Towards Flexible and Safe Technology for Runtime Evolution of Java Language Applications

M. Dmitriev
Sun Microsystems Laboratories, 901 San Antonio Road, Palo Alto, CA 94303, USA
Email: Mikhail.Dmitriev@Sun.COM

September 27, 2001

Abstract

There is a class of important computer applications that must run without interruption and yet must be changed from time to time to fix bugs or upgrade functionality. In this paper, we present the initial runtime evolution framework which we have developed for the HotSpot Java Virtual Machine, that allows us to change running applications on-the-fly, without interruption. We describe our staged implementation plan, where stages correspond to increasing levels of implementation complexity, yet on each stage a reasonably complete set of facilities is provided. The first stage — support for changes to method bodies only — has already been implemented and is to be included in the forthcoming Java 2 Platform release. We discuss multiple policies for dealing with active methods of running applications, present our thoughts on instance conversion implementation, and suggest that runtime evolution technology can be used for dynamic fine-grain profiling of applications.

1 Introduction

There is a class of computer applications that must run continuously and without interruption, and yet must be changed from time to time to fix bugs or upgrade functionality. The most spectacular examples of such applications are financial transaction processors, telephone switches, air traffic control systems, etc. A common solution to make highly important, mission-critical systems run non-stop, is to design them in a special way and run on a specially configured, redundant hardware (the latter is usually needed anyway to support fault tolerance). For example, according to [8], Visa makes use of 21 mainframe computers to run its transaction processing system that comprises some 50 million lines of code. The company is able to selectively take machines down and upgrade them, while preserving the application internal state in the remaining on-line computers. As a result, the system that is updated on average 20,000 times a year, requires about 0.5% downtime.

However, such a complex and expensive solution may not be affordable to many less sophisticated users, e.g. those running e-commerce servers. Still, many of them would benefit from a relatively easy-to-use and general-purpose technology supporting runtime changes to applications. At the very least, the developers can use it to diagnose certain bugs (which may be hard to exercise during test runs with simulated workloads) by adding trace printing operators in arbitrary places of the running program. Ideally, a runtime evolution technology should allow the developers to make any changes that they would otherwise make off-line, while preserving the internal application state and providing as many guarantees as possible that the transition will complete safely.

The Java programming language, which at the dawn of its history was targeted at relatively small applications, such as applets and those used in hand-held devices, is now becoming increasingly popular in many other areas, including server-side applications. Therefore, developing a mechanism that would support runtime changes of Java applications (or *runtime evolution* as we will further call it) is a quite natural step. It is greatly facilitated by the fact that in the classic case Java applications are initially compiled into machine-independent bytecodes, which then run on top of a Java Virtual Machine (JVM). Compared to a statically compiled application, any JVM already has much of the infrastructure that would be needed to inspect and change running applications. Bytecodes are also much easier to analyse and instrument than machine code.

We present the initial runtime evolution framework that we have developed for the HotSpot JVM. Our approach allows us to change applications as they run, by feeding the JVM new versions of some of the classes that it currently executes and making it "hot swap" from old classes to the new ones. New class versions are prepared as usual - that is, typically their source code is edited and compiled by an ordinary Java compiler. The developer can then choose the classes they want to replace, compare their old and new versions, set some options, and transmit the classes to the JVM using a special *runtime evolution client* GUI tool. The client and the JVM communicate using the infrastructure provided in the *Java Portable Debugger Architecture* (JPDA) [9]. The class files, along with the instruction to perform their "hot swap", are transmitted using *Java Debugger Wire Protocol* (JDWP) over e.g. a socket connection (the JVM should be started with a set of command line options that would make it support this or other *transport*). Finally, a JVM-internal call `RedefineClasses()` is invoked. It takes essentially names of classes to be redefined and bytecodes for their new versions, and performs the actual job. This call is the key element of the described chain; the rest of this mechanism can be redefined by the user. That is, a different client can be used, or a different way of communication, or the application itself can be designed to accept new class versions and invoke `RedefineClasses()` through a native method.

