# Writing Java Games: How We Did It

Scott Williams
Jim Van Verth
Garner Halloran
John O'Brien

Red Storm Entertainment
2000 Aerial Center, Suite 110
Morrisville, NC 27560
{scottw,jimvv,garnerh,johno}@redstorm.com

## 1 Introduction

There has been a lot of discussion about Java in the gaming community, particularly whether Java is suitable for writing games or whether C/C++ will continue to be the language of choice. We don't care to take sides in this debate – we believe that C++ will continue to be the dominant development language for some time, but that Java also has its place in game development. Instead, the purpose of this paper is two-fold. One is to introduce developers to some of the issues involved in using Java in creating games. The second is to present some specific challenges we had in developing Tom Clancy's Politika, how we handled them, and how future developments in Java technology and our hindsight will deal with those problems in our next Java-based product.

### 1.1 What is Politika?

Politika has sparked some interest in regards to its connection with Tom Clancy, but also as the first large-scale, commercial Java game. The game itself began as a simple concept: a number of us were long-time Diplomacy™ fans and wanted to create a similar experience in an online environment. There are mail servers that permit that – but we wanted to play in three hours as opposed to three months. Based on that central idea, we worked closely with Tom Clancy and designed a board game that we thought would translate well – a game set in Russia, where the sudden death of Boris Yeltsin has left a power vacuum, and eight factions rise up to try and take control. The layout would be 2-D, where the primary activities are moving pieces, talking and negotiating with other people. Because of this, the need for processing power would be relatively low.

Once the design period was done, the question of implementation came up. Should we tie ourselves down to one platform and write it in C++ with DirectX, or should we try writing a cross-platform game in the new language Java?

### 1.2 Why Java?

The turn-based strategy game we planned wouldn't need to be technology driven like our other projects. Politika could be implemented with 2-D board game-style graphics and withstand long Internet latencies. Meanwhile, Java "just in time" (JIT) compilers had just dramatically improved bytecode performance. Ultimately, the business decision for Java was made for the sake of an easy Macintosh port (after all, Tom Clancy is a proud Mac driver). The limited market share of the Mac platform would be offset by strong interest from game-starved Mac owners. The promise of *write once - run anywhere* meant it would be virtually free.

Still, this was something of a radical approach. We spent time looking through the sites like Gamelan (www.gamelan.com), university web pages and Java-tool companies' sites for high performance applets to convince ourselves the necessary speed was there. We felt it was. Java would get the job done and enable some interesting avenues, like browser Applets to advertise and demo the game.

As an additional bonus, Java as a language is logically designed and nearly identical to C++ at the method level. (O'Reilly's "Java In A Nutshell", first edition, was already our constant companion by this point.) Sun makes the Java source to the core packages available with the development kit, a fact much appreciated as we got down to writing code.

## 2 Tools

### 2.1 Version 1.0.2 vs. 1.1

The first question involving Java came early and was unresolved for some time: go with version 1.0.2, which was already out and established with a selection of compilers and virtual machines, or with 1.1, which at the beginning of our development cycle was in beta on Windows 95 and was non-existent on the Macintosh. Our colleagues at IBM (see InVerse, below) urged us to go with version 1.1, because 1.0.2 was bug-ridden – in particular there were problems with the I/O and networking code that were fixed in 1.1. We held off this decision as long as we could; mainly we were waiting to see how the Macintosh Java situation would shake out, as the premise for doing Java at all was the Mac version. Eventually it became clear that a Mac 1.1 just-in-time implementation would not be out by our ship date, so the final decision was made to stick with 1.0.2 and work around any bugs we might encounter.

### 2.2 Virtual Machine

Related to the 1.0.2 vs. 1.1 question was which virtual machine we should use on each platform. Under Windows 95, the two major implementations were by Sun and Microsoft. We went with Microsoft's VM for three main reasons: 1) as the OS vendor they would have a better idea of how to implement an efficient VM than an outside company; 2) they provided some hooks into the operating system, which would be useful to us; 3) they provided support for creating an "executable" which wrapped around the classes. We did not anticipate becoming caught in a war between Sun and Microsoft, which would hurt us later when Internet Explorer 4.0 came out.

