UNIVERSITY
of
GLASGOW

Safe Class and Data Evolution in Large and Long-Lived Java Applications

Mikhail Dmitriev

Submitted for the Degree of Doctor of Philosophy
Department of Computing Science
University of Glasgow

March 2001

ProQuest LLC.
789 East Eisenhower Parkway
P.O. Box 1346
Ann Arbor, MI 48106 – 1346

Abstract

There is a growing class of applications implemented in object-oriented languages that are large and complex, that exploit object persistence, and need to run uninterrupted for long periods of time. Development and maintenance of such applications can present challenges in the following interrelated areas: consistent and scalable evolution of persistent data and code, optimal build management, and runtime changes to applications.

The research presented in this thesis addresses the above issues. Since Java is becoming increasingly popular platform for implementing large and long-lived applications, it was chosen for experiments.

The first part of the research was undertaken in the context of the PJama system, an orthogonally persistent platform for Java. A technology that supports persistent class and object evolution for this platform was designed, built and evaluated. This technology integrates build management, persistent class evolution, and support for several forms of eager conversion of persistent objects.

Research in build management for Java has resulted in the creation of a generally applicable, compiler-independent smart recompilation technology, which can be re-used in a Java IDE, or as a standalone Java-specific utility similar to `make`.

The technology for eager object conversion that we developed allows the developers to perform arbitrarily complex changes to persistent objects and their collections. A high level of developer's control over the conversion process was achieved in part due to introduction of a mechanism for dynamic renaming of old class versions. This mechanism was implemented using minor non-standard extensions to the Java language. However, we also demonstrate how to achieve nearly the same results without modifying the language specification. In this form, we believe, our technology can be largely re-used with practically any persistent object solution for Java.

The second part of this research was undertaken using as an implementation platform the HotSpot Java Virtual Machine (JVM), which is currently Sun's main production JVM. A technology was developed that allows the engineers to redefine classes on-the-fly in the running VM. Our main focus was on the runtime evolution of server-type applications, though we also address modification of applications running in the debugger. Unlike the only other similar system for Java known to us, our technology supports redefinition of classes that have methods currently active. Several policies for handling such methods have been proposed, one of them is currently operational, another one is in the experimental stage. We also propose to re-use the runtime evolution technology for dynamic fine-grain profiling of applications.

Acknowledgements

I am grateful to the following people for their support, encouragement, and constructive contributions.

- Malcolm Atkinson, my PhD supervisor, for the constant help and encouragement, outstanding enthusiasm and ability to generate new ideas, taking care even of my non-work circumstances, and, last but not least, providing the scholarship for me that covered both my tuition and living expenses in the course of my PhD. Malcolm was a head of the PJama project in Glasgow, but also played a role in the initiating of the "HotSwap" project at Sun Labs, on which I am currently working.
- Craig Hamilton, for implementing the low-level evolution and object conversion mechanisms for PJama, and being always ready to help. The work described in Chapter 6 of this thesis would hardly be complete without his participation. Talking to Craig for almost three years was also an invaluable experience of learning to understand a classic Scottish accent.
- Mick Jordan, for providing the funds and equipment that supported me in Glasgow, and for supporting my internships at Sun Labs
- Gilad Bracha, who was the main initiator of the HotSwap project at Sun Labs, provided many of the ideas that are developed in Chapter 7 of this thesis, and was always willing to advise me.
- Mario Wolczko, my manager at Sun Labs, for accepting me first for the internship, and then for a permanent job, giving valuable technical advice, and constantly helping me to resolve various unexpected and mysterious bureaucratic problems arising all but too often in the US.
- Roberto Zicari, for reviewing (anonymously, of course — but a real expert can be recognised by the quality of his writing) and providing valuable comments to my papers, which were eventually included in this thesis.
- Robert Griesemer from JavaSoft, for his time spent explaining to me the internals of the HotSpot JVM, reviewing my code and teaching me better coding habits, and kindly providing the text for Section 7.1.1 of this thesis.
- Lars Bak, one of the creators of the HotSpot JVM, for giving me the initial technical advice on implementing the dynamic class redefinition functionality.
- The members of the PJama group at the University of Glasgow and at Sun Labs: Greg Czajkowski, Laurent Daynès, Neal Gafter, Brian Lewis, Bernd Mathiske, Tony Printezis, and Susan Spence, for the interesting debates, helpful advice and enjoyable time spent together over the past three years.

The PJama project, within which a large part of my research was performed, was supported by a collaborative research grant from Sun Microsystems, Inc., and a grant from the British Engineering and Physical Science Research Council (EPSRC).

Contents

Abstract i

Acknowledgements ii

1 Introduction 1
- 1.1 The Java Programming Language and Platform . . . 3
- 1.2 Enterprise Applications, Object Persistence and Change Management . . . 5
  - 1.2.1 Enterprise Applications and Java . . . 5
  - 1.2.2 Object Persistence and Java . . . 6
  - 1.2.3 Evolution of Persistent Objects . . . 7
  - 1.2.4 Application Evolution and Build Management . . . 8
  - 1.2.5 Runtime Evolution of Java Applications . . . 9
- 1.3 Thesis Statement . . . 10
- 1.4 Thesis Overview . . . 11

2 PJama System and Its Evolution Model 12
- 2.1 PJama Persistent Platform . . . 12
  - 2.1.1 Orthogonal Persistence . . . 12
  - 2.1.2 History of the PJama Project . . . 14
  - 2.1.3 PJama Persistence Model Overview . . . 17
  - 2.1.4 Summary . . . 22
- 2.2 Requirements for the PJama Evolution Technology . . . 23
- 2.3 Main Features of the PJama Evolution Technology . . . 23
  - 2.3.1 The History of Evolution Tools For PJama . . . 24
  - 2.3.2 The Front End — the `opjb` Persistent Build Tool . . . 26
  - 2.3.3 Support for Operations on the Class Hierarchy . . . 27
  - 2.3.4 Support for Object Conversion . . . 28
- 2.4 PJama Evolution System Architecture Overview . . . 28
- 2.5 The Present Constraints . . . 30
  - 2.5.1 No On-Line or Concurrent Evolution . . . 30
  - 2.5.2 No Lazy Object Conversion . . . 30
  - 2.5.3 No Support for Java Core Classes Evolution . . . 31
- 2.6 Related Work — OODB Systems . . . 32
  - 2.6.1 General Approaches to Change Management . . . 32
  - 2.6.2 Types of Changes Allowed . . . 34
  - 2.6.3 Data and Code Consistency Preservation . . . 34
  - 2.6.4 Support for Evolution in Java Binding . . . 35

2.7 Summary . . . 35

3 Persistent Build Technology **37**
3.1 Motivation and Context . . . 37
3.1.1 General Build Management . . . 38
3.1.2 Smart Recompilation of Java Applications . . . 40
3.2 Management of Evolvable Classes in PJama . . . 41
3.2.1 Evolvable Class Directory . . . 42
3.2.2 Locating `.class` and `.java` Files for Persistent Classes . . . 42
3.2.3 Classes with no `.java` Sources . . . 43
3.3 Persistent Build Algorithm . . . 43
3.3.1 Source Incompatible Changes . . . 44
3.3.2 References in Java Binary Classes . . . 49
3.4 Related Work . . . 50
3.4.1 Build Management in Persistent Platforms . . . 50
3.4.2 Smart Recompilation for Java . . . 50
3.5 Future Work . . . 52
3.5.1 Speeding Up Smart Recompilation . . . 52
3.5.2 "JavaMake" Utility . . . 53
3.5.3 Formal Proof of Completeness and Correctness . . . 53
3.6 Summary . . . 53

4 Persistent Object Conversion **55**
4.1 When Conversion is Required . . . 55
4.2 Types of Conversion: Default and Custom . . . 57
4.3 Default Conversion . . . 58
4.4 Custom Conversion: Class Version Naming . . . 58
4.5 Bulk Custom Conversion . . . 60
4.5.1 Class Modification . . . 61
4.5.2 Class Deletion . . . 62
4.5.3 Conversion for Subclasses of Evolving Classes . . . 63
4.5.4 Semi-automatic Copying of Data Between "Old" and "New" Instances . . . 63
4.5.5 `onConversionStart()` and `onConversionEnd()` Predefined Methods . . . 64
4.5.6 An Example – an Airline Maintaining a "Frequent Flyer" Programme . . . 64
4.6 Fully Controlled Conversion . . . 65
4.7 Copying and Conversion of Static Variables . . . 67
4.8 Access to Non-Public Data Fields and Methods . . . 68
4.9 Related Work . . . 69
4.9.1 Java Object Serialization . . . 69
4.9.2 Commercial OODBMS . . . 71
4.9.3 Experimental OODBMS . . . 75
4.10 Future Work . . . 76
4.10.1 Dictionary-Assisted Conversion . . . 77
4.10.2 Usability Aspects . . . 77
4.10.3 Concurrency and Scalability Issues . . . 77
4.11 Summary . . . 78

5 Conversion Code Semantics and Internals **80**

5.1 Stability of the "Old" Object Graph during Conversion . . . 80
5.2 Loading Classes Referenced from Conversion Code . . . 81
5.3 Programmer's View of the Old Class Hierarchy . . . 83
5.3.1 Uniform Renaming of Old World Classes . . . 83
5.3.2 Supporting the Naming Arrangement for Stable Classes . . . 86
5.4 Extended Implicit Type Casts . . . 87
5.5 Implementation Issues . . . 88
5.5.1 Fetching Classes with Mangled Names . . . 89
5.5.2 Implementation of Uniform Renaming of Old Classes . . . 89
5.5.3 Management of Stable Classes . . . 91
5.5.4 Dealing with Multiple Class Loaders . . . 92
5.6 Evaluation . . . 93
5.7 Future Work — an Alternative Design Investigation . . . 94
5.7.1 Changes to Java with the Present Mechanism . . . 94
5.7.2 A Solution That Does not Change the Java Language . . . 95
5.8 Summary . . . 98

6 Store-Level Support for Evolution 100
6.1 Sphere Persistent Store Overview . . . 101
6.2 Goals for the Evolution Platform . . . 102
6.3 The Store-Level Evolution Algorithm . . . 103
6.3.1 Terminology . . . 103
6.3.2 Temporary Storage for Converted Objects . . . 103
6.3.3 Associating and Replacing Old and Converted Objects . . . 103
6.3.4 The Store-Level Instance Conversion Algorithm . . . 107
6.3.5 Tradeoffs . . . 108
6.4 Initial Performance Evaluation . . . 109
6.4.1 Benchmark Organisation . . . 109
6.4.2 The Experimental Results . . . 112
6.4.3 A Real-Life Application . . . 117
6.4.4 Related Work . . . 118
6.5 Summary . . . 118

7 Runtime Evolution of Java Applications 120
7.1 HotSpot Java VM . . . 121
7.1.1 A Historical Note . . . 122
7.1.2 The Features of the HotSpot JVM . . . 122
7.1.3 Summary . . . 131
7.2 Staged Implementation . . . 131
7.3 Stage 1: Support for Changing Method Bodies . . . 133
7.3.1 Dealing with Active Methods . . . 133
7.3.2 Interface to the Runtime Evolution Functionality . . . 134
7.3.3 New Class Version Loading and Validation . . . 136
7.3.4 Class Transformation inside the JVM . . . 137
7.3.5 Methods, Equivalent Modulo Constant Pool . . . 140
7.3.6 Summary . . . 141
7.4 Stage 2: Support for Binary Compatible Changes . . . 141

7.4.1 Loading New Class Versions . . . 142
7.4.2 Change Validation and Replacement Method Determination . . . 144
7.4.3 Unregistering and Re-registering New Classes . . . 145
7.4.4 Subclass Expanding . . . 146
7.4.5 Class Replacement . . . 147
7.4.6 Summary . . . 147
7.5 Future Work . . . 148
7.5.1 Stage 3: Support for Binary Incompatible Changes . . . 148
7.5.2 Instance Conversion . . . 149
7.5.3 Implementing Multiple Policies for Dealing with Active Old Methods of Changed Classes . . . 150
7.5.4 Dynamic Fine-Grain Profiling . . . 152
7.6 Related Work . . . 153
7.6.1 Dynamic Class Versioning . . . 153
7.6.2 Load-Time Transformation . . . 154
7.6.3 Dynamic Typing . . . 154
7.7 Summary . . . 155

8 Review, Conclusions and Future Work 157
8.1 Thesis Review . . . 157
8.2 Future Work . . . 159
8.2.1 Runtime Evolution — Work in Progress . . . 159
8.2.2 Development of VMs that Support Runtime Evolution . . . 160
8.2.3 Other Directions of Future Work . . . 161
8.3 Conclusions . . . 161

A Command Line Options of the PJama Persistent Build Tool 162

B Conversion Code Examples 164
B.1 Version 1 of the Code . . . 164
B.2 Version 2 of the Code . . . 168
B.3 Conversion Class 1 . . . 171
B.4 Version 3 of the Code . . . 172
B.5 Conversion Class 2 . . . 173

C Specification of JVM Calls Supporting Runtime Class Evolution 175
C.1 `RedefineClasses()` . . . 175
C.2 `PopFrame()` . . . 177

Bibliography 179

List of Figures

2.1 Persistence Architectures used for versions of PJama 15
2.2 The essential PJama API. 19
2.3 An example of using PJama. 19
2.4 An example of two persistent classes, one using another. 25
2.5 Class Hierarchy Evolution Example 27
2.6 PJama Evolution System Architecture 29

3.1 The Java class describing an ECD entry. 42
3.2 Binary Class Constant Pool Entries Structure 49

4.1 An example of conversion function declaration in O_2. 59
4.2 An example of access to "old" fields via reflection. 59
4.3 Instance field initialiser. 61
4.4 Two versions of evolving class `Customer`. 64
4.5 A simple conversion class for evolving class `Customer`. 65
4.6 A conversion method returning instances of multiple types. 66
4.7 An example of fully controlled conversion implementation 67
4.8 An example of method performing conversion in Java Serialization 70
4.9 An example of conversion code from Odberg's work 76

5.1 Management of references during conversion 81
5.2 Referencing both "old" and "new" instances in the conversion code 82
5.3 An evolving class referencing an evolving class 83
5.4 An example of conversion code referencing old and new class versions implicitly. 84
5.5 Old class versions definitions as viewed by the evolution system 85
5.6 An example of referencing a stable class in the conversion code 86
5.7 Addition to the conversion method in Figure 5.6 and the resulting output 87
5.8 Example where explicit "old-to-new" assignment is required, and desirable conversion code 88
5.9 Example of type unsafety in conversion code. 95
5.10 Conversion code in old and new models. 97

6.1 *PID* format and partition organisation. 101
6.2 A hypothetical low-level conversion implementation using global *PID*-table 104
6.3 Low-level conversion implementation using limbo objects 105
6.4 Test store layout example 110
6.5 Old and new versions of the evolving class in test 1. 110
6.6 The initial version of the evolving class in test 2. 111
6.7 Conversion class used in test 3. 111

6.8 Test 1 results . . . 113
6.9 Test 2 results . . . 113
6.10 Test 1 results – fixed $g = 0$ . . . 114
6.11 Test 1 results – fixed $n = 200,000$ . . . 114
6.12 Test 3 results . . . 115
6.13 Test 3 graph cross-section at $n = 200000$ . . . 116
6.14 Test 1 with large number of objects, fixed $g = 5$ . . . 116
6.15 Representations of Geographical Lines . . . 117

7.1 The core of a simple interpreter. . . . 124
7.2 An example of interpreter code generation . . . 124
7.3 An example of a non-guaranteed monomorphic call site. . . . 126
7.4 Internal representation of a class object in HotSpot VM . . . 128
7.5 Java Platform Debugger Architecture (JPDA). . . . 135
7.6 Class transformation when method bodies only are changed. . . . 138
7.7 Linking hierarchically related classes during redefinition. . . . 143
7.8 Loading new class versions. . . . 144

List of Tables

2.1 Changes to Java core classes between JDK1.2 and JDK1.2.2 32

3.1 Changes to class and interface modifiers and affected classes. 46
3.2 Changes to superclass/superinterface declarations and affected classes. 46
3.3 Changes to field declarations and affected classes. 47
3.4 Changes to interface method declarations and affected classes. 47
3.5 Changes to class method/constructor declarations and affected classes. 48

6.1 Benchmark organisation 109
6.2 Summary of the experimental results obtained in synthetic tests. 117

Chapter 1

Introduction

With the widespread adoption of object-oriented languages for server-side application development, there is a growing number of applications which can be characterised by one or more of the following properties:

- objects that they create need to survive application shutdown/restart;
- they consist of a large number (hundreds or thousands) of classes;
- they should run non-stop for weeks or months.

Development and maintenance of such, large and long-lived, applications can present a number of interrelated challenges:

- If some form of object persistence is supported, i.e. objects can outlive an application that created them, the problem arises of how to perform changes. One aspect of this problem is the difficulty of changing both the code (class definitions) and the data (persistent objects) in such a way that they remain mutually consistent. There is also an issue of making this technology scalable, i.e. making it work for an arbitrarily large disk database with limited-size main memory.
- Large numbers of classes constituting an application can lead to a large number of interdependencies between them, which are difficult to keep track of manually and easy to break. For object-oriented languages where compiled applications are not monolithic binaries, but rather relatively loose collections of dynamically loaded and linked classes, broken links (for example, when a method is deleted in one class, but other classes that call it are not updated) can remain unnoticed until a linking exception is thrown at run time. When a language is complex enough, and thus dependencies between classes can take multiple forms, traditional language-independent makefiles do not work well, either causing a lot of redundant, time-consuming recompilation, or failing to recompile programming modules when it is really needed. In the context of systems supporting persistent objects, this can further exacerbate the problem of keeping code and data mutually consistent.
- Currently most of the industrial-strength language implementations do not allow dynamic (on-the-fly) changes to running applications. However, there is a potentially high demand for such a functionality. It may be used at debug time (to fix a buggy method and continue, avoiding shutdown and restart of

the target application), at testing time (e.g. to avoid a painful process of taking down and restarting some complex applications), and even for a deployed application at run time (to diagnose or fix bugs, tune, or to upgrade it).

The research presented in this thesis addresses the above issues. It was undertaken using the Java language and platform for experiments. Java was an obvious choice for us considering its high and still growing popularity, increasing adoption for development of large and long-lived (or *enterprise*) applications, and, last but not least, the relatively open policy of Java's original developer, Sun Microsystems Inc., with respect to the source code of its implementations of Java (as well as the personal connections between the people at the University of Glasgow and Sun Laboratories). As a result, it became possible to establish collaboration between the University of Glasgow and Sun Labs, and to get access to the latest sources of Sun's implementations of Java. To use an industrial-strength product as a research platform is not an opportunity to miss for a computing science researcher! On the other hand, we believe that a number of aspects of our research can be generalised for other languages and platforms, or be used to determine how future systems should be designed to support evolution of large and long-lived applications.

The first part of this research was undertaken in the context of the *orthogonally persistent platform* for Java, the PJama system, that was being developed collaboratively by the University of Glasgow and Sun Microsystems Laboratories. For PJama, we have, first of all, developed a mechanism of flexible, powerful and scalable *persistent object conversion*, that transforms persistent objects to make them match the new definitions of their classes once the latter have changed. A high level of the developer's control over conversion process was achieved in part due to introduction of a mechanism for dynamic renaming of old class versions, which uses minor non-standard extensions to the Java language. However, we also show how to achieve nearly the same results without modifying the language specification. We believe that many aspects of our technology can be re-used with other persistent solutions for Java.

Another piece of work done in the context of PJama evolution subsystem, has resulted in creation of the generally applicable, compiler-independent smart recompilation technology for Java. Such a technology is designed to help the developer handle changes to applications consisting of a large number of interconnected classes. It ensures that, once a change to a particular class is made, all of the other application classes that depend on or may be affected by this particular change, are recompiled, yet avoiding the majority of redundant recompilations. In this way, the development turnaround time is reduced significantly, compared with the case when "recompile all" operation is used all the time. At the same time, the resulting application is guaranteed from linking exceptions or misbehaviour due to links set incorrectly.

