifies that the action which is triggered by this button has the ID Action1
   --> <nite:ActionReference actionID="Action1" />
  - <!--
   The content to be displayed on the button
   -->
   Change element content
   </nite:DisplayObject>
```

Change the value of an attribute in an element

The user often needs to change the value of an attribute of an xml element. Example stylesheets ActionDemo2.xsl , ActionDemo3.xsl and ActionDemo4.xsl illustrate three different interfaces for carrying out this action.

Figure 22 illustrates the interface which is built from the specification in ActionDemo2.xsl. Each animal element is displayed on the list with the animal name followed by the value of the “Habitat” attribute in brackets. The user has clicked on the “Change Habitat” button which has triggered the dialogue box to appear. The dialogue box allows the user to enter a new value for the habitat attribute on the selected animal element. Once the user clicks on “enter”, the xml is updated and the interface is redisplayed.

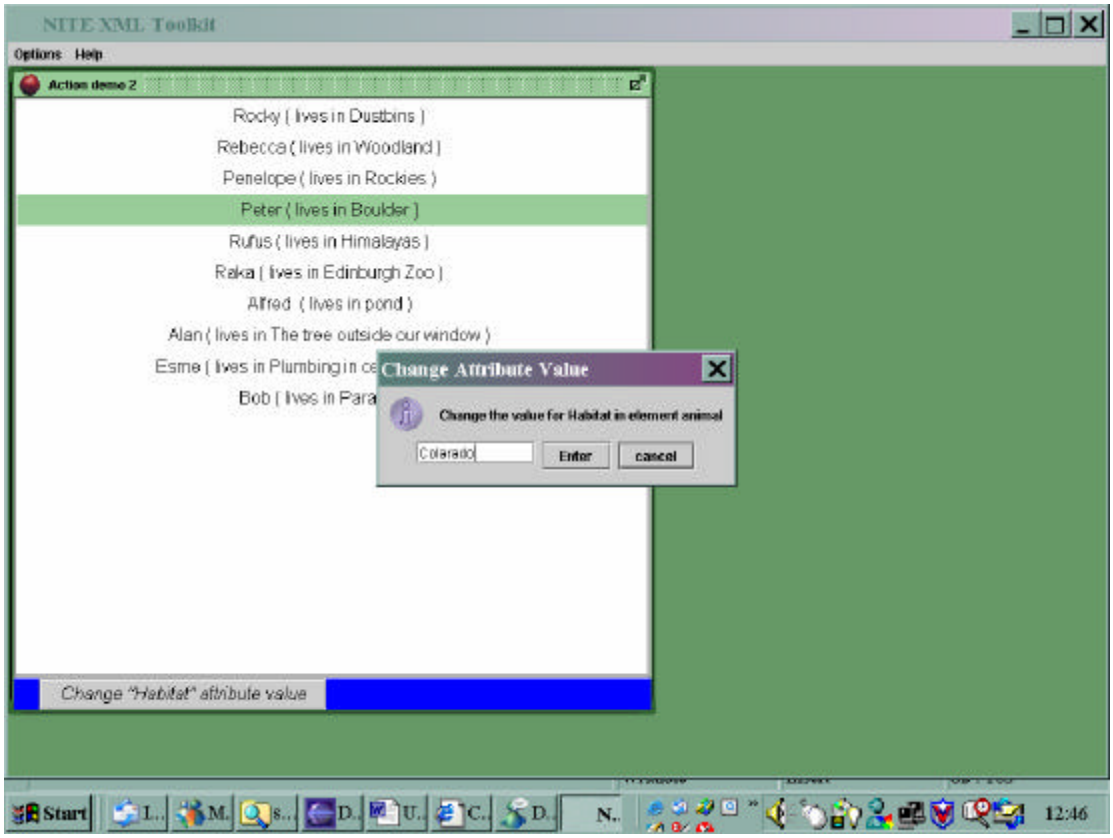


Figure 22 ActionDemo2. Changing an attribute value using an option pane

The action is specified in a similar way to the change textual content action in the previous example. The action specification in the xsl is shown below:

```
<nite:Action id="Action1" description="Change attribute value for xml element"
dialoguebox="ChangeAttributeValueOptionPane" source="List1" attribute="Habitat" />
```

This time the “dialoguebox” tag specifies a ChangeAttributeValueOptionPane display object which is shown in Figure 22. There is an additional tag “attribute” which is used to specify which of the attributes in an element should edited. In this case, the attribute “habitat” can be edited. Note that this assumes that a component will display elements of the same type, or at least that all elements which might be displayed in the component share an attribute specified by this tag. If this assumption does not hold, another of the display objects from the library will be more suitable. The action reference is specified in the same way as in the change textual content example.

