Tutorial Booklet

December 1997

Confidentiality and Legal Notice

Please read the following agreement carefully.

> *If you read this document, then you agree to keep the document confidential for three years, with the exclusion that prior knowledge and work already in progress at your company are not within the scope of this agreement should they overlap. You may disclose the document to a small number of your colleagues to evaluate your company's interest in Harmonia's products.*

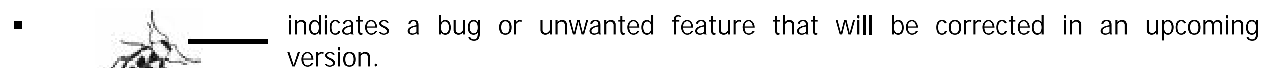
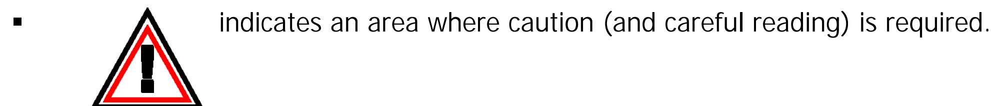
Harmonia, UIML, User Interface Markup Language, and UIMLRenderer are trademarks of Harmonia, Inc.

Java and all Java-based trademarks and logos are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and other countries.

Conventions used.

The examples in this document reflect the state of the UIML renderer as of 5:00pm EST on December 15, 1997.

This document uses several conventions for ease of reading.

- Text that is formatted as "`fixed width`" is a literal value. These values are actually in the UIML examples shown.
- Text that is formatted as "*`italic fixed width`*" are pseudonyms for values that would occur in a UIML document.
-  indicates a bug or unwanted feature that will be corrected in an upcoming version.
-  indicates an area where caution (and careful reading) is required.

Example 1: A Simple dialog.

This very simple interface will illustrate some of the basic concepts of UIML design. Some of these concepts include:

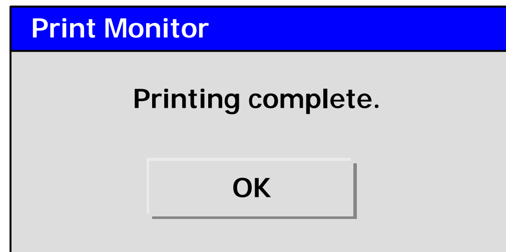
1. UIML language elements and structure.
2. Separation of structure, style and content.
3. Content keys and their usefulness.

Note to the Reader:
This tutorial assumes that you are familier with Java AWT component names, attributes and layout. For further information on available AWT components and attributes see the "Attribute Dictionary".

Design:

First we want to design the dialog. We assume that it is a simple informative. The dialog states a fact and is accompanied by an acknowledgement button. We want it to look something like this.

The Dialog Box has three interface objects: the box (a Frame), the message, and the acknowledgement button. We want to put the message and button inside the dialog and arrange them vertically.



Structure:

The first thing we need to do is create the structure of the dialog. This is done with a user interface (UI) description. UI descriptions look very much like HTML, except that the tags are different. The UI description for this example looks like this:

```
<UIML>
<HEAD>
   <AUTHOR>Stephen Williams</AUTHOR>
   <DATE>December 3, 1997</DATE>
   <VERSION>1.0</VERSION>
</HEAD>
<APP CLASS="App">
   <GROUP CLASS="Dialog" NAME="PrintFinishedDialog">
      <ELEM CLASS="DialogMessage" NAME="PrintFinishedMsg"/>
      <ELEM CLASS="DialogButton" NAME="OKButton"/>
   </GROUP>
</APP>
<DEFINE NAME="OKButton">
   <PROPERTIES>
   <ACTION
       VALUE="PrintFinishedDialog.EXISTS=false"
```

 © 1997 Harmonia, Inc

```
        TRIGGER="select"
    />
    </PROPERTIES>
</DEFINE>
</UIML>
```

The entire document is enclosed in a <UIML>...</UIML> tag pair. UIML is an XML compliant language and follows the convention of enclosing all information in descriptive tags in a nested fashion. This is very much like HTML with a few exceptions.

Note: The syntax for the UI definition is: <TAGNAME> all things belonging to the TAG </TAGNAME>. There are no restrictions about staying on the same line or indentations.

There are three primary sections to a UI definition file. They are

HEAD: The <HEAD> section contains information about the file itself. The <AUTHOR>, the <DATE> and the <VERSION> are the currently supported values that can be contained in <HEAD>

APP: The <APP> section contains the actual structure of the interface. The two basic elements <GROUP> and <ELEM> allow the designer to create a structure without filling in all of the particulars. <APP> requires a CLASS in order for the Style to properly render it. In this case we just make CLASS="App". We can also give the APP a NAME so that other APPS can control it.