The second part of this research was undertaken using as an implementation platform the HotSpot Java Virtual Machine (JVM), which is currently Sun's main production JVM. The author is working on the extensions to this VM, which allow a developer to modify a Java application currently being executed by the VM. The unit of evolution is a class, as in our work on persistent applications evolution. Our main focus is on runtime evolution of server-side applications, though we also address modification of applications running in the debugger (when this technology is used primarily to fix bugs without leaving a debugging session). In the server context, our technology can be used to diagnose and fix bugs as well, and can also be utilised for other purposes, such as application tuning and upgrading. In addition, we believe that it can be used (or its components re-used) for dynamic fine-grain profiling.

In the following sections, we present an overview of the Java language and platform, discuss the issues addressed in this research in more detail, make the thesis statement, and present an overview of this thesis.

1.1 The Java Programming Language and Platform

This description of Java's history builds on that of my colleague Tony Printezis [Pri00].

In 1990, a product-focussed project called *Green* was started at Sun Microsystems by James Gosling, Patrick Naughton and Mike Sheridan. The former recalls [Gos97]:

> "[The project] was chartered to spend time (and money!) trying to figure out what would be the "next wave" of computing and how we might catch it. We quickly came to the conclusion that at least one of the waves was going to be the convergence of digitally controlled customer devices and computers."

So, the Green project concentrated on consumer devices, such as televisions, videos, stereos, etc., and how they can communicate with each other. Such devices are made with different CPUs and have limited amounts of program memory. This encouraged the team to develop a new programming language (initially named "Oak" by James Gosling after a large oak tree outside his window), designed to allow programmers to support more easily dynamic, changeable hardware. To make applications written in this language portable, they were compiled into machine-independent bytecodes that would be then interpreted by an Oak interpreter (virtual machine). The primary goal when designing the bytecode set was to minimise the size of compiled applications. This defined the present shape of Java bytecodes, where each bytecoded instruction is encoded by just a single byte, however some of these instructions are pretty complex. For example, a single variable-length instruction effectively represents the whole "`switch () ...`" statement of the language.

The Oak language was object-oriented, syntactically very similar to C++. However, it was "stripped-down" to a bare minimum. This was done partially to make applications and the virtual machine fit into limited-size memory that small devices would offer, and partially to meet the deadlines. In particular, Gosling writes:

> "My major regret about that spec is that the section on assertions didn't survive: I had a partial implementation, but I ripped it out to meet a deadline[1]"

This problem can be understood easily, considering that in just 18 months the group consisting of a handful of people managed to design, build and successfully demonstrate a new SPARC-based, hand-held wireless device (Personal Digital Assistant, PDA) called *Star7*. It contained an impressive list of features, both hardware and software. The fact that all of the application software was implemented in such a short period of time and was easily changeable, was largely due to the design of the language, which eliminated most of

[1] Interestingly, language support for assertions was one of the most common requests from the developers since Java was released. The forthcoming release of Java (JDK1.4) will finally contain this feature.

the problematic aspects of C++. It supported single class inheritance, but also allowed classes to implement multiple *interfaces*. It was type safe, did not allow direct pointer manipulations, and its memory management relied on a garbage collector. Some extra features included in Oak were exceptions and built-in concurrency facilities (language-level support for multiple threads and their synchronisation).

Unfortunately, the Green project never achieved any commercial success and was eventually cancelled. The reasons were that the TV set-top box and video-on-demand industries, which it was targeting, were in their infancy, and still trying to settle on viable business models.

However, around that time, in 1993, the National Center for Supercomputing Applications (NCSA) introduced the World Wide Web (WWW) and the Mosaic (later Netscape) browser. The Internet was becoming popular as a way of moving media content – text, graphics, video – throughout a network of heterogeneous devices. Java technology had been designed in parallel to move media content across networks of heterogeneous devices, but it also offered the capability to move "behaviour" in the form of applets along with the content. HTML [RHJ98], the text-based markup language used to present information on WWW, could not do that alone, but it did set the stage for Java technology. So, in 1994 Oak was retargetted to the WWW and was renamed Java. Later a deal was made with Netscape Corp. for Java interpreter to be included in the Netscape browser.

One reason for the subsequent huge success of Java was the decision to release the first full public alpha version (1.0a2) of the Java source code on the Internet. This happened in March 1995. "We released the source code over the Internet in the belief that the developers would decide for themselves" recalls team member Lisa Friendly [Byo98]. The team knew that releasing code to developers for free is one of the fastest ways to create widespread adoption. It was also the easiest way to enlist the help of the development community to inspect code and find any overlooked bugs.

In just a few months, the downloads began to surge into the thousands. The 10000 downloads barrier that Gosling considered an indicator of a huge success, was broken in surprisingly short time, just a few months, and the amount of downloads continued to grow exponentially. Soon Sun realised that the Java technology team's popularity was quickly and haphazardly outpacing its own carefully orchestrated popularity. All of this was happening while Java had virtually no marketing budget or plan.

Six years later, Java is hailed as one of the biggest advances in computing. Software companies have started marketing products written entirely in Java and developers report improvements by up to a factor of two in development time and by up to a factor of three in robustness, when using Java instead of C++ [Phi99]. The Java language has been augmented with a large set of standard classes and APIs, covering a large number of facilities needed by developers. These include 2D/3D graphics (Java 2D/3D APIs [Sun99a, Sun00g], networking (Java RMI [Sun00n]), relational database connection (JDBC [Sun00p]), component technology (Java Beans [Sun00a]), etc. The combination of the Java language and the set of standard APIs (currently comprising several thousand Java classes and interfaces) is known as the *Java Platform*. The presence of these APIs on all hardware/OS platforms that Java supports greatly increases the portability of Java code, since in most cases the developers can avoid using *native methods*, i.e. adding native and platform-dependent code to their Java applications. Sun's implementation of the Java Platform (*Java Development Kit* (JDK) [2]), currently in version 1.3 [Sun00d], is considered to be the standard.

[2]Lately, JDK has been renamed to *Java*[tm] *Software Development Kit* (SDK), introducing naming complexity and some confusion, that we would like to avoid. The official full name for what is referred to both internally at Sun and throughout this thesis as "JDK1.3" is "Java 2 Platform SDK Version 1.3".

At the same time, a number of companies are marketing complete Java platform systems and development environments. A few examples are JBuilder from Borland [Bor00c], CodeWarrior from Metrowerks [Met00], VisualAge from IBM [IBM00b], J++ from Microsoft [Mic00b], etc. Ironically, the development tools from Sun itself remained in the same old-fashioned, command-line style for long time. Only recently, in summer 2000, Sun has finally introduced its own integrated development environment (IDE) for Java called *Forte for Java* [Sun00b]. However, this IDE has a significant advantage from the point of view of researchers and independent developers: it is not just a yet another proprietary product. Instead, Forte for Java is actually a stable "snapshot" of the code of the project called NetBeans [Net00]. The latter project is open source, which means that its code (and hence most of the code of Forte, though the latter also contains some proprietary modules) is freely available to anyone, and anyone is welcome to submit new code and bug fixes.

Despite the huge success of Java, it is worth noting that it did not introduce any new concepts or ideas. Instead, it incorporates a lot of already existing ones. Some of them were already popular and familiar to many programmers (e.g. C++ like syntax), some of the others were useful parts of otherwise not very popular systems — e.g. bytecoded representation of compiled programs or garbage-collected memory management. Perhaps the secret of Java's huge success was in the right combination of all of these features, the attractiveness of "write once, run anywhere" concept (though in reality it is not always easily achieved), availability of the free implementation, and, last but not least, the unprecedented hype around it that was orchestrated by Sun.

At present Java Platform implementation from Sun is available in three separate *editions*: Standard Edition, Enterprise Edition and Micro Edition. Standard Edition is oriented towards individual developers or groups working on client-side, standalone or simple server-side applications. Enterprise Edition combines a number of technologies relevant for development and deployment of large server-side and distributed enterprise applications, e.g. Enterprise Java Beans [Sun00a], Java Server Pages [Sun00m], Compatibility Test Suite [Sun00c], etc. Micro Edition is a highly optimised Java runtime environment targeting a wide range of consumer products, including pagers, cellular phones, screen phones, digital set-top boxes and car navigation systems. This looks very close to the original goal for Oak.

The official definition of the Java language is given in the Java Language Specification (JLS), the second edition of which by Gosling, Joy, Steele and Bracha was recently published [GJSB00]. The official Java Virtual Machine Specification, now also in Second Edition, is published by Lindholm and Yellin [LY99].

1.2 Enterprise Applications, Object Persistence and Change Management

1.2.1 Enterprise Applications and Java

The constant advances in hardware technology resulting in cheap and ubiquitous computing, and the wide adoption of object-oriented programming languages and advanced programming tools, that allow the developers to build much bigger applications much faster than before, has provoked an unprecedented growth in number, size and complexity of software applications. This was further stimulated by the growth of popularity of the World Wide Web (WWW), that has forced many organisations to establish their presence on it, first in the form of mere static information sites, and then in the form of interactive Web applications de-

livering various kinds of services — from news and education to banking and e-commerce. An application that runs on the Web site of a computer manufacturer, and controls all of the stages of making a purchase: interactive selection of machine configuration and price determination, taking an order using a credit card, sending an order to the manufacturing workshop, and shipping the package — is an example of an *enterprise application*, characterised by high complexity, and also high demand for constant changes. Computer models, parts and possible configurations are constantly being upgraded, price determination algorithms are being changed, various seasonal or conditional discounts are introduced, etc. Managing such an application, in the circumstances when any significant downtime leads to substantial financial losses, is certainly not a trivial task.

The main focus of the Java platform has been shifting in the recent years from client-side to server-side, enterprise applications, and a number of features, mentioned in the previous section, have been introduced to support such applications. Combined with the original properties of Java, such as the automatic memory management, availability for all of the major hardware/OS platforms, a vast number of standard libraries supporting virtually any programming area, etc., they make Java an increasingly attractive choice for enterprise application development.

1.2.2 Object Persistence and Java

All of the data which is created and manipulated in the main memory during the execution of a given process, is typically discarded automatically at the end of its execution by the operating system[3]. However, most non-trivial applications need to retain at least some data across multiple invocations. Data (and, maybe, application state as well) preservation outside the main memory is called *persistence*. It is usually achieved by storing the data on secondary storage, using *ad hoc* custom-made schemes (e.g. flat files), more advanced programming language facilities (such as object serialization in Java), or heavy-duty databases (e.g. relational or object-oriented).

A recent report [Atk00] contains an exhaustive survey of the presently existing persistence solutions for Java. These include Java Object Serialization (JOS) [Sun00j], connections to relational databases (JDBC) [Sun00p], automated object-relational mappings [Sun00h], object databases with a Java binding (see Section 4.9.2), Java Data Objects technology [Rus00], and the Enterprise Java Beans (EJB) framework [Sun00a]. The very number of these solutions suggests that none of them is ideal — though, of course, there are certain practical reasons justifying their co-existence. E.g. JOS may be a reasonable choice for easy-to-implement storage of small amounts of non-critical data, whereas connections to relational databases are often an inevitable choice for legacy RDBs.

Orthogonal Persistence, discussed in detail in Section 2.1.1 was proposed by Atkinson in 1978 [Atk78] to provide seamless integration between the programming language and the storage mechanism, that renders unnecessary the need for a separate database system, along with the complexity such a separation introduces. The latter may include a schema modelling language, rules and restrictions that need to be followed during application development, new concepts and APIs to be understood, etc. In applications running on top of an orthogonally persistent platform, data modelling, operations such as long-lived data reading/writing, and

[3]The exception are persistent operating systems, such as Grasshopper [DdBF+94] or L4 [Lie96]. In such systems, processes can be made persistent and survive across machine power-downs. Such systems, however, are still at an experimental stage and are not widely used.

ensuring schema consistency, are achieved in a way that is familiar and natural to the developer: the standard constructs of the programming language.

PJama was a project that aimed at providing orthogonal persistence for Java by modifying the JVM[4]. Any Java application running on PJama platform executes against a *persistent store*, an equivalent of a database, which is specified at the VM startup using a command line parameter or an OS environment variable, and can not be changed for an active VM. The choice of the *roots of persistence* model should also be made at the VM startup time. Either all of the `static` variables of the main application class, or all of the static variables of an explicitly specified (using a simple Java API) class can serve as roots of persistence. All of the objects reachable transitively from the persistent roots are saved atomically and durably into the persistent store either upon a normal (i.e. not an abrupt, caused by an unhandled exception) program shutdown, or using a "checkpoint" API call. The above two API calls, that is, persistent root specification and checkpoint, are the only calls that the application programmers typically need to use in most of the applications. More detailed description of PJama is presented in Section 2.1, and also, along with the analysis of the mixed success of the this project, in [Atk00, AJ00].

Whether or not orthogonal persistence succeeds in future, technologies allowing programming language objects and complex data structures (rather than their, often awkward and inconvenient, representations as e.g. relational database records) to persist, will remain highly appealing to developers. However, to make the structure of the saved data malleable, and hence the data itself able to survive application specification changes, bug fixes, and application evolution, a special evolution technology is required. Why this technology is so important for object persistence, is discussed in the next section.

1.2.3 Evolution of Persistent Objects

Once objects and their collections are made persistent, a question arises immediately of what to do if the definition of any class(es) describing these objects (database schema) has changed. The answer depends significantly on the details of the particular persistence solution, and also on the degree of convenience, safety and scalability we want to achieve. If we consider all of these characteristics, then on the lower end of the scale we will have the "technology" of writing an application that saves a persistent object collection in some custom format, e.g. in a text file. This intermediate representation should then be re-read by another application that uses new class definitions and creates the matching persistent collection in the new format. However, encoding all of the translations between formats manually is obviously very inconvenient, labour intensive and error prone. Furthermore, such an approach can simply not work in certain situations, e.g. when a large amount of objects are connected into a complex graph. In this case, to correctly preserve all the links between the objects in the intermediate representation, we would have to load them all into main memory and then walk the graph according to some algorithm. Main memory may be insufficient to accommodate all of the objects at once, but splitting the graph may also be problematic. Thus, more sophisticated technologies for object evolution should automate the evolution process by at least preserving the links between objects while they are transformed, and they should be capable of handling arbitrarily large graphs of objects.

[4]Other approaches, such as pre- or post-processing the application classes, are also possible, but modifying the VM generally seems to provide better results in terms of the execution speed, as well as the development convenience. This is due to the fact that no special additional bytecodes are executed, and it is not required to perform an additional code processing operation.

Another issue is what kinds of changes can be actually performed using a particular evolution technology. Again, on the lower end of the scale will be the technologies allowing only very simple, e.g. *default*, transformations. For example, if in both the old and the new class definitions there is a `String name` data field, such a technology would correctly copy the value of this field from the object in the old format to the corresponding object in the new format. It would, however, be unable to sensibly handle a transformation where each `name` field should be split in the new object into two fields, `firstName` and `lastName`. Obviously, to support such a change, we need to provide a mechanism that allows the developer to specify how to perform the required transformation, preferably in the same language that is used to write normal persistent applications.

Thus, in our opinion, a really adequate conversion technology should allow the developers to perform arbitrary transformations of objects and larger data structures that embrace them (for example, convert a list into a tree or vice versa), and work for arbitrary large amounts of persistent objects with limited-size main memory. The technology which we built for PJama, satisfies, as we believe, these requirements. On the other hand, after surveying the object conversion facilities presently available in the existing persistent solutions for Java and other languages (see Sections 2.6 and 4.9) we can conclude that, to the best of our knowledge, there is no other system that matches them equally well.

1.2.4 Application Evolution and Build Management

Java is distinct from most of the other widely adopted industrial-strength programming languages in at least one aspect, and that is its late linking model. Java applications are not monolithic[5] — rather, they are loose collections of classes, for which the final linking phase is replaced with a lazy incremental resolution process that happens every time a new class is loaded by the JVM. In addition, dependencies between Java classes can take many forms. Class `C` can call a method of class `D`, be `D`'s subclass, implement interface `D`, declare a field or a local variable of type `D`, to name but a few. Some changes to `D` will affect `C` and some not, depending on the actual kind of relation(s) between `C` and `D`. For example, consider what happens if method `m()` is deleted from `D`. If `C` previously called this method, it should necessarily be updated and recompiled. However, if `C` simply declares a local variable of type `D`, nothing needs to be done to `C`.

While allowing independent evolution of classes that constitute a single application or participate in a number of them (which is especially useful in case of distributed, geographically disperse development), these properties of Java also increase significantly the probability of incompatible changes going unnoticed until a linking exception is thrown at run time. Worse than that, some of those changes may not lead to an exception, but result in incorrect program execution. One example is how `static` methods are actually called by Java bytecodes. Suppose that we have two classes, `C` and its subclass `D`, and a static method `m()` defined in `C`, which we call in yet another class `A` as `D.m()`. If we then override `m()` in class `D`, the call from `A` will still go to `C.m()`, contrary to what one would expect from the late linking model. To correctly handle this change, the source code of `A` must be recompiled.

Considering these properties of Java, it is highly desirable to have a *smart recompilation* technology that, given all the classes constituting an application, would automatically keep track of incompatible changes and recompile all the dependent classes, thus assuring that any links that may potentially be broken are

[5] According to the classic model. There are at present a number of Java native compilers, e.g. Visual Cafe [Web00], which compile Java sources into monolithic binary executables — but this eliminates portability and precludes evolution.

either checked or updated. On the other hand, classes should not be recompiled unnecessarily. It turns out that very few of the existing products for Java support smart recompilation completely (i.e. handle correctly all of the incompatible changes that we have managed to discover). Those that do, support it in an undocumented way, and seem to exploit intermediate, memory-intensive data structures created by the Java compiler. In contrast, in this work we present a comprehensive list of incompatible changes that can be made to Java classes, and for each such change — a method for determining all of the classes that might be affected. Using them, it is possible to implement an efficient smart recompilation technology that would be independent of a particular Java compiler. We have done that for PJama, creating what we call the *persistent build* technology, that combines smart recompilation with persistent class evolution.

1.2.5 Runtime Evolution of Java Applications

For a large class of critical applications, such as business transaction systems, telephone switching, call centers or emergency response systems, any interruption poses very serious problems. Therefore facilities that allow engineers to update an application on-the-fly, or at least to, say, modify it to help diagnose a bug that is hard to reproduce in synthetic tests, can be very attractive. The same technology can be used to fine-tune the code for e.g. specific hardware configurations or security requirements. During debugging, the same technology allows the developer to fix a bug in a method and continue, without leaving the debugging session.

Facilities allowing an application to be modified at run time have been around for some languages, e.g. Smalltalk [GR83, Gol84] and CLOS [Sla98], for decades. However, until recently all of the work of this kind for Java was in the area of load-time bytecode instrumentation. A class could be modified at load-time, but once it was active, it could not be changed. A lack of works on runtime evolution for Java can probably be explained in part by more complex design of the Java language and of the JVMs, compared to e.g. Smalltalk. Thus, it would cost a lot to develop from scratch any JVM, let alone one which supports runtime evolution. On the other hand, existing JVMs, at least those for which the source code is publically available, were not designed with runtime evolution in mind, and thus can not be easily modified to support it. In addition, Java is a type safe language. Therefore (again in contrast with Smalltalk or CLOS) a technology for runtime class redefinition should either allow only limited (e.g. *binary compatible*, see [GJSB00], Chapter 13) changes to be made, or be able to check that the updated application still conforms to the type safety rules of Java. The latter is not easy, as we show in this work.

So, only about a year ago the first work [MPG+00] appeared, that described a type safe runtime evolution technology for Java. Type safety was achieved simply by allowing only restricted, namely binary compatible, changes. Furthermore, classes whose methods are currently active, can not be replaced by this system. Some other restrictions and the fact that it was implemented for Sun's first JVM, which is currently no longer in use, made this system more like an interesting experiment, rather than an industrial-strength implementation. Nevertheless, at the ECOOP'2000 conference where this work was presented, it was a huge success.

The author started to work on the runtime evolution extensions to the HotSpot JVM, which is Sun's present production VM, in August 2000 as a summer intern. From the beginning we decided that we should support evolution of classes that have methods currently active, and we devised several policies for dealing with such methods. The simplest one, which we currently have working, is to allow all of the calls to old method to

complete, while dispatching all of the new calls to new methods. We have also suggested two other policies, and for one of these (on-the-fly switch from old method code to the new code) developed an experimental implementation.