In this paper we concentrate on the details of the runtime evolution implementation in the HotSpot JVM. This is obviously the most interesting part of the story, a non-trivial problem posing a number of hard-to-answer questions. A running application has a state, the main part of which is the contents of the stacks of its threads and the heap. To evolve an application dynamically, we have to somehow match its current code and the current state of its stack with the new code (and possibly new stack frames and instance formats that it defines). This is the main issue of runtime evolution, and we do not yet have a solution for it in an arbitrary case. Thus our current goal is to identify possible specific cases, when this transition can be performed such that its effects are predictable.

To make the research more manageable and to facilitate converting the experimental implementation into a product, the decision was taken to implement runtime evolution in the HotSpot JVM in a number of stages. In the first stage, a very limited subset of class evolution functionality was implemented, which would allow developers to make only very restricted changes. On the other hand, it required limited implementation effort, while still being useful, at least in the context of debugging. At the following stages, the range of allowed changes and, consequently, implementation complexity, gradually increase. We believe that in future, additional functionality at the client side would also be required, that would help the developer to understand the consequences of changes, or e.g. apply these changes only when certain conditions are met, such that their results become more predictable.

This paper is structured as follows. We first describe our implementation platform — Sun's HotSpot JVM. Implementation stages as we currently see them are presented in Section 3. The format of this paper does not allow us to go deep into the implementation details, but the interested reader can find them in [3]. In Section 4 we present the policies for dealing with active methods of classes being changed. We then discuss the directions of future work: conversion of instances of changed classes (Section 5.1), implementation of the remaining policies for dealing with active methods (Section 5.2), and application of our technology for dynamic fine-grain profiling (Section 5.3). Finally, we present conclusions.

2 HotSpot Java VM

The HotSpot VM is Sun's main Java VM, available in the Java 2 Platform releases starting from Version 1.3. It was first officially launched in April 1999. In the following sections we describe its features that are important in the context of our work.

2.1 Mixed Mode Execution and Deoptimization Mechanism

The HotSpot VM is equipped with both a dynamic compiler[1] and an interpreter, and runs applications in *mixed mode*. This means that the most heavily used parts of the application are compiled, whereas the rest is interpreted. Supporting dynamic substitution of the native code would be not an easy task (at least because we would have to implement it for two compilers and two target hardware architectures, SPARC and Intel). The problem would be further exacerbated by the fact that the compilers actively use method inlining. As a result, the code of a method that we may want to replace can be actually inlined in many other unrelated methods, and we would have to replace them all to make our technology work correctly. Fortunately, the so-called *deoptimization* mechanism (see below) available in HotSpot, allows us to consider only the interpreted mode and manipulate only with the architecture-independent bytecodes that are never inlined.

The deoptimization (decompilation) mechanism was originally introduced (in HotSpot's predcessor, the Self VM [5]) to make *aggressive method inlining* possible. When aggressive inlining is used, a larger number of methods can be inlined than would be possible to if compilation was purely static (more precisely, if its results could not be changed later). For example, a virtual method `m()` of class `C` can be inlined at a particular call site if it is determined that (currently at least) only one instantiation of this method can be called at that site. This happens, in particular, if in the JVM there are no subclasses of `C` that override `m()`. However, if such a subclass is loaded later, it will be necessary to "de-inline" the above invocation of `m()`. In HotSpot this is done by throwing away the compiled code and switching to interpretation of all of the methods that previously inlined `m()`. If method `m()` of class `C` is changed dynamically, we should perform the same decompilation of dependent code, to prevent calling the old code or executing it in the inlined form by other code which is unchanged.

2.2 Memory Management

In the HotSpot VM heap objects reference each other directly by their memory addresses. Some other VMs for earlier object-oriented languages (e.g. Smalltalk) and Java, were using a level of indirection called *handles* to simplify implementation of garbage collection. A handle was a small fixed-size data area associated with an object and containing a direct pointer to the latter. All objects pointed to each other (and their classes) indirectly, via handles. Thus during garbage collection objects could be moved around freely without changing pointers to other objects inside them — only the pointer to the object in its handle must to be patched. Equally, we would not have to patch pointers to class in all instances of class `C` if `C` is changed

[1] Actually, there are two native compilers — a relatively lightweight one oriented towards client-side applications, and a slower compiler producing high-quiality native code, which is more appropriate for long-running server applications. The user chooses one compiler or another at the VM startup, using a command line option.

and thus the memory address of its internal class object becomes different. The advantage would be greater if some class is changed such that its instances (and thus the pointers to them) need to be updated.

