Using Java™ APIs with Native Ada Compilers

Shayne Flint
Ainslie Software Pty Limited
Suite 802, 2 Marcus Clarke Street
Canberra City ACT Australia
+61-2-6247-4801
shayne@ainslie-software.com

1. ABSTRACT

Ada is an ISO standard Object Oriented programming language specifically designed to support the cost effective development of robust, maintainable software. Because of this, Ada is widely used in the development of critical systems such as commercial aircraft. However, despite its advantages and general purpose nature, Ada is not often used for the development of main stream applications. This is partly because of Ada's poor integration with contemporary technologies such as Graphical User Interfaces. Described within this paper is a technique which uses the Java Native Interface to provide Ada programmers with immediate access to any software that has a Java API, thus substantially improving the suitability of Ada for the development of a wide range of applications.

1.1 Keywords

Ada, Java, Bindings, Java Native Interface

2. INTRODUCTION

For more than a decade Ada has proved to be a superior language for supporting the development and maintenance of large systems [7][8]. Its use leads to better productivity, reduced costs, more reuse, much lower error rates (especially early on) and significantly lower maintenance costs. While Ada may be an appropriate language for the development of large long lived systems, its wider applicability is somewhat limited by a lack of standardised Application Programmer Interfaces (APIs) to software such as Graphical User Interfaces (GUIs). Because of this, Ada has been largely restricted to the development of software for critical systems such as aircraft, railways and air traffic control. Ada has seen very little use in main stream computing.

Java, on the other hand, has become popular for the development of portable main stream applications. Its flexibility is due in part to the availability of a wide range of portable APIs. If these Java APIs are made available to Ada programmers, then Ada can become a more viable language for development of the same kinds of applications while maintaining the characteristics required for the cost effective development and maintenance of large systems.

This paper contains a description of a system called *AdaJNI* which has been developed to support the use of Java APIs with native Ada compilers. A number of issues associated with the mapping of Java classes to Ada, along with approaches for dealing with them are also discussed.

3. BACKGROUND

The integration of Ada with Java APIs can be achieved in a number of ways. One approach is that adopted within the AppletMagic™ compiler [1] developed by Intermetrics Inc. This is a special Ada compiler which can generate code to run on the Java Virtual Machine (JavaVM) and is supplied with Ada bindings to the Java API's. This approach provides a good solution for the development of portable software such as applets for the internet.

There are, however, cases in which the use of a native Ada compiler is required or desirable. In such cases a different approach to that of AppletMagic is needed to integrate Ada with Java APIs. The Java Native Interface (JNI), which supports the integration of Java and C, offers an approach to using Java APIs with native Ada compilers. The JNI is a C library supplied with the Sun Java

Development Kit (JDK) [2]. It allows C programmers to make requests on the JavaVM to create Java objects, call methods and to read/write fields. By creating an Ada binding to the JNI, Ada applications could be written to create and manipulate Java objects in the same way.

While JNI Ada bindings are sufficient to provide Ada with access to Java classes, they do not provide an 'Ada like' interface to each Java class. To call a Java method using JNI bindings, the programmer would ordinarily make a series of calls to the JNI to obtain a class ID and method ID, to marshal the required parameters for the call and to request the JavaVM to call the method. This does not follow the normal Ada procedure calling mechanism and would be impractical for normal application development.
Ideally, an Ada package corresponding to each Java class in a given Java API should be available to the Ada programmer. The specifications of such packages would contain sub-programs corresponding to the methods of each class thus providing a simple 'Ada style' interface. The Ada package bodies would manage the low level interface to the JNI using the JNI bindings.

4. ADAJNI

AdaJNI (**Ada** to **J**ava **N**ative **I**nterface) is an implementation of the JNI based integration approach described above. The system comprises four major components which interact with an Ada application and the Java environment as depicted in Figure 1. The *Java Native Interface Ada Bindings* are the basis upon which AdaJNI is constructed. They provide a low level interface to the Java Virtual Machine using the JNI API. The *AdaJNI Runtime Packages* manage the mapping of Java language features such as arrays, strings and exceptions to Ada, as well as the marshalling of parameters passed to Java methods.

The *Java Class Ada Bindings* provide an interface between the Ada application and Java classes. These bindings comprise an Ada package for each Java class required by the application. The specifications of such packages provide an "Ada Style" interface to a specific class including all of the class' constructors, methods, and fields. The binding package bodies contain code which maps the Ada interface to the underlying JavaVM via the *JNI Ada Bindings* and *Runtime Packages*.

The *Event Management* component integrates the Java 1.1 Event Model [3] with the AdaJNI environment and thus Ada. The AdaJNI event management approach is more fully described in section 6.13 of this paper.