DEFINE: The <DEFINE> section defines the named groups and elements from the APP section in more detail. In this example, we define what action occurs when we "click" on the OKButton element.

Let's go through the <APP> section one line at a time and talk about each of the tags and their properties.

<APP CLASS="App">

The APP tag begins the official UI definition for this application. Although a dialog box may not be considered a separate application by most, it is a required convention to have an <APP>...</APP> in each UI definition file. You can, of course define multiple frames within a single UI definition.

<GROUP CLASS="Dialog" NAME="PrintFinishedDialog">

The tag GROUP indicates that this is a group of UI objects. In other words, there are other things nested within the group. While it is not absolutely necessary for their to be ELEMS or other GROUPS defined within the group, it is a good convention to follow.

The CLASS attribute is used to define a "CLASS" of UI objects that this GROUP belongs to. The value for CLASS can be any single alphanumeric word.[†] The CLASS value must be given a RENDERING in the style for the UI object to render. In this case the chosen value for the CLASS is "Dialog". By choosing names carefully, a single style can server many ui descriptions.

The NAME attribute is used to give this particular GROUP a name. As with CLASS, the value of NAME is chosen by the designer. The NAME given here is "PrintFinishedDialog".

[†] 32 characters or less, beginning with a letter and including only letters numbers and underscores '_'.

© 1997 Harmonia, Inc

At first glance, there may not seem to be a difference between NAME and CLASS. However, there can be many elements with the same CLASS value, but only one element with a particular NAME. If you want to create a group of buttons to perform some functions, you would give them all the same CLASS but different NAMES. That way, the style can define the RENDERING of that class as "Button" but put in a different LABEL for each NAME. You can also use the NAME to call different functions for each of the buttons.

<ELEM CLASS="DialogMessage" NAME="PrintFinishedMsg"/>

This next element is an ELEM, which means that it does not contain any other UI objects. As with GROUP the CLASS and NAME are given values by the designer that indicate that the ELEM is a member of a CLASS of UI objects that has a certain look and that it has a unique NAME.

You may have notice that the tag is syntactically different from the GROUP. The GROUP tag starts as <GROUP ...> and the ELEM tag is <ELEM..../>. This is because the tag does not actually enclose anything. It stands alone. This is one of the places where UIML is different from HTML. HTML is not XML compliant because you can have a <P> tag with no </P> which would not be allowed in XML. The language parser understands that the next value after a <*tagname*/> is not within the previous tag but a "sibling" to it. If the above line had been <ELEM CLASS="DialogMessage" NAME="PrintFinishedMsg"> (without the / before the closing >) then the parser would assume that the following line is a nested member of the <ELEM>, which would be an error because ELEMS cannot enclose other UI objects.

<ELEM CLASS="DialogButton" NAME="OKButton"/>

Another ELEM.

The order that these UI objects appear can be important. In this particular example it doesn't because "PrintFinishedMsg" and "OKButton" are positioned in the style (next section) with the layout manager. Some layouts, however, do not require each element to be positioned. If the layout were FlowLayout or GridLayout then "PrintFinishedMsg" would be to the left of "OKButton" because it is in the ui definition first.

Because FlowLayout and GridLayout exhibit this property, it is best to assume that the layout is partially determined by the order of element definitions.

</GROUP>

An end GROUP tag finishes out the Dialog definition. Any tag that encloses other tags or content must be ended in the form of "</*tagname*>".

</APP>

An end of APP tag finishes out the APP definition.

Now that the structure of the Dialog Box has been created, we can move on to defining the actions of this interface. In this example only one element requires an action definition, the "OKButton". The action that the button performs is shown in a DEFINE.

<DEFINE NAME="OKButton">

A <DEFINE> tag allows the designer to add more detail to a previously NAMEd UI object. This gives the designer the ability to explain the interface within the <APP>..</APP> tags in a brief and concise fashion so that others can easily grasp the structure without having to wade through the details.

 © 1997 Harmonia, Inc

<PROPERTIES>

A <PROPERTIES> tag allows the designer to add add certain information about the responses that a UI object make when an event occurs.

<ACTION VALUE="PrintFinishedDialog.Exists=false" TRIGGER="select" />

A <ACTION> tag defines actions that can be taken by this UI element on other UI objects. In this case the action taken is to change the attribute called Exists on the entire dialog box when the button is "clicked".

The VALUE attribute contains a list of element attribute settings. In other words, the action performed is to change attributes on UI objects, including the element performing the action. NOTE, that the only attributes that can be currently changed are "Visible", "Exists" and "Enabled". Where "Visible=true" means that the UI object can be seen on the screen. "Enabled=true" means that the UI object is in the renderer, setting it the last frame to "false" exits the renderer. "Enabled=true" means that the element is active, "Enabled=false" means it's inactive (grayed-out buttons and menu items in Windows programs are inactive). The form of the attribute setting is "*ElementName.Attribute=value*"

</PROPERTIES>

The </PROPERTIES> tag closes the property creation of the "OKButton" element.

