Dynamic Code Evolution for Java [*]

Thomas Würthinger[†] Christian Wimmer[§] Lukas Stadler[†]

[†]Johannes Kepler University Linz, Austria [§]University of California, Irvine
{wuerthinger, stadler}@ssw.jku.at cwimmer@uci.edu

Abstract

Dynamic code evolution is a technique to update a program while it is running. In an object-oriented language such as Java, this can be seen as replacing a set of classes by new versions. We modified an existing high-performance virtual machine to allow arbitrary changes to the definition of loaded classes. Besides adding and deleting fields and methods, we also allow any kind of changes to the class and interface hierarchy. Our approach focuses on increasing developer productivity during debugging. Changes can be applied at any point a Java program can be suspended. The evaluation section shows that our modifications to the virtual machine have no negative performance impact on normal program execution. The fast in-place instance update algorithm ensures that the performance characteristics of a change are comparable with performing a full garbage collection run. Standard Java development environments are capable of using the code evolution features of our modified virtual machine, so no additional tools are required.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors—Run-time environments

General Terms Algorithms, Languages, Evolution

Keywords Java, virtual machine, class hierarchy, runtime evolution, dynamic software updating, garbage collection

1. Introduction

Updating the code of a running program has been investigated early in programming history [12]. With the introduction of executing programs in virtual machines (VM), the possibilities for dynamic code evolution increased because of the additional layer between the executing program and the hardware. Nevertheless, support for this feature in current production-quality VMs is limited. The ability to evolve the code of a running program has advantages in several areas. We distinguish four main applications of dynamic code evolution and their specific requirements:

Debugging. When a developer frequently makes small changes to an application with a long startup time, dynamic code evolution significantly increases productivity. After modifying and compiling the program, the developer can resume it directly from where it was suspended instead of stopping and restarting it. For example, modifying the action performed by a button in a graphical user interface does no longer mean that the whole program has to be closed. The main requirement for success is that the code evolution step can be carried out at any time and the programmer does not need to perform additional work, e.g., provide transformation methods for converting between the old and new version of object instances or specify update points. The performance of program execution is also important as an application could do intensive calculations before the first breakpoint is reached or between two consecutive breakpoints.

Server Applications. Critical applications that must not be shutdown can only be updated to the next version using dynamic code evolution. The focus lies on safety and correctness of an update. We believe that this can only be achieved by designing an application with code evolution in mind and restricting updates to certain predefined points. The server applications must not be slowed down before or after performing the code evolution.

Dynamic Languages. There are various efforts to run dynamic languages on statically typed VMs (see for example [33]). Dynamic code evolution is a common feature of dynamic languages. Having it as a mechanism of the VM simplifies the implementation of dynamic languages. The requirement here is that small incremental changes, e.g., adding a field or method, can be carried out fast.

Dynamic AOP. Dynamic code evolution is also a feature relevant for aspect oriented programming (AOP). There are several dynamic AOP tools that use the current limited possibilities of the Java HotSpot™ VM for dynamic code evolution [5, 46]. Those tools can immediately benefit from enhanced code evolution facilities.

Our approach to dynamic code evolution focuses on improving developer productivity during debugging. It can carry out the change at any point a Java program can be suspended, i.e., at any point a developer can set a breakpoint. Additionally, a Java program can be paused by requesting that every thread stops at the next *safepoint*. These points are usually used to suspend all threads before a garbage collection run. The Java VM guarantees that at any point during program execution, all threads will reach the next safepoint within a finite time span. Once the VM is suspended, code evolution can be performed.

The strong need for advanced dynamic code evolution features is also expressed by the votes for enhancement requests for the Java HotSpot™ VM: The request for improving the current support for code evolution that allows only swapping method bodies is within the top five enhancement requests. Additionally, the increased productivity through fast code modifications is considered one of the advantages of dynamic languages compared to statically

[*] This work was supported by Oracle.

PPPJ '10, September 15–17, 2010, Vienna, Austria.
Copyright © 2010 ACM 978-1-4503-0269-2. . . $10.00

typed languages such as Java. Enabling dynamic code evolution for Java VMs gives this advantage to the Java programming language too.

The main contributions of this paper are:

- We describe the modifications necessary for dynamic code evolution in a production-quality VM.
- Our approach allows arbitrary changes, including the modification of subtype relationships. Nevertheless, it does not introduce any indirections and is without performance loss before or after the code evolution step.
- We allow co-existence of old and new code. An update is possible at any point a Java program can be suspended.
- Our modified version of the Java HotSpot^{TM} VM can be used from within any Java IDE that uses the standard Java Debug Wire Protocol (JDWP), e.g., NetBeans or Eclipse.

2. Levels of Code Evolution

Several classifications of runtime changes have been published [20, 38]. From the aspect of implementation complexity and impact on Java semantics, we propose the distinction of four levels of code evolution as shown in Figure 1:

Swapping Method Bodies: Replacing the bytecodes of a Java method is the simplest possible change. No other bytecodes or type information data depend on the actual implementation of a method. Therefore, this change can be done in isolation from the rest of the system. The current version of the Java HotSpot^{TM} VM allows this kind of change.

Adding or Removing Methods: When changing the set of methods of a class, the virtual method table that is used for dynamic dispatch needs to be modified. Additionally, a change in a class can have an impact on the virtual method table of a subclass (see Section 3.2). The virtual method table indexes of methods may change and make machine code that contains fixed encodings of them invalid (see Section 3.6). Machine code can also contain static linkings to existing methods that must be invalidated or recalculated.

Adding or Removing Fields: Until this level, the changes only affected the metadata of the VM. Now the object instances need to be modified according to the changes in their class or superclasses. The VM needs to convert the old version of an object to a new version that can have different fields and a different size. We use a modified version of the mark-and-compact garbage collector in order to change the object layout (see Section 3.5). Similarly to virtual method table indexes, field offsets are used in various places in the interpreter and in the compiled machine code. They need to be correctly adjusted or invalidated.

Adding or Removing Supertypes: Changing the set of declared supertypes of a class is the most complex dynamic code evolution change for Java. For a class, this can mean changes to its methods as well as its fields. Additionally, the metadata of the class needs to be modified in order to reflect the new supertype relationships.

When a developer changes the signature of a method or the type or name of a field, the VM sees the change as two operations: a member being added and another being deleted from the class. Modifications to interfaces can be treated in a similar way as modifications to classes. Adding or removing an interface method affects subinterfaces and the interface tables of classes which implement that interface, but has no impact on instances. Changes to the set of superinterfaces have a similar effect.

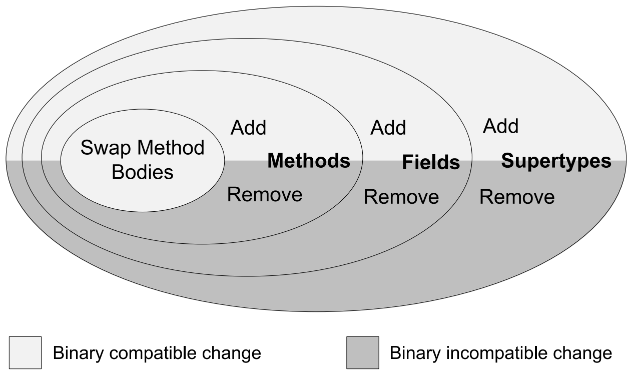


Figure 1. Levels of code evolution.