On the Macintosh, the two main choices were between Apple and Metrowerks. The Sun implementation was clearly incomplete – in particular there were problems in reading GIF files, which we needed. The Metrowerks environment was appealing, but in the end we went with the Apple VM for all of the same reasons stated above.

## 2.3 Compiler

We tried a number of compilers – Natural Intelligence's Roaster, the Sun javac compiler and Metrowerks, but the final decision was made to use jvc and Microsoft Visual J++. The primary reason was familiarity – we had been using Visual C++ as our C++ compiler, were familiar with the Microsoft Developer Studio IDE, and liked its easy access to the SourceSafe database (although, due to Java's strict source-file naming conventions, i.e. *Classname.java*, we had to upgrade to SourceSafe 5.0 for long filename support). We also were unable to determine any major difference in quality between the code produced by any of the compilers. The speed of compilation by Visual J++ was also another factor in its favor.

Because of Java's portability, we were able to use the Visual J++ compiler on both platforms, by compiling under Windows 95 and then moving the classes over to the Mac. The only key was copying the classes that supported the Macintosh native interface over to the PC, and then ensuring that those classes were in the classpath for the compiler.

One caveat though – while we have no evidence one way or another, we did have a number of problems which may have been due to the J++ compiler. In particular, the problems we had with class file size may be related to both the compiler and the VM we used. For our next project, we will be using the Symantec Visual Café development system, which solves both the VM and compiler questions, by compiling directly to an executable. We believe this solution will provide a more robust environment for Java code than depending on stability during the continual VM wars.

## 2.4 InVerse

Early in the year, Gennaro Cuomo of IBM contacted Doug Littlejohns, Red Storm CEO, about *Interactive Universe* or *InVerse,* a collaboration-oriented networking package recently developed by an IBM group in nearby Research Triangle Park. In addition to a simple API for linking online participants, the system featured robustness and performance evolved with published research in servlet-based transaction processing and media networking. Gerry's team felt that a multiplayer Internet game might be the ideal application for InVerse; the Politika team took immediate interest when we found that it was written in pure Java. After the requisite forms were signed (and we could actually talk with their engineers) it was clear InVerse could do the job and save us a lot of time. IBM would write sample lobby server and client applications to demonstrate the toolkit. Although Inverse didn't use any Java 1.1- specific calls, it relied on some of the bug fixes in the 1.1 release. Fortunately, IBM was able to handle the problems on the InVerse side and keep our Mac version on-track.

### 2.5 Sybase

Politika was to be innovative in some non-engineering ways as well. During the first half of the project, we planned to emphasize the multiplayer game (indeed, there was no plan for single-player mode at all). We would sell Politika on-line; people would download the product and play free-of-charge for a limited time then switch to a very modest monthly subscription. In order to realize this revenue model, our lobby server would need a database back-end to support secure login/logout capability. The database could also be used for registration and personal data aspect (interests, age, profession, etc) to enrich the desired sense of community. Compared to hard-core database application, this was not a huge amount of data, but mustn't be a bottleneck either. We went with the JDBC (*Java DataBase Connectivity)* package included in Java 1.1. Sybase's SQL Server 11.0 and JDBC drivers were the right combination of reputation, compatibility (web-server) and cost (vs. Oracle and Informix).

### 2.6 Installer, Crema, jexegen

The installer for both platforms would be a bit more complex than average in that it would handle a separate Java Virtual Machine installation as well as native enhancements such as ActiveMovie and DirectSound. Also, Java bytecode retains a great deal of symbolic information; bytecode disassemblers can produce disturbingly accurate Java code from the corresponding .class file. Before the 'executable' was ready to distribute, it had to be obfuscated to counter this vulnerability. *Crema*, shareware by the late Hanpeter Van Vliet, performed this process. (Crema is now part of Borland's Jbuilder package). Finally, for the Win95 version, Microsoft's utility JEXEGEN bound up the class files and a JVM launcher into an .exe file.