Figure 23 shows another way to change an attribute value of an element. This method would be suitable in cases where the attribute values should be constrained rather than freely typed. The list has an entry for every species in the animals xml file, and the covering of the species is denoted inside brackets. The user selects a species with a left mouse click. Once an element is selected, a right mouse click will trigger a popup menu which allows the user to select a new value for the “Covering” attribute: one of fur, feathers, or scales. Once the right mouse button is released, the value of the selected item on the popup menu will be used to update the underlying xml and the interface will be refreshed.

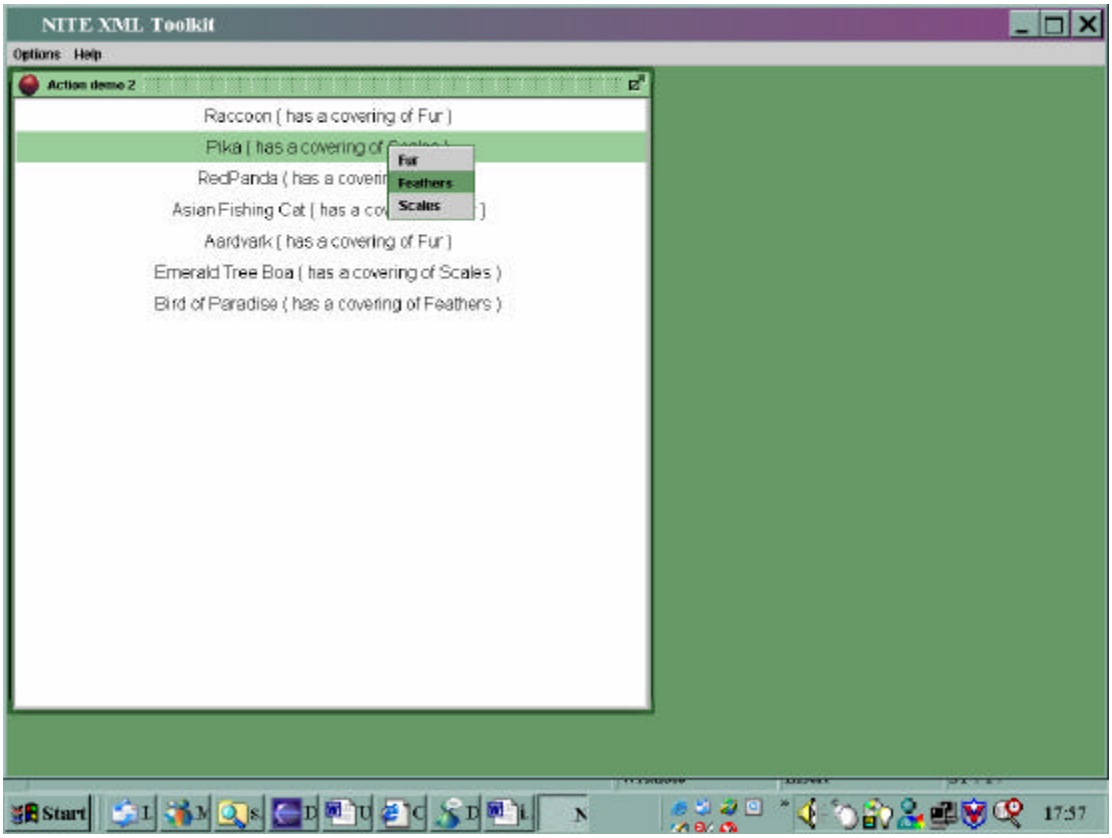


Figure 23 Change attribute value of an element using a pop-up menu

The specification for the interface in Figure 23 is in ActionDemo3.xsl. This time the type of dialogue box used is a "ChangeAttributeValuePopUp" which produces a popup menu. The interesting section of the stylesheet is the specification of nite:PopupContexts. These are used to populate the different options which will appear on the popup menu. The "attribute" tag species that the attribute which will be altered has the name "Covering". The values for tags "option1", "option2" and "option3" will appear on the popup menu. Note that any number of options can be specified for these popup menus – the tags should take the form "optionN", where N is the position in the list where this option should appear.