Another possible kind of change in a Java class is modifying the set of static fields or static methods. This does not affect subclasses or instances, but may invalidate existing code, e.g., when it contains static field offsets. Additionally, a code evolution algorithm needs to decide how to initialize the static fields: either run the static initializer of the new class or copy values from the static fields of the old class.

Changes to Java programs can also be classified according to whether they maintain binary compatibility between program versions or not [8]. The light grey areas of Figure 1 represent binary compatible changes; the dark grey areas indicate binary incompatible changes. With binary compatible changes, the validity of *old code* is not affected. We define old code as bytecodes of methods that have either been deleted or replaced by a different method in the new version of the program. When an update is performed at an arbitrary point, a Java thread can be in the middle of executing such a method. Therefore, old code can still be executed after performing the code evolution step.

Binary incompatible changes to a Java program may break old code. The semantics of instructions that were valid in the old version but are no longer valid in the new version of the program are not clear as neither the Java language specification [15] nor the Java VM specification [29] takes code evolution into account. We classify binary incompatible changes as follows:

Removing Fields or Methods: The bytecodes of deleted or replaced methods can contain references to class members that no longer exist in the new version of the program. During this continued execution of old code, the VM might reach those bytecodes and needs to decide what to do when deleted methods are called or deleted fields are accessed.

Removing Supertypes: When narrowing the type of a class, an important invariant during Java program execution can be violated: The static and dynamic type of a variable no longer necessarily have a subtype relationship. Additionally, the receiver object of a dynamic call need no longer be compatible with the class of the called method.

Section 4 gives a description of how we handle the case where old code is not compatible with new types and in Section 7 we discuss possible future work that targets this problem.

3. Implementation

We implemented our approach as a modification to the Java HotSpot^{TM} VM [34], a high-performance VM with an interpreter and two just-in-time compilers (the *client compiler* [26] and the *server compiler* [36]). The implementation is based on the existing mechanism for swapping method bodies [7] and extends it to allow arbitrary changes to loaded types. Our approach focuses on implementing code evolution in an existing VM while keeping the necessary changes small. In particular, we do not modify any of the just-in-time compilers or the interpreter. Our changes affect the garbage collector, the system dictionary, and the class metadata. They are however small and do not influence the VM during normal program execution.

Figure 2 gives an overview of the modifications to the VM that are described in the following subsections. The code evolution is triggered by a Java Debug Wire Protocol (JDWP) command [32]. First, the algorithm finds all affected classes and sorts them based on their subtype relationships. Then, the new classes are loaded and added to the type universe of the VM forming a side universe. A modified full garbage collection performs the actual version change. After invalidating state that is no longer consistent with the new class versions, the VM continues executing the program.

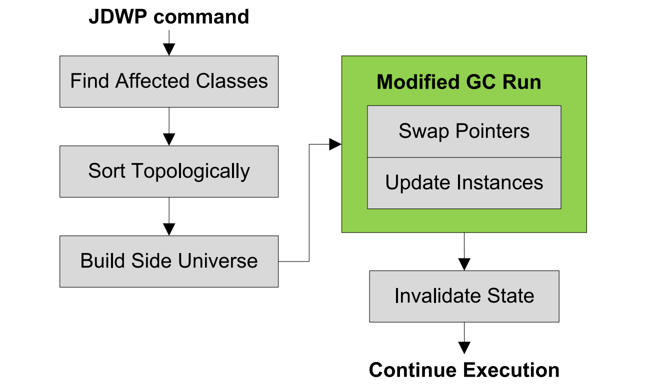


Figure 2. Steps performed by the code evolution algorithm.

3.1 Class Redefinition Command

We use the existing JDWP command for class redefinition to trigger dynamic code evolution. JDWP is is a specification for the interface between a Java VM and a Java debugger. Therefore, our modified VM is immediately usable from within standard Java development environments that use the JDWP protocol for debugging Java applications such as NetBeans or Eclipse.

The command requires that all redefined classes are already loaded in the VM. If a class is not yet loaded, redefinition is not necessary. The new version can be loaded as the initial version of the class. For each class, a number identifying the class and an array with the new class bytes is transmitted. Our modified VM implements this command exactly based on its specification and does not need additional information to perform the code evolution.

The first steps of the redefinition (described in the next three sections) can be done in parallel with normal program execution. Only the subsequent garbage collection run that performs the instance updates needs to stop all running Java threads. We use the same safepoint mechanism as the garbage collector to suspend all active threads.

3.2 Finding Affected Types

When applying more advanced changes than just swapping method bodies, classes can be indirectly affected by the redefinition step. A field added to a class is implicitly also added to all its subclasses. Adding a method to a class can have effects on the virtual method tables of its subclasses.

Therefore, the algorithm needs to extend the set of redefined classes by all their subtypes. Figure 3 gives an example with three

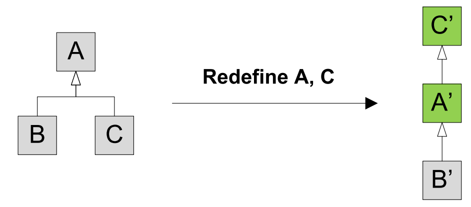


Figure 3. Code evolution example changing the subtype relationship between class `A` and `C`.

classes `A`, `B`, and `C`. Class `A` and `C` are redefined, but this also affects class `B` as it is a subclass of `A`. Class `B` is added to the set of redefined classes and is replaced by `B'`, which has the same class file data as `B`, but possibly different properties inherited from its superclasses. We need to fully reload `B`, because its metadata including the virtual method tables need to be initialized based on the new supertype.

The same rule applies when redefining interfaces. All subinterfaces and also all classes implementing the interface need to be redefined, because adding or removing methods of the superinterface has effects on their interface method tables.

3.3 Ordering the Types

The redefinition command does not specify an order in which the classes must be redefined. From the user’s perspective, the classes must be swapped atomically. Our algorithm does a topological sorting of the classes based on their subtype relationships. A class or interface always needs to be redefined before its subtypes can be redefined. The new version of a class could be incompatible with the old version of its superclass. In that case, class loading only succeeds if the superclass was already replaced by its new version.

In order to make changes to the class hierarchy possible, we need to order the types based on their relationship *after* the code evolution step and cannot use the information about their current relationship. Subtype relationship information is available in the VM only after a class has been loaded. Therefore, we parse parts of the class files prior to class loading in order to find out about the new subtype relationships. In the example of Figure 3, we need to first redefine `C` to `C'` and subsequently `A` to `A'`, because in the new version of the program `A` is a subclass of `C`.