</DEFINE>

The </DEFINE> tag closes the definition of the "OKButton" element.

Style:

Now that structure of the dialog box has been determined we can move on to style. The power of UIML is in the ability to completely change the look of a screen of arbitrary complexity by changing only the style definitions. This means we can define many different applications or screens and have them all use the same style.

The syntax of UIML style is different from that of the structure to parallel CSS (Cascading Style Sheets) which is available for HTML documents (this may change to an XML compliant style in the future).

Style defines how the elements described in the ui definition will be displayed on the screen. This includes both minor attributes such as Color or Size as well as the type of element such as Button or List. So a group of 5 commands, that are related, could be rendered as 5 buttons, or as a radio-group with 5 choices, or as a menu with 5 menuitems.

 © 1997 Harmonia, Inc

In this case all we want is an AWT Frame with a Label and a Button. UIML style can define default attributes that will cascade to all "descendant" elements. There are two style elements that MUST be defined. Those are "TOOLKIT" and "RENDERING". These two items must be defined for all elements for them to render. The following is the style definition for the described dialog box:

```
APP.App{
    +TOOLKIT:              jfc;
    +RENDERING-PREFIX:     java.awt;
}
GROUP.Dialog {
    +RENDERING: Frame;     /* This is an AWT frame */
    +LAYOUT:    BorderLayout;
    +FONT-NAME: "sanserif";
    +FONT-SIZE: "11";
    SIZE:      "150,105";
    +CONTENT:  "Error: No Label";
    +BACKGROUND:     "lightgray";
}

ELEM.DialogMessage{
    RENDERING:   "Label";
    FONT-STYLE:  "bold";
    ALIGNMENT:   "Center";
}

ELEM.DialogButton{
    RENDERING:   "Button";
    FONT-STYLE:  "Plain";
    ALIGNMENT:   "South";
    SIZE:        "80,25";
}
```

Note that the style is defined based on the `CLASS` of the elements, not the name. This allows the designer to change ALL of the buttons into a menu by changing one or two lines.

The syntax of the style is radically different from that of the ui definition. Each style definition is for a single `CLASS`. All attribute ←→ value pairs are enumerated inside a set of "`{}`". The order of the attribute ←→ value pairs is not important. If the attribute is preceded by a "+" sign, then that value is "scoped" to all decendents. In this example the `Font-Face` applies to all elements in the dialog box because the `GROUP.Dialog` is the parent of all elements.

Now let's do as before and go through the style one line at a time.

`GROUP.Dialog {`
This line starts the style for a particular class. Note that the `CLASS` is denoted as *`UITAG.CLASSNAME.`* This will provide future flexibility in naming schemes and style assignments (e.g. you can have a `GROUP` and and `ELEM` with the same `CLASS` name but rendered differently).

After denoting the CLASS, we use an "{" to start the style definition.

`+TOOLKIT:   jfc;`
The toolkit is the name of the GUI toolkit used to put the elements on the screen. Currently only jfc is offered. Note that a "+" precedes the attribute name TOOLKIT. This means that all enclosed elements will also have a jfc TOOLKIT.

 © 1997 Harmonia, Inc

The act of passing an attribute to the child components is also known as "scoping". Note that scoping is not quite the same as inheritance. Scoping is syntactically based, whereas inheritance is based on programming object type.

The syntax of a style assignment is *ATTRIBUTENAME: value;*

+RENDERING-PREFIX: java.awt;
The RENDERING-PREFIX is the prefix to the standard Java classname. All AWT components begin with java.awt. This is merely a typing reduction mechanism. The two prefixes supported now are java.awt and com.sun.java.swing. Note, currently, if you mix awt and swing components you must put the full class name for each RENDERING.

+RENDERING: Frame; /* This is an AWT frame */
This is the Java component you wish to use in the interface for the particular class. In this case a Frame is desired. Frames are separate windows that can be moved, iconified and destroyed. Currently, all other user interface components MUST be contained within a Frame (e.g. a Frame is it's farthest ancestor).

This line also has a comment. Style comments look just like C or Java comments. Comments in the ui definition look like <!--comment goes here -->.

+LAYOUT: BorderLayout;
A Frame is a container and therefore has a layout. If no layout is specified the default is FlowLayout. Here, we want a BorderLayout because we only have two elements and one is below the other. We could have chosen GridLayout with one column if we wanted.

Layout fundamentals and layout design are discussed in another document.

+FONT-NAME: "sanserif";
When text is used, we want it to be a Serif or "footed" font. Recall that the "+" sign means that all enclosed elements will also use a Serif font unless they override it. There are five FONT-NAMES: "serif", "Sanserif", and "monospaced" look like they do here. "dialog" and "dialoginput" depend on the computer setting for those types in the operating system.

