*Java*Komp (Comspec/Comlink)

Results of investigation and proposals for improvements

1999-10-01, Peter Norrhall, Anders Quist, Linné (Cell Network) Göteborg
(Delvis översatt och bearbetad av Mats Lundälv, DART 1999-2000)

Introduction

The purpose of this document is to summarise the outcome of our five-week investigation concerning Comspec/ComLink (and Java Accessibility) as part of the *Java*Komp project September - October 1999.

Summary

General Impressions

Our impressions of Comspec/ComLink and its architecture are positive. The work done is of high quality. This may partly be due to the re-design and re-implementation following the disposal of the first *OpenDoc* version.

A key word for the Comspec/ComLink architecture is *modularity*. This means that the interfaces between components are thin, and that only relevant information is published and accessible by other components.

Comspec/ComLink is developed according to standards for visual development environments for *Java*, as *JavaBeans* have been used to build sub-systems, components etc. The implementation is carried out in a consequent manner. The code is relatively easy to follow and get into in relation to the size of the application.

Results

The result of our work is displayed in this document, and in the documents referenced in [1]-[3] below.

In terms of implementation we have upgraded the code to the JDK 1.2 platform and started the work to make it 100% Pure Java. We have carried out some minor de-bugging and performance improvements along the work.

We have developed some new components; SelectScan, ComboboxControl, ListControl and a preliminary version of a MSWordOutput. This work has been documented in the form of a Comspec Component Writer’s Guide [2].

Regarding Accessibility in ComLink itself we have made a sample implementation of "Mouseless Operation", i.e. support for keyboard control, in the Layout Editor part.

Areas of Improvement

Areas in need of improvements for ComLink are:

100% Java	Correction of all references to file and path names, so that ComLink can be used on all *Java* platforms and not just in Windows.
100% Swing	The current user interfaced of ComLink builds on a mix of older AWT and newer Swing based components. A move to purely Swing based components would stabilise and enhance performance, and it would simplify the future maintenance and expansion of the system. The consequent use of Swing based

components would also provide basic support for the Accessibility API and simplify additional Accessibility support in ComLink internally and externally.

Performance Performance could be improved in several ways: By analysis and removal of unnecessary messages and events; By analysis and improvements in graphics handling; By analysis and improvements in the management of data storage and retrieval etc.

Accessibility The whole application should be adapted according to Accessibility guidelines (see [1], Accessibility and Java), including ”Mouseless Operation” etc. (partly implemented in the Layout Editor)

Components To make it possible to build full-featured applications in ComLink, more functionality needs to be added in the form of new components or further development of the existing ones.

External links The usability of the system would generally be greatly expanded by enabling a smooth interaction between ComLink and external applications. This may be accomplished in several ways: Complete ComLink run-time applications could be encapsulated as JavaBeans - to be implanted in other JavaBeans container environments; The ComLink editors could be enabled to accept standard JavaBeans as output components, communicating with some standardised protocol, e.g. InfoBus; Another alternative is to turn ComLink into a server, communicating with other applications via RMI or CORBA.

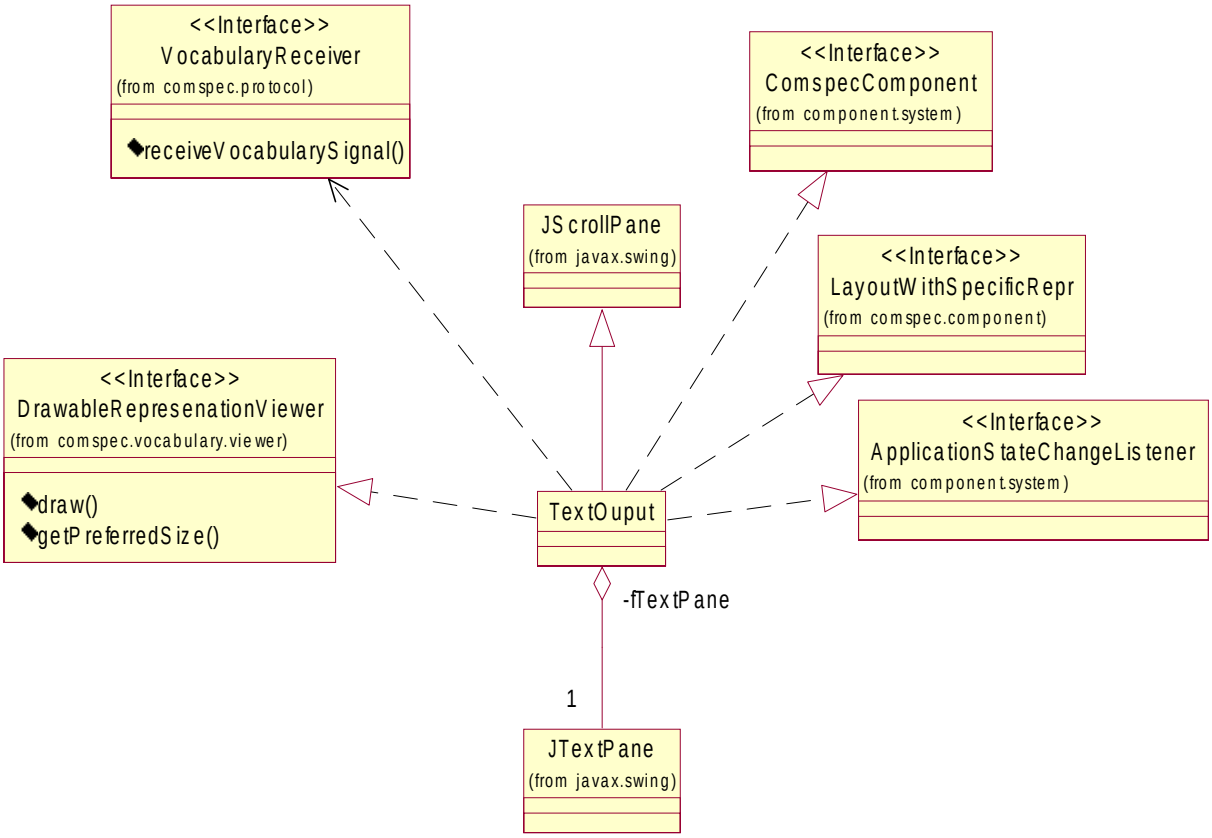
Scripting Making the components scriptable, by means of a scripting language, would eventually enhance the whole environment.