```
- <nite:Action id="Action1" description="Change attribute value for xml element"
    dialoguebox="ChangeAttributeValuePopUp" source="List1">
- <nite:PopupContexts>
  <nite:PopupContext attribute="Covering" option1="Fur" option2="Feathers" option3="Scales" />
    </nite:PopupContexts>
    </nite:Action>
```

This time the action is triggered by the xsl code shown below:

```
<nite:ActionReference actionID="Action1" keybinding="right_mouse" />
```

As well as specifying which actionID is associated with the component, a keybinding is also specified. The value "right_mouse" indicates that the action should be triggered by a right mouse click. "Left_mouse" would also be valid. The keybinding tag could alternatively be used to specify a keyboard key which triggers an action – this will be discussed further in the DeleteElement example.

Naturally, the actions can be used with components other than lists. Figure 24 shows an interface where data is displayed on a text area, and the user can edit the value of attributes using a popup menu. The animal element from the animal data are displayed with their names in red and their habitat in blue. When the user selects an element and right clicks on it, a popup menu appears. The options in the popup menu depend on which part of the text area is selected. When a habitat is selected, as shown in the figure, the popup menu options are related to habitat – Cannonmills, Leith and New Town (areas of Edinburgh). If the user had clicked on an animal's name, the popup menu would have contained Ringo, John, Paul and George. When the user releases the right mouse button, the xml is updated and the interface is redisplayed.

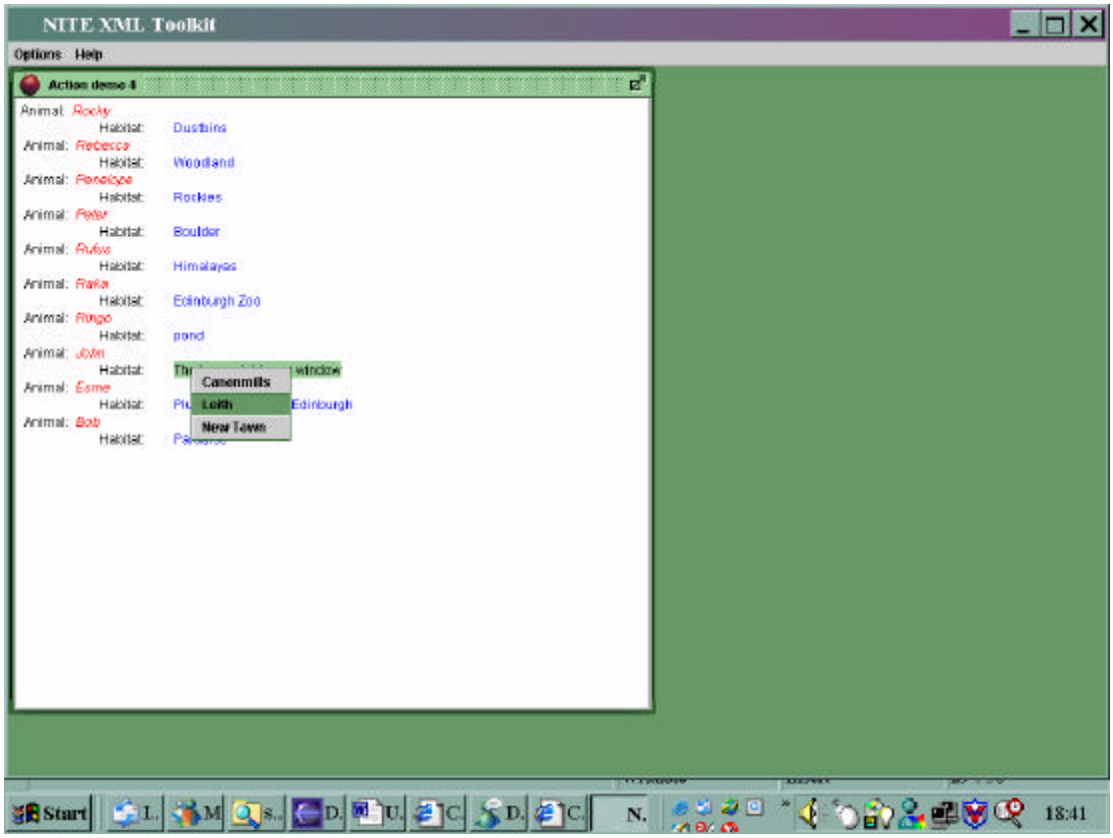


Figure 24 Change attribute values using popup menus on a text area

This example interface is specified in the stylesheet ActionDemo4.xsl. The code which specifies the action is below:

```
<nite:Action id="Action1" description="Change attribute value for xml element"
    dialoguebox="ChangeAttributeValuePopUp" source="TextArea1">
- <nite:PopupContexts>
  <nite:PopupContext attribute="name" option1="Paul" option2="John" option3="Ringo"
    option4="George" />
  <nite:PopupContext attribute="Habitat" option1="Canonmills" option2="Leith" option3="New Town"
    />
    </nite:PopupContexts>
    </nite:Action>
```

It specifies that the source for selected elements which need edited is TextArea1. There are two popup contexts in the action, because the popup menu will display different actions depending on what is selected in the text area. If a name attribute is

selected, the options should be Paul, John, Ringo or George. If a habitat attribute is selected, the popup options are Canonmills, Leith and NewTown.