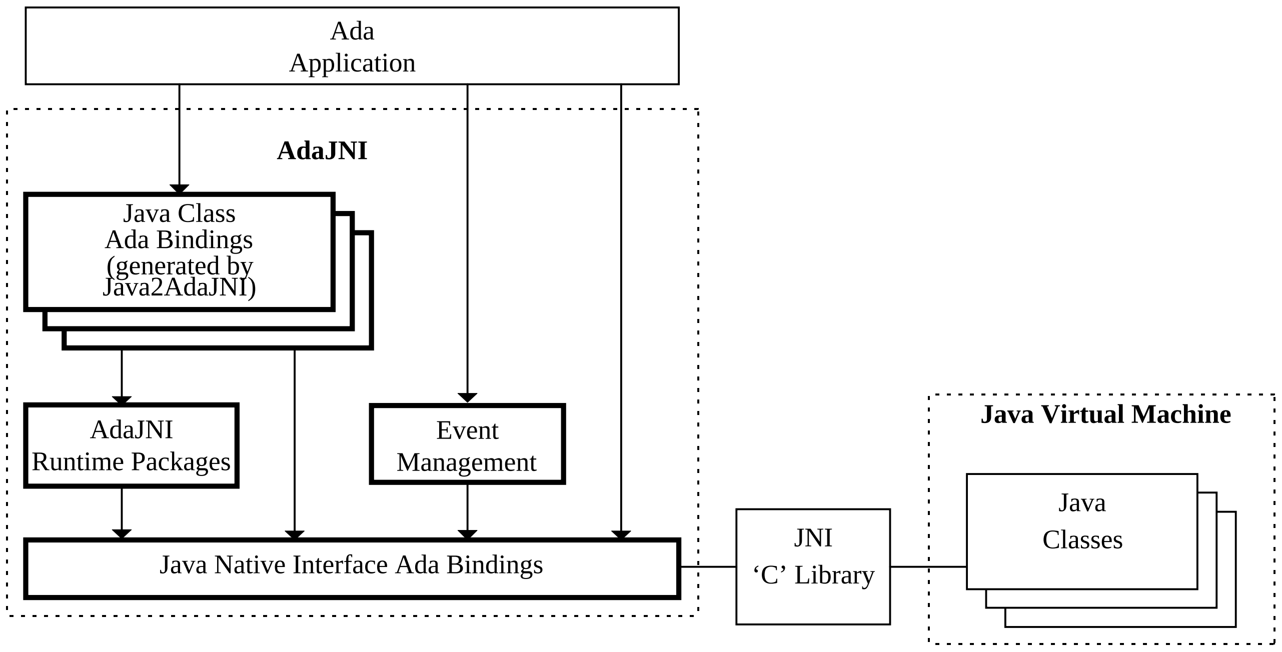

Figure 1. The AdaJNI Architecture

5. BINDING GENERATOR TOOL

In order to ensure that Ada developers are able to make use of any Java API, a tool has been developed to automatically generate the *Java Class Ada Bindings* described above. As depicted in Figure 2, *Java2AdaJNI* processes Java classes and (optionally) Java source code to generate Ada source code in the form of Ada packages which provide access to the services of a specified Java class. When a class is processed, *Java2AdaJNI* will extract information about the class using the standard *java.lang.reflect* API. This information is used to generate an Ada package specification containing data types and subprograms which correspond to the constructors, methods and fields of the Java class. *Java2AdaJNI* also generates a corresponding package body which contains the Ada code required to interface the Java class.

While functional bindings can be generated solely from information provided via the *java.lang.reflect* API, such bindings will not have correct parameter names for constructors and methods because the API does not provide parameter name information. In such cases, parameters are named P1, P2 etc.

If, on the other hand, a Java source file is processed, parameter names and *JavaDoc* comments are extracted from the source file. This information is used in conjunction with information extracted via the *java.lang.reflect* API to generate a complete binding which includes the correct parameter names for constructors and methods along with documentation comments.

Java2AdaJNI has been used to generate complete bindings to all of the normal Java API's (Awt, lang, net etc.) [4] and the Java Foundation Classes [5] without any manual re-working. The tool can also be used by the Ada programmer to generate bindings to other Java classes. In short, *Java2AdaJNI* provides the Ada programmer with immediate access to any software which has a Java API.

6. MAPPING JAVA CLASSES TO ADA

In general terms, the AdaJNI approach maps each Java class to a separate Ada package. Each package contains a tagged record *Object* type which reflects the corresponding Java class inheritance hierarchy. A *Reference* type is also declared as an access type to the *Object* type. Values of these *Reference* types are initialised by constructors and refer to Java objects managed by the JavaVM. The remainder of each package comprises Ada sub-programs corresponding to each constructor and method of the class along with sub-programs to manipulate any fields in the Java class.

For a tool such as *Java2AdaJNI* to be effective it must automatically and completely generate these Ada packages so that the Ada programmer can make use of any Java class without delay. In order to achieve this aim a number of issues have been addressed during the development of AdaJNI and are discussed in the following paragraphs.

6.1 Circularities in Java Classes.

Many Java classes contain circularities in dependencies. For example the *java.lang.String* class depends on *java.lang.StringBuffer* which in turn depends on *java.lang.String* causing a circularity. This is a problem when generating Ada because the language does not allow circularities between Ada package specifications. A strategy which automatically handles circularities in Java classes is therefore adopted by *AdaJNI*.