Being aware of various difficulties we could encounter, we have devised a plan of staged implementation, where each stage corresponded to a relatively consistent level of functionality, that could be included in some release of HotSpot. In the first stage, which is now complete and operational, our implementation supports changes only to method bodies. It works both in the debugging and in the server context, supporting evolution of applications that run in mixed (interpreted and compiled) mode. This code will be included in the forthcoming release of HotSpot (JDK1.4), scheduled for the end of the year 2001.

In the subsequent stages we are going to implement support for less restrictive changes: binary compatible (in the second stage, which is now close to completion), and any changes that do not violate type safety for a particular application (in the third stage). We are also considering implementing object conversion support. Since the latter facility does not by itself raise any type safety issues (which determine the difference between the stages 2 and 3), it can be implemented at any time.

1.3 Thesis Statement

There is a growing class of applications implemented in object-oriented languages that are large and complex, that exploit object persistence, and need to run uninterrupted for long periods of time. Java is becoming increasingly popular platform for such applications. Existing technologies supporting evolution of Java code and data, whether in the context of object persistence, build management, or runtime class redefinition, are mostly inadequate or simply undeveloped.

A technology that supports persistent class and object evolution for the PJama orthogonally persistent platform was designed, built and evaluated. This technology integrates build management, persistent class evolution, and facilities for various forms of eager conversion of persistent objects. The complexity and depth of changes to the persistent data, that the conversion facilities of PJama allow the developers to perform in parallel with schema change, are, to our best knowledge, unmatched by other systems. We demonstrate that parts of this technology can be re-used in other contexts. For example, smart recompilation for Java would be useful in a Java IDE or can be implemented in a standalone utility, and the slightly modified solution for managing class versions during custom conversion can be adopted for practically any persistent object solution for Java. It is argued that the feasibility of this technology and the appropriateness of its semantics has been demonstrated.

A technology that supports runtime evolution of Java applications was developed for the HotSpot production JVM. It allows the developers to redefine classes on-the-fly in the running JVM, and, unlike the only other existing similar system for Java, supports redefinition of classes that have methods currently active. Several policies for handling such methods have been proposed, and one of them is currently operational. Reusing the runtime class redefinition technology for dynamic fine-grain profiling of applications has been proposed. It is argued that the dynamic evolution technology will be very useful to application engineers during development, maintenance and operation. At least simple versions of this technology have been shown to be practical.

1.4 Thesis Overview

The thesis is organised as follows.

In Chapter 2 we discuss the concept of orthogonal persistence, and give an overview of the PJama platform, which was the base system for developing and experimenting with persistent class and object evolution. We explain how our evolution system has itself evolved, present its architecture, and explain why it has certain limitations.

Chapter 3 describes our persistent build technology, which combines class evolution and smart recompilation. We present the algorithm which is used by our smart recompilation procedure, and the comprehensive tables of source incompatible changes.

In Chapter 4 we present our object conversion technology for PJama. We describe several types of conversion that we provide, discuss the issue of naming of old class versions when custom conversion is used, and present our solution to this problem. The discussion is illustrated on a simple running example.

In Chapter 5 we discuss the semantic issues which come to the fore when databases more close to real-life should be converted, our solutions, and the important internal implementation details. We also present an alternative design of custom conversion support, that, unlike our present one, does not require changes to the Java language — at a price of providing somewhat less convenience to the developer.

Chapter 6 describes the details of the persistent store – level layer of our evolution technology, which makes the latter reliable and scalable. We also present the initial performance evaluation results.

Chapter 7 is devoted to the description of the runtime evolution technology which we are developing for the HotSpot JVM. The plan of staged implementation of this technology is presented, and the implementation of the first (complete) and the second (close to completion) stages is discussed. A number of design ideas for the future is presented.

Finally, Chapter 8 summarises the thesis and presents the conclusions.

Chapter 2

PJama System and Its Evolution Model

In this chapter we first give an overview of the PJama system, which was the base system for developing and experimenting with persistent class and object evolution. In addition to PJama details, we also present the concept of orthogonal persistence, on which PJama is based, and a brief history of this project. We then establish the context of PJama evolution and consequently identify the functions it must fulfill in order to maintain the integrity of persistent applications while changing them (Section 2.2). We explain how, through several intermediate implementations, we came to the present persistent build technology, which combines evolution with smart recompilation (Section 2.3). In Section 2.4 we outline the layered architecture of our evolution system. Finally, in Section 2.5 we explain why our system has some constraints at present, and in Section 2.6 present the related work.

2.1 PJama Persistent Platform

Before we proceed to the details of the PJama platform which are important in the context of this work (Section 2.1.3), we will explain the principles of orthogonal persistence on which this platform is based (next section). We will also present the history of this project to the present time (Section 2.1.2).

2.1.1 Orthogonal Persistence

Orthogonal Persistence (OP) [AM95] is a language-independent model of persistence, defined by the following three principles:

1. Type Orthogonality
2. Persistence by Reachability
3. Persistence Independence

These were first introduced by Atkinson *et al.* [Atk78, ACC82, ABC+83], summarised by Atkinson and Morrison [AM95], and their applicability to Java analysed by Atkinson and Jordan [JA98, AJ99, AJ00]. Below we explain the above three principles in more detail.

2.1.1.1 Type Orthogonality

> "Persistence is available for *all* data, irrespective of type."

Many systems claim that they provide orthogonal (sometimes also referred to as *transparent*) persistence. However, deeper investigation often reveals that rather than allowing "any data type" to persist, they allow "any data type, provided something". "Something" may be one of the following: the data type is a subclass of a persistent-enabled class, implements a certain persistent-related interface, has mapping–unmapping facilities defined on it, etc. It can be argued that this is not orthogonal persistence, and definitely not transparent, since the application programmer has to decide for *each* data type whether it can persist or not, and, in some cases, has to write mapping code by hand.

2.1.1.2 Persistence by Reachability

> "The lifetime of all objects is determined by reachability from a designated set of root objects, the so-called *persistent roots*."

This concept (also referred to as *transitive persistence*) is a natural extension to the world of long-lived objects, of main memory object management model that utilises garbage collection. A counter-example to this concept is a system which enforces the programmer to use explicit delete operations in order to reclaim space in the persistence storage, whereas the programming language that it targets is one with garbage-collected main memory.

2.1.1.3 Persistence Independence

> "It is indistinguishable whether code is operating on short-lived or long-lived data."

Again, a counter-example that demonstrates the benefits of this principle is a system that requires the programmer to use separate (though in practice they are typically very similar) data structures (classes) for memory and disk data (objects). This can lead to parallel existence of two versions of the similar code, the API of the storage system "getting in the way" of the algorithm being implemented, problems with porting the existing code to a different storage system, problems with using third-party libraries over persistent data, etc.

2.1.1.4 Benefits of Orthogonal Persistence

Given the above three principles, the number of operations required to use the orthogonally persistent system, and hence the amount of additional knowledge required from the programmer, are minimal. In fact, apart from the usual facilities that a programming language provides, the only extra persistence-specific calls needed are the following:

- **Register and retrieve the persistent roots.** The programming style that maximises the benefits of OP is when persistent roots are *not* associated with small object graphs, but rather with entire applications. Therefore, the calls that deal with persistent roots are invoked very rarely, typically only once inside the application-startup code. Code analysis by Grimstad *et al.* [GSAW98] supports this claim.
- **Perform a checkpoint.** This operation forces any changes performed on the data to propagate to the disk atomically and durably. If a previously transient object becomes persistent as a result of this operation, we say that this object is *promoted* into the store.

Another important benefit typically provided by an orthogonally persistent system, is incremental on-demand object fetching. This makes such systems much more responsive and eliminates the "big inhale" problem, which is a slow initialisation of a system due to fetching from disk all of the data that it needs (and often much data that it doesn't need). A feature symmetrical to incremental object fetching is incremental *object eviction*, which removes currently unused persistent objects from main memory, thus allowing an application to work with large numbers of persistent objects using relatively small-size main memory.

2.1.2 History of the PJama Project

PJama was initially started as a sub-project of the Forest project initiated by Mick Jordan and Michael Van De Vanter at Sun Laboratories (Sun Labs), California, in 1994. The goal of Forest was to investigate advanced, scalable and distributed configuration management for software development environments. Instead of traditional, and apparently inappropriate, approach of using files, directories or ad-hoc persistence schemes, it was proposed to use some kind of persistent object technology [JV95]. The initial prototype was developed, written in C++ and using ObjectStore [LLOW91] for the persistent facilities.

In 1995 the growing popularity of Java became obvious, so the Forest team refocused their efforts on it and decided to re-implement their system in Java. However, no persistence mechanism was available at that time that was free of the pitfalls encountered when using ObjectStore (such as explicit free-space management or manual object clustering). Thus, in September 1995, the collaboration between Sun Labs and the Department of Computing Science of the University of Glasgow was initiated with the goal to investigate the feasibility of introducing orthogonal persistence to the Java programming language. This project was named *PJava*, standing for Persistent Java. However, that name was later reserved by Sun Microsystems, Inc. for their *Personal Java* product. Therefore the project was later renamed *PJama*. Meanwhile, the Forest team built the second prototype of their system, targeted for development in Java. It was written in Tcl/Tk [Ous93, Ous90], used files/directories for its persistent facilities, and was a temporary measure until the PJama system was operational.

The development in Glasgow was led by Malcolm Atkinson, and the first two team members were Laurent Daynes and Susan Spence. Tony Printezis joined them later, in the end of 1995. The first usable version of PJama was shipped to Sun Labs from Glasgow in July 1996. It was based on Sun's JDK 1.0 and used the architecture summarised in Figure 2.1(a). Durability and atomicity depended on Recoverable Virtual Memory (RVM) [SMK+94]. There was a buffer pool into which pages were brought, and a separate object cache, in which objects appeared to the VM as if they were in the standard garbage-collected heap. The persistent store was implemented as a single file that had flat structure, i.e. object addresses on disk (*PIDs*) were simply byte offsets from the start of the file. More features, such as sophisticated object cache management that included eviction of persistent objects to recycle the main memory space occupied by non-mutated persistent objects [DA97], explicit stabilisation (checkpoint) during execution, a simple disk garbage collector [Pri96, Ham97], etc., were added to it and a port to JDK1.0.2 was done before the first public PJama release, around January 1997. At the same time, the configuration management system developed by the Forest group was being rewritten in Java, using PJama for its persistent facilities. It was eventually named *JP*, after a series of name changes. More information on JP and the configuration management work is given by Jordan, Van De Vanter, and Murer [JV95, JV97, Van98, VM99, MV99].

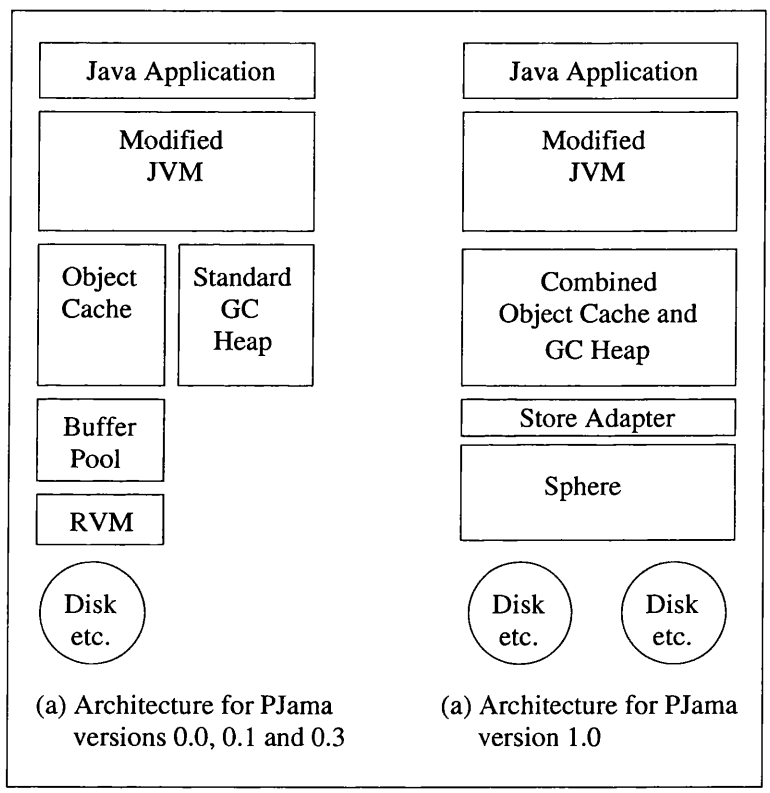

Figure 2.1: Persistence Architectures used for versions of PJama

During 1997 and 1998, PJama Virtual Machine development slowly migrated from Glasgow University to Sun Labs, where Laurent Daynès moved in 1997. Bernd Mathiske joined the PJama Sun Labs team at about the same time, and was followed next year by Brian Lewis and Neal Gafter. A sequence of ports to 1.1.x versions of the JDK were made. The PJama system based on the first Sun's Java Virtual Machine (Sun's *Reference Edition* Virtual Machine, informally known as Sun's *Classic JVM*) and the initial flat stable store architecture, started being referred as *PJama Classic*. The work in Glasgow also continued, e.g. distribution support (*persistent RMI* [SA97, Spe99, Spe00]) was implemented. In 1997 the author joined

the PJama Glasgow team to work on evolution technology. At the same time, Craig Hamilton started to work on logging and recovery support for the next generation persistent store, and later played the key role in implementing low-level support for evolution (see Chapter 6).

The final PJama Classic release was based on JDK1.2 Reference Edition. Due to a large amount of internal changes made in the Sun's JVM between versions 1.1.x and 1.2, porting PJama to JDK1.2 proved more complex than originally anticipated. Still, after a substantial effort at Sun Labs, it shortly followed the release of JDK1.2. By August 1999, 150 sites worldwide had downloaded the PJama Classic system.

However, by the end of 1998 it was all but clear that the current PJama architecture, (which was initially viewed as write-and-throw-away prototype), was no longer adequate. First of all, it did not scale in several ways. The number of persistent objects that could be updated per stabilisation was limited due to the no-steal policy imposed on the buffer pool and object cache by RVM. This policy required all updated objects to be resident in the object cache and their corresponding store pages to be resident in the buffer pool until stabilisation operation has been committed. Then, only a single disk file could be used for a persistent store, hence store size was limited to 2GB — the maximum size for a Unix file. Addressing objects by their physical offsets rendered on-line object relocation (and hence any concurrent relocating disk garbage collection or reclustering schemes) virtually impossible, since if an object was relocated, its *PID* would have changed. The same problem was also precluding efficient and scalable implementation of store-level support for object evolution, forcing it to struggle with changing *PID*s of evolving objects. Main memory management was also not completely adequate, since the object promotion mechanism imposed equal size requirements on the GC heap and object cache. Both of them had to be linearly proportional to the volume of promoted objects, since objects being promoted were copied from the heap to the object cache.

To address these and other problems, two fundamental design shifts were made. First, in the beginning of 1997 the work on the new store architecture called *Sphere* started in the University of Glasgow. This store implementation was designed to be independent on any particular VM or other workload, communicating with it via clean and well-defined C language API. For more detailed Sphere overview, see Section 6.1. Second, by the end of 1998 Sun Microsystems already had two new Java Virtual Machines: HotSpot VM (see Section 7.1) and Sun Labs Research VM (SRVM), informally known as Exact VM (EVM) [WG98][1]. Both of them were substantially faster, more scalable and reliable than the classic VM. Eventually EVM, which by that time looked more stable and scalable of the two, and, compared to the old Java VM, offered significantly improved pointer tracking, better memory management and a dynamic native compiler, was chosen as a platform for the development of new PJama architecture, denoted PJama_1. This architecture is depicted on Figure 2.1(b), and its detailed description can be found in [LM99, LMG00].

Adding initial persistence support to EVM making it Persistent EVM (PEVM) started at Sun Labs in January 1999 and was achieved in just about five months. Subsequent integration with Sphere also took very little time and was done completely remotely, i.e. there was no need for the developers of both parts of the system to work together at the same geographical location. The fast and successful delivery of the initial working version of PJama_1 proved the quality of modular design of PEVM and Sphere. It also justified our deliberate rejection of practically any kinds of optimisations at this stage of development. This was in

[1] Actually, this Java VM was derived from the Sun's Classic VM and was originally called Exact VM. This name refers to its ability to support *exact* memory management, i.e. all the pointers on the runtime stacks and objects are known to the garbage collector (such an approach is also called *non-conservative* or *accurate* [Jon96]). EVM was released by Sun in the JDK1.2, Solaris Production Release. However, in the subsequent JDK releases it was replaced with the HotSpot VM. At the same time, in Autumn 1999, EVM was renamed Sun Labs Research VM and was retargetted to purely research purposes.

a sharp contrast with PJama Classic, where some early and unjustified optimisations have, due to gradually increasing complexity of the Sun's VM itself, eventually resulted in a high amount of complex and hard to manage code. For example, one of these optimisations was saving classes in the persistent store in the format more close to the main-memory representation rather than as a usual class file, in the hope that this will speed up loading persistent classes. When this optimisation was removed as a matter of experiment, it appeared that the performance of the resulting "simplified" system improved, rather than degraded!

Over a year between spring 1999 and spring 2000 PJama$_1$ has undergone a series of gradual improvements and bug fixes. Persistent object eviction mechanism in PEVM was implemented, a number of internal optimisations were made that significantly improved the performance, evolution functionality previously available in PJama Classic was re-implemented and new evolution facilities added: *persistent build* technology (Chapter 3) and store-level support for scalable instance conversion (Chapter 6). The system achieved the level of scalability and robustness it never had before. Unfortunately, in summer 2000 the project was discontinued at Sun Labs. At present PJama is used as a research system at the University of Glasgow.

By this time, a number of application projects exploiting PJama were implemented. An interested reader can find this information, as well as the exhaustive description of all aspects of PJama, in [AJ00, Atk00].

2.1.3 PJama Persistence Model Overview

PJama makes data continue to be available for as long as it may be accessed by some future computation, in other words, makes data (objects) and its schema (classes) persistent. In this section we are presenting the definition of precisely what data is made persistent and when this happens. This discussion of PJama's persistent mechanism is closely based on the OPJ Specification [JA00].

2.1.3.1 When Objects are Made Persistent

Applied to PJama, the principle of persistence by reachability means that an object is made persistent at a checkpoint if it is *peristence reachable*. An object is persistence reachable if it is reachable by the standard rules for object reachability in the Java language, unless it is reachable solely through chains of references that pass through variables marked by the "transient" modifier[2].

Continuing this definition, a reachable object, as specified by the *Java Language Specification* [GJSB00] (or *JLS*, as we will refer it from now), is any object that can be accessed in any continuing computation from any live thread. The question of what classes are reachable except those that are directly referenced as first-class objects by reachable objects, is more complex. We discuss it in detail in the next subsection. For now, we point out that if an instance is reachable, then at least its class, along with its superclasses and implemented interfaces, transitively, are also reachable.

According to another principle of orthogonal persistence, *type orthogonality*, all of the types defined in the

[2]In the Java language, the keyword `transient` has been reserved to denote transient fields in the context of Java Object Serialization [Sun00j]. We therefore have to use a different way to mark fields transient — a special API call. This issue is discussed in detail in [PAJ99].

JLS, in particular class `java.lang.Thread`, can be used in a persistent application and can persist. The consequence of persistent `Thread` instances is that, on resumption, active threads should continue from their state at the previous checkpoint.

From these definitions it follows that active suspended threads are what are called *roots of persistence* for the data promoted into the persistent store. However, it is possible that a persistent application terminates completely, i.e. all of its non-daemon threads exit, but it is still desirable to retain the accessible data in the store. This would require defining explicit roots of persistence. Also, in our high-performance implementation of PJama, providing persistent native threads has proved too complex without severely compromising performance. For these reasons, the mechanism of explicit persistent roots management is predominant at present.

In the course of time PJama has undergone several persistent root models. The present one can be viewed as two-level. On the level above the Java language itself, the programmer can define the behaviour of the Persistent Java VM using command line switches. The VM either automatically saves the *main* class into the store or not (the main class is the class that contains the `public static void main(String args[])` method, which was an entry point in this particular application execution). When a class is saved into the store, its static variables are saved as well (note that this is a feature making PJama different from most of the other persistent solutions for Java presently known to us). Therefore if the main class `M` is saved into the store, its static variables are roots of persistence. During the subsequent runs of applications over this store, whether they initiate from the same class `M` or not, their code can access static variables of `M` and retrieve persistent data from them.

