

## Java Simplifies Embedded Programming, Reducing Errors

*Java is a modern, high-level object-oriented programming language. As such, it builds on time-proven technologies that have been shown to greatly improve the test-and-verification productivity of software developers. Developers estimate that test and verification represents between 30 and 50% of the total work involved in a typical development effort. In this article, I describe some of the ways that Java object-oriented features simplify the embedded developer's job.*

### MODERN PROGRAMMING TECHNIQUES IMPROVE EFFICIENCY

**D**One of the earliest lessons taught to aspiring software engineers is the importance of decomposing large problems into small problems. Way back in 1956, the psychologist George Miller discovered that typical humans can simultaneously comprehend a maximum of seven "chunks of information," plus or minus two (see "The magical number seven, plus or minus two: Some limits on our capacity for processing information," published in volume 63 (1956), number 2 of *The Psychological Review*).

This message has become a unifying theme throughout undergraduate software engineering programs. This is the reason we teach students not to use global variables, to narrow the scope of local variables, and to avoid go-to statements. And for this same reason, we tell them to break large programs into many small procedures and to clarify division of responsibility between independent components by asserting preconditions, postconditions, and system invariants.

The main point: good software engineering consists of good organization. When we architect a system, we clearly identify the responsibility of each software component. When we build the software, we make sure that each component fulfills its complete responsibility without doing anything more than it was originally assigned. Good software engineering partitions information consistent with assigned responsibility. Information is made visible only to those components that need to know the information. There are several reasons for all of these guidelines:

- to reduce the introduction of errors during development,
- to reduce the effort required to test, debug, and inspect the code,
- to improve the flexibility and to reduce the costs of maintaining the code, when it becomes necessary to make future modifications, and
- to facilitate reuse of the code, by simplifying its integration into a large and complex system.

Any discussion of embedded system test, debug, and verification must certainly take into consideration these principles of separating concerns between indepen-

dent software components.

### EMBEDDED PROGRAMMING: THE CRITICAL CHALLENGE

It turns out that much of what we teach undergraduate software engineers is only half true when it comes to real-world development of embedded systems software. One of the reasons that it's so difficult to write embedded real-time software is because, due to resource limitations, it is not practical for developers of one component to entirely ignore the concerns of other components. For example, many embedded computers have severe memory constraints. If the total application must use no more than 384 KB of memory, then a failure to release the temporary memory used by one component may prevent another component from being able to carry out its intended function. Likewise, if a given component has a real-time deadline to carry out a particular operation within 10 milliseconds, some other component might hold a critical shared resource locked during that time span, forcing this task to miss its deadline. Or another higher priority task might simply hog the CPU throughout the window of time that this task needs to run in order to meet its deadline. To ensure that his or her own components function correctly, each embedded developer must worry about what every other embedded developer is doing. And that's what makes it so difficult to develop and debug embedded software.

### JAVA: STATE-OF-THE-ART IN EMBEDDED SOFTWARE DEVELOPMENT

Java is a modern, high-level object-oriented programming language. As such, it builds on time-proven technologies that have been shown to greatly improve the test-and-verification productivity of software developers. Developers estimate that test and verification represents between 30 and 50% of the total work involved in a typical development effort. In this article, I describe some of the ways that Java object-oriented features simplify the embedded developer's job.

Java programs are composed of multiple class definitions. Each class definition describes a combination of information (variables) and operations (methods) that make use of the information. Generally, the Java pro-

programmer carefully structures the class definition so the variables of the class are only visible to the operations of the class. And the programmer restricts the visibility of the Java methods so that only components that are authorized to perform particular operations are able to perform those operations. Methods are tightly bundled with the variables they pertain to, and methods are only made available to other software components on a need-to-know basis. We call this "encapsulation."