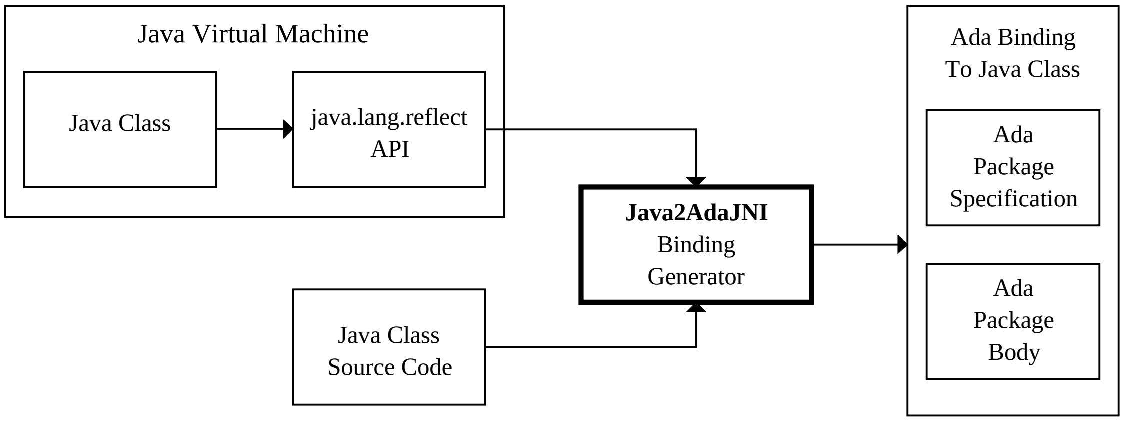

Figure 2. The Java2AdaJNI Binding Generator

The strategy involves the use of two package specifications for each Java class. The first (primary) package contains data types for the subject class together with subprograms for each Java class constructor, method and field. A portion of the primary specification for the *java.awt.Button* class is shown below.

```
with Ada_JNI.Bindings.Java_Awt.Component;

package Ada_JNI.Bindings.Java_Awt.Button is

  type Object is new
    Ada_JNI.Bindings.Java_Awt.Component.Object
    with null record;

  type Reference is
    access all Object'class;

  ... constructors, methods etc.
```

The second (Secondary) package contains similar data types along with array types and associated array handling packages. A portion of the secondary specification for the *java.awt.Button* class is shown below. The major difference between the two packages is in the parent package name. Primary packages are children of the *Ada_JNI.Bindings* package, while secondary packages are children of the *Ada_JNI.Binding_Types* package.

```
with Ada_JNI.Binding_Types.Java_Awt.Component;

package Ada_JNI.Binding_Types.Java_Awt.Button is

  type Object is new
     Ada_JNI.Binding_Types
    .Java_Awt.Component.Object
        with null record;

  type Reference is
    access all Object'class;

  type Reference_Array is
    array ...

  type Reference_Array_2D is
    array ...

  ... array management packages
```

Note that the Secondary specification data types mirror those in the primary specification except that *Object* in the secondary specification descends from *Object* defined in the secondary specification for the class *java.awt.Component*. In other words, the type hierarchy in secondary packages is the same as that in the primary packages except that they have a different root.

The primary specification contains the methods and fields of the Java class. It is the parameter data types of these methods that cause the circularities described earlier. By declaring parameters of types declared in secondary packages rather than types defined in primary packages, all circularities can be removed. For example, the following *String* constructor refers to the secondary specification for the *String_Buffer* data type rather than the primary specification. Because the secondary specification contains only the data types and no methods etc., there are no dependencies. There is therefore no chance of circularities.

```
function New_String
  ( Buffer : in  Ada_JNI.Binding_Types
                .Java_Lang.String_Buffer.Reference
  )
    return Reference;
```

Unfortunately, this approach presents a new problem. While all application objects will be of types declared in primary specifications (constructors return data types from primary specifications), all constructor and method parameters are of types declared in secondary specifications. This means that Objects created with constructors cannot be passed as parameters to methods. This problem has been solved in AdaJNI by providing unary "+" operators in each primary specification which convert between primary and secondary data types. Examples of these operators from the *Java_Lang.String* package are shown below.

```
function "+"
  ( Ref : in Reference
  )
    return  Ada_JNI.Binding_Types
           .Java_Lang.String.Reference;

function "+"
  ( Ref : in  Ada_JNI.Binding_Types
             .Java_Lang.String.Reference
  )
    return Reference;
```

By using these operators, as shown below, the programmer can effectively ignore the presence of the secondary package. In this example, *My_String* is of a primary type, while the *Label* parameter of the *Set_Label* method is a secondary type. The "+" operator is therefore used to convert *My_String.*

```
declare
  My_String :  Ada_JNI.Bindings
              .Java_Lang.String.Reference;
begin
  My_String := New_String("Hello World");
  Ada_JNI.Bindings.Java_Awt.Button.Set_Label
    ( Ref   => My_Button,
      Label => +My_String
    );
end;
```

This simple imposition requiring use of the "+" operator is the only impact of the strategy to remove all circularities. It is considered to be superior to other techniques which rely on the manual rearrangement of package specifications to resolve circularities.

6.2 Mapping Java Names to Ada Names.

Java is a case sensitive language which distinguishes between names such as 'Test' and 'test'. Ada, on the other hand, is not case sensitive so that 'Test' and 'test' are considered the same name. Within Java packages and classes it is common to find entities which have the same profile (i.e. methods with the same parameter types or fields with the same data type) and names which only differ in case. Such entities would be considered the same in Ada and therefore illegal.

AdaJNI uses various strategies to resolve such naming conflicts. In general, simple name conflicts such as Test() and TEST() are resolved by appending integers so that TEST() become Test_1(). Only those name conflicts which are illegal in Ada are resolved and, where possible, the names of methods are changed first, followed by fields, then constructors and finally inner classes. This prioritisation minimises the impact of awkward names with appended numbers.