This behaviour might be convenient in some cases and less convenient in others. Since the experience of the PJama developers group shows that the second variant is more frequent, the default behaviour of the PJama VM is not to promote the main class. Instead, the application developer is free to choose any object, and make it an explicit root of persistence. This is done using a very simple API contained in the standard PJama library class `org.opj.OPRuntime`. This class contains a public static variable called `roots`, of `java.lang.util.Set` type. By calling its `add(Object)` method, e.g. by executing

```
org.opj.OPRuntime.roots.add(obj);
```

somewhere in the text of the application, the programmer makes `obj` an explicit root of persistence.

Note, though, that this API doesn't actually contain a convenient mechanism for explicit retrieval of any particular root. This, however, is not required, because it is expected that the user will use class objects as persistent roots, e.g.

```
org.opj.OPRuntime.roots.add(M.class)
```

Once some class is made persistent, it will be automatically loaded from the store whenever it is requested by the running PJama application. Therefore if the above line was executed, and then the persistent application checkpointed successfully, this or another application running next time over the same store will get a persistent copy of class `M`, when it references this class. Again, the static variables of this class can be used to retrieve persistent data.

2.1.3.2 The PJama API

The API of the latest version of PJama$_1$ (that is, the latest line of PJama releases based on PEVM; the API in the PJama Classic was slightly different) is available through the special package `org.opj`, which contains a number of classes. The main implementation class for this API is called `org.opj.OPRuntime`. The publically available members of this class are presented in Figure 2.2.

```
public static final java.util.Set roots;

public static int checkpoint() throws org.opj.OPCheckpointException;
public static void suspend() throws org.opj.OPCheckpointException;
public static void halt(int rc);

public static void addRuntimeListener(org.opj.OPRuntimeListener listener);
public static void removeRuntimeListener(org.opj.OPRuntimeListener listener);
```

Figure 2.2: The essential PJama API.

The static variable `roots` allows programmers to explicitly introduce their own roots of persistence. The method `checkpoint()` causes the computation state to be preserved, and then the computation continues. The method `suspend()` causes the computation state to be saved for a future resumption, after which the execution of the JVM is terminated. The method `halt()` terminates the execution of the JVM without preserving the computation, so that resumption will be from the previously recorded state.

The methods dealing with runtime listeners allow application programmers to adopt the resumable programming style. A more complete description of the PJama API can be found in [AJ00], Chapter 5, and also in the PJama online documentation [PJa00]. In addition to the API oriented towards application programmers, there is also a small API which should normally be used only by advanced programmers. It allows to e.g. check if a certain class or object is persistent, get its *PID*, etc.

Making objects persistent is very simple with the present PJama API, as illustrated in Figure 2.3. Here a common scenario is presented, when a single class is used as a container for one or more persistent roots. The class is added to the set of roots by a single call in the static initialiser, after which its static fields serve as persistent roots.

```
import org.opj.OPRuntime;

public class PersistentData {
  static public java.util.Hashtable dictionary;

  static {
    OPRuntime.roots.add(PersistentData.class);
  }
}
```

Figure 2.3: An example of using PJama.

2.1.3.3 Class Promotion

We are going to return now to the discussion of class promotion strategy in PJama. But first a couple of remarks to give the reader a broader view of this question. In OODBMS, which are the class of systems functionally closest to PJama and other persistent languages, it is typically not classes themselves that are saved into the database. In a language such as C++, which is supported by most of OODBMS, compiled classes are something quite different from Java classes. A compiled C++ class is just binary code, basically a collection of subroutines representing class methods, which tells us practically nothing about, for example, the layout of objects of this class. So, instead, what is typically saved into the database is some kind of *schema definition* that can be either separately defined by the user or obtained by preprocessing the source code of classes. The actual details can vary significantly, though, and in some systems, e.g. GemStone/J [Gem98], the situation currently resembles PJama rather than more traditional OODBMS. Nevertheless, all of these systems consider classes as essentially descriptors of instance layout (inside a database) and placeholders for collections of methods (outside it). This is reflected, among other things, in the fact that none of the OODBMS known to us at present, save into the database static variables of classes.

In contrast, in PJama, as it was shown in the previous section, classes are also treated as first-class persistent objects. We believe preserving classes, including the code, along with persistent instances, is the most reliable way to preserve both *structural consistency* of the data (that is, correspondence between class definitions and real structure of their objects) and *behavioural consistency* (that is, formal correctness of the program in the sense that every method which is called is defined and every field which is read/written is defined). Preservation of these properties is, in turn, very important for large and long-lived applications, which may operate on large amounts of valuable data.

If classes were not promoted, then an accidental or malicious change to a class could introduce a mismatch between the real format of its persistent instances and its new definition, leading to program crash or data reading/writing errors. In Java it is also possible (since classes can be recompiled independently) that a method is deleted from one class, but is still called by another class. Therefore in case of careless code modification and compilation, it is quite easy to create a set of classes which are supposed to run together, but will really not (non-persistent Java applications can equally well suffer from this code inconsistency problem).

When a class becomes persistent, its static fields also become persistent. If this was not the case, then any state in the static fields, which is shared by all instances, would be lost; this would force the class implementation to change, breaking the concept of persistence independence.

The only time the static initialiser of a class is called is when the (not yet persistent) class is loaded from the file system. If a class becomes persistent, its static initialiser is *not* called when a class is fetched from the store. Otherwise, its static fields would potentially be re-initialised and their values could become inconsistent with the state of the persistent instances of that class. If some initialisation (e.g. a native library loading) must be performed every time a class is loaded from the store, this can be done by *Action Handlers* that PJama supports [JA98].

We have already mentioned that classes that are required to define the format of reachable instances, and that are referenced directly from persistent objects, are considered reachable. However, one other way for classes to become reachable exists: it is the reachability that is implicit in the class definition. We are talking here

about symbolic references to other classes in the field declarations, method signatures and method bodies of a class. The discussion of the issues related to implicit class reachability can be found in [AJ99]. Eventually the solution was adopted which is presented in [AJ00], and we quote here the two paragraphs from this work.

> A class `Class` instance is persistence reachable if it is directly referenced under the standard rules for object reachability, or is referenced by a `ClassLoader` instance that is itself persistence reachable, if at least one of its instances is persistence reachable, or if some array of its instances is persistence reachable. A recent clarification to the JLS states that the *bootstrap class loader* instance is always reachable. It follows that all classes loaded by the bootstrap class loader (bootstrap classes) are persistence reachable, as are all the objects that are persistence reachable from static variables in those classes.
>
> Any *resolved* (see JLS, Chapter 12) symbolic reference to a class `C` in the definition of some class `CC` will ensure that if `CC` is persistence reachable, then so is `C`. A symbolic reference exists from `CC` to `C` if `C` is the superclass of `CC`, or if `CC` implements `C`, if `C` is used as the type of a variable (static, instance or local) or as the type of the parameters or result of a method or constructor, or if `CC` uses a static variable of `C`. Note that this constraint does not restrict an implementation in its choice of eager or lazy resolution of symbolic references. However, once resolution has occurred, the binding must be maintained.

The implementation of JVM on which PJama is currently based (PEVM) resolves symbolic references in classes lazily. Therefore, of the implicitly reachable classes, only those that were actively used at least once are promoted. This makes practical sense, particularly with the contemporary releases of the Java Platform that contain roughly 4000 standard library (core) classes. The previous implementation of PJama (PJama Classic) used to resolve all symbolic references in classes transitively and eagerly on promotion. In some cases that could result in several thousand classes saved into the store unnecessarily — just because they were in the transitive closure of classes referenced by a single class really required by an application.

2.1.3.4 Representation of Classes in the Persistent Store

As mentioned in Section 2.1.2, in the PJama Classic implementation classes were saved in the store in the format close to their main memory representation. Quite "hairy" main-memory representation of a Java class, that consists of a large number of variable-size structures, was "flattened" when saving class into the store, and then "unflattened" when loading the class into main memory next time. This design, originally suggested to improve class loading time (no usual Java class file parsing and verification is required, some optimisations that the interpreter makes to bytecodes are preserved), eventually resulted in a number of very serious drawbacks. First of all, whenever something would change in the internal representation of class in the new release of the JVM, it had to be reflected in our code and in our disk format of classes, making stores created with older versions of PJama unusable. Then, the C code (often quite tricky) that dealt exclusively with flattening and unflattening classes (i.e. swizzling and unswizzling pointers inside class objects) had been, along with the memory class format defined by the JVM vendor (Sun), gradually becoming more complex from one JVM release to another. Its size has finally grown to the scary figure of more than 3000 lines, and it became very difficult to manage. At the same time, the overall performance of the system

compared to the one where classes are saved in their usual `.class` file format, has decreased, rather than increased, as our measurements have eventually shown.

All references from a persistent class object to other classes (as well as all other objects, e.g. constants defined in the class's *constant pool* (see Section 3.3.2) were via *PIDs*. This was an additional burden for evolution, since patching these *PIDs*, which were not easy to locate inside a class object, was required if e.g. a change to class hierarchy was made. The whole idea of saving classes in such a format has proved totally inadequate in the end.

These problems were realised in PJama$_1$ for which a portable representation of class objects in the store was adopted. "Disk class object" now contains a reference to a persistent array of bytes with the corresponding class file contents, plus a small number of additional fields, e.g.

- transient fields map (information about fields that are transient in the PJama sense is not contained in the class file — rather, it has to be provided by the programmer with the help of explicit API calls and specially preserved by the persistent platform);
- pointer to the class loader instance for this class;
- pointers to arrays that contain, respectively, values of primitive and class type static fields of the class.

There are also several other fields, that are essentially redundant (i.e. their values can be obtained after parsing the class file), and exist mainly for convenience.

In such a format, all references from a persistent class on disk to other classes are strictly symbolic, which simplifies evolution a lot, since we don't have to patch the pointers (*PIDs*) to evolving classes in non-evolving persistent classes during evolution. It is also easy to convert a store with classes in such a format from one hardware architecture to another (i.e. high-endian to low-endian and vice versa), since the locations of all numeric values in which bytes should be swapped during such a conversion, are well-known.

Lookup of persistent classes in the store is performed solely on the basis on their names, via the *Persistent Class Directory* (PCD) — a persistent data structure that maps combinations of class names and class loaders to the corresponding class objects.

2.1.4 Summary

PJama is the only system presently available and known to us that preserves class objects (including static variables and methods) in the store along with their instances. Moreover, more classes than just those that are required to define the format of persistent instances, are typically preserved. This is due to the model of class promotion, which is designed with the intention to preserve the behavioural consistency of a persistent application along with the structural consistency, thus making future computations consistent with the past. Classes are saved in the store in a portable format, which essentially consists of a class file byte array, a block of static variables, and some auxiliary fields. All references from a class to classes inside the store, are symbolic, which aids store-level support for evolution.

2.2 Requirements for the PJama Evolution Technology

In the previous section we have demonstrated the reasons why class objects are made persistent in PJama, and why typically more classes than just those that have persistent instances, are promoted into the store. Since this mechanism exists precisely to prevent any (dangerous) changes to persistent classes and instances, the application developer needs special *evolution technology* that allows required (safe) changes to be performed. The main objective of this technology is to aid and facilitate safe changes. More precisely, we can formulate its responsibilities as follows:

1. Verify that the new set of classes, i.e. unchanged classes plus new versions of changed classes, plus any newly created classes, are mutually consistent. The criterion for this formal consistency that we adopt, is that all the classes in the new set are *source compatible*, that is, the Java source code of all these classes can be compiled together without problems.
2. For each changed class, check if the format of instances that it defines became different. If so, perform *conversion* of instances of this class, so that eventually all persistent instances agree with their classes.
3. If both of the previous tasks succeeded, replace classes and instances in the persistent store atomically and permanently.

This is not the only possible variant of requirements for the evolution technology. Another approach is called *versioning*. This term corresponds to a broad range of techniques, for which the common feature is long-term coexistence of multiple versions of either individual classes, or collections of classes (*schemas*), and/or instances in the formats corresponding to these different versions (see Section 2.6.1).

In PJama, however, we choose simpler requirements for the evolution technology, which we described above. This simplicity, first of all, provides greater robustness for the whole system. This is an important factor — our experience has shown that even the support for "simple" replacement evolution interferes significantly with many other parts of the PJama system. We have also heard that implementing versioning technology in other systems was sometimes a tremendous effort, with eventual very low end-user demand for this technology, e.g. as it happened in the O_2 project (this was discussed at O_2 Presentation at PASTEL meeting in 1998).

Another consideration is that PJama at present does not support multiple applications running concurrently over the same store. Hence so far we simply can't experience most of the problems that might have provoked development of versioning techniques in other systems. Last but not least, due to its present age, available resources and the user base, the PJama evolution technology is oriented more towards development-time evolution, rather than deployment-time [ADHP00]. Again, this makes the demand for versioning technology negligible at present.

2.3 Main Features of the PJama Evolution Technology

In the previous section we have formulated the requirements for the evolution technology in a relatively abstract fashion. We will now show how they are implemented.

The front end of our evolution technology is a standalone utility, a command-line tool which the application developer invokes over the given store in order to evolve classes and convert instances. In addition to this tool, there is a small additional API and functionality contained in one PJama library class (`org.opj.utilities.PJEvolution`), and some programming API/methodology that can be followed by the programmer and recognised by the evolution system. This API is used only when persistent object conversion is required. To simply replace classes when the format of their instances doesn't change, it is enough to use just the evolution tool.

The front end of the PJama evolution technology has itself undergone intensive evolution over several successive releases. Its initial and present shapes differ significantly. To facilitate the reader's understanding of the motivation behind the present design, and to answer the questions that may arise regarding other possible design alternatives, we will now describe the history of our successive evolution tools.

2.3.1 The History of Evolution Tools For PJama

The initial version of the PJama evolution tool was called `opjsubst` and was purely a change validation and class replacement tool. This and other PJama tools are command-line utilities, in line with other utilities coming with the implementation of the Java platform on which PJama is currently based — Sun's JDK. In order to evolve classes using `opjsubst`, the application developer had to do the following:

1. Modify the `.java` sources for the classes.
2. Recompile them using the standard `javac` compiler.
3. Run the tool over the given store, passing the classes to it, e.g.
   `opjsubst -store mystore C1 C2`

The tool then verified if the new versions of classes are valid substitutes for the old ones, and if they were, replaced the classes in the store. It also converted persistent instances in case their format had changed (see Chapter 4). If any problems were detected during verification or conversion, the tool would leave the store unchanged.

The problem with this evolution tool was that the user had to identify correctly which classes were modified, manually recompile them, and then pass them to the tool. If there are more than, say, five changed classes, the above procedure was likely to become quite inconvenient and error prone. Therefore we soon developed another tool, which was called `opjc` and was a hybrid of the `javac` compiler and `opjsubst`. The tool would take the `.java` sources for the modified classes, recompile them and then do all the things that `opjsubst` did for those classes that had persistent counterparts. Thus, the application developer had to make fewer steps and had less information to keep in mind. An additional function of `opjc` was compilation of *conversion classes* (see Chapter 4) — these classes make use of some special Java language extensions and therefore can not be compiled by the ordinary `javac` compiler.

However, both tools had a common disadvantage, which we will explain with a simple example. Suppose we initially have two persistent classes `A` and `B` presented in Figure 2.4.

```
class A {                    class B {
  ...                          void m() {
  B.m();                       ...
  ...                          }
                               ...
}                            }
```

Figure 2.4: An example of two persistent classes, one using another.

Suppose then that we delete method `m()` from `B`. In the JLS terminology, this change is both *binary incompatible* (because bytecodes of recompiled class `B` will not work correctly with class `A` anymore), and *source incompatible* (because the source files of classes `A` and `B` can not be compiled together anymore. Section 3.3.1 tells more about source incompatibility). If, however, we invoke

```
>javac B.java
>opjsubst -store mystore B
```
or just
```
>opjc -store mystore B.java
```

then in either case, class `B` will be successfully recompiled and substituted, whereas class `A` will remain unchanged. The next run of the code of persistent class `A` will therefore crash with "method not found" exception.

This problem is not unique to PJama evolution, and its source is the Java model of compilation and linking, in which classes that eventually constitute a single application can be compiled and evolved separately. The fact that there is no need to link the entire application or bother about interdependencies between classes (as long as all changes to classes are compatible) is a great advantage of this model. Since incompatible changes are usually relatively rare, and since ordinary makefiles don't help much in this case, as discussed in Section 3.1, developers often ignore the problems associated with such changes. Typically, if they make an incompatible change (and notice that, which is not always the case), they would just update the dependent classes they manage to identify and recompile them, or they would recompile the whole application. However, we consider that in case of persistent applications, which operate on large and long-lived volumes of data, it is too dangerous to rely on the developer's memory (and also on whether or not they understand that they make an incompatible change), since the cost of class incompatibility problems, when they propagate into the store during evolution and possibly show up much later, can be much greater than in conventional (transient) applications. On the other hand, repeated recompilation of the entire application can be too expensive. Therefore our next step was to investigate how to make class evolution safe in the sense that no incompatible class changes could propagate into the store.

After considering some alternatives we finally chose to combine evolution with selective (or, as it also called, *smart*) recompilation of classes (see Section 3.1 for detailed discussion). We called this *persistent build technology* and developed a tool which controls both class recompilation and evolution. The main features of this tool, called `opjb`, are discussed in the next section.

2.3.2 The Front End — the `opjb` Persistent Build Tool

When `opjb` is first invoked over the given store, it creates a persistent table which contains information on every evolvable class, its class file located on the CLASSPATH and its `.java` source file. This table, called *Evolvable Class Directory* (ECD) is then used to keep track of these classes and determine Java sources and/or class files that have changed between the invocations of the tool.

On the first invocation of the tool the developer has to provide it with enough information to locate all of the `.java` and `.class` files for the persistent classes. In the simplest case, when the tool is the first application invoked over an empty persistent store, it may look just like

```
>opjb -store mystore mypackage/*.java
```

The detailed explanation for more complex cases is presented in Section 3.2.2. On the subsequent runs the presence of the class management table in the store minimises the amount of information that should be repeatedly passed to the tool. In most cases, the developer simply modifies the `.java` sources for the necessary classes and invokes `opjb`, e.g:

```
>opjb -store mystore
```

To add a class to the ECD (if this class is not yet persistent) the developer can invoke either of

```
>opjb -store mystore MyClass
>opjb -store mystore MyClass.java
```

The tool checks whether any classes on the file system have changed compared to their counterparts in the persistent store. Classes on the file system exist both in the source and in the bytecode form. So the tool first recompiles the modified `.java` files (modification is recognised by comparing the timestamp and the footprint with the previously recorded values preserved in the store), then compares the changed `.class` files with their older persistent versions. This comparison (or, as we call it, *change validation*) may reveal more classes that need to be recompiled. We say that these classes are *potentially affected* by incompatible changes. An example of such a change was presented in the previous section. Potentially affected classes then undergo the same recompilation and validation procedure. This process, described in detail in section 3.3, continues transitively until there are no more classes to recompile or verify. After that, all classes that have changed are atomically replaced, and the ECD is updated.

Though in most cases the interaction between the developer and the tool is minimal, as above, sometimes more information should be specified. One such case is when the programmer needs to customise how the values of static variables are copied between the old and the new versions of an evolving class. The command line options of `opjb` that control that are described in Appendix A. Other cases are several operations supported by `opjb` that work on the level of class hierarchy rather than individual classes. They are discussed in the next section. Finally, if *object conversion* (see chapter 4) is to be performed, additional information may be required by the tool, as discussed in section 2.3.4.

2.3.3 Support for Operations on the Class Hierarchy

Since PJama does not maintain an explicit database schema, changes to the class hierarchy such as adding a subclass, insertion of a class in the middle of the hierarchy or moving class around the hierarchy, happen implicitly, when the developer modifies the sources for the involved classes. For example, if class `E` is to be inserted in the middle of the class hierarchy (this operation is denoted "Evolution 1" in Figure 2.5), the developer simply has to change the source code for classes `D1` and `D2`, replacing the "`extends C`" statement with "`extends E`", and then to run the persistent build tool. Class `E` will be automatically promoted into the store. Thus the developer essentially does not have to bother most of the time, whether or not the changes they make affect the class hierarchy.