References

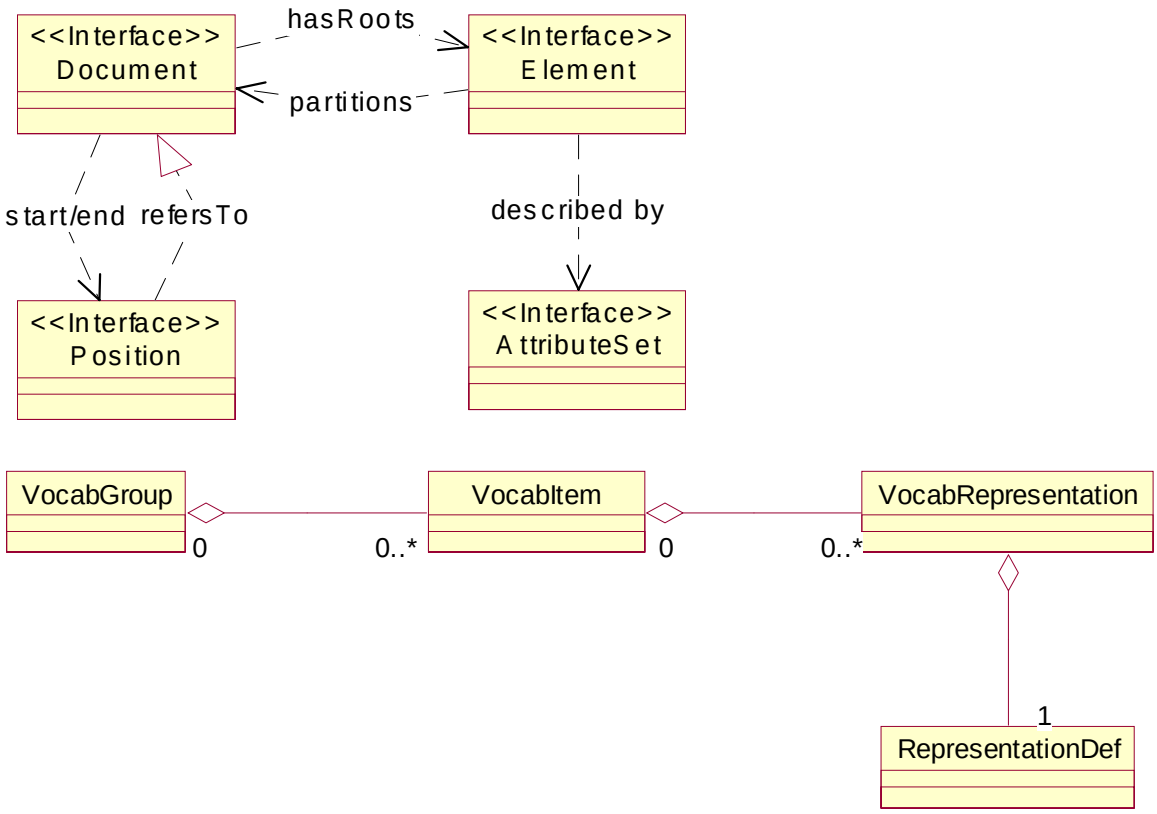
[1] Accessibility and Java, Anders Quist, Linné (Cell Network) Göteborg 1999-10-11
[2] Comspec Component Writer’s Guide, Peter Norrhall, Linné (Cell Network) Göteborg 1999-10-01
[3] Lathund .Doc (Lathund – Deployment av Comspec/ComLink), Anders Quist, Linné Göteborg AB (Cell Network), 1999-10-11