However, since it has been long observed that accessing objects via handles slows down application execution, they are not used in modern industrial-strength JVMs. On the other hand, in HotSpot a convenient internal interface supporting traversal of all of the objects on the heap is available. Furthermore, the infrastructure of the *fully accurate* garbage collector allows the system to find all of the pointers to Java objects, including those located on native stack frames. Therefore, locating and updating the pointers to objects is not a problem for us.

2.3 Internal Data Structures

Our experience with several of Sun's JVMs shows that initial design of the internal representation of objects and classes in the VM memory may affect the implementation of many additional facilities quite significantly. In particular, the degree to which the internal representation of a class is monolithic or dispersed makes a big difference in the effort required to evolve a class and may affect the performance of the resulting system. It is obvious that if a class representation consists of a number of separate small structures pointing to each other, rather than a single compound structure, we can often change only one or two of these structures in order to redefine a class, leaving others undisturbed. This, in turn, can prevent us from the time-consuming operation of patching other classes or instances. For example, in Sun's Solaris Research JVM [10] instances point to a special data structure called a *near class*, and the latter points to the main class object. A near class contains, in particular, the instance layout map and the pointer to the Virtual Method Table (VMT) for the given class. In such a VM it would be possible to make almost arbitrary changes to a class without patching its instances, since only the pointers in the near class would have to be changed.

In the HotSpot VM the internal class structures were designed with performance as a first priority. The resulting class representation can be called half-monolithic. It consists of a number of separate objects: the main class object, the method array, individual methods, compiled methods and the *constant pool* [7]. However, the main class object itself is a variable-size, compound data structure, into which the class VMT and the static variables are embedded. Thus, when the new version of a class has only method bodies changed (we will see that this is an important and quite common case), we can modify a class without replacing the main class object to which its instances point. But whenever we add virtual methods or static variables to a class, we have to replace the main class object, and subsequently patch all of its instances to make them point to the new main class object. Furthermore, adding a virtual method to a class increases the VMT size for all of its subclasses, and thus we need to replace the main class objects and patch instances for all of them.

2.4 Summary

HotSpot, being a high-performance industrial JVM that comprises a significant amount (nearly 300,000 lines) of non-trivial C++ code, has nevertheless proved to be malleable enough for the purposes of this work. Certain properties of this system, such as support for compiled method deoptimisation, the infrastructure for iterating over Java objects and stack frames, and quite good code quality, facilitate implementation of runtime evolution a lot. Some others, e.g. the design of some internal data structures, make our work harder, and affect the performance of the resulting evolution sub-system. Overall, however, our impression is that HotSpot, at least compared to Sun's previous JVMs (Classic and Solaris Research VM), is more suitable for experimental work, particularly for adding runtime evolution support.

3 Staged Implementation

Since implementing runtime evolution for Java is a risky research work, with many conceptual and technical aspects still not entirely clear, it was decided from the beginning to split this work into a number of stages. Each stage corresponds to a certain reasonably consistent level of functionality, that can be delivered with some release of the HotSpot JVM. Currently envisaged stages look as follows:

1. **Changing method bodies only.** At this stage, only one kind of change to classes is allowed and supported, and these are changes to method bodies (code) only. Everything else in the new class version should remain exactly as it was in the old class version. This is a reasonable level of functionality to expect in an advanced debugger[2], and in fact it is planned to use our mechanism both to evolve server-type applications running in the production mode, and applications being debugged. This stage is complete now, the code is being tested and will be included in the forthcoming Sun's Java 2 Platform release.

2. **Binary compatible changes.** At this stage, only *binary compatible* (see [4], Chapter 13) changes to classes should be allowed. The reason for this limitation is that allowing binary incompatible changes presents an additional problem of ensuring application consistency (type safety). Unless we trust the client (which would not be in line with the JVM policy in other cases), we would have to check the consistency of the whole set of classes currently loaded by the VM, every time a binary incompatible change is made. E.g. if a static method is deleted from a class, we should check that no references to this method remain in unchanged classes. For some changes, safety checks are much harder to perform than for most of the others. For example, if an interface `I` is removed from the

[2]There is one IDE currently on the market, IBM's VisualAge for Java [6], which supports changing method bodies in the application running in the debugger.

`implements` list of class `C`, we have to check that there are no attempts to cast `C` (or any of its subclasses) to `I` anywhere in the code.