3.4 Building a Side Universe

We keep both the old and the new classes in the system. This is necessary to be able to keep executing old code that depends on properties of the old class. It would also open the possibility to keep old and new instances in parallel on the heap. Additionally, it is the only way to solve the problem of cyclic dependencies between code evolution changes, e.g., when one change requires class `A` to be redefined before `B`, but the other one `B` to be redefined before `A`: While adding the new classes, the type universe is always kept in a consistent state, because we build a separate side branch for the new classes. Therefore, the old version of a class will not affect the loading and verification of the new version of another class.

The Java HotSpot^{TM} VM maintains a system dictionary to look up classes based on their name and class loader. We replace the entry for the old class with the entry for the new class immediately after loading the new class. The pre-calculated order in which we redefine classes ensures that the side universe is created correctly. When we load class `A` in the example of Figure 3, the lookup for class `C` returns `C'`, because class `C` was redefined before `A`. The VM copies the value of static fields of the old class to the static fields the new class if both name and signature match. We do not execute the static class initializer of the new class. Figure 4 shows the state of the universe after building the side universe for the new class versions.

We keep the different versions of the same class connected in a doubly linked list. This helps navigating through the versions during garbage collection. The system dictionary, however, always contains just a reference to the latest version of a class.

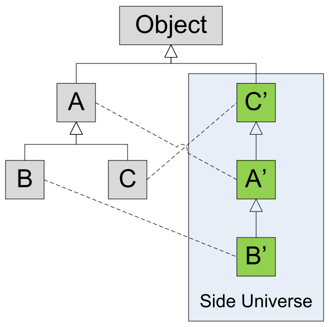


Figure 4. The new state of the type universe after code evolution.

3.5 Garbage Collector Adjustments

The core part of the redefinition algorithm is implemented as a modification of the mark-and-compact garbage collection algorithm. This algorithm calculates a *forward pointer* for each live heap object, which points to the address of the object after the heap compaction. In the following heap traversal, all references are adjusted to point to this newly calculated address instead of the referenced object itself. In the final compaction phase, the objects are moved to their new addresses.

Changing references of the old class to references of the new class can be done during the pointer adjustment phase. Additionally, updating the instances is similar to the operation performed in the compaction phase. Therefore, implementing code evolution as a modification of the garbage collector leads to code reuse and high performance for updating the object instances. The approach also makes sure that we do not need any indirections or additional data structures to enable code evolution. The following two subsections contain descriptions of the two major modifications.

3.5.1 Swapping Pointers.

When updating a class `C` to `C'` we must ensure that all instances of class `C` are updated to be instances of class `C'`. The instance of an object on the heap contains a reference to its class. The Java HotSpot^{TM} VM does not keep track of the instances of a given class, therefore a heap traversal is necessary to find all existing instances. Additionally, other parts of the system (e.g., native code) can have references to the old class that need to be updated too.

Figure 5 shows the steps performed by the garbage collector as well as our modifications. Assume that the initial heap contains an object `x` of class `A` as well as a new version of class `A` denoted by `A'`. First, the collector computes the addresses of all alive objects after compaction and installs a forward pointer into every object that points to the new address. In the pointer adjustment phase, the pointer fields and the class pointers of every object are modified to point to the new addresses after compaction. We intercept this step to ensure that pointers to the old class are adjusted to the destination address of the new class. This ensures that after the compaction phase, every previous reference to the old class is converted to a reference to the new class.

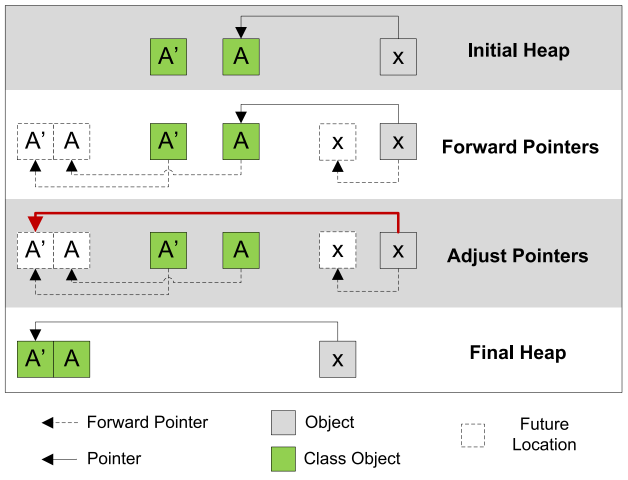


Figure 5. Swapping pointers during garbage collection.

3.5.2 Updating Instances.

For updating instances, we need a strategy how to initialize the fields of the new instance. We have a simple algorithm that matches fields if their name and type are the same. For the matching fields, we copy the values from the old instance to the new instance. All other fields are initialized with `0`, `null`, or `false`.

With this approach, we can efficiently encode the memory operations necessary for an instance update (filling an area with zero or copying bytes from the old version of the object). The information is calculated once per class and temporarily attached to the class meta object. The modified garbage collector reads the information and performs the memory copy or clear operations for each instance. This makes instance updates faster compared to other approaches that work with custom transformation methods for converting between the old and new version of object instances. We believe that the programmer wants to provide as little additional input as possible during debugging and so the lost flexibility compared to transformation methods is balanced by the ease of use.

As the update from the old to the new version of an instance is carried out during the compaction phase of the garbage collector, we do not need any additional memory for co-existence of old and new instances. The old version of an instance is deprecated immediately after the new version has been created and filled with values.

We adjusted the forward pointer calculation algorithm in order to consider the new object sizes. An additional modification of the garbage collector is necessary to support increasing object sizes. In this case, the instances are not necessarily always copied to lower addresses, which is a necessary condition for the compaction phase of a mark-and-compact garbage collector. Therefore, every instance that would be copied to a higher memory address must first be rescued to a side buffer. Otherwise the garbage would overwrite objects that are not yet copied and destroy their field values. After all instances have been either copied or rescued, the side buffer is processed and used to initialize the new versions of the rescued instances. To reduce the number of objects that need to be copied to a side buffer, our forward pointer calculation algorithm places objects that are copied to the side buffer automatically at the very end of the heap. This makes space for other objects to increase their size while still being copied to lower addresses.

In Figure 6, the size of `x` is increased and therefore the object `x` at its destination address overlaps other objects such as `y` and overwrite their contents before they are copied. Our modified forward pointer algorithm detects that x is an instance that needs to be rescued and therefore places it at the end of the heap. This makes free space for the destination addresses of the instances y and z such that they need not be rescued. The optimization to place rescued objects at the end of the heap significantly reduces the number of rescued objects and therefore the necessary size of the side buffer. In the compaction phase, the object x is copied to the side buffer, objects y and z are processed normally. Afterwards, the new version of x is constructed based on the data of the old version in the side buffer.