TextOutput

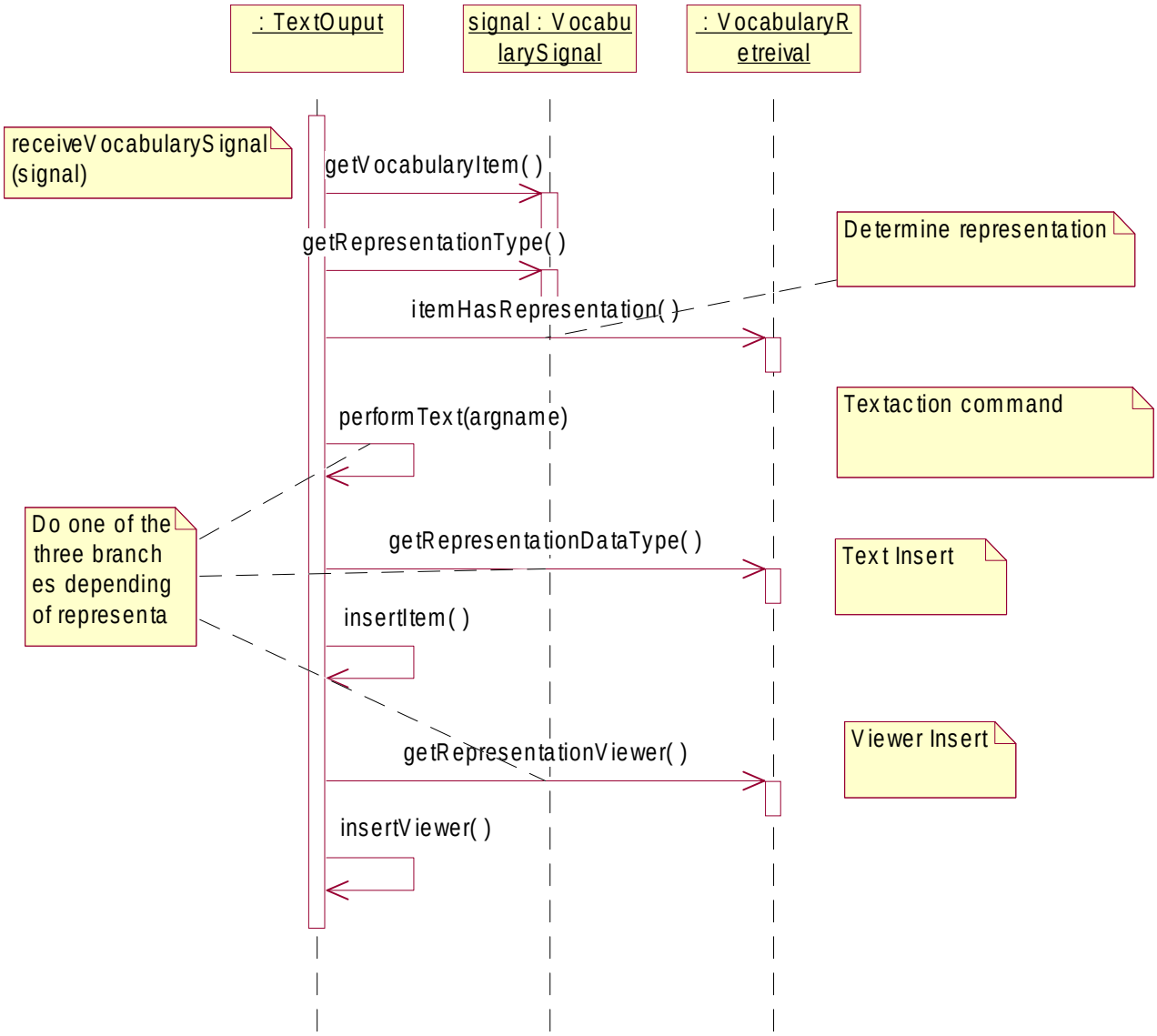
The TextOuput is a descendant of JScrollPane. The text area is a javax.swing.JTextPane. A JtextPane is able to contain (styled)text, components and icons.



