JOHANNES KEPLER UNIVERSITÄT LINZ | JKU

TNF

Technisch-Naturwissenschaftliche Fakultät

Dynamic Code Evolution for Java

DISSERTATION

zur Erlangung des akademischen Grades

Doktor

im Doktoratsstudium der

Technischen Wissenschaften

Eingereicht von:
Dipl.-Ing. Thomas Würthinger

Angefertigt am:
Institut für Systemsoftware

Beurteilung:
Univ.-Prof. Dipl.-Ing. Dr. Dr.h.c. Hanspeter Mössenböck (Betreuung)
Univ.-Prof. Dipl.-Ing. Dr. Michael Franz

Linz, März 2011

Oracle, Java, HotSpot, and all Java-based trademarks are trademarks or registered trademarks of Oracle in the United States and other countries. All other product names mentioned herein are trademarks or registered trademarks of their respective owners.

Abstract

Dynamic code evolution allows changes to the behavior of a running program. The program is temporarily suspended, the programmer changes its code, and then the execution continues with the new version of the program. The code of a Java program consists of the definitions of Java classes.

This thesis describes a novel algorithm for performing unlimited dynamic class redefinitions in a Java virtual machine. The supported changes include adding and removing fields and methods as well as changes to the class hierarchy. Updates can be performed at any point in time and old versions of currently active methods will continue running.

Type safety violations of a change are detected and cause the redefinition to fail gracefully. Additionally, an algorithm for calling deleted methods and accessing deleted static fields improves the behavior in case of inconsistencies between running code and new class definitions. The thesis also presents a programming model for safe dynamic updates and discusses useful update limitations that allow a programmer to reason about the semantic correctness of an update. Specifications of matching code regions between two versions of a method reduce the time old code is running.

All algorithms are implemented in Oracle's Java HotSpot VM. The evaluation shows that the new features do not have a negative impact on the peak performance before or after a dynamic change.

Kurzfassung

Dynamische Evolution ermöglicht den Eingriff in das Verhalten eines laufenden Programms. Das Programm wird temporär angehalten, der Programmierer verändert den Quelltext und dann wird die Ausführung mit der neuen Programm-Version fortgesetzt. Der Quelltext von Java-Programmen besteht aus den Definitionen von Java-Klassen.

Diese Arbeit beschreibt einen neuartigen Algorithmus für die unlimitierte dynamische Neudefinition von Java-Klassen in einer virtuellen Maschine. Die unterstützten Änderungen beinhalten das Hinzufügen und Entfernen von Feldern und Methoden sowie Veränderungen der Klassenhierarchie. Der Zeitpunkt der Veränderung ist nicht beschränkt und die aktuell laufenden Ausführungen von alten Versionen einer Methode werden fortgesetzt.

Mögliche Verletzungen der Typsicherheit werden erkannt und führen zu einem Abbruch der Neudefinition. Ein Algorithmus für das Aufrufen von gelöschten Methoden und für den Zugriff auf gelöschte statische Felder verbessert das Verhalten im Fall von Inkonsistenzen zwischen den gerade laufenden Methoden und den neuen Klassendefinitionen. Die Arbeit präsentiert auch ein Programmiermodell für sichere dynamische Aktualisierungen und diskutiert nützliche Limitierungen, die es dem Programmierer ermöglichen, über die semantische Korrektheit einer Aktualisierung Schlussfolgerungen zu ziehen. Die Angabe von gleichartigen Quelltextbereichen zwischen zwei Versionen einer Methode reduzieren die Zeit, in der noch alte Methoden ausgeführt werden.

Alle Algorithmen sind in der Java HotSpot VM von Oracle implementiert. Die Evaluierung zeigt, dass die neuen Fähigkeiten weder vor noch nach einer dynamischen Veränderung einen negativen Einfluss auf die Spitzenleistung der virtuellen Maschine haben.

Contents

Acknowledgements

I want to thank all the people who greatly supported the work presented in this thesis. First and foremost, I thank my PhD advisor Hanspeter Mössenböck for his many helpful comments and directions as well as for encouraging me to write the thesis. I thank the second thesis advisor and dissertation committee member Michael Franz from the University of California in Irvine for taking the responsibility of reviewing this thesis.

The work was funded by the HotSpot compiler team at Oracle. I thank current and former Oracle employees for their persistent support by providing feedback to my work and answering all questions related to internals of the virtual machine. In particular, I want to acknowledge John Rose, David Cox, Thomas Rodriguez, Kenneth Russell, and Doug Simon.

I thank my master's thesis advisor Christian Wimmer for providing important comments on my PhD work although he moved to a different university. The insurance software company Guidewire supported the project with additional funding. Also, their feedback from using prototypes of this work and the support from Alan Keefer were especially helpful for increasing my motivation.

The work highly benefitted from a collaboration with the University of Lugano in the area of dynamic aspect-oriented programming. I thank Walter Binder, professor at the University of Lugano, as well as his PhD student Danilo Ansaloni for working together with me on this topic and for contributing feedback and ideas. The case study presented in Section 6.3 is a direct result of this collaboration.

The funding from Guidewire was used to employ two undergraduate students to help with the implementation and testing efforts. I thank Kerstin Breiteneder and Christoph Wimberger for supporting my efforts in creating a stable implementation. Additionally, their work on dynamic mixins provides an interesting use case for the algorithms presented in the thesis (see Section 6.2).

I thank my colleagues at the Institute for System Software for numerous discussions and for providing valuable feedback. In particular, I thank Lukas Stadler and Thomas Schatzl.

Last but not least, I thank all the people whose care and mental support made this thesis possible. I thank my parents, my two sisters Elisabeth and Daniela, and also my friends from the KHG Linz.

Chapter 1

Introduction

1.1 Dynamic Changes to Programs

Researchers investigated the problem of updating the code of a running program early in programming history [1]. The research focused on procedural programming languages, where a dynamic update replaces the implementation of functions and conversion routines update the data sections. With the advent of object-oriented languages, class member declarations and subtype relationships became important parts of a program's behavior. A true dynamic code update now also has to include object layout changes and changes to the semantics of method calls due to a new class hierarchy.

Using a virtual machine (VM) to execute programs helps solving these new challenges. The VM increases the possibilities for dynamic code evolution because of the additional abstraction layer between the executing program and the hardware. The main tasks of this intermediate layer are automatic memory management, dynamic class loading, and program verification. The algorithms for dynamic changes to class definitions presented in this thesis make heavy use of the existing infrastructure of the VM.

In Java, dynamic changes are currently known as *hotswapping*, because they are restricted to swapping method bodies only. The enhancement of hotswapping to allow additional kinds of dynamic changes is however a high priority for many Java developers. This is shown by the votes for requests for enhancements on the official Oracle website (Bug ID: 4910812) [2], where the request for enhancing hotswapping is among the requests with the most votes.

Old Program

```
class A {
  protected int x;
  public final int foo() {
    return x;
  }
}

class B {
}
```

New Program

```
class A' {
  protected int x;
  public int foo() {
    return x;
  }
}

class B' extends A' {
  private int y;
  public int foo() {
    return x + y;
  }
}
```

Figure 1.1: An example for a change to an object-oriented program.

Figure 1.1 shows a motivational example of redefining two Java classes with new versions. The new version defines a new supertype for class `B`. Instead of inheriting from the `Object` class, `B` now inherits from `A`. Additionally, `B` has a new method `foo` that overwrites the `foo` method of `A`. At the same time, the `foo` method of `A` is no longer declared `final`.

Changes to classes may depend on each other such that they have to be carried out together. In the example, the change of `B` cannot be performed without the change of `A`, because the new version of `B` overrides a method that was declared `final` in the old version of `A`. Changing first `A` and then `B` would be a valid order. However, such an order cannot be found in the case of cyclic dependencies where an additional change in `A` depends on a change in `B`. Therefore, the class redefinition in Java must be performed as an atomic replacement of a set of classes.

Changes to class definitions may have implications on already existing object instances of those classes. In the example, the instances of `B` get two new fields: One field `x` inherited from `A` and another field `y` that was added to class `B`. Therefore, the size and the object layout of existing instances of `B` change.

The goal of this thesis is to explore the problem of dynamically changing definitions of statically typed, object-oriented programs. The thesis presents an algorithm for efficient implementation of arbitrary changes to loaded classes in a VM. Additionally, it discusses semantic correctness of changes and presents possible applications of the new dynamic update features. The VM for unlimited class redefinition that was developed as part of this thesis is called Dynamic Code Evolution VM (DCE VM).

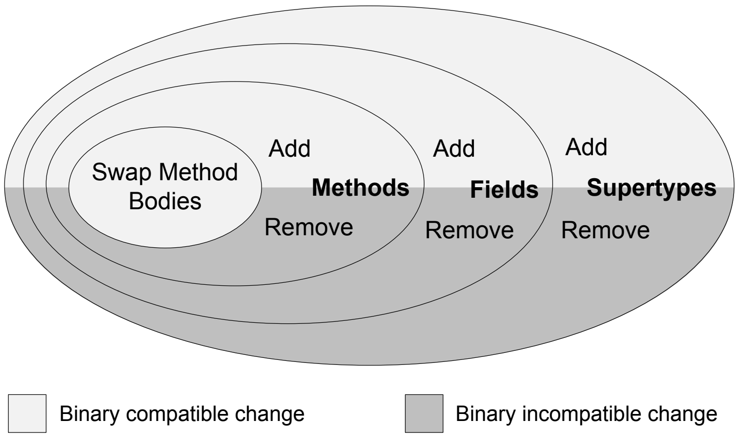


Figure 1.2: Different levels of code evolution.

1.2 Levels of Evolution

Several classifications of dynamic changes have been published [3, 4]. Based on our experiences of impact on the complexity of the implementation within the VM, we propose the distinction of four levels of code evolution as shown in Figure 1.2:

Swapping Method Bodies: Modifying only the bytecodes of a Java method without changing other parts of the class definition is the simplest possible change. There are no references from other classes to the actual bytecodes of a method, so the change can be done in isolation from the rest of the system. This is the only kind of change that is implemented in the current product version of the Java HotSpot VM. It is known as *hotswapping*.

Adding or Removing Methods: The VM maintains a data structure for every class that contains a virtual method table and an interface method table (see Section 2.3). Changing the set of methods of a class can imply changes to the entries and the size of those tables. Additionally, a change in a class can have an impact on the method tables of a subclass (see Section 3.2). The 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 direct references 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 may need to convert the old version of an object to a new version that has different fields and a different size (see Section 3.4). If object sizes increase, we use a modified version of the mark-and-compact garbage collector (see Section 3.5). If they decrease or stay the same, no garbage collection is necessary. Added fields need to be either initialized with default values, or by running custom code for every existing instance of a redefined class. 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 object-oriented languages. For a class, this can mean changes to its methods as well as its fields. Additionally, the VM class objects need to be modified in order to reflect the new supertype relationships. Removing a supertype requires additional type safety checks and is not possible in all cases (see Section 4.2).

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 of interfaces can be treated similarly to modifications of 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.

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 can invalidate existing machine code (e.g., when this code contains static field offsets). Additionally, a class redefinition 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 [5]. The light grey areas of Figure 1.2 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 different methods 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. Chapter 4 discusses possible problems and the solution chosen for the DCE VM.

1.3 Applications and their Requirements

Dynamic code evolution can be used in different domains with their own sets of requirements. We distinguish four main applications of dynamic code evolution:

Accelerated Development. When a developer frequently makes small changes to an application with a long startup time, dynamic code evolution significantly increases productivity. The developer can modify and recompile the program after suspending it at a breakpoint. Then, he can resume the program including the modifications 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 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 the 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.

Long-Running Server Applications. Critical server applications that must not be shut down can only be updated using dynamic code evolution. The server applications must not be slowed down before or after performing the code evolution. The main 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 by performing the update only at certain points in time. Also, not using all possible kinds of updates helps to obtain safety.

Dynamic Languages. There are various efforts to run dynamic languages on statically typed VMs (see for example [6]). Dynamic code evolution is a common feature of dynamic languages. VM support for this feature therefore simplifies the implementation of dynamic languages on statically typed VMs. The requirement here is that small incremental changes (e.g., adding a field or method) can be carried out fast.

Dynamic Aspect-Oriented Programming. 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 [7, 8, 9]. Those tools can immediately benefit from enhanced code evolution facilities.

The focus of our implementation is on supporting accelerated development. In this thesis, we broaden the view by also approaching the safety challenges when updating long-running server applications and discuss a programming model for safe updates (see Chapter 5). We present a case study that combines the DCE VM with a dynamic AOP tool (see Section 6.3). Support for small incremental changes is discussed as possible future work (see Section 9.2).

1.4 State of the Art

This section briefly describes two commonly used approaches to dynamic class redefinition for Java. For a detailed comparison of our work with other systems, see Chapter 8. The two main solutions for Java available at the time of writing this thesis are:

Hotswapping. The current version of the Java HotSpot VM has the ability to swap the definitions of method bodies at run time. This is based on the work done by Dimitriev [5, 10]. Many Java debuggers have a command for applying code changes made during a debugging session that triggers hotswapping. The program will continue running with the new method definitions. Old methods that are currently active will continue to run the old version. The main limitation is that it is only allowed to swap the definition of method bodies. Changing any other part of the class definition requires a restart. In particular, hotswapping does neither support adding or removing of methods or fields nor changing the supertypes of a class.

Bytecode Rewriting. Based on the ability to swap method bodies, bytecode rewriting techniques try to simulate more advanced class redefinition. Additional indirections are inserted in order to intercept the normal semantics of Java bytecodes. For example, an instance field access is replaced by a method call that looks up the field value from a hash table. While this relaxes some of the restrictions of hotswapping, bytecode rewriting techniques suffer from bad performance due to the additional indirections. Also, Java stack traces and debugger information do not reflect the original program but contain additional entries due to introduced artificial methods.

This thesis advances the state of the art for dynamic code evolution for Java. To the best of our knowledge, the DCE VM is the first VM for a statically typed, object-oriented language that supports unlimited class redefinition capabilities without compromising execution performance. For a detailed list of contributions, see Section 9.1.

1.5 Project Context and Activities

The thesis was created as part of the long-running and successful research collaboration between the Institut for System Software (formerly Institute for Practical Computer Science) at the Johannes Kepler University Linz and Oracle (formerly Sun Microsystems). The collaboration started in 2000 when Hanspeter Mössenböck enhanced the HotSpot client compiler with an intermediate representation in static single assignment (SSA) form [11]. Since then, various research papers and theses have been published in the area of compiler construction and virtual machines. Here is a list of selected publications:

- Michael Pfeiffer and Christian Wimmer implemented a linear scan register allocator [12, 13].
- Thomas Kotzmann explored an algorithm for escape analysis [14, 15].
- Christian Wimmer created a version of the VM that supports object co-location [16], object inlining [17], and array inlining [18]. His work is summarized in a journal publication [19].
- The author of this thesis developed an array bounds check elimination algorithm that optimistically moves bounds checks out of loops and groups checks [20, 21]. Additionally, he created a visualization tool for the program dependence graph of the Java HotSpot server compiler [22, 23].
- Christian Häubl published optimizations for the representation of `String` objects [24].
- Lukas Stadler created an experimental VM version that supports continuations and coroutines [25, 26].

- A visualization tool for the client compiler was developed as a combined effort of several student projects [27].

The author's work on dynamic code evolution started in 2008 as part of the collaboration with Sun Microsystems. In summer 2008, the author worked as an intern at Sun Labs as a member of the Maxine VM [28] team. He implemented a debugging client for the Maxine VM [29]. In February 2009, the first open source version of the DCE VM was published. In summer 2009, the author interned again at Sun Labs working on a Java port of the HotSpot client compiler and compiler-runtime separation [30]. In September 2009, the current state of the dynamic code evolution research was presented at the JVM Language Summit in Santa Clara, CA.

In March 2010, Guidewire (an insurance software company heavily using Java) joined the project by funding two research assistants (Kerstin Breiteneder and Christoph Wimberger). They helped stabilizing the DCE VM by writing additional test scenarios and implemented dynamic mixins for Java (see Section 6.2). In April 2010, the project was enriched by starting a collaboration with Walter Binder and his Dynamic Analysis Group from the University of Lugano. Their dynamic aspect-oriented programming tool HotWave benefits from the dynamic evolution capabilities of the DCE VM and the combination of the two tools opens up new possibilities (see Section 6.3). In June 2010, the implementation reached an important milestone by passing all of Oracle's internal class redefinition tests, in addition to an increasing set of self-written unit tests.

In September 2010, the dynamic code evolution project was presented for the first time to a broader audience at the JavaOne conference in San Francisco, CA. At the same time, the author launched a website for the DCE VM with links to the source code and binaries for easy installation. The website immediately received significant attention in terms of page views and binary downloads.

Several papers have been published covering different aspects of the author's research on dynamic code evolution. This thesis summarizes and extends the algorithms and results presented in these publications:

- Conference paper about the class redefinition algorithm [31] (see Chapter 3), published in the Proceedings of the 8th International Conference on the Principles and Practice of Programming in Java (PPPJ'10).

- Workshop paper on the design of the dynamic aspect-oriented programming tool built on top of the DCE VM [32], published in the Proceedings of the 7th ECOOP'10 Workshop on Reflection, AOP and Meta-Data for Software Evolution.
- Tool demonstration paper outlining two use cases of the DCE VM [33], published in the Proceedings of the 9th International Conference on Generative Programming and Component Engineering (GPCE'10).
- Journal paper extending the base algorithm with performance improvements and safe binary incompatible changes [34] (see Chapter 4), currently under review for publication in the journal Science of Computer Programming.

1.6 Structure of the Thesis

Chapter 2 gives an introduction to the Java HotSpot VM with a focus on the parts modified for the DCE VM. Chapters 3 to 5 comprise the main body of this thesis, which is divided into three parts: Chapter 3 describes the algorithm developed for the core functionality. Chapter 4 discusses problems due to changes that remove class members or supertypes and presents the solutions chosen for the DCE VM. Safe update regions are introduced in Chapter 5. This section also presents restrictions necessary to guarantee safety properties of a class redefinition.

Chapter 6 describes three case studies of applications of the DCE VM: A modified NetBeans version to support on-the-fly GUI development, a prototype implementation of dynamic mixins for Java, and a dynamic aspect-oriented programming tool. Chapter 7 gives a performance and a functional evaluation of the DCE VM. Chapter 8 discusses related work and Chapter 9 lists the contributions, discusses future work, and concludes the thesis.

Chapter 2

The Java HotSpot VM

The algorithms described in this thesis were being developed on top of the current production version of Oracle's Java HotSpot VM. This section presents the main components of the HotSpot VM with a focus on the parts that are most relevant for class redefinition.

The HotSpot VM is an open source production-quality VM that executes Java bytecodes. It forms part of the OpenJDK project and is actively developed by Oracle. The name derives from its main characteristic: Instead of optimizing the whole program, it focuses on frequently executed "hot spots" in the code. It is available for a wide variety of operating systems (Solaris, Windows, Linux, . . .) and for different processor architectures (IA32, AMD64, SPARC, . . .). The machine code examples of this section focus on the AMD64 version of the VM.

Figure 2.1 shows the components of the VM that were adapted for dynamic code evolution. The Java bytecodes (see Section 2.1) represent the executed Java program. At first, all bytecodes are interpreted (see Section 2.5). When the invocation frequency of a method exceeds a predefined threshold, the bytecodes of the method are compiled to machine code. Further calls to the method then target the machine code and execute faster. At the start of the VM, the user can choose between two compiler configurations: The client compiler performs light-weight optimizations and is therefore suitable for short-running client applications whose startup time matters. The server compiler performs heavy-weight optimizations and is suitable for long-running server applications.

Both compilers use optimistic assumptions that may later be invalidated (e.g., by a newly loaded class). Therefore, it is necessary that compiled code can be *deoptimized* [35]: The execution of this code is resumed in the interpreter. In order to be able to opti-

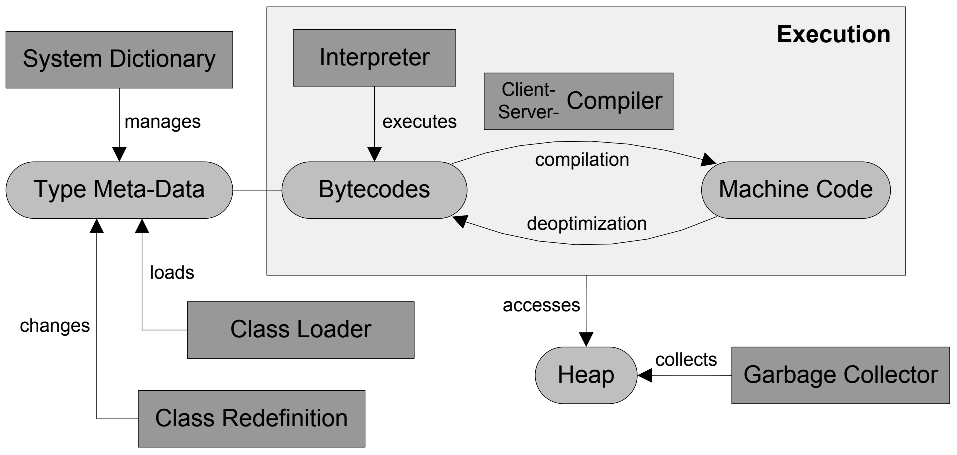


Figure 2.1: Main components of the Java HotSpot VM.

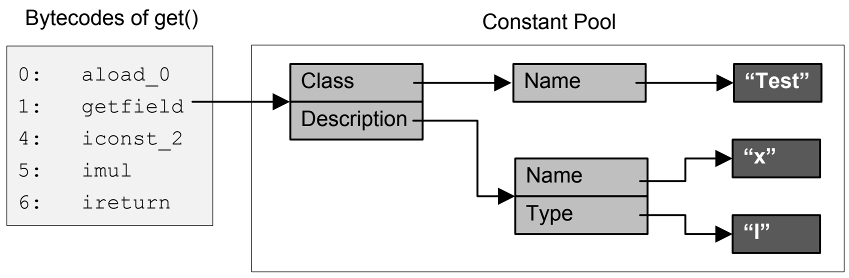
mize methods with long-running loops, the VM also supports the opposite mechanism: Transferring execution of a method from the interpreter to the compiler, also known as *on-stack-replacement* [36]. The garbage collector enables automatic memory management (see Section 2.7). Unreachable Java objects are collected and removed from the heap.

Java classes are dynamically loaded at run time (see Section 2.2). Class loaders are responsible for providing the bytecodes of a class, whenever it is necessary to use a class while executing a Java program. The VM class objects are stored in the system dictionary (see Section 2.3 and Section 2.4). The class redefinition mechanism is responsible for replacing the definition of loaded classes (see Section 2.8). It is part of the HotSpot VM since version 1.4, but it is limited to only change the bodies of methods. This thesis extends the mechanism to allow arbitrary changes.

2.1 Java Bytecodes

Java source files are compiled to Java class files (e.g., with the `javac` command line tool). The class files contain a compressed and platform-independent description of the Java class. When compiling method bodies, the structured Java control flow (e.g., `for`, `while`, or `if`) is converted to conditional jump bytecodes. Also, the `javac` compiler transforms several Java language constructs into more low-level constructs to simplify their implementation in the VM. Such Java language features include generics, enumerations, the `finally` block, and variable length parameter lists.

```
class Test {
  int x;
  int get() { return x * 2; }
}
```



```
0:   aload_0
1:   getfield
4:   iconst_2
5:   imul
6:   ireturn
```

Figure 2.2: A Java method and its corresponding bytecodes and referenced constant pool entries.

A class file contains a *constant pool* that is organized as a symbol table of the class. Java bytecodes reference constant pool entries when their specification requires additional parameters (e.g., a field or a method description). A constant pool entry can be referenced by multiple bytecodes. Also, it can reference other constant pool entries. This way the size of the class file remains small.

Figure 2.2 shows a sample Java class with a field `x` and a method `get`. The method has one local variable: The receiver (i.e., the `this` pointer) is stored at the local variable with index 0. The bytecodes of the method operate on an execution stack where the inputs to a bytecode are taken from the stack and the result is pushed onto the stack.

The first bytecode is `aload_0` and pushes the value of the local variable with index 0 (i.e., in this case the receiver) onto the top of the stack. The `getfield` instruction pops the object instance from the stack and pushes the value of the accessed field. The specification of the field is provided as a reference to a constant pool entry. This entry points to two other entries: One for the field holder and one for more information about the field. The entry for the field holder finally points to a text entry containing the name of the class. The other entry is divided into one pointer to the field name "x" and one pointer to the field type, specified as the shortcut "I" for `int`.

After the execution of the `iconst_2` bytecode, two values are on the stack: The value of the field and the constant `2`. Those two values form the input for the `imul` instruction that pops two values from the stack and pushes the result. The `ireturn` instruction finally returns the value that is on top of the stack.

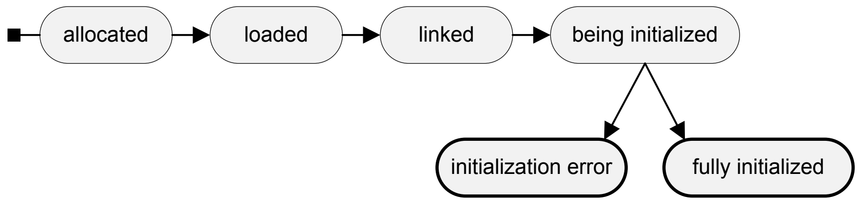


Figure 2.3: Initialization states when loading a Java class in the HotSpot VM.

2.2 Class Loading

Java classes are dynamically loaded while the program is running. When the interpreter finds a constant pool entry referring to a class name, it asks the runtime to resolve the name to a VM class object. The runtime delegates the task of finding the binary data of a class to a *class loader*. The class loader can then either try to find a Java class file on the local hard drive or retrieve the data from a different source (e.g., a network stream). Class loaders form a tree hierarchy with the initial (bootstrap) class loader at the root. A class loader first delegates the search for the class to its parent class loader. Only if the parent class loader cannot find the class, it tries to load the class itself.