3. **Arbitrary changes, except the changes to instance format.** At this stage, any changes to classes should be allowed and supported, with the exception of those that affect the format of their instances (unless there are no live instances of this class at the moment of transition).

4. **Arbitrary changes.** All kinds of changes are allowed. If any of them leads to instance format modification, all of the affected instances should be *converted* (see Section 5.1) to the new format to make them match new class definition.

Support for changes to instance format should not necessarily be implemented at the very last stage, though. Implementing instance conversion and supporting binary incompatible changes are "orthogonal" issues, i.e. compatible changes (e.g. adding a public instance field to a class) may require instance conversion, or changes not requiring instance conversion may be incompatible.

The status of the implementation beyond stage 1 is as follows at present. The initial implementation of stage 2 is complete now, and we are experimenting with it. From the technical point of view, the main difference between this stage and stage 1 is support for main class object replacement and instance rebinding (see "Internal data structures" part of Section 2).

We also have an experimental implementation for stage 3. This implementation supports most of the binary incompatible changes, and for each such change checks the unchanged classes for compatibility. However, incompatible changes to the class hierarchy, e.g. deleting classes or interfaces from the `extends` and `implements` lists of a class, are not allowed. The reason is that to ensure application consistency if such a change is made, we would have to analyse method bytecodes, which is much harder than analysing class constant pools for dangling references. The most economic way to implement these checks would be to re-use the *Java bytecode verifier* (see [7]) code, but the required code modification is unlikely to be technically easy.

4 Dealing with Active Methods

A class that we are replacing can have active methods, where an active method is a method for which one or more stack frames on stacks of running threads are present at the moment of class redefinition. The new version of such a method may be different, most likely with a new stack frame format. Something has to be done in order to match these stack frame formats or otherwise make the execution continue without a crash. For debuggers, the most typical approach is to pop (throw away) all stack frames of changed methods, thus diverting the normal execution order of an application. In the VM running in production mode such a technique seems unacceptable in most cases, since it is very likely to result in incorrect behaviour of the target application. Several alternative solutions can be considered:

1. Wait until there are no active old versions of evolving methods. This provides the "cleanest" way of switching between the old and the new program, in the sense that at no time does a mix of old and new active code exist. However, this solution may not always work (we will wait forever), e.g. if one of the evolving methods is the main method of the program.

2. Currently active calls to old methods complete, and all new calls (method calls initiated after the class redefinition is over) go to new methods. This solution is technically the easiest.

3. All existing threads continue to call old methods, whereas new threads (threads created after class transformation) call only new methods. This may be the most suitable solution for certain kinds of applications, e.g. servers that create a new, relatively short-lived thread in response to every incoming request.

4. Identify a point in the new method that corresponds to the current execution point in the old method, and switch execution straight from the old method code to the new one. In certain cases, e.g. when a method being evolved never or rarely terminates, the stack format for the old and the new method are the same, and the changes are free of side effects (for example, trace printing statements are added), this can be the desired and useful semantics. However, in more complex cases it may be very hard for the developer to understand all of the implications of a transition from one code version to another at some arbitrary point. One other application of the mechanism developed for this policy may be for dynamic fine-grain profiling (see Section 5.3).

It looks as if none of these solutions is a "single right" one. Rather, they are different policies, and each of them may be preferable in certain situations. However, solution number 2 is much easier to implement than the others, and it is the one which we have implemented so far.

5 Future Work

5.1 Instance Conversion

Once we allow the instance format of a class to change, we would have to implement some form of *instance conversion*, that is, some means of updating format and contents of existing instances of the changed class. Various forms of instance conversion have been existing for long in the context of object-oriented databases, where objects are made persistent and can exist for long time, often outliving several versions of their class. All of the conversion techniques can be broadly classified as *eager* or *lazy*, and also as *default* and *custom*. Eager conversion means that all objects are converted in one go once the definition of their class changes, whereas if lazy conversion is used, conversion of each object is deferred until this object is requested by the application. Some intermediate conversion forms are possible, e.g. when objects are divided into groups and all objects in one group are converted eagerly when a single object from this group is requested.