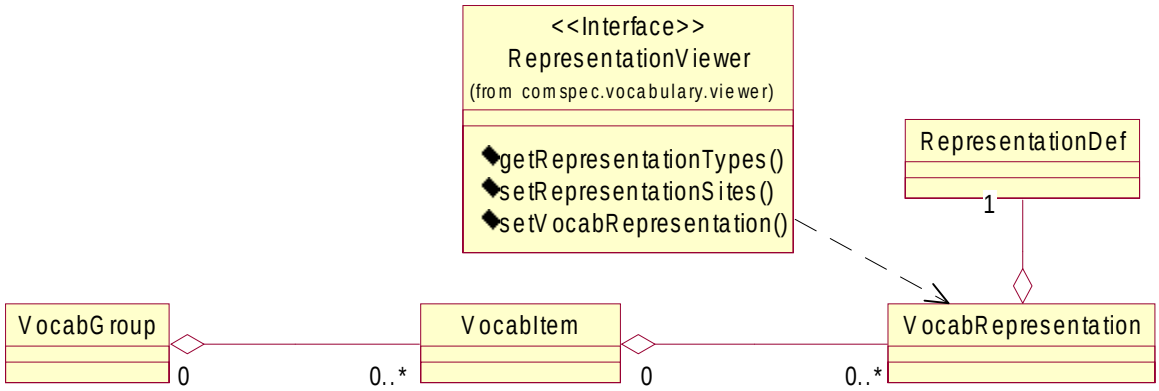
The contents of the JTextPane is a javax.swing.text.Document. A Document contains a tree of javax.swing.text.Element objects representing different kinds of objects, such as characters, components and icons.



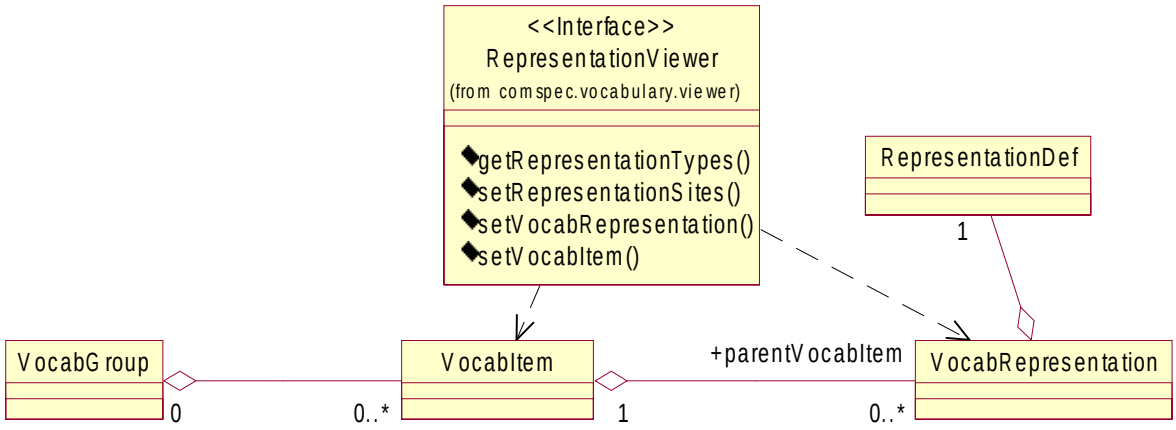
TextOut is a VocabularyReceiver that implements the getVocbularySignal. In getVocbularySignal TextOutput finds what kind of signal it is. It can be a Textaction, Text item or a non-text item with a viewer.



When a VocabItem is represented by a non-text representation (VocabRepresentation) TextOutput tries to find the corresponding representation viewer with getRepresentationViewer.



It is not the VocabItem that is inserted into the Document. It is the RepresentationViewer of the VocabRepresentation. The viewer has no knowledge about its ”parent”, VocabItem. There has to be a design change to make it possible to replace a representation with another.
Either should the VocabRepresentation have a parent reference to its VocabItem. Or, a new method, setVocabItem, should be added to the RepresentationViewer.



Picture – Two different design proposals to extend the class model

In the listing below all inserted Jbutton components will be replaced by a Jlabel in the TextOutput’s document.

```
  public void ReplaceJbuttonWithJLabel(Document doc) {
    Object attr;
    int start, end;
    // Create an iterator to visit all elements for the document
    ElementIterator it = new ElementIterator(doc);
    Element el = it.first();
    while (el != null) {
      // StyleConstants.CharacterConstants.ComponentAttribute = "component"
      attr = getAttribute(el,
                  StyleConstants.CharacterConstants.ComponentAttribute);
      // Is the element a JButton
      if (attr instanceof JButton) {
        // Remove the button. A component is represented as an empty
        // character in the document
        start = el.getStartOffset();
        end = el.getEndOffset();
        try {
          jTextPane1.getDocument().remove(start, end-start);
        } catch (BadLocationException exc) {
          ;
        }
        // Insert a new component
        // Place the caret at the same position as the JButton had
        jTextPane1.setCaretPosition(start);
        jTextPane1.insertComponent(new JLabel("Hej"));
      }
      el = it.next();
    }
}
```