The context sensitive menu population relies on the text elements knowing which attribute of an element which they are displaying. The same element may be represented in the display by many different objects. In this case, many text elements could display different attributes of the same element. This is accomplished by the "displayAttribute" tag on the text elements, as shown below:

```
<nite:DisplayObject type="TextElement" tabstop="{$tabDepth + 1}" style="habitatstyle"
      displayAttribute="Habitat">
  <xsl:value-of select="@Habitat" />

    </nite:DisplayObject>
```

This specifies that the text element should be indented by one tab stop, that it should be displayed in the habitat text style, and that the attribute of the element displayed by the text is "Habitat". The text itself is the value of the Habitat attribute.

Add a child to an element

Although it is useful to change attribute values of xml elements, the user might also want to edit the structure of the xml document by adding children. Figure 25 shows an example interface where the user can create a new child for an element. The species, animals and young from the animal data are displayed in a tree structure. The user has selected the element Rebecca (this is indicated by the red text colour on the selected node). A right mouse click has triggered the dialogue box for creating a new child to add to the selected element. As Rebecca already has the child Baby, the dialogue box has been populated with the attributes from this element. As Baby has the element name "young", the default value for the new child's element name is also "young". It is also assumed that the new child will need the attributes name, ID and habitat. If the user wanted to add a child to an element with only one child, the dialogue box would have been populated with the attribute names and element name from the parent, Rebecca. We intend to create further library objects for cases where these assumptions do not hold. Once the user has typed in the values for the new child, she can press "Ok", the xml will be updated, and the stylesheet will be redisplayed.

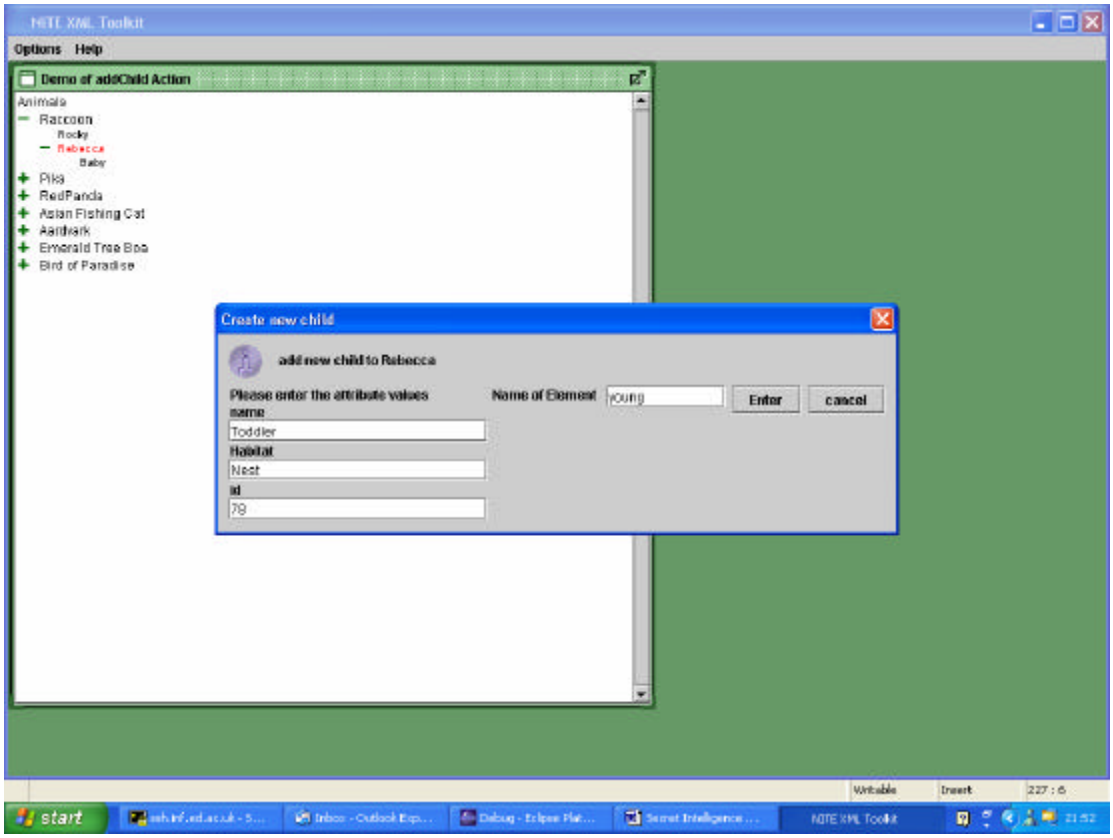


Figure 25 Add a new child to an element

This interface is specified in the stylesheet ActionDemo7.xsl. The code for specifying the action is shown below:

```
<nite:Action id="Action1" description="Add child to xml element"
dialoguebox="AddChildOptionPane" source="Tree1" />
```

The dialoguebox is AddChildOptionPane, which appears as shown in Figure 25. The action reference is specified as follows:

```
<nite:ActionReference actionID="Action1" keybinding="right_mouse" />
```

Each tree node displays an element of data and specifies which attribute of the data it represents on screen, as follows. The IDs for the displayobjects are generated from the element names.