+FONT-SIZE: "11";
The FONT-SIZE is the height of the text in points. *Note: 72 points is 1 inch.*

SIZE: "150,105";
This is the SIZE of the Frame in pixels the X (or horizontal) size is first and the Y (or vertical) size is second (X,Y). Note that there is no "+" sign here. Scoping the size of a container to child components can have odd effects on the interface if you should forget to set the size of those children. We recommend that the SIZE never be scoped.

+CONTENT: "Error: No Label";
This is the text that goes on the border of the Frame. In this case, I the text is an error message. Doing this allows me to visually check an interface for missing labels in individual components. The individual components are given their text labels in the content database, which will be discussed in the next section.

+BACKGROUND: "lightgray";
This sets the background color for the Frame (and all descendants).

}

 © 1997 Harmonia, Inc

When we have finished the style definition for the "`Dialog`" we close with a "`}`".

`ELEM.DialogMessage{`
Now we start the style definition for the "`DialogMessage`". This is a textual label.

`RENDERING:   "Label";`
We render this as an AWT label. Note that we could have called this a Button but the button would not do anything since we did not define any action for it in the ui definition.

`FONT-STYLE:  "bold";`
I decided that this label should be bold. The possible styles are "plain", "**bold**", "*italic*", and "***bolditalic***".

`ALIGNMENT:   "Center";}`
This message is to appear in the "`Center`" of the parent `Frame` according to the rules of `BorderLayout`. Since this is the last style attribute for the message, we close the definition.

`ELEM.DialogButton{`
Now we define the button.

`RENDERING:   "Button";`
It is rendered as an AWT Button.

`FONT-STYLE:  "Plain";`
Make the font plain. This isn't required because plain is the default.

`ALIGNMENT:   "South";}`
This button is to appear in the "`South`" of the parent `Frame` according to the rules of `BorderLayout`. Since this is the last style attribute for the message, we close the definition.

Content:

Although the style has listed HOW a particular class will look on the screen, there are certain features that have not been addressed. The style features that have not yet been inserted are those things that change based on the actual element rather than the class of the element. In other words, style information that is attached to the NAME of the element rather than the CLASS of the element. These declarations are put in a "Content Database". Most of the time labels for buttons, menuitems and text areas go here. Because of that, the content database also has a "key" which allows the designer to use multiple languages for the same application.

The Content database for this example is short because there are only three elements, the Frame, the message and the button.

```
# Dialog Example
# Author: Stephen Williams
# Date: December 3, 1997
#
# Record format:  Key, Name (from .uiml file), Attribute, Value
#
English PrintFinishedDialog    CONTENT Print Monitor
English PrintFinishedMsg       CONTENT Printing Complete
English OKButton               CONTENT OK
```

The format of the content database is:

```
Key    ElementName       Attribute  Value
```

Note that the separator is white space. But, because we know there are only four items, the "Value" can have white space without causing problems.

We have now finished the design. Assuming you name your files Ex1.ui Ex1.style and Ex1.cdb respectively, you can now look at the result by running the command.

```
java Render Ex1.ui Ex1.style Ex1.cdb -key English
```

The finished product does not look EXACTLY like what was envisioned, but is pretty close (it will get better as more functionality is added). If you click on the OK button, the frame disappears and the renderer exits.

 © 1997 Harmonia, Inc

Example 2: A Paint program

This interface will illustrate some of the powerful features that UIML allows. Some of these concepts include:

1. More UIML language elements and structure.
2. Multiple styles for different interfaces.
3. Advanced styles using Swing.
4. Quick changes to modify the look of your interface.

Design:

We would like to design a paint program. Of course we will design the interface in UIML to maximize the flexibility once the program is complete. Noting that we may want to use the paint program on a palm-top or PDA, we will want the ability to create an interface with standard menus, and another with a "toolbar" that has iconized buttons.

To do this we assume that certain "commands" will always be in the menubar and other items will either go in the menubar or into a toolbar. Therefore we want two classes for these items.