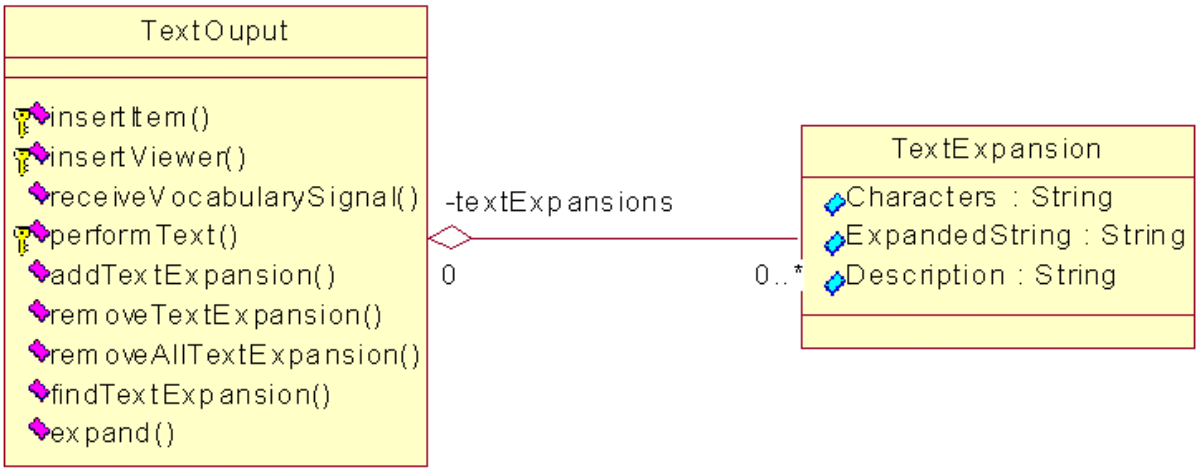
```
private Object getAttribute(Element e, Object attrName) {
    Object value = e.getAttributes().getAttribute(attrName);
    if (value == null) {

            // Neither the delegate or its resolvers had a match,
            // so we'll try to resolve through the parent
            // element.
            AttributeSet a = (e.getParentElement() != null) ?
                              e.getParentElement().getAttributes() : null;
            if (a != null) {
                value = a.getAttribute(attrName);
            }
    }
    return value;
  }
```

Text expansion

One of the suggestions of this project was to make a text expansion component. This component is a layout component that receives text items. For each character the component receives it compares those characters in a lookup table and sees whether they can be expanded to a word.

To implement this, the TextOutput component could be used or a Swing-based text component that doesn't necessary have to be editable as long as it implements the VocabularyReceiver interface

First of all the integrator has be able to add a list of text expansions. Each text expansion contains the abbreviated string of characters, the expanded word or sentence and perhaps a short description. The text component contains a list TextExpansion objects.



In the receiveVocabularySignal, we look for vocabulary items representing text. If so, check if the latest entered characters is found in the textExpansions list and replace those character with the ExandedString.

But how do the end-user add TextExpansion to the list? Either could the items be read from a file in a predefined format. We could also add a property editor that could be used to add and remove items or we could use both.

Letting the user have its own dictionary is more flexible and extendable. In that case a file name property should be added to the component and a customizer to set the file name, using an "open file dialog". A dialog to

If you prefer the second design proposition you have to make a customizer for the component. The customizer must be able to edit all properties as the property editor plus the textExpansions list.

Example of implementation for TextOutput:

```
protected Hashtable fTextExpansion;
protected int fLastOffsetSpace;


public TextOutput() {
…
  fTextExpansion = new Hashtable(10);
}

public void init() {
…
  // Test implementation
  // Replace this with a read/write method
  fTextExpansion.put(new String("mvh"), new TextExpansion("mvh",
                       "Med Vänlig Hälsning", ""));
}

public void receiveVocabularySignal(VocabularySignal signal) {
…
  if (voc.getRepresentationDataType(item, rep). equals(MIME_TEXT_PLAIN) ) {

    // Get the text
    String text = (String)voc.getRepresentationData(item, rep);
    Document doc = fTextPane.getDocument();
    String prevString = doc.getText(fLastOffsetSpace,
                            doc.getLength() - fLastOffsetSpace);
    String newString = new String(prevString + text);
    TextExpansion te = (TextExpansion)fTextExpansion.get(newString);
    if (te != null) {
      doc.remove(fLastOffsetSpace, doc.getLength() - fLastOffsetSpace);
      insertItem(item, rep, te.getWordExpansion());
    }
    else
      insertItem(item, rep, text);
…
}

protected void insertItem(String item, String rep, String text) {
…
  case CONTINIOUS_ITEMS :
    fTextPane.replaceSelection(text);
    if ((text.equals(" ")) || (text.equals("  ")))
      fLastOffsetSpace = fTextPane.getDocument().getLength();
    break;
…
}
```

