

Want to build a better Set-Top Box ?

By Mr. Rene Leermakers, (PhD) – CTO of NSIcom

Why Java

In the few years of its existence, Java has attracted so many in the computer field, that it is now a programming platform of utmost popularity. Java is a multi-faceted phenomenon, and its popularity stems from the combination of its facets:

The Java language is of superb design, the culmination of many years of research in object-oriented language design.

The Java libraries offer well-designed, standardized, programming abstractions.

The Java Virtual Machine (JVM) uniquely supports important emerging issues, such as security and persistence. At the same time, it is a universal computation engine that makes the same software run on different hardware.

The Java vision takes a simple yet profound idea to its logical extreme, namely that a computing device, such as a PDA or a set-top box, should ideally be modularized into three layers:

The application layer. This is the highest layer, the layer that provides functionality to the user, the layer of things like an electronic program guide, a browser, and games.

The hardware layer. This is the lowest layer, typically one central processor with all sorts of peripherals.

The abstraction layer. The intermediate between the application and hardware layers.

Applications access hardware only through functionality offered by the abstraction layer.

Examples are Linux/X API, Windows API, OpenTV, and Java.

The advantage of this universally accepted design approach is clear: the consequences of a change in the hardware are limited to the abstraction layer. Applications, if written in a third-generation language like C, need only be recompiled. What sets Java apart, is that it uniquely offers an application-oriented perspective:

Java not only abstracts *from hardware*, it abstracts *for applications*. The Java abstractions are higher, and this makes Java applications simpler to write and maintain. For instance, a program that accesses the Internet is a trivial exercise. Java also provides abstractions not directly related to hardware. Examples are numerous primitives in the Abstract Window Toolkit (AWT), and a tokenizer for the analysis of input streams.

The Java view is that it is not that the hardware that should be movable independent of applications. Rather, applications should be movable independent of hardware. A Java application runs on any hardware empowered by the Java. There is no need for accommodation to hardware whatsoever, not even recompilation.

In the same spirit, the Java Virtual Machine only loads Java classes when they are needed by the application. It is inherently dynamic.

Why Java for Set-Top Boxes

The dynamic nature of Java is the most compelling reason to use Java in the design of a set-top box. Almost all Interactive Television standards chose Java as their application engine.

The reason is basically economic: broadcast bandwidth is limited, hence broadcasting of hardware-dependent applications expensive. It follows that the application engine should be an interpretive one, and Java is really the only such engine that does not limit the range of applications.

Similarly, for the Set-Top Box producer, it is attractive to write resident applications, such as a browser and user-interface applications, like an electronic program guide, in Java. Updates are easy, the platform-independence eliminates a whole dimension of version management. As updates will likely occur through broadcasting, bandwidth cost is reduced. In addition, if resident applications are written in Java, device production cost is reduced. In fact, there is reduction of cost at every layer:

1. Application layer. The Java language and its high-level API reduce software development cost, and improve the quality of programs. The cost of software is inversely proportional to the size of its distribution. That is, the price per copy of applications on top of a certain abstraction layer is bounded from below by the market share of that abstraction layer. Windows applications run only on Windows. And Java is always there, already now.
2. Hardware layer. Abstraction layers create competition at the hardware level, and drive down prices. And Java is always there, already now.
3. Abstraction layer. The specification of the Java Virtual Machine is fairly simple, and open to competition. There are already some ten different implementations, whereas no one is going to "clone" Windows CE or OpenTV. Again, competition drives down prices.

So, where is the catch? Well, there is a cost associated to Java. Java needs a faster processor to achieve a given functionality, compared to an equivalent C program. And, until now, this cost was the problem. But Moore's law is saving Java. A browser written in Java, runs great on a 1000 Mhz X86 or equivalent PowerPC, Arm, Mips, TriMedia, and so forth, and so do user-interface applications. When it was launched, Java was a revolutionary and future-oriented idea. Only now all conditions for its realization are fulfilled. Java is mature, hardware is sufficiently powerful, and quality applications are coming up. The enormous Java programmer base will create a stream of killer apps. The Espial software is only a start.

Why NSIcom

Java is always there, already now. NSIcom provides you with a stable Java implementation (JSCP™) tuned to embedded systems, on the platform you choose. The graphics with its television fonts make it the leader in the set-top box market. For Windows CE devices, NSIcom offers the CrEme™ Virtual Machine, a multi-featured product with alternative implementations of the AWT, flexible footprint, and adaptive compilation.

The desired behavior of a JVM in an embedded device goes beyond the correct execution of Java programs. The JVM must operate robustly and predictably in the face of inevitable resource restrictions while minimizing interference with other native functionality inside the device. It must adapt smoothly to properties of device features, and show optimal performance under conditions pertaining to embedded devices. NSIcom addressed the needs of embedded systems in numerous ways.