Encapsulation eases testing and verification because it uses the compiler and linker to enforce the separation of concerns that software designers architect into their solutions. Unlike less mature object-oriented languages, such as C++, the Java language requires the entire program to be structured as object-oriented classes. In C++, programmers use a combination of object-oriented and more traditional imperative programming techniques. That portion of the program that is developed in the imperative programming style does not scale well. It is harder to develop, test, and debug. And it is much more difficult to reuse this code because interfaces to the data are not clearly defined and because naming conflicts are much more likely in the imperative programming system's single global name space. By restricting the ways that other components can make use of the abstractions represented by each program "capsule," the object-oriented programmer makes sure that he does not have to test and verify all possible combinations of each component with every other component. Instead, he tests and verifies the interfaces of properly encapsulated components independently.

Portability is another high-level benefit of developing with Java technologies. A portable programming language is one that allows developers to write applications that run the same on any CPU or operating system. The relevance of portability to testing, debugging, and verification is several fold. One of the key benefits of portability is that the software can be developed and verified in a completely different environment than where it will eventually be deployed. In other words, an embedded developer can develop, integrate, and test a complete system even before the relevant hardware becomes available. Another benefit of portability is that it aids reuse of software. When you decide to move the next version of your embedded product to a different microprocessor or a different real-time operating system, you can take comfort in knowing that all of the software will easily port to the new target. When developers reuse software, they are essentially reusing all of the effort that was required to develop, test, debug, and verify the software when it was originally created.

Java applications are more portable than C -and C++ applications for a number of reasons. First, the language itself is more completely specified so there are far fewer aspects of the Java platform that are "implementation defined." Second, the Java platform defines a portable byte-code representation that serves as the intermediate form for all Java programs. Java source programs are translated into class files, the equivalent of a C or C++ object file. The content of a class file is portable byte code. Any complying Java virtual machine can load and execute a Java class file. A third

reason that Java offers improved portability is because the Java platform defines a large assortment of high-level APIs that are supported by all complying Java virtual machines. This means that the Java application programmer writes to the Java API rather than to the Microware Maui, or Wind River Zinc, or Qnx Photon libraries.

One of the innovations of the Java platform is its support for secure dynamic loading. Dynamic loading refers to the notion that new class files can be loaded into a Java virtual machine while it is already running an existing workload. Thus, the software that is running on the system is able to evolve to adjust for changing needs and circumstances. To say that dynamic loading is "secure" is to emphasize that extensive type-consistency checking, known to Java developers as byte-code verification, is performed as part of the dynamic loading process. The byte-code verifier ensures that dynamically loaded components do not compromise the type integrity of the existing workload. In other words, the dynamically loaded component may not see variables that are declared to have private scope, and it may not, for example, treat the contents of a reference variable as if that variable represented an integer or floating-point value.

Another aspect of secure dynamic loading involves enforcement of restrictions on the operations that may be performed by certain dynamically loaded components. Untrusted dynamic components are prohibited from performing operations, such as reading or removing files, that might compromise the system's privacy or integrity. A key benefit of secure dynamic loading is that this enables components to be independently tested, debugged, and verified. Because the security system prevents many of the problems that arise when software components interfere with each other's correct operation, only minimal additional testing and verification is needed following integration of independently developed components.

Reflection is an object-oriented concept that refers to the ability of a software component to find out about itself and about other components. Among other capabilities, the Java reflection API allows programs to examine and modify the values of any object's variables. In combination with dynamic loading, reflection enables the introduction of new diagnostic software components into a running system. These newly loaded components serve to monitor performance and isolate bugs. Because the newly loaded components are subjected to byte-code verification, we are able to avoid many of the problems that arise when introduction of new debugging software into a running system changes the behavior of that system.

Dynamic memory management describes the ability of a programming language's run-time environment to temporarily allocate memory for particular needs and to reclaim the memory to serve future dynamic memory needs after its useful lifetime has ended. In older language technologies like C and C++, the dynamic memory management system requires programmers to explicitly release the memory that represents objects no longer being used. This is a common source of

programming errors. When programmers fail to release an object's memory after the object is no longer needed, we call this "a memory leak." The problem with memory leaks is that over time, they may exhaust the pool of available memory. This failure to reclaim memory is likely to compromise the ability of the program to continue running correctly.