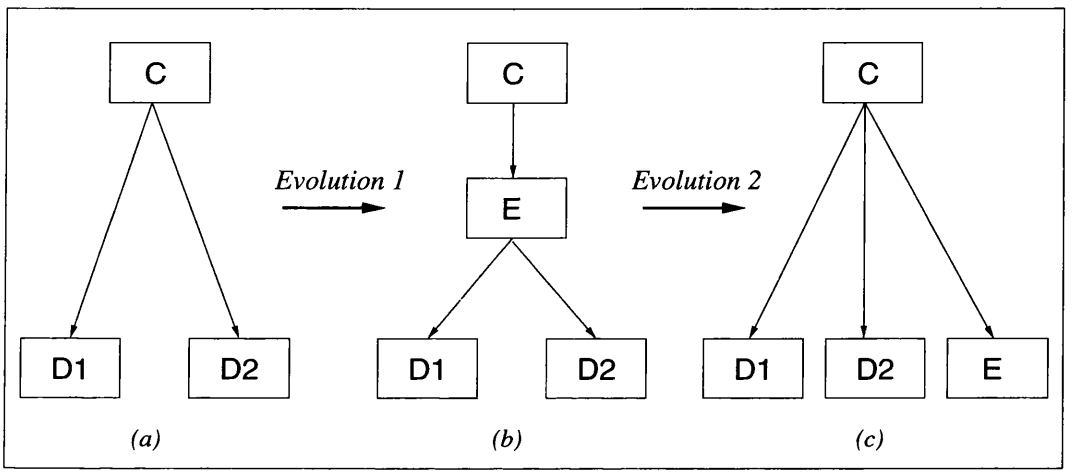

Figure 2.5: Class Hierarchy Evolution Example

However, if during the subsequent "Evolution 2" the developer changes the definitions of classes `D1` and `D2` back to the original form, class `E` will not be removed from the class hierarchy. The latter will be just re-shaped, as shown in Figure 2.5(c). In fact, that might be the developer's intention. But if the developer wishes to remove class `E` completely, and, consequently, to *migrate* its instances, if they exist, to another class, then they need to explicitly specify the deletion operation, e.g.

```
>opjb -store mystore -del E
```

Needless to say, the definitions of former subclasses of `E` should also be updated accordingly. The fact that no classes reference class `E` anymore will be verified by the system. It will then check whether any persistent instances of `E` exist. If they do, then the programmer should also specify the *default migration class* or give the system a conversion class that will deal with instances of class `E` (see section 4.5.2 and Appendix A for more details).

Another class hierarchy level operation supported by PJama evolution system is *class replacement.* This can be viewed as class modification combined with a name change for this class, whereas its identity, which in this case is binding between its instances and the class object, remains unchanged. Bindings between persistent classes in the store are purely symbolic in PJama, as was explained in section 2.1.3.4, so the developer has to take care of preserving them manually, by changing the `.java` sources of the respective classes. To replace class `C` with class `D`, the developer should simply create the source code for class `D` and

run the build tool with a special option, e.g.

```
>opjb -store mystore D.java -rep C D
```

The `.java` sources of classes that previously referenced `C` should be updated accordingly. The system will verify that no class in the store now references the name `C`, promote class `D` into the store, re-bind the existing instances of `C` to class `D`, and, finally, delete class `C` from the Persistent Class Directory.

2.3.4 Support for Object Conversion

If any classes are changed such that their instances need conversion, the user has a choice between relying on the default (automatic) conversion, or writing their own conversion code in Java (see Chapter 4 for details). In the latter case, one or more classes containing conversion code should be passed to the persistent build tool in addition to the evolved classes, for example:

```
>opjb -store mystore MyConvClass.java -convclass mypackage.MyConvClass
```

The `mypackage.MyConvClass` class is specified here twice, because this class is not a part of a persistent application. Therefore we have to explicitly specify the source file `MyConvClass.java` to make the tool compile it, and we also specify that class `mypackage.MyConvClass` is a conversion class.

If compilation and change validation proceed without problems, conversion of instances starts immediately. In other words, only *eager* conversion is implemented at present. See Section 2.5.2 for discussion of why lazy conversion is currently not supported.

2.4 PJama Evolution System Architecture Overview

The architecture of the PJama evolution system is outlined in Figure 2.6.

The system consists of three software layers. These layers are isolated from each other, communicating via clean programming interfaces. Layer isolation allows us to replace the code corresponding to each layer independently, with little or no disturbance to the other parts of the code. Such a replacement will be required, for example, if PJama migrates to a different JVM and/or persistent store. In fact, such a transition has happened already once, when our evolution technology was ported from the PJama Classic implementation based on Sun's Reference Edition JVM to Sun's Solaris Research VM plus Sphere store (see Section 2.1.2). During this transition, the lower software layers were changed quite significantly, but the upper layer remained practically unchanged [LMG00].

The code in the upper layer implements the high-level functionality: interaction with the user, management of the ECD (see Section 3.2.1), change validation and recompilation of changed classes, and, in part, class replacement in store and instance conversion. For class compilation we use the standard `javac` Java compiler, which is written in Java itself. We have made a number of patches to it to enable its integration with

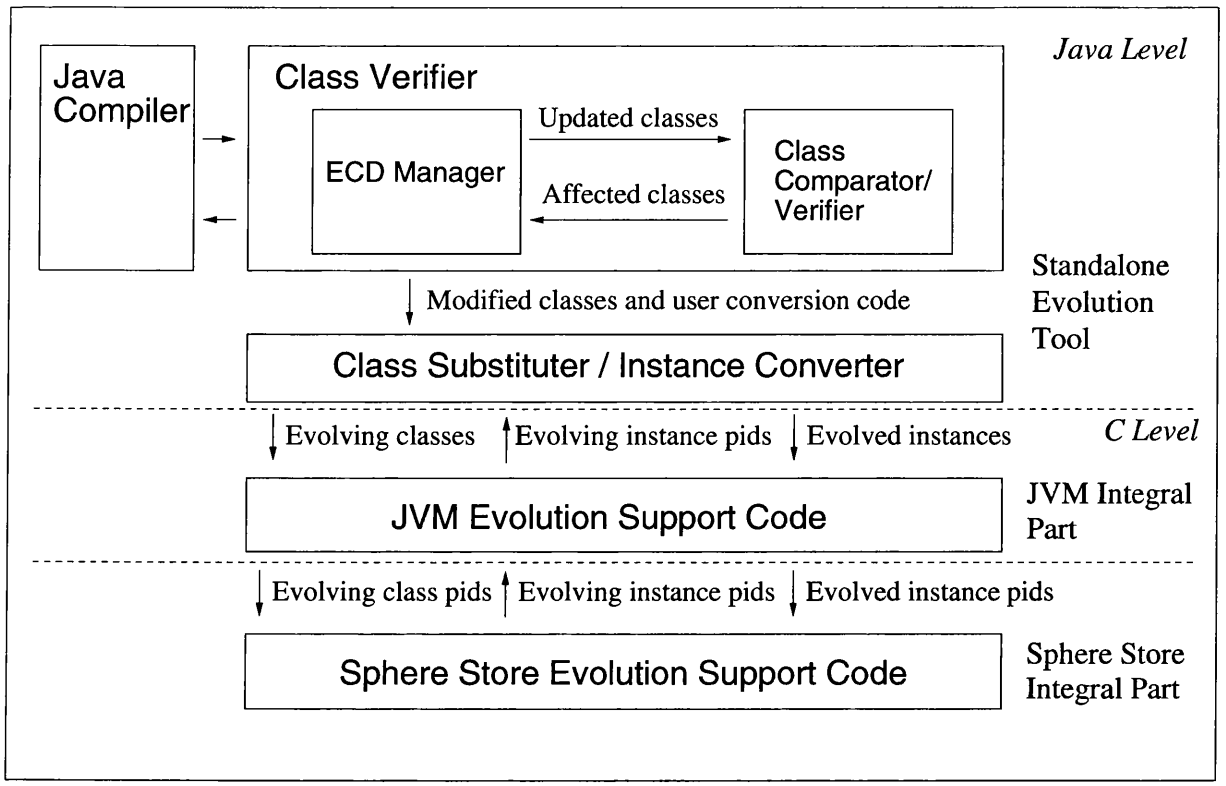

Figure 2.6: PJama Evolution System Architecture

our system and to make it "understand" the special syntax which is used in conversion classes (see Chapter 4).

The upper layer is written in Java, whereas the two other layers are written in C. Implementation in Java has a number of merits. It makes the code portable and gives it access to the large number of powerful facilities supported by the Java core classes, e.g. class reflection API or the classes implementing such data structures as sets or hashtables. The Java language also allows us to easily create and manipulate complex data structures of arbitrary size, which are used during class validation and recompilation. In general, we were trying to implement as much of the system as possible in Java.

The code of the middle software layer can be divided into two parts. The first part includes a number of miscellaneous utility functions supporting operations required by the upper layer, that can't be implemented in Java or found in the Java core classes, e.g. advanced reflection facilities. The second part of this code works as a bridge between the upper layer and the store layer. It also includes some functions that have to be implemented in C to provide sufficient execution speed, e.g. the support for default instance conversion.

The lowest software layer supports the store-specific operations, the most important of which are instance lookup, low-level support for instance conversion and recovery. It is a part of the Sphere persistent store. The detailed technical description of this layer can be found in [HAD99], whereas in this thesis we provide a high-level overview of its design and present some performance measurements (see Chapter 6).

2.5 The Present Constraints

2.5.1 No On-Line or Concurrent Evolution

The PJama evolution technology is currently off-line only, which means that when evolution runs, it should be the only activity on the given persistent store. The immediate reason for it is that PJama does not support multiple concurrent transactions on the same persistent store. Also, a reliable, industrial-strength implementation of concurrent evolution seems to be a complex problem, with some hard choices on both conceptual and technical levels. This may be a subject of future research.

2.5.2 No Lazy Object Conversion

At present we support only eager (immediate) conversion of persistent objects. *Lazy* or *deferred* conversion, where objects are converted at the fault-in time, when a usual application running over the given store requests them, is an alternative. Lazy conversion can be beneficial for some applications, first of all those where any significant downtime required to convert all of the evolving objects is undesirable. However, there are also a number of specific problems associated with lazy conversion. It looks technically difficult to implement it such that it does not compromise the performance of the system in the normal, non-evolution mode, and also to avoid imposing too much design complexity on many parts of the system that are not directly related to evolution support. The most fundamental problem, however, arises if we think about combining lazy conversion with user-defined conversion code.

This problem is described in, e.g. [FFM+95]. The authors call it a problem of *combined complex and lazy modification.* Complex modification is an evolutionary modification to an instance which involves accessing data not only of that instance, but of some others as well. The problems in this case can be caused by several reasons, and here is the simplest example. Suppose we have a conversion function for instances of class `C` that involves reading field `f` of an instance `d` of another class called `D`. At some point in time we have a persistent store with some of the instances of `C` converted and some not. Suppose then that a modification to `D` is introduced, such that field `f` is deleted. Nothing harmful would happen if conversion was performed eagerly for all instances of `C` and then for all instances of `D`. However, in the situation which we are describing, for some particular instance of `C` it can happen that `d` is converted *before* conversion of the corresponding instance of `C` is attempted. Therefore the latter conversion, when it is eventually initiated, will find `d` in already changed, unsuitable format.

In [FFM+95], where this problem was first presented, a mechanism of so-called *shielding* was suggested in order to eliminate it. It basically means retaining (invisibly for the programmer) the data in the old format as well as in the new one for every converted instance. A similar approach, also known as *screening* (which looks most consistent when it is combined with *schema versioning* or *object versioning*, that preserves and allows to access various versions of objects) was implemented in a number of systems, e.g. CLOSQL [MS92, MS93, Mon93] and ORION [BKKK87, Ban87, KC88].

We consider this approach too expensive and therefore impractical, since it means that a large amount of information (fields deleted in the course of evolution or even all of the old versions of each evolving instance) has to be kept in the database forever. Several alternatives to shielding (apart from relying on the

user's ability to recognise and avoid the above problems) can be suggested:

1. Simply rule out custom lazy conversion, allowing only default conversion to be done lazily. Default conversion does not set fields of any objects other than the current evolving one. It is not clear, though, how useful such a truncated lazy conversion functionality would be.
2. Allow custom lazy conversion, but prohibit reading/writing fields of objects other than the current one in conversion methods. This can be done by e.g. analysing conversion code and rejecting it if it attempts to access any fields or call any non-static methods of objects other than the current one.
3. More sophisticated mechanism that includes code analysis may, in the case of field or method deletion, use preserved information about previous evolutions and determine whether there are any incomplete lazy modifications which need to access these fields or methods. It can then inform the user about the problem, and, for example, eagerly complete the previous lazy modifications.

Implementation of lazy conversion for PJama might be a good topic for a separate research project in the future. Lazy conversion in several commercial systems is discussed in Section 4.9. In these systems, this kind of conversion is restricted to avoid the complex form, which confirms our concerns.

2.5.3 No Support for Java Core Classes Evolution

The evolution technology for PJama currently does not support evolution of *Java core classes*, in other words, evolving persistent stores across Java SDK releases. Java core classes are a set of standard library package classes delivered with every Java implemenation/release, for example all classes for which fully qualified name starts with "`java.`".

There are two problems with these classes. First, some of them, such as `java.lang.Object`, `java.lang.Throwable` or `java.lang.ClassLoader`, are intimately connected with the Java VM. The VM loads them in a special way, reads/writes their internal variables, maintains certain constraints associated with them (e.g. it is not allowed to load a second copy of class `Throwable`, no matter with what class loader), etc. Therefore working with two copies of these classes — which the VM has to do during evolution — will require a number of patches to the VM code, which are likely to conflict with internal invariants and security mechanisms.

However, another, much more serious problem, is that changes to these classes are beyond our control, and their amount between successive versions of the Java SDK is high and does not seem to diminish in the latest releases. For example, our investigation for the relatively recent transition from Sun's SDK version 1.2 to version 1.2.2, which is considered a "minor upgrade", has shown the figures summarised in Table 2.1 (more complete historical data can be found in table 9.1 of [AJ00]):

If we wanted to support evolution of core classes and user classes that extend them in this particular case, we would have to analyse what has changed, at least in those 338 classes for which the format of instances has changed, and possibly write conversion code for them. Ideally, to avoid problems, we would also have to inspect all other changed classes. This is not likely to be an easy task, at least for a limited-size research

Total number of old classes	4273
Total number of new classes	4337
Classes removed in the newer version	113
Classes added in the newer version	177
Classes changed	1256
Of those, changed minimally (code only)	695
Classes for which conversion of instances is required	338

Table 2.1: Changes to Java core classes between JDK1.2 and JDK1.2.2

group. Instead, the ideal situation would be if the programmer who modifies a class also determines how to evolve it — but that would be possible only in case of very close co-operation between the Java vendor and the PJama group, which is currently not the case. Therefore, our evolution technology at present does not support evolution of Java core classes. It is always assumed that those of them which are persistent are the same as their counterparts on the CLASSPATH.

2.6 Related Work — OODB Systems

Systems most close to PJama and other language-specific persistent platforms are OODBMS, and most of the research in management of change to persistent object schema and data has been done in the context of these systems. The number of such systems is large, and models of their support for change management vary tremendously. Below we discuss a number of aspects of change management technology, which we consider the most important, and illustrate them on concrete examples.

2.6.1 General Approaches to Change Management

A number of approaches to modifying the conceptual structure of an object database have been developed historically, which can be broadly categorised as follows [Ras00]:

- **Schema evolution**, where the database has one logical schema to which modifications of class definitions and class hierarchy are applied. Instances are converted (eagerly or lazily, but once and forever) to conform to the latest schema. Examples of systems that follow this approach are Orion[BKKK87, Ban87], OTGen [LH90], O_2 [FMZ94, FFM+95], GemStone/J [Gem98, Gem00], Objectivity/DB [Obj99a, Obj99b, Obj99d, Obj99c], Versant [Ver00]. PJama also employs this technique.
- **Class versioning**, which allows multiple versions of each class to co-exist. Instances can always be represented as if they belong to a specific version of their class, but how this is done (e.g. by creating a separate image of instance for each class version or by keeping one version-specific copy of the instance and dynamically converting it every time it is accessed using a different class version) depends on the concrete system. Examples of systems implementing class versioning are CLOSQL [MS92, MS93, Mon93] and POET [POE00].

- **Schema versioning**, which allows several versions of one logical schema to co-exist simultaneously. Similar to the previous approach, instances can be represented in multiple ways to confirm to a specific schema version. Schema versioning has been explored in e.g. the works of Odberg [Odb94b, Odb94a, Odb95], it was once implemented for the O_2 system [FL96].

It is possible for one system to employ more than one change management technique. For example, both schema evolution and schema versioning were implemented for O_2 [FL96].

The schema evolution approach provides the developers (maintainers) with a coherent and comprehensible view of the database, and it is presently followed by most of the well-established contemporary commercial systems[3]. Systems that implement schema evolution typically provide a facility for default conversion, and most of them also provide some kind of custom conversion, that allows the developer to specify how data should be transformed (see Section 4.9). Conversion can be performed either eagerly only (e.g. in GemStone/J or PJama), or both eager and lazy conversion is allowed (e.g. in O_2 and Objectivity/DB).

What sometimes is considered a drawback of schema evolution approach is that information about history of changes is lost, and thus a change, once performed successfully, can not be reversed. More serious problems may be experienced if one database serves a number of applications. In this case, it might be difficult to update all of these applications simultaneously with the database, so that they incorporate the changed class definitions.

Class versioning approaches, on the other hand, maintain information about all versions of each class. Thus the existing persistent objects may not be changed to reflect schema changes. To represent an object in a way conforming to the "current" class definition that is used when accessing it, default conversion (as in POET), error handlers or update/backdate methods (as in CLOSQL) can be used. However, as the number of versions for each class grows, the schema tends to become quite complex, making maintenance difficult. It becomes hard to obtain a coherent view of the database in presence of a large number of class versions. Furthermore, it seems unlikely that any non-trivial transformations (e.g. those that need to access objects outside the current one) can be represented by update/backdate methods — for instance, because other objects can also belong to versioned classes.

Schema versioning may somewhat alleviate the shortcomings of schema evolution and class versioning by providing a coherent view of the conceptual structure of the database, while at the same time preserving the history of changes and multiple views of data. It looks like a most reliable technique for maintaining forward and backward compatibility with existing applications. However, the historical information is not available at a finer granularity, as in class versioning. Then, schema versioning can be very expensive in terms of space usage, especially in the presence of a large number of versions. In some implementations, even minor class changes may lead to creation of a new schema version, which is a significant overhead.

To summarise, none of the existing general approaches to change management satisfies all of the (conflicting) requirements: coherency of the database view, automatic compatibility with the existing applications, and small space overhead. The schema evolution approach, however, is at least easier to understand, design and implement, and the most economic in terms of disk space. Therefore it is supported by most of the contemporary commercial systems.

[3]It's worth noting that it is the only available approach in RDBMS, and it has proved commercially and practically successful.

2.6.2 Types of Changes Allowed

An object database schema consists essentially of a hierarchy of classes, each of which defines members. Therefore, schema modifications can be broadly divided into modifications to class hierarchy and internal changes to a class, such as adding and removing fields or methods, or redefining them. Changes in each of the above categories (class hierarchy or internal class structure) may or may not lead to changes in the structure of class instances.

All of the systems that we have studied support arbitrary changes to class structure, i.e. allow to add, delete or modify data fields (attributes). Most of these systems, but not all of them, also support arbitrary changes to class hierarchy. However, some systems, e.g. O_2 and POET, do not allow developers to add or delete classes to/from the middle of the class hierarchy (such classes are called *non-leaf*), and CLOSQL did not allow any changes to class hierarchy. Some other constraints are possible, i.e. in POET affected objects in a database should be necessarily eagerly converted once a non-leaf class is added or deleted, whereas other types of changes can be handled by lazy object transformations.