Structure:

The following is the ui description for the paint program.

```
<UIML>
<HEAD>
  <TITLE>Paint Program..loosely based on MS Paint</TITLE>
  <AUTHOR>Stephen Williams, Harmonia</AUTHOR>
  <DATE>12/2/97</DATE>
  <VERSION>0.5</VERSION>
</HEAD>

<APP NAME="Paint" CLASS="App">
  <GROUP NAME= "MainFrame" CLASS="frame">

   <GROUP NAME="File" CLASS="menu">
     <ELEM NAME="New" CLASS="menuitem"/>
     <ELEM NAME="Quit" CLASS="menuitem"/>
     <ELEM NAME="Close" CLASS="menuitem"/>
   </GROUP>
   <GROUP NAME="Tools" CLASS="menuortoolbar">
     <ELEM NAME="Line" CLASS="itemoricon"/>
     <ELEM NAME="Box" CLASS="itemoricon"/>
     <ELEM NAME="Oval" CLASS="itemoricon"/>
     <ELEM NAME="Trap" CLASS="itemoricon"/>
     <ELEM NAME="Eraser" CLASS="itemoricon"/>
     <ELEM NAME="Pencil" CLASS="itemoricon"/>
     <ELEM NAME="Brush" CLASS="itemoricon"/>
     <ELEM NAME="Sprayer" CLASS="itemoricon"/>
     <ELEM NAME="Text" CLASS="itemoricon"/>
     <ELEM NAME="Magnifier" CLASS="itemoricon"/>
   </GROUP>
   <GROUP NAME="Help" CLASS="menu">
     <ELEM NAME="PHelp" CLASS="menuitem"/>
     <ELEM NAME="About" CLASS="menuitem"/>
   </GROUP>

   <ELEM NAME="Painting" CLASS="PaintArea"/>

  </GROUP>
</APP>
```

 © 1997 Harmonia, Inc

```
<DEFINE NAME="Quit">
  <PROPERTIES>
    <CLASS VALUE="menuitem"/>
    <ACTION
        VALUE="MainFrame.VISIBLE=false"
        TRIGGER= "select"
    />
  </PROPERTIES>
</DEFINE>
</UIML>
```

You will note that their are two CLASSes called "menu" and "menuortoolbar". This allows us to make use two different styles to change the look of all the items with that class. Similarly, there are "item"s and "itemoricon"s.

Style:

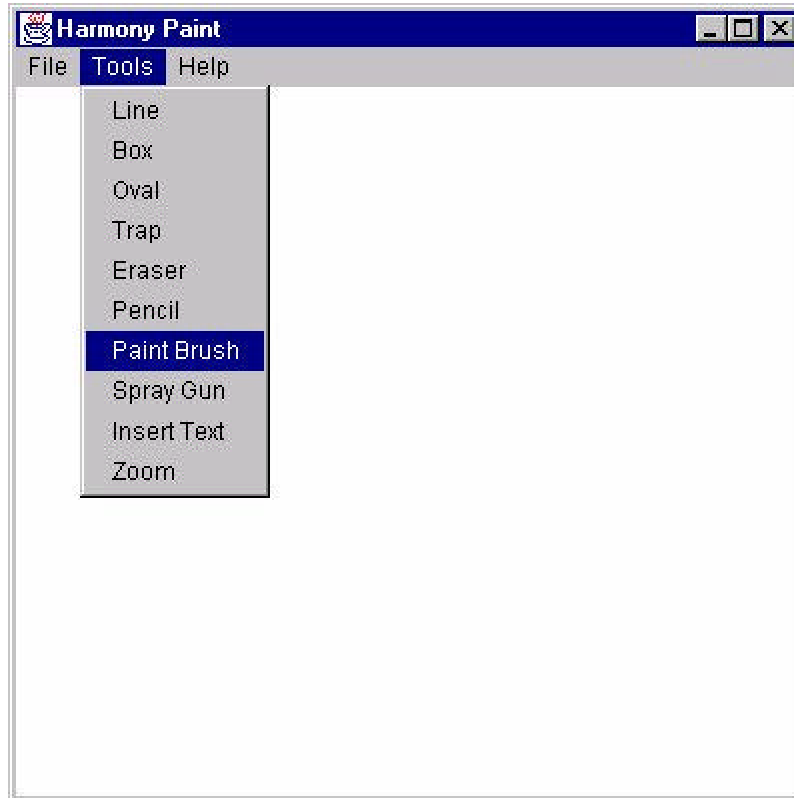
The following style shows how we can define the same RENDERING for multiple style so that they look all the same in the interface. In this case both the "menu" and "menuortoolbar" CLASSes will be rendered as "Menu"s.

```
/*
  <AUTHOR>Stephen Williams, Harmonia</AUTHOR>
  <DATE>12/2/97</DATE>
  <VERSION>0.5</VERSION>
*/
APP.App{
    +TOOLKIT:            jfc;
    +RENDERING-PREFIX:   java.awt;
}
GROUP.frame {          /* From content database:  CONTENT */
  RENDERING: "java.awt.Frame";
  LAYOUT:    BorderLayout;
  SIZE:      "400,400";
  FONT-FACE: Serif;
  FONT-SIZE: 10;
  FONT-STYLE:Plain;
  CONTENT:   "Error:No Content";
}

GROUP.menu {           /* From content database:  CONTENT */
  RENDERING: "java.awt.Menu";
}

ELEM.menuitem {               /* From content database:  CONTENT, ACCKEY */
  RENDERING: "java.awt.MenuItem";
}
GROUP.menuortoolbar {  /* From content database:  CONTENT */
   RENDERING: "java.awt.Menu";
}

ELEM.itemoricon {             /* From content database:  CONTENT, ACCKEY */
   RENDERING: "java.awt.MenuItem";
}
ELEM.PaintArea {     /* From content database:  CONTENT, SIZE */
  RENDERING: "java.awt.Panel";
  ALIGNMENT: Center;
}
```

Because the CLASSes have the same rendering, the rendition of the items in the interface is as menus and menuitems for both.

