



## Using Java For Soft Real-Time Simulation

January 28, 2004

Rob Hatcherson  
Keith Holt  
Stephen Tarter

ZedaSoft, Inc.  
2310 Gravel Dr.  
Fort Worth TX 76118  
817-616-1000  
<http://www.zedasoft.com>

© Copyright 2004 by ZedaSoft, Inc.

All rights reserved. All trademarks are property of their respective companies.

## Introduction

In late 2002 ZedaSoft was presented with the opportunity to implement an object-oriented simulation design that was originally conceived in the summer of 2002 by a ZedaSoft team whose members included the authors of this paper. We called the design the Container-Based Architecture (CBA). CBA is discussed in the ZedaSoft white paper titled “A Container-Based Architecture for Simulation”.

Our charter was to re-architect and implement the software behind Lockheed-Martin’s human-in-the-loop, soft real-time cockpit demonstration flight simulators, which are the primary marketing tools for the F/A-22 program. We quickly realized that the CBA design would be a perfect foundation for this task.

CBA is an object-oriented design from the ground up, and we were challenged to select the best language platform with which to implement it. In the best possible case the selected platform would at least:

- Include comprehensive support for object-oriented programming;
- Support cross-platform software reuse at the object level;
- Support plug-in architectures;
- Be capable of soft real time performance at a 50 Hz or greater update rate;
- Support OpenGL graphics;
- Be widely used and supported;
- Promote stability, extensibility, and maintainability;
- Minimize the possibility of software development errors;
- Not require a wizard to use its best features.

We considered C++, Objective-C, C#, Eiffel, and Java. Objective-C – arguably the best technical choice of all the C derivatives – and Eiffel were dismissed due to lack of widespread adoption. C# was dismissed primarily due to lingering undesirable language features and lack of maturity on our platform, though in some respects it was an attractive potential solution. This left us having to choose between C++ and Java. In this paper we’ll take a balanced look at the technical and business issues that eventually led to Java as the chosen development platform for CBA.

Note that this paper doesn’t focus on hard (or even firm) real-time issues in Java, though we briefly discuss progress to date in this area in a later section. Instead, we discuss how Java provided us a better platform on which a mid-sized system was implemented in a short amount of time, and how its features contributed to our architectural desires for the system.

## **Myth Busters**

We'd like to challenge some conventional wisdom before diving into the details about why we ultimately abandoned C++.

### *Myth: Java Is A Web Programming Language*

When Java was officially introduced in 1995 Sun promoted it as a language for implementing client-side components that could execute in a web browser. Like many technologies, Java's first intended use ended up not being its best use. While it remains an excellent platform for supporting web applications, Java has since evolved into a rich general computing platform.

### *Myth: Java Is Slow*

Java compilers generate platform-independent byte codes that are executed by a Java Virtual Machine (JVM). This interpretation step is what allows pure Java classes to execute on multiple platforms without recompilation; however, byte code interpretation introduces inefficiencies into the execution process. Early versions of the JVM from Sun operated exclusively in interpreted mode, and were unacceptably slow. This situation has improved dramatically with the introduction of JVMs that support runtime optimization. These JVMs use various techniques to improve the performance of an application while it's executing. For example one technique, called "adaptive compilation", watches for areas of byte code that are frequently executed, and dynamically converts them to native code. The latest HotSpot JVM from Sun operates with runtime optimizations enabled by default, and can approach the performance of C++ for certain operations. However some operations – such as formatted I/O – can still be quite slow, and must be used with discretion.

### *Myth: Performance Isn't Everything; It's The Only Thing*

We've sacrificed much over the years in the quest for performance, and the cost isn't always measured in dollars. As recently as the mid-1980's computer hardware capable of supporting real-time simulation was specialized and costly. Processing power was relatively small by today's standards, and many in the community spent seemingly endless hours writing cryptic code and forgoing any semblance of an architecture in the pursuit of raw throughput.