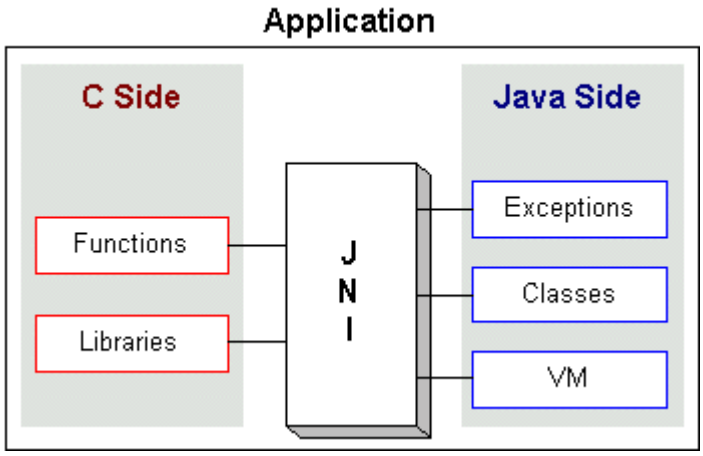
Connections to External Applications

I have made some investigations to connect Comlink to external applications. There are several techniques to use for this purpose. Below are those that are recommended to use that are platform independant.

- Remote Method Invocation (RMI) – Talk to other Java applications (local/remote)
- Common Object Request Broker Architecture (CORBA) – Talk to other applications through the ORB (local/remote)
- Java Native Interface (JNI) – To allow the Java code to operate with applications and libraries written in other languages.
- InfoBus – Talk to other Java applications in a producer/consumer way (local)

JNI

JNI serves as the glue between Java and native applications. The following diagram shows how the JNI ties the C side of an application to the Java side.

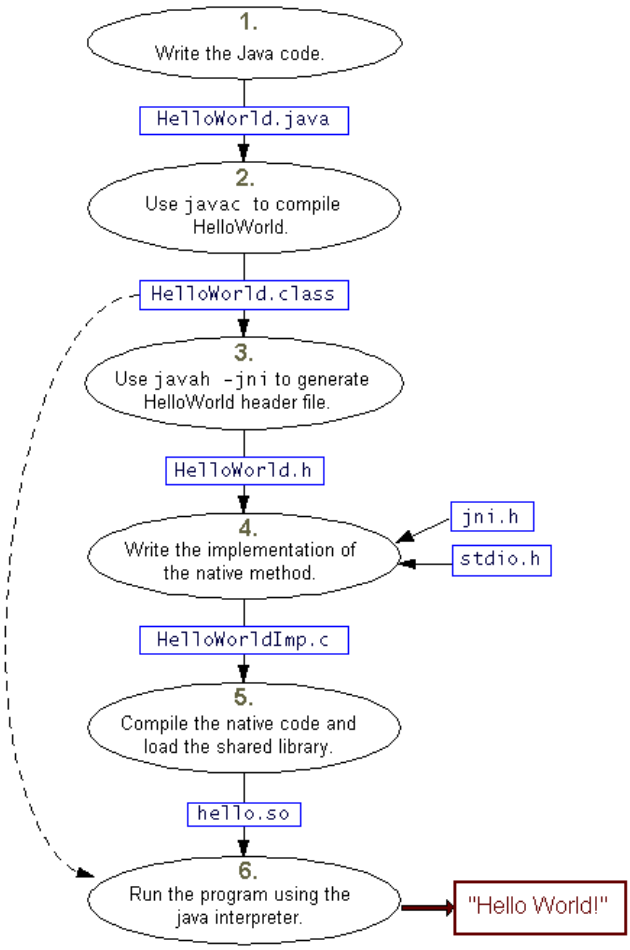


JNI is for example used in Comlink for the Gameport Switch Input, since it is tied to the Windows-platform. But, if someone makes a similar product for Linux, all that has to be done is a library which implements the native functions. Note that using JNI allow us to communicate in both directions

```
//*************************************************************
  //  Interaction with the native code
  //*************************************************************

  /**
   * Start the native code. Indicates the joystick should be activated
   * and that we want to receive joyButtonChanged messages.
   */
  private native void start();

  /**
   * Stop the native code. Indicates that we no longer want to receive
   * joyButtonChanged messages.
   */
  private native void stop();

  /**
   * joyButtonChanged is the call back function for native code.
   * @param joyNr the number of the joystick (1 or 2)
   * @param btnNr the number of the button changed (1..4)
   * @param down the down state of the button
   */
```

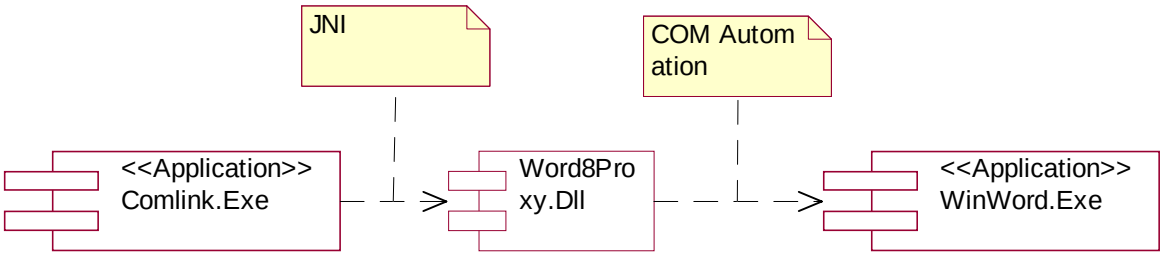
```
private void joyButtonChanged(int joyNr, int btnNr, boolean down) {
  System.out.println("joyButtonChanged called"+
    ", joyNr=" + joyNr +
    ", btnNr=" + btnNr +
    ", down="  + down);
  if (joyNr == fJoyStickNumber) {
    // find the switch number according to the button number
    int swnr;
    if (btnNr == fSwitch1Button)
      swnr = 0;
    else if (btnNr == fSwitch2Button)
      swnr = 1;
    else
      return;

    fKeyDownStates[swnr] = down;
    if (down)
      sendKeyDown(swnr);
    else
      sendKeyUp(swnr);
  }
}
```

