CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 2003; **00**:1–18 *Prepared using* cpeauth.cls *[Version: 2002/09/19 v2.02]*

Experience in integrating Java with C# and .NET

Judith Bishop[1,*], R. Nigel Horspool[2] and Basil Worrall[1]

[1] *Computer Science Department, University of Pretoria, Pretoria 0002, South Africa, jbishop,bworrall@cs.up.ac.za*
[2] *Computer Science Department, University of Victoria, Victoria BC, Canada V8W 3P6, nigelh@uvic.ca*

SUMMARY

Java programmers cannot but be aware of the advent of C#, the .NET network environment, and a host of new supporting technologies, such as web services. Before taking the big step of moving all development to a new environment, programmers will want to know what are the advantages of C# as a language over Java, and whether the new and interesting features of C# and .NET can be incorporated into existing Java software. This paper surveys the advantages of C# and then presents and evaluates experience with connecting it to Java in a variety of ways. The first way provides evidence that Java can be linked to C# at the native code level, albeit through C++ wrappers. The second is a means for retaining the useful applet feature of Java in the server-side architecture of web services written in C#. The third is by providing a common XML-based class for the development of GUIs, which can be incorporated into Java or C#. An added advantage of this system, called Views, is that it can run independently of the resource-intensive development environment that would otherwise be needed for using C#. A major advantage of the methods described in this paper is that in all cases the Java program is not affected by the fact that it is interfacing with C#. The paper concludes that there are many common shared technologies that bring Java and C# close together, and that innovative ways of using others can open up opportunities not hitherto imagined.

KEY WORDS: integration, Java, C#, XML, native code, web services, GUI

*Correspondence to: Computer Science Department, University of Pretoria, Pretoria 0002, South Africa
Contract/grant sponsor: National Research Foundation; contract/grant number: 2050319

1. INTRODUCTION

Java has been with us for seven years now and has made phenomenal inroads into the world of system, business, internet and educational programming. As demonstrated by presentations made at conferences such as JavaGrande, its influence extends also into scientific and high performance computing, specifically in parallel and distributed applications [11]. The reason for Java being used by these latter communities is that it has *something to offer* over and above the languages currently in use – chiefly Fortran, Visual Basic and C/C++.

Specifically, object-oriented programming, increased security both within a program and between programs, parallelism facilities, applets and access to new resources through class libraries are cited as features which could be profitably used by scientific programmers [19].

The move towards Java in distributed computing has not been without its problems [17, 15], however, and it is to be expected that programmers will be loathe to embark upon another change of language so soon. Yet, the advent of Microsoft's new language C# cannot go unnoticed, and the questions to be asked are:

- What are the additional advantages of C# as a language over Java?
- Can the new and interesting features of C# be incorporated into existing Java software?

Like Java, C# is not just a programming language, but co-exists with a particular runtime enironment (like Java's JVM), a means of communicating on the network (like Java's RMI but unlike Java's applets) and several independent technologies which are used by both languages (such as XML).

The purpose of this paper is to present experience of C# co-existing with Java in several ways, and to indicate which avenues of approach are likely to be fruitful in the near and medium future. The paper serves as a survey of possibilities, some of which are explained in more depth elsewhere [12, 21]. Whereas Lobosco *et al.* [15] survey some fourteen specialised projects for adapting Java for high-performance computing, we concentrate on exploiting freely available (if not always free) application independent technologies in this area.

The paper is organised as follows: Section 2 contrasts Java with C# at the language level. Section 3 considers the issue of integration and interoperability in general. Sections 4 to 6 look at integration between Java and C# across languages, across computers and with the user. The theme in these sections is to show how the integration with other technologies can be used to enhance Java programs. Section 7 outlines continuing and future work; Section 8 gives our conclusions.

2. ADVANTAGES OF C# FOR JAVA PROGRAMMERS

C# is an object-oriented language developed by Microsoft to act as a worthy successor to C++ and Visual Basic. Like its close cousin, Java, it compiles to an intermediate code (Common Language Infrastructure or CIL) and is JIT compiled to native code. A notable difference, and a plus for C# for the high performance computing, is that CIL is always JIT compiled, and the Microsoft's original VM implementation (known as the Common Language Runtime or CLR) is highly optimised, whereas Sun's JVM was not.

Much of C# looks the same as Java and has very similar semantics. At the language level, its object-oriented model is very similar in that it defines classes and interfaces, does not have multiple inheritance, and relies on garbage collection at runtime (for the parts of the program known as 'managed code').

C# has several new features which make it interesting for Java programmers. Those which we have identified (not an exhaustive list) are:

- Operator overloading, similar to that in C++, is provided.
- The `switch` statement can select between cases using a string value as the selector (see Figure 6 for an excellent example of this feature).
- The indexing operator `[ ]` is overloaded for all collections. It allows transparent substitution of arrays by more complex data structures, such as hash tables, as the need arises.
- Input-output and file input-output is simpler than in Java, and made more powerful by the introduction of formatting features.
- Objects can be serialized in either binary or XML. The XML format would typically be used across the network or between programs.
- There is a `Dispose` method for deterministic control of releasing resources held by classes.
- Structs (lightweight objects) are available. They are allocated on the stack and are not subject to garbage collection.
- Values are automatically converted to objects and back as required.
- The `foreach` statement provides iteration over collections of objects (similar to, but much simpler than, the iterators in Java).
- Properties are available for all data members of a class. They provide a simple syntax for the definition of `get` and `set` methods.
- Multidimensional arrays are available in both rectangular and jagged forms. Rectangular arrays represent a contiguous block of memory and are therefore more amenable to the compiler loop optimizations expected in high performance computing applications. Jagged arrays, implemented as vectors of references to subarrays, are the only form of array provided in Java.
- Verbatim strings avoid the use of escape characters, and allow multi-line strings, a feature which was useful for the development of the XML based GUI class described in section 6.2.
- Overflow can be detected or ignored in expressions and type conversions, as desired by the programmer.
- Delegates provide callback functions, akin to functors in C++.
- Enumerator types are supported and values can be read and written.