One note on Crema: our tests show that the Apple MRJ 2.0 doesn’t run its fully obfuscated files very well, while Microsoft Java VM 2.0 does (both are Java version 1.1). This may be due a liberal attitude by Microsoft to bytecode standards – however, caveat emptor.

## 3 Known Challenges

### 3.1 Limited Graphics Performance

The Java platform's media capabilities were (and remain) limited – graphic performance was adequate. The basic board game presentation was definitely within reach, but animation would have to be used sparingly. Java's Image class design didn't help matters any. The Image Producer/Consumer scheme was confusing at first, and more importantly, direct access to the image data was not allowed in any practical way (improvements to MemoryImageSource came in 1.1). Fortunately, GIF89a transparency support was included. The animated GIFs worked as well, but they were difficult to control since drawing a cel automatically advances to the next one (i.e. no random access to the cels).

We also wanted the flexibility to add non-rectangular screen components, e.g. oval buttons, but there was no way to give the basic 1.0.2 AWT classes a transparent background (Java 1.1 includes "Lightweight Components" that are not bound to a native window and can be transparent). We also held the belief that the game should look identical across platforms – a goal that the AWT does not quite allow. These issues prompted us to develop our own image-based AWT-like package including buttons, sliders, menus, text components and custom widgets.

### 3.2 Poor Audio, No Video

Until Java 1.2 comes along, audio generation in the pure-Java environment remains limited to 8-bit, 8k .au files. To make matters worse, the audio classes reside in the applet package. Scantily documented (sun.audio) workarounds had to be employed to produce any sound at all. Clearly, this would be inadequate for a major game release -- the writing was on the wall for native-code augmentation. Since Politika's back-story is taken from a possible real-world scenario, we also planned to include video sequences of staged newscasts. Sun's roadmap for Java includes support for video display, accelerated 2D graphics and CD-quality sound, but few, if any, Java-based options for these existed. Notions of a development-free Mac version were slipping away...

### 3.3 Slow File I/O

Other potential problem areas we learned about in the newsgroups: inter-platform AWT (Abstract Windowing Toolkit) inconsistencies and poor disk I/O performance. Because we were developing our own screen components, our main reliance on the AWT was basic image and text support and we avoided *most* of these problems. Our game save/load did suffer some performance penalty (vs. the Java 1.1 version).

## 4 Surprises

### 4.1 32-bit Images

Our memory goal for Politika was 16 Megabytes of RAM, which seemed reasonable. We were using 8-bit images, had no large data structures, and a medium amount of code. In theory, we should have been able to run in 8 Megs, but were constrained by the size of the VM. In practice, we were very wrong. About halfway into the project, our QA person's 16 Megabyte test machines began running sluggishly and thrashing heavily. System Monitor told us that Politika was using 30 Megabytes of RAM. By removing portions of code (Java has no memory measurement facility like sizeof, so this was the only way we could do it), we determined that the lion's share was in the images.

We've already discussed the awkward image scheme. In addition to that, the Java Image object stores the image data in whatever format you specify: 8-bit, 16-bit, 32-bit if you like. However, when you finally want to draw that image, it has to create a native version of that image, called an image representation. As far as we can tell, that native version is a 32-bit image. So, when you use an image, you not only have two copies of it in memory, but one is potentially four times bigger than what you want.

We solved this problem in the following ways: first we devised an image caching scheme. When you need to draw an image, and the native image representation is not in memory, the virtual machine loads it automatically from the corresponding Java ImageProducer (the Java half). We extended ImageProducer and created a class which stores the image data in compressed format, thereby saving some memory and avoiding any I/O hit on reload. The native representation of images were then purged periodically depending on the last time they were used or drawn.

We also made changes to the art to minimize the number of images. Standard dialogs which had been a single image were broken down into border, button and fill elements. Animation for the zoomed out map had to be dropped for lack of image space. Finally, we added a low-memory mode which disabled the large zoomed in map and saved 5 Megabytes of RAM.