Before the class bytes are deserialized, the VM notifies any registered *class file transformers* of the `java.lang.instrument` API (see Section 2.8.3). They may modify the class bytes before the new class is loaded. Then, the VM class objects holding VM specific data about the Java class are *allocated*. Figure 2.3 shows that this is the first initialization state of a class. The VM parses the class bytes and adds the VM class object to the type hierarchy (see Section 2.3). Now the state of the class is advanced to *loaded*.

In the linking phase, the methods of the class are verified. The VM must make sure that malicious bytecodes cannot cause security problems. The Java program could be loaded from an untrusted network source and it is therefore necessary to guarantee that the program cannot escape the Java security model. The verifier checks that the operands of every Java bytecode have the correct type (e.g., that a number is not misused as an object pointer). The verifier also checks subtype relationships (e.g., in case of assigning a value to a reference field). This subtype check can trigger loading of other classes. After successful verification, the class is marked as *linked*.

The next step is that the newly loaded class is *being initialized*. The VM makes sure that the superclass is initialized and then calls the static initializer. If the static initializer

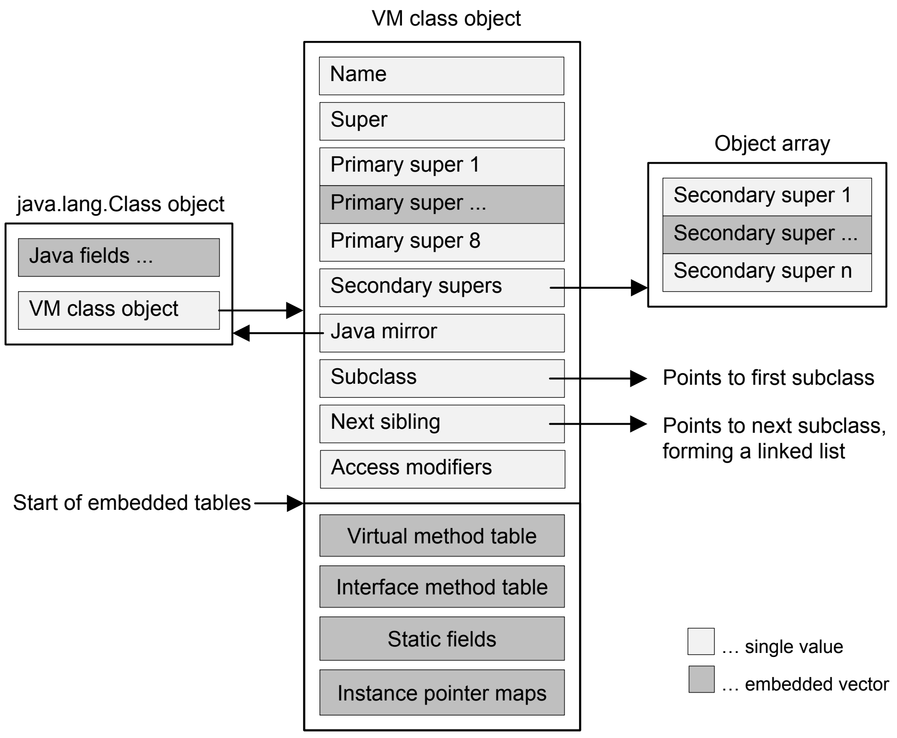


Figure 2.4: VM class object with embedded tables.

runs without exception the class is *fully initialized*. Otherwise an *initialization error* is reported.

2.3 Type Hierarchy

For every loaded class the VM maintains a *class object* that is linked with other class objects forming a *type hierarchy*. Figure 2.4 shows the structure of a VM class object. Most Java classes have only few supertypes, but the number of supertypes can be up to 2^{16} implemented interfaces plus one superclass. The first supertype is stored as a direct field, the next 8 supertypes as an embedded vector, and for classes exceeding 8 supertypes an extra object array is allocated. The first loaded subtype of a class is referenced by a direct pointer, additional subtypes form a linked list of siblings. The Java mirror is the `java.lang.Class` instance for reflective access to the class. It contains a reference back to the VM class object for fast access.

The class object contains four additional arrays that are embedded so that they can be accessed without additional memory indirection:

Virtual method table: A table with one entry for every method that can be overwritten in subclasses (i.e., for every method that is not declared `private` or `final`). Additionally, the table starts with one entry for every virtual method table entry of the superclass. If a class `B` overrides a method of a superclass `A`, the corresponding method table entry in `B` differs from that in `A`, otherwise they are the same.

Interface method table: This table contains one subtable per implemented interface. Each subtable contains one entry per interface method, which references the method of the class that implements the interface method.

Static fields: The static fields are divided into two parts: Static pointer fields that need to be processed for garbage collection and static primitive fields. The values of the fields are stored directly in the VM class object thus speeding up static field accesses.

Instance pointer map: A bitmap tells the garbage collector which heap words in instances of this class are pointers. Knowledge of the exact location of pointers is a prerequisite for precise garbage collection.

2.4 System Dictionary

The system dictionary is used to store the VM class objects for all currently loaded classes. When the compiler, the interpreter, or the verifier finds a constant pool entry with a class name, it tries to look up the class in the system dictionary. If the class is not found, the VM triggers class loading.

The dictionary is organized as a hash table with entries as shown in Figure 2.5. An entry is identified by the name of the class and the class loader. The hash index is computed using the identity hash codes of those two objects. The identity hash code of the name can be used, because the VM always uses the same objects for two identical class names. The entry itself contains a reference to the VM class object, a reference to the class loader, and optionally a list of *protection domains*. A protection domain can be used to restrict access to a Java class.

Java classes are loaded in parallel to the running application. Therefore, the VM needs to prevent possible race conditions when multiple threads try to define the same class. This is achieved by using the placeholder hash table: Before loading a new class, a thread checks whether a placeholder entry for the given class exists. If there is no such entry,

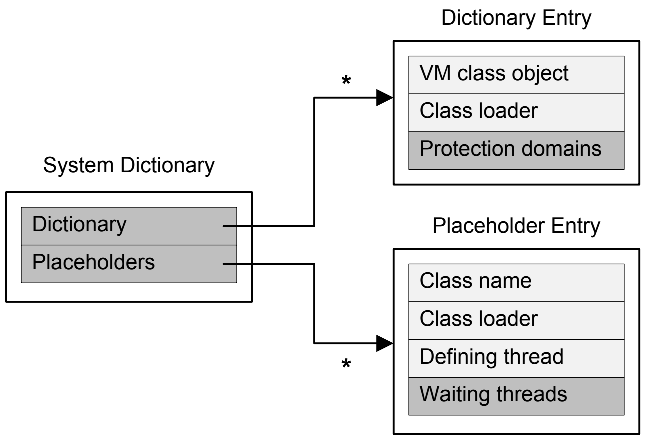


Figure 2.5: The system dictionary that manages all loaded Java classes.

the thread places a new entry into the placeholder hash table. If the thread finds such an entry, it knows that another thread is currently loading the class and waits for the other thread to finish the operation. After loading a class, the placeholder entry is removed and a normal entry is added to the dictionary. Possibly waiting threads are notified when the class becomes available. Before checking or modifying the placeholder table, a global system dictionary lock is acquired.

2.5 Template Interpreter

The HotSpot VM contains two interpreter implementations: One interpreter written in C++ and one that generates machine code templates for all Java bytecodes that can occur in Java class files. The advantage of the C++ interpreter is the better portability, because its core part can be compiled for any platform supported by a C++ compiler. The template interpreter however is faster, because the Java bytecodes are executed using handcrafted machine code. It is the default configuration and works on the main architectures supported by the VM (IA32, AMD64, and SPARC).

Figure 2.6 shows the calling conventions and the stack frame layout of the interpreter. Before invoking an interpreted method, the arguments are pushed onto the machine stack. Executing the AMD64 `call` instruction automatically pushes the return address. On method entry, the interpreter moves the return address above the area of the local variables. This is necessary, because the parameters are accessible as locals and should there-

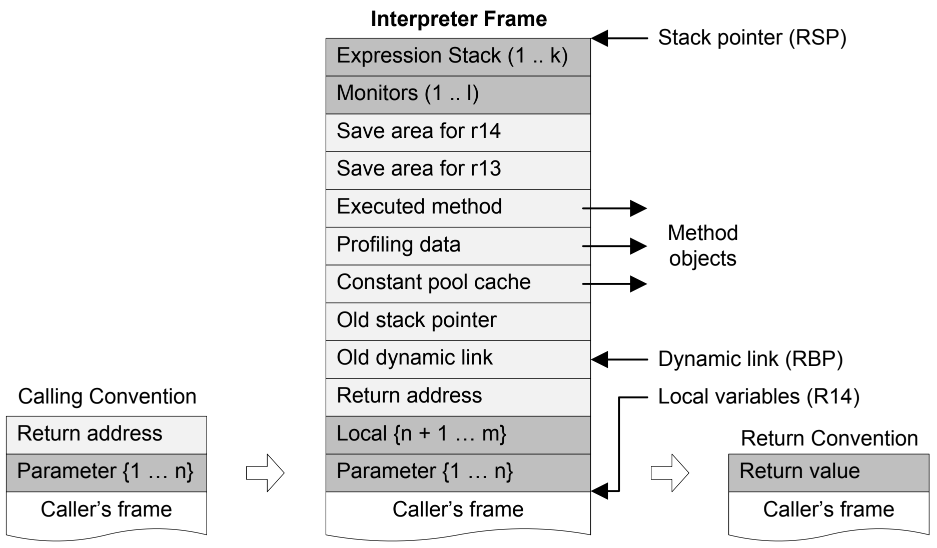


Figure 2.6: Interpreter calling convention, frame layout, and return convention.

fore be adjacent to any additionally reserved locals. Before returning from a method, the return value replaces the parameters as shown on the right side of the figure.

There are two frequently accessed values cached in registers: `r13` is a pointer to the memory location of the current bytecode that is incremented as the interpreter executes the method. `r14` is a pointer to the first local variable and is used by the bytecodes that load and store local variables. The interpreter has utility methods for storing and restoring the values of those registers. This is necessary, when the interpreter calls runtime methods that may destroy register values.

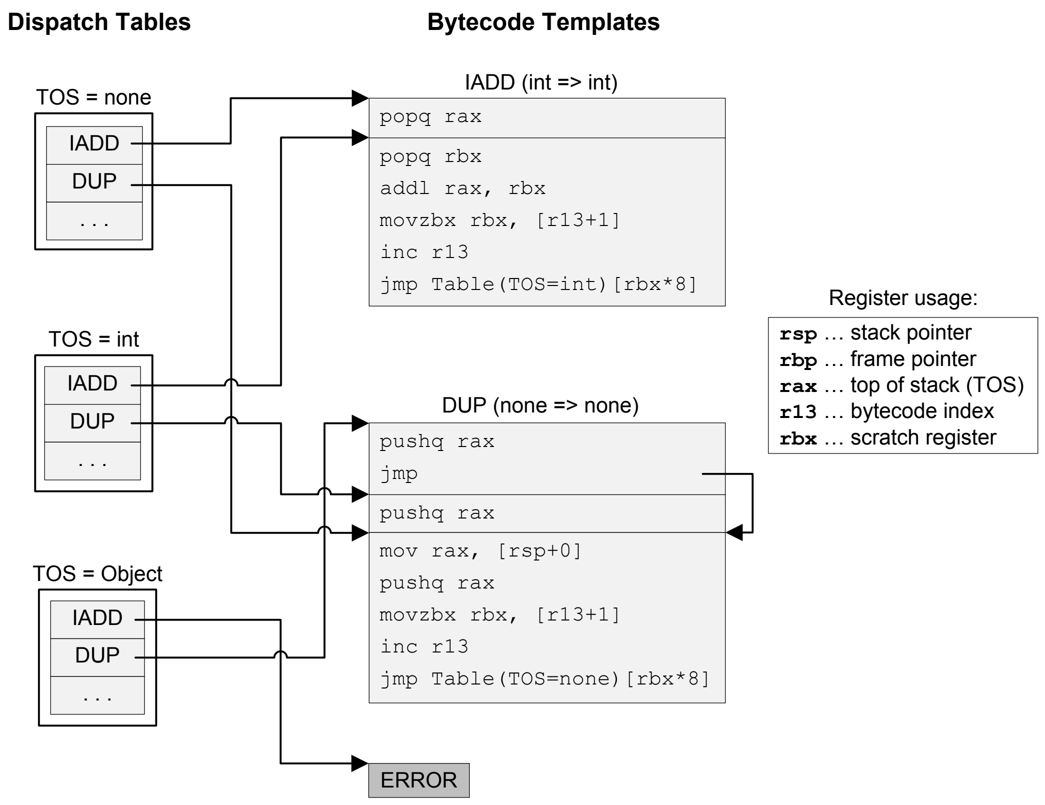
The interpreter frame has three pointers to method-specific objects for fast access: A pointer to metadata about the executed method, a pointer to an object where the interpreter should store profile information, and a pointer to the constant pool cache. The profile information is only used in the server compiler configuration. In this configuration, the interpreter saves branch target probabilities and dynamic types of values (e.g., for the receiver at invocation bytecodes). The compiler can then use this information for optimistic assumptions (e.g., that a certain code path is never executed). The constant pool cache saves information about resolved field accesses and method invocations. An uninitialized constant pool cache entry for a field access bytecode points to the constant pool entry that specifies the name of the field. Looking up the field offset based on the name is however a slow operation. Therefore, the interpreter stores the field offset directly in the constant pool cache entry for fast subsequent field access. For method invocations, it caches either a direct pointer to the target method, a virtual method table index, or an interface method table index.

The topmost element of the expression stack is the *top of stack* (TOS) element. The interpreter uses TOS caching, where this element is kept in a register between bytecodes. In case of the AMD64 template interpreter, the TOS register is `xmm0` for floating point values and `rax` for all other values. The *TOS type* specifies the type of the value that is currently stored in the TOS register. This can either be a Java primitive type (byte, boolean, char, short, int, long, float, or double), an object reference, or none (i.e., the TOS register does not contain a value). A bytecode template has a TOS input type (i.e., the type of the value that is expected in the TOS register) and a TOS output type (i.e., the type of the value that is put into the TOS register). For every bytecode template, the VM generates an entry for every TOS type that maps the input value to the expected TOS input type. A separate bytecode dispatch table for every TOS type points to those typed entries. After executing a bytecode, the interpreter creates the dispatch code using the table corresponding to the TOS output type of the bytecode.

Figure 2.7 shows two sample bytecode templates: One for `iadd` and one for `dup`. The `iadd` template has the TOS input type `int` and generates its value again into the TOS register `rax`. The template has two entry points: One where the first input value is in the TOS register and another one where it is on the stack. The input value must be an `int` value, therefore the dispatch table for `TOS=Object` has no pointer to the `iadd` template. The verifier guarantees that the input types for every bytecode are correct. The second parameter is retrieved from the machine stack using a `popq` instruction. The next bytecode is loaded into `rbx` using the bytecode address register `r13`. After incrementing the bytecode address, the `int` dispatch table and the retrieved bytecode is used for looking up the entry of the next template.

The `dup` template does not require a TOS input and also does not produce its value into the TOS register. Its entry for `TOS=none` gets the input value from the stack and pushes it again. There are entries for the different possible TOS values that translate to the general entry by pushing the TOS register onto the stack. The dispatch after the bytecode is then performed using the dispatch table for `TOS=none`.

The interpreter rewrites bytecodes when it has more specific information about them. An example is the `getfield` bytecode. The general version does not know the type of the retrieved value, but after resolving the field, the type is fixed. The bytecode is then



```
popq rax
popq rbx
addl rax, rbx
movzbx rbx, [r13+1]
inc r13
jmp Table(TOS=int)[rbx*8]
```

```
pushq rax
jmp
pushq rax
mov rax, [rsp+0]
pushq rax
movzbx rbx, [r13+1]
inc r13
jmp Table(TOS=none)[rbx*8]
```

Figure 2.7: Interpreter jump tables and machine code generated for the `iadd` and the `dup` bytecode.

rewritten to a new bytecode that produces its value into the TOS cache register. An additional optimization of the interpreter is the special handling of known math functions (e.g., `Math.min`). Instead of calling the method, it calls a machine code stub that performs the operation.

2.6 Deoptimization

Java threads can only be stopped at *safepoints*. Safepoints are, for example, method call instructions as well as backward jumps (in order to avoid long-running loops without safepoints). The number of safepoints should be small to reduce execution overhead, but it should be high enough such that every thread can reach the next safepoint quickly. Beside the use for deoptimization, safepoints are also used for pausing all threads before a garbage collection. At every safepoint, the VM knows which machine code locations contain object pointers, which is a precondition for precise garbage collection.

At a safepoint, the VM can also change the active execution point of a compiled method from the current machine code location to the corresponding bytecode position in the interpreter. This change from compiled code back to interpreted code is called *deoptimization* [35]. The VM constructs a new interpreter frame and sets the values of the local variables and the expression stack. For every safepoint, the compiler provides a mapping from machine locations (i.e., registers and stack slots) to interpreter values. In case of method inlining, more than one interpreter frame is constructed from one compiled frame.

A possible reason for deoptimization is that the compiled code is based on assumptions that become invalid and therefore executing the compiled code is no longer correct. An example is a leaf type assumption (i.e., that a certain type has no subclasses). The compiler can use this assumption, e.g., to convert an `instanceof` check with the leaf type into a simple compare instruction. Dynamic class loading could however later invalidate this assumption by introducing a new subtype. The simple comparison is no longer sufficient and has to be replaced by a full subtype check. The machine code must no longer be used. Anyone currently executing the machine code needs to continue executing the method in the interpreter.

The server compiler uses deoptimization also for placing *uncommon traps* in the compiled code. Instead of compiling a bytecode branch that is rarely or never executed, it places a deoptimize instruction that jumps to the interpreter if this branch is reached. The compiler makes such decisions based on branch probability information gathered by the interpreter. Reducing the number of compiled bytecodes speeds up compilation and in many cases also produces faster machine code, because the register allocator is not influenced by the liveness of values in rarely executed code paths.

The assumptions about the class hierarchy and about never executed code paths are optimistic. Therefore, a mechanism to invalidate the machine code is necessary. If the method is later again frequently executed in the interpreter, the compiler creates new machine code for the method.

Deoptimization is only possible for the topmost stack frame, because the interpreted frame can require a larger stack frame than the compiled frame. Therefore, deoptimization of a frame that calls another frame is delayed. The VM simply overwrites the machine code after the currently executed `call` machine instruction with a jump to the deoptimization code.

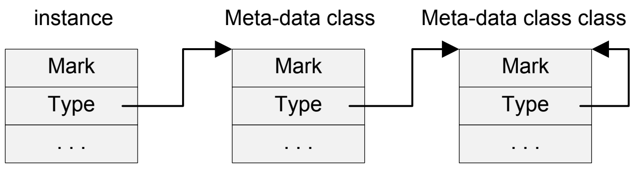


Figure 2.8: Layout of objects on the Java heap.

2.7 Garbage Collector

The VM distinguishes between four different allocation areas: The Java heap, the native method stack, allocation arenas, and the C heap. Every newly allocated Java object is put onto the Java heap. Metadata objects of the VM are also allocated on the Java heap and include a normal Java object header. They are never directly exposed to the Java application, but they are subject to automatic memory management. Temporary VM data structures that do not escape the current method are allocated on the native method stack. Also, data structures that are used for scoping (e.g., for temporary thread transitions) are allocated on the stack. The compilers use allocation arenas to be able to deallocate all their temporary objects by freeing the whole arena after compiling a method. The C heap is only used when all other allocation areas are not suitable. This is because of the error-prone manual deallocation and the bad performance in case of frequent small allocations.

Every object on the Java heap follows the layout shown in Figure 2.8. The object header consists of two heap words: The mark word is used for storing locks on the object and the identity hash code. If a single heap word is not sufficient, the mark word points to an extra heap object (e.g., when an object is locked whose mark word is already used for storing the identity hash code). The second object header word points to another heap object describing the type of the object. The minimal description available for an object is its size and the offsets where the object contains pointers, because the garbage collector requires this information for every object. In case of Java objects, the type word points to the VM class object that contains additional information (see Section 2.3). There is one object with a type pointer that recursively points to itself as shown for the right-most object in Figure 2.8. This object can give the description of itself to the garbage collector.

The Java heap is split into multiple generations. The idea is to separate young objects from objects that already survived several garbage collections. The generation with young objects can then be collected more often, thus benefitting from the observation that in object-oriented programs young objects often die young [37]. The HotSpot VM uses three different generations: The young generation, the old generation, and the permanent generation.

The young generation is divided into a new and an old space. It uses a stop-and-copy collector where only live objects are copied from the new to the old space and then the roles of the two spaces are swapped. When an object survives a predefined number of such young generation collections, it is promoted to the old generation. The permanent generation is reserved for VM-internal data structures such as the metadata objects and class file data.

When the old generation is full, the VM triggers a full garbage collection using a mark-and-compact algorithm. Figure 2.9 shows the four phases of the algorithm.

Mark. The goal of the first phase is to mark all live objects. The VM starts with the objects referenced by root pointers (e.g., pointers on the execution stack or pointers to the VM class objects in the system dictionary), marks them as live and then recursively visits the objects referenced by the marked objects.

Forward Pointers. The VM now sweeps over the heap from the object with lowest to the object with highest address. Whenever it reaches a live object, it calculates the object's destination position after the compaction. The new address of the object is stored at the position of the object's mark word. If the mark word of the object is already in use (e.g., because the object is locked or its identity hash code has been set), the mark word is backed up. The VM maintains a list of backup entries that save a pointer to the object and its original mark word.

Adjust Pointers. The VM needs to adjust the target of every pointer on the heap. Every pointer is updated to point to the forward location of its target. After this phase, the heap is in an inconsistent state, because the pointers are already adjusted to the forward location, but the objects are still at their original location.

Compact. In the last phase, the VM copies the objects from their current location to the forward location. The objects are processed again sequentially starting with the object with lowest to the object with highest address. Finally, the backup list with the mark words is used to restore the mark words of the objects.

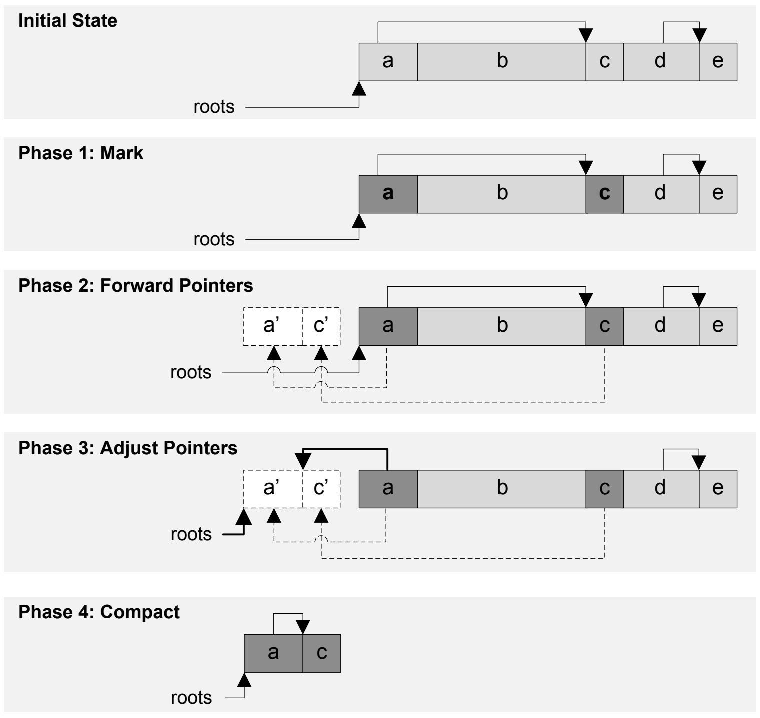


Figure 2.9: The mark-and-compact garbage collection algorithm.

There are VM command line options to select other garbage collection strategies that parallelize the young generation or old generation collection. The algorithms presented in this thesis are however based on the serial garbage collection configuration that uses the mark-and-compact garbage collector.

2.8 Triggering Class Redefinition

Since JDK 1.4, the HotSpot VM is capable of redefining classes at run time. The code integrated into the VM is derived from the work done by Dimitriev [5, 10]. One of the main use cases of his work is dynamic profiling. The NetBeans IDE includes a Java profiler that is implemented using class redefinition [38, 39]. The redefinition can be triggered in three different ways that all result in the execution of the same code within the VM: Via the native Java Virtual Machine Tool Interface (JVMTI) [40], the Java Debug Interface (JDI) [41], or the `java.lang.instrument` API [42].

All three possibilities require an additional agent running within the VM. There is no way to directly change the definition of a class from within the Java application code. This restriction does not have technical reasons, but hints at the fact that class redefinition is designed for use during debugging of Java applications.

2.8.1 JVMTI

The Java Virtual Machine Tool Interface (JVMTI) provides a native code interface into the VM that allows developers to install an agent for monitoring and controlling the running Java application. The agent must be written in C++ and compiled into a dynamic shared library. It can be attached to the VM with the command line option `-agentlib:`*`libaryname`*.

Listing 2.1 sketches an example agent that redefines classes. The `Agent_OnLoad` method is automatically called at VM startup time. An object of type `jvmtiEnv` is used to access the JVMTI methods. The agent can query or change the current state of the VM and also install callbacks for events (e.g., class loading or thread start events). The `RedefineClasses` method takes the number of classes and their bytecodes (as an array of type `jvmtiClassDefinition`) as arguments and immediately triggers class redefinition. The call blocks until the redefinition is completed and returns a JVMTI error code.

```
typedef struct {
   jclass klass;
   jint class_byte_count;
   const unsigned char* class_bytes;
} jvmtiClassDefinition;

JNIEXPORT jint JNICALL Agent_OnLoad(JavaVM* jvm,
                                    char* options,
                                    void* reserved) {
   jvmtiEnv *jvmti; jvm->GetEnv(&jvmti, JVMTI_VERSION_1_2);

   // Use the JVMTI environment to redefine classes.
   jint count = ...;
   jvmtiClassDefinition* classes = ...;
   jvmtiError result = jvmti->RedefineClasses(&count, classes);
   return JNI_OK;
}
```

Listing 2.1: Redefining classes using a native JVMTI agent.