On the debit side, C# does not yet have inner classes, `strictfp` (for enforcing IEEE 754 floating point) and of course cross-platform runnability. At present, Microsoft provides a professional version of C# only for Windows. However, a full version of C# but which lacks some optimizations and some class libraries has been made available as part of the SSCLI open source software distribution[20]. The SSCLI is available for Windows, FreeBSD Unix and MacOS X, and an independent implementation of C# is also available for Linux[16]. An important advantage of C# is that the language and the CLI have been standardised by ISO

[13] and ECMA [9]. The documents are available online and provide a reference point for implementations of the language by other vendors.

At this stage, there are several professional books on C#, such as [8, 23] and also several independent online resources such as [7]. In addition, there are some independent comparisons of C# and Java, including [18] and [14]. To date there has not been a similar comparison of supporting technologies, but [21] aims to fill that gap.

3. SUPPORTING TECHNOLOGIES FOR INTEGRATION

There is an almost overwhelming range of supporting technologies for Java, and an equal number for C#. Some of them, such as http, XML and SOAP, are independent and shared by both languages. Others are specific to Java or to C# and do not have counterparts in the other camp, such as JavaBeans on the one hand and Web Forms on the other. It is clearly important to know which technologies are shared and which are not. It is not always easy to make the distinction, since Sun and Microsoft, being vendors, have a vested interest in presenting technology in a propriety manner. For example, Sun calls its XML messaging capability JAXM and Microsoft's distributed and reusable components are called ActiveX controls.

A programmer trained in Java, say two or three years ago, is faced with two challenges:

- sifting out the new technologies that are relevant and would enhance productivity, and
- learning how to integrate these technologies into existing software successfully.

We shall go through three high-profile technologies, and show how they can be used to great effect to bring C# into Java programs. These are JNI, web services, and XML.

4. INTEGRATING VIA NATIVE CODE

Systems often achieve a high level of optimisation by utilising system level libraries. Consider, for example, the networking and threading APIs in Java, which rely almost completely on native system libraries for their functionality. From Java's point of view, this native code is represented in the object files compiled from C, C++ or Assembler source files. The Java classes are unaware of the fact that they are integrated with this *native code.*

It is possible, in time, that existing Java systems could be converted to C#. Even though Java and C# are very similar in their syntax, this conversion process is likely to be arduous. It is in the use of those language features that distinguish C# from its predecessor and near cousin that will hinder the conversion. We propose that, instead of retooling an entire system in C# and retraining programmers to do so, Java's natural ability to interact with native code be harnessed. Then, as and when needed, new parts of a system can be written in the new language.

We start our discussion by considering the classic case of native code interfacing in Java. Native code is called from Java using JNI, the Java Native Interface, which comes standard with the JDK. JNI supports code portability across all platforms and allows code written in C/C++/Assembler to run alongside the JVM (Java Virtual Machine). The interface works at

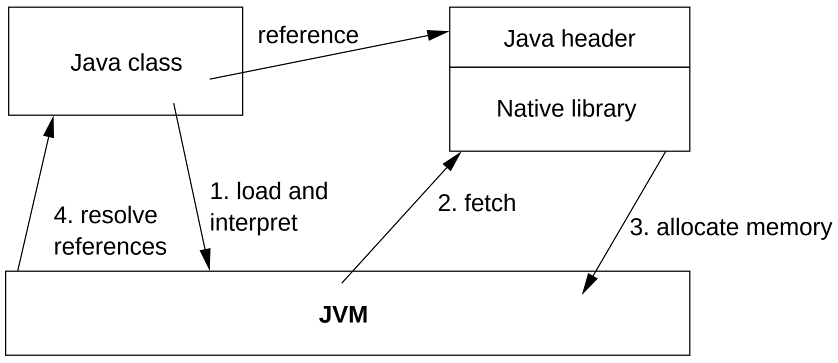


Figure 1. The players in classic native interfacing

the level of method calls, so that native methods can create, update and inspect Java objects and call their methods. Figure 1 shows the players in this game. The Java class is first written with embedded calls to native code. The native code need not be available at compile time – the methods can be prefixed with the `native` keyword that will alert the compiler to the special needs. The class will also include static code that references the native library in which the methods it refers to can be found. At run-time, the JVM loads and interprets the Java class file. It then fetches the native library and allocates memory to store it internally. Using the information it finds in the native library, it can resolve the references in the Java class. At this point normal execution can begin.

There are two important issues that the preceding discussion ignores. Firstly, the method headers in the native code need to be compatible with the Java method calling convention. This is achieved by producing a C/C++ compatible header file from the Java code and importing this header file in the native source. The `javah` command-line tool can be used to produce this header file. Secondly, the native library needs to be static. In other words, all references made in the library need to be resolved immediately after loading it into memory. Although this requirement does not seriously inhibit ordinary native code, it does play a role in our Java-to-C# interaction implementation. A full discussion of the steps to using JNI, and the issues raised in doing so, can be found in [22].