More information on various evolution models and supported changes can be found in the exhaustive survey of evolution facilities in OODBMS in Rashid's PhD thesis [Ras00]. Object conversion support in a number of systems is discussed in Section 4.9

2.6.3 Data and Code Consistency Preservation

[Sjø93a] categorises the effects of changes to the conceptual structure of a database into:

- Effects on other parts of the conceptual structure.
- Effects on existing data.
- Effect on application programs.

Most of the existing work on evolution in OODBMS is focusing on the first and second category of effects, i.e. on preserving of what is called structural consistency of the database. Very little has been done in the latter area, i.e. on the issues of preserving behavioural consistency of applications (see the discussion on how this it is supported in PJama in sections 2.1.3.3 and 2.3.1). As for other systems, in [ABDS94], for example, unsafe statements are detected at compile-time and checked at run-time by a specific clause automatically inserted around unsafe statements. The exception handling code is provided by the programmer. Therefore behavioural inconsistency is not eliminated, but at least its dangerous effects are somewhat contained. In [Ala97, Ala99] reflective techniques are employed to overcome the type system inconsistencies. Both of these approaches seem to cure the consequences and not the source of the problem. Therefore, as pointed out in [Ras00], "PJama is the only system offering support for behavioural consistency and partial support for language type system consistency; the object migration features are type unsafe". The latter statement is only partially true, more details on this issue are contained in Section 4.5.2.

2.6.4 Support for Evolution in Java Binding

Since the primary focus of this thesis is evolution of Java applications, we were interested in whether the Java binding, available in practically all of the contemporary OODBMS systems, provides equally good support for evolution as other bindings (typically C++). It turns out that in all systems Java is supported considerably worse than C++[4]. In the ODMG 2.0 standard [Cat97], which is a standard for object-oriented databases, and which includes the specification for Java binding, schema access/change facilities using the latter are not specified at all[5]. Thus an implementation that follows the standard would not allow the Java applications to even traverse the database schema, let alone change it. Those systems which allow schema traversal (reflection) or evolution in Java do it in a non-standard, vendor-specific way.

Systems presently known to us that allow schema traversal and change through a Java API are GemStone/J and Versant. In both of them, however, conversion facilities are rather limited (see Section 4.9). The only system presently known to us which has more advanced conversion facilities, i.e. can run programmer-defined conversion code, and supports Java binding — Objectivity/DB — does not allow conversion code to be written in Java. For additional information on evolution support through different language bindings in several contemporary commercial systems, the reader is referred to [RS99].

2.7 Summary

In this chapter, we have presented the model of evolution support in PJama. We formulated the requirements that we wanted this model to satisfy, in which the focus is on aiding and facilitating safe and consistent changes. In our model, new definitions of classes, usually in the form of source code, are supplied by the developer to the standalone utility, an evolution tool called `opjb`. The tool compiles the given classes, often recompiling other classes as well to ensure consistency (compatibility) of changes, then validates the changes, and finally evolves the persistent store atomically and durably. Since multiple concurrent applications can not operate over a single PJama store at present, we did not consider the issues of concurrent or on-line evolution. We then discussed the main features of the PJama evolution technology in detail. First we described the history of evolution tools that we developed and thus showed how we came to the present technology, which combines build management and evolution. We then described how we support some operations on class hierarchy, for which just new definitions of changed classes may not be sufficient. Finally, we mentioned support for object conversion, which is discussed in detail in Chapter 4.

We then explained why our technology presently has some constraints: lack of on-line or concurrent evolution, lack of lazy object conversion and lack of support for evolution of Java core classes.

The survey of the related work has shown that the approach taken by PJama (schema evolution in favour of class or schema versioning) is at least the easiest to understand by the user, and relatively easy to design and implement. It is also the most economic in terms of database space. These are probably the reasons for most of the contemporary commercial systems to support this approach. We then showed that PJama's evolution support is unique among other systems in two respects:

[4]Of course, this may be a temporary effect, because the C++ systems are more mature.

[5]For C++ no schema modification API is specified as well — only schema traversal API.

- PJama seems to be the only system that preserves behavioural consistency (performs safety checks on the code, not just the data).
- It is also the only system providing advanced conversion facilities (custom conversion code) in Java.

Chapter 3

Persistent Build Technology

In this chapter we describe our persistent build technology, which was first briefly presented in Section 2.3.2. Section 3.1 discusses *build management* in a broader context and explains why this technology for PJama took its present form. Section 3.2 describes the details of management of evolvable classes. Section 3.3 discusses the algorithm which is used in the persistent build procedure, and presents a comprehensive table of source incompatible changes. The last two sections describe related work and outline possible future work directions.

3.1 Motivation and Context

The PJama persistent build technology is a combination of smart recompilation of classes and evolution. The initially pure evolution technology was given an ability to recompile classes in order to make evolution safer by preventing propagation of incompatible changes to classes into the store, as was explained in Section 2.3.1. We will now discuss this issue in more detail.

Initially we didn't consider combining evolution and recompilation. Our approach was that these are two separate activities, and though evolution technology should guarantee safety, it should not deal with class recompilation. Rather, it should just be able to find and report the problems with class incompatibilities.

However, one problem with this approach is its technical complexity. In case of some simpler changes, such as method deletion, it is easy to analyse the `.class` file for class `A` and determine if it still calls method `B.m()`. However, for some other types of incompatible changes, e.g. when a particular interface `I` is deleted from a list of interfaces implemented by class `C`, it is quite difficult to determine whether any class referencing `I` is now incompatible with class `C`, i.e. is it, for example, trying to cast an instance of `C` or of one of its subclasses to type `I`. That would require development of sophisticated class bytecode inspection procedures. The complexity of this problem is illustrated by the fact that many of the incompatible changes, such as the above, are not captured by the bytecode verifier (at least of the Sun's Solaris Research VM, which is our current Java platform). Instead of this (more desirable) behaviour, in that system at least, the problems caused by such changes are revealed only at run time.

In addition, a couple of known types of source incompatible changes are binary compatible and simply can not be detected by bytecode analysis (see the tables in Section 3.3.1).

Class recompilation, on the other hand, re-uses the existing compiler, which is itself the best code consistency verifier. And, what turns out to be equally important, automatic or smart recompilation for Java is a very useful feature by itself. It is relevant to any large and complex Java application. To justify this statement, we will first discuss the issues with general application *build management.*

3.1.1 General Build Management

Build management is essentially a process of creating the executable code of an application out of the source code. The source code can be spread among many modules, that are typically first compiled into an intermediate form (e.g. object-code files) and then linked into the final application. In addition, these modules can be written in different languages, operations involved in application assembling may not be just compilation and linking, but, for example, source code generation, obtaining updated sources from a remote location in case the current sources are found obsolete, binary code surgery, etc. The two main issues in build management are therefore the following:

1. Optimising the number of recompilations. All modules that have changed should be recompiled, but, on on the other hand, unnecessary recompilations of the modules whose sources have not logically changed, should be avoided.
2. Making management of multiple diverse activities that can be involved during application build, more convenient.

Most of the aspects of the second issue is beyond the scope of this thesis, and the rest of this section discusses the techniques for optimising the number of recompilations.

3.1.1.1 Make

The classic tool to help rebuild applications after change is *make* [Fel79]. In order to determine dependencies between programming modules, their code should be investigated. Typically, the programmers derive the dependencies and specify them in a *makefile* manually. This information, together with some implicit rules, enables `make` to rebuild the executable code after a change has been made to the source code. The general rule used in `make` is stated as follows in [Fel79]:

> To "make" a particular node N, "make" all the nodes on which it depends. If any has been modified since N was last changed, or if N does not exist, update N.

Creating and maintaining makefiles may be a cumbersome task; it is up to the user to continuously infer dependencies and ensure that the referenced files actually exist. For a programming language such as C,

with relatively simple relations between the modules, there are several tools (e.g. *makedepend* [Bru91]) that automatically determine such dependencies and generate makefiles. However, it has been widely observed that `make` is not particularly helpful in avoiding unnecessary recompilations. For example, many C programmers who worked with makefiles are familiar with the situation when a large number of files are recompiled after a small change is made to just one header file. It is unlikely that any language independent tool can be smart in this respect. To be fair, though, we have to say that generality and language independence is in some sense an advantage of `make`, since it does not only support compilation and linking — any user-specified commands can be executed on the files dependent on the ones that have been changed, and quite complex sequences of operations can be encoded.

3.1.1.2 Smart Recompilation

In large application systems, recompilations represent a significant part of the maintenance costs. For example, it has been once reported, that in a large Ada application more than half of the compilations were redundant [ATW94]. Avoiding unnecessary recompilations is therefore an important issue, and, as we have already mentioned, `make` is certainly not an ultimate solution here.

Primitive recompilation techniques have a result that if a module containing declarations is shared by many other modules, any change to that module initiates recompilations of all the other modules — whether or not they use a changed declaration. Tichy [Tic86] has proposed a "smart recompilation" method for reducing the number of recompilations after a change to such declarations. The compiler's[1] symbol table was extended to keep track of finer granularity dependencies between declarations (type definitions, constants, variables, etc.) in a compilation context and the items in the compilation units referencing the declarations. The possible changes in the context are classified. For each kind of change, the dependency information is used in a test to decide whether recompilation is necessary.

An extension of Tichy's "smart recompilation" to "smarter recompilation" is described in [SK88]. It is argued that Tichy's definition of compilation consistency could be relaxed without the risk of introducing new errors and thus reduce the turn-around time even further.

A proposal for reducing unnecessary recompilations by analysing the source code, detecting dependencies and then clustering related declarations, files, etc. is described in [SP89].

At present smart recompilation techniques are widely adopted in integrated development environment (IDE) products for popular languages. Most of the IDEs for C/C++, e.g. Microsoft Visual C++ [Mic00a] or Borland C++ Builder [Bor00a], have this feature[2], as well as IDEs for Pascal and its descendants, e.g. Borland Delphi [Bor00b]. However, there seem to be very few implementations of smart recompilation for Java. We discuss this in more detail in the "Related Work" section of this chapter.

[1] A Pascal compiler was used in a prototype implementation, but the method is generally applicable

[2] Surprisingly, however, smart recompilation is not supported in C/C++ compilers for major Unix platforms, e.g. Solaris or Linux.

3.1.2 Smart Recompilation of Java Applications

From the point of view of build management, Java is distinct from most of the other industrial-strength programming languages in at least one aspect, and it is its late linking model, which we discussed in Section 2.3.1. This model, on one hand, is a strength of Java, since (as long as binary compatibility is maintained) it allows applications to be constructed out of binary classes coming from different sources and updated independently. Being able to run applications assembled dynamically from distributed locations, e.g. various Internet sites, was actually the original purpose that led to this model. It converts the final linking phase into a lazy incremental resolution process that happens every time a new class is loaded by the JVM[3].

An additional feature of Java, is that dependencies between programming modules (classes), can take many forms. Class `C` can call a method of class `D`, be `D`'s subclass, implement interface `D`, declare a local variable of type `D` in one of its methods, to name but a few. Some changes to `D` will affect `C` and some not, depending on the actual kind of relation(s) between `C` and `D`. For example, consider what happens if method `m()` is deleted from `D`. If `C` previously called this method, it should necessarily be updated and recompiled. However, if `C` simply declares a local variable of type `D`, nothing is need to be done with it.

Combined together and applied to large, complex applications with many internal dependencies, these two properties substantially reduce the degree of control over the consistency of the resulting binary application when a change to some class is made. There is no linking of the entire application, that could have diagnosed many of the problems such as "member declared but not defined". And the complex nature of dependencies between classes makes the traditional hand-crafted makefiles very unreliable. Therefore the developers basically have a choice only between recompiling individual changed classes and recompiling the whole application. Since the latter is time-consuming[4], the developers tend to use mostly the former option, and only occasionally the latter. Thus the problems they experience are mostly due to broken, or, worse than that, working but now-incorrect links. It would therefore make sense to define smart recompilation for Java as a technology that would guarantee change validation (or, in other words, source compatibility verification) and recompilation of all necessary classes after a change is made, yet avoiding the majority of redundant recompilations.

To be fair, we should mention that to comply with the standard, a Java compiler should contain a kind of "make" functionality, which tries to find the Java sources for all of the classes that the given class references, and recompile them if they are newer than the corresponding class files. This partially eliminates the above problem. However, `javac` won't work in the other direction, i.e. find any classes that depend on a given class. This is not supported for at least one reason: there needs to be some way of identifying the set of classes that constitute an "application" for which we want to maintain consistency. Some kind of a database need to be maintained, that refers to application classes and their sources. Ideally all of these programming units should be kept inside the database, to prevent malicious or accidental file deletions, moves, etc., but it is also acceptable to keep them on the file system and store paths to these files, plus some additional information in the database. A standard Java compiler does not support anything like that.

However, for a tool that always runs over a persistent store, the store itself can be used to host such a

[3]Some development platforms for Java, e.g. Visual Cafe [Web00] have an option that allows to compile a Java application into a single binary executable file. This, however, makes the application non-portable, eliminating one of the main advantages of Java.

[4]For the latest Java compilers written in C, compilation speed is typically much higher than for the original Sun's `javac` compiler written in Java itself, which we used in this work. But applications are also getting larger, and people are still unlikely to use the "recompile all" option all the time.

database. For this reason, evolution for a persistent platform and smart recompilation work together in a very natural and convenient way.

An additional benefit of using tracking of incompatible changes and smart recompilation in persistent applications evolution (and also runtime evolution, see Chapter 7) is that in both of these cases the persistent store (active VM) can contain more classes than the developer may think or expect. Classes can come from different sources, can be loaded dynamically using custom class loaders, or can be synthetic. "Recompile all" may simply not work for such classes, whereas the build management technology would necessarily inspect all of them and detect any potential problems.

3.2 Management of Evolvable Classes in PJama

PJama persistent build technology keeps track of every persistent class (excluding Java Core classes that can't evolve, see Section 2.5.3), as well as of non-persistent classes that are part of persistent applications that run over a given store. We decided to manage persistent and non-persistent classes uniformly for the developer's convenience. When the number of classes of which a persistent application consists becomes large enough, it is quite difficult for the developer to keep track of which classes are currently persistent and which are not. Therefore it would be very inconvenient to use the standard Java compiler for some classes and persistent build tool for the others. It is also useful to maintain source compatibility for as many classes as possible.

However, how does the tool know which non-persistent classes belong to the persistent applications running over a given store? One solution would be to manually maintain an explicit "project", i.e. a list of all files of an application. It is done this way in many integrated programming environments, e.g. Forte for Java [Sun00b] or JBuilder [Bor00c]. This solution, though it gives the developer the greatest degree of control over the application, will require some manual work. It will again bring the issue of distinguishing persistent and non-persistent classes, since persistent classes must always be included in the "project". It also exposes the system to inconsistent recording of this information by the developer.

Another solution would be to patch the VM such that every time a persistent application terminates, a special routine is called which iterates over all classes currently in memory and saves their names in the store. We believe this operation will not bring any noticeable latency to the system, so it might be implemented in future[5]. For now, however, a simpler solution is adopted: the developer informs the system about application classes by just using the tool consistently over the given store, i.e. calling `opjb` over the given store every time they would otherwise run `javac`. Once the name of a `.java` source is passed to the tool, it registers the corresponding class and then keeps track of it on each invocation.

[5]This can be formed efficiently by iterating over the current class loaders. However, this would miss classes loaded by loaders that became unreachable and were garbage collected. In the present PJama context such class loaders are irrelevant — we can't evolve classes loaded by them (see Section 5.5.4). However, if this changes, then to register all classes ever loaded by the VM, we might modify class loaders to inform the `opjb` databases of the classes they load.

3.2.1 Evolvable Class Directory

Evolvable Class Directory is the main persistent data structure supporting the persistent build technology. This structure contains the information on every evolvable class and is updated after each successful run of `opjb`. The format of records of which ECD consists is presented in Figure 3.1. Since it is a Java structure, we are presenting the original Java class describing an individual record:

```
public class ECDEntry {
  String className;
  String javaFileFullPath;
  long javaFileLastModified;
  long javaFileFingerprint;
  String classFileFullPath;
  long classFileLastModified;
  long classFileFingerprint;
  boolean noJavaSource;
  byte[] classFile;
}
```

Figure 3.1: The Java class describing an ECD entry.

Most of the field names are self-explanatory. Fingerprints are long numbers, essentially check sums, that are calculated for the contents of both the `.java` and the `.class` files for the given class, and used for quick identity tests. The `noJavaSource` field is `true` for those persistent classes for which `.java` sources are not available, e.g. third-party libraries. They are treated specially, as discussed in Section 3.2.3.

The `classFile` field is `null` for persistent classes and refers to an array containing class bytecodes for a transient class. This information is kept to enable comparison between versions of non-persistent classes. If a class subsequently becomes persistent, its class file is promoted into the store along with the static variables, etc., and in the ECD the `classFile` for this class is set to `null`.

3.2.2 Locating `.class` and `.java` Files for Persistent Classes

The first time the tool is called over the given persistent store, it needs to locate the `.class` and `.java` files on the file system for all evolvable classes. The `.class` files are looked up in the directories specified in the standard `CLASSPATH` environment variable. Alternatively, the "`-classpath` *directories*" command line option can be used, which overrides the settings in the environment variable. To specify the directories containing the source files for classes, the "`-sourcepath` *directories*" command line option is used.

It is assumed that the source code for a class is contained in the file with the same name and the `.java` extension. In practice this, however, is not always the case. The `javac` compiler requires that the class whose source code is contained in a `.java` file, has the same name as this file. However, more than one top-level class may be contained in one source file, so it is actually only the name of the first class in the given source file which is checked. `opjb` first tries to find a file called `package/C.java` for a class `package.C`. If it can't find a file with this name, it asks the developer to provide the full path to the file. The developer

has to give the name of an existing file or abort the whole operation. The same thing happens if the tool is unable to find the source file in the previously stored location, e.g. if this file was moved.

3.2.3 Classes with no `.java` Sources

Our persistent build tool requires that for all evolvable classes the corresponding source files exist. We believe this is a sensible requirement, since, after all, changing the source code of a class is the only way to evolve it in PJama. If accidentally for some persistent class the source file is lost, the developer may use the functionality provided in the "engineering" class library for PJama to extract the bytecode of the class from the store, and then use one of freely available Java disassemblers to recreate the source code of this class.

However, one special case with respect to source code for class is allowed. It is due to the fact that some applications may use third-party class libraries, for which source files are unavailable. So on the first invocation of `opjb` over the store, the developer can specify the names of class packages for which source files do not exist, using special "`-nosources`" command line option (see Appendix A).

The tool will then treat all of the classes in these packages separately. It will not attempt to check or evolve these classes in the same way as all of the others, for two reasons. First, third-party library classes are not supposed to evolve frequently, so it will be wasteful to check them every time the user builds an application. Second, more importantly, it does not make sense to apply our standard procedure of change validation (source compatibility verification) and recompilation (see Section 3.3) to these classes, since they do not have source code to recompile, and since, according to the recommendations of the JLS, all changes to such classes should be binary compatible anyway. For these reasons, third party classes are evolved separately from other classes, bypassing most of the standard validation procedure, when the developer explicitly applies a special command line key.

In summary, third-party classes available only in class file format can be replaced, but fewer safety checks are applied.

3.3 Persistent Build Algorithm

Below is the algorithm of operations performed by the PJama evolution tool:

1. Initialize to empty sets the following sets:
   $\{UJF\}$ - a set of updated `.java` files;
   $\{RJF\}$ - a set of recompiled `.java` files;
   $\{UC\}$ - a set of updated classes;
   $\{UCV\}$ - a set of updated classes for which the changes have been validated successfully;
2. Find any evolvable classes in the Persistent Class Directory (PCD) (see Section 2.1.3.4) that are so far not in the ECD. Find `.java` files for them and enter all relevant information into the ECD. Then find the `.class` files for them on the file system, and enter their paths into the ECD. However, take

the dates and fingerprints for these class files from the persistent copies of these classes, not the file copies, since the ECD at this time will contain information on old class versions.

3. Put into the $\{UJF\}$ set all `.java` files which have been updated since the last run of the tool. These are the files for which actual date or fingerprint differs from that saved in the ECD, or which are newer than the respective `.class` files. The latter method, though less reliable, is the only way to spot a change to a `.java` file for a class which has just been entered into the ECD, i.e. for which the information on an older version of the `.java` source is not available.