Below is a figure of the steps to go in the JNI “process”.



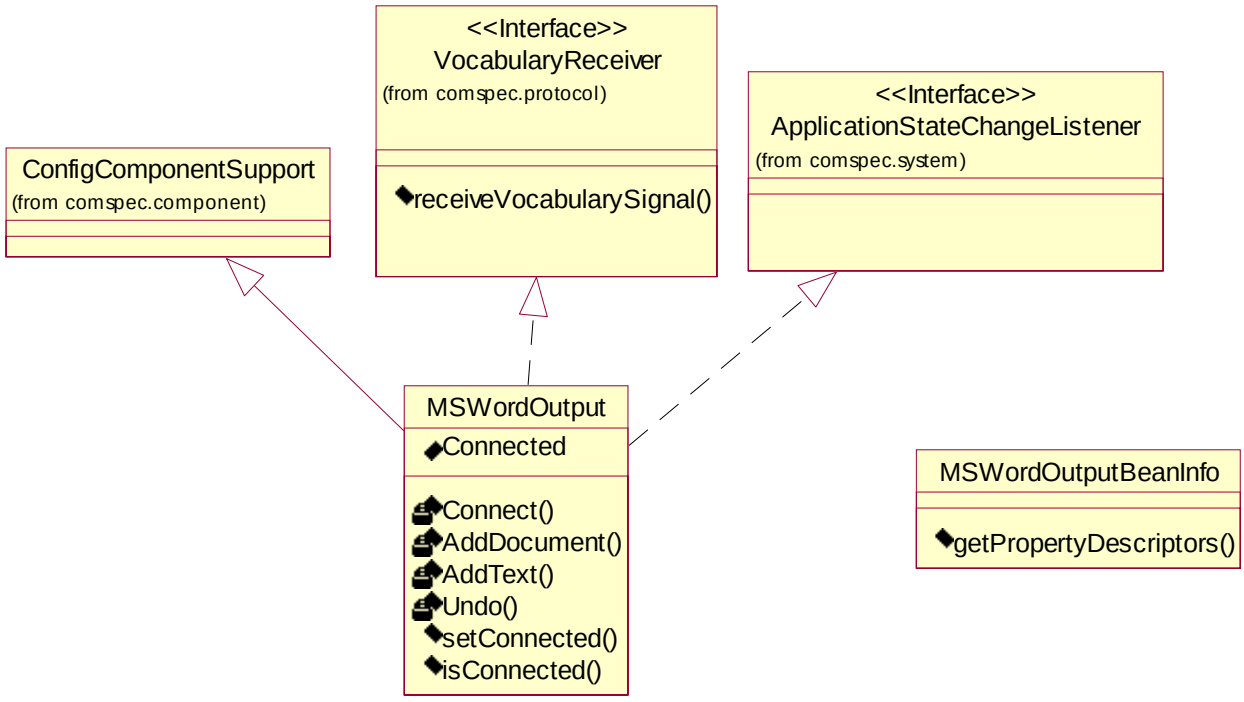
Microsoft Word 8 (Word97)

We decided to use JNI to try to talk to Microsoft Word using JNI. The library Word8Proxy.dll is a Dynamic Link Library (DLL) and it was written in C++, using C++Builder 3. The DLL talked to Word using *Automation*.



In Comspec we wrote a simple test component, MSWordOutput, which is a ConfigComponent. We did not fully implement the component. For some reason ComLink crashed when the MSWordOutput was used.
To verify the technique, we created a simple Java application MyApp which uses the same native methods as MSWordOutput. All sources are can be found on the CD.

Below is the architecture described of MSWordOutput.