2.8.2 JDI

The Java Debug Interface (JDI) provides a Java API for connecting to a remote VM. The connection is accomplished using a network stream and the Java Debug Wire Protocol (JDWP) [43]. The remote VM must be started with the command line option `-Xrunjdwp:transport=dt_socket,server=y`, such that it accepts incoming client requests. The JDI interface also allows starting the remote VM in a new process on the same machine.

Listing 2.2 shows an example of starting a new VM instance and performing a class redefinition using JDI. A `Connector` object is used to launch a VM in a new process. A `Map` object specifies the classes to be redefined and their new bytecodes. The call to `redefineClasses` blocks until the redefinition is complete and throws an exception if it fails.

```
import com.sun.jdi.*;

// Launch and connect to the VM.
Map<String, ? extends Connector.Argument> arguments = ...
VirtualMachine vm = Bootstrap.virtualMachineManager().defaultConnector().
    launch(arguments);

// Create map with classes that should be redefined.
Map<? extends ReferenceType, byte[]> classes = ...;

// Redefine classes. Throws exception on failure.
vm.redefineClasses(classes);
```

Listing 2.2: Redefining classes in a remote VM using the JDI interface.

2.8.3 Instrumentation API

The `java.lang.instrument` package contains the classes of the Instrumentation API. To access its functionality, a Java application starts with an attached Java agent. The command line option `-javaagent:jarpath` specifies the JAR file of the agent. The manifest of the JAR file must contain an entry `Premain-Class`, which points to a Java class with a static `premain` method as shown in Listing 2.3.

The instance of class `Instrumentation` provides methods for retrieving all loaded classes and to install class file transformers. Such transformers engage in the class loading process as an intermediate step that can change the bytecodes of any class. Also, when a class is redefined with new bytecodes, the VM invokes the callback of the class file transformers. The `Instrumentation` instance is capable of triggering class redefinition as described in Listing 2.3.

```
import java.lang.instrument.*;

class ClassDefinition {
  public ClassDefinition(Class<?> theClass, byte[] theClassFile) { ... }
}

public static void premain(String agentArgs, Instrumentation instr) {
  instr.redefineClasses(new ClassDefinition[] { ... });
}
```

Listing 2.3: Redefining classes using the `java.lang.instrument` API.

Chapter 3

Class Redefinition

This is the main chapter of the thesis in which we present an algorithm for unlimited dynamic class redefinition for object-oriented programs. The supported modifications to classes include adding and removing fields and methods as well as complex changes to the class hierarchy.

Our algorithm is implemented as a modification to a recent version of the open source Java HotSpot VM. However, the general concepts and algorithms presented in this thesis apply to any VM that executes statically typed object-oriented programs. We chose a production-quality, high-performance VM instead of a research VM to make sure that the evaluation results are valid in a production environment. Also, the improvement of the class redefinition capabilities in the HotSpot VM is one of the top-most voted requests for enhancement on the official Oracle website (Bug ID: 4910812) [2].

The implementation is based on the existing mechanism for swapping method bodies developed by Dmitriev [10] and extends it to allow arbitrary changes. Only small parts of Dmitriev's implementation remain unchanged (e.g., basic validity checks on the parameters of the redefinition command or computing the differences in the declared methods between two classes), because the task of performing arbitrary changes to classes requires a different approach than the task of changing only method bodies (see Section 8.7). Our algorithm has the following properties that we consider important for class redefinition:

Unlimited. The algorithm is unlimited in terms of possible changes. The supported changes include changes to the set of declared methods, the layout of instances, and the class hierarchy. Changes to a class that affect its subclasses are properly propagated.

Atomic. The redefinition of a set of classes is performed atomically. This is a requirement, because two changes to Java classes might depend on each other, such that performing either change alone produces an incorrect Java program. An important property of our algorithm is that there is a safe rollback if parts of the redefinition are invalid. The VM always performs either all of the changes or none of the changes.

Noiseless. The additional flexibility of dynamic changes does not decrease the performance of normal program execution. This means that the VM runs at full speed before and after a change. A measurable performance penalty would constrain the adoption of the new class redefinition features.

Transparent. There are no indirections visible in Java stack traces, and the Java debugging experience is not affected negatively in any way. The debugger does not notice whether the currently running program is the original program version or a redefined program version.

Ready. The VM is ready to apply a change at any time the Java threads can be suspended. There are no restrictions on the currently active methods or the currently live heap objects. This is especially important for the use of class redefinition during program development. When a programmer halts a program at a breakpoint and applies a change, he wants the change to become effective immediately.

Comprehensible. The semantics of a change must be clear and comprehensible to the programmer. The VM has a clear and simple strategy for initializing newly introduced fields. More clever strategies or heuristics can always be implemented using the Java debug interface and we do not see them as part of the core of the class redefinition algorithm. However, our VM supports custom transformer methods for converting between two instance versions (see Section 5.1.4).

We focus on implementing class redefinition 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 VM metadata. These changes are small and do not influence the VM during normal program execution. The choice of implementing our approach as a VM modification instead of using bytecode rewriting techniques is especially beneficial for obtaining an unlimited, noiseless, and transparent algorithm. Instead of simulating a class definition change at the bytecode level, we directly modify the underlying VM metadata objects.

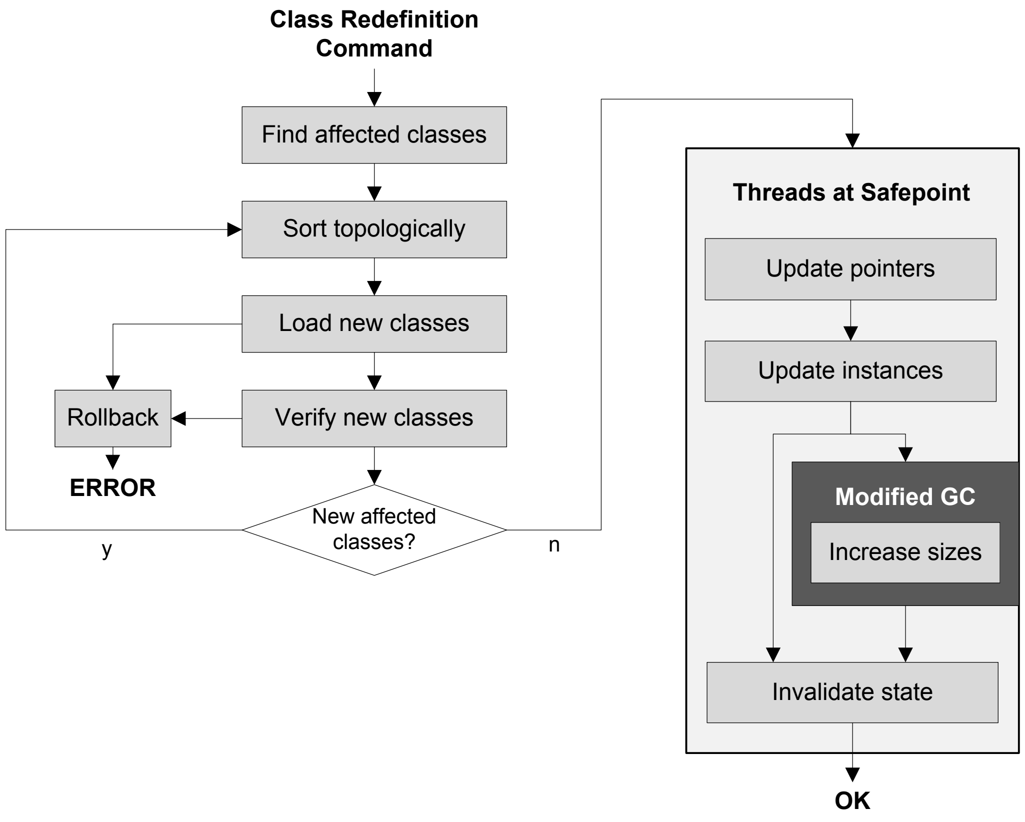


Figure 3.1: Steps performed by the class redefinition algorithm.

3.1 Overview

Figure 3.1 gives an overview of the steps performed by our class redefinition algorithm. The following sections describe the steps in more detail. The class redefinition is triggered using the Java Virtual Machine Tool Interface (JVMTI), the Java Debug Interface (JDI), or the `java.lang.instrument` API (see Section 2.8). The parameter of the redefinition command is an array of pairs, where each pair defines a pointer to the class that should be redefined and a byte array with the new class file data.

First, the algorithm creates a list of all affected classes that contains the redefined classes and their subclasses. The list is sorted topologically based on subtype relationships and defines the order in which the new classes are processed (see Section 3.2). Then, the new classes are loaded and added to the type universe of the VM forming a side universe (see Section 3.3). At this point in time, the VM class objects for the new and the old class versions coexist in the system. This is necessary, because the Java program still runs in parallel to the class redefinition and uses the old class hierarchy. The bytecodes of the new classes are verified using the standard verification mechanism of the VM. If during class loading or verification an error occurs, the class redefinition is rejected, all modifications are undone, and an error code is returned. In this phase of the algorithm, the Java program running in parallel can still load a new class `C` that has to be redefined too, because it is a subclass of a redefined class `A`. In this case, the VM reloads `C` with the new version of `A` as the superclass.

If the change is valid, all threads are suspended at the next safepoint. Additionally, we use global locking to prohibit concurrent compilation and class loading (see Section 3.7). A heap iteration modifies any pointers to old classes to become pointers to the new classes (see Section 3.3.1). During the iteration, we also update those object instances whose size decreased or remained unchanged (see Section 3.4). For objects with an increased size, a modified full garbage collection is performed in which the instance sizes are increased (see Section 3.5). This step is optional and is only performed if there is at least one instance on the heap that needs a size increase. In the next step, the VM invalidates state that is no longer consistent with the new class versions (see Section 3.6). The last step is to release all locks and continue executing the new program version.

3.2 Affected Types

When the layout or the supertypes of a class change, other classes may be affected as well. 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. Changing the set of supertypes also affects the supertype set of subclasses. We also need to verify subclasses again, because a change in the base class can make a subclass invalid (e.g., because the subclass overrides a method that is now declared `final` in the new version). The same rule applies when redefining an interface: All direct and indirect subinterfaces and also all classes implementing the interface need to be redefined, because adding or removing methods of the redefined interface changes entries of their interface method tables.

Therefore, the algorithm needs to extend the set of redefined classes by all their subtypes. Figure 3.2 gives an example with three 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

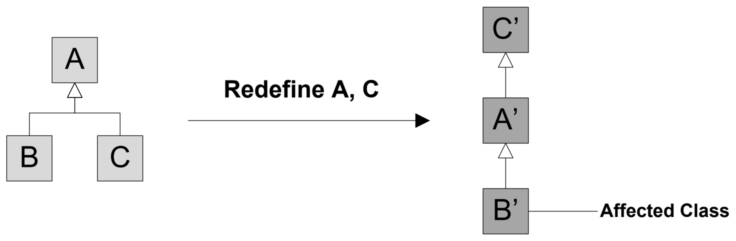


Figure 3.2: Example redefinition that changes the class hierarchy.

needs to be initialized based on the new supertype. This includes its virtual method table and its interface method table.

If a type is affected by the redefinition, but its new bytecodes are not specified in the redefinition command, we use its original bytecodes. The VM does not store the bytecodes of loaded classes, but it has a built-in class file reconstitutor that serializes the loaded information about a class into a byte stream. We use this mechanism to get the bytecodes of the affected but unmodified classes, which in turn are used to reload and re-verify the classes with respect to the new class hierarchy.

The redefinition command does not specify an order in which the new classes must be loaded and redefined. From the user's perspective, the classes must be swapped atomically. Our algorithm sorts the classes topologically 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 can be incompatible with the old version of its superclass. In that case, class loading only succeeds if the superclass is already replaced by its new version. The Java classes and their supertype relationships form a directed acyclic graph (DAG). Therefore, a valid topological order is always possible.

In order to support changes to the class hierarchy, we order the types based on their subtype relationship *after* the class redefinition step and do not 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 determine the new subtype relationships. In the example shown in Figure 3.2, we first redefine `C` to `C′` and subsequently `A` to `A′`, because the class `A` is a subclass of `C` in the new version of the program. Finally, we redefine `B` to `B′`.

Old Program

```
class A {
  public final void foo() { }
  public void bar() { }
}

class B extends A {
  public void bar() { }
}
```

New Program

```
class A′ {
  public void foo() { }
  public final void bar() { }
}

class B′ extends A′ {
  public void foo() { }
}
```

Figure 3.3: Two classes with changes that must be performed atomically.

3.3 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. Additionally, it is the only way to solve the problem of cyclic dependencies between class redefinitions, e.g., when one change requires class `A` to be redefined before `B`, but another change requires `B` to be redefined before `A`. Figure 3.3 shows such an example. Class `A` has two methods `foo` and `bar` and `B` is a subclass of `A`. In the old version `foo` is declared `final` in `A` and `bar` is overridden in `B`. In the new version, `bar` is declared `final` in `A′` and `foo` is overriden in `B′`. Therefore, `A` and `B′` are incompatible and so are `A′` and `B`, because in both cases a `final` methods would be overridden. In order to avoid such inconsistencies when mixing class versions, we build a separate side branch for the new classes. This way, the old class versions and the new class versions do not affect each other.

The Java HotSpot VM maintains a system dictionary to look up classes based on their name and class loader (see Section 2.4). We replace the entry for the old class with the entry for the new class immediately after loading the new class. However, running Java code still sees the old class until the class redefinition becomes effective. We use a thread-local flag to implement these two different views of the system dictionary.

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 3.4, the lookup for class `C` returns `C′`, because class `C` was redefined before `A`. The VM copies the values of static fields of the old class to the static fields of the new class if both name and signature match. We do not execute the static class initializer of the new class. Figure 3.4 shows the state of the type universe after building the side universe for the new class versions.

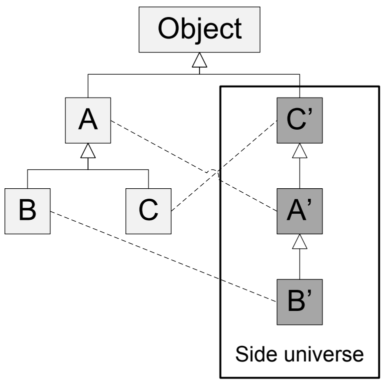


Figure 3.4: Building a side universe for the new version of the class hierarchy.

We maintain pointers between the old and the new VM class object. This helps navigating through the versions during garbage collection and pointer updates. The system dictionary, however, always contains just a since reference to the newest version of a class.

The side universe is only created for the VM class object and not for the Java class object that is used for reflection. The instance of `java.lang.Class` exists only once in the system. It has a field that points back to the VM class object. This field cannot be accessed from a Java application and is used only by VM native methods for getting reflection information. When redefining the classes, the field is changed to point to the VM class object of the latest program version. The `java.lang.Class` object contains an additional field with the current version number of the class that is increased on every class redefinition. The Java code of `java.lang.Class` polls this field's value and automatically clears its cached information when the value has changed.

We verify the new classes before the actual redefinition changes the pointers to the new classes in the system dictionary. This enables us to easily roll back and abort the redefinition if the new classes cause a verification error. In order for the verifier to see already the new class versions, we use a thread local flag. If the flag is set, instead of returning the current class version from the system dictionary, the newest version is returned. If verification of a new class fails, the redefinition is rolled back and a JVMTI error code is returned.

3.3.1 Pointer Updates

The class redefinition becomes effective when the pointers to the old VM class object are replaced by pointers to the new VM class object. This is an irrevocable step, so all checks whether the redefinition is valid have to be done beforehand. Additionally, this step needs to be carried out atomically. Therefore, the VM suspends all running threads at the next safepoint. Now only the redefinition thread is running, thus it can safely iterate over the heap objects and update pointers. During the iteration of the heap objects, we distinguish two types of objects:

No Change: The pointers of VM internal data structures must not be updated. For example, old VM class objects must retain their pointers to other old VM class objects, because updating their pointers to the new version would destroy the sanity of the old side class hierarchy.

Header Update: We check and update the pointers in the object header of instances of user-defined Java classes. However, such an object cannot have fields pointing to VM class objects. Therefore, we can skip the object's contents to speed up the heap iteration.

During the heap iteration, the VM also collects information about objects of redefined classes. In particular, it finds out whether there is at least one object instance that has an increased size due to the redefinition. Additionally, it creates a list of all objects of redefined classes. The list is allocated in a separate area and not on the heap in order to avoid concurrent heap modifications during the iteration.

3.3.2 Class Initialization

The values of the static fields of a class are directly stored as an embedded vector in the VM class object. During class redefinition, the static fields of the new classes are initialized with the values of the static fields of the old classes. Immediately after the version change, all static field accesses target the static fields of the new class version. The static constructor of the new class is only run if the static constructor of the old class was not run before. This can happen when the old class has been loaded and linked, but its static constructor was not yet invoked.

If the static constructor of an old class is currently running, then it continues running after the redefinition with static field accesses targeting the new class. If it has already set values to static fields of the old class in its partial execution, then those values are copied to the new class and therefore continue to be visible to the executing initializer.

3.4 Instance Updates

For updating instances, we need a strategy of how to initialize the fields of redefined instances. We have a simple algorithm that matches fields if their names and types 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 the values `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 VM class object. Figure 3.5 shows an example change that requires an instance update. The field `x` is removed from class `A` and a new field `z` is added. The instance size does not change, but the new field `z` must be initialized with `0`. Also, the location of the field `y` changes from offset 24 to offset 16.

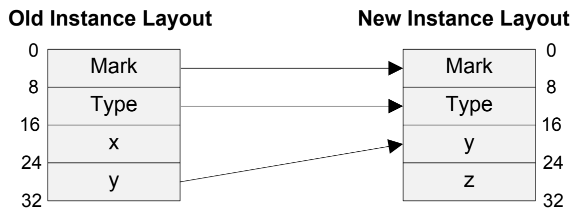
For building the list of necessary update commands, the VM iterates over the fields of the new version of the class. For every field, it tries to find a matching field in the old version of the class (or one of its super classes). If such a field is found (e.g., in the case of field `y`), then a copy update command is created with two parameters: The size of the field and the original memory offset of the field in the old class. If no such field is found (e.g., in the case of field `z`), then a clear command is created with the size of the field as parameter.

Every list of update commands begins with a copy command for the object's header. Two subsequent commands of the same kind are coalesced if possible. The commands are encoded into a flat integer list that ends with the value 0 and encodes the difference between copy and clear commands using the sign bit. Therefore, the total space needed for the update information encoding in the example of Figure 3.5 is six integer values (16, 0, 8, 24, -8, 0).

When updating an object instance, the VM reads the update commands and performs the specified memory copy or clear operations. This makes instance updates faster com-

Old Program | **New Program**

```
class A {                    class A' {
  long x;                      long y;
  long y;                      long z;
}                            }
```



Update Commands *(Encoding)*

Copy 16 bytes from index 0 *(16, 0)*

Copy 8 bytes from index 24 *(8, 24)*

Zero 8 bytes *(-8)*

End *(0)*

Figure 3.5: Encoded update commands when updating object instances.

pared 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. Our approach would also allow us to compile machine code snippets that perform the update. This would improve performance if a high number of object instances participates in the update.

The old version of an instance is overwritten with the new version, so we do not need additional memory for co-existence of all old and new instances. We need however a temporary storage space of the size of one object in case the target and source destination overlap and fields are shuffled. In order not to overwrite values that are not yet copied, the content of the object is first copied into the temporary storage space. The destination area is then filled using the data from the temporary storage space.

If the size of an instance on the heap does not increase, then the update can be carried out immediately, otherwise it is delayed to a subsequent full garbage collection. Because of object alignment (two heap words), adding an instance field does not always imply that the heap size of the class instances increases.

If the size of an instance decreases, the remaining space at the end of the object is turned into a filler object. The filler object is necessary in order to preserve the property that heap objects are adjacent such that it is possible to iterate over them from lowest to highest address. The smallest possible size decrease is two heap words, because of the alignment of heap objects. In this case, we can put a dummy instance of class `java.lang.Object` in the unused area, because such an instance fits exactly. For a decrease larger than two heap words, we fill in a Java byte array. Such an object has at least two heap words for the object header and one heap word for the length of the array. The length is chosen such that the array exactly fits the gap between the old and new object size.

3.5 Garbage Collection Modification

A full garbage collection is only necessary if there is at least one object with an increased size. Those objects cannot be updated in-place and therefore need to be updated during a garbage collection. We modified the mark-and-compact garbage collection algorithm (see Section 2.7) and added the capability to change object sizes during the garbage collection. Although it would be possible to build the modification also into other garbage collection configurations, the DCE VM currently only supports the serial garbage collector.

We adjusted the forward pointer calculation algorithm in such a way that it is able to deal with the problem of increased object sizes. Increased object sizes mean that the object 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 whose destination area ends at a higher memory address than its source area is first rescued to a side buffer. Otherwise the garbage collector would overwrite objects that are not yet copied and would destroy their field values.

Listing 3.1 presents the modified forward pointer algorithm. The algorithm has a variable `forwardTop` that is a pointer to the current end of the compacted heap. During the heap iteration, it is increased by the size of every live object. In our algorithm, we need to distinguish between an old object size (before class redefinition) and a new object size (after class redefinition). The `forwardTop` is increased by the new size of the object after class redefinition. An object is rescued if `forwardTop` plus its new size points to a higher address than the end of the object. The forward destinations for the rescued objects are calculated after the heap iteration such that they are placed at the very end of the new compacted heap. Putting rescued objects at the end of the heap reduces the number of

```
address forwardTop = start of heap

// Compute forward pointers.
for each live object o {
  int oldSize = size of o before class redefinition
  int newSize = size of o after class redefinition
  if forwardTop + newSize > address(o) + oldSize {
    add o to the list of rescued objects
  } else {
    store forwardTop in the object header of o
    forwardTop += newSize
  }
}

// Place rescued objects at the end of the heap.
for each rescued object o {
  store forwardTop in the object header of o
  forwardTop += new size of o after class redefinition
}
```

Listing 3.1: Modified forward pointer algorithm.

rescued objects, because it makes space for other objects to increase their size while still being copied to lower addresses. It has however the disadvantage that the algorithm does no longer preserve the order of objects.

We did not modify the update pointer phase of the mark-and-compact algorithm. The subsequent compaction phase is however adjusted such that rescued objects are copied to a side buffer. We can find out whether an object must be rescued by checking whether its forward pointer is pointing to a higher memory address than the object's location. At the end of the compaction phase, the rescued objects are copied from the side buffer to their destination location. The object layout is adjusted to the new class version as part of this copying.

In the example shown in Figure 3.6, the size of `x` is increased and therefore the object `x` at its destination address would overlap the object `y` and would therefore overwrite its contents during the copying. 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 frees space for 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

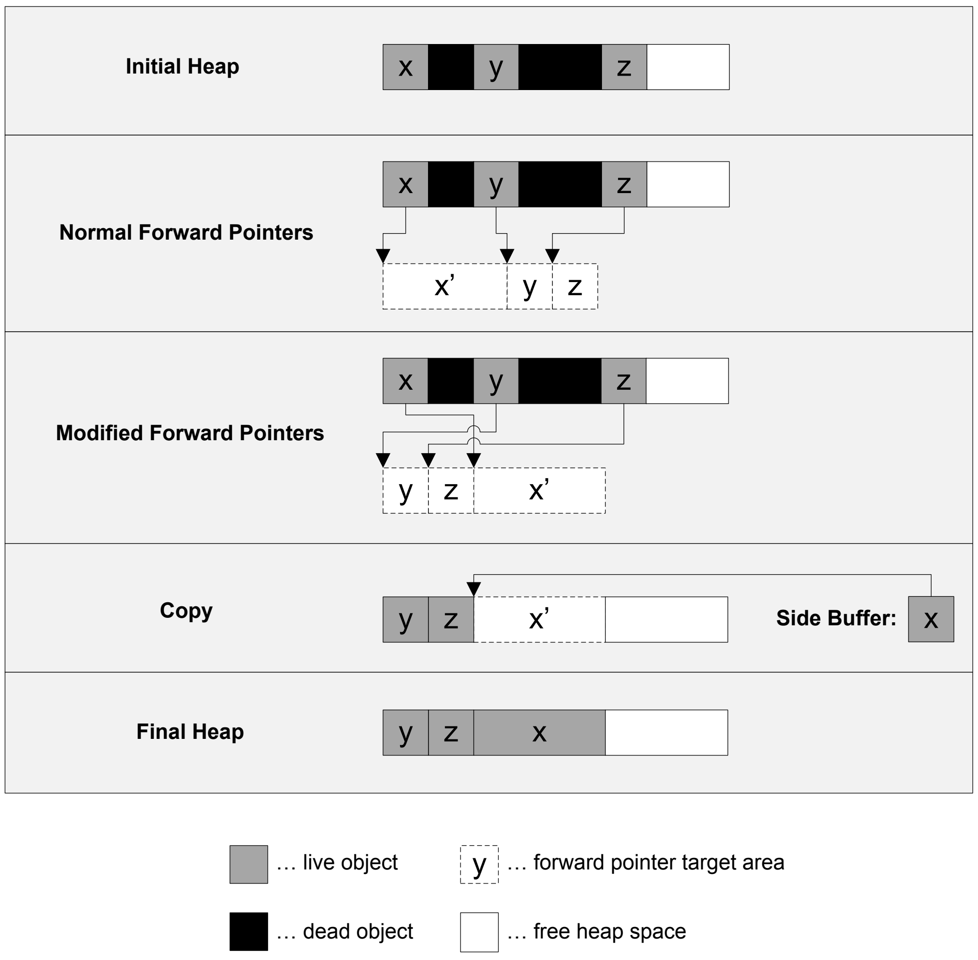


Figure 3.6: Increasing object sizes during a modified mark-and-compact garbage collection.

objects and therefore the necessary size of the side buffer. In the compaction phase, objects `y` and `z` are processed normally, while the object `x` is copied to the side buffer. Afterwards, the new version of `x` is constructed based on the data of the old version in the side buffer.

3.5.1 Permanent Generation Objects

The permanent generation contains the VM class objects and also some Java instance objects (e.g., the instances of `java.lang.Class`). During the compact phase, the garbage collector finds out about the size of a Java instance object by accessing its class pointer in the object header. Therefore, the VM class object has to be already compacted before the garbage collector reaches an instance. Normally, this is guaranteed, because the VM class objects are always allocated before one of their instances and thus occupy areas with smaller memory addresses. However, our class redefinition algorithm can violate this condition, because the new version of the class is allocated at a higher memory address than previously allocated object instances of the class.