A more severe programming error consists of accidentally releasing an object's memory while the object is still in use. We call this a "dangling pointer error." In this case, the memory for the in-use object may be reallocated to serve other purposes. This situation almost always results in an error so severe that the program cannot continue and is aborted. The great difficulty of dealing with these kinds of programming errors is that the results of the error manifest themselves much later than when the actual error first occurs. In large complex software systems, these are among the most difficult bugs to deal with.

Automatic garbage collection is a feature of the Java run-time environment that greatly simplifies the task of dynamic memory management. With automatic garbage collection, programmers do not release the memory of dynamically allocated objects. Instead, the run-time environment automatically reclaims the memory of objects that are not reachable by following a chain of pointers starting with any programmer-declared or run-time environment variables.

In systems that use automatic garbage collection, it is much easier for developers to design protocols to make sure that the memory for unused objects is returned to the free pool. As long as each component that desires shared access to an object nulls its references to that object after it no longer requires access, the object's memory will be reclaimed when all components are done using that object. Even more important, automatic garbage collection completely eliminates the possibility of dangling pointer errors. Studies performed on large software development projects suggest that programmers are approximately 40% more productive with automatic garbage collection than without it. The large majority of this productivity saving can be attributed to significant reductions in the test-and-debug phase of development.

The Java run-time environment includes built-in support for monitoring and debugging Java applications. This support is built on the Java Debug Wire Protocol (JDWP), which serves to connect the run-time environment to a debugger user interface running on another computer. This capability allows any of a variety of different JDWP-enabled integrated development environments, such as Metrowerks' Code Warrior and OTI's Visual Age Micro Edition, to debug Java applications running on any Java virtual machine that supports the JDWP debugging protocol, like NewMonics' PERC and OTI's Visual Age Micro Edition virtual machines.

Remote debugging support is especially useful for debugging embedded computer systems, many of which lack the memory and user interfaces that would be required for self-hosted debugging. Another key benefit of remote debugging capability is that technical support engineers can attach JDWP user interface

computers to embedded products that are operating in the field to monitor and diagnose systems as the products are running.

## **JAVA'S SUPPORT OF REAL-TIME NEEDS**

It is important to recognize that most Java run-time environments are not tailored to the special needs of embedded real-time developers. The Java language specification does not preclude virtual machine implementations from addressing the concerns of this special marketplace, but it doesn't require it either. A number of vendors, such as Esmertec, Hewlett Packard, NewMonics, and Object Technology International (a subsidiary of IBM) have all undertaken to deliver Java virtual machine technologies that provide the control and determinism required by embedded real-time developers. In this section, I describe a few of the issues that need to be addressed in supplying real-time implementations of Java virtual machine technologies.

Automatic garbage collection, a key enabling technology offered by Java, is both a blessing and a curse to the embedded real-time developer. I've already described many of the benefits. The problems with garbage collection are that it is very hard to provide implementations that meet the reliability and determinism needs of embedded system developers. Virtual machines designed for desktop or server computers generally ignore many issues that are of critical importance to the embedded system product's reliability. For this reason, the testing and verification of your embedded system product must focus not only on the application code, but on the implementation of the virtual machine's garbage collector as well.

Here are some of the issues you need to consider when evaluating garbage collection technologies. First, some virtual machines use type-accurate garbage collection and others use conservative garbage-collection techniques. A conservative garbage collector makes conservative estimates as to which objects are still reachable from the root set of pointer variables. Whenever there is any question, the conservative garbage collector assumes the object is still in use and does not reclaim its memory. Over time, an accumulation of overly conservative approximations may result in failure to replenish the free memory pool, which may hinder the ability to guarantee long-term reliability. Second, some virtual machines defragment memory as a side effect of garbage collection. This means that scattered free segments are relocated to contiguous locations and then coalesced into larger free segments. When deploying an application on a virtual machine that does not defragment memory, the test and verification effort must satisfactorily demonstrate that memory-fragmentation problems will not arise during execution of the application. Third, if your application has real-time constraints, you want to make sure that the garbage collector does not interfere with the ability of your application tasks to meet their real-time deadlines. Many garbage collectors lock out execution of Java threads for relatively long periods of time (up to several seconds) during execution of particular