```
<nite:DisplayObject type="TreeNode" id="TreeNode{@name}">
- <nite:DisplayObject type="Label" id="Label{@name}" displayAttribute="name">
  <xsl:value-of select="@name" />
    </nite:DisplayObject>
```

Add sibling to an element

There are cases where it might be more convenient for the user to add a sibling to an element, rather than to add a new child to the parent of that element. This can be achieved with the addSibling action. An example of this action is shown in Figure 26. The user has selected the element “Bob” and has right clicked with the mouse to trigger the addSibling action. The add new sibling dialogue box allows the user to enter attribute values and element name. The attribute names are copies of the

attributes in the selected element, and the element name value defaults to the element name of the selected element. When the user clicks on the OK button, the xml is updated, and the interface is redisplayed.

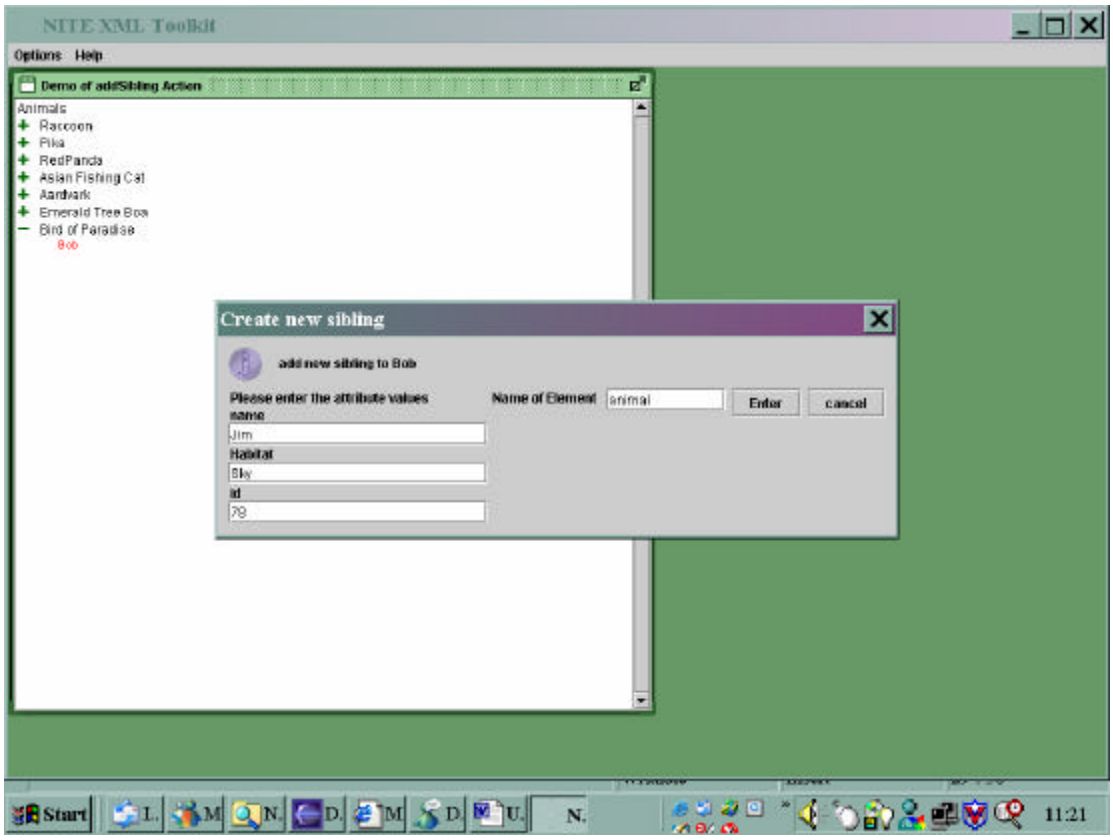


Figure 26 Add a sibling to an element using a tree representation

The stylesheet ActionDemo8.xsl produces the interface shown in Figure 26. The stylesheet is identical to ActionDemo7.xsl, except for the specification of the addSiblingOptionPane instead of the addChildOptionPane:

```
<nite:Action id="Action1" description="Add sibling to xml element"
dialoguebox="AddSiblingOptionPane" source="Tree1" />
```

Delete an element from the xml

Users may also wish to delete elements from the underlying data. There is no screenshot example because it is difficult to depict! An example delete element specification can be found in ActionDemo9.xsl. In this example, the animals from the animal data are represented in a tree view, as in the previous example. When the user selects an element and presses the delete key, the selected element is deleted from the xml, and the interface is redisplayed.