One solution that we tried, but was not completed in time for Politika, was implementing DirectDraw versions of the standard Graphics and Image classes (we tried using the Microsoft DirectX hooks, but they were not functional enough for our purposes in version 1.5). Preliminary tests showed that this improved graphics performance considerably, and reduced the memory footprint – however, there was not enough time to finish the implementation and test it before shipping. Our next product will have DirectDraw support which should solve many of the 32-bit image problems.

Lack of palette control also hurt us when running under 256 color mode. We had no control over what palette was used, despite the color tables in our images. The best we could do is to tune our images to fit whatever the VM chose for us, and hope that in the end it would turn out all right. After a day or two of image tuning, in 256 colors it might look fine, only to seriously degrade a few weeks later due to who knows what. This constant problem was another great annoyance to us.

## 4.2 No Good Debugger or Profiler

With all the hype and enthusiasm surrounding Java during this time, the lack of some basic developer tools was an unwelcome surprise. In retrospect, it should not have been; all the articles describing Java's bright *future* distracted us from Java's more humble *present*. Simple multithreading was often too much for VJ++'s debugger (tracking down bugs with print statements doesn't have the charm it used to). Profiling tools for Java are even now just starting to appear. (Intel's VTune 2.6 is one of three commercial Java profilers.)

## 4.3 AWT Side-effects

Another surprise was the effect of using certain AWT classes. The engineer working on file loading and saving had serious problems in getting his routines working properly: when he wrote and saved files directly, he had no problems, but when he used an AWT dialog to input the file names, some memory corruption occurred, the board was not loaded correctly, and there was the occasional crash afterwards. Our solution was to not use AWT dialogs, but to create our own.

### 4.4 Java class file limitations

The last was extremely annoying and pervasive. We kept having problems where the program would run fine for many iterations, but would unexpectedly crash in the virtual machine and hang the entire system. Without a debugger we could not determine exactly where the crash was occurring, but it did happen at certain times of high activity in the game, and as best as we could tell, this was due to class files being too large. Our belief is that the VM would try and load a new class into memory, not have enough room due to memory fragmentation, and then try to remove existing classes from memory. Occasionally the VM would remove a class that you might need, try to jump to that code, and then crash. Our solution was to break classes up into smaller classes, and then ensure that classes that we needed were still in memory by giving them busy work. This solved most of the problems – the last remaining one was fixed by rewriting the suspect code in a different way. This may only be a bug in the just-in-time compiler with the Microsoft VM – because of its non-repeatability we unable to determine if also occurred in the Sun VM under Windows 95, but it never happened on the Macintosh.

## 5 Cross-platform Issues

In addition to the problems we had with Java itself, we also ran into problems with cross-platform development, even with the promise of “write once, run anywhere.” The main problem was differences of implementation across platforms, but there were others.

### 5.1 Windows 3.1, Windows NT and OS/2

Politika was released only for Windows 95 and Mac OS even though it was developed in Java. The decision to limit Politika to Win95 and the Mac was made mainly to limit testing complexity, but we also looked at Politika on Windows NT, OS/2 and Windows 3.1. At the time of our tests, the OS/2 virtual machine behaved differently than the Win95 and Mac versions. Due to testing time restrictions, at the first sign of trouble we put it aside and OS/2 was no longer a possibility. The only Windows 3.1 JVM we found was a relatively obscure IBM product, and performance was unacceptably poor. Even Windows NT 4.0 exhibited graphics glitches – however we did manage to create a non-official version.

### 5.2 Mac network bug, font metrics, deployment schedule

PC-PC, PC-Mac, and Mac-Mac games were played. These two virtual machines were by no means identical. A bug in Apple's beta *java.net* package sidetracked Mac multiplayer testing for some time but fortunately was fixed by ship. On the positive side, visual inconsistencies were limited to minor errors in reported font metrics. Perhaps the most profound difference between them was also the most obvious: the schedules under which they were built. The Win95 Java 1.1 was in beta 6 or 7 months ahead of Apple's. Using Java 1.1 for the project might have saved time by preventing 1.0.2 network bugs. (Outside schedule uncertainty is a problem in itself. See the Java Media Framework, below.)