 © 1997 Harmonia, Inc

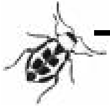
Those of you that use Java AWT often may notice that their is no mention in either the ui definition or the style of a "`MenuBar`". This feature is taken care of automatically. Since a frame can have, at most, one `MenuBar`, it is assumed that any `GROUP`s with a `RENDERING=Menu` go into that one MenuBar. This allows you to assign menus at any point...or change the `RENDERING` of a



`GROUP` of buttons (or something else) to a Menu without having to rearrange the ui definition.

All of the items with `CLASS=menuitemoricon` show up in the Tools menu. Because the Help menu information comes after the Tools info in the ui definition, the Help menu comes third.

Recall that we would also like create an interface with a toolbar as well. We can reuse the ui definition with no modifications. However, we need some extra elements that are not available in AWT. These elements include toggle buttons and the ability to put icons in the buttons instead of text. For this we will need to use Swing (the JFC widgets for jdk 2.0).



The new style is listed below. You will immediately note that the `RENDERING-PREFIX` is not used (the `APP.App` requires it but it starts with a `'-'` sign), but the fully qualified name is inserted for each `RENDERING`. This is required now, but will not be in future versions.

```
/*
  <AUTHOR>Stephen Williams, Harmonia</AUTHOR>
  <DATE>12/2/97</DATE>
  <VERSION>0.5</VERSION>
*/

APP.App{
    +TOOLKIT:            jfc;
    -RENDERING-PREFIX:   java.awt;
}
GROUP.frame {         /* From content database:  CONTENT */
  RENDERING: java.awt.Frame;
  LAYOUT:    BorderLayout;
  SIZE:      "400,400";
  FONT-FACE: Serif;
  FONT-SIZE: 10;
  FONT-STYLE:Plain;
  CONTENT:   "Error:No Content";
}

GROUP.menu {
  RENDERING: "java.awt.Menu";
}

ELEM.menuitem {
  RENDERING: "java.awt.MenuItem";
}

GROUP.menuortoolbar {
  RENDERING: "java.awt.Panel";
  SIZE:      "400,20";
  ALIGNMENT: "North";
  LAYOUT:    GridLayout;
  +BACKGROUND: "lightgray";
  ROWS:      1;
}

ELEM.itemoricon {
  RENDERING: "com.sun.java.swing.JToggleButton";
  SIZE:      "30,30";
  CONTENT:   " ";
  TEXT-HORIZONTAL: Center;
}

ELEM.PaintArea {
  RENDERING: "java.awt.Panel";
  ALIGNMENT: "Center";
}
```

The new style is listed below. You will immediately note that the `RENDERING-PREFIX` is not used. The current version of UIML does not allow `RENDERING-PREFIX` to be used with mixed class paths in the same style. This will change in the next release.

A few notes on choices. In general, it is better to pick the AWT component if you don't need the extended features of Swing. This is especially true for contains (e.g. panel, frame, window). In this case the only Swing component is the `JToggleButton` which gives us two new capabilities; embedded images and maintaining button state. There is also an added bonus ---Tool Tips---.

You may have noticed that the Frame layout is `BorderLayout` and the toolbar is in `GridLayout`. Why not `FlowLayout`? The answer is that, right now, there is a conflict with the `ALIGNMENT`

 © 1997 Harmonia, Inc

attribute. In `GridBagLayout` and `BorderLayout`, the `ALIGNMENT` attribute is associated with the children of the layout `GROUP`: but, in `FlowLayout`, the `ALIGNMENT` is associated with the actual layout `GROUP`. Because `BorderLayout` uses "North,East,West,South, and Center" as values, and `FlowLayout` uses "Left,Center and Right", there would be a name conflict. This problem will be fixed in later versions.

We have waited till now to show you the Content Database because it has all of the values for both the Toolbar and Menu version. We can do this and use the "`-key`" to distinguish between them (I used "`NoSwing`" and "`Swing`" as keys).