The delete element action is specified with the following xsl:

```
<nite:Action id="Action1" description="Delete an xml element" dialoguebox="DeleteElement"
source="Tree1" />
```

The “dialoguebox” tag is set to DeleteElement. This means that when the action is triggered, the DeleteElement action should be carried out on the selected element (dialoguebox is possibly not the best name for this tag). The action reference is specified as shown below. This time the keybinding, the interface input which triggers the action, is set to “Delete”. This refers to the delete key on the keyboard. The

keybinding tag can be used with any of the other actions and option panes. The key can be set to any keyboard key, such as “F2”, “Space”, “Shift”, or “A” (see http://java.sun.com/products/jdk/1.2/docs/api/java/awt/event/KeyEvent.html#getKeyModifiersText(int)).

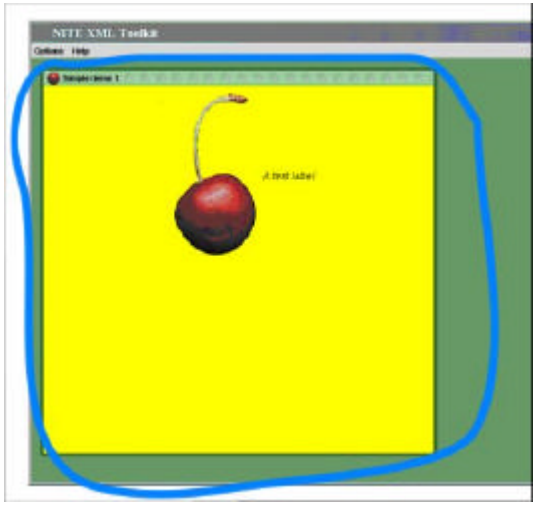
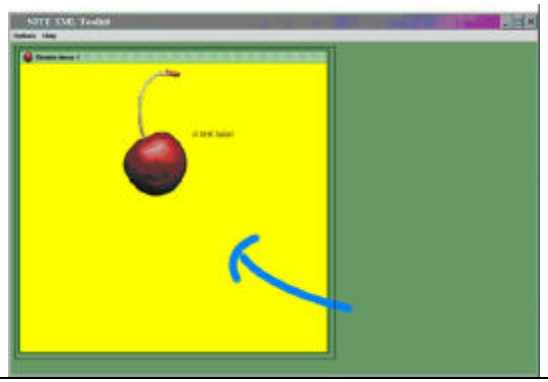
```
<nite:ActionReference actionID="Action1" keybinding="Delete" />
```

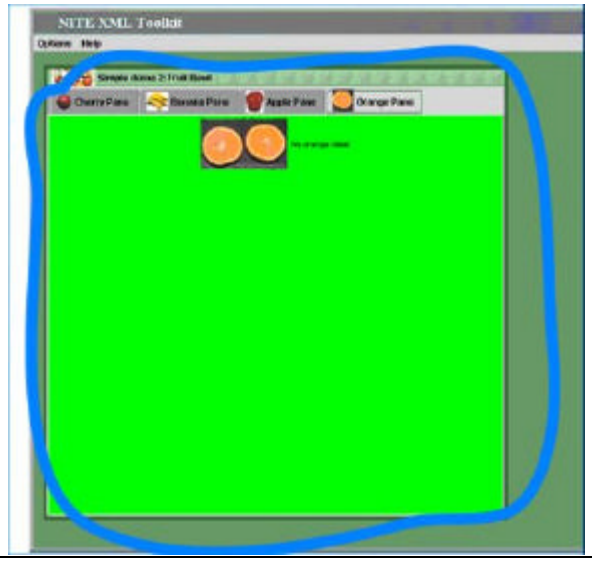
7. Future Work

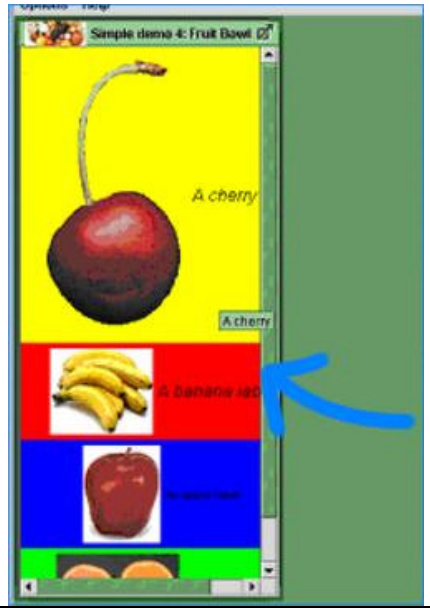
The NITE display library is not yet complete. We intend to add further library components, which will mostly be used for actions. Here are some additional stylesheet examples which we intend to make:

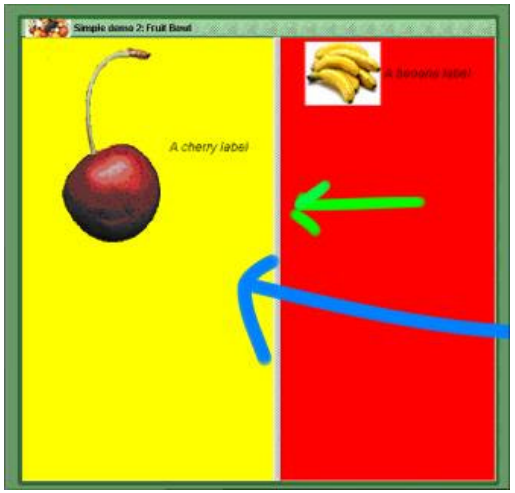