4.1. Summary of JNI for C

We have looked at the native interface process from a relatively high level. A small example that will help solidify the roles of the players is given below. The Java side includes a method stub like the following.

```
    public native void displayHelloWorld();
```

The `javah` tool is then used to generate a C header file, which includes the declaration

```
JNIEXPORT void JNICALL
   Java_HelloWorld_displayHelloWorld(
   JNIEnv *, jobject);
```

The method we started out with, displayHelloWorld, is now prefixed with the word *Java* and the name of the class, making it `Java_HelloWorld_displayHelloWorld`. We implement this method in C, say, as follows.

```
#include <jni.h>
#include "HelloWorld.h"
#include <stdio.h>
JNIEXPORT void JNICALL
   Java_HelloWorld_displayHelloWorld(
     JNIEnv*env,jobject obj) {
       printf("Hello world!\n");
     return;
}
```

Then the Java code to activate everything is as follows.

```
class HelloWorld {
  public native void displayHelloWorld();
  static {
    System.loadLibrary("HelloWorld");
  }
  public static void main(String[] args) {
     new HelloWorld().displayHelloWorld();
  }
}
```

When the `HelloWorld` class is instantiated, the JNI subsystem loads the library consisting of the object code generated from the HelloWorld.cpp file. Several layers of wrapping are needed because the DLL produced by compiling a C# program into a library is not compatible with Java's JNI facility. There are also calling convention difficulties. The header files which Java requires (and which are supplied by a C program) specify the nature of the calling convention, return types, parameters types and exception handling. There is no easy way, were it even feasible, to force the C# compiler to accept these restrictions.

The double layer of C++ above is required because the DLL with which the JNI subsystem interacts has to be static, procedural code (i.e. not object-oriented). So the first layer is essentially C code. Fortunately, such static C code will accommodate (static) pointers to C++ objects, which is where the second C++ layer comes in. It is doubtful whether the static C code could interact directly with a C# object, since there are things like garbage collection to consider.

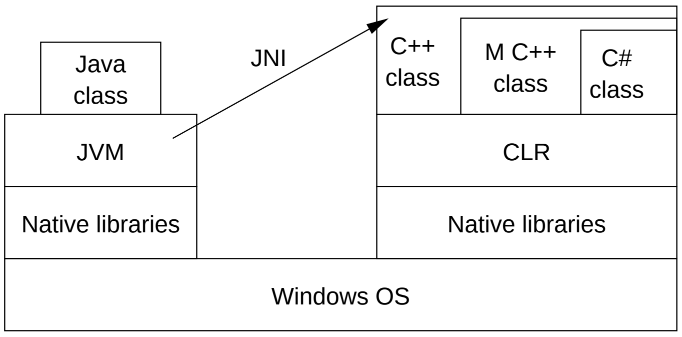


Figure 2. Using JNI for Java-to-C# interface

4.2. Linking Java to C# via JNI

The interface between Java and native code is well-established and has been in use since the very beginning of the language. Our objective was to attempt the same functionality, but with C# libraries taking the place of the native code. Those familiar with the C# environment will know that library code (and executables) can, in fact, be produced from C# source code. The question, therefore, is whether these libraries are compatible with the Java native interface model. Our investigation has shown that this is not the case. The incompatibility is introduced because of the calling convention. The header files produced from the Java code are intended to be used with C or C++ source code, and it not a trivial task to emulate the same behaviour in C#.

There does exist, however, a workable solution to this incompatibility. The CLR that executes C# code also provides support for many other languages through the use of the Intermediate Language (IL) representation that source code is compiled to before execution. Versions of Visual Basic and C++ have been provided which are fully compatible with the CLR. It is possible to use the C++ implementation to trick Java into believing that it is involved in an ordinary native code interaction.

Figure 2 shows how the Java-to-C# interface is set up. On the left-hand side we have the Java class, which contains references to the methods written in C#. This class interacts with the JVM in as in Figure 1. The difference is that the native library the class refers to is in fact C++ wrapper code. Inside the wrapper we have a reference to a *managed* C++ class. This is actually a class written in C++ with Managed Extensions, a special version of the language that can interact with CLR libraries, and also supports garbage collection. We need this class to set up the reference to the C# class (which is also subject to garbage collection). In doing this, we ensure that all references remain intact when garbage collection is performed. In short, when the native method in the Java class is invoked, a wrapper method in C++ is called, which in turn calls a method in Managed C++, and finally this method calls the C# method.

To summarise, the interface is complicated by two concerns:

1. C# does not compile to true native object code, and does not support C++ header files; and
2. objects in both Java and C# are subject to garbage collection, so we cannot assume that references to them will be valid after a collection cycle.

We now supply an example of a working Java-to-C# interaction based on the discussion above. This is the C# method that corresponds to the C++ method given in the previous example

```
using System;
public class CSharpHelloWorld {
  public CSharpHelloWorld() {}
  public void displayHelloWorld() {
    Console.WriteLine("Hello, World From C#!");
  }
}
```

The code that wraps the C# class in Managed C++ is given below. The class defines a reference to the C# class, and assigns that reference to a new instance in the constructor. The "method" method is provided for the (normal) C++ wrapper to call.

```
#using <mscorlib.dll>
#using "CSharpHelloWorld.netmodule"
using namespace System;

__gc class HelloWorldC {
public:
  CSharpHelloWorld __gc *t;

  HelloWorldC() {
    t = new CSharpHelloWorld();
  }

  void method() {
    t -> displayHelloWorld();
  }
};
```

The instruction to use mscorlib.dll at the top of this class corresponds with the inclusion of the jvm.h file at the top of the C code in the previous example. Note that we also use the compiled version of the C# code (`CSharpHelloWorld.netmodule`) to help resolve the reference made to it.