4. Pass $\{UJF\}$ to the compiler. The compiler will produce a new `.class` file (maybe more than one) for each `.java` source. However, some of these `.class` files may be the same as their older versions. On the other hand, the compiler may pick up and recompile additional `.java` files not contained in the $\{UJF\}$, and produce some changed `.class` files. Therefore, find and add all `.java` files processed by the compiler, to $\{RJF\}$.

5. Initialize $\{UC\}$ and $\{UJF\}$ as empty sets. Find all `.class` files on the file system, which are not yet in $\{UCV\}$ and which have been changed since the last run of the tool, i.e. the fingerprint of the `.class` file has changed compared to the value preserved in ECD. Put these classes into $\{UC\}$ and add them to $\{UCV\}$.

6. Validate each class C_i in $\{UC\}$, i.e. compare its old and new versions:

   (a) Check if the object format for C_i has changed. If so, ensure there is a conversion method for this class or get the confirmation of default conversion from the user.

   (b) Check if any changes to the new version of C_i are *source incompatible* (see Section 3.3.1). For each detected source incompatible change:

      i. Find all evolvable classes that may be affected by this change.

      ii. Get the `.java` sources for these classes which are not yet in $\{RJF\}$ and add them to $\{UJF\}$.

   (c) If $\{UJF\}$ is not empty, go to step 4.

In case of any problems this algorithm aborts without saving any updates to persistent data structures, including the ECD. However, the recompiled `.class` files remain in place. Algorithm abort can happen either if the system is unable to find a `.java` file for a class that should have one, or if errors are detected during recompilation, or during change validation, e.g. when a class that has some instances is made `abstract` and no way of converting these instances is provided.

The results of work of the code that implements this algorithm are recompiled classes on the file system plus the classes in the $\{UCV\}$ set that should now be substituted in the persistent store. "Physical" substitution of a persistent class involves copying the static variables between the old and the new class versions, updating the Persistent Class Directory (PCD) of the store, plus some low-level operations related to instance rebinding and possible conversion. The detailed description of this procedure is presented in Chapter 6.

3.3.1 Source Incompatible Changes

A source incompatible change to a class is a change to its `.java` source which, once made, may break the contract between this class and other classes that use it (its *client classes*), by either

- preventing successful joint compilation of this class and classes that use it (its client classes); or
- causing an application that includes this class and its client classes, to run incorrectly, if client classes are not recompiled.

An example of a source incompatible change which causes the first of the above problems is deletion of a public method from class `C`. Another class which calls this method and is not updated accordingly, will not pass compilation with the new version of `C`. An example of the second kind of a source incompatible change is adding a `static` method `m()` to class `C`, which overrides a method with the same name and signature in `C`'s superclass `S`. The problem here is as follows: if there are any calls in the form `C.m()` in other classes, they were compiled into hard-wired calls `S.m()` in class binaries. Therefore, even in presence of the actual `C.m()` method, these classes will still call `S.m()`, until they are recompiled.

We have formulated the above definition of source compatibility after reading Chapter 13 of the JLS, which is the only place in this book where source compatibility is mentioned, though never detailed. JLS in fact describes only *binary compatibility* issues. Quoting the JLS, "a change to a type is binary compatible with (equivalently, does not break binary compatibility with) pre-existing binaries if pre-existing binaries that previously linked without error will continue to link without error". So, binary compatibility is concerned about successful class linking and does not always guarantee formally correct program execution, i.e. the execution that the programmer would expect looking at the source code of classes (see examples of such a behaviour in e.g. JLS §13.4.23). Source compatibility, on the other hand, is about successful class compilation, and formally correct program execution. Several examples in the JLS show that the requirement for source compatibility is stronger than the requirement for binary compatibility, i.e. every source compatible change is also binary compatible, but not every binary compatible change is source compatible.

Our choice to maintain source compatibility, rather than binary compatibility, was made to ensure that the existing programs after a change work precisely as expected, and thus the application developers are given the most reliable support.

Tables 3.1 – 3.5 define all of the source incompatible changes to classes which are detected by the PJama evolution technology, and for each such change — the classes that may be affected by it. The list of source incompatible changes was created by studying Chapter 13 of JLS, and also the work by Drossopoulou *et al.* [DWE98] (the latter group worked on issues related to formal verification of Java programs for several years, and has discovered some "holes" in the JLS). Then we determined what statements in other classes, if not changed, will cause source incompatibility problems. These statements refer to `C` or `I`, the class or interface being changed, in some way. After compilation, these references are placed in various parts of the binary Java class (see details in the next section). Summarising the above information, we can finally derive criteria for selecting a set of classes that are affected by a particular change. In some cases we, however, extended these sets of classes, to avoid implementing complex procedures that parse method bytecodes. For example, it is clear that if class `C` is made `abstract`, then only those classes which previously contained the "`new C()`" expression, need to be recompiled. We, however, replace the test for the presence of a bytecode corresponding to "`new C()`" with a simpler test for a reference to class `C` from the *constant pool* (see next section) of another class. Our observations, though currently limited to applications consisting of at most one or two hundred classes, show that the number of classes recompiled unnecessarily and, what is more important, the overall time overhead caused by such a lack of precision, is acceptable, i.e. is a small fraction of the total execution time. In future we may consider making our smart recompilation more precise by analysing the bytecodes of methods of client classes as well.

Class and Interface Modifiers	
Change to class `C` *or interface* `I`	*Potentially affected classes*
Adding `abstract` modifier	Classes referencing `C` (but not `C` array class) *directly* (see Section 3.3.2) from their constant pools. In addition, check that there are no persistent instances of `C`, or that all such instances are migrating to other classes.
Adding `final` modifier	Immediate subclasses of `C`.
Removing `public` modifier	Classes that are members of different packages and reference `C` (or `C` array class) as follows: directly from their constant pools, as the type of a data field, as the type in a method (constructor) signature or a thrown exception, as a directly implemented interface or a direct superclass.

Table 3.1: Changes to class and interface modifiers and affected classes.

Superclasses and Superinterfaces	
Change to class `C` *or interface* `I`	*Potentially affected classes*
Deleting class (interface) `S` from the list of superclasses (directly or indirectly implemented interfaces) of class `C`	Classes that reference **both** `S` and `C` (or array classes for one or both) as follows: from their constant pools (directly or *indirectly*), as a type of a data field, as a type in a method (constructor) signature or a thrown exception, as a direct or indirect superclass or superinterface.

Table 3.2: Changes to superclass/superinterface declarations and affected classes.

Class and Interface Field Declarations	
Change to class `C` *or interface* `I`	*Potentially affected classes*
Adding a non-private field `f` to `C` that hides a non-private field with the same name in `C`'s superclass `S`[6]	Classes that reference `S.f`
Deleting a non-private field `f` from `C`[7]	Classes that reference `C.f`
`public` field `f` made `protected`	Classes referencing `C.f` that are members of different packages and are not `C`'s subclasses (direct or indirect)
`public` or `protected` field `f` made "default access"	Classes referencing `C.f` that are members of different packages
`public`, `protected` or "default access" field `f` made `private`	Classes that reference `C.f`
Non-private field `f` made `final`	Classes that reference `C.f`
A `final` modifier is deleted for `f` field which is a *primitive constant*, or its initial value changed[8]	All application classes
Non-private instance field `f` made `static` or vice versa	Classes that reference `C.f`
Non-private field `f` made `volatile` or vice versa	Classes that reference `C.f`

Table 3.3: Changes to field declarations and affected classes.

Method Declarations in Interfaces	
Change to interface `I`	*Potentially affected classes*
Adding method `m()`[9]	Classes directly implementing `I`
Deleting method `m()`	Classes referencing `I.m()`

Table 3.4: Changes to interface method declarations and affected classes.

[6]This is a binary compatible change (JLS, §13.4.7). However, it is source incompatible.

[7]This means that the field is no longer *declared* in class `C`. According to the Java class bytecode specification, all references to a field in other classes have a strict form `C.f`, so even if `f` is moved to a superclass of `C`, classes compiled against the old version of `C` will not run with the new version.

[8]A field is called primitive constant if it is `final`, `static` and initialised with a compile-time constant expression. The above change is binary compatible (JLS, §13.4.8). However, it is source incompatible, since pre-existing Java binaries include the *value* of the constant, rather than a reference to it, and will not see any new value for the constant, unless they are recompiled. This is a side-effect of the support of conditional compilation in Java, as discussed in JLS §14.19.

[9]In JLS this change is considered binary compatible (§13.5.3). In [DWE98] the authors point out that it is source incompatible and also raise some issues about how binary compatibility should really be defined.

Method and Constructor Declarations in Classes	
Change to class `C`	*Potentially affected classes*
Deleting non-private method `m()` or constructor `C()`[10]	Classes referencing `C.m()` or `C()`
Adding one or more constructors, all of which have non-zero number of parameters, to class `C`, which previously had no constructors[11]	Classes referencing parameterless constructor `C()`
`public` method `m()` or constructor `C()` made `protected`	Classes referencing `C.m()` or `C()` that are members of different packages and are not `C`'s subclasses (direct or indirect)
`public` or `protected` method `m()` or constructor `C()` made "default access"	Classes referencing `C.m()` or `C()` that are members of different packages
`public`, `protected` or "default access" method `m()` or constructor `C()` made `private`	Classes referencing `C.m()` or `C()`
Changing the signature (includes the result type) of a non-private method `m()`	Classes referencing `C.m()`
Making a non-private method `m()` `abstract`	Classes referencing `C.m()`
Making a non-private method `m()` `final`[12]	Direct and indirect subclasses of `C` implementing `m()`
Making a non-private instance method `m()` `static` or vice versa	Classes referencing `C.m()`
Extending the set of exceptions thrown by non-private method `m()` or constructor `C()`	Classes referencing `C.m()` or `C()`
Adding to `C` a non-private method `m(xxx)` or constructor `C(xxx)` which overloads an existing (declared or inherited) method `m(yyy)` or constructor `C(yyy)`[13]	Classes referencing `C.m(yyy)` or `C(yyy)`
Adding a non-private `static` method `m()` to class `C`, that overrides an inherited static method with the same name defined in class `Csuper`[14]	Classes referencing `Csuper.m()`
Adding a non-private method `m()` to class `C`, when a method with the same name is declared in `C`'s subclass `D`, such that now `m()` in `D` overrides or overloads `m()` in `C`[15]	`D` and classes referencing `D.m()`

Table 3.5: Changes to class method/constructor declarations and affected classes.

[10]For methods, this means that `m()` is no longer *declared* in class `C`. According to the JLS, all references to this method in the bytecodes of other classes have a strict form `C.m`, so even if `m()` is moved to a superclass of `C`, classes compiled against the old version of `C` will not run with this new version.

[11]This is equivalent to deletion of the only existing parameterless constructor.

[12]Making a static method final is a binary compatible change (JLS, §13.4.16). However, it is source incompatible.

[13]This is a binary compatible change (JLS, §13.4.22). However, it is generally source incompatible, because for some client classes a problem of finding the most specific (JLS, §15.11.2.2) method or constructor can arise.

[14]This is a binary compatible change (JLS, §13.4.23). However, it is source incompatible, since due to strict references to method `Csuper.m()` from bytecodes of `C`'s client classes, their behaviour will be not as expected, if their sources are not recompiled.

[15]Citing the JLS, §13.4.5, "Adding a method that has the same name, accessibility, signature, and return type as a method in a superclass or a subclass is a binary compatible change". However, in a more general case, i.e. when the accessibility or the signature of the added method may be different, this kind of change is generally source incompatible. The compilation errors like "method made less accessible in a subclass" or "failure to find most specific method" may arise, or method calls can be re-bound to different methods during recompilation.

3.3.2 References in Java Binary Classes

The table in the previous section gives the conditions under which classes should be recompiled, in a technically precise form. Therefore, we use some terms, first introduced in the specification of Java binary class format, plus our own terms *direct* and *indirect references* from one binary Java class to another.

In order to develop a better idea of how Java binary classes reference each other, the reader is encouraged to read Chapter 4 of the Java Virtual Machine Specification [LY99], which details the class file format. Here we are presenting a short summary of class referencing issues.

A binary Java class references other classes only symbolically, i.e. by names. All these names are stored in the form of string constants in this class' *constant pool*, which is a table of variable-length structures representing various constants that are referred to by the class. How can we find out which string constants contain real class names? We can do that firstly by tracing references from other entries of the constant pool, which have more explicit types, denoted as follows: `CONSTANT_Class`, `CONSTANT_Fieldref`, `CONSTANT_Methodref` and `CONSTANT_InterfaceMethodref`. These entries are added to the constant pool of class `C` if `C` references some other class `D` explicitly, or references `D`'s fields or methods. In addition, every class that declares its own fields or methods has a non-empty *Field table* and *Method table*. Entries of this table, denoted `field_info` and `method_info`, contain indexes into the constant pool, and at these indexes there are string constants (constant pool entries of type `CONSTANT_Utf8`) representing the names and signatures of the respective fields and methods. The structure of each of the presented constant pool entry types, as well as of a Field/Method table entry, is depicted in Figure 3.2. Arrows represent indexes of other entries which an entry contains.

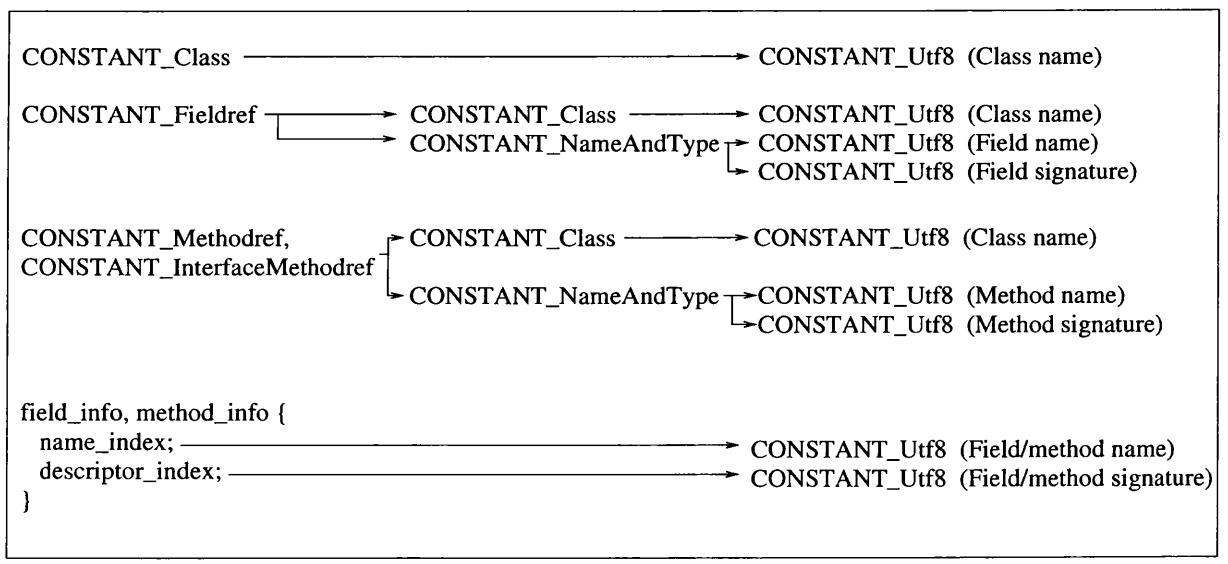

Figure 3.2: Binary Class Constant Pool Entries Structure

By saying that class `C` is referenced *directly* from the constant pool of another class, say `D`, we mean just that there is a separate `CONSTANT_Class` entry for class `C` in `D`'s constant pool. This kind of reference appears, for example, if "`new C()`" statement or a declaration of a local variable of type `C` appears somewhere in the source code of `D`. `CONSTANT_Class` entries for `C` also appear in `D`'s constant pool (in addition to the relevant `CONSTANT_Fieldref` or `CONSTANT_Methodref` entries), if class `D` uses a field or calls a method declared in

class C.

On the other hand, if, for example, class D calls a "void m(C c)" method of yet another class E, then the only reference from D to C will be through the signature of method m() which is stored in D's constant pool. We call this an *indirect* reference to C from the constant pool of D.

To be able to look up all these references, we have developed our own "advanced reflection" API. At present it is implemented partly in Java, and partly in C, in the form of Java native methods. The reason for using C is that we currently obtain most of the information we need by scanning the internal representation of classes in the JVM memory. This works very fast. However, to increase the code portability, this can be replaced with analysis of the class bytecodes, in which case this API can be rewritten in pure Java.

3.4 Related Work

3.4.1 Build Management in Persistent Platforms

Sjøberg's PhD thesis [Sjø93b] seems to be the only work at this time where build management in the context of persistent platform (in this case, Napier88 persistent language [MBCD89]) is investigated. The work addresses the general problem of supporting changes in large applications over long periods of time, includes one case study for a real industrial application, etc. Its main contribution is a persistent thesaurus mechanism, that analyses persistent software and stores information about name occurrences and dependencies between occurrences. The tools that use this information are a querying tool and EnvMake, whose functionality is similar to our persistent build tool. EnvMake could keep track of several kinds of dependencies to determine what operations had to be redone, but to a large extent the system depended on naming and file-usage conventions, that must be followed by programmers.

3.4.2 Smart Recompilation for Java

So far we have found three products supporting smart, or incremental, compilation of Java applications. Unfortunately, there are no publications on any of them, that would explain the product's internals. Therefore the following discussion is based on our own informal evaluation and guessing.

3.4.2.1 Borland JBuilder

Borland JBuilder [Bor00c] IDE for Java supports smart compilation (called "smart checking" in its terminology), as one of the compiler options, along with two others: "recompile the current file" and "recompile the whole project". When we evaluated this product, we found that it does not recognise all source incompatible changes to classes. Unrecognised changes include, for example, modifying the value of a primitive constant and adding a non-private method to class that overloads an existing method.

3.4.2.2 IBM Jikes

The second product supporting smart (incremental, in their terminology) compilation is the Java compiler called Jikes [IBM00a], an open source project being developed at IBM. It is written in C++ and its sources are available for free, as well as the binary code for many hardware/OS platforms. This product is not an IDE, but a command-line compiler, which has an additional "incremental build" mode. The compiler enters this mode if it is invoked with a special "++" command-line option for a `.java` source file containing the `main(String args[])` method, e.g.

```
>jikes ++ myprog.java
```

This will compile `myprog.java` and other files that it depends on, as needed, and leave Jikes running. The developer can then change `myprog.java` or any of the source files it depends on, and simply hit "Enter" at the command line to tell Jikes to re-check the dependencies and only recompile files as required to bring the entire project up to date. The compiler will stay in this "hit Enter/rebuild" loop until the developer enters a "q", which tells it to terminate.

Support for dependency checking looks good in this compiler — we tested it using our specification of source incompatible changes in Section 3.3.1 and did not find any "holes". The only minor disadvantage that this system has is that its support for incremental build seems to be based on very close interaction with the compiler, and it uses compiler's intermediate memory-intensive data structures, such as abstract syntactic trees, symbol tables, etc., which occupy a large amount of memory. This is what can be deduced from the documentation, which does not specify explicitly what mechanism is used to implement incremental recompilation or what checks are performed. Repeated creation and destroying of the supporting memory structures sometimes leads to noticeable latency on each invocation of rebuild.

3.4.2.3 IBM VisualAge

IBM VisualAge for Java [IBM00b] seems to be the most sophisticated IDE for Java currently on the market. It provides visual project and class navigation, incremental compilation, source-level debugging, visual program design, and version control. In VisualAge, classes are presented to the developer as objects, referencing the objects for their members. All of these objects are contained in a special repository, eliminating the notion of source files. The developer edits, saves and recompiles methods individually. This is similar to the approach pioneered in Smalltalk (in fact, there is a hint in [AAB+99] that VisualAge itself is written in Smalltalk), and is in contrast with most of the other IDEs which operate with the more traditional file-based source code representation. The incremental compiler is invoked automatically whenever a new Java class, interface, or method is created, or an existing one is changed. It recompiles the changed item, as well as all of the other items within the work space, that are affected by this change (we say "items" since it looks as if the granularity of recompilations in VisualAge can be smaller than individual classes — perhaps it recompiles individual methods). We have not found any "holes" in its mechanism of class consistency validation. When the developer redefines or deletes a method, or imports a new class, interface, or package, the compiler immediately flags any inconsistencies created by that change.