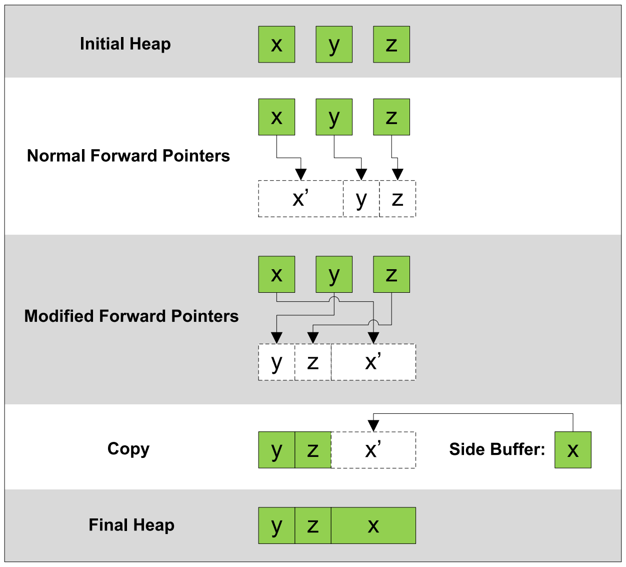


Figure 6. Increasing object size during garbage collection.

3.6 State Invalidation

The changes performed by code evolution violate several invariants in the VM. The Java HotSpot^{TM} VM was not developed with code evolution in mind and makes assumptions that no longer hold, e.g., that a field offset never changes. In this section we outline different subsystems of the VM that need changes in order to prevent unexpected failures due to broken assumptions.

3.6.1 Compiled Code.

Machine code generated by the just-in-time compiler before code evolution needs to be checked for validity. The most obvious potentially invalid information are virtual method table indexes and field offsets. Additionally, assumptions about the class hierarchy (e.g., whether a class is a leaf class) or calls (e.g., whether a call can be statically bound) become invalid.

The Java HotSpot^{TM} VM has a built-in mechanism to discard the optimized machine code of a method, called *deoptimization* [24]. If there is an activation of the method on the stack, the stack frame is converted to an interpreter frame and execution is continued in the interpreter. Additionally, the machine code is made *non entrant* by guarding the entry point with a jump to the interpreter. We can use it to deoptimize all compiled methods to make sure that no machine code produced with wrong assumptions is executed. Analyzing the assumptions made for compiled methods could reduce the amount of machine code that has to be invalidated. However, the evaluation section shows that the time necessary to recompile frequently executed methods is low.

3.6.2 Constant Pool Cache.

The Java HotSpot^{TM} VM maintains a cached version of the constant pool for a class. This significantly increases the execution speed of the interpreter compared to working with the original constant pool entries stored in the Java class files. The original entries only contain symbolic references to fields, methods, and classes, while the cached entries contain direct references to metadata objects. The entries relevant to code evolution are field entries (the offset of a field is cached) and method entries (for a statically bound call a pointer to the method meta object, for a dynamically bound call the virtual method table index is cached). We iterate over the constant pool cache entries and clear those entries that are possibly affected by code evolution (i.e., that correspond to members of a redefined class). When the interpreter reaches a cleared entry, it is resolved again. The lookup in the system dictionary automatically returns the new version of the class and therefore the entry is reinitialized with the correct field offset or method destination information.

3.6.3 Class Pointer Values.

Several data structures in the Java HotSpot^{TM} VM depend on the actual addresses of the class meta objects, e.g., a hash table mapping from classes to JDWP objects. We need to make sure that these data structures are reinitialized after code evolution. While class objects may also be moved during a normal garbage collection run, our pointer swapping potentially also changes the order of two class objects on the heap. The just-in-time compiler uses a binary search array for compiler interface objects that depends on the order of the class objects and therefore must be resorted after a code evolution step.

4. Binary Incompatible Changes

Changes are binary incompatible if they may break old code. This section describes how we currently handle the two classes of binary incompatible changes that we introduced in Section 2. In Section 7 we discuss different solutions that we want to explore as future work.

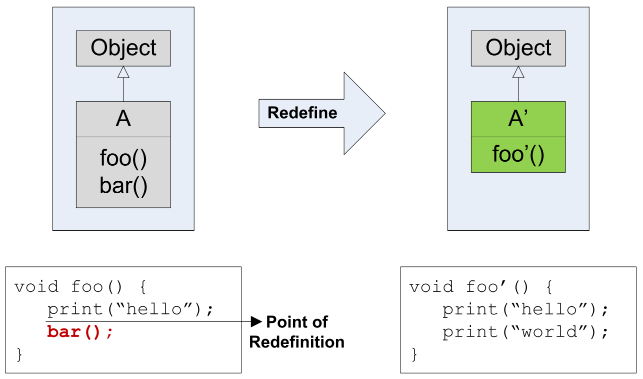
4.1 Deleted Fields and Methods

As long as only method bodies are swapped or fields and methods are added to classes, the old code can execute normally. It does neither call new methods nor access new fields. However, when a field or method is deleted, old code is possibly no longer valid. In our system, old code may still be executed when old methods are on the stack, so it can happen that old code accesses a deleted field or tries to call a deleted method.

Figure 7 shows an example for this case. The program is paused in method `foo` between the calls to `print` and `bar`. Method `foo` is redefined to a new version `foo'` while method `bar` is deleted. Subsequent calls to method `foo` immediately target the new code, but the old activation of `foo` continues to execute in the old code. Therefore, it reaches the call to the deleted method `bar`.

The new version of `foo` is correct because it no longer calls `bar`. It is possible to develop heuristics for translating from the stack values and bytecode position in the old method to new stack values and a new bytecode position in the new code. In the general case, however, it is impossible to find a match that is intuitive for developers.

In our current implementation, the old method continues execution in the interpreter. When it reaches the bytecode for the call to `bar`, the reference to `bar` needs to be resolved (because we cleared the constant pool cache during the redefinition step, see Section 3.6). The resolution fails to find the method and throws a `NoSuchMethodException`.



```
void foo() {
   print("hello");
   bar();
}
```

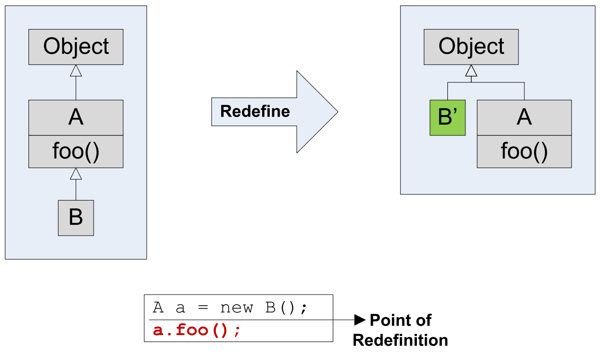
```
void foo'() {
   print("hello");
   print("world");
}
```

Figure 7. Deleting a method using dynamic code evolution.

4.2 Type Narrowing

When the set of implemented interfaces or the set of all superclasses of a class increases, old code can execute as normal. It does not use instances of the class as instances of their added interfaces or supertypes, but executes as before. On the contrary, when the set is narrowed, old code is possibly no longer valid.