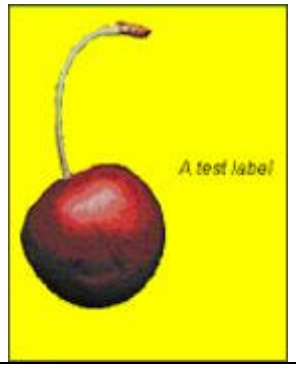
- Example interface with actions for performing all five edits on the xml.
- Example interface which allows the user to alter a tree structure using cut and paste to add children to elements.
- Example interface which allows structural changes on the xml(add child, add sibling and deleteElement) on a text area.
- Example interface which allows add xml edits to be carried out with a grid panel.

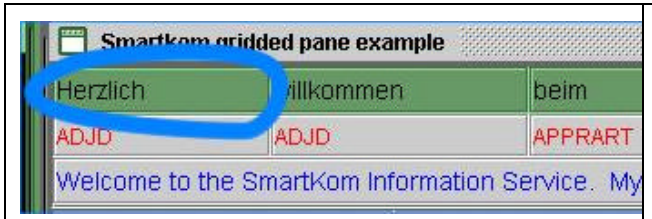
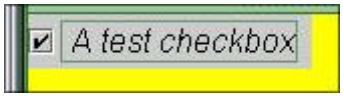
8. Quick Reference Guide

Example	Type	Swing peer	Legal Parent	Legal Children	Parameters
	InternalFrame	JInternalFrame	None	Panel ScrollPane SplitPane TabbedPane Label TimedLabel List Button Checkbox GridPane TextArea Tree	ImagePath background
	Panel	JPanel	InternalFrame Panel ScrollPane SplitPane	Panel InternalFrame ScrollPane SplitPane TabbedPane Label TimedLabel List	background

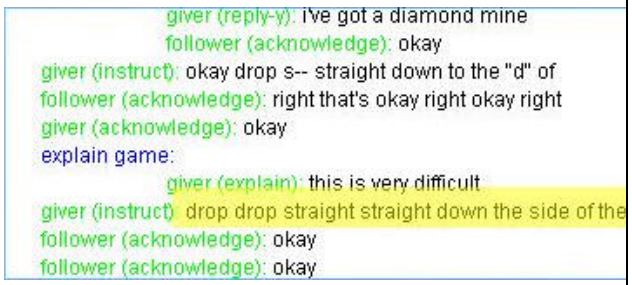
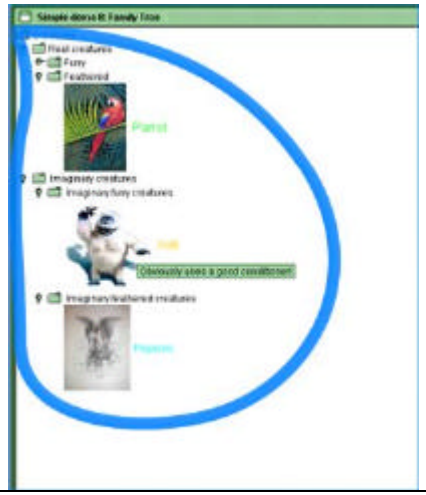
				Button Checkbox GridPane TextArea Tree	
	TabbedPane	JTabbedPane	InternalFrame Panel ScrollPane SplitPane	Panel	background

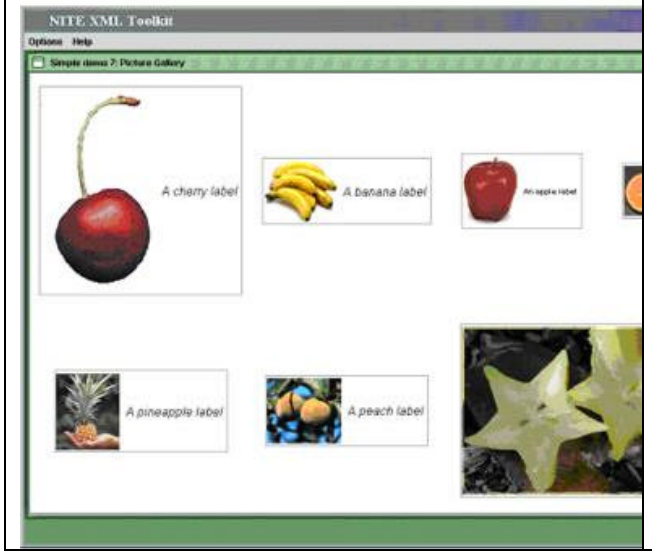
	ScrollPane	JScrollPane	InternalFrame Panel	Panel InternalFrame ScrollPane SplitPane TabbedPane Label TimedLabel List Button Checkbox GridPane TextArea Tree	