Computer hardware has now matured to the point where COTS PCs are capable of supporting the throughput required for human-in-the-loop vehicle simulation, and sufficient capability exists to sacrifice a few CPU cycles in exchange for a balance between performance, development efficiency, maintainability, and having a decent platform on which to realize large-scale architectures.

Of course this isn't the first time we've borrowed from the CPU to make our lives easier. Software professionals who claim to write everything in C to achieve optimum performance ought to be reminded that if they really wanted performance they would be writing everything in assembly language. We don't do that because we're willing to sacrifice a little performance in exchange for a more expressive platform. This trend doesn't have to stop with C and C++.

## The Technical Details – C++ Challenges

Our team was experienced with both C++ and Java, so the pros and cons were thoroughly understood. C++ was established in the simulation community, but was at the time a 22-year-old language with many shortcomings that tended to work against code correctness, and that often introduced difficult-to-diagnose bugs. We also recognized that C++ had at least the following advantages over “stock” Java:

- Performance (how much better depends on what you’re doing);
- Predictability;
- Low-level machine access;
- Efficient formatted I/O;
- Direct C API integration;
- Acceptance in the simulation community.

We also recognized that C++ had a host of nuisance problems that worked against software quality, and we were not alone in making this assessment. For example, at a recent trade show a C++ developer described C++ applications as a “series of skid marks leading up to a train wreck”, and we couldn’t argue. The issues that contribute to that perception are at least mitigated, and in many cases eliminated, by Java:

### *Minimal Public Runtime System*

The public features of the C++ runtime system provide little beyond a cryptic name discovery capability, and some level of runtime type determination via the dynamic cast mechanism. There is no provision for discovering and interacting with the attributes of a class or object at runtime. You *can* dynamically load and create instances of subclasses of known superclasses via platform-dependent factory techniques, but can only interact with those instances via information known at compile time. In our experience even these capabilities have not been fully exploited in the simulation community, probably because the C++ support mechanisms are platform-dependent and esoteric, and architectures requiring those capabilities are infrequently used.

Java supports a rich reflection mechanism that provides runtime discovery of a class’s fields and methods, along with ways to interact with them. While a purist might argue that a well-designed application would have no need for such features, we disagree. There are situations where the ability to interact with totally unknown classes is useful. A real-world example is initialization of object graphs from XML textual descriptions, which is something we take advantage of in CBA to load simulation scenarios that may contain classes the core system knows nothing about.

### *No Intrinsic Memory-Management Policy*

C++ has no intrinsic mandatory philosophy for defining ownership of memory allocated from the heap. This usually results in one or more of the following problems:

- Confusion regarding if, when, and where in a program allocated memory should be released;
- Code with inconsistent pointer handling, which can be a maintenance nightmare;
- Dynamically allocated memory never being released, in which case the offending program “leaks” memory, i.e. virtual memory usage continues to grow until in the worst case the operating system can no longer perform;
- Dynamically allocated memory being released multiple times, in which case the program crashes.

The last problem is aggravated by the fact that the offending code snippets often reside in parts of the program that are far away from one another, so the apparent point of failure often isn’t at the location of the root cause of the problem.

C++ eventually introduced the “auto\_ptr” mechanism to help with this problem. auto\_ptr was an attempt to automatically delete dynamically allocated objects that were no longer referenced.

`auto_ptr` has a complicated history, undergoing at least two major revisions since its introduction. This gives us some indication of the challenges the C++ community faced trying to fit this fundamental notion into a language that had no such support already built in.

Reference-counting strategies can also be implemented in a variety of ways. The `ref_ptr` template approach used by Open Scene Graph is one such example, though its use is restricted to objects that also inherit from the OSG "Referenced" class.

Much work has been done on implementing automatic garbage collection for C and C++, but all of these approaches are add-ons, and are not an intrinsic part of the platform. In other words, good luck finding a large collection of C++ code that's using a consistent garbage collection strategy across the board.