Figure 8 gives an example for this case. Class `B` is redefined to no longer extend class `A` but directly inherit from `Object`. Now, instances of `B` must no longer be treated as instances of `A`. There could already be variables of type `A` referencing instances of `B` as shown in the code listing. After code evolution, the values of such variables become invalid, because `B` is no longer a subtype of `A`. In the listing of Figure 8, the call `a.foo()` no longer makes sense, because `B` does neither declare nor inherit a method `foo`.



```
A a = new B();
a.foo();
```

Figure 8. Type hierarchy change using dynamic code evolution.

The current version of our system correctly performs the code evolution step, but the call to `foo` leads to termination of the VM. We believe that this is an acceptable solution when code evolution is used in the context of debugging. We discuss possible different solutions in Section 7.

5. Evaluation

We evaluate our implementation by looking at it from three sides: First, we discuss our support for different levels of code evolution. Second, we show that our modified VM produces equal benchmark results than the unmodified baseline. After performing a code evolution step the original peak performance is reached again. Finally, we present timing results for micro benchmarks to discuss the performance characteristics of our garbage collector modifications.

5.1 Functional Evaluation

Our implementation supports any possible modifications to a set classes. When the changes are binary compatible, the code is guaranteed to execute as expected. Binary incompatible changes are performed, but depending on the program state (e.g., which methods are currently active), they may lead to exceptions being thrown (e.g., when a deleted field or method is accessed by old code) or to VM termination (e.g., when the dynamic type of a variable does no longer match its static type). Such exceptions, however, are rare, because deleting methods and fields is less common than adding them, and even if members are deleted, it is unlikely that a method that uses them is active just when the program is redefined. In all cases, except when removing a super type, the continued execution of the Java program is fully compliant with standard Java semantics. Table 1 gives an overview of the supported changes classified as discussed in Section 2.

Method	Possible problems after resume
Swap Method Body	
Add Method	
Remove Method	`NoSuchMethodError`
Add Field	
Remove Field	`NoSuchFieldError`
Add Supertype	
Remove Supertype	can lead to VM termination

Table 1. Supported code evolution changes.

When debugging an application, possible problems after resuming the program are more acceptable than when updating a server application. The worst case scenario is that the developer needs to restart the application. Without code evolution, this would be necessary anyway. The semantics of the problematic instructions are not clear as the Java standard was not designed with code evolution in mind. Therefore, we believe that throwing an exception or terminating the VM is an acceptable behavior and leads to less confusion compared to trying to hide and absorb the problem.

Since version 1.4 of Java, the JPDA (Java Platform Debugger Architecture) defines commands for class redefinition. A VM specifies three flags to inform the debugger about the code evolution capabilities: `canRedefineClasses` if class redefinition is possible at all, `canAddMethod` if methods can be added to classes, and `canUnrestrictedlyRedefineClasses` if arbitrary changes to classes are possible. To the best of our knowledge, our modified version of the Java HotSpot^{TM} VM is the first VM that can return `true` for all three flags. Based on the considerations in Section 2 about the implementation complexity of changes in the VM, we propose a more fine grained distinction between different levels of code evolution. The step between adding methods and allowing arbitrary changes is too coarse grained.

It is difficult to measure the usage characteristics of code evolution as it heavily depends on the application domain and also on the developer behavior. Gustavsson [20] published a case study in which the updates to a web server between different versions are examined. The result is that 37% of the modifications only redefine method bodies, 16% only add or remove methods, 33% are arbitrary code evolution changes, and 14% are changes that cannot be performed, e.g., because of code that never becomes inactive or a need to change things outside the VM. Our implementation can therefore increase the percentage of possible changes in this case study from 37% to 86%.

5.2 Effects on Normal Execution

We use two benchmarks selected from the DaCapo benchmark suite [4] with different characteristics regarding the warm-up phase. We show that our modifications to the VM have no effects on normal program execution. Additionally, after a code evolution step, the application reaches the peak performance again. We measure the times of 24 subsequent benchmark runs within the same VM. A garbage collection between two subsequent runs is performed in order to reduce the noise introduced by the garbage collector. In our modified VM, we redefine a single class between the 8th and 9th and between the 16th and 17th run, causing the VM to discard previously compiled machine code as described in Section 3.6.

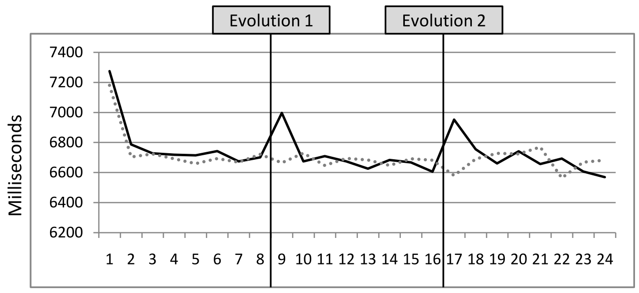
Our implementation is currently based on an early access development build of the Java HotSpot^{TM} VM for Java 7 (build 1.7.0-ea-b36). The baseline run is performed by the unmodified version of the Java HotSpot^{TM} VM. We use the same command line flags for both VMs. All performance measurements were performed on an Intel Core^{TM}2 Quad CPU with 2.40 GHz per core and 8 GByte memory. The operating system is a 32-bit version of Windows Vista.

The heap size is specified with 1 GByte and the client compiler is used as the just-in-time compiler. Additionally, we use the following command line flag:

```
-Xrunjdwp:transport=dt_socket,
server=y,address=4000,suspend=n
```

This starts a JDWP agent for receiving JDWP commands. The agent is used for debugging the Java program running in the VM. Later, we can connect to this agent and send the commands for triggering code evolution.

We executed the test setup 20 times and calculated the mean. Figure 9 shows the results for the unmodified reference VM without code evolution (dotted line) and our modified VM (solid line) when executing two DaCapo benchmarks.



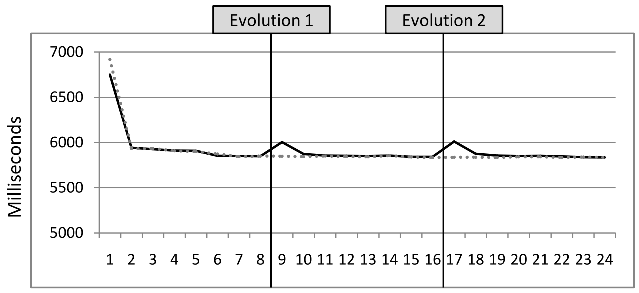


Figure 9. Executing 24 runs of the `bloat` (top) and the `chart` (bottom) benchmark. The dotted line represents an unmodified reference VM, the continuous line our code evolution VM.

The performance characteristics after a code evolution step are similar to the warm-up phase. The first run after the code evolution is significantly slower. This is because the VM needs some time to recompile the frequently executed methods again. In the second run, the performance difference is hardly measurable and subsequent runs do not show any difference. As the profiling information is reused, the first run after code evolution is faster than the first run overall. For the `chart` benchmark, the slowdown of the first run compared to the peak performance is about 15%, but the slowdown of the first run after code evolution is only about 3%.