Within AdaJNI a different approach is applied to the naming of Java packages. There are many cases in the Java API's where Java's case sensitivity is used to distinguish the names of packages and classes. For example, there is a package called *java.awt.event* and a class called *java.awt.Event*. These names cannot be distinguished in Ada and adding a number to one of the names would result in awkward class or package names. In order to eliminate this problem, Java hierarchical package names are mapped to a single Ada package name (e.g. *Java.awt* maps to *Java_Awt* and *java.awt.event* maps to *Java_Awt_Event*). Essentially, dots in the Java package name are replaced by underscores in the Ada package name thus avoiding a large class of name conflicts without having to resort to awkward numbering schemes.

Note that the class *java.awt.Event* maps to *Java_Awt.Event*, and that if there were two packages named *java.awt.event* and *java.awt.Event*, their names would map to *Java_Awt_Event* and *Java_Awt_Event_1*.

Finally, *Java2AdaJNI* converts Java names to Ada style names using a simple well defined algorithm. For example '*clearRect*' becomes '*Clear_Rect*' and '*setXORMode*' becomes '*Set_XOR_Mode*'.

6.3 Java Constructors.

Java constructors are mapped to Ada functions which take the same parameters as the corresponding Java constructor and return a *Reference* to a new object created in the Java Virtual Machine. This reference is used to refer to the object in subsequent operations. Constructor functions are used as follows:

```
My_Button :  Ada_JNI.Bindings
            .Java_Awt.Button.Reference
  := Ada_JNI.Bindings.Java_Awt.Button.New_Button
       ( Label => +New_String("Press Me")
       );
```

6.4 Abstract Classes and Methods.

AdaJNI does not map abstract Java classes and methods to abstract Ada types and sub-programs. Instead, AdaJNI maps abstract classes to Ada bindings in a similar way as it does for normal classes. That is, abstract classes are mapped to Ada packages and tagged types, and methods defined in abstract classes, including abstract methods, are mapped to normal Ada subprograms. The only difference between bindings to abstract classes and those to normal classes is that the former do not include any constructors. This prevents the Ada programmer from creating instances of abstract classes.

This approach is used because Java allows abstract classes to be used as data types for objects returned by methods. The *java.awt.Component.getGraphics()* method, for example, returns an object of the *java.awt.Graphics* abstract class. When an Ada programmer calls the Ada binding to this method, an object of some unknown class extension of *java.awt.Graphics* is returned. The AdaJNI approach allows the Ada programmer to manipulate such objects using the Ada bindings generated by *Java2AdaJNI* for the *java.awt.Graphics* abstract class. Calls to Ada sub-programs which map to abstract methods in *java.awt.Graphics* are passed to the Java Virtual Machine for dispatching to the appropriate concrete methods.

6.5 Java Interfaces.

Java interfaces comprise a set of abstract method specifications which are implemented by one or more classes which claim to implement the interface. *Java2AdaJNI* processes an interface by generating a normal Ada package specification without any constructors or subprograms corresponding to abstract methods (ie. all methods declared in an interface).

In addition to processing the interfaces themselves, *Java2AdaJNI* is required to perform additional processing on classes which implement interfaces. While a given class can only implement (extend) a single abstract class, it may implement any number of interfaces. In addition, Java interfaces can be used as types for method and constructor parameters. An object of any class which implements an interface can be used as a value of that interface type. For example the *java.awt.GridLayout* class implements the *java.awt.LayoutManager* interface. A

GridLayout object can therefore be used anywhere a *LayoutManager* object is required. Because a class can implement any number of interfaces, such classes can be treated as being of more than one type.

These are important characteristics of Java interfaces which must be supported in the Ada bindings generated by *Java2AdaJNI*. When *Java2AdaJNI* processes a class that implements one or more interfaces, it will generate the bindings in the normal way. That is, it will generate subprograms for each method in the class including those which implement abstract methods declared in the implemented interfaces. *Java2AdaJNI* will also generate a special function for each implemented interface which converts objects of the current class to each interface type. These functions allow an object of the class to be used wherever one of the implemented interfaces is required.

For example the *Ada_JNI.Bindings.Java_Awt.Grid_Layout* package contains the following conversion function because the *GridLayout* class implements the *LayoutManager* interface.

```
function To_Layout_Manager
  ( Ref : access Object
  )
    return  Ada_JNI.Bindings
           .Java_Awt.Layout_Manager.Reference;
```

This function can be used as shown below. Because the *Set_Layout* procedure requires a *Layout_Manager* (an interface) parameter, the "*To_layout_Manager*" function is used to convert "*My_Grid_layout*" (an instance of the *GridLayout* class which implements the *LayoutManager* interface) to the required type. Note the use of the "+" operator to convert the parameter to the secondary data type required by Set_Layout.

```
Ada_JNI.Bindings.Java_Awt.Container.Set_Layout
  ( Ref => My_Container,
    Mgr => +To_Layout_Manager(My_Grid_Layout)
  );
```

6.6 Java Arrays.

Java arrays are either arrays of Java object references or arrays of primitive types (int, short, char etc.). Java arrays can have multiple dimensions and all indices are natural ranges starting at 0 (ie. 0..n).

Ada array types for primitive Java types are declared within primitive data type packages such as *Ada_JNI.Java.Boolean* and *Ada_JNI.Java.Float.* Child packages provide subprograms to create Java arrays and to convert between Java and Ada arrays. AdaJNI supports one, two and three dimension arrays.