The VM solves this by reordering the relevant objects in the permanent generation: Permanent generation object instances of redefined classes are always copied to a rescue buffer and inserted at the end of the heap after the compaction phase. This is necessary to make sure that they are located at a higher memory address than the new VM class objects of the redefined classes although they were allocated earlier. This also means that if a class with instances in the permanent generation is redefined, the garbage collection is always necessary. Most notable examples of such classes are `java.lang.Class`, `java.lang.String`, and `java.lang.Object`.

3.6 State Invalidation

The changes performed by class redefinition violate several invariants in the VM. The Java HotSpot VM was not developed with code evolution in mind and made 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 class redefinition 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 VM has a built-in mechanism to stop executing the optimized machine code of a method, called deoptimization (see Section 2.6). If there is an activation of the method on a 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 this 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 VM maintains a cached version of the constant pool for each 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 class redefinition are field entries (the offset of a field is cached) and method entries (for a statically bound call a pointer to the method object, for a dynamically bound call the virtual method table index is cached). We clear all constant pool cache entries. 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. As a performance optimization, the VM could only clear cache entries that refer to methods or fields of redefined classes. However, it is not clear which entries of the new program version will need to be resolved again. Furthermore, checking each entry whether it needs to be cleared slows down the class redefinition itself. Therefore, we simply clear all entries. We report the combined performance penalty of method deoptimization and constant pool clearance in Section 7.2.

3.6.3 Class Pointer Values

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

3.7 Locking

At the time of actually performing the change, the class redefinition runs in a single thread with all other threads being suspended at safepoints. During the time of loading and verifying the new classes however, the Java application threads and the just-in-time compiler threads run in parallel.

We need to use locking mechanisms in order to prevent parallel class redefinition. Additionally, we need to prevent parallel compilation and parallel class loading for the main part of the redefinition algorithm that updates the pointers and object instances. During the instance and pointer updates, we also need to make sure that no other thread is running.

Figure 3.7 shows the locks and the order in which they are acquired. Immediately after receiving the class redefinition command, a global lock makes sure that only one thread is currently performing a class redefinition. This is a requirement, because it is only possible to build a single side universe that links back to the current class versions. During loading of the new classes, the callback methods of the class file transformers are called (see Section 2.8.3). They must not trigger a new class redefinition, because this would result in a deadlock.

Concurrently running compilations are also a problem for class redefinition. Even if the compiler is paused during the redefinition, the continued compilation after replacing the old class with the new class version can cause a VM crash. The compiler could access an entry of the constant pool of the new class at an index read from the bytecodes of the old class. Therefore, we wait for all active compilations to finish and prevent new compilations until the end of the redefinition step. In order to stop the compiler more quickly, we set a bailout flag that is regularly checked by the compiler. When the flag is set, the current compilation is aborted and its results are discarded.

We acquire a class loading lock and then search the class hierarchy for new affected classes that were loaded in parallel. If new affected classes are found, the class loading and the compilation lock are freed and the new classes are processed. We need to release the locks, because class loading and verification can again trigger new class loading. Also, we must not hold a class loading lock while calling the registered class file transformers.

After building a side universe for all affected classes, we can proceed with the redefinition by stopping all threads at the next safepoints. During the pointer and instance updates,

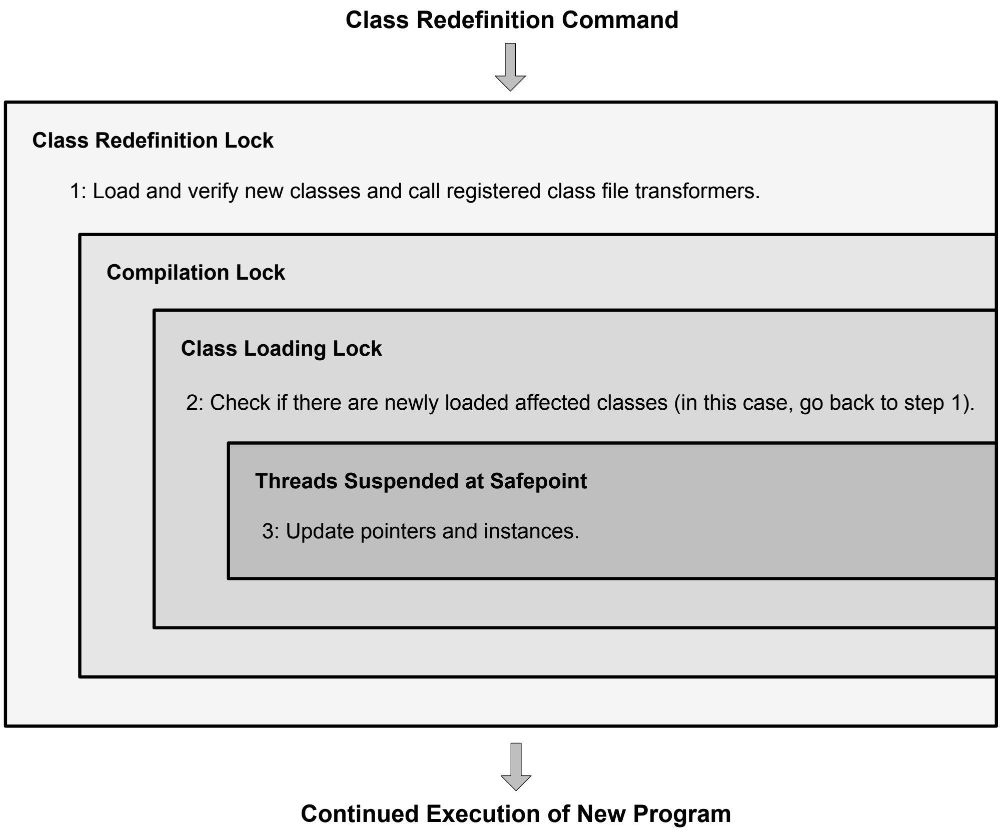

Class Redefinition Command
Class Redefinition Lock
1: Load and verify new classes and call registered class file transformers.
Compilation Lock
Class Loading Lock
2: Check if there are newly loaded affected classes (in this case, go back to step 1).
Threads Suspended at Safepoint
3: Update pointers and instances.
Continued Execution of New Program

Figure 3.7: The four different locks acquired during the class redefinition.

every Java thread must be suspended. After this step, the class redefinition is completed and the VM can release all acquired locks and continue program execution.

3.8 Limitations

The VM can perform arbitrary updates including complex changes to the class hierarchy. However, it does not check the correctness of an update. Deleting a field or method can lead to an exception being thrown in continued program execution as outlined in the next chapter. In Section 4.1.1, we describe a further enhancement of the DCE VM with the ability to call methods that have been deleted in the new classes. The most dangerous update, however, occurs when a supertype is removed, because this can cause type safety problems and lead to a VM crash. In Section 4.2, we discuss this problem and present a solution in Section 4.2.1 as an extension to our base algorithm. Table 3.1 gives an overview

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 3.1: Supported code evolution changes of the base version of the DCE VM.

of the supported code evolution changes as classified in Section 1.2 and possible problems during continued program execution.

An additional limitation is that several system classes have a special meaning to the VM and therefore must not be redefined in arbitrary ways. The following list outlines the most important restricted classes:

java.lang.Object: The object class has two characteristics that must also be true for every redefined version. First, it must not have any supertype. By definition, the object class is a supertype of any other type. Therefore, declaring a super interface or a superclass for `java.lang.Object` introduces prohibited circularities in the class hierarchy. Second, it must not have any instance field. This limitation could be relaxed, but would require significant changes to many parts of the VM. As every array type is also a subtype of `java.lang.Object`, it would affect the layout of arrays. Also, several algorithms in the VM assume that the size of a `java.lang.Object` instance is fixed to the size of the object header (i.e., two heap words).

Adding methods and static fields is however allowed. Changing the existing methods (e.g., `hashCode` or `equals`) is possible, but if the just-in-time compilers are enabled, such a change is ignored in compiled code. The compilers remove invocations to those methods by special machine code sequences that reproduce the behavior of the methods as specified by the JDK implementation. Therefore, a change to those methods has no effect in compiled code. Similar restrictions apply to other methods that are well-known to the compiler.

java.lang.ref.Reference: The reference class and its subclasses (`PhantomReference`, `SoftReference`, and `WeakReference`) are treated specially by the garbage collector. They have a special field that points to the reference target. This field is not considered a normal object pointer field. For class redefinition, the imposed restrictions are that the class hierarchy relationships between those four classes must not change and it is not allowed to add an additional reference field to the classes.

java.lang.Class: The VM installs three fake pointer fields into this class immediately after the object header. They are used for VM-internal purposes and reference the VM class object, the VM array class object, and the default constructor of the class. The location of those fields is fixed and therefore it is not allowed to add a superclass that defines an instance field to `java.lang.Class`.

There are several additional system classes whose fields are accessed by the VM using a field offset that is calculated at VM startup time. Therefore, changing the field layout of system classes is discouraged as in many cases it damages the correctness of the VM. Modifying the VM implementation to reinitialize the offset on every class redefinition is possible but requires a significant engineering effort.

Chapter 4

Binary Incompatible Changes

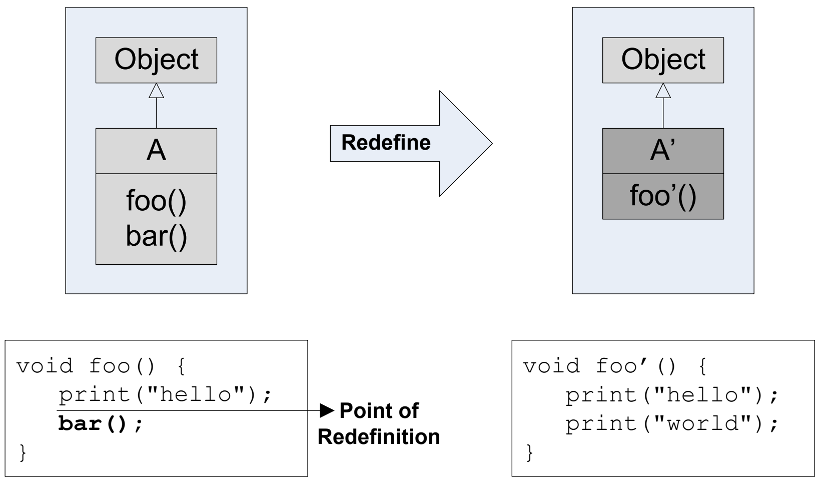
This chapter discusses dynamic class redefinition in the context of binary compatibility. A detailed formal definition of binary compatibility of Java programs can be found in the paper of Drossopoulou et al. [44]. We present two extensions to our algorithm that improve the support for binary incompatible changes.

For a change to be binary compatible, every class member access that was valid before a change must still be valid after the change. Additionally, the subtype relationship between two classes must be preserved. Binary compatible changes to a class are: Adding a field, adding a method, or adding an interface to the set of implemented interfaces. Adding a method to an interface or adding an abstract method to a base class is considered a binary compatible change in the context of class redefinition, if the verification of each subclass is still succeeding (i.e., the subclass is either abstract itself, or implements the newly introduced method in its new version).

Binary incompatible changes to a class are: Removing a field, removing a method, or removing a supertype. Removing a supertype can either mean changing the super class to a class that is not a subclass of the original super class, or it can mean removing an interface from the set of implemented interfaces.

4.1 Deleted Methods and Fields

As long as only method bodies are swapped or fields and methods are added to classes, the old bytecodes can execute normally. However, when a field or method is deleted, old bytecodes are possibly no longer valid. In our VM, old code continues to be executed when



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

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

Figure 4.1: Redefinition example that deletes a method.

old methods are active on the stack at the time of redefinition, so it can happen that old code accesses a deleted field or calls a deleted method.

Figure 4.1 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 the old code. Therefore, it reaches the call to the deleted method `bar`. The interpreter tries to resolve the reference to `bar` (because we cleared the constant pool cache during the redefinition step, see Section 3.6). The resolution fails to find the method and the interpreter throws a `NoSuchMethodException`.

The new version of `foo` is correct because it no longer calls `bar`, but the continued execution of the old version of `foo` causes the problem. It is possible to develop heuristics for switching from the stack values and the 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 both valid and intuitive for developers.

The following section describes a better way to handle the access to deleted class members in the interpreter. In Chapter 5, we present a further extension to the DCE VM with the ability to read a specification of safe update regions from the class files and then safely switch from the old execution to the new one.

4.1.1 Executing Multiple Versions

We distinguish four different possibilities of dealing with the problem of deleted members. The user selects one of these behaviors on a per-class level by specifying a custom bytecode attribute in the new version of a redefined class. A custom bytecode attribute is a mechanism to add information to a class file while keeping backward compatibility (see Section 4.7.1 in the Java Virtual Machine Specification [45]). A VM that does not recognize the attribute must be able to skip its contents. It is possible to specify different behaviors for dealing with methods, static fields, or instance fields.

Static check: A static reachability analysis checks whether the continued execution of the program can reach an instruction referring to a deleted member. If this can happen, then the class redefinition is rejected. The VM looks at the stacks of all threads at the time of redefinition. If a currently executing method is being redefined, then it performs a reachability analysis starting from the currently executed bytecode index until the end of the method. The analysis of the bytecodes of called methods is not necessary, because a call always targets the newest version of a method. This behavior can block redefinition if a member is deleted that is accessed from a long-running loop.

Dynamic check: The redefinition is always performed and the execution of old methods resumes in the interpreter. When the interpreter 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`. This is the default behavior of the DCE VM.

Access deleted members: This behavior avoids throwing an exception, but accesses the old deleted member. The behavior is only supported for methods and static fields. The access of deleted instance fields still causes an exception. This is because we remove deleted instance fields from all active instances and therefore cannot access their data after the redefinition step.

Access old members: Calling a new method from an old method is dangerous if the semantics of the target method changed in the new version of the program. Therefore, our VM also offers a behavior where a member access is always performed from the viewpoint of the version of the executing method. An old method will call the old method version based on the compatible class version even if a newer version of the method exists. This behavior can delay the redefinition to take effect, because a thread starts executing the new program version only after returning from the methods with modified bytecodes. However, this behavior makes sure that functional changes to a method do not cause semantic problems for the caller.

4.1.2 Old Member Access

This section describes our algorithm for accessing older versions of class members and deleted class members. We make sure that old code always executes in the interpreter, so we do not need to modify the compiler for this functionality. After class redefinition, the interpreter's constant pool cache for resolved methods and fields is cleared. When resolving a member again, the interpreter checks the policy for accessing deleted members of the method's class.

If the policy is that old members should only be accessed when they do no longer exist in the new version of the program, the interpreter first tries to resolve the member based on the new version. Only if this fails, the interpreter tries to look up the old version of the member. When looking for a member of an old class version, it uses the latest class version of the target class that is *compatible* with the class of the executed method. Two specific versions of two classes are compatible if they coexisted at any time as the latest version of their class. The versions of the VM class objects are organized in a doubly-linked list such that the VM can easily iterate over them. We also keep the constant pools of old classes and the bytecodes of old methods in the VM.

Figure 4.2 shows an example with a source class s and a target class t. There are seven different revisions of the program, with t being redefined five times and s being redefined two times. The classes t', t'', and t''' are all compatible target classes of the source class version s'. The class t' is compatible with both s and s' because it coexisted as the latest version with both of them (with s in revision 2 and with s' in revision 3). The interpreter looks up the latest compatible target class. If the method holder is class s' then t''' is used; if the method holder is s then t' is used as the compatible target class. If the accessed member is also not found in the compatible target class, then throwing an exception is the correct behavior.

Listing 4.1 shows the algorithm for selecting the correct version of the target class that should be chosen for resolving the class member. The source class is the class that defines

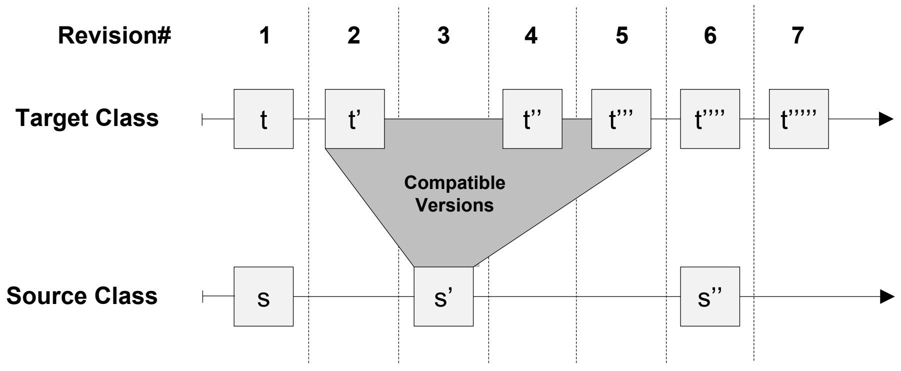


Figure 4.2: Compatible versions when a method of the source class accesses a member of the target class.

the currently executing method. The target class is the dynamic type of the receiver in case of virtual calls and interface calls. For static field accesses or calls, the target class is the defining class of the accessed static member. The target class is always the newest version of a class.

When accessing a field of class t while executing an old method of class s, an example input for the `getCompatibleClass` method would be `source=`s' and `target=`t'''''. At first, we select the class that was used to redefine the source class. If no such class version exists, then the source class is at the latest version and we use the latest version of the target as the compatible target class. Otherwise, we set the variable `newerSource` to the newer version of `source` (i.e., s'' in our example). Now, we iterate over the versions of the target class and stop at the first version that has been loaded before the `newerSource` class (i.e., t'''). This class is the selected compatible target class and we perform the class member look up based on this class.

For virtual calls and interface calls, the interpreter would usually save the table index in the constant pool cache to be faster on subsequent executions of the bytecode. However, in this case receiver objects with different types may resolve to different table indexes. Therefore, the table index is not constant for a certain call site and must be resolved every time.

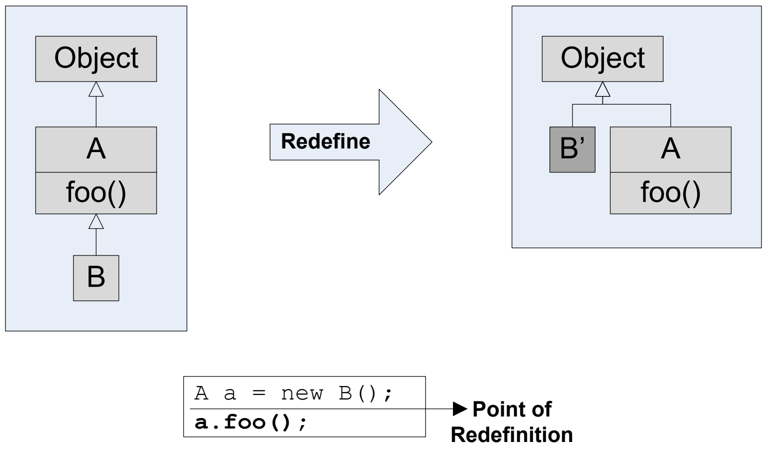
```
Class getCompatibleClass(Class source, Class target) {
  Class newerSource = newerVersion(source)
  if (newerSource == null) {
    // Source is at the latest version =>
    // we must use the latest version of target.
    return target
  }
  while (target is not younger than newerSource) {
    target = olderVersion(target)
  }
  return target
}

Class newerVersion(Class c) {
  if (c is the latest version of the class) {
    return null
  }
  return the VM class object that replaced c during a class redefinition
}

Class olderVersion(Class c) {
  if (c is the first loaded version of the class) {
    return null
  }
  return the VM class object that was replaced when redefining c
}
```

Listing 4.1: Finding the latest compatible target class for a source class.

4.2 Removed Supertypes

When the set of all direct and indirect supertypes of a class increases, old code can execute as normal. It does not use instances of the class as instances of their added supertypes, but executes as before. On the other hand, when the set is narrowed, old code is possibly no longer valid. We call such a change to be a *type narrowing* change.

It is a necessary invariant in a statically typed VM that at any point in time, the type of a value is a subtype of the declared static type of the field or local variable that is holding the value. Otherwise, the type safety can no longer be guaranteed and this will most likely result in a VM crash when the value is used the next time.



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

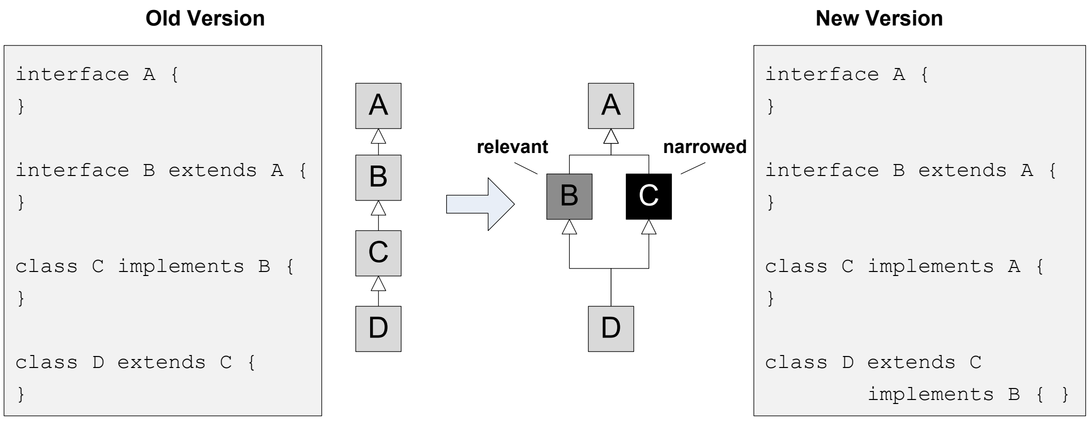
Figure 4.3: Example class redefinition with type narrowing.

Figure 4.3 gives an example for a type narrowing change. 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 4.3, the call a.foo() no longer makes sense, because B does neither declare nor inherit a method foo. The interpreter will try to look up the target method using the virtual method table index of foo. The class B does however no longer contain the virtual method table entry. This leads to a call with an undefined target address.

4.2.1 Safe Dynamic Type Narrowing

In order to allow the safe removal of a supertype, the VM needs to guarantee that the subtype relationship between the static and the dynamic type of variables and fields is valid right after a class redefinition. Additionally, it has to verify methods again to make sure that the relationship is not violated in continued execution. In the example of Figure 4.3, the assignment of the newly created instance of type B to the local variable of type A is valid in the old version of the program. In the new version of the program, such an assignment is however not allowed and results in a verification error.

The VM places two boolean flags on the types: A type is *narrowed* if one of its original supertypes is no longer a supertype in the new version of the type. A type is *relevant* with respect to the type narrowing check, if it was removed from the set of supertypes



Old Version

```
interface A {
}

interface B extends A {
}

class C implements B {
}

class D extends C {
}
```

New Version

```
interface A {
}

interface B extends A {
}

class C implements A {
}

class D extends C
        implements B { }
```

Figure 4.4: A class redefinition change that includes type narrowing with class `B` being relevant and class `C` being narrowed.

of a narrowed type. Using this definition, the check whether a type narrowing change is safe only has to be performed if the static type of a variable or field is a relevant type. Additionally, the check is only necessary if the type of the value is a narrowed type. Only if the check is necessary taking the declared type and the dynamic type into account, the VM actually performs a costly subtype check between the two.

Figure 4.4 shows a type narrowing example with four types `A`, `B`, `C`, and `D` involved. The classes `C` and `D` are redefined. The set of supertypes of `C` is narrowed, because `B` is no longer a supertype of `C` in the new version. The set of supertypes of `D`, however, remains unchanged. In both versions `A`, `B`, and `C` are supertypes of `D`. Therefore, instances of class `D` can never violate the relationship between dynamic and static type after the redefinition. For checking values, interface `B` is marked as relevant, because `C` is no longer a subtype of `B` in the new version. All other types were not removed from any supertype set.

The VM calculates a check map for each class that specifies at which field offset a pointer needs to be checked. Only fields with a declared type that is relevant are added to the map. A class with an empty check map means that its instances are ignored in the heap iteration. Object arrays are processed by iterating over their elements only if their declared element type is relevant.

For checking the local variables and the expression stack, the VM walks over the execution stacks of all active threads. Each object pointer in the stack frames is checked. If its dynamic type is a narrowed type, the VM tries to find out the declared type of the value.

This is only possible for local variables listed in the `LocalVariablesTable` attribute in the bytecodes. If the declared type is not available, then the VM conservatively rejects the redefinition when the value is an instance of a narrowed type.

In addition to violating values, the VM also has to make sure that loaded bytecodes still pass verification taking the new class hierarchy into account. The verifier relies on subtype relationships (e.g., in assignments) and therefore a change in the hierarchy can also change the verification result. We trigger verification of all methods whose classes are not affected by the redefinition. The methods of redefined classes are already verified when the new class versions are loaded. The algorithm also has to verify the old methods that are currently active on the stack, because those methods will complete their execution after class redefinition. They also have to be correct with respect to the new class hierarchy.

In order to avoid the costly verification of all classes, we do a quick check of the constant pool of a class. Every type that is used in a class is referenced in the constant pool of this class (see Section 2.1). Therefore, the verification of methods can be skipped, if the constant pool of their class does not contain a reference to a relevant type.

Only if all checks are positive, the type narrowing class redefinition is performed. Otherwise, the VM rolls back any changes and aborts the class redefinition by returning an error code. Listing 4.2 describes our algorithm in pseudo code.

```
check() {
  // Check pointers on the heap.
  for each heap object o {
    if o is an object array {
      if o’s element type is relevant {
        for each element e in o {
          subTypeCheck(e’s value, element type of o)
        }
      }
    } else if o is an instance object {
      for each field f whose type is relevant {
        subTypeCheck(f’s value, f’s declared type)
      }
    }
  }

  // Check pointers on the execution stack.
  for each thread {
    for each stack frame s {
      for each local variable l of s {
        if l has a declared object type {
          subTypeCheck(l’s value, l’s declared type)
        }
      }
    }
  }

  // Check bytecodes of old currently running methods.
  for each thread {
    for each stack frame s {
      if the method m of s is the old version of a redefined method {
        verify m with respect to the new class hierarchy.
        if verification fails { rollback() }
      }
    }
  }

  // Check bytecodes of loaded methods.
  for each loaded class c {
    if c was not redefined {
      if the constant pool of c references a relevant type {
        for each method m of c {
          verify m with respect to the new class hierarchy.
          if verification fails { rollback() }
        }
      }
    }
  }
}

// Checks the validity of a value.
subTypeCheck(value, staticType) {
  dynamicType = value’s dynamicType
  if dynamicType’s supertype is narrowed {
    if dynamicType is not a subtype of staticType { rollback() }
  }
}

// Rolls back the redefinition.
void rollback() {
  undo all changes made for class redefinition
  return from class redefinition with error code
}
```

Listing 4.2: An algorithm that checks whether a type narrowing class redefinition change is safe.