Java solves the memory management problem by providing an automatic garbage-collection mechanism that's an intrinsic part of the platform, at expense of some performance and predictability. Automatic garbage collection is an old idea that has been used in many languages that predate Java. The mechanism frees developers from having to worry about if, when, and where allocated memory should be released. Java's garbage collection implementations are capable of detecting and removing "reference cycles", i.e. objects that reference each other directly or indirectly (this is a common downfall of reference-counting mechanisms, where such cycles result in objects that can never go away).

However, all is not perfect even with a garbage collection mechanism. Memory leaks are easy to generate, and the garbage collection mechanism is not directly controllable and contributes to unpredictable behavior. For simulation applications developers must remain aware at all times of how the code they're writing is going to interact with the garbage collection system. Furthermore, the points at which garbage objects are collected are determined by the runtime system. API calls exist to suggest when garbage collection should take place, but strictly there's no guarantee that the suggestion will be followed.

All things considered, it's still an easy choice to go with automatic garbage collection. It frees the developer from having to constantly think about who owns a piece of memory, and turns a potential error condition that's typically hard to diagnose into, at worst, a potential performance issue that's easy to diagnose.

### *No Formal Interface Support*

Interfaces are API declarations without an implementation. Code written in terms of interfaces can be used with instances of any classes that implement the interface, so interfaces play a major role in the ability of a system to deal with objects without having to care about their specific class.

C++ mimics interfaces through classes containing pure virtual function declarations, and these informal interfaces are integrated into subclasses via the multiple inheritance mechanism. This isn't necessarily a bad solution, other than the multiple inheritance mechanism forces developers to think about the multiple virtual base class issue.

Java simplifies the problem by introducing the formal notion of an interface, and permitting any class to implement zero or more interfaces without the need for multiple inheritance.

### *Pointers And References*

Pointers are a powerful mechanism, and all types of tricks can be done with them in C and C++. They are a necessary evil, but also a common source of mistakes. Developers often forget to allocate memory for a pointer, or remember to do it but don't do it correctly. Sometimes pointer values are left uninitialized on the stack and the program either crashes randomly, or in the worst case seems to work only to crash later.

C++ references are closely related to pointers. References provide a way to create aliases for other variables. References can be used to pass arguments to functions by reference instead of by value, and otherwise to reduce the use of class and struct pointer dereferencing syntax. While

this seems to be good, in conjunction with pointers we now have two ways to achieve the same result. Developers don't use these facilities consistently, and this can lead to clumsy APIs (though this problem could be alleviated by adherence to strict development guidelines along with code reviews to enforce them).

Java simplifies the situation by not giving developers access to pointers, and always using call-by-value semantics for methods. While this can seem an inconvenience, APIs are easier to understand since there's only one way to do it.

### *Unchecked Array Access*

Arrays in C and C++ are simple contiguous sequences of memory that can be indexed by a zero-relative offset. Access to arrays is unchecked in absence of special tools and/or compilation options, so it's easy to under-index or over-index an array. This may manifest itself as some type of fatal error, or more insidiously, as unusual behavior.

These errors often occur because arrays have no built-in properties such as their length, and/or information about the type of objects they contain.

The unchecked array access problem can be avoided in C++ through the use of STL vectors, but developers tend to not use them because declaring and using an array is much easier, and C APIs that need multi-valued variables frequently use pointers. Some open source library developers have also defined structures similar to STL vectors to help with the array indexing issue.

Java solves the unchecked array access problem by making arrays first-class objects that know their length, know what kind of objects they contain, and that check every indexed access. Since arrays carry their length with them, array lengths can be determined even in arrays that are passed into methods, and errors in array indexing are caught at the point of occurrence.

### *Random Initial Field Values*

Initial field values in instances are not guaranteed in C++, so if you don't remember to include initializers for all member variables they're probably not going to contain the values you intend. This is another situation where the developer may "get lucky" and the application will seem to work, only to fail later.