Default conversion means that the object contents are copied between the old and the new object versions automatically. Typically, the values of the fields with exactly the same name and type are copied unchanged, the values of the fields that do not exist in the new version are lost, and the values of the new fields are initialized with default values. If custom conversion is provided, the programmer can write more or less sophisticated code that would define how the information is copied and possibly transformed (e.g. a pair of Cartesian coordinates can be converted into polar, etc.).

In [3] we describe a wide range of conversion methods we have developed for a persistent object system for Java called PJama [1, 2]. We implemented eager default conversion and several flavours of eager custom conversion. To the best of our knowledge, our system was the only one supporting custom conversion in the form of Java language code. It allowed us to perform almost arbitrary object transformations.

We believe that the runtime evolution system would benefit from support of both default and custom conversion. Of course, it is hard to imagine making as complex changes to classes at run time as in the case of object database evolution, since making a complex change at run time would require the engineers to take many more possible effects into account. But at least some simple form of custom conversion that would handle cases such as field renaming with type and value unchanged, is likely to be very useful.

Lazy conversion implementation, which is quite desirable in the context of runtime evolution, since it minimises latency, seems very problematic for a handleless VM such as HotSpot. The problem is, if the new object version is larger than the old one, the object has to be relocated. Since the pointers to this object are direct, all of them have to be located and patched at once. This operation would have a prohibitive cost if it is applied to each object individually, over and over again. Thus, it may be more feasible to implement eager concurrent conversion, which would work in much the same way as incremental concurrent garbage collection (and can actually re-use the code of the latter). Such a mechanism will convert objects in parallel with the normal execution of the evolved application, and will only block an application thread if it tries to access an object which has not been converted yet.

5.2 Implementing Multiple Policies for Dealing with Active Old Methods

Several policies for dealing with active methods of old class versions, that we can think of, were first briefly presented in Section 4. We will now discuss possible ways of implementing these policies.

5.2.1 On-the-fly Method Switching

This policy means that we identify a point in the new method that corresponds to the current execution point in the old method, and then continue execution from this point in the new method (provided also that the stack frame formats defined by the old and the new method versions are the same, and thus the new method version can inherit the old stack frame). The main conceptual issue here is how to define the above "correspondence". The answer that we suggest is to compare the bytecodes of the old and the new methods, in much the same way as utilities like Unix `diff` compare texts. We should, however, take into account that the layouts of constant pools in the two class versions may be different, which will lead to different constant pool indices for the same constants and, consequently, different bytecodes for even functionally absolutely the same methods. "Smart" textual comparison of the bytecodes will identify the minimum differences and, consequently, the matching segments. If the current execution point is in such a segment, we can find the matching point in the new bytecode and switch the execution to it.

This description of bytecode comparing, matching point identification and execution switching implies that this process happens for bytecodes in the interpreted mode. If the code we are going to redefine has been compiled, HotSpot always allows us to first deoptimise it, i.e. switch to its interpretated form.

The above method of establishing matching points in bytecode versions does not, of course, guarantee the "correct" execution of the program being transformed — as our technology in general does not guarantee it. However, we currently can identify one case when such a transition is likely to be harmless. That is the case when the changes to the bytecodes in the new version are purely additive, and the added code does not produce any known harmful side effects. Examples of such modifications are adding code that prints some tracing information, e.g. to help identify a bug, or supports profiling (see Section 5.3). Other cases, where an experienced developer is likely to be able to predict the effects of a transition, may also be possible. In any case, in the end we have to trust the skill of the software engineer initiating the change, but we would like to provide as much support for them as possible.

If in the new bytecode version some old code fragments are not present, a question arises of how to perform the transition if the execution is currently inside one of these deleted code fragments. A sensible answer may be to let this code fragment complete, i.e. wait until the execution reaches the nearest "common point". A mechanism of temporary bytecode patching and event notification, similar to the one used to set breakpoints, can be used to implement this efficiently.

Temporarily ignoring the issue of exact stack frame format equivalence (which may be non-trivial in Java, since a class file does not really define the complete layout of a stack frame), we have, as a matter of experiment, implemented the functionality that can substitute methods and switch execution on-the-fly. In the simple tests, where we were just adding `System.out.println()` statements to our code, it worked well for us. Thus, we believe that at least limited applications of this technology guaranteed to not violate safety restrictions, e.g. the one discussed in the next section, and also dynamic fine-grain profiling discussed in Section 5.3, can be utilised almost immediately.