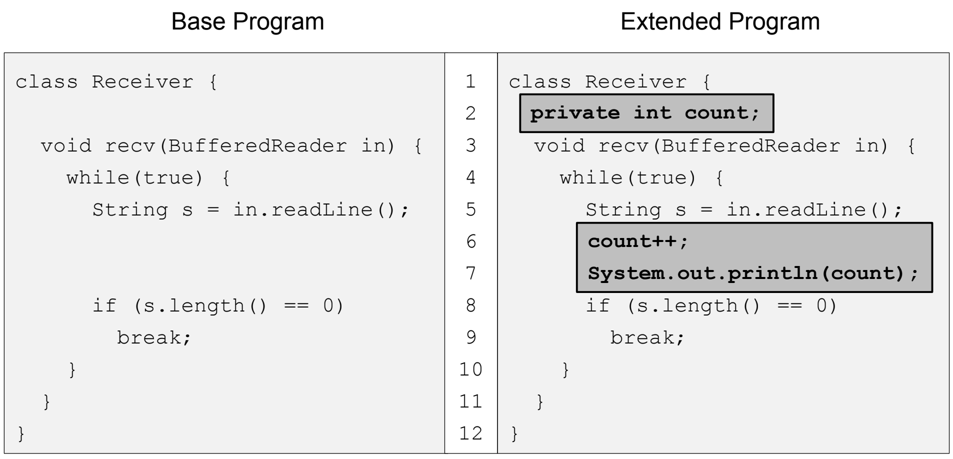
Chapter 5

Safe Version Change

The class redefinition techniques as outlined in Chapter 3 and Chapter 4 define changes between two program versions at the level of fields, methods, and supertypes. In this chapter, we take this one step further by looking at the detailed differences between two methods at the bytecode level. In particular, we outline a new solution for the problem of changing Java methods that are active on the stack at the time of redefinition. This is especially helpful for updating methods with long-running or endless loops (e.g., the message receiving loop of a server application).

We define a programming model that allows us to switch between a *base program* and an *extended program*. The extended program must be derived from the base program by adding fields and methods but not removing them. The extended program must not change the type hiearchy of base program classes, but it may load new classes. The bytecodes of previously existing methods may be modified, with some restrictions that are explained in this chapter. First, we describe how to switch from a base program to an extended program (see Section 5.1). Then we explain how we can switch back safely (see Section 5.2). Finally, we show how the model can be extended to switch between two arbitrary programs by extracting their common base program (see Section 5.3).

Our programming model describes conditions and rules for the specification of the extended programs, but the VM does not provide an algorithm for automatically deriving the specification from the differences between two programs. The implementation of the safe version change in the VM and how the specification of the extended program can be added to Java class files are described in Section 5.4.



```
class Receiver {

  void recv(BufferedReader in) {
    while(true) {
      String s = in.readLine();


      if (s.length() == 0)
        break;
    }
  }
}
```

```
class Receiver {
  private int count;
  void recv(BufferedReader in) {
    while(true) {
      String s = in.readLine();
      count++;
      System.out.println(count);
      if (s.length() == 0)
        break;
    }
  }
}
```

Figure 5.1: A base program that reads lines of text from an input stream and an extended program that additionally counts the number of lines.

5.1 Changing to the Extended Program

The left part of Figure 5.1 shows an example base program implementing a receiver loop that reads text lines from a `BufferedReader` object. The loop is exited when an empty line is read. The right hand side shows a possible extension of the base program that counts and prints the number of received text lines. It adds a field `count` to the class and increases its value in the loop. Additionally, it adds a `println` statement.

The version of the DCE VM as presented in Chapter 3 can correctly perform the class redefinition of the `Receiver` class from the base version to the extended version. The newly introduced `count` field is initialized to 0 and the bytecodes of the `recv` method are updated. However, currently running activations of the `recv` method are unaffected and continue to execute the bytecodes of the base version.

We extended the DCE VM with the ability to switch atomically from the base program to an extended program at any time. The switch is performed instantaneously. If a modified method is currently active on the stack, the program counter immediately switches to the equivalent position in the new version of the method. Thus, currently executing receiver loops are immediately affected by the change in our example program. They start printing the number of received text lines since the redefinition.

For every position in the base program, the VM must know the corresponding position in the extended program that should be used for continued program execution after the redefinition. This is necessary, because all threads executing a base method at the time of redefinition will immediately change to executing the new extended version of the method. Therefore, the bytecodes of a method in the base program must not be changed in arbitrary ways. In particular, the extended program must be derived from the base program by only adding new bytecodes to the existing bytecodes of the base program. We define a bytecode region in the base program that matches a code region in the extended program as a *base code region*. In the example of Figure 5.1, lines 4 to 5 and lines 8 to 10 form base code regions. Newly introduced code is contained in *extended code regions* (e.g., lines 6 to 7 in the example).

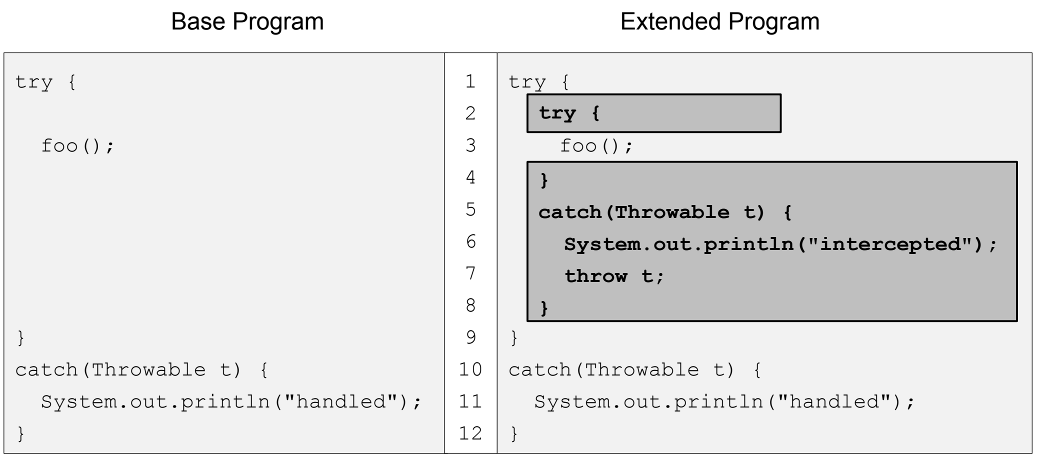
5.1.1 Class Definition Changes

The extended program may add class members to the base program. Also, the extended program can refer to new classes that are not used in the base program. Changes to the access modifiers of members are allowed as long as it does not change the outcome of accessibility checks triggered by bytecodes in base code regions. Changing the signature of a field or method is prohibited. The same applies to changes to the modifiers `static`, `synchronized`, and `native`. An `abstract` base method can be replaced with a concrete implementation in the extended version, but not vice versa.

The extended program may however not override a non-abstract base method. Overriding a method changes the semantics of the dynamic dispatch at the call site. At the time of the version change, the program could currently execute a method that was invoked due to a dynamic dispatch in the base program, while the same dynamic dispatch in the extended program would have led to the execution of a new method. This would invalidate our claim of an atomic version change. The only allowed change to the class hierarchy is to add an interface to a class. Changing the superclass of a class is also prohibited, because it could result in new overridden methods.

5.1.2 Method Body Changes

When the bytecodes of a method are different in the base program and the extended program, we distinguish between the *base method* and the *extended method* respectively. We define the modifications to the Java bytecodes of the base method that are allowed when forming the extended method as follows:



```
try {

  foo();






}
catch(Throwable t) {
  System.out.println("handled");
}
```

```
try {
  try {
    foo();
  }
  catch(Throwable t) {
    System.out.println("intercepted");
    throw t;
  }
}
catch(Throwable t) {
  System.out.println("handled");
}
```

Figure 5.2: An extended code region that intercepts an exception that is thrown by a base code region.

Constant Pool References: A bytecode of the base program may be modified to contain a different constant pool index as long as the referenced constant pool entry represents the same constant. The new and old bytecode are then said to be *equivalent modulo constant pool* [10].

Extended Code Regions: An extended code region is a sequence of n bytecodes that is used to extend the base program. It can be inserted before any bytecode of the base program with some index i. In the Java bytecodes of the base program, every reference to a bytecode with an index greater than i must be adjusted by adding n. For references to i, the adjustment by n is optional. Such references are jump or branch targets, exception handler ranges or targets, and line number entries. The adjustment of jump targets of the base program guarantees that the new extended code region can only be entered at the first bytecode.

In order to avoid semantic problems when an extended program causes parts of the base program to be skipped, we furthermore define the following restrictions on control flow modifications:

Jumps and Branches: Jump or branch instructions in extended code regions must always target bytecodes in the same code region. Therefore, an extended code region cannot be exited with a jump or branch instruction.

Return: The extended code sections may contain a return instruction. This is the only way how parts of the base program can be skipped.

Exception Handlers: An extended code region must catch any exception that it throws. It is allowed to add exception table entries that cover a range within an extended code region. The exception handler block must however be within the same code region.

Exception Interception: An extended code region may intercept an exception thrown by a base program bytecode. It must however rethrow the same exception again. Figure 5.2 shows such a case. The `try` statement itself does not produce a bytecode, therefore the extended code section that intercepts the exception is continuous to the base code section at the bytecode level.

With the above constraints, the VM can match every bytecode position in the base method to a corresponding bytecode position in the extended method.

The extended code regions and the base code regions are using the same execution stack. We restrict the way the execution stack can be modified by the extended code regions in the following way:

Expression Stack: The expression stack height upon exit of an extended code region must be the same as the expression stack height at the entry of the region. Additionally, all expressions on the stack at the entry of an extended code region must remain unmodified.

Local Variables: The extended code region may introduce new local variables and also read from and write to all local variables of the base program. The Java verifier ensures that the bytecode accesses to local variables are consistent with their types. If a local variable is only modified by either the base program or the extended program, then independent verification of both program versions is sufficient. Otherwise *cross verification* of the two method versions is necessary as explained in the next section.

As the version change can happen at any bytecode position of the base program, the extended code regions cannot assume that any other extended code region was executed before them. Therefore, an extended code region must not rely on the initialization of local variables in other extended code regions.

Base Method		Intermediate		**Extended Method**
iconst_1	1	iconst_1	1	iconst_1
istore_1	2	istore_1	2	istore_1
	3		3	**fconst_1**
	4		4	**fstore_1**
iconst_0	5	iconst_0	5	iconst_0
	6	**fload_1**	6	**fload_1**
	7	**fstore_2**	7	**fstore_2**
return	8	return	8	return

Figure 5.3: Problems with verification of two versions of a Java method.

5.1.3 Cross Verification

Figure 5.3 shows an example of a base method and an extended method that both verify correctly, but the version change would still be invalid. At line 6, the local variable with index 1 is of type `float` in the extended program. This is necessary, because the extended program performs a `fload_1` instruction at this line. If the version change happens at line 5, the type of the local variable would however be `int` (set by the execution of `istore_1` in the base program). The bytecodes in the middle of the figure show the executed bytecodes sequence for this case.

In order to prevent this problem, the verifier must perform a *cross verification* of the two programs, if the extended and the base code regions write to the same local variable. It infers the possible types of local variables in the base program after every bytecode that is followed by an extended code region (i.e., the bytecodes at line 2 and 5 in Figure 5.3). For each of these lines, it starts a modified verification pass of the extended program: It sets the types of the local variables to those inferred during the base program verification and then performs a verification beginning at the corresponding extended code region (i.e., the regions starting at line 3 and 6). This makes sure that the bytecode sequence shown in the center of Figure 5.3 is also subject to verification.

5.1.4 Transformer Methods

Fields added in the extended version of the program are initialized with 0, `false`, or `null`. An extended program could however require other initialization values for its fields. Figure 5.4 shows such an extended program. The `buffer` field is initialized in the constructor of the `Receiver` class to a new `StringBuilder` object. At the time

```
class Receiver {
  StringBuilder b;
  public Receiver() { b = new StringBuilder(); }
  void $transformer() { b = new StringBuilder(); }
  void recv(BufferedReader in) {
    while(true){
      String s = in.readLine();
      b.append(s);
      if (s.length() == 0)
        break;
    }
    System.out.println(b);
  }
}
```

Figure 5.4: An extended program with a transformer method.

of the version change, there can however already be `Receiver` objects on the heap. For those objects, the VM offers the possibility to specify *transformer methods* for manually converting between the old version and the new version of the object. For every class, the programmer can write a transformer method with no parameters, a `void` return type, and the name `$transformer`. This method is then called for every instance of that class that exists on the Java heap. The example transformer method initializes the `buffer` variable to an empty `StringBuilder` object. The initialization is guaranteed to be executed before entering any extended code regions.

The two extended code regions in the method `recv` of Figure 5.4 both require that the value of `buffer` is properly initialized. Therefore, it would not be possible to model the variable as a local variable in the extended program that does not exist in the base program. By defining `buffer` as a field, it is possible to write an adequate transformer method that initializes its value.

For the purpose of static initialization, it is also possible to specify a method called `$staticTransformer` in the extended program version. All transformers are executed on the thread that triggered the version change. Static transformers are always executed before instance transformers. All transformers are guaranteed to be executed before any extended code region is entered. It is therefore safe to initialize new fields in transformer methods and use them in extended code regions.

5.2 Changing Back to the Base Program

The DCE VM is also capable of converting from the extended program back to the base program. This is for example useful when the extended program adds logging output to the base program that should later be disabled again.

When converting back from the extended program to the base program, the VM deletes fields and methods that are only defined in the extended program. In order to prevent any exceptions in continued program execution, the VM guarantees that those deleted class members are never accessed after the change. This is achieved by applying the update only at a point where it can immediately change all methods on the stack to their new version. The update is delayed until the method activations of every thread are in base code regions and not in extended code regions. Only then the update is safe, therefore we refer to base code regions also as *safe update regions*. There is no guarantee that a safe update region will be reached by all threads. Therefore, the command for changing back to the base program has to specify a timeout. If there is a thread that does not reach a safe update region within the given time, the operation gracefully fails.

There are two additional restrictions on the difference between the base program and the extended program in order to make switching back to the base program a safe operation:

Verification: For a type-safe conversion back to the base method, another kind of cross verification is necessary. The VM performs a modified verification of the base program for each extended code region. It infers the types of local variables at the end of every extended code region. Then, it starts the verifier at the base program bytecode following the code region using the inferred types as the initial types of the local variables. This makes sure that changing from the extended to the base program is a valid operation for each base program position.

Type Narrowing: When changing from the base to the extended program, it is allowed to add a superinterface to a class. When switching back, this would however result in a type narrowing change. Therefore, such a change is only valid if there is no local variable or field violating the subtype relationship between its static and dynamic type. The VM performs a stack and heap analysis and cancels the operation if it finds such a violation (see Section 4.2.1). The verification pass over the currently active old methods is not necessary, because those methods will not continue to execute as in the version of the DCE VM as presented in Chapter 3. If the added and removed interface is only known to the extended program, then the type narrowing change is always safe, because the base program cannot have fields or variables of the type of the interface.

5.2.1 Additional Restrictions

To make the change back to the base program safe, we need to make sure that the base program is not aware that it was executed in an extended version. This imposes additional restrictions on the kind of changes allowed to form the extended program. We suggest to refrain from using the following actions in extended code regions to ensure correctness of the continued base program execution after the switch back:

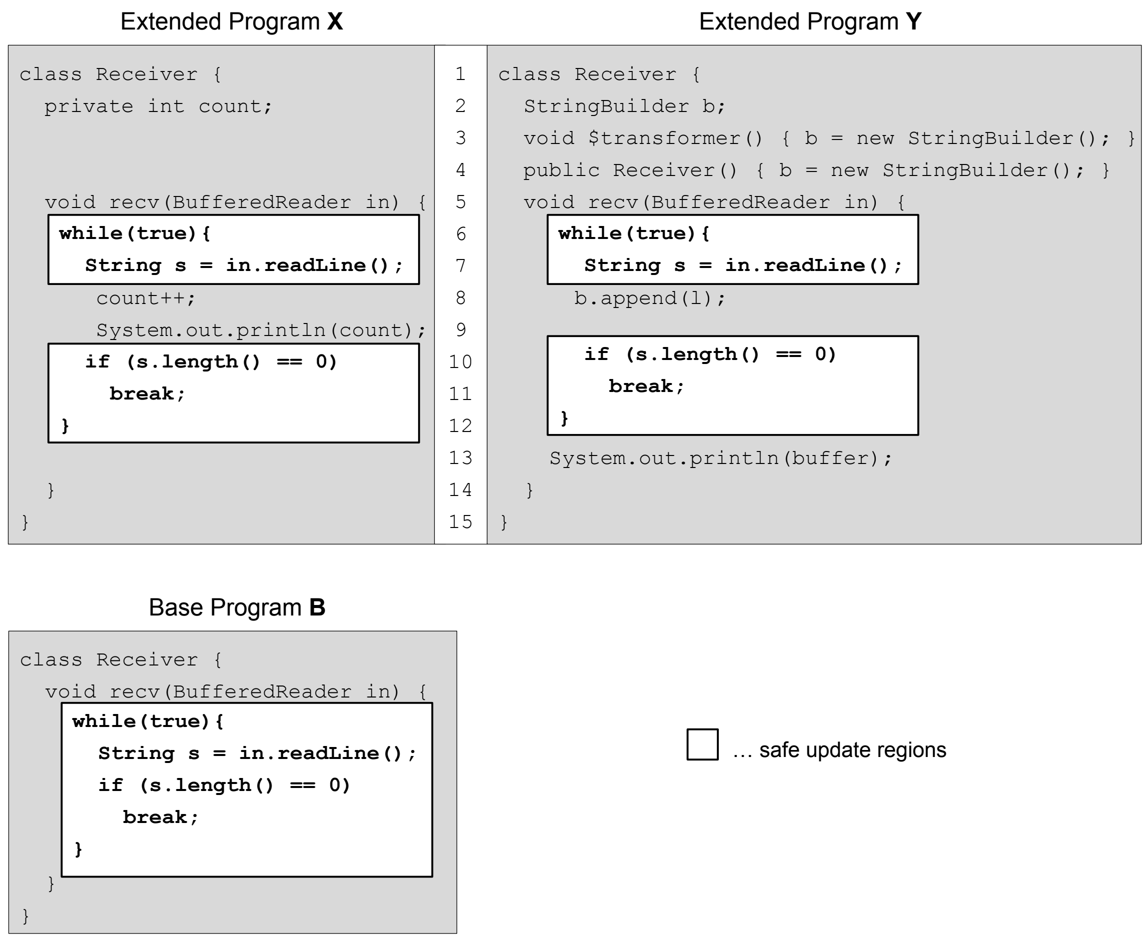
State: In order to sustain invariants of the base program, the extended code regions should not write to local variables and fields known to the base program.

Return: A `return` statement in the extended code region may lead to base program code with side effects not being executed. Additionally, it could violate invariants on the return value of the modified method or just change the base program behavior because of the different return value.

Call: Calling a base program method from an extended code region should only be allowed if the called methods do not modify the state of the base program.

If one of those actions is used, possible negative effects on continued base program execution have to be checked manually. If all specified restrictions are followed, then the base program is completely unaware that it has been executed in an extended version. Only timing differences due to the execution of extended code regions can result in a different base program behavior.

The two extended programs presented in Figure 5.1 and Figure 5.4 both do not use any of the problematic actions. They only access their own local variables and fields. There is a call to the base program method `println`, but this method does not change the state of the base program. Therefore, it is completely safe to switch between the base program and any of those two extended programs.



Extended Program **X**

```
class Receiver {
  private int count;


  void recv(BufferedReader in) {
    while(true){
      String s = in.readLine();
      count++;
      System.out.println(count);
      if (s.length() == 0)
        break;
    }

  }
}
```

Extended Program **Y**

```
class Receiver {
  StringBuilder b;
  void $transformer() { b = new StringBuilder(); }
  public Receiver() { b = new StringBuilder(); }
  void recv(BufferedReader in) {
    while(true){
      String s = in.readLine();
      b.append(1);

      if (s.length() == 0)
        break;
    }
    System.out.println(buffer);
  }
}
```

Base Program **B**

```
class Receiver {
  void recv(BufferedReader in) {
    while(true){
      String s = in.readLine();
      if (s.length() == 0)
        break;
    }
  }
}
```

Figure 5.5: Changing between two program versions.

5.3 Changing between Two Arbitrary Programs

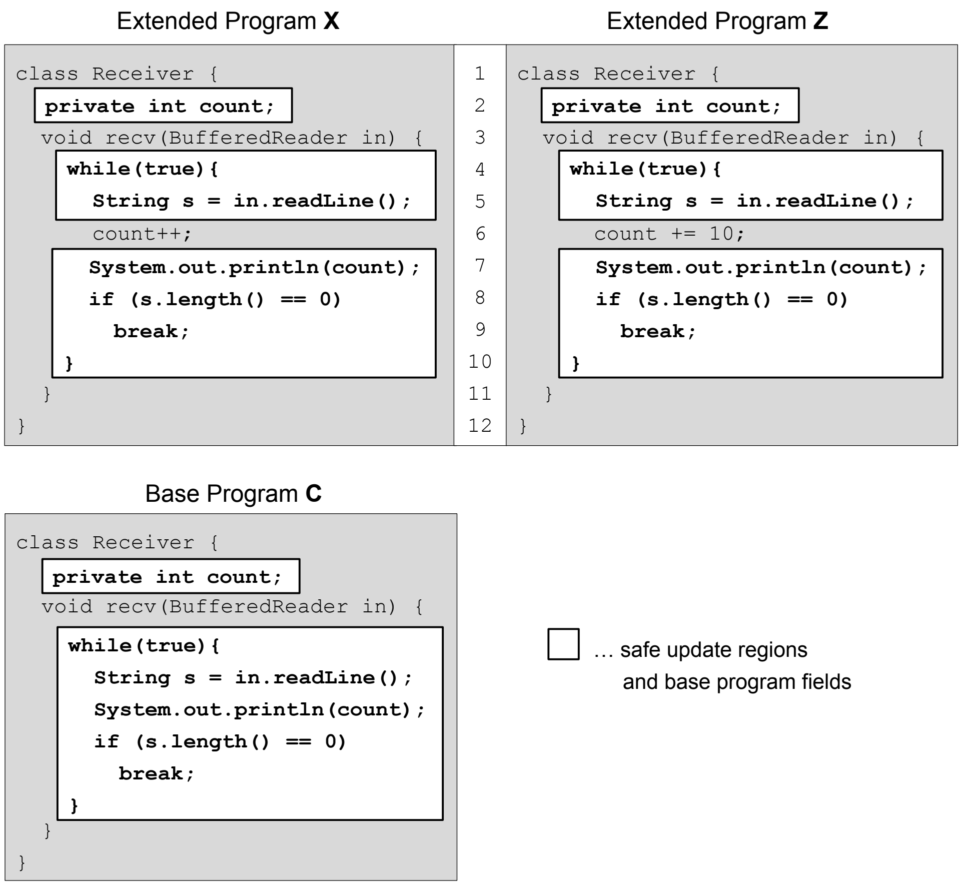
The restrictions for the differences between the base program and the extended program that were presented in the last two sections are severe and prohibit many possible changes if the currently running program is considered to be the base program. However, we can define a safe update between a currently running extended program X and a new extended program Y. This enables us to safely change between two arbitrary program versions by defining their common base program. A safe update is possible, if a valid base program B can be extracted from a comparison of X and Y. The safety of the transformation from X to Y can then be evaluated based on the safety of the transformation from X to B and from B to Y (including possible state transformers defined by Y).

Figure 5.5 shows the two extended receiver loop programs. The program X on the left side that counts the number of text lines and the program Y on the right side that accumulates the text lines in a StringBuilder object. The common base program is marked in dark gray. All of the restrictions between the extended program and the base program as defined in Section 5.1 and Section 5.2 are followed. Therefore, we can safely switch between the two extended program versions. The base program is never actually executed, but only serves as a definition for the safe update regions between the two versions.

When changing between three program versions X, Y, and Z, different base program definitions can be used. The right side of Figure 5.6 introduces a program Z, which is a third version of the example program. It increases the count variable by 10 instead of 1 as in program X. The common base program C between X and Z can now also include the call to println and the field count. The common base program between Y and Z is program B as listed in Figure 5.5.

The definition of the additional base program member count means that the value of the counter is preserved between the two program versions. It is however a violation of the condition that extended programs must not write to fields of the base program. The programmer needs to be aware of this possible problem and manually decide whether the transition between X and Z with C as the base program is semantically safe (i.e., that the value produced by one program does not violate the invariants of the other program). Another possibility would be to exclude count from the base program and perform the version change without preserving the field value. The println statement can be added safely to the base program C and has the effect that the transition between X and Z is also possible while this statement is currently executing.

As long as two Java programs X and Y share a common main method, it is always possible to define a common base program. The most trivial base program consists of the main method with an empty body. However, the change between those two programs will not succeed, unless program X has not yet entered the main method. The more parts of the two programs can be defined as a common base, the more likely a change succeeds, because a precondition for the change to happen is that all threads of the executing program can be gathered in base program regions. Also, every class member that is not defined in the base program has to be initialized by program Y using transformer methods.



```
class Receiver {
  private int count;
  void recv(BufferedReader in) {
    while(true){
      String s = in.readLine();
      count++;
      System.out.println(count);
      if (s.length() == 0)
        break;
    }
  }
}
```

```
class Receiver {
  private int count;
  void recv(BufferedReader in) {
    while(true){
      String s = in.readLine();
      count += 10;
      System.out.println(count);
      if (s.length() == 0)
        break;
    }
  }
}
```

```
class Receiver {
  private int count;
  void recv(BufferedReader in) {
    while(true){
      String s = in.readLine();
      System.out.println(count);
      if (s.length() == 0)
        break;
    }
  }
}
```

Figure 5.6: Changing between two program versions that share an instance field.