Object arrays are handled by array management packages generated by *Java2AdaJNI* for each Java class. These packages provide subprograms to create Java arrays and to convert between Ada and Java arrays. The packages support one, two, and three dimension arrays.

From the Ada programmers' point of view all arrays are Ada arrays. All parameter and function return array types are defined in bindings as Ada arrays. Conversion to and from Java arrays goes on within the bodies of the bindings and need not concern the Ada programmer.

6.7 String Handling.

When *Java2AdaJNI* processes the *java.lang.String* class, it generates a number of convenience subprograms in addition to its normal processing. Additional constructors are provided to support the creation of Java strings based on the values of Ada strings. Functions are also provided to convert between Java and Ada strings.

6.8 Java Exceptions.

When an exception occurs in Java, an Exception object, derived from the *java.lang.Throwable* class, is created and propagated up the call chain until an appropriate exception handler is found. When a call is made to a Java method via the JNI and a Java exception is thrown, control is returned to the native code without any indication that an exception was thrown. An exception object is created, but it is not propagated to the native calling function. After each call to the JNI, a native function should call a special JNI function to check if an exception has been thrown on the Java side. If an exception has been thrown the function will return a reference to the exception object.

The *Java2AdaJNI* binding generator handles Java exceptions in a simple and effective manner. When a class, ultimately derived from *java.lang.Throwable,* is processed by *Java2AdaJNI*, an Ada exception declaration is added to the corresponding Ada package specification. When such a package is elaborated, the exception is registered with the AdaJNI runtime along with a reference to the corresponding Java exception class.

Java2AdaJNI also inserts a call to an exception checking procedure (part of the AdaJNI runtime) after every call to a Java method which may throw an exception. The procedure checks for a Java exception and if one has been thrown by the Java method just called, it will raise the corresponding Ada exception as determined by looking up all exceptions registered with the AdaJNI runtime.

From the Ada programmers' point of view, subprograms in the bindings raise exceptions like any other Ada subprogram.

6.9 Inner Classes.

Java Inner classes come in various flavours. The only ones that affect the bindings are ‘Nested top-level classes and interfaces’ and ‘Member Classes’. Normally these inner classes would be mapped to nested Ada packages declared within the package corresponding to the outer class. Because current releases of the Java Development Kit (JDK 1.1.x and 1.2) do not implement the reflection APIs for inner classes, *Java2AdaJNI* must rely on the fact that Java compilers generate a separate class file for every class and inner class found in a given source file. Each of these class files is processed separately by *Java2AdaJNI* in the normal way. If a class is found to be an inner class, the generated Ada package is declared as a child of the Ada package corresponding to the inner class’s enclosing class.

6.10 Method and Constructor Parameter Names.

The *java.lang.reflect* API, upon which *Java2AdaJNI* relies for class information, does not provide information about constructor and method parameter names. In order to generate bindings with correct parameter names, *Java2AdaJNI* is able to process Java source files. When processing a source file, *Java2AdaJNI* will use the *java.awt.reflect* API to obtain information about every class declared in the source code. This information, supplemented with parameter names extracted from the source code, is then used to generate Ada bindings.

In order to support the use of parameter names obtained from other places such as class documentation, *Java2AdaJNI* is able to generate a text file (called a Java Class Specification - JCS) which closely resembles the structure of a classes Java source code. These JCS files are accepted by *Java2AdaJNI* as standard Java source code. So, if the original source code is not available for a given class and correct parameter names are required, *Java2AdaJNI* can be directed to process the class file and to generate a corresponding JCS file. The JCS file, which contains parameter names such as P1, P2 etc., can then be edited by hand to add the correct parameter names for methods and constructors. *Java2AdaJNI* can then be used to process the updated JCS file resulting in the generation of a set of bindings with the correct parameter names.

6.11 JavaDoc Comments

JavaDoc is a tool, delivered as part of the Java Development Kit, which generates HTML documentation for any given Java class. The documentation is based, in part, on specially formatted comments in the source code for a class. *Java2AdaJNI* is able to extract these comments when processing Java source files and to add them to the corresponding Ada package specification.

6.12 Ada Tasking.

When using the JNI, a Java environment reference must be passed to all JNI functions. This reference is created by the JavaVM and is only valid for a particular thread. When a thread wishes to make use of JNI, it must first 'attach' itself to the JavaVM. This operation returns a new environment reference which must be used in subsequent calls to the JNI from that thread.

Because most Ada compilers use threads to implement tasks, a strategy is needed to ensure that tasks can safely make calls to the JNI. This is achieved in *AdaJNI* by requiring all tasks to make calls to the AdaJNI runtime procedures *Attach_Task* and *Detach_Task*. The *Attach_Task* procedure is called at the start of a task. It attaches the calling task’s thread to the JavaVM and stores the associated Java environment reference using the standard *Ada.Task_Attributes* package. The *Detach* procedure is called at the end of a task to disassociate the calling tasks thread from the Java VM.

Note that the *Attach_Task* and *Detach_Task* procedures are called automatically for the main thread and can therefore be ignored by the programmer of non-tasking applications.