In Java all field variables are guaranteed to have specified initial values, even if not explicitly initialized.

### *Header Files*

To a C or C++ developer life without header files seems unimaginable, but what header files are really bringing to the table from a developer's perspective is an increased code maintenance workload since you're responsible for maintaining API consistency in both the interface and implementation. Header files are a necessary evil in the world of C and C++ as a place to declare the interfaces to library code, and to serve as the compiler's interface into pre-built libraries.

Java does away with the header files; the only source file is the class implementation. The compiler figures out what it needs to know about an API by examining built classes, and interface documentation is provided in the form of HTML pages generated from the source code by the javadoc utility.

## *Namespace Facilities Added As An Afterthought*

In C and C++ external symbols from different library vendors can “collide” when two libraries being linked into the same application define symbols with identical names. The concept of namespaces was eventually introduced into C++ to reduce the possibility of this happening. A namespace provides a (possibly) nested naming context for classes. The full name of the class ends up being the outermost to innermost namespace names all strung together, followed by the simple class name as the final component, with some kind of separator in between.

C++ namespaces are a problem because they are *not* used as often as they should be. They weren’t introduced until later in the evolution of the language, so the development community never got in the habit of using them. When they finally were introduced it was a while before most compilers supported the feature, so library vendors who needed to support diverse platforms declined to use the mechanism, and simply added a short prefix onto their class names. Fortunately this situation is improving, in large part due to the open source community who tends to trail along behind GNU’s g++ compiler (which has namespace support).

Namespaces are an intrinsic part of Java in the form of packages, and every distributed Java class library we’ve encountered has made use of them according to Sun’s guidelines.

## *Optional Virtual Methods And Virtual Inheritance*

In C++ methods are only virtual if you declare them as virtual. We have to guess that this is a leftover 1980’s concession to performance. While it probably was a worthwhile optimization at the time, and certainly can still be used to pare off some additional execution time, it’s just another thing for developers to have to think about, and probably leads to surprising behavior as often than it provides a true benefit (at one time the GNU g++ compiler included an option to make all method calls virtual, so somebody out there agreed).

Similarly, classes may or may not inherit “virtually” from a superclass. This mechanism was introduced to handle the possibility – introduced by multiple inheritance – of a class inheriting more than once from a given base class. This is yet another thing for developers to have to think about, and can affect one’s ability to update production class libraries.

In Java the developer has no decision to make because all accessible instance methods are implicitly virtual, and the absence of multiple inheritance eliminates the need for virtual inheritance.

## *No Mandatory Exception Handling Policy*

The C++ exception mechanism has some unfortunate features:

- A class is not obligated to declare that it might throw an exception;
- You are therefore free to ignore any exception, and leave it to be handled by something higher up the call stack;
- There’s no mandatory exception class hierarchy; you can throw and catch anything, which makes it difficult for the compiler to do any checking.

These features all contribute to a loose approach to exception handling that can lead to surprises.

Java introduces the concepts of checked and unchecked exceptions. The compiler enforces a catch-or-declare policy for checked exceptions. This policy says that that if any method you call declares that it throws a checked exception, you must either catch the exception and handle it locally, or declare that you throw the exception. This is a much tighter mechanism where it’s always clear where exceptions may occur in a given section of code. Note that the catch-or-declare policy isn’t enforced for unchecked exceptions, for good reasons that are discussed in the Java Language Specification (available at the time of this writing at <http://java.sun.com/docs/books/jls/index.html>).

### *No Fail-Fast Philosophy*

In C and C++ it's common for crashes to occur at some point in the software otherwise unrelated to the error's true origin. You would like the system to fail as soon as the problem happens, and not wait until later. This philosophy is called "fail fast", and it's a big benefit to software developers because the location of errors is clear.