The original C++ wrapper is altered to interface with the Managed C++ class, by creating an instance of it in the Java method, and then invoking the method.

```
#include <jni.h>
#include "HelloWorld.h"
#include "HelloWorld.cpp"

JNIEXPORT void JNICALL Java_HelloWorld_displayHelloWorld
  (JNIEnv *jn, jobject) {
    HelloWorldC* t = new HelloWorldC();
    t->method();
  }
```

`HelloWorld.h` is the header file generated from the Java source, while `HelloWorld.cpp` is the Managed C++ source file.

4.3. Assessment

Our research has shown that it is possible to disguise the fact that C# is not native from Java's point of view. The technique to do so is complicated by the necessity of two layers of wrapping between the two languages. The question is whether this technique can be used generally, and with heavy-duty C#.

The example above is actually simplified because the "native" code is stateless. When state information needs to be maintained, it is necessary to implement static references that do not change when garbage collection is performed. Furthermore, the interaction shown above was one-way. Often native methods return references or values, or, indeed, expect to have values or references passed to them as parameters. JNI does provide mechanisms to facilitate this kind of interaction, especially when type marshalling between the languages is necessary. For example, the String objects in Java are not compatible with String objects in C#, or vice versa. However, they can both be decomposed into character array representations, and this allows the JNI methods to perform the marshalling. For a complete example that illustrates calls to the large C# class, Views, described above, using the techniques described here, see [24].

5. INTEGRATION USING WEB SERVICES

A language such as Java has several ways of connecting to programs in other computers. Some of the well known ones are:

- http and file protocols
- sockets
- remote method invocation (RMI)
- database connectivity (JDBC)
- server pages (JSP)

In each of these cases, a client in Java (either an applet or an application) is linked to a server, also in Java. Using the techniques discussed in section 4, it is possible to have the server written in C# or another modern language. A detailed description of such a system for GIS imaging with a database in C/C++ and clients in Java is covered in [5].

However, the advent of .NET has popularised web services. A *web service* is a software component that exposes useful functionality to other programs (clients) on the web. Web services can be seen as independent server components, with interfaces carefully defined in a standard *interface notation* and connected via a standard *communication protocol* (rather than a Java-specific one). The notation for defining the data format is provided by the XML-based Web Services Definition Language (WSDL). WSDL defines the set of operations and messages that can be sent to and received from a given web service in an abstract, language-independent manner. The communication protocol favoured by web services is the Simple Object Access Protocol (SOAP), and WSDL has built-in support for binding to SOAP. A SOAP link enables a client to do a remote procedure call via a SOAP message on all the functions of the web service that are exposed to the web in its WSDL definition. The response from the web service will also be in the form of a SOAP message. The question we address is: to what extent, and how easily, can Java access web services which are written in C#?

5.1. The .NET framework

The .NET framework .NET supports many languages by providing a Common Language Runtime (CLR) which implements the Common Language Infrastructure (CLI) (see Figure 2). Application execution is not interpreted, as it may be in Java, but all supported languages are compiled to an intermediate representation, which is then JIT compiled to the operating system's native code. The .NET framework is the programming model of the Windows platform and is used for the construction and deployment of both local and distributed services and applications.

A primary goal of .NET is inter-operability of languages, especially in a distributed and networked environment. Unfortunately, full Java is not supported on .NET, and therefore to interact between Java and C# at the distributed level requires additional steps.

Typically, we would like to have an existing Java applet call a C# component running on .NET. However, unlike Java, .NET relies on server side processing. There is no concept of an applet, where the runtime machine is held in the browser. Instead, the state is kept on the server, and passed down to the client as server pages. Although intrinsically more complex than the client-side program model, the .NET environment in reality simplifies the server side processing structure.

When integrating Java with C#, we have developed an interesting alternative to the standard .NET approach. We can create and maintain a Java applet on the client side that is able to interact with the server as a .NET web service. Web service capability is inherent to the .NET framework. It allows developers to create web service through simple inheritance. Classes inherit from the System.Web.WebService class. A C# method is marked with a [WebMethod] attribute to indicate to ASP.NET that it should be exposed. .NET automates the installation of the web service object on its Internet Information Services (IIS) and also automates the generation of a *service contract file* in WSDL. The WSDL contract file contains all the types and host location information to allow clients to communicate and ultimately use the service. The client in our case is intended to be written in Java, which as we have already indicated, is not inherently supported on .NET.

5.2. Linking Java to a C# web service

At this point, the developer has to acquire tools to handle the SOAP and XML generation. The Java XML package JAX can be used for generating SOAP messages, and Xerxes [25] is also a well-known parser used to interpret XML files. Axis [1] is a SOAP implementation toolkit which enables static proxy generation given a web service URL. Xerxes is a parser used to interpret XML files. Together, these two packages are used to enable SOAP protocol communication and XML (SOAP, WSDL) file interpretation.

The logical steps in the consumption of a web service are as follows.

1. The user locates the WSDL file on the IIS server via a URL.
2. The Java client generates a Java proxy to access the web service using Axis and Xerxes.
3. The user accesses the generated proxy.
4. Communication between Axis and IIS via SOAP is established.
5. The C# web service executes and creates a SOAP message in return.
6. The Java client consumes the reply.

SOAP messages are interpreted and processed by the .NET web service object hosted on IIS. The WSDL contract file specifies the types required to transfer as well as details on how the client can use the service. The client does not need to be aware of the implementation technologies used to execute the service. The Java program is in no way aware that a C# web service is being used. Figure 3 shows the components involved in this process.