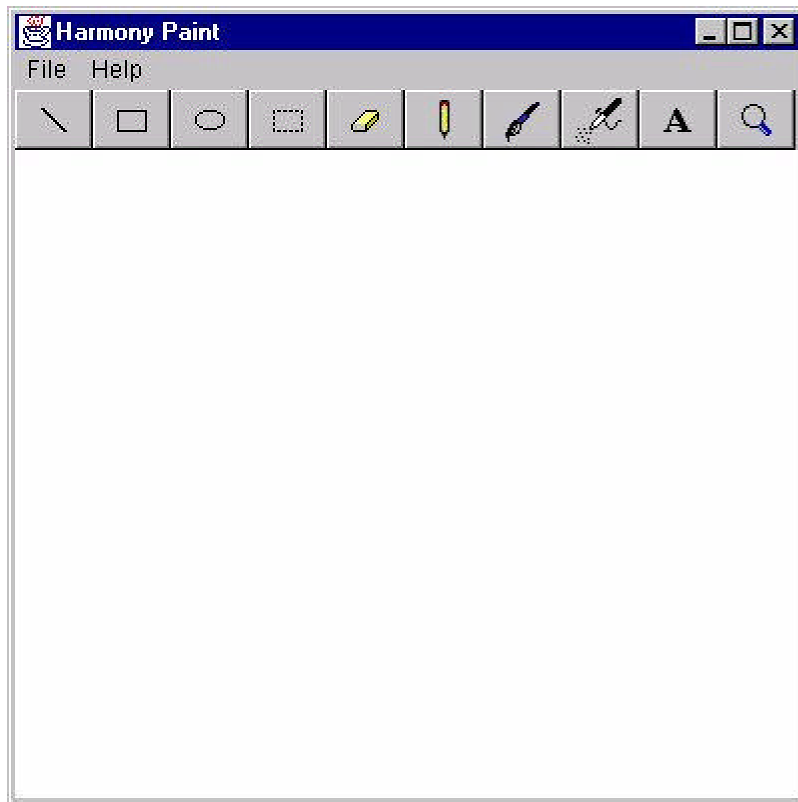
```
# Records to initialize content data base for Ex8-1 from Nutshell book.
#
# Stephen Williams (12/2/97)
#
# Record format:  Key, Name (from .ui file), Attribute, Value
#
NoSwing MainFrame CONTENT Harmony Paint
NoSwing File      CONTENT File
NoSwing New       CONTENT New
NoSwing Quit      CONTENT Quit
NoSwing Close     CONTENT Close
#
#-------Tools Menu--------
NoSwing Tools     CONTENT Tools
#
NoSwing Line      CONTENT Line
NoSwing Box       CONTENT Box
NoSwing Oval      CONTENT Oval
NoSwing Trap      CONTENT Trap
NoSwing Eraser    CONTENT Eraser
NoSwing Pencil    CONTENT Pencil
NoSwing Brush     CONTENT Paint Brush
NoSwing Sprayer   CONTENT Spray Gun
NoSwing Text      CONTENT Insert Text
NoSwing Magnifier CONTENT Zoom
#
#-------Help Menu--------
NoSwing Help      CONTENT Help
#
NoSwing PHelp     CONTENT Contents and Index
NoSwing About     CONTENT About Paint
#
NoSwing Painting SIZE    400,400
#--------------------SWING STUFF
Swing MainFrame CONTENT Harmony Paint
#
Swing File      CONTENT File
Swing New       CONTENT New
Swing Quit      CONTENT Quit
Swing Close     CONTENT Close
#-------Tools Menu--------
#Swing Tools     CONTENT Tools
#
Swing Line       IMAGE   Icons/Line.gif
Swing Line       TOOLTIP Line
Swing Box        IMAGE   Icons/Box.gif
Swing Box        TOOLTIP Rectangle or "Shift" for Square
Swing Oval       IMAGE   Icons/Oval.gif
Swing Oval       TOOLTIP Oval or "Shift" for Circle
#
Swing Trap       IMAGE   Icons/Trap.gif
Swing Trap       TOOLTIP Object Trap
Swing Eraser     IMAGE   Icons/Eraser.gif
Swing Eraser     TOOLTIP Eraser
Swing Pencil     IMAGE   Icons/Pencil.gif
Swing Pencil     TOOLTIP Draw with sharp pencil
Swing Brush      IMAGE   Icons/Brush.gif
```

 © 1997 Harmonia, Inc

```
Swing Brush      TOOLTIP Draw with paintbrush
Swing Sprayer    IMAGE   Icons/Sprayer.gif
Swing Sprayer    TOOLTIP Draw with spray gun
Swing Text       IMAGE   Icons/Text.gif
Swing Text       TOOLTIP Add text
Swing Magnifier  IMAGE   Icons/Magnify.gif
Swing Magnifier  TOOLTIP Zoom In/Out on image
#
# -----Help Menu
Swing Help      CONTENT Help
Swing PHelp     CONTENT Contents and Index
Swing About     CONTENT About Paint
#
Swing Painting SIZE    400,400
```

With the exception of the "Tools Menu" section, all of the database entries are the same. The "Tools" section has "CONTENT" attribute for the "NoSwing" key. This fills in the text for the menuitem. For the "Swing" key, the "CONTENT" is replaced with "IMAGE" and "TOOLTIP". The image places the icon in the button instead of text. "TOOLTIP" adds a message that pops up when that button is in-focus.