6.13 Event Handling.

The *Abstract Windows Toolkit* (AWT)[4] and *Java Beans*[6] share a common event model based on *event listeners.* A number of pre-defined *event listener interfaces* are provided with Java to model the handling of events associated with things such as mouse use, window manipulation and slider control. Classes which implement these interfaces are developed by the programmer to provide application specific event processing. Instances of these classes are then registered with event source objects such as buttons. When an event is generated by a source object, it is passed to each registered event listener object for processing.

The event management system adopted by AdaJNI integrates the standard Java event model with Ada by providing a set of Java adapter classes and corresponding Ada bindings which implement each of the standard event listener interfaces. These adapter classes process events by placing them on a queue to the AdaJNI runtime as depicted in Figure 3.

In order to use the AdaJNI event model an Ada call-back procedure is declared for each event of interest. Each call-back procedure is then registered with the AdaJNI runtime which allocates a call-back identifier for each procedure. Once a call-back procedure is registered, its identifier can be used during the construction of an AdaJNI event

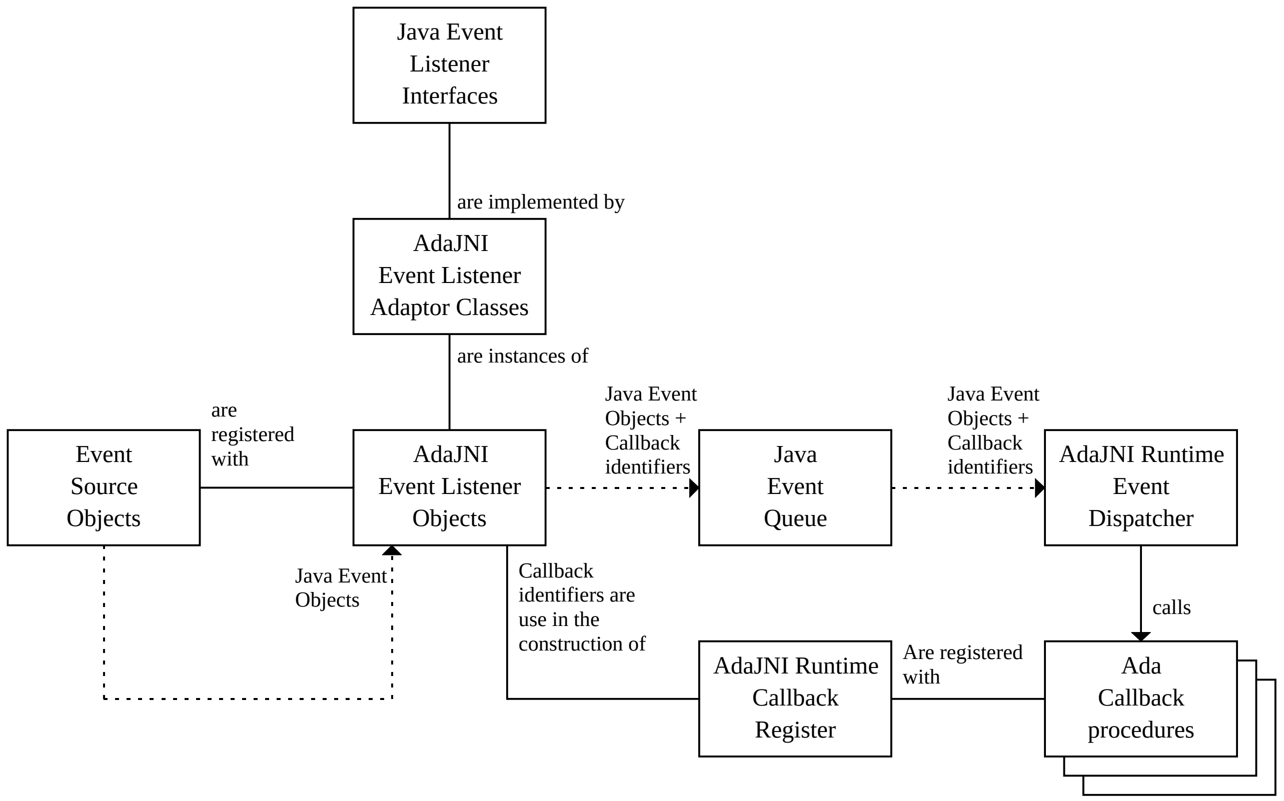
Java Event Listener Interfaces
are implemented by
AdaJNI Event Listener Adaptor Classes
are instances of
Event Source Objects
are registered with
AdaJNI Event Listener Objects
Java Event Objects + Callback identifiers
Java Event Queue
Java Event Objects + Callback identifiers
AdaJNI Runtime Event Dispatcher
Java Event Objects
Callback identifiers are use in the construction of
calls
AdaJNI Runtime Callback Register
Are registered with
Ada Callback procedures

Figure 3. AdaJNI Java Event Management

listener object. Once created, a listener object is associated with a particular Ada call-back procedure.

The next step, is to register AdaJNI event listener objects with event source objects such as buttons and windows. When an event source generates an event, it is passed to each registered event listener object. These objects then place the events, along with the identifiers of associated Ada call-back procedures, onto a simple queue.

Event processing is completed when an Ada application calls an event dispatching procedure in the AdaJNI runtime. This procedure will remove an event from the queue and pass it to the associated Ada call-back procedure for processing as required by the application.