Java supports the fail-fast philosophy by raising exceptions as soon as erroneous conditions are detected, and eliminating features of C++ – such as unchecked pointer indirection and array accesses – that are often the cause of problems.

### *Relatively Small Standard Library Support*

Today's C++ developers can make use of the Standard Template Library, which provides classes that support common data structures such as lists, hash maps, as well as other utility functions. While this is a good thing, the number of classes supported by the STL is relatively small. Furthermore, much of the STL is implemented using the C++ template mechanism, which can be challenging to understand. Even conceptually simple operations, such as getting an iterator for the elements of a collection, can be syntactically messy (on the positive side, the template mechanism supports compile-time type checking, and eliminates the need for downcasting that's prevalent in Java).

The functional voids in the C++ STL have been filled to some degree by open source libraries such as Open Scene Graph, Xerces, ZThreads, and wxWindows, to name just a few, and most of these provide some level of cross-platform support.

The Java SDK provides access to a rich set of utility classes that include everything the STL supports plus much more, and the interfaces to these classes are simpler to use than some of the STL API.

### *No Guarantee Of Library Compatibility*

C++ libraries for a given architecture aren't guaranteed to be compatible. Compatibility instead is defined by the compiler and linker used to build the native library. For example, object libraries built with Visual C++ aren't compatible with object libraries built with g++ due to differences in name mangling and calling conventions.

Pure Java classes are in most cases able to run on any platform with a compatible Java Virtual Machine.

### *No Guarantee Of Type Sizes*

C++ inherits C's loose specification of the sizes of various primitive types. For example, the size of a short is only guaranteed to not be longer than an int, and the size of a long is only guaranteed to not be shorter than an int. This can lead to challenges when porting code from one platform/architecture to another.

Well written C and C++ software frameworks hide this problem from the developer via typedefs. However, typedefs are a nuisance because when things go wrong (and they will) you often end up bouncing around between header files trying to discover the true type of a field via a chain of typedefs.

Java guarantees the sizes of all primitive types across all platforms. This is one of the features that supports Java's "write once run anywhere" potential.

## *Platform-Dependent Dynamic Loading/Linking*

In C and C++ dynamic library building and usage is platform dependent.

In Java dynamic class loading is a core mechanism of the runtime system. Classes are simply referenced by name, and the system takes care of locating and loading them in a platform-independent way. Classes also can be loaded from remote network sources, such as a web server, and if written in pure Java, can run on any platform with a compatible Java Virtual Machine.

## *General Platform Dependency*

Porting C and C++ code from one platform to another can be a chore depending on the complexity of the software. Header files may be named differently and/or be in different locations, APIs may be partially supported or not supported at all, and the operation of the compiler, linker, and other tools needed to complete the build may differ. The open source community has addressed these issues by developing tools such as “autoconf” that interrogate the target system and determine what it can and cannot support. Also, the impact of platform-dependent functions can be minimized by collecting those functions in middleware frameworks. While this is an added level of effort for C++, its long-term benefits cannot be over emphasized.

In most cases Java eliminates these problems. The classes in the Java SDK already encapsulate the most common platform-dependent operations. Header files don't exist at all, pure Java classes will almost always run on other platforms without the need for modification, and the compiler works nearly identically on all platforms. Finally, build tools such as “ant” take care of most platform dependencies that affect the build process (though “ant” introduces its own set of minor quirks).

Note that if your Java components are associated with native code you wrote yourself you will still need to handle this using the usual tools, and therefore have the usual problems.

## *Deployments Are Susceptible To The “Fragile Base Class” Problem*

The “Fragile Base Class” problem is a side effect of compiled C++ classes being tightly coupled to class sizes and the offsets of the fields in each class. Field offsets are carried forward into subclasses, which means if the size of a superclass changes, or accessible fields are rearranged thereby changing their offsets, then subclasses will no longer function correctly. A more insidious manifestation of this problem is code that doesn't crash but instead “sort of” works. The practical effect of this on software development is that any time a C++ superclass is modified in one of these ways you'll need to rebuild everything that inherits from that class. The practical effect of this on deployment is that when deploying libraries you have to take special care that binary compatibility is maintained between classes.