The native methods of MSWordOutput are declared as below

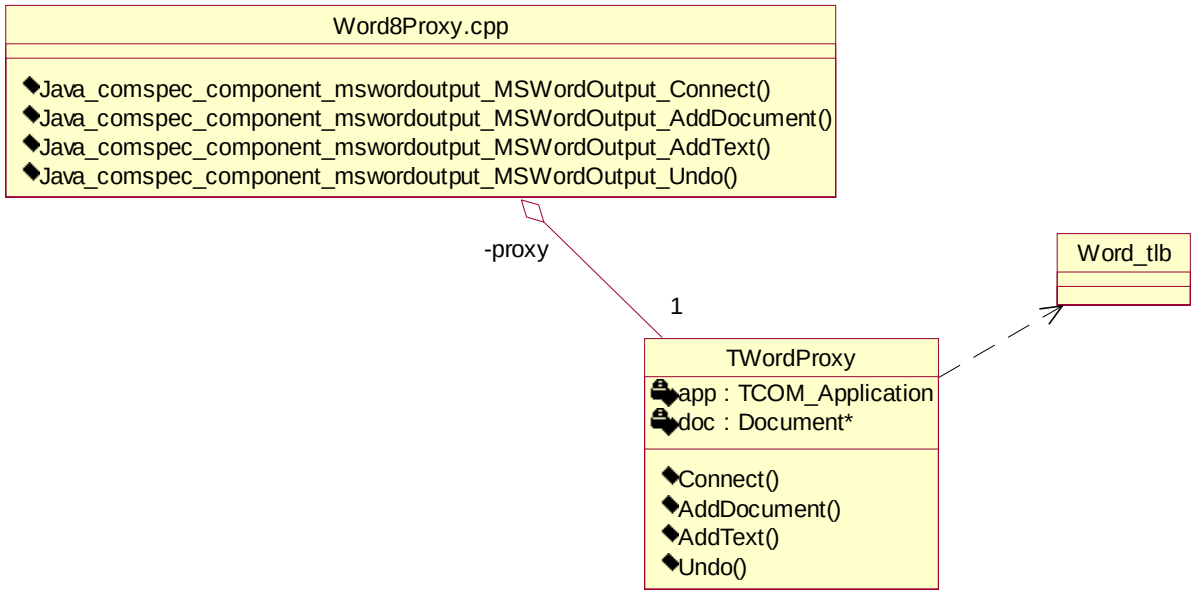
```
private native void Connect(boolean connect);
private native void AddDocument();
private native void AddText(String s);
private native void Undo(boolean undo);
```

To implement these methods in C++ we used the utility JavaH. JavaH is included in the JDK SDK and it generates a C/C++ header file out of a class with native methods. In our case it generates a comspec_component_mswordoutput_MSWordOutput.h file.

Below is an extract of the header file showing how the methods are generated

```
JNIEXPORT void JNICALL Java_comspec_component_mswordoutput_MSWordOutput_Connect
  (JNIEnv *, jobject, jboolean);

JNIEXPORT void JNICALL Java_comspec_component_mswordoutput_MSWordOutput_AddDocument
  (JNIEnv *, jobject);

JNIEXPORT void JNICALL Java_comspec_component_mswordoutput_MSWordOutput_AddText
  (JNIEnv *, jobject, jstring);

JNIEXPORT void JNICALL Java_comspec_component_mswordoutput_MSWordOutput_Undo
  (JNIEnv *, jobject, jboolean);
```

To communicate with Word we use COM Automation. In C++Builder you import the type library of the Automation Server. In this case it is \Microsoft Office\Office\MsWord8.olb. Out of this file a header file is generated to reflect the type library, Word_tlb.h (and Word_tlb.cpp).



InfoBus

The InfoBus is a public specification of dynamic data-sharing technology that enables developers to equip their JavaBean components to communicate with other JavaBean components. InfoBus was jointly designed by Lotus Development Corporation and Sun Microsystems' JavaSoft division.

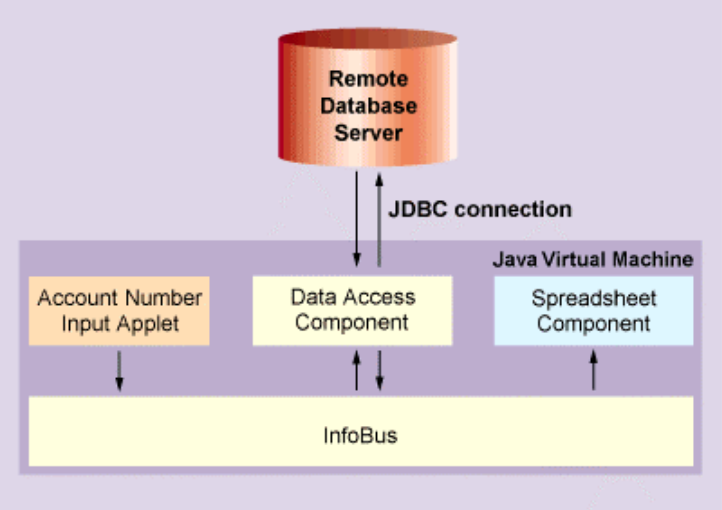


Figure 2. InfoBus Handles Data Flows. From a logical data flow perspective, our composite application consists of a series of three applets. The applets are connected according to HTML statements that specify, for each applet, the name(s) of the data items handled.

As you can see in the picture above communicating with other applications/applets only works within the same JVM.