5.2.2 Wait Until No Active Old Methods

This policy of handling active methods is the "cleanest", since it guarantees that two versions for any method can never co-exist simultaneously for a given application. Of course, such a (potential) convenience comes at a price: the developer who would like to use this policy must somehow ensure that the execution will actually reach the point when there are no active old methods. This may become quite complex if an application is multi-threaded.

The implementation of this policy would have to keep track of all of the activations of the old methods. Once the last such activation is complete, the threads should be suspended and method replacement should be performed. Again, this task is more complex in case of a multi-threaded application, since it may happen that while one thread completes the last activation of the old method, another thread calls this method once again.

Method entries and exits can be tracked using, for example, the mechanism of debugging events generation already available in HotSpot. However, this mechanism is expensive, since the event is generated upon entry into and exit from every method. A good alternative may be to implement this policy using the mechanism of on-the-fly method switching discussed in the previous section. This way, method redefinition will consist of the following two stages. On the first stage, we patch the code of the old methods, adding to them the calls to two methods predefined in a special class. These calls increment and decrement a counter, which is initially set equal to the total number of activations of the old methods. The method that decrements the counter should check if it becomes equal to zero. When this happens, it should call the standard class redefinition procedure, that will redefine the classes as originally requested by the user.

5.2.3 Old Threads Call Old Code, New Threads Call New Code

This policy looks more difficult to implement than the others, since it allows the old and the new code to co-exist forever, and the old code can be called over and over again. It could have been possible to implement it in a relatively simple way by allowing two copies of the same class to co-exist legally, and making only one copy "visible" to each thread (this would require indexing the class dictionary by thread ID in addition to class name and loader). Unfortunately, an individual instance may not belong exclusively to one thread, and we would have to somehow use one or another class for the same instance depending on the thread in whose context the execution of a particular method on this object happens (this also makes this policy hardly compatible with changes to instance format). This can be done by creating multiple copies of the VMT and method arrays for the same class, and dispatching each method call depending on the thread. Such a modification to the JVM is obviously too expensive compared to the limited importance of the goal that it should achieve.

So, a better alternative may be to once again use bytecode instrumentation instead of a serious JVM modification. Instead of making the JVM dispatch calls depending on the thread that makes a call, we can make the bytecodes themselves do that. We can synthesise a method that will check the thread that executes it, and, depending on whether the thread is an "old" or a "new" one, call an appropriate "old" or "new" method. All of these methods: the old, the new, and the dispatcher, will have to belong to the "temporary new" class version, that our technology would synthesise. The standard method replacement policy, where old calls are allowed to complete and new calls go to new code, will be used to replace the old class version with the "temporary new" version. The latter will have to be used until all of the "old" threads terminate. After this happens, we can replace the "temporary new" class version with the original new class, thus eliminating the overhead imposed by bytecode-level call dispatching.

It is not entirely clear how to determine when all "old" threads are finished, given, for example, that the main thread usually never terminates until the whole application terminates. One option may be to make the developer who initiates the transformation provide the criteria for sensible distinguishing between "old" and "new" threads, based, for example, on thread groups defined in Java.

5.3 Dynamic Fine-Grain Profiling

One interesting application of class redefinition technology and the on-the-fly method switching policy may be dynamic fine-grain profiling. The standard profiling mechanism presently available in the HotSpot JVM has a granularity limited to a method, since it only generates an event on method entry and exit. It is also expensive, since an event is generated on entry into/exit from each method. Thus, if the user wants to profile just one method, they can only hope that some profiling tool will help them to extract the required information out of the flow of incoming events. To the best of the author's knowledge, no other JVM at present implements anything more sophisticated. If the user wants to measure the time spent in a code section smaller than a method, the only solution is to explicitly bracket the measured code with the user's own measurement code, e.g. calls to `System.currentTimeMillis()`, recompile and re-run the application. Obviously, this is a very tedious procedure, especially when it is used to gradually narrow down the suspicious code area in order to find out which piece is a bottleneck.

Dynamic class redefinition mechanism can be used to implement cheap, dynamic fine-grain profiling by essentially automating the above procedure[3]. Calls to an equivalent of

[3]The author originally discovered this solution himself — only to find out later that some Smalltalk systems had the same mechanism in the past.