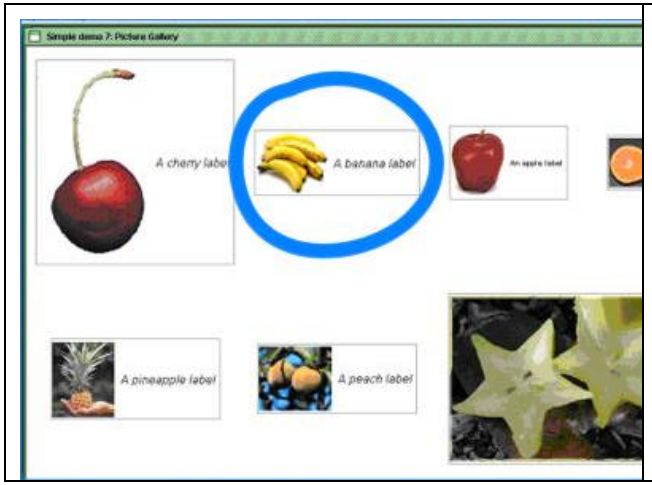
 Note: The green arrow is pointing to the split pane divider (split), which is vertical	SplitPane	JSplitPane	Internal Frame Panel ScrollPane	Panel Restriction: *only two panels may be added to a split pane*	Background split
	Label	JLabel	Any	None	background textcolour FontStyle Font FontSize ToolTip ImagePath

	TimedLabel	NTimedLabel	Any	None	background textcolour FontStyle Font FontSize ToolTip ImagePath Start end
	CheckBox	JCheckBox	Any	None	Background textcolour FontStyle Font FontSize ToolTip

	List	JList	Any	Label	
	TextArea	NTextArea (extension of JTextPane)	InternalFrame Panel	TextElement FontStyle	
Specifies how a piece of text should be printed on screen, not visible by itself	FontStyle	Style	TextArea	None	Name textcolour FontSize

					Font FontStyle
 The text highlighted in yellow is a single TextElement	TextElement	N/A	TextArea	None	style start end
	Tree	NTree (extension of JTree)	Panel ScrollPane InternalFrame	TreeNode	ExpandedImage CollapsedImage OpenedImage ClosedImage

	TreeNode	NTreeNode (extension of DefaultTreeNode)	Tree TreeNode	TreeNode Label TimedLabel GridPane	start end
	GridPanel	A JPanel using the PnutsLayoutManager	ScrollPane InternalFrame Panel	GridPanelEntry	Columns border background

	GridPanelEntry	N/A	GridPanel	Label TimedLabel	RowSpan ColSpan position