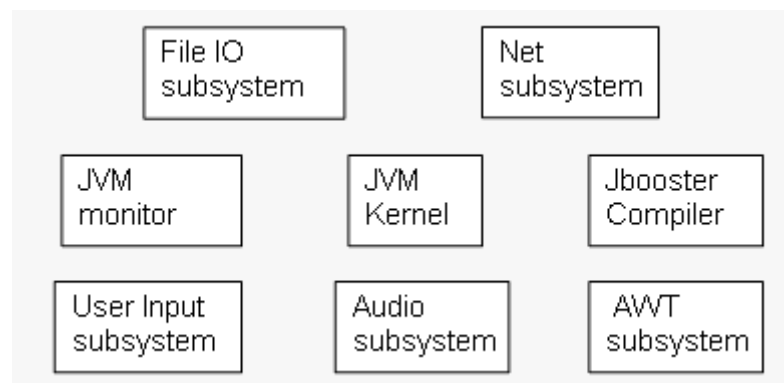
Take graphics. The simplest way to implement Java's graphics library, the AWT, is to express it in terms of a general graphics library provided by the OS. Alternatively, the graphics can be handled completely inside the JVM, except for a simple pixel-level function that transforms a JVM screen buffer into the appropriate format required by the platform. If designed well, the latter approach leads to superior graphics performance, as shown by NSIcom's products. An important corollary is the availability of virtual machines with the exact same look-and-feel on different platforms. Another advantage is that as the use of a usually heavy-weight software layer (e.g., X-Windows) is avoided. The hardware platform needs to provide only primitives for controlling placement of graphical output on the screen, overlaying, and the like.

Another example is multi-threading. The Java language supports and in a sense promotes the use of parallelism to structure programs. Java programs must therefore be expected to create many, often small, threads. Operating systems provide an entity parallel to Java threads, OS tasks. Hence, the simplest way to implement Java threads is to map each thread to an OS task. However, the kernel services used to manage OS tasks are generally optimized to handle a limited number of tasks, and may use priority schemes and communication

primitives that may not suit Java. Alternatively, the JVM may be implemented as a fixed number of OS tasks (one, for instance), and manage Java multi-threading internally. In this way, the JVM has complete freedom to tailor multi-threading in Java programs, to the needs of Java. For instance, NSIcom's JSCP implements parallel execution through a platform-independent polling machine and supports dynamic thread stacks. Dynamic thread stacks can grow and shrink as circumstances require, and eliminate the problem of stack overflow. The mechanism obviates the need to permanently allocate a worst-case size stack for every thread, and provides a level of robustness that cannot be attained otherwise. A JVM with complete built-in support of threads becomes independent of the idiosyncrasies of the OS task support. An important corollary is that a Java program will display identical timing behavior on altogether different platforms.

And so, encapsulation of OS-like functionality inside the JVM gives NSIcom an important competitive edge. The technique maximizes the platform-independency of Java, and the likelihood that Java programs developed on a desk-top machine will run in the embedded device in the exact same way. Because the JVM is implemented as a fixed number of tasks with fixed resources, interference between the JVM and the other native functionality can easily be controlled. If no other native functionality exists, it may be possible to strip down the OS to a very small kernel that provides only the simple primitives needed by the largely self-contained JVM.

NSIcom's products are designed to be highly configurable to device features. They are constructed in terms loosely-coupled software components. The following is a typical decomposition:



One may think of loosely-coupled components as Active-X components or Java Beans, though there are many ways to implement them. The essential thing is that components can be left out in a simple way, without any changes to other components, so that the footprint can be reduced if application demands are limited or hardware components missing. A JVM that is composed of software components can be distributed as a collection of dynamic libraries to enable the addition or removal of functionality after installation. NSIcom's CrEme product is in fact distributed in this way. Configurability is essential for an embedded Java product, given the diversity of embedded systems.

NSIcom has developed a scalable adaptive compilation technology, JBooster, which achieves maximal efficiency improvement for all memory sizes. Only active code is compiled and compiled code that ceases to be active can be un-compiled, dependent on memory conditions. The basic observation underlying the JBooster technology is the observation that, except for the case of trivial short methods, 90% of execution time is spent in a Java method is actually spent within 10% of the code of the method. For that reason, JBooster compilation is conservative, and limited to very active code within methods, such as the bodies of often-executed loops, and very- often invoked methods. Rarely executed code, such as code for exception handling, is never compiled. This results in a dramatic reduction of the size of compiled code, for the same performance gain.

Dr. Rene Leermakers

CTO NSIcom

rene.leermakers@nsicom.com

About the author

Dr. Rene Leermaker, CTO of NSIcom, is a well known scientist in the area of Digital TV.

Mr. Leermakers holds a PhD in physics and formerly was a senior scientist in Philips Electronics Laboratories in Europe and the US, where he developed early prototypes of Interactive Television. Mr. Leermakers published many scientific articles and authored various books on physics and computer science.