5.3 Micro Benchmarks

To measure the performance of instance updates and our garbage collector adjustments, we use three micro benchmarks in which we increase, decrease, or reorder the fields of a class and update all instances of this class to their new version. We compare the performance of this updating garbage collection to the performance of a normal garbage collection run. Table 2 describes the different class versions used for the benchmarks. It has three `int` fields and three `Object` fields resulting in a total object size (including 8 bytes object header) of 32 bytes. The three benchmark configurations are:

Increase	Decrease	Reorder
`class C' {` `int i1;` `int i2;` `int i3;` `Object o1;` `Object o2;` `Object o3;` `Object o4;` `}`	`class C' {` `int i1;` `int i2;` `int i3;` `Object o1;` `}`	`class C' {` `int i3;` `int i1;` `int i2;` `Object o3;` `Object o1;` `Object o2;` `}`
40 bytes	24 bytes	32 bytes

Table 2. The three redefined versions of class `C`

Increase: An object field is added to the class resulting in an increased instance size of 40 bytes (because the size of an object is rounded upto 8 bytes). Therefore, the changed objects need 25% more heap area.

Decrease: Two object fields are removed resulting in a decreased instance size of 24 bytes. Therefore, the changed objects need 25% less heap area.

Reorder: All fields of the object are reordered to be in a different position than before. The size of the heap area of updated objects remains unchanged.

We create a total of 4,000,000 objects, resulting in an approximate size in memory of 128 MByte. For our benchmarks, we create fractions between 0% and 100% of the objects as instances of the redefined class. The rest of the objects are created as instances of another class with the same fields, but this class is kept unmodified. The baseline is a full garbage collection without code evolution. As there are no dead objects on the heap, this garbage collection run only marks all objects, but does not need to copy memory in the compaction phase. We execute each configuration 10 times and report the mean of the runs. Figure 10 shows the results.

The slowdown of the code evolution garbage collection run when no objects are affected is about 35%. This includes issuing the JDWP command, loading the new classes, and also the additional work needed during garbage collection for swapping the pointers of the old and new class.

Increasing the size of all objects on the heap needs about three times more time than the no-load garbage collection run. About 20% of the objects need to be copied to a rescue buffer, because one object (32 bytes) makes room for four objects to increase their size

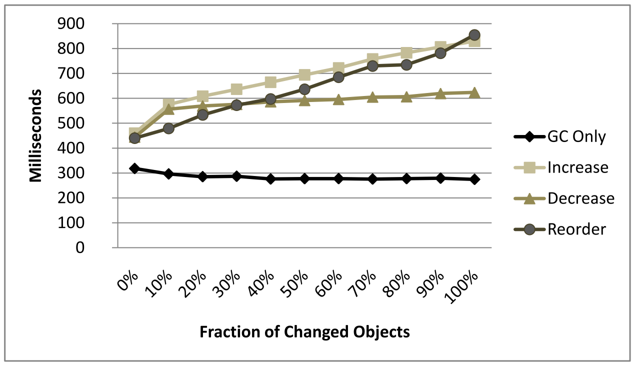


Figure 10. Micro benchmark results for changing the fields of a class compared against a garbage collection run.

by 8 bytes each. The performance of this benchmark is improved when there are dead objects at the beginning of the heap. The dead objects provide space for instances to increase their size and lead to less objects being copied to the rescue buffer.

Reordering the fields of all objects (and therefore copying the objects field by field instead of as a whole) is a likewise costly scenario. No objects are dead and therefore the instances must be updated in place. We need to make a copy of each object and then copy the field values one-by-one. This is about three times slower than a normal garbage collection run.

Redefining a class such that all object sizes decrease needs about twice the time of a normal garbage collection run. In contrast to the increased size, we do not need a rescue buffer. In the compaction phase, the objects are copied to their new location skipping deleted fields.

6. Related Work

6.1 General Discussions

Several classifications of runtime changes of programs have been published [20, 38]. Ketfi et al. outline the adaptation of component-based applications [25]. Ebraert et al. present a general discussions on the problems and pitfalls of software evolution [11] and examine dynamic runtime changes from the point of view of the user [10].

Kramer and Magee investigate the problem of how to design applications to allow a consistent state transfer between new and old programs [27, 28]. Vandewoude et al. extend their work and introduce the notion of tranquillity [45]. Runtime evolution in the context of aspect-oriented programming is discussed by Gustavsson et al. [21].

6.2 Procedural Languages

Early work on dynamic code evolution was done by Fabry [12]. The dynamic code aspect is achieved by jump indirection to procedures. Data conversion routines can be specified and the old and new version of a module can execute in parallel. Lee and Cook implemented a dynamic modification system for the StarMod language, which is an extension of Modula-2. Their system is called DYMOS [6]. It includes a command interpreter that can perform update actions based on certain conditions, e.g., when certain procedures are inactive.

Frieder and Segal developed a procedural dynamic updating system called PODUS [13, 40]. They require a binding table for methods that is updated accordingly. The concept of semantic dependency between methods is introduced.

Gupta implemented a hot code replacement mechanism for C programs on a Sun workstation based on state transfer [17, 18]. The system supports adding and deleting of functions. For adding global data, extra pointers must be declared in advance. Interprocedures are installed in case the return value or parameters of a method change. Those interprocedures map between a call from a method of the old program version to a new method of the new version. Gupta also developed a formal framework for program evolution [19]. Different formalizations of dynamic software updates were published by Bierman et al. [3] and Stoyle et al. [42].

Hicks et al. present a dynamic modification system for C-like languages [22]. They apply patches to the running program that are mostly automatically generated and can contain verification code. The patches contain the new code as well as the code needed to do the state transformation from the old to the new program. Neamtiu et al. extend this work to support the update of procedures with long-running loops and the ability to transform data of local variables [31].

In contrast to the work described in this subsection, our algorithm targets the challenges of code evolution for object-oriented languages. Also, we can leverage the advantages of dynamic compilation in a virtual machine and do not need to insert hooks into the statically compiled program. The modification of the garbage collector gives our algorithm maximum flexibility when performing instance updates.

6.3 Object-Oriented Languages

Hjalmtysson et al. [23] present an approach for dynamic classes in C++. They use the C++ template mechanism and proxy classes to realize the dynamic code evolution aspect. There has to be an interface definition for every dynamic class and it is only possible to change the implementation behind this interface. Therefore, it is not possible to add or remove any public members of a class and it is also not possible to change the class hierarchy.

The Common Lisp Object System (CLOS) [14, 41] includes the possibility to redefine a class. They allow the definition of transformer methods that update the instances from the old to the new version. A conceptional difference to our approach is that in CLOS the classes can only be redefined one-by-one, while we atomically redefine a set of classes. In the former case, the programmer is responsible for performing the redefinitions in the correct order. Additionally, our approach can deal with static typing and also with methods being defined as class members. This is not necessary for a class redefinition algorithm for the Lisp language. The class redefinition command for Java is designed for debugger use, while the method for redefining a class in Lisp can be called from user code. We believe that parts of our algorithm (e.g., the garbage collection modifications) can be used for efficiently implementing the CLOS class redefinition mechanism.