A worked example of this process can be found at [21]. Here, a multi-currency calculator in Java is linked to a C# web service which produces exchange rates. The web service can be online, and constantly updated with the latest rates.

5.3. Assessment

If tools for proxy and SOAP generation are used, then the connection between Java and C# web services is remarkably straightforward. As far as the programmer is concerned, only one or two extra lines of Java are required to link to the Locator in the proxy, which goes via IIS to find the service. The call to the web service method is then no different to a normal method call, or for example, a normal RMI call. With an increasing number of web services coming on line, the ability to access them from Java, and in particular, from existing applets will considerably enhance their value and prolong their lifetime.

6. LANGUAGE INDEPENDENT USER INTERFACING

A significant portion of a programmer's time is taken with developing user interfaces. If these have been developed in one language, then transferring them to another language

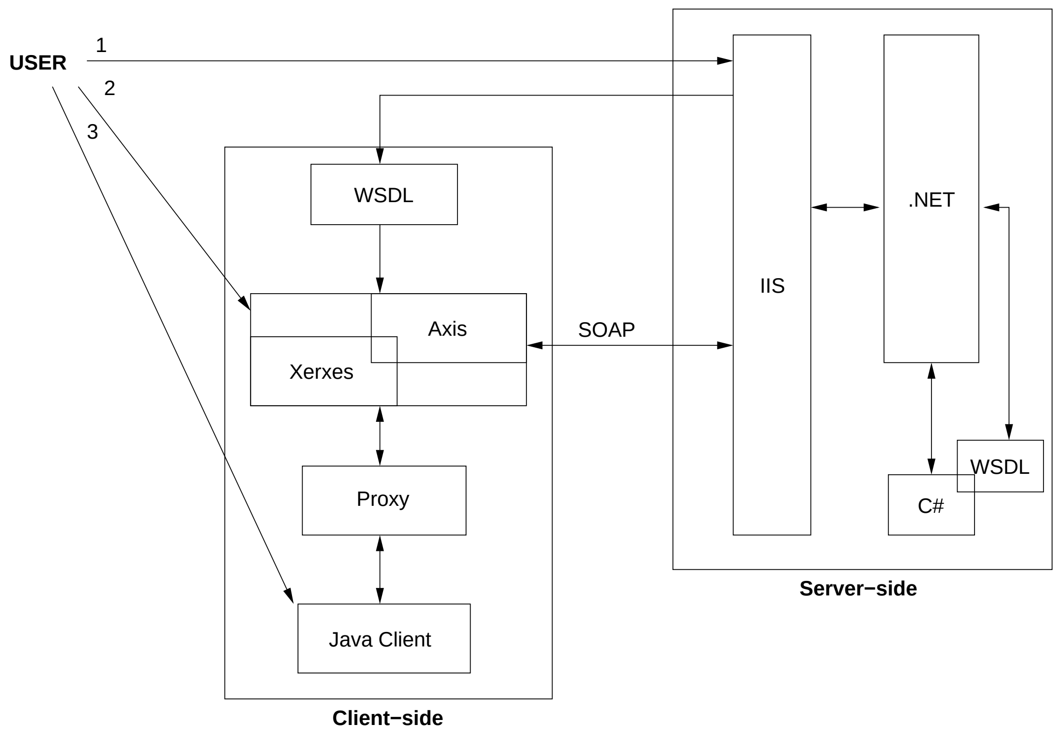


Figure 3. Client and server components involved in web service process

requires considerable effort to re-tool that section of the system. Programmers developing GUIs (Graphical User Interface) would normally use to builder tools with point and click interfaces, or would hand-write GUI code for class libraries such as Windows.Forms, AWT or Swing. We argue below that both approaches have disadvantages, and that a new approach is required.

We meet that challenge by using XML. XML is a universal format for data that provides structure to information so that it can be easily parsed [10]. Scientific programmers often have complex data sets to input or transfer. By using XML to describe the structure of the data, integrity can be ensured across programs, and between runs of a program. XML can be enhanced by XML-schemas that can specify meta-structure, and by XSL, which describes stylesheets, used for the presentation of a class of XML documents.

We illustrate the power and ease of use of XML by describing a tool which considerably eases the burden of writing visually appealing user-interfaces in both C# and Java. The tool is called Views [12, 2]. It was originally written in C#, but can be called from Java, as described in 4. It has also recently been successfully translated into Java.

6.1. Approaches to creating C# GUIs

We have investigated three options for C# GUIs:

Option 1: Use a visual form designer as is. Creating a GUI using such a tool is easy, especially for those who have experience in this paradigm. Components (or controls) are dragged onto the screen in the desired positions, and then properties are filled in to connect the actions associated with them to handler code. To avoid filling in the action properties, an interface class that will enforce a strict calling convention can be used. This is the approach used in Java's AWT and Swing packages. However, the code generated by the environment to implement the form is intimidatingly long and verbose. Furthermore, it is impossible to separate out dumped code from the programmer's written code in any viable form without editing the dumped code, which spoils the purpose. While some development environments allow the user to "collapse" lines of code that aren't meant to be edited, certain key features are also hidden. This key information includes the names of components, the text that is displayed and so on. Unfortunately this information is mixed up with non-key information such as the pixel position of the component, the text alignment and the tab index. The final two issues that cast such form designers and development environments in a bad light are their size and their platform dependence, which makes them unsuitable for running on all computers. For instance, the open-source version of C# distributed with the SSCLI (for Windows, FreeBSD UNIX and Mac OS X) lacks this environmental support [20].