Java goes a long way towards eliminating this problem. Subclasses are no longer sensitive to superclass sizes and field locations. Fields are instead located dynamically, and field rearrangements no longer affect subclasses.

## *Inline Functions Work Against Maintenance Of Deployed Software*

C++ inline functions cause function code to be emitted in line at the point of call, and can be a useful optimization for some applications. However, if such functions are public they can work against the ability to update deployed software because they require every use of the inline function to be recompiled if the function is changed.

Java doesn't let the developer specify when a function should be inlined. This optimization is instead deferred until run-time.

## *Core Dumps*

C and C++ provide a variety of ways for programs to fail. Access violations, segmentation faults, and bus errors often result in a core dump, and are just part of the daily routine.

Java can core dump too, but the situations are rare, and usually involve native code that's been linked into the system. If you stay away from native code and keep everything pure Java, you may never see another core dump.

## *Stack Traces*

Walking the call stack with C and C++ is a cryptic and platform-dependent exercise that usually involves a debugger. The call stack trace – assuming you can get one with symbols – can be difficult to interpret, and its interpretation by the debugger can be unreliable.

In Java you can generate an easily interpreted call stack trace at any point in a program by inserting a single statement. Furthermore, the call stack itself is an object that can be interpreted via some standard Java classes, so special stack trace dumps can be implemented by the developer.

## **The Technical Details – Java Challenges**

Java addresses most of the nuisance problems of C++, but in exchange presents at least the following challenges that threaten its ability to meet the demands of real-time simulation:

- Java performance is not generally as good as C-based languages (generally around 2x to 3x slower, though some operations such as formatted I/O can be up to 10x slower);
- Java introduces predictability issues due to processes that can't be directly controlled, such as the garbage collection mechanism;
- Java doesn't support low-level machine access;
- C APIs can only be integrated through the somewhat clumsy Java Native Interface (JNI);
- The idea of using Java is a stretch for the simulation community.

## *Performance*

As we previously discussed, Java performance has dramatically improved in recent years due to the introduction of JVMs that perform runtime optimization. Our experience has been that observation of common-sense programming guidelines can usually keep Java executing within 2x to 3x of C++, and work continues on ways to improve this further.

However, operations such as formatted I/O can be up to 10x slower than what you can get out of languages such as C. This is a nuisance for software supporting displays of textual data. The C and C++ "sprint" mechanism, while slightly dangerous due to array overflow potential and varargs errors, is flexible and fast. Java can't provide a similar mechanism primarily because there's no varargs support. We solved the formatted I/O problem by implementing a simple and efficient formatting mechanism that handled the majority of our data formatting needs.

## *Predictability*

Java's garbage-collection mechanism, while one of its most useful features from a code correctness perspective, introduces an element of execution unpredictability. The JVM from Sun supports several garbage collection implementations, and the behavior of each of them can be tuned to some degree. However, all of the implementations occasionally go through "stop all threads" pauses, which means your user code is stopped while the garbage collector sweeps up reclaimable objects. It's easy to see how this would adversely affect a simulation that's trying to operate on a fixed time step.

To counter the unpredictability of the garbage collector we observed the following simple programming guideline: don't give it too much to do. We allocated as many objects as possible prior to starting the real-time run loop, and worked around routine memory allocation that would have contributed to the pool of reclaimable objects. However, we still freely used the allocator to support sporadic events, such as launching a munition. These types of operations happened so infrequently that they didn't contribute significantly to heap usage.