7. APPLICATIONS OF ADAJNI

AdaJNI supports the use of Ada and Java in a wide variety of software architectures including stand alone applications, client server systems, and systems distributed over local and/or wide area networks. Applications can comprise components written in any combination of Ada and Java. Ada components can make use of Java APIs and can communicate with components written in Java using the AdaJNI system.

In standalone applications, Ada programs can make use of Java API's via AdaJNI bindings. If such applications are limited to using only the standard Ada packages and Java API's which are supported on a variety of platforms, the application code will compile on those platforms without any changes. That is, the applications are portable at the source level.

Applications can also comprise a combination of Ada and Java components. AdaJNI can be used to generate Ada bindings to Java components of the application as well as standard Java and third party APIs. This architecture allows use of the most appropriate language for each component. More importantly, it allows the integration of existing/legacy Ada and Java code to build new systems.

Java provides a comprehensive set of APIs for the development of distributed applications. By using AdaJNI, Ada programs can make use of the same APIs thus allowing the development of applications comprising Ada and Java components distributed over local and/or wide area networks. The use of AdaJNI in client server architectures would allow the server component to be developed using robust native Ada compilers. The server would use AdaJNI bindings to communicate with clients running in a Java Virtual machine. The client software could be written in Java or Ada using one of the Ada byte code compilers.

8. A SMALL EXAMPLE

This section shows how AdaJNI can be used to generate an Ada binding to a simple Java class and how that binding can be used to call Java methods from Ada.

8.1 The Java Class

The following Java class contains a single method which displays the text 'Hello Ada!' a number of times as determined by the integer parameter 'count'. The remainder of this section shows how this class can be called from Ada.

```
public class ExampleClass {
  public void sayHello(int count) {
    int i;
    for ( i=0; i<count ;i++ )
      System.out.println("Hello Ada!");
  }
}
```

8.2 The Ada Binding

The *Java2AdaJNI* binding generator tools can be used to create an Ada binding to the class described above. The primary specification for such a binding is shown below. In producing this specification, *Java2AdaJNI* analyses the Java source file described above along with the associated Java class file produced when the class is compiled using the *javac* Java compiler.

```
package Ada_JNI.Bindings.Example_Class is

  type Object is
    new Ada_JNI.Bindings.Java_Lang.Object.Object
        with null record;

  type Reference is
    access all Object'class;

  function New_Example_Class
    ( Env : in Ada_JNI.Java.Java_Env
                := Ada_JNI.Java.Current_Env
    )
      return Reference;

  procedure Say_Hello
    ( Ref   : access Object;
      Count : in Ada_JNI.Java.Int.Java_Int;
      Env   : in Ada_JNI.Java.Java_Env
                  := Ada_JNI.Java.Current_Env
    );
end Ada_JNI.Bindings.Example_Class;
```

Note that the Java source code is not necessary to generate the above specification. *Java2AdaJNI* can generate a complete binding from the class file alone. In such cases, however, *Java2AdaJNI* is unable to generate proper parameter names for methods and constructors because they are not stored in Java class file.

8.3 Use of the Ada Bindings

The following Ada program makes use of the Ada binding described above. It declares a new object reference called "*My_Object*". The body of the procedure creates a Java object by calling the constructor function "*New_Example_Class*". Finally, the "*Say_Hello*" method is called for the newly created object.

```
with Ada_JNI.Bindings.Example_Class;

procedure Example is
  My_Object :  Ada_JNI.Bindings
              .Example_Class.Reference;
begin

  My_Object :=  Ada_JNI.Bindings.Example_Class
               .New_Example_Class;

  Ada_JNI.Bindings.Example_Class.Say_Hello
    ( Ref   => My_Object,
      Count => 10
    );
end Example;
```

9. A TYPICAL BINDING - java.awt.Button

Figure 4 shows the primary specification of a typical Java API binding generated by the *Java2AdaJNI* tool. The example shown was generated from source code for the *java.awt.Button* Java class. Note that the code shown is exactly as produced by the tool. It has not been edited in any way and includes the correct parameter names for all methods as declared in the original Java source code.

10. FUTURE WORK

While the AdaJNI technology has been developed on the Windows NT platform, it does not contain any platform specific software and should port easily to common UNIX variants. Future work will address these porting issues.