6.4 Java

There are various approaches of dynamic code evolution for Java based on proxy objects and bytecode rewriting [35, 37, 39, 44]. The main advantage of this approach is that it requires no change of the runtime system and can therefore be applied to any Java VM. Disadvantages are the performance penalty introduced by the indirection and the limitations of flexibility, e.g., changes of the class hierarchy are not supported. Additionally, support for triggering the code evolution from within development environments is not available or requires special plugins. Furthermore, the reflection facilities of the VM are affected (e.g. stack traces are obscure because of generated proxy methods). Gregersen et al. [16] advocate the idea of having a dynamic-update-enabled virtual machine and outline its advantages.

The project JDrums [1] is an implementation of a dynamic Java VM based on JDK 1.2. Its main limitations are that the just-in-time compiler must be disabled, active methods cannot be updated, and superclasses cannot be changed.

Malabarba et al. [30] present an implementation of code evolution based on JDK 1.2. They require that only the interpreter is used and cannot handle code evolution in the context of just-in-time compilation. In case of instance changes, they perform a global update using a mark-and-sweep algorithm during which all old version objects are converted to new version objects. In contrast to our solution, their modifications to version 1.2 of the JDK impose a significant performance penalty on normal program execution. Additionally, they allow only binary compatible changes and their VM uses object handles instead of direct object references.

Subramanian et al. [43] implemented code evolution for the Jikes RVM. They support adding and removing methods and fields, but do not support changes to the class hierarchy like we do. A special tool is used to generate update specification files. A transformation method is executed every time an object is converted between two versions. In contrast to our algorithm, their implementation is not focused on code evolution for debugging and can therefore neither perform an update at an arbitrary point nor be used from within standard Java development environments.

Dimitriev et al. [9] present a class evolution algorithm for the persistent object system for the Java programming language called PJama [2]. They introduce transformer methods for updating the stored objects. While some principles of class evolution also apply when updating the schema of an object-oriented data store, our main contribution is to perform dynamic class evolution and update heap objects while the user application is running.

The work most closely related to ours was done in attempt to apply PJama principles to runtime evolution of Java applications by Dmitriev [7, 8]. His implementation is part of the Java HotSpot^{TM} VM since JDK 1.4. While instance and schema changes are discussed in the thesis, the actual implementation is only capable of swapping method bodies. It does change the original class metadata object and uses a constant pool merging algorithm to make sure that the new and the old methods can work with the constant pool associated with the old class. Our approach of using a side universe (see Section 3.4) requires less code and is the key design decision that enables us doing more complex changes to classes. Also, we do not need a custom classloader for loading the new class versions for validating the changes. Our code is based on the current implementation of swapping method bodies in the Java HotSpot^{TM} VM and significantly enhances it to allow arbitrary changes, including changes to the instance format and the class hierarchy.

7. Future Work

While the current focus of our implementation lies on debugging, we plan to extend it to the use case of updating server applications. Compared to debugging, the security requirements are higher, but the update may be delayed until a more suitable point in time. We want to extend the redefinition command in such a way that a *safe* update request can be made. This request is only performed if the problems introduced by binary incompatible changes of the redefined classes that we describe in Section 4 cannot occur. For deleted methods and fields, this means that we have to make sure that they are not accessed in the subsequent execution of the program. We need a reachability analysis starting from all redefined methods that are currently active. In case of hierarchy changes, where class `A` was formerly a superclass of `B`, we need to guarantee that no variable of static type `A` references an instance of type `B`. In order to check this, we plan to do a stack walk for all active threads as well as a garbage collection run to make sure that all fields will still reference objects of valid types after the update. In Java, the static type of a local variable is not encoded in the bytecodes and only temporarily calculated by the verifier, therefore we need to modify the verifier to cache this information.

8. Conclusions

We presented a technique for dynamic code evolution in Java and described its implementation for the Java HotSpot^{TM} VM. We allow arbitrary modifications to Java classes including changes to the class hierarchy. The update can be performed at any point during program execution, and our VM works with standard Java development environments. Old and new code may co-exist in the VM, and therefore our approach allows redefining methods that are currently active. We discussed the problems introduced by binary incompatible changes, our current solutions, as well as future plans to deal with them.

We showed that a production-quality VM can provide code evolution without losing performance during normal program execution. Our algorithm needs no additional indirections and works with the interpreter as well as with both just-in-time compilers of the Java HotSpot^{TM} VM. The code evolution step is combined with a modified garbage collection run and has similar performance characteristics. The focus of our VM modifications lies on improving debugging productivity.

Our approach is integrated in the latest development version of a high-performance VM. Only few modifications were necessary, and they were limited to specific components although the VM was not designed with code evolution in mind. The implementation is part of a larger effort to extend the Java HotSpot^{TM} VM with first-class support for languages other than Java (especially dynamic languages), called the *Da Vinci Machine Project* or *Multi-Language Virtual Machine Project* (MLVM) [33]. The source code and binaries of our modified virtual machine are available from `http://ssw.jku.at/dcevm`.

Acknowledgments

We would like to thank current and former members of the Java HotSpot^{TM} VM team at Oracle, especially Thomas Rodriguez, John Rose, David Cox, and Kenneth Russell, for their persistent support, for contributing many ideas and for helpful comments on all parts of the Java HotSpot^{TM} VM.

References

[1] J. Andersson and T. Ritzau. Dynamic code update in JDrums. In *Workshop on Software Engineering for Wearable and Pervasive Computing*, 2000.

[2] M. Atkinson, M. Jordan, L. Daynès, and S. Spence. Design issues for persistent Java: a type-safe, object-oriented, orthogonally persistent system. In *Proceedings of the ECOOP Workshop on Object-Oriented Databases*, 1996.

[3] G. Bierman, M. Hicks, P. Sewell, and G. Stoyle. Formalizing dynamic software updating. In *Proceedings of the International Workshop on Unanticipated Software Evolution*, 2003.

[4] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 169–190. ACM Press, 2006.

[5] S. Chiba, Y. Sato, and M. Tatsubori. Using HotSwap for implementing dynamic AOP systems. In *Workshop on Advancing the State-of-the-Art in Run-time Inspection*, 2003.

[6] R. P. Cook and I. Lee. Dymos: a dynamic modification system. In *Proceedings of the Symposium on High-Level Debugging*, pages 201–202. ACM Press, 1983.

[7] M. Dmitriev. *Safe Class and Data Evolution in Large and Long-Lived Java Applications*. PhD thesis, University of Glasgow, 2001.

[8] M. Dmitriev. Towards flexible and safe technology for runtime evolution of Java language applications. In *Proceedings of the Workshop on Engineering Complex Object-Oriented Systems for Evolution*, 2001.

[9] M. Dmitriev and M. Atkinson. Evolutionary data conversion in the PJama persistent language. In *Proceedings of the Workshop on Object-Oriented Technology*, pages 25–36. Springer-Verlag, 1999.