A related nuisance "feature" of Java is the inability to get stack-based objects. In C++ you can either dynamically allocate objects, or simply declare them as stack variables. Obviously this approach avoids the heap entirely. In Java there's no such facility; all objects must be dynamically created, including arrays since they are also instances of some class. This means that the creation of temporary local objects will contribute to the cleanup task of the garbage collector. As a point of interest the C# language – which is notionally similar to Java in many respects – *does* provide a way to get stack-based arrays in unsafe contexts, but in doing so reintroduces pointers.

### *C API Integration*

Java integrates with "native" components via a bridge mechanism called the Java Native Interface, or JNI. The Java Virtual Machine is itself a C program, and the JNI simply exposes an interface into the JVM so methods in Java classes can be implemented in C where necessary.

Interacting with the JNI is clumsy, and brings back all the disadvantages of C that we would like to avoid. It's therefore important to minimize, or in the best case eliminate, usage of native code.

Unfortunately OpenGL graphics integration – one of our greatest risk areas – was tightly coupled to this problem. Since a key product of most simulations are its graphical presentations (and in our case it was the only product), it was important to have access to an efficient graphics API. We ultimately had to turn to the JNI to provide a reasonable interface to OpenGL, but not in the way you might expect. This solution will be covered in another ZedaSoft white paper in the near future.

Generally speaking, you will always require some native code for supporting real-time simulation. For example, there's always some low-level device to interface with, and there are often requirements for simulations to 'integrate' with legacy government-furnished software that is typically written in Fortran or C, requiring additional JNI support and integration. This adds a layer of complexity not present when developing with C or C++.

### *No Low-Level Machine Access*

Java is designed to be platform-independent plus it doesn't support pointers, so it makes sense that it doesn't support low-level machine access. However, all simulators with hardware devices attached eventually have to do this. The only solutions with "stock" Java are to either implement the functions requiring low-level access in native code and bridge to them using the JNI, or implement them in native code processes and communicate with them over the network.

### *Simulation Community Acceptance*

Java's last major issue is that of acceptance within the simulation community, and we would argue that this is the greatest challenge of all. Resistance to platform changes comes in practical and cultural forms.

It's probably safe to say that today's simulation community bases most of its work on C and C++, and to a lesser degree, Fortran and Ada. A tremendous body of code has been built up over the years, and in many cases it's impractical to consider a platform switch due to schedule and/or budget concerns. Of course one has to wonder how long we're going to keep using this rationale to cling to legacy platforms. There's a school of thought that says the community should retire the old platforms and start down a new path, pulling older solutions in through bridge mechanisms until they can be replaced.

There's also a cultural influence at work that can make a platform switch seem undesirable. The informal pecking order among software developers has always been defined by the amount of esoteric knowledge one has about a platform. Whether or not we want to admit it, nobody likes to move away from platforms on which they are experts to platforms on which they know little or nothing; in other words, few software developers like to go back to grade school where they're no longer the experts. Fortunately the designers of Java elected to base much of the language on C++. The result is a platform that fixes the problems of C++, yet is similar enough to provide an easy transition by experienced developers.

After overcoming these issues we'll have only the handful of remaining technical challenges to solve to make Java a robust general development platform for real-time simulation.

## **Real-Time Java**

"Real-time Java" is not an oxymoron. There are at least two significant efforts underway to build a foundation for real-time Java, and these efforts have done a lot of work towards eliminating the shortcomings mentioned above.

The "Real-Time for Java Expert Group" was chartered under the Java Community Process to develop a real-time specification for Java. The resulting "Real-Time Specification for Java" (RTSJ) documentation can be downloaded at [www.rtsj.org](http://www.rtsj.org). At least one commercial implementation of the RTSJ has been released by TimeSys, who also provided the RTSJ reference implementation. Their product is called JTime, and more information on it is available at <http://www.timesys.com>.

The other significant effort is from the J Consortium (<http://www.j-consortium.org>). This effort has roots going back to NIST initiatives started in 1998.

Both of these efforts deserve to be closely watched, and you should bother to visit the web sites noted above if you're at all interested in using Java for real-time systems development.