## 5.3 Native methods

In general, Java gave us all the pieces we needed for the game – a good set of libraries, 2D graphics code, some limited sound capability. However, as mentioned it did not provide a way to play video and music. The Java Media Framework library, which played .wav and .mpeg files was in beta at the time, but did not exist on the Macintosh, and did not go final until after our ship date. We needed a solution, and the answer was to create native C libraries that Java would call, using the appropriate Java native interface.

One result of the Java wars between Sun and Microsoft is the plethora of means of accessing native code through Java, and this paper is too brief to cover all the methods possible in creating a Java native interface. Instead we will point you to the appropriate documentation in the Microsoft Java SDK on their Remote Native Interface (http://www.microsoft.com/java/sdk/20/jnative/rni_introduction.htm). Sun also has a tutorial on their Java Native Interface at http://java.sun.com/docs/books/tutorial/native1.1/index.html. Microsoft also provides direct access to the Win95 operating system through their VM, using an interface called J/Direct (http://www.microsoft.com/java/sdk/20/jdirect/jdirect.htm), and Apple provides similar functionality through JDirect.

In our case, we used the Microsoft Remote Native Interface on the Win95 side (since we were using the Microsoft VM, and J/Direct was not available yet), and JDirect on the Apple side. This gave us access to the DirectSound and ActiveMovie/DirectShow libraries , and on the Macintosh, Quicktime and other useful Toolbox calls.

In general the native library implementation went smoothly. The one key was to do the parameter conversions correctly. For instance, since Java has no pointers, they are usually represented as ints on the Java side. Pointers to structures are a little trickier, but some VMs provide ways to access such data structures. You should also never refer to or change the pointer variables, unless you're looking for trouble. Speaking of trouble, running native libraries from Java is relatively easy, compared to running Java code from C or C++. This is a nightmare of conversions and obscure calls, and is recommended only for the foolhardy or technologically advanced.

One serious problem was not discovered until Internet Explorer 4.0 shipped, two weeks before our ship date. Microsoft had significantly changed their Remote Native Interface, requiring that the native library return a version number for the virtual machine to accept it. This hurt us quite a bit, and has soured us on depending on the presence of a reliable virtual machine. The short term solution was to have users either install the old VM, thereby disabling Java support in Internet Explorer, or tell them that they wouldn't have music or videos. The long-term solution was released in a recent patch, which was to create two separate .dlls, and load one or the other depending on what version VM the game is run under.

## 6 Rewards

We would not have taken on the challenges if Java didn't have some benefits. Many of these rewards make Java a better development language than C++. Some of these benefits came as a surprise during Politika, but others we knew ahead of time.

### 6.1 What we knew

Java's automatic garbage collection removes the worry over memory leaks. Java automatically handles the deallocation of all objects created. Right away you will notice there is no free() method or delete operator like in C++. In fact there is no corresponding destructor to go along with a class's constructor. You get used to this feature very quickly and you'll find that you're not creating common bugs like forgetting to destroy an object and wasting memory or allocating an object that has been freed. The lack of pointers and pointer arithmetic eliminates another common source of bugs, for instance trying to access memory that is either not allocated or is not really within the data structure you want. Java has a ArrayIndexOutOfBounds exception, which can be caught and fed to the appropriate error handling routine.

Having access to the source code of the language is very useful. Most of the time if you want to change the default behavior in Java you will just extend a class and override the methods you need. Occasionally you notice that a class is just not doing what you'd like it to do. For example, we wanted a Vector that handled ints. Since Vector only handles objects, we wrote a simpler IntVector class based on Vector. It was much easier to modify Vector than to completely write the class from scratch and test.