[10] P. Ebraert, T. D'Hondt, Y. Vandewoude, and Y. Berbers. User-centric dynamic evolution. In *Proceedings of the International ERCIM Workshop on Software Evolution*, 2006.

[11] P. Ebraert, Y. Vandewoude, T. D'Hondt, and Y. Berbers. Pitfalls in unanticipated dynamic software evolution. In *Proceedings of the Workshop on Reflection, AOP and Meta-Data for Software Evolution*, pages 41–49, 2005.

[12] R. S. Fabry. How to design a system in which modules can be changed on the fly. In *Proceedings of the International Conference on Software Engineering*, pages 470–476. IEEE Computer Society, 1976.

[13] O. Frieder and M. E. Segal. On dynamically updating a computer program: From concept to prototype. *Journal of Systems and Software*, 14(2):111–128, 1991.

[14] R. P. Gabriel, J. L. White, and D. G. Bobrow. CLOS: integrating object-oriented and functional programming. *Communications of the ACM*, 34(9):29–38, 1991.

[15] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java^{TM} Language Specification*. Addison-Wesley, 3rd edition, 2005.

[16] A. R. Gregersen, D. Simon, and B. N. Jørgensen. Towards a dynamic-update-enabled JVM. In *Proceedings of the Workshop on AOP and Meta-Data for Software Evolution*, pages 1–7. ACM Press, 2009.

[17] D. Gupta and P. Jalote. Increasing system availability through on-line software version change. In *Proceedings of the International Conference on Fault-Tolerant Computing*, pages 30–35. IEEE Computer Society, 1993.

[18] D. Gupta and P. Jalote. On-line software version change using state transfer between processes. *Software - Practice and Experience*, 23(9):949–964, 1993.

[19] D. Gupta, P. Jalote, and G. Barua. A formal framework for on-line software version change. *IEEE Transactions on Software Engineering*, 22(2):120–131, 1996.

[20] J. Gustavsson. A classification of unanticipated runtime software changes in Java. In *Proceedings of the International Conference on Software Maintenance*. IEEE Computer Society, 2003.

[21] J. Gustavsson, T. Staijen, and U. Assmann. Runtime evolution as an aspect. In *Proceedings of International Workshop on Foundations of Unanticipated Software Evolution*, 2004.

[22] M. Hicks and S. Nettles. Dynamic software updating. *ACM Transactions on Programming Languages and Systems*, 27(6):1049–1096, 2005.

[23] G. Hjlmtysson and R. Gray. Dynamic C++ classes – a lightweight mechanism to update code in a running program. In *Proceedings of the USENIX Technical Conference*, pages 65–76, 1998.

[24] U. Hölzle, C. Chambers, and D. Ungar. Debugging optimized code with dynamic deoptimization. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 32–43. ACM Press, 1992.

[25] A. Ketfi, N. Belkhatir, and P.-Y. Cunin. Adapting applications on the fly. In *Proceedings of the IEEE International Conference on Automated Software Engineering*, page 313. IEEE Computer Society, 2002.

[26] T. Kotzmann, C. Wimmer, H. Mössenböck, T. Rodriguez, K. Russell, and D. Cox. Design of the Java HotSpot^{TM} client compiler for Java 6. *ACM Transactions on Architecture and Code Optimization*, 5(1), 2008.

[27] J. Kramer and J. Magee. The evolving philosophers problem: Dynamic change management. *IEEE Transactions on Software Engineering*, 16(11):1293–1306, 1990.

[28] J. Kramer and J. Magee. Analysing dynamic change in software architectures: a case study. In *Proceedings of the International Conference on Configurable Distributed Systems*, pages 91–100. IEEE Computer Society, 1998.

[29] T. Lindholm and F. Yellin. *The Java^{TM} Virtual Machine Specification*. Addison-Wesley, 2nd edition, 1999.

[30] S. Malabarba, R. Pandey, J. Gragg, E. Barr, and J. F. Barnes. Runtime support for type-safe dynamic Java classes. In *Proceedings of the European Conference on Object-Oriented Programming*, pages 337–361. Springer-Verlag, 2000.

[31] I. Neamtiu, M. Hicks, G. Stoyle, and M. Oriol. Practical dynamic software updating for C. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM Press, 2006.

[32] Oracle Corporation. *Java Debug Wire Protocol (JDWP)*, 2009. `http://java.sun.com/javase/6/docs/technotes/guides/jpda/jdwp-spec.html`.

[33] Oracle Corporation. *Da Vinci Machine Project*, 2010. `http://openjdk.java.net/projects/mlvm/`.

[34] Oracle Corporation. *OpenJDK*, 2010. `http://openjdk.java.net/`.

[35] A. Orso, A. Rao, and M. J. Harrold. A technique for dynamic updating of Java software. In *Proceedings of the IEEE International Conference on Software Maintenance*, pages 649–658, 2002.

[36] M. Paleczny, C. Vick, and C. Click. The Java HotSpot^{TM} server compiler. In *Proceedings of the Java Virtual Machine Research and Technology Symposium*, pages 1–12. USENIX, 2001.

[37] M. Pukall, C. Kästner, and G. Saake. Towards unanticipated runtime adaptation of Java applications. In *Proceedings of the Asia-Pacific Software Engineering Conference*, pages 85–92. IEEE Computer Society, 2008.

[38] M. Pukall and M. Kuhlemann. Characteristics of runtime program evolution. In *Proceedings of the Workshop on Reflection, AOP and Meta-Data for Software Evolution*, 2007.

[39] B. Redmond and V. Cahill. Supporting unanticipated dynamic adaptation of application behaviour. In *Proceedings of the European Conference on Object-Oriented Programming*, pages 205–230. Springer-Verlag, 2002.

[40] M. E. Segal and O. Frieder. On-the-fly program modification: Systems for dynamic updating. *IEEE Software*, 10(2):53–65, 1993.

[41] G. L. Steele, Jr. *Common LISP: the Language*. Digital Press, second edition, 1990.

[42] G. Stoyle, M. Hicks, G. Bierman, P. Sewell, and I. Neamtiu. Mutatis mutandis: Safe and predictable dynamic software updating. *ACM Transactions on Programming Languages and Systems*, 29(4), 2007.

[43] S. Subramanian, M. Hicks, and K. S. McKinley. Dynamic software updates: a VM-centric approach. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 1–12. ACM Press, 2009.

[44] Y. Sugiyama. A mechanism for runtime evolution of objects. In *Proceedings of the International Workshop on Principles of Software Evolution*, 1999.

[45] Y. Vandewoude, P. Ebraert, Y. Berbers, and T. D'Hondt. An alternative to quiescence: Tranquility. In *Proceedings of the IEEE International Conference on Software Maintenance*, pages 73–82. IEEE Computer Society, 2006.

[46] A. Villazón, W. Binder, D. Ansaloni, and P. Moret. HotWave: creating adaptive tools with dynamic aspect-oriented programming in java. In *Proceedings of the International Conference on Generative Programming and Component Engineering*, pages 95–98. ACM Press, 2009.