It's unfortunate that the effort to define a specification for real-time Java already has a major schism running down its center. Hopefully these will find a way to come together and result in a solid platform for hard real-time Java software development.

## **Business Issues**

Most software development groups are part of some business, and the folks running the business are worried about a lot more than the favorite development platform of the day. They want something that allows the business to compete, that reduces cost, and that makes it easy to find staff on the correct side of the supply-and-demand curve.

### ***Cost***

The Java SDK from Sun is freely downloadable, including source code for most of the software development kit packages. Note that the GNU C++ compiler is also freely available, so this by itself isn't an advantage of Java (but the price is still right).

### ***Open Source Availability***

As a side effect of Java's popularity in the business world there are many organizations contributing to a growing pool of open-source libraries and tools. These often can be freely used, and source code is usually provided. C++ has plenty of open-source resources too, but we doubt they are as plentiful, or growing in number as fast.

### ***Staff Availability***

There are a lot of competent Java developers on the market, many of them fallout from dot-com failures and increasing use of overseas technical labor in the business world.

## *No Wizards Required*

Due to its C heritage, C++ is capable of some wizardly things. To see just how exciting things can get you should make a point to look at some of the winners of the International Obfuscated C Code Contest at [www.ioccc.org](http://www.ioccc.org). While this type of thing is interesting and fun, it has no place in the development of large software systems. However, tempt the potential wizards with powerful features and more often than not they'll find a way to use them in convoluted ways.

Most of Java's features are understood by typical developers, though Java is not without its own cryptic syntax (see anonymous inner classes for an example).

## Conclusions

It was an easy decision to use Java based on language features alone, once we concluded our overall performance requirements could be met. We would much rather have greatly improved overall language support and just a handful of challenges to overcome than the reverse situation with C++. The price is paid in performance and predictability, but careful coding and increasingly capable PC hardware can help us around these issues. This is an easy tradeoff to make for a stable, flexible, and maintainable software base.

Java's software development efficiencies compared to C++ enabled a team of three experienced developers to complete the initial internal release of CBA, and the first production entity deployment for F/A-22 program, in nine months from scratch. This software product consisted of over 150K significant lines of source code distributed among over 1100 classes. The final bug list averaged about one bug per 500 lines of code, and nearly all of those problems were easily fixed mechanization issues rather than problems caused by the language platform being a little too loose. Some of this productivity is a result of an experienced development staff, but we credit most of it to the features and stability provided by the Java platform.

It's interesting to note that despite the performance issues noted in this paper our project was successfully completed on dual 1-GHz PCs, which are low-end by today's standards. We intentionally chose this platform and stayed on it even when offered faster machines so throughput issues would reveal themselves early. In other words, we wanted to stay at the less capable end of the platform spectrum so we had nowhere to go but up.

Most of Java's remaining technical hurdles are addressed by the Real-Time Specification for Java and/or the efforts of the J-Consortium. Implementations of the RTSJ are just now becoming available, and it probably will be several years before we see RTSJ implementations with performance comparable to that achieved by Sun's HotSpot JVM. This timeline would be accelerated if companies with more substantial resources were to get involved with implementing RTSJ-compliant products.

The idea of using Java for soft real-time simulation remains controversial, though we believe our success with it proves we're closer than many may think. At the time of this writing (January 2004) we find Java more than sufficient for exercises with modest participant population and complexity. However, it probably will be several years before we can expect a Java simulation to support the multi-thousand entity exercises that are possible today with C and C++. There are also specific corners of the simulation problem domain, such as out-the-window image generation, that require a level of performance that Java probably won't be able to reliably support for a few more years (though recent Java3D demonstrations show that it's getting close).

If the Java platform were accepted and momentum were to build, the remaining technical issues would vanish quickly because tool vendors would respond to increased demand for features and improvements from the development community. Given our overwhelming positive experience with Java in this problem domain, we can't wait to see it happen.