critical sections of code. You'll want to select a virtual machine that breaks the complete garbage-collection effort into many small increments of work, that has a garbage collector that continues to make forward progress even when it is frequently preempted by high-priority tasks, and that paces execution of the garbage collector so that memory can be reclaimed at a rate consistent with the application workload's demand for memory allocation. Real-time virtual machines supporting all of these capabilities with worst-case preemption latencies of less than 200 microseconds on typical Pentium hardware are commercially available.

In industries for which software is viewed as a mission-critical component, it is common for the test and verification process to take great care to ensure that the deployed product is exactly the same product that has been subjected to the test and verification process. With traditional technologies such as C and C++, this generally means that products are deployed with certain optimizations disabled and certain debugging capabilities enabled. Deciding how to configure a Java application for mission-critical deployment is not as straightforward.

To support dynamic loading of Java class files, many Java virtual machines use a technology known as just-in-time (JIT) translation. The idea is that portable byte codes are loaded into the virtual machine, and these byte codes are translated to the native instruction set just in time to be executed. Some of the more sophisticated JIT translators (so-called dynamic JIT compilers) monitor the behavior of translated components and dynamically retranslate performance-critical components to optimize the component based on its execution profile. These JIT and dynamic JIT technologies provide reasonably good performance in general, but reliability and predictability may suffer. When a bug is discovered in the field, it is nearly impossible to replicate that bug in the test lab because doing so requires information about the translations of all the various Java components in the system, and that information is simply not available.

Rather than using JIT compilers, developers of mission-critical Java products often prefer to use ahead-of-time (AOT) compilers to translate all of the mission-critical components in their system. An AOT compiler translates class files to native code before the program begins to execute. This way, developers can be sure that they have tested exactly the same product they are shipping. And if any new problem reports come in from the field, the technical support team has a much better chance of replicating the problem because they have the exact same software configuration in the lab.

As you contemplate the use of JIT and AOT compilation technologies in your embedded Java product, there are a few more issues for you to carefully consider. First, make sure that the VM supports the JDWP debugging protocol, and that the JDWP debugger interface works for compiled code. You'd be surprised at how many virtual machines simply disable all native translation technologies whenever you enable debugging. Second, make sure that the virtual machine sup-

ports AOT translation and that you can use the JDWP debugger interface on AOT-translated components. And finally, if you need dynamic loading of mission critical components, make sure the virtual machine has the ability to dynamically load AOT components.

## THE FUTURE FOR JVMs

You can use today's real-time virtual machines to develop and deploy high-performance, reliable, and deterministic Java applications. The future will bring even greater flexibility and improved support for integration of independently developed software components in embedded real-time systems, further reducing the amount of time and effort you spend on test and verification.

Embedded Java objects will have CPU-time and memory budgets that are enforced by the platform's run-time environment. Since these resource budgets are enforced, developers of components do not need to worry about the possibility that some other component is going to steal away the memory or CPU time that their components need for reliable operation. This further reduces the effort of integration testing and verification.

Embedded real-time Java components will be expressed in portable binary representations that encode not only the sequences of operations performed by each component, but also the ability to determine its memory and CPU-time resource requirements within a given execution environment. When such components are loaded into a real-time virtual machine environment, they will configure themselves to determine their resource requirements and will negotiate with the run-time environment's resource manager to obtain resource budgets.

Abstraction mechanisms will represent hardware devices as objects with well-defined I/O interfaces. Simulation environments will allow hardware device objects to be emulated by software, enabling development to precede availability of hardware. Time also will be abstracted as a programming-language concept that programmers use to describe real-time deadlines, timeout delays, periods of execution, and CPU-time resource budgets. Given that time is an abstraction of the programming environment, simulation environments will enable slow-motion execution of complex systems, with individual components synchronized according to the passage of virtual time under control of the simulator's run-time environment. Even emulated hardware devices can be simulated in slow motion. Such mechanisms greatly improve the ability of a developer to test and debug complicated race-condition interactions between independent components.