5.4 Implementation

We implemented the functionality of switching between program versions on top of the base algorithm for class redefinition as described in Chapter 3. The change still has no negative effect on any of the properties of the DCE VM and using the feature is optional.

For specifying the ranges of extended code regions and base code regions, we do not change the class file format, but use the built-in possibility to specify a new custom bytecode attribute with the name "CodeRegions". The attribute specifies the extended code regions as a list of tuples. Each tuple specifies a single extended code region with its starting bytecode position in the extended program and its length. Every extended code region must begin with a `NOP` instruction. This `NOP` is used as a special marker bytecode, which enables us to perform actions at the code region entry in the interpreter, while the code generated by the just-in-time compiler is not affected.

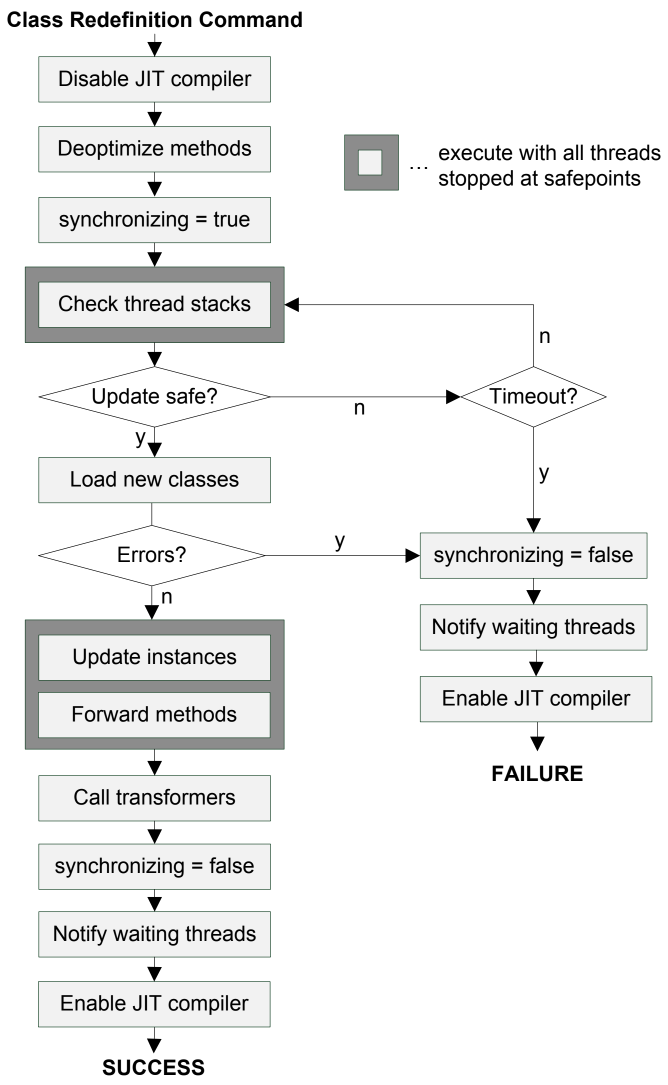


Figure 5.7: Steps performed by the class redefinition algorithm for a safe version change.

The safe version change is triggered by a standard class redefinition command that specifies the bytecodes of all classes modified in the new version. The only difference to a normal redefinition is the optionally specified extended code regions in the "CodeRegions" bytecodes attribute. Figure 5.7 shows the steps performed in the VM for a safe version change. The base class redefinition algorithm is extended with four parts that are explained in detail in the following sections: Compilation prevention, update synchronization, method forwarding, and transformer execution.

5.4.1 Compilation Prevention

Every extended code region begins with a `NOP` instruction. We modified the implementation of the `NOP` instructions in the interpreter to enable update synchronization at the beginning of extended code sections as explained in the next section. When a `NOP` instruction is executed, the interpreter can look up the extended code region that starts at the bytecode index of the `NOP` instruction. The `NOP` instructions are ignored by the just-in-time compilers. This guarantees that despite the presence of extended code regions, the peak performance of the VM is not negatively affected.

We must make sure that at the time of the safe version change, the artificial `NOP` instructions are executed by the interpreter. Therefore, we deoptimize methods that contain extended code regions after receiving the redefinition command. Also, we temporarily disable the just-in-time compiler for those methods. Both steps are necessary to guarantee that during the update synchronization, methods with extended code sections are always executing in the interpreter. After the redefinition is complete, we enable the just-in-time compiler again for all methods.

5.4.2 Update Synchronization

For a safe version change, we need to make sure that every thread is in a safe update region and not in an extended code region. We do this with two mechanisms:

- If a thread is in a safe update region, it is prevented from entering an extended code region.
- We iteratively check the threads, whether all of their activations are in safe update regions.

The first mechanism is implemented as a modification to the `NOP` template of the interpreter (see Section 2.5 for a detailed description of the interpreter and its bytecode templates). When the global `synchronizing` flag is set, then the interpreter checks whether the current `NOP` instruction is at the start of an extended code section. If this is the case, it iterates over the stack frames of the currently executing thread. If there is at least one stack frame that is currently in an extended code region, then the interpreter continues execution. Otherwise, it suspends execution because it is now safe for the current thread to perform the version change.

We iteratively check whether it is safe for all threads to perform the version change. If there is at least one stack frame that is in an extended code region, we keep on waiting. If all of them are outside extended code regions, we can guarantee that none of them will enter such a region before the version change. Therefore, we can guarantee that we will be able to find a corresponding bytecode position in the new program for all methods active on the stack and can safely perform the update.

Whether a safe update region is reached by all threads within a given time depends on the properties of the program and therefore cannot be guaranteed. The user may specify a timespan that the algorithm tries to wait until the version change is rejected. There is a flag `-XX:ForwardTimeout` that specifies the number of milliseconds that should be waited for the threads to reach safe update regions.

5.4.3 Method Forwarding

Now, we can perform the standard class redefinition algorithm that loads the new classes and then updates the pointers and instances. In addition to the normal redefinition, the algorithm iterates over all active threads and looks at the bytecode index of active methods. If the method is a forwarding method (i.e., the old or the new method version defines an extended code region), we check the current bytecode index. If the method is currently in a base program region, we immediately perform the forwarding by changing the current bytecode index and the current method of the interpreter frame. We can safely do this, because it is guaranteed that the new bytecode is equivalent to the currently executing bytecode (i.e., even if the interpreter has already executed parts of the bytecode, continued execution is safe). If the execution of the method is currently halted at the `NOP` instruction at the entry to an extended code region, the forwarding will happen after we wake up the thread.

The calculation of the new bytecode index is done in a two step process: First, we calculate the base program index based on the current bytecode index. Second, we can translate the base program index to an index in the new version of the method.

5.4.4 Transformer Execution

After the redefinition, threads waiting at normal safepoints immediately start running (they will however not be able to enter extended code regions), while threads waiting at extended code regions are still blocked. It is therefore still guaranteed that no thread executes an extended code region. At this time, we execute the static transformer methods and the instance transformer methods for all active instances of classes.

After executing the transformers, we notify the threads that are waiting at extended code region entries. They will immediately recognize that they need to transfer the current execution point to the appropriate bytecode index in the new version of the method. Also, we enable the just-in-time compiler again for all methods.

Chapter 6

Case Studies

This section presents three case studies of tools built on top of the DCE VM with enhanced dynamic code evolution features. First, we present a solution for live modifications to the GUI of an active Swing dialog using a modified version of the NetBeans IDE. Second, we discuss how to implement dynamic mixins for Java. Finally, we describe a tool that combines AspectJ and the DCE VM to provide dynamic aspect-oriented programming features.

6.1 Modified NetBeans

Redefining classes changes the behavior of a program, but it does not change its state. This is especially problematic in case of changes to the configuration of a program that is read at startup. The application would have to modify its state based on the changes to the configuration in order to apply the change without a restart.

The following example motivates the difference between changes to the behavior of a program and changes to its configuration. Listing 6.1 shows a class representing a global buffer using the singleton pattern. The buffer is represented by a byte array of fixed size. The size is specified in the static final field `BUFFER_SIZE`. When the programmer changes the value of the field (e.g., because he wants to increase the buffer size), the effects on the running application are not immediately clear. If the singleton instance already exists, the change has no effect to the running application. Only a restart would make the change effective, because the part of the program that is influenced by the change can only be executed once. If the instance was not yet created, the change is immediately valid. A

```
class GlobalBuffer {
  public static final int BUFFER_SIZE = 100;

  private static GlobalBuffer instance;

  public static GlobalBuffer getInstance() {
    if (instance == null) {
      instance = new GlobalBuffer();
    }
    return instance;
  }

  private byte[] buffer = new byte[BUFFER_SIZE];

  public void write(byte[] data) {
    ...
  }
}
```

Listing 6.1: Problems with changes to declarative program specifications.

manually written transformer method is necessary to ensure that the change to the size of the buffer becomes effective in both cases. A change to the behavior of the buffer by modifying the `write` method will however affect the program immediately.

6.1.1 Mantisse GUI Builder

The Mantisse GUI builder generates code that initializes the state of a Swing dialog. Similar to the example with the singleton pattern, this code is supposed to run only once before displaying the dialog. Therefore, any changes to the dialog (e.g., changing its title or moving a button), only take effect after restarting the dialog.

Listing 6.2 shows the code that is generated by the Mantisse GUI builder for the dialog with a button and a text box that is shown in Figure 6.1. The dialog is represented by the `NewJFrame` class that extends `JFrame` and has one field for the button and one field for the text box. The constructor calls the automatically generated method `initComponents()` that creates the frame contents (i.e., the button and the text box) and adds them to the frame layout. In order to support action events, the dialog implements the `ActionListener` interface to handle a button click. The `addActionListener`

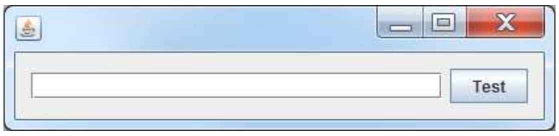


Figure 6.1: Example dialog with a button and a text box.

```
public class NewJFrame extends JFrame implements ActionListener {
    private JButton button;
    private JTextField textField;
    public NewJFrame() { initComponents(); }
    // Code for dispatching events from components to event handlers.
    public void actionPerformed(ActionEvent evt) {
        if (evt.getSource() == button) { buttonActionPerformed(evt); }
    }
    private void initComponents() {
        button = new javax.swing.JButton();
        button.setText("Test");
        button.addActionListener(this);
        textField = new javax.swing.JTextField();
        // ... add button and textField to layout ...
    }
    private void buttonActionPerformed(ActionEvent evt) {
        System.out.println("Hello world");
    }
}
```

Listing 6.2: Code generated by the Mantisse GUI builder for an example Swing dialog.

statement in the `initComponents` method is also automatically generated by the GUI builder. The only line of the code that is manually written by the programmer is the `println` statement in the `buttonActionPerformed` method.

When the user changes the dialog in the GUI builder (e.g., moves a component to a different place), the `initComponents` method is regenerated and the Java source file is recompiled. The class redefinition mechanism can then load the new version of the dialog. However, the change only affects new instances of the dialog, because the `initComponents` method is not called again for existing dialogs.

6.1.2 Modified GUI Builder

To overcome the limitation that a class redefinition does not affect active instances of a dialog, we modified the code that is generated by the GUI builder [33]. Our system allows us to apply changes to a Swing dialog while it is running. We preserve the state of the dialog components (e.g., the content of a text area or a list view). Therefore, the programmer can enhance an existing dialog without the need to reinitialize its state, which is especially beneficial for complex dialogs.

Listing 6.3 shows the code that is generated by the modified GUI builder for the example dialog. The differences to the standard GUI builder code as presented in the previous listing are highlighted. The change is still triggered by NetBeans using the standard class redefinition mechanism, but the modified generated code makes sure that any change to the dialog immediately takes effect. To make this possible, the generated code includes an instance transformer method that clears the contents of the dialog and then calls the `initComponents` method. This transformer method recomputes the layout and reinitialize the GUI components using the `initComponents` method after modifications to the dialog class.

We also changed the code generated for the `initComponents` method, such that it can be called more than once. In the changed version, a new instance of a Swing component is only created when it is added to the dialog for the first time. If an instance of a component already exists (i.e., the field generated for the component is not `null`), then the existing instance is used. This makes sure that the components preserve their states (e.g., the text in a `JTextField` component remains unmodified after class redefinition).

Another change to the generated code is necessary to ensure the correct semantics of registering listeners. The registered listeners of a component are first removed before adding new listeners based on the current dialog specification. This ensures that listeners are not registered multiple times for one component when the `initComponents` method is executed more than once. The listeners are always implemented by the main dialog class. Because type narrowing changes are problematic, we make sure that the set of listeners implemented by the dialog class only expands. Adding a new interface to the class always succeeds with the DCE VM.

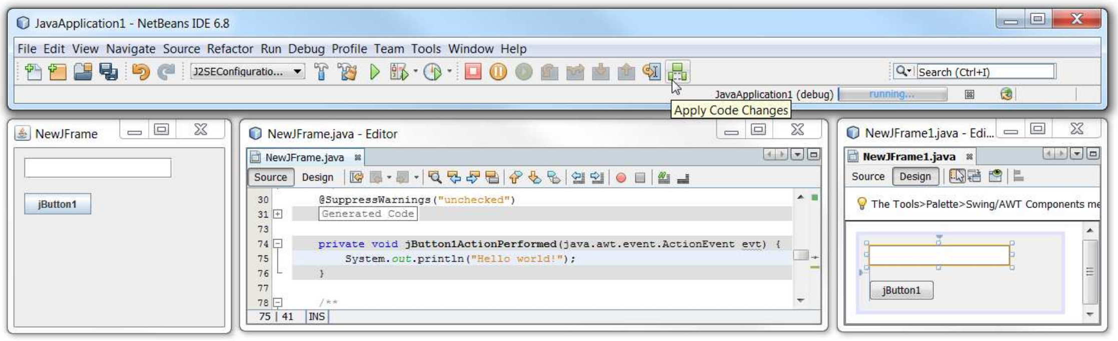
Figure 6.2 demonstrates how NetBeans and the DCE VM can be used to modify a Java GUI application without a restart. The Java application is running all the time and need not be manually suspended by the debugger. The programmer has three different views on

```
public class NewJFrame extends JFrame implements ActionListener {
    private JButton button;
    private JTextField textField;
    public NewJFrame() { initComponents(); }
    void $transformer() {
        getContentPane().removeAll();
        initComponents();
    }
    // Code for dispatching events from components to event handlers.
    public void actionPerformed(ActionEvent evt) {
        if (evt.getSource() == button) { buttonActionPerformed(evt); }
    }
    private void initComponents() {
        if (button == null) button = new javax.swing.JButton();
        button.setText("Test");
        button.removeActionListener(this);
        button.addActionListener(this);
        if (textField == null) textField = new javax.swing.JTextField();
        textField.removeActionListener(this);
        // ... add button and textField to layout ...
    }
    private void buttonActionPerformed(ActionEvent evt) {
        System.out.println("Hello world");
    }
}
```

Listing 6.3: Code generated by the modified GUI builder with additionally introduced code regions highlighted.

the dialog: The running dialog where he can interact with the components on the left side, the Java source code of the frame class in the middle, and the edit mode representation of the frame in the Mantisse GUI builder on the right side. When the dialog is changed in either the source code window or the GUI builder window, a toolbar button is enabled to apply the code changes directly to the running application. The modified version of the program is loaded and the instance transformer method ensures that the layout of the components is recomputed and the components are reinitialized.

In contrast to the DCE VM, the current version of the Java HotSpot VM would not allow us to implement the described extension of the Mantisse GUI builder, because there is neither support for adding fields or methods at run time nor for transformer methods. However, these features are essential, since the Mantisse GUI builder often creates new



```
    @SuppressWarnings("unchecked")
    Generated Code

    private void jButton1ActionPerformed(java.awt.event.ActionEvent evt) {
        System.out.println("Hello world!");
    }

    /**
```

Figure 6.2: Screen shot that shows a running Java GUI application that can be modified using the DCE VM and the "Apply Code Changes" feature of the NetBeans debugger.

fields (e.g., for new components) or new methods (e.g., an action handler for a button). Additionally, the transformer method is necessary to re-layout the frame.

6.2 Dynamic Mixins

Bracha and Cook presented mixins as a new inheritance mechanism [46]. A *mixin* is a class that can be added to *target classes* in order to extend their functionality. A mixin adds new fields, methods, and supertypes directly to the target class and therefore changes the layout and behavior of all target class instances and all instances of subclasses of the target class. In contrast to subclassing, however, target class instances cannot be used as mixin class instances.

A mixin that is added to a target class at compile time is a *static mixin*. Static mixins for Java can be supported by either inserting their fields and methods into the bytecodes of the target class or by weaving them into the class using aspect-oriented programming techniques. The DCE VM allows to dynamically apply such changes to the bytecodes of the target class. Therefore, the DCE VM supports *dynamic mixins* that are added to target classes at run time.

Kerstin Breiteneder and Christoph Wimberger implemented a mechanism that merges the bytecodes of the mixin class and the target class [47]. The result of this merge is a new class definition for the target class. The DCE VM is then used to swap the target class definition and update its instances.

```
interface Z {
  public int getZ();
  public void setZ(int z);
}

public class MixinZ implements Z {
  private int z;
  public int getZ() { return z; }
  public void setZ(int z) { this.z = z; }
}
```

Listing 6.4: Mixin example that adds a new field to a target class.

Listing 6.4 shows an example definition of a mixin class that defines an integer field `z`. It also defines an interface with methods to access the field. The mixin implements this interface. Due to limitations of the Java type system (i.e., it is not possible to have more than one superclass), instances of the target class cannot be used as instances of the mixin class. Therefore, the additional interface declaration is necessary to expose the mixin methods. All interfaces of the mixin class are automatically added to the target class.

Listing 6.5 defines a target class `Point2D`. At any time while the program is running, the mixin class can be added to the target class using the `Mixin.addMixin` method. After this call, all instances of `Point2D` have an additional field `z`. Additionally, they can now be safely cast to the `Z` interface.

By merging mixins with their target class and by using dynamic code evolution we obtain a mixin approach that is both fast and memory efficient. In order to simulate the behavior of adding additional fields to existing instances, a hash table would be necessary in which the keys are the object instances and the values are the additional fields. Then, accessing the additional data would always need a lookup in the hash table. In our example, the newly introduced `z` field is accessed as efficiently as any other field of the `Point2D` class. The call to the interface method `setZ` can be inlined by the just-in-time compiler, thus eliminating any overhead. Also, the hash table workaround needs more memory due to the hash table data structure. Finally, our solution to dynamic mixins does only change the target class bytecodes and does not require any changes to classes except the target class.

```
public class Point2D {
  public int x;
  public int y;
}

Point2D p = new Point2D();

// Add the mixin class to the target class.
Mixin.addMixin(Point2D.class, MixinZ.class);

// Set a new value for the z coordinate using the
// interface implemented by the mixin class.
((Z)p).setZ(10);
System.out.println("Z coordinate is: " + ((Z)p).getZ());
```

Listing 6.5: Adding a mixin to a class that is modeling 2-dimensional points.

6.3 Dynamic Aspect-Oriented Programming

The new features of the DCE VM open up new possibilities in the area of dynamic aspect-oriented programming (AOP) [48] for Java. Previously, AOP for Java was either severely limited in the type of supported aspects or could only be applied at load time or compile time. The most commonly used Java AOP framework is AspectJ [49]. It comes with an aspect language based on Java.

HotWave is an aspect-oriented programming tool developed by Villazón et al. [9] that uses AspectJ and the current class redefinition capabilities of the Java VM (see Section 2.8) to provide dynamic aspect-oriented programming features. Figure 6.3 gives an architectural overview of HotWave. The tool is implemented as a Java agent that is attached to the VM (see Section 2.8.3). On VM startup, it registers a class file transformer that intercepts any class loading or class redefinition. HotWave maintains a set of currently active aspects. A command line console allows the user to change this set of aspects while the application is running. When the user requests a change, the tool sends a request to retransform all currently loaded classes to the VM. This causes the VM to reload all classes with their current bytecodes.

The registered class file transformer is called before each redefined class is loaded, such that HotWave can apply the currently active aspects to the bytecodes. By using the class file transformer instead of calling `redefineClasses` directly, the tool can make sure that

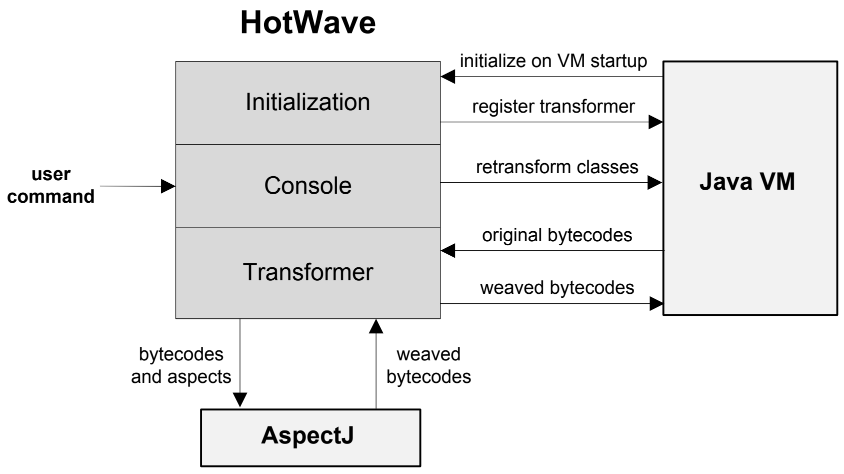


Figure 6.3: Interaction of HotWave with the Java VM and with AspectJ.

any newly loaded class is affected immediately, because the loading is also intercepted by the class file transformer. The class file transformer of HotWave uses AspectJ to weave the currently activated aspects into the bytecodes of the class and returns the result to the VM. The VM then performs the redefinition of the classes with the weaved bytecodes.

HotWave has several restrictions that are caused by the limitations of the class redefinition capabilities of the Java HotSpot VM. Therefore, we worked on the design of a new version of HotWave that is using the new features of the DCE VM [32, 33].

The DCE VM helps dynamic AOP tools such as HotWave in two ways: First, it removes the need for workarounds in case AspectJ requires the introduction of a new static field or a new method for an aspect-oriented programming construct. Second, it enables advanced AOP techniques that require explicit structural transformations on classes, such as the insertion of fields or methods (e.g., static cross-cutting). The DCE VM provides the advanced class redefinition features via the Java Platform Debugger Architecture (JPDA) [50] implementation of the HotSpot VM. Therefore, a dynamic AOP tool that is based on the DCE VM does not need to use additional or modified APIs.

Chapter 7

Evaluation

All algorithms described in this thesis are implemented in the Dynamic Code Evolution VM (DCE VM) that is based on a recent version of the Java HotSpot VM. Binaries and source code of the DCE VM can be downloaded from `http://ssw.jku.at/dcevm/`. The sections of this chapter evaluate the DCE VM from three perspectives:

Functional Evaluation: We list the class redefinition changes that are supported by the DCE VM and describe possible problems when dealing with complex changes.

Long-Term Execution Performance: One of the key features of the DCE VM is that its class redefinition capabilities do not decrease performance. In order to support that claim, we show the results of two benchmarks from the DaCapo benchmark suite [51].

Micro Benchmarks: We report on the performance of our instance update algorithm using micro benchmarks that add, remove, and reorder the fields of a class.

All timing measurements in this chapter were performed on an Intel Core i5-750 CPU running at 2.67GHz with 4 cores. The operating system was a 64-bit version of Windows 7. The results were obtained when running a 32-bit VM using the client compiler configuration. For comparisons with a baseline, we ran an unmodified build of the HotSpot VM using the version tagged as `hs21-02`, which is part of the JDK build `jdk7-b131`. At the time of writing this thesis, the source code of the DCE VM was based on the source code of this HotSpot VM version that was published in February 2011.

7.1 Functional Evaluation

Table 7.1 gives an overview of the supported class redefinition changes classified as discussed in Section 1.2. The DCE VM supports arbitrary changes to a set of classes, with two restrictions: When a deleted instance field is accessed by old code, an exception is thrown. Additionally, a type narrowing change is not carried out if the dynamic type of a variable would no longer match its static type after redefinition. In all cases, the continued execution of the Java program is fully compliant with standard Java semantics.

Method	Restrictions
Swap Method Body	
Add Method	
Remove Method	
Add Field	
Remove Field	`NoSuchFieldError` when accessing a deleted instance field
Add Supertype	
Remove Supertype	only allowed when dynamic type safety verification succeeds

Table 7.1: Supported code evolution changes of the DCE VM.

The base version of the DCE VM (as presented in Chapter 3) was significantly improved with the techniques for accessing deleted class members and safe dynamic type narrowing (see Chapter 4). The programmer can specify the desired behavior for accessing deleted class members (see Section 4.1.1). Therefore, removing a method or a static field from a class can be made completely safe by configuring the VM to access the old version of the class member if it is not present in the current version. Additionally, the VM now checks at class redefinition time whether removing a supertype can cause a problem in continued execution. This is a significant enhancement compared to the earlier VM version that was not able to guarantee continued execution without a VM crash in case of type narrowing changes.

The two remaining limitations of the DCE VM are that deleting an instance field can cause an exception in continued execution (e.g., because an old method still accesses the field) and type narrowing changes are not always allowed (i.e., because a variable would contain an invalid value). It would be possible to use the algorithm for accessing deleted class members (described in Section 4.1.2) for deleted instance fields too. However, the VM would have to keep the values of deleted fields for all active objects, thus increasing memory usage. The type narrowing changes could be handled by enclosing values violating the type system in proxy objects that contain a single field pointing to the value and act like the expected static type. It is however unclear what the semantics for method calls and field accesses on those proxy objects would be. Therefore, we believe that this would make predicting the program behavior after such a dynamic change overly complex. Both cases are rare when updating a running program to a new version, therefore we believe that those two restrictions are acceptable and do not impact the usability of the DCE VM.