The resulting screen when using the second style, a key of "Swing" and the original ui definition looks like:



 © 1997 Harmonia, Inc

Modification:

Now that we have gone through ALL this trouble, what happens if someone says: "But I wanted the tools down the side, not on the top." Exactly how much trouble is it to make that kind of modification.

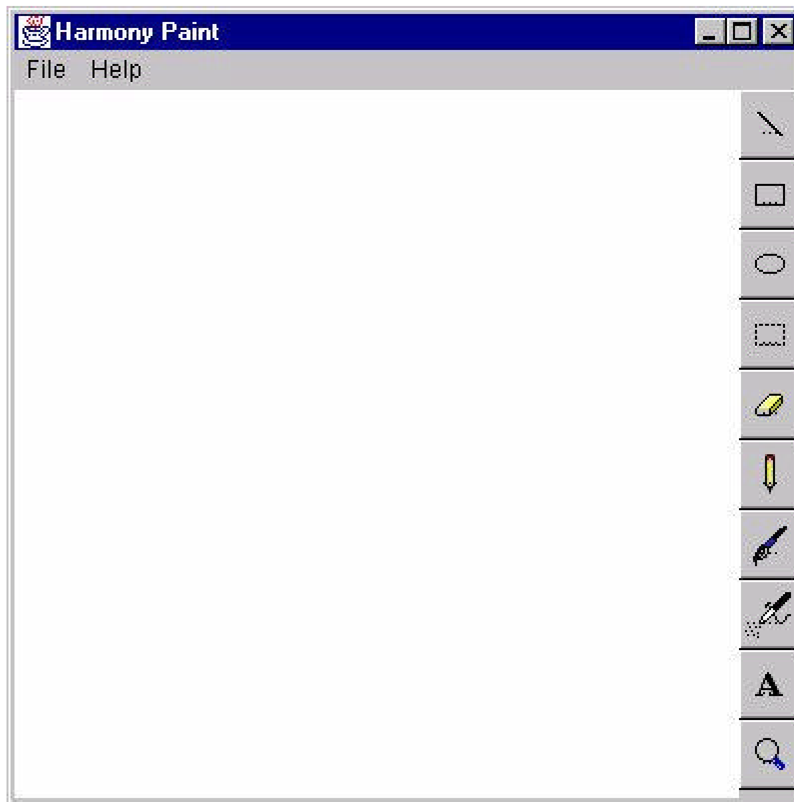
This is actually very simple. We only need to change WHERE the toolbar (which is a panel) is located in the parent, and change the Layout of the toolbar to indicate that we are restricting columns instead of rows.

The following shows the changes: (the crossed-out lines are removed...the bold lines are inserted).

```
/*
  <AUTHOR>Stephen Williams, Harmonia</AUTHOR>
  <DATE>12/3/97</DATE>
  <VERSION>0.6</VERSION>
*/

GROUP.frame {
  +TOOLKIT:   jfc;
  RENDERING: java.awt.Frame;
  LAYOUT:    BorderLayout;
  SIZE:      "400,400";
  FONT-FACE: Serif;
  FONT-SIZE: 10;
  FONT-STYLE:Plain;
  CONTENT:   "Error:No Content";
}

GROUP.menu {
  RENDERING: "java.awt.Menu";
}

ELEM.menuitem {
  RENDERING: "java.awt.MenuItem";
}

GROUP.menuortoolbar {
  RENDERING: "java.awt.Panel";
  SIZE:      "400,20";
  ALIGNMENT: "North";
  ALIGNMENT: "East";
  LAYOUT:     GridLayout;
  +BACKGROUND: "lightgray";
  ROWS:       1;
  COLUMNS:    1;
}

ELEM.itemoricon {
  RENDERING: "com.sun.java.swing.JToggleButton";
  SIZE:      "30,30";
  CONTENT:   " ";
  TEXT-HORIZONTAL: Center;
}

ELEM.PaintArea {
  RENDERING: "java.awt.Panel";
  ALIGNMENT: "Center";
}
```

 © 1997 Harmonia, Inc

Now the interface looks like this:



You may have noticed in the toolbar examples that the Help menu is on the MenuBar immediately after the File menu. As we discussed earlier, any Menu added is put into the ancestor Frame's MenuBar in the order that it occurs.

 © 1997 Harmonia, Inc