The Java run-time environment will enable new functionality to be dynamically loaded into the system's workload, replacing or enhancing the system's functionality. The run-time environment will prevent dynamically loaded components from interfering with the resource requirements of previously installed components. The run-time environment will empower application-specific profiling, monitoring, and debugging of embedded systems. These application-specific diag-

nostic tools will load themselves dynamically into deployed systems, add themselves to the system workload without interfering with the resource requirements of the existing workload, and make use of the run-time environment's reflection and introspection services to obtain necessary information. In this way, system analysts will be able to inject application-specific debugging tools into field-deployed systems, without even requiring those field-deployed systems to be shut down and/or restarted.

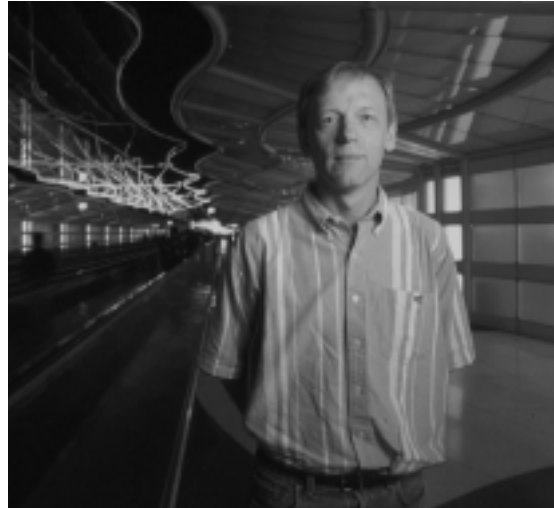
Additional information describing future enhancements to the Java programming language and deployment platform is available in two reports that I authored over four years ago. The first, "Issues in the design and implementation of real-time Java," was published in The Java Developers Journal and in Real-Time Magazine. You may download this article from [www.newmonics.com/dat/rjji.pdf](http://www.newmonics.com/dat/rjji.pdf). The second paper, titled "The PERC real-time API," was one of the key references for discussions carried out within the NIST-sponsored Real-Time Java Requirements meetings. This paper is available for download from [www.newmonics.com/dat/perc\\_api.pdf](http://www.newmonics.com/dat/perc_api.pdf).

## SUMMARY

In conclusion, object-oriented Java technologies significantly reduce the test and verification efforts associated with developing and integrating software components in embedded real-time systems. Future

enhancements to Java real-time virtual machines will offer further reductions in test and verification efforts ■

*Dr. Kelvin Nilsen founded NewMonics, a leading provider of advanced Java and Linux development tools for next-generation embedded systems, in 1996 after developing break-through technology with his research on high-level programming of real-time software. As Chief Technology Officer at NewMonics, Kelvin continues to direct the design and architecture of NewMonics' current and forthcoming products.*



## Dedicated Systems Magazine

Dedicated Systems Magazine is an international quarterly publication of Dedicated Systems Experts. A yearly company subscription entitles up to 5 company members to receive each 4 double issues. To subscribe, complete the subscription form at the end of the magazine and mail or fax it back with your check, purchase order or credit card references. All payments must be in one of the currencies listed here below.

### Subscription rates

#### One Year Subscription:

4 Dedicated Systems Magazine Issues: 209 EURO or 270 USD (5 copies including Gazette)

#### Two Year Subscription:

8 Dedicated Systems Magazine Issues: 359 EURO or 470 USD (5 copies including Gazette)

Dedicated Systems Magazine  
Rue de la Justice 21 - 1070 Brussels - Belgium  
Phone. 32-2-520.55.77 - Fax. 32-2-520.83.09 - [Info@dedicated-systems.com](mailto:Info@dedicated-systems.com)  
Visit us at <http://www.dedicated-systems.com>  
HRB-RCB 517.203 - BTW-TVA BE 441.901.415