Since version 1.4 of Java, the JPDA (Java Platform Debugger Architecture) defines commands for class redefinition (see Section 2.8). A VM specifies three flags to inform the debugger about the code evolution capabilities: The flag `canRedefineClasses` if class redefinition is possible at all, the flag `canAddMethod` if methods can be added to classes, and the flag `canUnrestrictedlyRedefineClasses` if arbitrary changes to classes are possible. To the best of our knowledge, DCE VM is the first Java VM that returns `true` for all three flags. We propose a more fine-grained distinction between different levels of code evolution based on our classification in Section 1.2 that takes the implementation complexity in the VM into account. The step between adding methods and allowing arbitrary changes is too large.

The DCE VM does not specify additional APIs for class redefinition, but works with the standard API as defined by the JPDA. This means that every standard Java debugger that uses the JPDA can immediately use the advanced class redefinition features of the DCE VM. The VM is already tested to work with the three major Java IDEs NetBeans, Eclipse, and IntelliJ IDEA. A debugger can treat a redefined program in the same manner as the original program and does not have to hide artificially introduced methods or proxy objects from the programmer. The DCE VM transfers field watch points of the debugger from the old to the new program if a matching field can be found in the new program. Also, method breakpoints are preserved if the body of a method does not change during a class redefinition.

It is difficult to measure the usage characteristics of code evolution, because it heavily depends on the application domain and also on the developer behavior. Gustavsson [4] 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%. For the last 14%, the DCE VM would at least be capable of executing manually written transformer methods. We believe that supporting a high percentage of changes is especially important so that developers adopt the technology.

At the time of writing this thesis, the implementation of our class redefinition algorithms in the DCE VM has already reached a high stability level. The VM passes 100% of Oracle's internal class redefinition test suite, which is used for HotSpot development. This means that the DCE VM also correctly handles the effects of class redefinition on JVMTI objects (e.g., when the debugger holds a JVMTI reference to a method or class that is replaced with a new version). We also developed a new test suite for the changes that are not tested by Oracle's class redefinition tests because the HotSpot VM does not support them. Our test suite is available from `http://ssw.jku.at/dcevm/tests`.

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, which would have been necessary anyway without code evolution. Therefore, we strongly recommend using the DCE VM for increasing development speed, but still advise against using the VM for updating critical server applications.

7.2 Long-Term Execution Performance

In order to show that our modifications to the VM have no negative performance impact on normal program execution, we compare the DCE VM with the unmodified Java HotSpot VM that the DCE VM is based on. Both VMs execute the benchmarks using the same JDK version (`jdk7-b131`). We demonstrate that the DCE VM performs equally well during normal program execution. Additionally, we show that while the DCE VM is slightly slowed down after a class redefinition, it reaches peak performance again as the program continues to execute.

We selected two benchmarks from the latest version (9.12) of the DaCapo benchmark suite [51]: The `fop` benchmark that reads XML files to generate PDF files and the `jython` benchmark that runs a Python interpreter executing a Python program. We measure the times of 20 subsequent runs of each benchmark within the same VM. A garbage collection between two subsequent runs is performed in order to reduce the random noise introduced by the garbage collector.

The heap size is specified with 1 GByte, the permanent generation size is set to 200 MByte, and the client compiler is used as the just-in-time compiler. Additionally, we use the following command line flag:

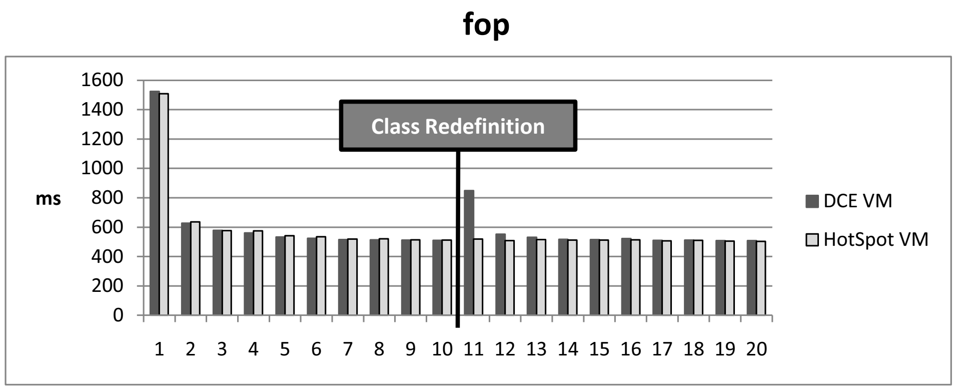
```
-Xrunjdwp:transport=dt_socket,
server=y,address=4000,suspend=n
```

This starts a JDWP agent for receiving JDWP commands. The agent is used for debugging the Java program running in the VM. When starting the DaCapo benchmarks, we register a callback class that can execute Java code between two subsequent benchmark runs. This callback class connects to the JDWP agent and sends the command for redefining classes.

The callback class triggers class redefintion between the 10th and 11th run of a benchmark. It is only performed in the DCE VM and not when the benchmark is executed in the unmodified HotSpot VM that serves as the baseline. We redefine a class from the DaCapo benchmark runner (`org.dacapo.harness.TestHarness`) with a new version that has a new public method. This kind of change is not supported in the HotSpot VM. The class redefinition causes the DCE VM to discard previously compiled machine code and constant pool cache entries as described in Section 3.6.

We executed the test setup 10 times and calculated the mean. Figure 7.1 shows the results for the unmodified reference HotSpot VM and our DCE VM when executing the two DaCapo benchmarks. The first ten runs of each benchmark show that the VMs run the benchmark at the same speed.

The performance characteristics after a code evolution step between the 10th and the 11th run are similar to the warm-up phase. The first run after the code evolution is significantly slower, because the compiled code is deoptimized and the constant pool cache entries are cleared. For both benchmarks, the number of cleared methods is more than 2000. However, the VM quickly resolves the constant pool cache entries and recompiles the frequently executed methods again. In the second run after the code evolution, the



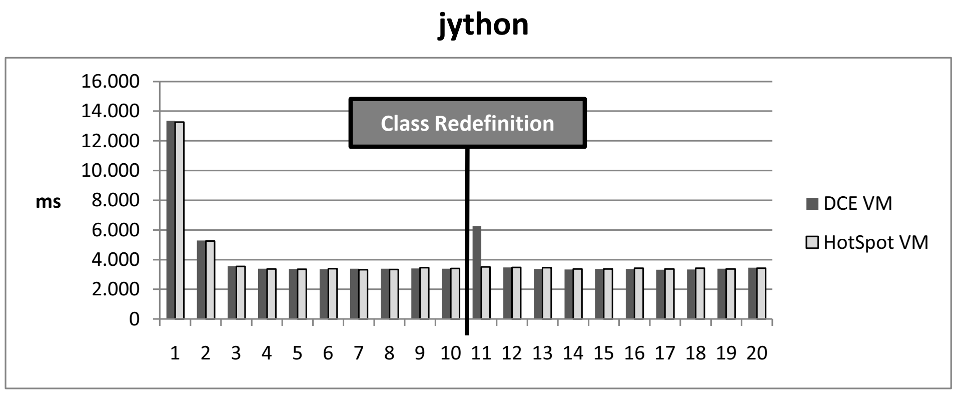


Figure 7.1: Executing 20 runs of the `fop` and the `jython` DaCapo benchmark with a class redefinition after 10 runs.

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.

Figure 7.2 reports the relative slow-down between the best run of a benchmark (first bar) and the first run after redefinition (second bar). Additionally, the relative slow-down of the first run of a benchmark is shown (third bar). In both benchmarks, the first run is also the slowest. The `fop` benchmark has a smaller difference between the best run and the first run (3.0x) than the `jython` benchmark (4.0x). Therefore, the first run after class redefinition is also less affected for the `fop` benchmark (1.7x) than for the `jython` benchmark (1.9x).

The class redefinition does not involve a garbage collection, because we do not increase the size of object instances. The average time for performing the class redefinition is 18ms for the `fop` benchmark and 35ms for `jython`. The `jython` benchmark loads more classes

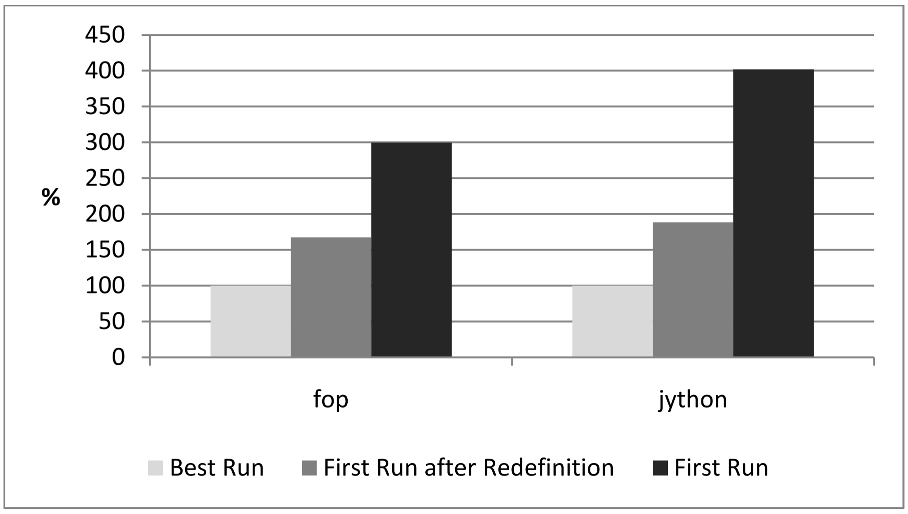


Figure 7.2: Slow-down of the first run after class redefinition and the first run of the benchmark versus the best run.

than `fop` and the additional metadata heap objects cause the heap iteration for updating pointers to take more time.

7.3 Micro Benchmarks

To measure the performance of instance updates and our garbage collector adjustments, we use three micro benchmarks in which we increase and decrease the field counts, reorder the fields of a class, and update all instances of this class to the new version. We compare the performance of this redefinition to the performance of a normal garbage collection run. Table 7.2 shows the different class versions used for the benchmarks. The leftmost column contains the initial version of the class. It has three `int` fields and three `Object` fields resulting in a total object size of 32 bytes (including 8 bytes object header). All three modifications are applied to the initial version, i.e., they are not consecutive. The three benchmark configurations are:

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 up to a multiple of 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.

Initial	Increase	Decrease	Reorder

`class C {`
`int i1;`
`int i2;`
`int i3;`
`Object o1;`
`Object o2;`
`Object o3;`
`}`
	`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;`
			`}`

32 bytes	40 bytes	24 bytes	32 bytes

Table 7.2: The initial and the three redefined versions of the class `C` that are used for the micro benchmarks.

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. For comparison, we also provide the time for 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 7.3 shows the results.

When no objects are affected, the cost of a class redefinition is about a third of the cost of a full garbage collection run. Most of the cost comes from the heap iteration to update pointers and the check for instances of redefined classes. The time also includes issuing the JDWP command and loading the new classes.

Increasing the size of all objects on the heap needs close to three times more time than the no-load garbage collection run. Our improved forward pointer calculation makes sure that not all of the objects, but only 20% of them need to be copied to a rescue buffer. One rescued object (32 bytes) makes room for four objects to increase their size by 8

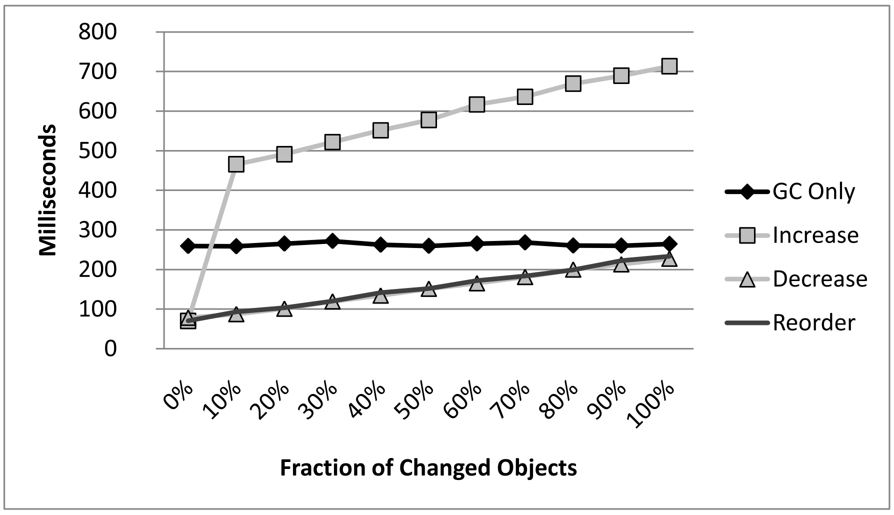


Figure 7.3: Timing results for changing the fields of a class compared against a garbage collection run.

bytes each while still being copied to lower memory addresses. The performance of this benchmark is additionally 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 comparable to a full garbage collection run if all objects are affected. As the size of the objects does not change for this micro benchmark, the objects are updated in-place during the heap iteration and so the modified garbage collection run is not necessary. We need to make a copy of each object and then copy the field values one-by-one.

Redefining a class such that all object sizes decrease is also slightly faster than a full garbage collection run. Decreasing the sizes can be done during the heap iteration using filler objects to fill the deprecated heap areas (see Section 3.4). This is significantly faster than adjusting the object sizes in a full garbage collection run like in the case where object sizes are increased. The filler objects are automatically removed by the garbage collector in subsequent garbage collection runs, because there are no pointers to them.

Chapter 8

Related Work

This chapter compares our work with other research in the same domain. We describe systems that implement dynamic code evolution for procedural languages, for dynamically-typed languages, and for C++. Additionally, we discuss other implementations of dynamic code evolution for Java. At the end of the chapter, we present a table that gives an overview of the features supported by the DCE VM in comparison to other systems.

8.1 General Discussions

Several classifications of runtime changes of programs have been published [3, 4]. Ebraert et al. presented a general discussion on the problems and pitfalls of software evolution [52] and examined dynamic changes from the user's point of view [53]. Additionally, they discussed how changes to the component structure of a software system can help future dynamic software evolution [54]. Our classification of changes looks at dynamic changes from the point of view of the implementation complexity in the VM (see Section 1.2). The programming model for safe dynamic version changes approaches the problem at the Java bytecode level (see Chapter 5) and not at the level of whole software components. This allows us to update components and even methods although they are currently executing.

Kramer and Magee investigated the problem of how to design applications to allow a consistent state transfer between new and old programs [55, 56]. Vandewoude et al. extended their work and introduced the notion of tranquillity [57]. Gupta developed a formal framework for program evolution [58]. Different formalizations of dynamic software updates were published by Bierman et al. [59] and Stoyle et al. [60]. Run time evolution in the context of aspect-oriented programming was discussed by Gustavsson et al. [61].

8.2 Procedural Languages

Early work on dynamic code evolution was done by Fabry [1]. The ability to change method implementations at run time is achieved by jump indirections 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 [62]. 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 [63, 64]. They require a binding table for methods that is updated accordingly. An update is prohibited if one of the updated methods is currently executing. Additionally, PODUS prohibits an update if there is a method currently executing that can call one of the updated methods. In case of Java, such a rule would be impractical due to virtual method dispatch and the possibility to invoke methods via reflection.

Gupta implemented a hot code replacement mechanism for C programs on a Sun workstation based on state transfer [65, 66]. The system supports adding and deleting of functions. For adding global data, extra pointers must be declared in advance. So-called 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.

Hicks et al. presented a dynamic modification system for C-like languages called Popcorn [67]. 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.

Later, they built a new system on top of it, called Ginseng that brings significant improvements [68]. They need to compile the programs in a different way and report a performance overhead of up to 32% if several updates are applied. Safe update points have to be specified in the original program as calls into the runtime system. Therefore, the user has to anticipate all possible update points in the original program version. For each update point, Ginseng performs an analysis on the update constraints (e.g., which types may be changed at that point). Procedures can only be updated as a whole unit. Later, the authors of Ginseng presented Proteus, a calculus for automatically inferring the update constraints [60].

In contrast to the work described in this section, our algorithm targets the challenges of code evolution for object-oriented languages. We support an atomic change of a set of class definitions and can guarantee type safety even for complex changes to the class hierarchy. Also, we leverage the advantages of dynamic compilation in a VM and therefore do not need to insert hooks into a statically compiled program. This enables us to add the dynamic class redefinition capabilities to the VM without imposing a performance penalty for normal program execution. The modification of the garbage collector gives our algorithm the possibility to increase the size of an object without the need of pointer indirections.

8.3 C++, CLOS, and Smalltalk

Hjalmtysson et al. [69] presented an approach for dynamically changing the implementation of 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 or change the class hierarchy.

The Common Lisp Object System (CLOS) [70, 71] and Smalltalk [72] both allow class redefinition in a dynamically-typed system. Both systems allow the definition of *metaclasses* that describe the behavior of other classes. This contrasts the design of the HotSpot VM where the VM class objects implement a fixed behavior that cannot be redefined by the user. While user defined metaclasses allow maximal flexibilty for changing the behavior of method calls, they constrain many compiler optimizations.

A conceptual difference to our class redefinition approach is that in CLOS and Smalltalk the classes can only be redefined one-by-one, while we can atomically redefine a set of classes. In the former case, the programmer is responsible for performing the redefinitions in the correct order. Additionally, the programmer needs to manually avoid concurrency problems in multi-threaded applications when one thread redefines classes while another thread uses it. The DCE VM automatically calculates the correct order for class redefinition and performs the change atomically from the point of view of the program. Furthermore, the DCE VM finds out about classes that are affected by the current redefinition command.

In case of deleted methods, CLOS and Smalltalk offer the possibility to implement a method that is called when a method is not found. In CLOS, this method is called `NO-APPLICABLE-METHOD`, in Smalltalk, it is called `doesNotUnderstand`. The default implementation of this method throws an exception. The programmer can either catch this exception or provide a different implementation. Our algorithm to access old deleted class members resolves the problem of deleted methods without manual intervention from the programmer and introduces a modified dynamic dispatch mechanism that takes the version of the executing method into account (see Section 4.1.2). Also, we provide the possibility of a static check at the time of the redefinition whether the call of a deleted method can occur in continued program execution.

Our algorithm for checking the type safety of a type narrowing change is not necessary for dynamically-typed languages, because variables in such languages do not have static types that might become inconsistent with the dynamic types of the referenced objects. Also, the verification of the new methods that they comply with the new type hierarchy is not required for CLOS or Smalltalk, because silently converting between two types is always valid for dynamically-typed languages. The definition of transformer methods that update the instances from the old to the new version is possible in CLOS and Smalltalk as well as in the DCE VM. We believe that parts of our algorithm (e.g., the garbage collection modifications) can be used for efficiently implementing the CLOS class redefinition mechanism. Both CLOS and Smalltalk do not offer the possibility to change the execution point of a currently executing method during a redefinition.

8.4 Java Bytecode Rewriting and Proxification

Iguana/J [73] is a system that allows the definition of metaclasses in Java code similar to the metaclasses in Smalltalk or CLOS. The system offers a high flexibility but its authors report a slowdown factor of about 25 for object creation, method calls, and method returns.

There are various implementations of class redefinition for Java based on proxy objects and bytecode rewriting [74, 75, 76]. 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 that supports the JVMTI class redefinition command. However, this advantage is small given that the DCE VM is available for many different platforms and operating systems. Disadvantages are the significant 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 using 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 they contain also generated proxy methods).

Techniques that are based on object wrappers require that every changeable class implements an interface that specifies its public API [74, 75]. While the implementation of a class can be redefined, this interface must remain unmodified. Additionally, the declared type of object variables and fields that can hold instances of changeable classes must be defined as such an interface type. The additional indirection incurs a performance penalty on normal program execution and hinders the use of the Java Reflection API.

Gregersen and Jørgensen presented a system that uses bytecode rewriting techniques [76]. Their system performs the updates at the granularity of NetBeans components. They use proxification techniques to be able to change the implementation behind the well-defined API of a NetBeans module. They report an increased startup time due to the bytecode rewriting and normal program execution is slowed down by 89.4%. Later, Gregersen et al. advocated the idea of having a dynamic-update-enabled virtual machine and outline its advantages [77].

We showed that our implementation does not slow down normal program execution at all (see Section 7.2). Also, the use of the Reflection API or the JVMTI debugging interface is not negatively affected. The direct modification of the internal data structures of the VM keeps the class redefinition transparent in contrast to the techniques presented in this section.

8.5 Java Virtual Machines

The project JDrums [78] 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. [79] presented an implementation of code evolution based on JDK 1.2 called DVM. 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. Their main performance loss comes from acquiring a global lock at every bytecode that involves an object reference or a method invocation. Additionally, they allow only binary compatible changes, their VM uses object handles instead of direct object references, and they have to disable the just-in-time compiler.

Subramanian et al. [80] implemented code evolution for the Jikes RVM. They support adding and removing methods and fields, but do not support changes to the class hierarchy. 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 thus cannot be used by standard Java development environments. Additionally, they cannot perform an update while an updated method is currently executing.

Dimitriev et al. [81] presented a class evolution algorithm for the persistent object system for the Java programming language called PJama [82]. They introduced 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.

To the best of our knowledge, the DCE VM is the first Java VM that allows arbitrary changes to its classes including changes to the class hierarchy. Further distinguishing characteristics of our VM are that there is no performance penalty on normal program execution and that the update can be performed at any point the Java program can be suspended even if redefined methods are currently active.

8.6 Stack Updates

All dynamic update systems described so far are not capable of updating the implementation of currently executing methods. Therefore, they are not suitable for updating long-running loops or the main method of a program.

The authors of Ginseng proposed a technique called *loop extraction* to reduce this drawback [68]. The body of the loop is extracted into a newly generated method such that it can be updated. The user has to manually select loops that should be extracted and the extraction implies a performance overhead.

Upstare is a dynamic update system for C developed by Makris and Bazzi that allows replacing active executions [83]. They perform a full stack unrolling and allow the user to specify continuation mappings between old and new method versions. Update points are automatically inserted at the beginning of loops and methods. Due to the additional indirections and update point checks, they report an overhead of up to 38.5% for their benchmarks. The update model of Upstare offers high flexibility, because the new method can effectively run completely different code starting from the current execution point in the old method. It is however not possible to switch back to the old version, and the user is responsible for checking the semantic correctness of a continuation mapping. Every specified update point has to provide a valid mapping to the new version, it is therefore not possible to delay an update.

Our programming model for safe version change is specified at the Java bytecodes level. We do not allow manual mappings between the stack of the old method and the stack of the new method. However, we define constraints such that an automatic mapping is possible and also verify that the change between the two method versions is correct. We can switch between two versions not only at update points, but at safe update regions that cover a range of bytecode positions. Additionally, the update regions do not have to be specified before starting the program but can be changed while the program is running.

8.7 Hotswapping

The work most closely related to ours was done in an attempt to apply PJama principles to run-time evolution of Java applications by Dmitriev [5, 10]. His implementation is part of the Java HotSpot VM since JDK 1.4 and is widely known as *hotswapping* due

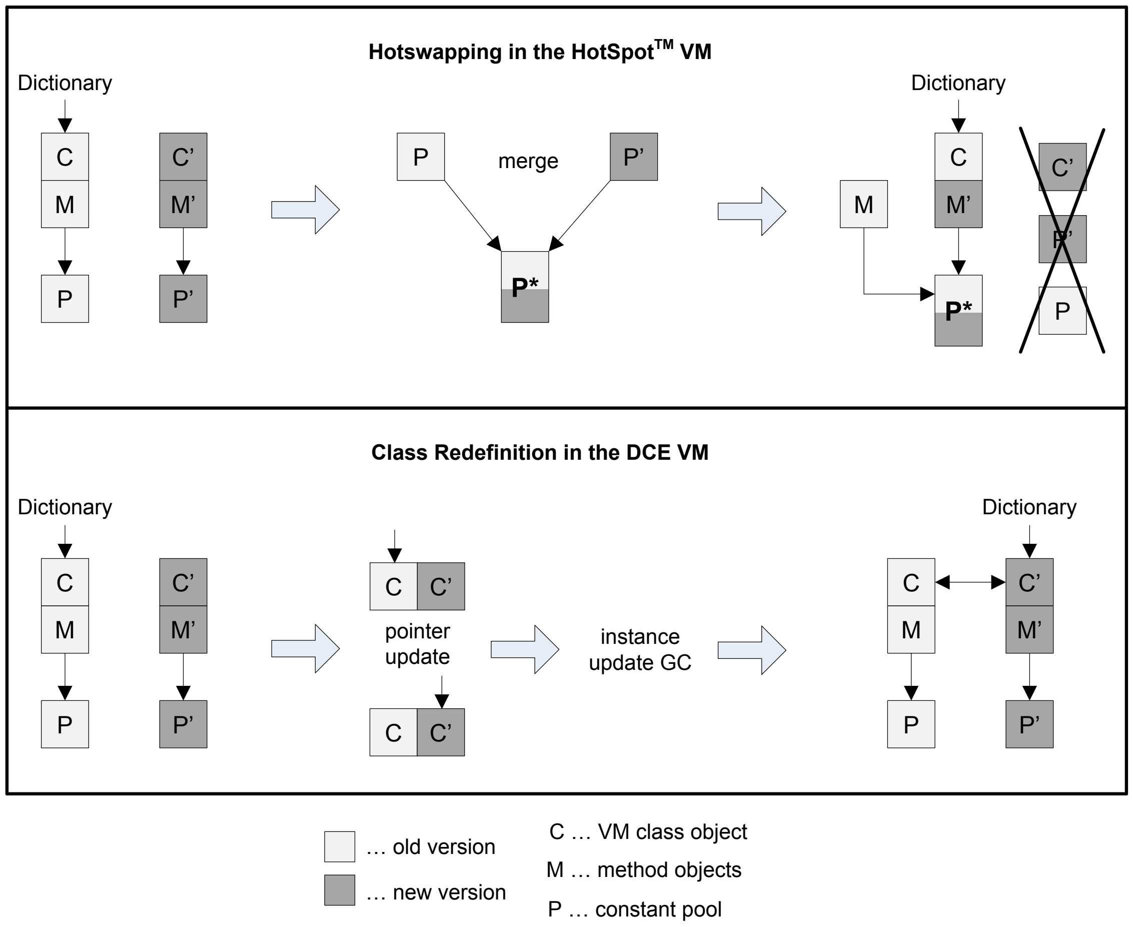


Figure 8.1: Different strategies for loading the new versions of redefined classes in the HotSpot VM and the DCE VM.

to its capability of swapping method bodies at run time. The DCE VM is based on this implementation and enhances it to allow arbitrary changes. To achieve this, we changed the way how new classes are loaded and how new classes replace old classes.