`System.currentTimeMillis()` (or special bytecodes, that would invoke a JVM-internal function) can be inserted into the original method bytecode, around the code section that the user wants to measure. The `return` statements within this code section, which are a headache if this procedure is performed manually, can be handled automatically. Then, the original method can be replaced with the patched one, such that, if necessary, the execution switches to it on the fly, as discussed in Section 5.2.1. This can be repeated an arbitrary number of times during the same session, allowing the user to change the profiled code section(s) and observe the results without interrupting the running application.

6 Conclusions

We have described the technology for runtime evolution of Java applications (dynamic class redefinition), that allows developers to modify running Java applications. The technology is being developed for the HotSpot JVM, Sun's main general-purpose JVM. We have presented the features of this JVM that are important for our work, and their effect on the runtime evolution implementation. We then introduced the plan of staged implementation of our technology, where each stage corresponds to some, relatively consistent, level of functionality. Our technology allows developers to evolve classes whose methods are currently active, and we have suggested several possible policies for dealing with such methods. These policies are independent of the functionality levels corresponding to the implementation stages. The same is true about the support for instance conversion, which can be introduced at any stage except the first one.

In the first stage we allow the developers to modify only method bodies. We support only one policy for dealing with active methods: "active invocations of old methods complete, new calls go to new methods". The implementation is completely operational now, and will be included in the forthcoming Java 2 Platform release.

In the second stage, which is now close to completion, we support less restrictive, though only binary compatible, changes to classes. In the third stage, we are planning to support all possible changes, including the binary incompatible ones, provided that they are type safe in the context of the current running application.

We have also presented our thoughts on the following aspects of the future work: instance conversion, implementation of multiple policies for dealing with active methods, and applying dynamic class redefinition technology for fine-grain profiling.

Making small changes to running applications, such as fixing minor bugs or adding trace printing statements, is easy, in the sense that their results are predictable enough. Once a JVM supports more serious changes, the biggest problem, in our opinion, will be to ensure that the transition from the old code to the new one happens smoothly, without runtime errors or undesirable effects. It is hard to see at present what mechanisms can aid the developer in this respect, but we believe that some form of automatic verification of change correctness will be crucial to make the technology for serious runtime application changes widely accepted.

References

[1] M.P. Atkinson. Persistence and Java — a Balancing Act. In *Objects and Databases. International Symposium, Sophia Antipolis, France, June 2000. Revised Papers*, volume 1944 of *LNCS*. Springer-Verlag, 2000.

[2] M.P. Atkinson and M.J. Jordan. A Review of the Rationale and Architectures of PJama: a Durable, Flexible, Evolvable and Scalable Orthogonally Persistent Programming Platform. Technical report, Sun Microsystems Laboratories Inc and Department of Computing Science, University of Glasgow, 901, San Antonio Road, Palo Alto, CA 94303, USA and Glasgow G12 8QQ, Scotland, 2000. TR-2000-90, http://www.sun.com/research/forest/COM.Sun.Labs.Forest.doc.pjama_review.abs.html.

[3] M. Dmitriev. *Safe Evolution of Large and Long-Lived Java Applications*. PhD thesis, Department of Computing Science, University of Glasgow, Glasgow G12 8QQ, Scotland, 2001. Available at http://www.dcs.gla.ac.uk/∼misha/papers.

[4] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification, Second Edition*. Addison-Wesley, June 2000.

[5] U. Hölzle. *Adaptive Optimization for Self: Reconciling High Performance with Exploratory Programming*. PhD thesis, Stanford University, March 1995. Also published as Sun Microsystems Laboratories Technical Report SMLI TR-95-35.

[6] IBM Inc. VisualAge for Java. http://www-4.ibm.com/software/ad/vajava/, 2000.

[7] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification, Second Edition*. Addison-Wesley, 1999.

[8] D. Pescovitz. Monsters in a Box. *Wired*, 8(12):341 – 347, 2000.

[9] Sun Microsystems Inc. Java Platform Debugger Architecture. http://java.sun.com/products/jpda/doc/architecture.html, 2000.

[10] D. White and A. Garthwaite. The GC Interface in the EVM. Technical Report TR-98-67, Sun Microsystems Laboratories Inc, 901, San Antonio Road, Palo Alto, CA 94303, USA, 1998.