In addition, investigations will be undertaken as to how common bindings may be developed so as to work with both standard Ada compilers and the JavaVM Ada compilers produced by Intermetrics and Aonix. The aim would be to modify *Java2AdaJNI* so that it generated a single set of package specifications for the bindings along with two sets of package bodies. One set of bodies would be designed to work with native compilers while the second would work with the JavaVM compilers. Such an approach would allow the Ada programmer to develop a single code base which could be compiled on different compilers depending on portability, target and performance requirements.

```
with Ada_JNI.Binding_Types.Java_Awt_Event.Action_Listener;
with Ada_JNI.Binding_Types.Java_Lang.String;
with Ada_JNI.Bindings.Java_Awt.Component;
with Ada_JNI.Binding_Types.Java_Awt.Button;
with Ada_JNI.Java;
with Ada_JNI.Bindings.Java_Lang.Object;

package Ada_JNI.Bindings.Java_Awt.Button is

  type Object is
    new Ada_JNI.Bindings.Java_Awt.Component.Object with null record;

  type Reference is
    access all Object'class;

  --------------------------------------------------------------------------------
  -- Type Conversions                                                           --
  --------------------------------------------------------------------------------

  function "+"
    ( Ref : in Ada_JNI.Binding_Types.Java_Awt.Button.Reference
    )
      return Reference;

  function "+"
    ( Ref : in Reference
    )
      return Ada_JNI.Binding_Types.Java_Awt.Button.Reference;

  --------------------------------------------------------------------------------
  -- Constructors                                                               --
  --------------------------------------------------------------------------------

  function New_Button
    ( Env : in     Ada_JNI.Java.Java_Env := Ada_JNI.Java.Current_Env
    )
      return Ada_JNI.Bindings.Java_Awt.Button.Reference;

  function New_Button
    ( Label : in     Ada_JNI.Binding_Types.Java_Lang.String.Reference;
      Env   : in     Ada_JNI.Java.Java_Env := Ada_JNI.Java.Current_Env
    )
      return Ada_JNI.Bindings.Java_Awt.Button.Reference;

  --------------------------------------------------------------------------------
  -- Methods                                                                    --
  --------------------------------------------------------------------------------

  procedure Add_Notify
    ( Ref : access Ada_JNI.Bindings.Java_Awt.Button.Object;
      Env : in     Ada_JNI.Java.Java_Env := Ada_JNI.Java.Current_Env
    );

  function Get_Label
    ( Ref : access Ada_JNI.Bindings.Java_Awt.Button.Object;
      Env : in     Ada_JNI.Java.Java_Env := Ada_JNI.Java.Current_Env
    )
      return Ada_JNI.Binding_Types.Java_Lang.String.Reference;

  procedure Set_Label
    ( Ref   : access Ada_JNI.Bindings.Java_Awt.Button.Object;
      Label : in     Ada_JNI.Binding_Types.Java_Lang.String.Reference;
      Env   : in     Ada_JNI.Java.Java_Env := Ada_JNI.Java.Current_Env
    );

  procedure Set_Action_Command
    ( Ref     : access Ada_JNI.Bindings.Java_Awt.Button.Object;
      Command : in     Ada_JNI.Binding_Types.Java_Lang.String.Reference;
      Env     : in     Ada_JNI.Java.Java_Env := Ada_JNI.Java.Current_Env
    );

  function Get_Action_Command
    ( Ref : access Ada_JNI.Bindings.Java_Awt.Button.Object;
      Env : in     Ada_JNI.Java.Java_Env := Ada_JNI.Java.Current_Env
    )
      return Ada_JNI.Binding_Types.Java_Lang.String.Reference;

  procedure Add_Action_Listener
    ( Ref : access Ada_JNI.Bindings.Java_Awt.Button.Object;
      L   : in     Ada_JNI.Binding_Types.Java_Awt_Event.Action_Listener.Reference;
      Env : in     Ada_JNI.Java.Java_Env := Ada_JNI.Java.Current_Env
    );

  procedure Remove_Action_Listener
    ( Ref : access Ada_JNI.Bindings.Java_Awt.Button.Object;
      L   : in     Ada_JNI.Binding_Types.Java_Awt_Event.Action_Listener.Reference;
      Env : in     Ada_JNI.Java.Java_Env := Ada_JNI.Java.Current_Env
    );

  --------------------------------------------------------------------------------
  -- Fields                                                                     --
  --------------------------------------------------------------------------------

end Ada_JNI.Bindings.Java_Awt.Button;
```

Figure 4. Ada Bindings Specification for java.awt.Button

11. CONCLUSION

An approach to using Java APIs with native Ada compilers has been described. The approach, which is based on use of the Java Native Interface, has been implemented in a system called AdaJNI. AdaJNI includes a tool which automatically generates Ada bindings to any given Java class as well as a small runtime to support the operation of such bindings. A number of issues associated with the mapping of Java to Ada have been raised and addressed during the development of AdaJNI. These issues have been discussed in this paper.

12. ACKNOWLEDGMENTS

The author wishes to thank Ed Falis, a software engineer at Aonix, for using AdaJNI and providing very useful feedback which has led to a number of significant improvements in the technique. In addition, the author wishes to thank Dr Clive Boughton, managing director of Software Improvements Pty Limited, for his ongoing encouragement and for his review of this paper.

13. REFERENCES

[1] Intermetrics Inc. “AppletMagic”, http://www.appletmagic.com

[2] JavaSoft, Sun Microsystems Inc., “Java Development Kit”, http://java.sun.com/products/jdk/1.1

[3] JavaSoft, Sun Microsystems Inc., “Introduction to the New AWT Event Model”, http://java.sun.com/docs/books/tutorial/ui/components/eventintro.html

[4] JavaSoft, Sun Microsystems Inc., “Java API User’s Guide”, http://java.sun.com/products/jdk/1.1/docs/api/API_users_guide.html

[5] JavaSoft, Sun Microsystems Inc., “Java Foundation Classes”, http://java.sun.com/products/jfc/index.html

[6] JavaSoft, Sun Microsystems Inc., “Java Beans”, http://java.sun.com/beans/index.html.

[7] Masters, M.W. “Programming Languages and Life Cycle Costs”, Naval Surface Warfare Center, March 1996, http://wuarchive.wustl.edu/languages/ada/docs/advocacy/Masters1.zip)

[8] Zeigler, S.F. “Comparing Development Costs of C and Ada”, Rational Software Corporation, March 1995, http://sw-eng.falls-church.va.us/AdaIC/docs/reports/cada/cada_art.html)