Option 2: Create the form without Visual Studio. Avoiding the environment helps very little. We programmed a simple "ButtonTest" example through Visual Studio first; it came to some two pages of dumped code and 10 lines of an `actionPerformed` event. The length of the dumped code is due to Windows Forms controls only having one parameterless constructor, and there being no flow managers. So even a simple instantiation of `Button` has to be followed by several (usually six) assignment statements. For example:

```
this.waitButton.BackColor = System.Drawing.SystemColors.Control;
this.waitButton.Location = new System.Drawing.Point(7, 128);
this.waitButton.Name = "waitButton";
this.waitButton.TabIndex = 2;
this.waitButton.Text = "Wait";
this.waitButton.Click += new System.EventHandler(
        this.CompleteOp);
```

We tried to massage the code down into its bare bones, much as it would be written by a human, but we could not achieve the desired effect. Windows forms are clearly not meant to be programmed by hand.

Option 3: Design a customised class. The final possibility is to hide the complexity in a special multi-purpose class which uses ordinary method calls to operate on a reasonably complex GUI. The setting up of the GUI could be done in one of two ways:

1. Have a small set of methods to call to create simple buttons and text boxes, or
2. use an XML description of the form to drive the setup.

The first alternative was used with the Display class in a current Java textbook [4]. However we now prefer the second alternative because it is far more extensible and XML has become an accepted notation. It is analogous to the embedding of SQL in JDBC calls. All of the above programming shown in *Option 2* would be replaced by the XML specification

```
<Button Name=Wait/>
```

which would be used by the constructor of the GUI class to create a button control.

6.2. Views – an independent GUI system

Each instance of the Views class creates a single Windows *form* (similar to a *Frame* in Java terminology) on the screen. Each form contains some simple *controls* (similar to *components* in Java), such as labels, buttons and text boxes that the calling program can use for GUI input and output.

The controls displayed on the form and their layout are specified by an XML string passed to the constructor. The user can subsequently set and get information that appears within the controls, e.g., the user can call the `GetText` method to obtain the text available inside a TextBox control. Figure 4 shows a Views Windows form for a simple scientific program to calculate the capacitance charge using Simpson's Rule.

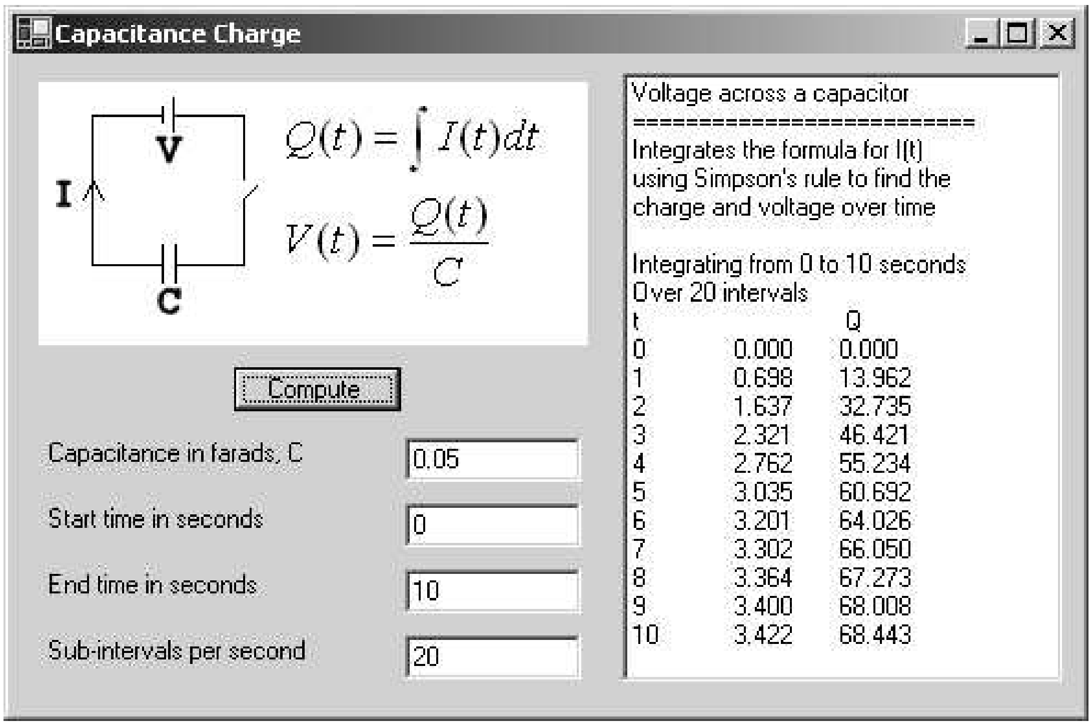
Figure 5 shows the XML specification needed to define that form. The `<horizontal>` and `<vertical>` XML tags simply group controls in horizontal or vertical lists. Otherwise, we have singleton tags for each control that is supported. The XML notation is infinitely extensible. We can add attributes to specify the size of a control, its colour, its alignment relative to other controls, its position in the form, and so on.

Tag names are the same as the names of the corresponding components in the C# Windows Forms class library and the attribute names match the names of classe properties. Case is ignored for the tag names and attribute names, though we normally follow the C# capitalization. Everything has a sensible default. Layout and sizing are automatic unless overridden by explicit attribute specifications.

The Views object created by the program maintains the controls specified in the XML. To interact with them, we use a series of methods, such as:

`string GetControl()` returns the id of the button that was pushed; by default, the id is the same as the text the user sees displayed on the button.