The richness of the built-in libraries was also a big win for us. With classes like Vector and Hashtable we were able to get to the fun coding faster since we didn't have to write common data structures and worry about their robustness. Some might argue that the Standard Template Library provides the same functionality, but it has a steep learning curve. The Java libraries have a very well-designed and intuitive interface.

### 6.2 Pleasant surprises

Try/catch/finally statements are not new, but Java forces you to use them, especially when dealing with data structure access and reading and writing files. Because of this, we found ourselves using try and catch more often in other parts of the code. While we didn't necessarily write fewer bugs, the enforcement of exception handling made us write better code so we could catch many bugs early on.

Another surprising advantage is that Java compiles very fast. Compiling in Java generates byte-codes which are interpreted by the virtual machine at run-time. This takes considerably less time than a C++ compiler, which also has a relatively lengthy link phase. So your typical edit-compile-run cycle of development goes much quicker.

Javadoc is a great tool provided with Java to automate documentation. By using special comment blocks in your code and tags you can get some very useful documentation about all of your classes. Even if you don't provide any comments, javadoc will output an html document that gives the hierarchy of all your classes, plus for each class it will say whether it is extended from another class, list any interfaces, and list all the methods and variables. You can specify all methods to be listed or only a combination of public, private, and protected. Each method will also list its arguments, the return type and any exceptions it might throw. All of this comes in very handy when you have to use someone else's class in a project and you don't want to search through their code just to find out how to access a certain variable.

## 7 Current Project (tentatively called Hostile Takeover)

Hostile Takeover is a turn-based strategy game that has similar performance requirements as Politika. But instead of a 2D map, we are using an isometric view to represent the playing field. The turn-based nature of the game still makes Java a good choice.

### 7.1 Why continue to use Java?

First and foremost, Java is a great development language. We wouldn't continue to program in Java if we didn't like the language or if we believed it wasn't up to the task. We have learned a lot from our previous experience on Politika, know the sorts of pitfalls to avoid, and know the good features that we can exploit to our advantage. And now that 1.1 is the established version, we are able to take advantage of some improvements to the language, most notably the more efficient event model, improvement in graphics functionality, and faster file I/O. And while we can’t use them in this project, we are looking forward to the new features in version 1.2, namely JavaSound, Java2D and Java3D (with built-in DirectX support).

Another factor in using Java again was an existing code base. We have the framework and tools for building games in Java, particularly the custom interface code which took up a large part of our time during Politika. InVerse is now a familiar entity and has been tuned to our specifications, which means we don’t have to spend time writing and/or debugging something new.

Finally while we didn’t use it in Politika, we also still have the ability to create an applet version of the game. We plan to take advantage of this in Hostile Takeover, as this will allow us to have an online demo with relative ease.

### 10.3 What we're doing differently

As previously mentioned, we have switched to Visual Café for our development environment. The main feature that attracted us was its ability to compile Java to a native Win32 executable. This means we don't have to install a virtual machine or worry about other virtual machines being installed that cause trouble (like the problems with Internet Explorer 4.0 mentioned under Native Code, above). Also, an executable means no interpreted code and therefore faster execution.

Due to some of the cross platform issues (again, mainly testing time) and the poor graphics performance built into Java, we have decided to only develop for the Windows platform. By writing native DirectDraw libraries, we can increase the graphics performance and make features of our games more competitive with other products. We haven't completely abandoned the idea of a 100% Pure Java game, but we need to make the best game we can on the Windows platform – and that means native code. Our plan is that at the end of the project we will have written portable code, with the native portions encapsulated so that we can release versions on other platforms with minimal effort.

## 8 Conclusion

We hope this gives you some sense of what is involved in writing Java games. We have presented some of issues involved in general, as well as some of the specific challenges we had with Politika. For game engine development, we feel Java is fast enough – for non-media-intensive applications the just-in-time virtual machines are nearly as fast as native C++ code. At the current time it can use help with graphics and sound, though, which is where the native libraries can provide additional support. Until the introduction of Java 1.2 and Java3D, we believe that this hybrid approach will allow us the flexibility we need to write games we love in the language we want.