Figure 8.1 shows the conceptual difference between Dmitriev's algorithm and our algorithm with respect to replacing the VM metadata. Dmitriev's implementation in the HotSpot VM first loads the new VM class object `C'`, its method objects `M'`, and its constant pool `P'`. Then, it merges the old constant pool `P` and the new constant pool `P'` into a newly allocated constant pool `P*`. The goal of this merge is that the old methods `M` and the new methods `M'` can share a single constant pool. At first, the entries from `P` are copied into `P*`. Then, for every entry in `P'` either an existing entry in `P*` is found or a new entry is appended to `P*`. The constant pool entries of `P'` have a different index in

P*, therefore constant pool references in M′ must be rewritten such that M′ can use P* as the constant pool. The maximum size of the merged constant pool P* is the sum of the sizes of P and P′. However, in many cases the size is significantly less, because of the two class versions sharing common constant pool entries. After the constant pool merge, the old VM class object C is connected with the new methods M′. The new VM class object C′ and the two constant pools P and P′ are disconnected from the object graph and will be removed in a subsequent garbage collection run.

Dmitriev also describes how binary compatible changes can be performed using a custom class loader. A new VM class object is first linked to the old class hierarchy and later reconnected with the newest version of its superclasses. In contrast, the DCE VM loads the new classes such that they are immediately connected with the new version of their superclasses by ordering the classes before redefinition (see Section 3.3). Also, we duplicate affected classes in the class hierarchy such that the set of new classes forms a side universe. This enables us to reuse the standard class verification mechanisms and it is also a precondition for performing complex class hierarchy changes (e.g., when A was a superclass of B in the old version, but A′ is a subclass of B′ in the new version).

The DCE VM does not merge the old and the new constant pool but instead keeps both constant pools in the system. This is less memory efficient but makes the class redefinition simpler and faster. We believe that the approach of merging the two constant pools was chosen for the HotSpot VM to prevent problems with the concurrently running just-in-time compiler that could save an index of an old constant pool entry. The DCE VM skips all active compilations anyway and prohibits concurrent compilation.

Our VM performs a heap iteration that swaps pointers to the VM class objects in a heap iteration and then performs the instance updates that are necessary for changes to fields. The garbage collection for the instance updates can significantly slow down the class redefinition in the DCE VM, however such a garbage collection is skipped if there are no instances with an increased size. The final state of the class hierarchy with the different class versions in a doubly-linked list uses more memory than the class redefinition in the Java HotSpot VM, but enables us to access old class members (see Section 4.1.2). Our algorithms on safe type narrowing (see Section 4.2.1) and safe version change (see Chapter 5) further extend the class redefinition capabilities of the DCE VM compared to the Java HotSpot VM.

8.8 Comparison Table

Table 8.1 compares the features of various dynamic code evolution systems with the features of the DCE VM. Most of the systems allow changes to the methods and fields of a program. In case of procedural systems like DYMOS, there are no fields of classes but global data sections that can be modified during a dynamic change.

Class hierarchy changes are only supported in the systems that support metaclass definitions (i.e., CLOS and Smalltalk). However, those systems cannot atomically redefine a set of classes like the DCE VM. The ability to update active methods is available in Ginseng. While their system is more flexible than ours, they cannot reason about the semantic correctness of an update and do not support safe update regions.

Dynamic code evolution comes without a performance penalty only when the program is executed in a VM (e.g., CLOS, Smalltalk, JVolve, Hotswap, and DCE VM). Other techniques often report significant performance losses (e.g., 38.5% for Ginseng). The possibility to call deleted virtual methods based on the version of the currently executing method and the type of the receiver is only available in the DCE VM.

	DYMOS	Ginseng	Upstare	CLOS	Smalltalk	JDrums	DVM	JVolve	Hotswap	DCE VM
Method Bodies	X	X	X	X	X	X	X	X	X	X
Add/Remove Methods	X	X	X	X	X	X	X	X		X
Add/Remove Fields	X	X	X	X	X	X	X	X		X
Class Hierarchy Changes				X	X					X
Atomic Class Redefinition								X	X	X
Active Methods Updates			X							X
No Performance Overhead				X	X			X	X	X
Deleted Virtual Methods										X
Literature	[62]	[68]	[83]	[70]	[72]	[78]	[79]	[80]	[10]	

Table 8.1: Feature comparison of systems that allow dynamic changes to running programs.

Chapter 9

Summary

This chapter summarizes the contributions, outlines possible areas for future work, and finally concludes the thesis.

9.1 Contributions

The main contribution of this thesis is the Dynamic Code Evolution VM (DCE VM), a modification of the Java HotSpot VM. To the best of our knowledge, the DCE VM is the first VM for a statically typed object-oriented language that offers unlimited support for class redefinition (see Section 7.1) without compromising execution performance (see Section 7.2). Here is a detailed list of contributions of this thesis:

- We describe a new algorithm for class redefinition in a Java VM (see Chapter 3).
- We allow adding and removing fields and methods at run time and also support changes to the type hierarchy (see Chapter 3).
- We discuss possible problems caused by binary incompatible changes (see Chapter 4).
- We describe a solution for the problem of deleted class members (see Section 4.1.1).
- We propose an algorithm for checking type safety in case of removed supertypes (see Section 4.2.1).
- We present a restricted programming model for safe dynamic updates to Java programs (see Chapter 5).

- We describe three different case studies to outline possible usages of our VM (see Chapter 6).
- We show that our approach does not imply any performance penalty before a class redefinition and reaches peak performance again after a change (see Section 7.2).
- We evaluate the performance of our instance update algorithm on selected micro benchmarks (see Section 7.3).
- We implemented our algorithms as a modification of the production-quality Java HotSpot VM and made the source code and binaries of the modified VM available for download.

9.2 Future Work

The new functionalities of the DCE VM form a basis for further research in the area of dynamic code evolution. We propose possible future work in the following areas:

IDE support: When the developer changes an application, the IDE often has additional information about the change that is not transmitted in the redefinition command. An example would be a refactoring that renames a field: For the VM, the two versions of the changed field are unrelated, because they have different names. The IDE however knows that they are related and could automatically create the correct transformer method that copies the value from the old to the new field.

A different example for IDE support is the use case of the modified GUI builder (see Section 6.1). This work could be further improved by incorporating the difference of the NetBeans form description between the two versions (e.g., only reinitializing components with modified properties).

Additionally, the separation between the running dialog and the GUI builder view of the dialog could be removed. This could be done by changing the base classes of Swing components (e.g., `JFrame`, `JLabel` or `JButton`) to add GUI builder functionality (e.g., the renaming of a label).

Automatically generated code regions: The programming model introduced in Chapter 5 requires the specification of safe update regions. Automatic specification of safe update regions could be added to an aspect-oriented programming framework in order to safely weave an aspect into a running application.

For the case of updating a normal Java program, those code regions would have to be generated by a general-purpose bytecode comparison. Also, a semi-automatic user assisted process to specify the safe update regions could help updating long-running server applications.

Redefinition of resources: The work in this thesis is about the redefinition of Java classes only. However, updating Java programs can also mean updating of resources (e.g., XML configuration files or localization files). This requires a different approach where the main problem is the reinitialization of the Java data structures that are built from the resource files.

Incremental updates: A possible functional enhancement to the VM would be the support of fast and small incremental updates. This requires a different API where the redefinition command specifies the change (e.g., adding a field to the class) instead of a whole new class definition. Additionally, the need for the GC run would have to be completely removed even for increased instance sizes. This could either be achieved by splitting objects into multiple parts on the heap or by adding read barriers to fields and updating the instances lazily (i.e., only at the first field access). Both techniques would require significant changes to both the interpreter and the just-in-time compiler.

9.3 Conclusions

We discussed dynamic code evolution in the context of statically typed, object-oriented languages and implemented the DCE VM that allows unlimited class redefinitions at run time. The redefinition can happen at any point during program execution. Old and new code may co-exist in the VM, and therefore our approach allows redefining methods that are currently active. Our VM works with the debuggers of standard Java development environments and can be used to avoid restarts during program development.

We showed that the flexibility of dynamic updates does not impose any performance penalty on the system, even when the system is a production-quality, high performance VM. Although the dynamic update itself can temporarily slow down the system, it quickly reaches peak performance again. We believe that this is an important characteristic of our approach, because it brings the new possibilities without drawbacks for normal program execution. The flexibility of dynamic updates is currently considered one of the advantages of languages with dynamic typing, compared to languages with static typing. Our VM brings this flexibility to the statically typed Java language.

We implemented improvements to the DCE VM to support binary incompatible changes. First, we described a solution to the problem of calling a deleted method or accessing a deleted static field. We keep the old and the new class version in the system to be able to select appropriate class versions in continued execution. Second, we showed how to guarantee type safety for changes that remove a supertype using a heap and stack iteration.

We presented a restricted programming model that allows us to reason about the semantic correctness of an update. Our approach to dynamically changing between two program versions does not require manual specification of safe update points. We believe that safe dynamic updates can only be adopted for common use if an automatic algorithm checks the semantic correctness of an update and manual specifications are unnecessary. Our programming model can form a basis for IDE support of safe dynamic updates as suggested in the future work section.

The DCE VM has already attracted significant interest from the developer community. There are plans to integrate our VM modifications back into the main OpenJDK source code repository in order to support unlimited class redefinition for the standard HotSpot VM. The DCE VM is currently based on the repository that is tagged with `jdk7-b131` and we want to carry on updating the DCE VM in regular intervals such that it keeps being based on a recent version of the HotSpot VM repository. Binaries and source code of the DCE VM can be downloaded from `http://ssw.jku.at/dcevm/`.

List of Figures

Listings

Bibliography

[1] Robert 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.

[2] Oracle Corporation. *Top 25 RFEs (Requests for Enhancements)*, 2011. `http://bugs.sun.com/top25_rfes.do`.

[3] Mario Pukall and Martin Kuhlemann. **Characteristics of Runtime Program Evolution**. In *Proceedings of the Workshop on Reflection, AOP and Meta-Data for Software Evolution*, 2007.

[4] Jens Gustavsson. **A Classification of Unanticipated Runtime Software Changes in Java**. In *Proceedings of the International Conference on Software Maintenance*. IEEE Computer Society, 2003.

[5] Mikhail 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.

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

[7] Shigeru Chiba, Yoshiki Sato, and Michiaki Tatsubori. **Using HotSwap for Implementing Dynamic AOP Systems**. In *Proceedings of the Workshop on Advancing the State-of-the-Art in Run-time Inspection*, 2003.

[8] Alex Villazón, Walter Binder, Danilo Ansaloni, and Philippe Moret. **Advanced Runtime Adaptation for Java**. In *Proceedings of the International Conference on Generative Programming and Component Engineering*, pages 85–94. ACM Press, October 2009.

[9] Alex Villazón, Walter Binder, Danilo Ansaloni, and Philippe 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.

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

[11] Hanspeter Mössenböck. **Adding Static Single Assignment Form and a Graph Coloring Register Allocator to the Java HotSpot$^{\mathrm{TM}}$ Client Compiler**. Technical Report 15, Institute for Practical Computer Science, Johannes Kepler University Linz, 2000.

[12] Hanspeter Mössenböck and Michael Pfeiffer. **Linear Scan Register Allocation in the Context of SSA Form and Register Constraints**. In *Proceedings of the International Conference on Compiler Construction*, pages 229–246. Springer-Verlag, 2002.

[13] Christian Wimmer and Hanspeter Mössenböck. **Optimized Interval Splitting in a Linear Scan Register Allocator**. In *Proceedings of the ACM/USENIX International Conference on Virtual Execution Environments*, pages 132–141. ACM Press, 2005.

[14] Thomas Kotzmann and Hanspeter Mössenböck. **Escape Analysis in the Context of Dynamic Compilation and Deoptimization**. In *Proceedings of the ACM/USENIX International Conference on Virtual Execution Environments*, pages 111–120. ACM Press, 2005.

[15] Thomas Kotzmann and Hanspeter Mössenböck. **Run-Time Support for Optimizations Based on Escape Analysis**. In *Proceedings of the International Symposium on Code Generation and Optimization*, pages 49–60, Washington, DC, USA, 2007. IEEE Computer Society.

[16] Christian Wimmer and Hanspeter Mössenböck. **Automatic Object Colocation Based on Read Barriers**. In *Proceedings of the Joint Modular Languages Conference*. Springer-Verlag, 2006.

[17] Christian Wimmer and Hanspeter Mössenböck. **Automatic Feedback-directed Object Inlining in the Java HotSpot$^{\text{TM}}$ Virtual Machine**. In *Proceedings of the ACM/USENIX International Conference on Virtual Execution Environments*, pages 12–21. ACM Press, 2007.

[18] Christian Wimmer and Hanspeter Mössenböck. **Automatic Array Inlining in Java Virtual Machines**. In *Proceedings of the International Symposium on Code Generation and Optimization*, pages 14–23. ACM Press, 2008.

[19] Christian Wimmer and Hanspeter Mössenbösck. **Automatic Feedback-directed Object Fusing**. *ACM Transactions on Architecture and Code Optimization*, **7**, October 2010.

[20] Thomas Würthinger, Christian Wimmer, and Hanspeter Mössenböck. **Array Bounds Check Elimination for the Java HotSpot$^{\text{TM}}$ Client Compiler**. In *Proceedings of the International Conference on the Principles and Practice of Programming in Java*, pages 125–133. ACM Press, 2007.

[21] Thomas Würthinger, Christian Wimmer, and Hanspeter Mössenböck. **Array Bounds Check Elimination in the Context of Deoptimization**. *Science of Computer Programming*, **74**:279–295, March 2009.

[22] Thomas Würthinger. *Visualization of Program Dependence Graphs*. Master's thesis, Institute for System Software, Johannes Kepler University Linz, 2007.

[23] Thomas Würthinger, Christian Wimmer, and Hanspeter Mössenböck. **Visualization of Program Dependence Graphs**. In *Proceedings of the International Conference on Compiler Construction*, pages 193–196. Springer-Verlag, 2008.

[24] Christian Häubl, Christian Wimmer, and Hanspeter Mössenböck. **Optimized Strings for the Java HotSpot$^{\text{TM}}$ Virtual Machine**. In *Proceedings of the International Conference on the Principles and Practice of Programming in Java*, PPPJ '08, pages 105–114. ACM Press, 2008.

[25] Lukas Stadler, Christian Wimmer, Thomas Würthinger, Hanspeter Mössenböck, and John Rose. **Lazy Continuations for Java Virtual Machines**. In *Proceedings of the International Conference on the Principles and Practice of Programming in Java*, pages 143–152. ACM Press, 2009.

[26] Lukas Stadler, Thomas Würthinger, and Christian Wimmer. **Efficient Coroutines for the Java Platform**. In *Proceedings of the International Conference on the Principles and Practice of Programming in Java*, pages 20–28. ACM Press, 2010.

[27] Christian Wimmer. *Java HotSpot^{TM} Client Compiler Visualizer*, 2008. `http://c1visualizer.dev.java.net/`.

[28] Oracle Corporation. *The Maxine Virtual Machine Project*, 2011. `http://labs.oracle.com/projects/maxine/`.

[29] Thomas Würthinger, Michael L. Van De Vanter, and Doug Simon. **Multi-Level Virtual Machine Debugging using the Java Platform Debugger Architecture**. In *Proceedings of the Conference on Perspectives of System Informatics*. Springer-Verlag, 2009.

[30] Ben L. Titzer, Thomas Würthinger, Doug Simon, and Marcelo Cintra. **Improving Compiler-Runtime Separation with XIR**. In *Proceedings of the ACM/USENIX International Conference on Virtual Execution Environments*, pages 39–50. ACM Press, 2010.

[31] Thomas Würthinger, Christian Wimmer, and Lukas Stadler. **Dynamic Code Evolution for Java**. In *Proceedings of the International Conference on the Principles and Practice of Programming in Java*, pages 10–19, New York, NY, USA, 2010. ACM Press.

[32] Thomas Würthinger, Walter Binder, Danilo Ansaloni, Philippe Moret, and Hanspeter Mössenböck. **Improving Aspect-oriented Programming with Dynamic Code Evolution in an Enhanced Java Virtual Machine**. In *Proceedings of the Workshop on Reflection, AOP and Meta-Data for Software Evolution*, pages 5:1–5:5. ACM Press, 2010.

[33] Thomas Würthinger, Walter Binder, Danilo Ansaloni, Philippe Moret, and Hanspeter Mössenböck. **Applications of Enhanced Dynamic Code Evolution for Java in GUI Development and Dynamic Aspect-oriented Programming**. In *Proceedings of the International Conference on Generative Programming and Component Engineering*, pages 123–126. ACM Press, 2010.

[34] Thomas Würthinger, Christian Wimmer, and Lukas Stadler. **Unrestricted and Safe Dynamic Code Evolution for Java**. Submitted to *Science of Computer Programming*, 2011.

[35] Urs Hölzle, Craig Chambers, and David 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.

[36] Stephen J. Fink and Feng Qian. **Design, Implementation and Evaluation of Adaptive Recompilation with On-stack Replacement**. In *Proceedings of the International Symposium on Code Generation and Optimization*, pages 241–252. IEEE Computer Society, 2003.

[37] David Ungar. **Generation Scavenging: A Non-disruptive High Performance Storage Reclamation Algorithm**. *SIGSOFT Softw. Eng. Notes*, **9**:157–167, April 1984.

[38] Mikhail Dmitriev. **Application of the HotSwap Technology to Advanced Profiling**. In *Proceedings of the International Workshop on Unanticipated Software Evolution*. Springer-Verlag, 2002.

[39] Mikhail Dmitriev. **Profiling Java Applications Using Code Hotswapping and Dynamic Call Graph Revelation**. *SIGSOFT Software Engineering Notes*, **29**:139–150, January 2004.

[40] Oracle Corporation. *Java VM Tool Interface*, 2011. `http://download.oracle.com/javase/6/docs/platform/jvmti/jvmti.html`.

[41] Oracle Corporation. *Java Debug Interface*, 2011. `http://download.oracle.com/javase/6/docs/jdk/api/jpda/jdi/`.

[42] Oracle Corporation. *Java java.lang.instrument Package*, 2011. `http://download.oracle.com/javase/6/docs/technotes/guides/instrumentation/`.

[43] Oracle Corporation. *Java Debug Wire Protocol*, 2011. `http://download.oracle.com/javase/6/docs/technotes/guides/jpda/jdwp-spec.html`.

[44] Sophia Drossopoulou, David Wragg, and Susan Eisenbach. **What is Java Binary Compatibility?** In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 341–361. ACM Press, 1998.

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

[46] Gilad Bracha and William Cook. **Mixin-based Inheritance**. In *Proceedings of the European Conference on Object-Oriented Programming*, pages 303–311. ACM Press, 1990.

[47] Kerstin Breiteneder, Christoph Wimberger, and Thomas Würthinger. **Implementing Dynamic Mixins for the Java Virtual Machine**. In *Proceedings of the Work-in-Progress Session at the 8th International Conference on the Principles and Practice of Programming in Java*, 2010.

[48] Gregor Kiczales, John Lamping, Anurag Menhdhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. **Aspect-Oriented Programming**. In *Proceedings of the European Conference on Object-Oriented Programming*, pages 220–242. Springer-Verlag, 1997.

[49] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. **An Overview of AspectJ**. In Jorgen Lindskov Knudsen, editor, *Proceedings of the European Conference on Object-Oriented Programming*, pages 327–353. Springer-Verlag, 2001.

[50] Oracle Corporation. *Java Platform Debugger Architecture*, 2011. `http://java.sun.com/javase/technologies/core/toolsapis/jpda/`.

[51] 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.

[52] Peter Ebraert, Yves Vandewoude, Theo D'Hondt, and Yolande 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.

[53] Peter Ebraert, Theo D'Hondt, Yves Vandewoude, and Yolande Berbers. **User-centric Dynamic Evolution**. In *Proceedings of the International ERCIM Workshop on Software Evolution*, 2006.

[54] Peter Ebraert, Theo D'Hondt, and Tom Mens. **Enabling Dynamic Software Evolution through Automatic Refactoring**. In Ying Zhou and James R. Cordy, editors, *Proceedings of the International Workshop on Software Evolution Transformations*, pages 3–6, 2004.

[55] Jeff Kramer and Jeff Magee. **The Evolving Philosophers Problem: Dynamic Change Management**. *IEEE Transactions on Software Engineering*, **16**(11):1293–1306, 1990.

[56] Jeff Kramer and Jeff 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.

[57] Yves Vandewoude, Peter Ebraert, Yolande Berbers, and Theo D'Hondt. **An Alternative to Quiescence: Tranquility**. In *Proceedings of the International Conference on Software Maintenance*, pages 73–82. IEEE Computer Society, 2006.

[58] Deepak Gupta, Pankaj Jalote, and Gautam Barua. **A Formal Framework for On-line Software Version Change**. *IEEE Transactions on Software Engineering*, **22**(2):120–131, 1996.

[59] Gavin Bierman, Michael Hicks, Peter Sewell, and Gareth Stoyle. **Formalizing Dynamic Software Updating**. In *Proceedings of the International Workshop on Unanticipated Software Evolution*. Springer-Verlag, 2003.

[60] Gareth Stoyle, Michael Hicks, Gavin Bierman, Peter Sewell, and Iulian Neamtiu. **Mutatis Mutandis: Safe and Predictable Dynamic Software Updating**. *ACM Transactions on Programming Languages and Systems*, **29**(4), 2007.

[61] Jens Gustavsson, Tom Staijen, and Uwe Assmann. **Runtime Evolution as an Aspect**. In *Proceedings of the International Workshop on Unanticipated Software Evolution*. Springer-Verlag, 2004.

[62] Robert P. Cook and Insup Lee. **DYMOS: A Dynamic Modification System**. *SIGSOFT Software Engineering Notes*, **8**:201–202, March 1983.

[63] Ophir Frieder and Mark E. Segal. **On Dynamically Updating a Computer Program: From Concept to Prototype**. *Journal on System Software*, **14**:111–128, February 1991.

[64] Mark E. Segal and Ophir Frieder. **On-the-Fly Program Modification: Systems for Dynamic Updating**. *IEEE Software*, **10**:53–65, March 1993.

[65] Deepak Gupta and Pankaj Jalote. **On-line Software Version Change Using State Transfer Between Processes**. *Software - Practice and Experience*, **23**(9):949–964, 1993.

[66] Deepak Gupta and Pankaj 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.

[67] Michael Hicks, Jonathan T. Moore, and Scott Nettles. **Dynamic Software Updating**. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 13–23. ACM Press, 2001.

[68] Iulian Neamtiu, Michael Hicks, Gareth Stoyle, and Manuel Oriol. **Practical Dynamic Software Updating for C**. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM Press, 2006.

[69] Gísli Hjálmtýsson and Robert Gray. **Dynamic C++ Classes: A Lightweight Mechanism to Update Code in a Running Program**. In *Proceedings of the USENIX Annual Technical Conference*, ATEC '98, pages 6–6, Berkeley, CA, USA, 1998. USENIX Association.

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

[71] Richard P. Gabriel, Jon L. White, and Daniel G. Bobrow. **CLOS: Integrating Object-oriented and Functional Programming**. *Communications of the ACM*, **34**(9):29–38, 1991.

[72] Adele Goldberg and David Robson. *Smalltalk-80: the Language and its Implementation*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1983.

[73] Barry Redmond and Vinny Cahill. **Supporting Unanticipated Dynamic Adaptation of Application Behaviour**. In *Proceedings of the European Conference on Object-Oriented Programming*, pages 205–230. Springer-Verlag, 2002.

[74] Alessandro Orso, Anup Rao, and Mary Jean Harrold. **A Technique for Dynamic Updating of Java Software**. In *Proceedings of the International Conference on Software Maintenance*, pages 649–. IEEE Computer Society, 2002.

[75] Mario Pukall, Christian Kästner, and Gunter Saake. **Towards Unanticipated Runtime Adaptation of Java Applications**. In *Proceedings of the 2008 15th Asia-Pacific Software Engineering Conference*, pages 85–92. IEEE Computer Society, 2008.

[76] Allan Raundahl Gregersen and Bo Nørregaard Jørgensen. **Dynamic Update of Java Applications – Balancing Change Flexibility vs Programming Transparency**. *Journal of Software Maintenance and Evolution*, **21**:81–112, March 2009.

[77] Allan Raundahl Gregersen, Douglas Simon, and Bo Nørregaard Jørgensen. **Towards a Dynamic-update-enabled JVM**. In *Proceedings of the Workshop on Reflection, AOP and Meta-Data for Software Evolution*, pages 1–7. ACM Press, 2009.

[78] Jesper Andersson and Tobias Ritzau. **Dynamic code update in JDrums**. In *Workshop on Software Engineering for Wearable and Pervasive Computing*, 2000.

[79] Scott Malabarba, Raju Pandey, Jeff Gragg, Earl Barr, and J. Fritz 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.

[80] Suriya Subramanian, Michael Hicks, and Kathryn 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.

[81] Misha Dmitriev and Malcolm P. Atkinson. **Evolutionary Data Conversion in the PJama Persistent Language**. In *Proceedings of the Workshop on Object-Oriented Technology*, pages 25–36. Springer-Verlag, 1999.

[82] Malcolm P. Atkinson, Laurent Daynès, Mick J. Jordan, Tony Printezis, and Spence Spence. **An Orthogonally Persistent Java**. *SIGMOD Rec.*, **25**:68–75, December 1996.

[83] Kristis Makris and Rida Bazzi. **Immediate Multi-threaded Dynamic Software Updates Using Stack Reconstruction**. In *Proceedings of the USENIX Annual Technical Conference*. USENIX Association, 2009.