`string GetText(string id)` returns the text that is currently contained in the TextBox whose id is specified.



$$Q(t) = \int I(t)dt$$

$$V(t) = \frac{Q(t)}{C}$$

Capacitance in farads, C	0.05
Start time in seconds	0
End time in seconds	10
Sub-intervals per second	20

```
Voltage across a capacitor
==========================
Integrates the formula for I(t)
using Simpson's rule to find the
charge and voltage over time

Integrating from 0 to 10 seconds
Over 20 intervals
t              Q
0     0.000    0.000
1     0.698    13.962
2     1.637    32.735
3     2.321    46.421
4     2.762    55.234
5     3.035    60.692
6     3.201    64.026
7     3.302    66.050
8     3.364    67.273
9     3.400    68.008
10    3.422    68.443
```

Figure 4. A user interface created by Views

`void PutText(string id, string text)` writes the string text into the textbox or ListBox (a scrollable text window) whose id is specified.

There are a few others which get values from and put values into specific kinds of controls. Controls implemented in the Views class include Label, Button, TextBox, ListBox, OpenFileDialog, SaveFileDialog, TrackBar, ProgressBar and several others, including support for images. A full description of Views is given in [3].

Figure 6 shows some sample C# code that interacts with the controls in the form. This model of interaction reflects the *event loop* concept: the program waits for something to happen, and then passes control via a switch statement to the correct code representing an event handler. A more powerful, but perhaps more difficult to understand, mechanism is that of the *callback*, as used in Java and other languages. Views supports callbacks, by allowing the name of a method to be specified for a given event associated with a control, as in:

```
<Button Name=Wait Click=CompleteOp/>
```

```
Views.Form form = new Views.Form(@"<Form Text='Capacitance Charge'>
  <vertical>
    <horizontal>
      <vertical>
        <vertical Height='100'>
          <Picturebox Image='sketch.GIF'/>
          <Button Name='Compute' halign=centre/>
        </vertical>
        <horizontal>
          <vertical>
            <Label Text = 'Capacitance in farads, C' Width=150/>
            <Label Text = 'Start time in seconds' Width=150/>
            <Label Text = 'End time in seconds' Width=150/>
            <Label Text = 'Sub-intervals per second' Width=150/>
          </vertical>
          <vertical>
            <Textbox Name='capacitor' Text='0.05' Width=80/>
            <Textbox Name='starttime' Text='0' Width=80/>
            <Textbox Name='endtime' Text='10' Width=80/>
            <Textbox Name='intervals' Text='20' Width=80/>
          </vertical>
        </horizontal>
      </vertical>
      <Listbox Name = 'list' Width=200 Height=280/>
    </horizontal>
  </vertical>
</Form>");
```

Figure 5. XML specification for a GUI

where `Click` is an event associated with the `Wait` button control, and `CompleteOp` is a method which will be called when that event occurs. (Compare this line to the lengthy code in Section 6.2). At present, it is not known whether both models can exist in the same program, or whether it makes sense to have them do so.

6.3. Assessment

The XML notation is very powerful and extensible, and is revolutionising the way we write programs. For example, supposing we wished to add a button to cause the program to draw a graph of the process. All we need to add is

```
<Button Name='Draw Graph'/>
```

```
// ... in the main program
for (; ;) {
  string c = form.GetControl();
  if (c == null) break;
  ActionPerformed(c);
}
form.CloseGUI();
// ...
public void ActionPerformed(string c) {
  switch (c) {
  case "Compute":
    C =         double.Parse(form.GetText("capacitor"));
    start =     int.Parse(form.GetText("starttime"));
    end =       int.Parse(form.GetText("endtime"));
    intervals = int.Parse(form.GetText("intervals"));
    form.PutText("list", "Integrating from "+start+" to "+end+" seconds");
    form.PutText("list", "Over "+intervals+" intervals");
    form.PutText("list", "t\t\t Q\t\t  V");
    // ... integrate
    break;
  default:
    break;
  }
}
```

Figure 6. The C# Handler for the Views object

wherever we would like it to appear in the GUI, and then add the appropriate case to the `switch` statement (C#) or equivalent if-else statement (Java). While XML has been used in many contexts, we do not know of any attempts to integrate it as thoroughly into coding as we have done. The idea of using XML notation to specify the layout of a GUI is not a new one. The User Interface Markup Language (UIML) is an XML tagging scheme which was invented for exactly that purpose [6]. However, UIML is a complex tagging scheme, while our XML tags are greatly simplified by making them have the same names and attributes as the Winforms controls that they generate. The purpose of UIML is to be platform independent, while Views is very much oriented towards Windows. XUL is another XML scheme for specifying webforms layout, used with the Mozilla browser [26]. It too is platform-independent, and it incorporates scripting (similar to Javascript) for associating actions with mouse and keyboard events. The result is a XML scheme which is, again, much more difficult to learn and use than Views.

It should be noted that the implementation of Views is non-trivial. Its current implementation uses reflection extensively, as well as the XML class library, delegates, and implicit conversions for indexing the polymorphic table of controls..

7. FUTURE WORK

Work continues in two main areas:

- implementing more examples of the use of the technologies mentioned here, and assessing their performance;
- investigating other technologies for interaction, such as ODBC.

Within the high performance community, it is clearly important to assess the impact of layered approaches to software (as in the JNI solution). With SOAP messaging from applets, an important consideration would be the speed of graphics dispatch. The Views class has been ported to the SSCLI (Rotor), so that it can be run on FreeBSD UNIX and Mac OS X. Here the interactions between Java and C# can be tested all over again.

8. CONCLUSIONS

The relative merits of C# and Java will doubtlessly fuel arguments for many years. As a language, C# has corrected some deficiencies of Java and added a few new features. More importantly, C# is integrated into the .NET environment which provides access to web services and operating system services for the Windows platform. C# will join C++ and Visual Basic as a major programming language for developing Windows applications and web applications that run on Windows servers.

Java is also intended for general application development and web services development, on both the client side and server side. Java has the advantage of being platform independent, which C# may never become due to its dependence on Windows class libraries. However, there are certainly situations where C# will be preferable to Java. If applications are being developed for Windows, code written in C# will normally execute more efficiently, will have direct access to operating system services, and will more easily interoperate with programs written in other languages. The 'unmanaged code' feature of C# allows exactly that. This paper provides an example of Java calling C++ which, in turn, calls C#. The approach is entirely object-oriented, with class instances in Java communicating with class instances in C# via method calls.

While it is not easy to combine C# and Java in the same application program, it is possible. The example `HelloWorld` application in Java invokes an instance of a C# class, Views, for access to a GUI on Windows. There are potential difficulties working with pointers to objects, but these can easily be avoided.

The C# language provides direct access to operating system services and web system services in the Windows environment. This paper shows how Java code can be linked with C# code to obtain similar access, albeit yielding a program which is no longer platform independent.

ACKNOWLEDGEMENTS

This work was supported by grant 2050319 from the National Research Foundation of South Africa and a grant from the Natural Sciences and Engineering Research Council of Canada. We are grateful for the technical input and advice of Johnny Lo, Cobus Smit, John Muller, Kathrin Berg, Theo Danzfuss, Theo Crous and Rajwinder Panesar-Walawege. We acknowledge the helpful comments of the anonymous referees which improved the paper.

REFERENCES

1. AXIS: Apache SOAP implementation. `http://xml.apache.org/axis` [January 2003].
2. Bishop Judith, Horspool R Nigel. Views: A vendor-independent extendable windowing system. Presented at IFIP WG2.4 Dagstuhl, 15 November 2002. `http://www.cs.up.ac.za/~jbishop/rotor/`
3. Bishop Judith, Horspool R Nigel. *C# Concisely* Addison Wesley Publishers, Harlow, 2004.
4. Bishop Judith, Bishop Nigel T. Object-orientation in Java for scientific programmers. In *Proc. 22nd International SIGCSE Technical Symposium on Computer Science.* Austin, March 2000; 205-216.
5. Coetzee Serena, Bishop Judith. GIS and the Internet. Presented at *Minisymposium: High performance GIS: from Parallel Algorithms to Systems.* Naples Italy, September 2001.
6. Cover Robin. User Interface Markup Language. October 2002. `http://www.oasis-open.org/cover/uiml.html` [June 2003].
7. C# Links and Resources. Several articles on C#. `http://www.webreference.com/programming/csharp/` [June 2002].
8. Drayton Peter, Albahari Ben, Neward Ted. *C# in a Nutshell: A Desktop Quick Reference* (1st edn). O'Reilly & Associates: California, March 2002.
9. ECMA Standard 334 December 2002. `www.ecma-international.org/publications/files/ecma-st/Ecma-334.pdf` [September 2003].
10. Extensible Markup Language (XML). `http://www.w3c.org/XML/` [June 2003].
11. Getov Vladimir, von Laszewski Gregor, Philippsen Michael, Foster Ian. Multiparadigm communications in Java for grid computing. *Communications of ACM* 2001; **44**(1):118-125.
12. Horspool R Nigel, Bishop Judith. Views website. `http://www.cs.up.ac.za/~jbishop/rotor` [September 2003].
13. ISO/IEC Standard 23270:2003 `http://www.iso.ch/` [September 2003]
14. Johnson Mark. C#: a language alternative or just J–?. *JavaWorld*, December 2002. `http://www.javaworld.com/javaworld/jw-11-2000/jw-1122-csharp1p2.html` [May 2003].
15. Lobosco Marcelo, Amorim Claudio, Loques Orlando. Java for high performance network based computing: a survey. *Concurrency and Computation – Practice and Experience* 2002; **14**(1):1-32.
16. MCS: The Ximian C# compiler. `http://www.go-mono.com/c-sharp.html`
17. Moreira J, Midkiff S, Gupta M, Artigas P, Wu P, Almasi G. The Ninja Project. *Communications of ACM* 2001; **44**1:102-109.
18. Obasanjo Dare. A comparison of Microsoft's C# programming language to Sun Microsystems' Java programming language. `http://www.25hoursaday.com/CsharpVsJava.html` [January 2002].
19. Pancake C M, Lengauer C. High-performance Java. *Communications of ACM* 2001; **44**(1):99-101.
20. Rotor Projects. `http://research.microsoft.com/collaboration/university/europe/rotor/` [September 2003].
21. Smit Cobus, Muller John, Bishop Judith and van Zyl Jay. J2EE platforms and Microsoft .NET: a technological comparison. Technical report, Department of Computer Science, University of Pretoria, May 2002.
22. Stearns Beth Trail: Java Native Interface. `http://java.sun.com/docs/books/tutorial/native1.1/` [September 2003].
23. Troelsen Andrew. *C# and the .NET platform* (1st edn). Apress: New York, 2001.
24. Worrall Basil, Lo Johnny. Integrating C# and Java. March 2002. `http://www.cs.up.ac.za/polelo/interests.html` [June 2002].
25. Xerxes: Apache XML Parser. `http://xml.apache.org` [January 2003].
26. XUL: XML-based User Interface Language. June 1999. `http://www.mozilla.org/xpfe/xptoolkit/xulintro.html` [September 2003].