However, the IDE's high resource consumption, the fact that it is not available on all platforms, and finally the fact that it may not necessarily be useful in some situations where smart recompilation is still beneficial,

is probably the only disadvantage of this product. Indeed, in [AAB+99], the authors (the developers of the Jalapeño JVM written in Java) express the following complaint:

> "One source of perennial headaches is the impedance mismatch between Java's compilation strategy and the `make` facility. There seems to be no way to ensure that class files are up-to-date short of erasing them all and rebuilding the whole system."

Given that VisualAge and Jalapeño are being developed at the same company, and that the authors of the above quotation are well aware of VisualAge's existence, it looks as if their problem was either that they were developing on a hardware/OS platform for which VisualAge was not available, or, more likely, that their build process was more complex than just a pure Java application build. A solution that may help in the latter situation is suggested in Section 3.5.2.

3.5 Future Work

3.5.1 Speeding Up Smart Recompilation

At present our mechanism analyses the actual class objects of client classes every time an incompatible change is detected to the "current" class. Such an implementation is relatively simple, but not speed-optimal, since, for example, it can result in repeated parsing of constant pools or method signatures of *all* of the application classes. So far in our informal experiments, where the largest application consisted of about 100 classes, we did not notice any substantial slowdowns in the worst cases, when many classes were parsed over and over again (and we believe those time differences, about one or two seconds, which may have occurred, are not worth measuring). However, we realise that the situation may become worse for applications consisting of larger numbers of classes. Therefore in future it may be worth implementing an optimised variant of this technology that includes essentially a persistent cross-reference dictionary data structure.

For optimum performance, we propose maintaining a dictionary with $n : n$ relationship between classes, annotated with the dependency kind. For each class `C` there is a variable-size dictionary entry, which is divided into sections that represent types of references to `C` from its client classes, such as "extends", "references from a method signature", etc. In each such section, all of the `C`'s client classes which have a reference to `C` of the respective type, are recorded. Thus if class `D` extends `C`, declares a field of type `C` and declares a method `m(C c)`, `D`'s name will be recorded in three sections of the dictionary entry for `C`: "extends", "references from the constant pool" and "references from a method signature". With such a scheme, once we have detected a change to `C`, we can instantly obtain all its client classes that may be affected by this change. However, once `C` is recompiled, we will have to update entries for all classes for which `C` itself is a client class, and delete `C` from records of those classes for which it is no longer a client.

3.5.2 "JavaMake" Utility

Using almost the same technology as we currently have, it is possible to implement a standalone command line utility, let us call it `javamake`, with the functionality similar to `make`, but for Java applications. Such a utility will use as its input a simple list of all application source files, the target directories for the `.class` files, and the mapping between the source and the class where it can not be deduced from the file names. An algorithm used by this utility for determining which classes to recompile would be essentially the same as the one in `opjb` described in this chapter. However, there may be one difference. Our present implementation relies on the fact that old class versions (except for those classes which have just been added to the application) are preserved in the persistent store. `javamake` will either have to do the same, i.e. implement its own "persistent store" for old class versions, or work according to the part of the algorithm which is used when old class versions are not available. It seems to us that to force such a utility to always use the first option may not be practical, since it may not always be convenient for the developer to manage an additional class repository created by the utility. On the other hand, the presence of this preserved data would allow `javamake` to still work correctly if, accidentally or deliberately, some classes are recompiled bypassing it (such events inevitably happen once the file system, with its freedom of changes and lack of access consistency control, is used). The advantage of such a utility over an IDE is that it is light weight, and that it can be easily used within more complex build procedures, that may involve many more operations than compilation of the sources of a single Java application.

3.5.3 Formal Proof of Completeness and Correctness

The tables of source incompatible changes and affected classes presented in Section 3.3.1 were defined using our knowledge of Java and experience, but they have no formal proof. Consequently, we can not be absolutely sure that these tables embrace all of the affected classes. There is more confidence that they cover all possible changes to Java classes, since, after all, the number of class elements is limited and the taxonomy of changes is created as a series of additions, removals and changes to these elements. Formal proof of completeness and correctness can probably be achieved using some kind of formal technique, e.g. similar to one used in [DWE98] and other works by this group (see [SLU00]). This, however, might be a very difficult task, considering that, for example, the above group has, in our opinion, achieved quite modest practical results, despite using elaborate mathematical techniques. Basically they were able to prove some (already quite obvious) statements, such as that most of the binary compatible changes defined in the JLS are really safe. However, we presently can't see how this machinery can e.g. automatically provide us with the list of classes affected by some unsafe change.

3.6 Summary

In this chapter we have explained the motivation for, and then described a technology that we had developed for PJama, which combines persistent class evolution and smart recompilation. This technology tracks incompatible changes that the developers make to Java classes, and guarantees that unsafe changes, i.e. those that break links between classes or make these links incorrect, do not propagate into the persistent store. We have defined tables of incompatible changes and affected classes that we believe are complete,

though we don't have a formal proof for this.

Of the three products we found that support smart recompilation for Java, one implementation (Borland's JBuilder) is weaker than ours (does not recognise some incompatible changes), whereas the others (IBM Jikes and VisualAge) work correctly, but rely on the compiler's runtime memory-intensive data structures. None of these products documents how its implementation of smart recompilation works.

We believe that smart recompilation technology that both reduces compilation turnaround time and provides an additional degree of safety, is essential for development and maintenance of large and long-lived applications. Tracking of incompatible changes may be less important during the development of relatively small Java applications in an IDE equipped with a very fast compiler, where "recompile all" command can be used frequently. However, we believe that smart recompilation is very important for safe upgrading of persistent applications and building large and complex Java applications.

Chapter 4

Persistent Object Conversion

In this chapter, we present our implementation of *conversion* for persistent instances of evolved classes, i.e. the technology that makes instances consistent with revised definitions of their classes when necessary. Section 4.1 explains exactly what kinds of changes to classes result in the need to convert their instances. Section 4.2 presents the two main types of object conversion that can be used by the developer on PJama platform: *default* (or automatic) and *custom* (or programmer-defined). The next section describes in detail the mechanism for default conversion. Section 4.4 discusses a very important problem of naming for multiple class versions when custom conversion is used. In section 4.5 we describe *bulk* custom conversion, and section 4.6 presents another subtype of custom conversion —*fully controlled* conversion. The utilisation of all kinds of conversion is illustrated with a simple example that continues throughout this chapter. Sections 4.7 and 4.8 explain the relatively minor issues of static variable conversion and access to private members in conversion code. Finally, sections 4.9 and 4.10 present related work and thoughts on future work.

4.1 When Conversion is Required

Evolutionary operations on classes may or may not affect persistent objects. Whether objects of some class are affected or not depends on whether the modification to the class is such that the format of its instances changes.

The most general and (hopefully) applicable to any persistent Java implementation rule for instance-format equivalence is strict equivalence of the number, names, types and order of all of the instance (non-static) data fields of both versions of the evolved class. The declared order of fields may be important, as it is in PJama, because the fields are typically laid out in objects in the same order as they are declared, and accessed by their physical offsets.

However, in PJama there are a number of cases when the type of a field can be changed, but the format of instances remains the same, and the compatibility between the current value of the field and the new field's type is guaranteed. Examples of such changes are replacement of the primitive type `short` with the type `int` or replacement of a class type with its superclass type. It is possible for some primitive types in PJama

to be compatible in the above way, because in the implementation of JVM on which PJama is currently based (Sun's EVM), each primitive type field (except for fields of types `long` and `double`) occupies a fixed-size 32-bit slot in an object[1]. Class type fields are always *physically* compatible, because they are all just pointers to objects. However, there are only a limited number of cases when two class types are guaranteed to be *logically* compatible, i.e. the assignment of a field of one type to a field of another will never cause problems. It turns out that the rules of *widening reference conversions* (JLS, Chapter 5) are exactly what we need to define compatibility of class types, since our situation is equivalent to the one when the above conversions are used. As for scalar types, only the subset of *widening primitive conversions* where both arguments are of the same internal representation (which in our case means only integer), and occupy the same space in objects, is applicable — that is, guarantees the same object size and logical compatibility between old and new field types. Filtering out the JLS rules for widening conversions, we get the following types of, respectively, original and substitute data fields that are compatible in the context of class evolution in PJama:

- `byte` and `short` or `int`.
- `short` and `int`.
- `char` and `int`.
- Any class type `S` and any class type `T`, provided that `S` is a subclass of `T`.
- Any class type `S` and any interface type `K`, provided that `S` implements `K`.
- Any interface type `J` and any interface type `K`, provided that `J` is a subinterface of `K`.
- Any interface or array type and type `Object`.
- Any array type and type `Cloneable`.
- Any array type `SC[]` and any array type `TC[]`, provided that `SC` and `TC` are reference types and there is a widening conversion from `SC` to `TC`.

If the number, names and order of instance data fields of an evolved class are unchanged, and all changes to their types are compatible according to the above rules, the evolution system simply substitutes a class and doesn't do anything with its instances. Otherwise, it is necessary to *convert* all persistent instances of this class (if they exist, which is always checked by the system; this operation is quite fast, see Section 6.1).

If a class that has some persistent instances is to be deleted from the class hierarchy, its "orphan" instances should be *migrated* to other classes.

Since the mechanism that we exploit looks very similar for both object conversion and migration, in the further discussion we will often use the term "conversion" in a wider sense to denote both kinds of operation: "real" conversion and migration.

[1] In arrays, however, elements of these types always take the smallest possible space, e.g. 8 bits for type `byte` or 16 bits for type `short`.

4.2 Types of Conversion: Default and Custom

If the application developer wishes to convert instances of an evolving class, they have a choice between *default* and *custom* conversion. If conversion, in the strict sense, is implied, i.e. some class has been changed and all instances of this class should be made compatible with its new definition, it is often enough to use default conversion. This means that for each persistent instance of the evolving class, the evolution system will automatically create a new instance in the new format. The data will be copied between the old and new instances according to the simple rules described in the section 4.3. *Default migration* is also applicable when the programmer wants all instances of a class nominated for deletion to migrate to a single other class.

However, sometimes the old and new definitions of a class, and especially the application logic behind them, are such that the conversion becomes non-trivial. For example, it might be necessary to recalculate sizes from feet and inches to centimetres or split a string field, e.g. a full person's name, into several separate strings. Furthermore, transformations may be on a larger scale. For example, a value represented as a list of objects may be replaced by arrays holding each of the fields of these objects, and so on[2]. The most convenient option for the programmer in this case is to be able to encode such transformations in the same programming language that is used to create the data, in our case Java. This type of data conversion is called *custom* or *programmer-defined.*

In PJama, there are two ways of performing custom conversion. The first and simpler one is called *bulk* conversion. Bulk custom conversion is supposed to be used when all instances of some class should be converted in the same way. To perform it, the programmer should provide appropriate *conversion methods* written in Java. One such method can be defined for each evolving class. Conversion methods should have predefined names and signatures so that the evolution system can recognise them and call them with correct arguments. Conversion methods should be placed in one or more classes called *conversion classes.* Given these classes, the evolution system will scan the store linearly. For each detected instance of an evolved class it will call an appropriate conversion method, and that will create a new instance and initialise its fields appropriately.

During both default and bulk custom conversion, the evolution system automatically remembers all "old instance – new instance" pairs. After conversion is finished, it "swaps" every old instance and the respective new instance, so the new instances take over the identity of the old ones. The details of this transition are discussed in Section 5.1.

Bulk custom conversion can be combined with default conversion, that is, in one evolution transaction instances of some classes can undergo default conversion, and others custom conversion. In addition, in the conversion methods the programmer can avoid writing the code that copies the contents of fields with the same names and types from "old" instances to "new" ones, again by applying default conversion. The descriptions of the available conversion methods and the details of combining default and custom conversion are discussed in Section 4.5.

In addition to bulk conversion, *fully controlled custom conversion* is available in PJama evolution technology. It is intended to be used when the developer wants to have complete control over *how* and *in what order* the instances are converted. To run fully controlled custom conversion, the programmer should simply put

[2]Conversions where new field values depend on examining data outside the instance that is being converted are called *complex conversions* in some parts of the literature, e.g. [FFM+95]

the method called `conversionMain()` into a conversion class. If this method is present, the evolution system will call it, ignoring any other conversion methods that might be present. No automatic linear scan of the store will be performed. The programmer has the total freedom and the full responsibility for the results of such conversion. In particular, the programmer should ensure that all instances of the modified class are either converted or made unreachable, and that the correspondence between the old and the new instances is established. "Swapping" of old and new instances, however, operates on JVM's internals, and therefore is still performed automatically, after `conversionMain()` method termination.

It may be worth observing that fully controlled conversion is very similar to running an application against the store. The difference is that certain facilities, provided for evolution, are available to this application. The detailed description of fully controlled conversion is given in section 4.6.

4.3 Default Conversion

If during the change validation phase the persistent build tool detects a format-transforming modification to some class, and does not find a conversion method for it, it requests the programmer's confirmation that they want to perform default conversion of instances of this class. If the programmer wants to perform default migration of instances of a deleted class, they should specify, together with a class to delete, a class to which they want to migrate the "orphan" instances.

The system then scans the persistent store, and for every instance of the evolved class which it finds, creates a new instance in the format of the new class version. The values of the fields that have the same name and the same or compatible (as specified in Section 4.1) type in both versions of the class are copied from the old instance to the new one. In addition to that, values are converted according to the standard Java rules between the same named fields of the following numeric types that are logically, but not physically (i.e. hardware-level) compatible:

- `byte`, `short`, `char`, `int` and `long`, `float`, `double`
- `long` and `double`
- `float` and `double`

The fields of the new instance that either don't exist in the old class version, or have a different type, which is not compatible with the old one in any of the above ways, are initialized with default values (as specified in the JLS), i.e. `0` or `null`.

4.4 Custom Conversion: Class Version Naming

We are now going to discuss one of the most important issues that every implementation of programmer-defined conversion faces. It is formulated as follows: if in one piece of code we want to declare variables of

two versions of one class that doesn't change its name, how do we distinguish between these two versions? That is, if there is an evolving class called C, then declaration of a variable such as "C c;" could refer to either the old or the new version of C. How do we then declare variables of the other version of class C?

Among the contemporary persistent solutions for Java known to us, none supports Java conversion code. In fact the only product known to us at present, where simultaneous explicit handling of two versions of an evolving class in the conversion code is implemented, is O_2 [FFM+95]. This former product used its own programming language, which allowed the programmer to declare a compound type of a variable in place, for example as presented in Figure 4.1. In-place class declarations were used to declare variables of old class type, whereas the new class version was called by its ordinary name. But the Java language does not allow classes to be declared in an arbitrary place and without methods. In addition, it can be very impractical to replicate such a class declaration every time the programmer wants to declare a variable of this type. We don't know, for example, how to declare in the above way a class for a linked list element, where in the class definition there is a field of the same class type.

```
conversion function
mod_Obj(old_object : tuple(name : string, price : real)) in class C
{ ... }
```

Figure 4.1: An example of conversion function declaration in O_2.

So, if we think about this problem in the context of Java, the first thing that comes to mind (and is actually used in some commercial systems, see Section 4.9) is to use Java *reflection* mechanism [Sun00e]. This way, we could declare variables of, say, old version of the evolving class as just `Object` type, and then use classes of the standard `java.lang.reflect` package, such as `java.lang.reflect.Field`, to read/write data fields and call methods on them. The problem with this mechanism is that it is quite inconvenient to use. For example, if there is an integer field called `price` in the old version of class C, then to just read its value in the variable `c_old`, we have to write the Java code presented in Figure 4.2

```
Class C_old_version = c_old.getClass();
int price;
try {
  Field priceField = C_old_version.getDeclaredField("price");
  price = priceField.getInt(c_old);
} catch (Exception e) {
  // Several different exceptions can be thrown by both lines above
  ...
}
```

Figure 4.2: An example of access to "old" fields via reflection.

Though we need to initialize a `Field` type variable only once, the code still remains rather cumbersome. Calling methods using reflection is even more inconvenient, since their parameters should be passed as an array, which first has to be declared and initialized. Considering all of the above, we have rejected the idea of using reflection in general case, though at present we still use it to access private class members during conversion (see Section 4.8).

Another option that we have considered was to temporarily automatically rename old-version classes, so that names of their packages change in a regular way. Thus if we have an evolving class `mypackage.C`, then its old version would be temporarily assigned (by the underlying compiler and PJama VM mechanisms) a name of the form `oldversion.mypackage.C`. This naming arrangement would exist only for the duration of conversion code execution, after which old versions of evolved classes are abandoned. It is undesirable to just put all of the "old" classes into the same package such as `oldversion` due to the possibility of name clashes between classes with same names initially declared in different packages.

This approach would allow programmers to declare variables of both old-version and new-version class types in the same piece of Java code and then refer to their fields and methods as usual, so it is much more practical than the previous one. However, it still has some drawbacks. One is that we will have to use the fully qualified name to refer to either old or new version of an evolved class, to avoid name clashes. With long package names this can be inconvenient. Another drawback is due to the fact that very often Java programmers declare fields and methods as "default" (also called "package") or `protected`. In both cases all classes in the same package can access these fields. If both the old and the new version of an evolving class are in the same package, we can put the conversion code into the class which is also a member of this package, and thus it would have free access to the "default" and `protected` class members. However, the presented approach prevents the programmer from doing that, forcing them to use reflection as a last resort.

So we have finally chosen the third solution, which is temporary automatic change of the old class version's own name. To refer to the old version of an evolving class `C` the programmer adds a special predefined suffix `$$_old_ver_` to it, which results in `C$$_old_ver_` name. Our modified Java compiler and the PJama VM load from the persistent store and actually rename the appropriate classes in response to such *mangled* names. After conversion is finished, old versions of evolved classes are invalidated, so the only place where it is possible to operate on two versions of one class and distinguish them this way is in conversion code.

The unusual suffix `$$_old_ver_` was chosen as an entirely valid sequence of characters to use in a Java identifier, which is, hopefully, unlikely to clash with any "normal" class name. That's why it is a bit cumbersome. We wanted to avoid more fundamental changes to the Java language, such as introducing new characters to denote old class versions. The present naming arrangement can even be viewed as not really breaking or extending the Java Language Specification, and it requires relatively small changes to the Java compiler. However, if we had more control over Java, we would probably opt for more compact form of the old-version suffix, for example a single "pound" (#) character, perhaps optionally followed by an integer denoting class version number, to accommodate further support for incremental evolution.

4.5 Bulk Custom Conversion

In the following discussion we will denote an evolving class for which conversion is required as `C`. `Csuper` means any superclass of `C`. Let us first describe the categories of conversion methods that correspond to categories of changes to class `C`.

4.5.1 Class Modification

The signatures of the conversion methods recognised by the evolution system if class C is modified, are given below. The programmer can choose a suitable signature (only one method per evolving class) and write the method body.

```
public static void convertInstance(C$$_old_ver_ c0, C c1)
public static C convertInstance(C$$_old_ver_ c)
public static Csuper convertInstance(C$$_old_ver c)
```

The first, symmetric form of convertInstance method is the most straightforward, and also the only one that supports automatic default conversion in addition to custom conversion. Before the system calls such a method, it creates an instance c1 of new version of class C and copies from c0 to c1 the values of all data fields that have the same name and compatible types in both versions of class C. The new instance is created by just allocating memory, without invocation of a constructor.

The latter solution has been taken to avoid undesirable side effects, such as class instance counter increment or resource allocation, that the execution of a constructor might cause. However, it has one implication. It is due to the fact that Java classes are compiled such that the code for instance field initialisers is physically placed in the body of the constructor, in addition to the constructor's own code. Therefore if the programmer has declared, for example, a field new_field in class C along with the initialiser, as shown in Figure 4.3, they should not expect this field to be initialized to value 1 automatically on entering the method convertInstance(C$$_old_ver_, C). Instead, it will have the default value of 0.

```
class C {
  int new_field = 1;
  ...
}
```

Figure 4.3: Instance field initialiser.

The second and third forms of the convertInstance method are assymmetric. They allow the programmer to flexibly choose the actual class of the substitute instance during conversion. The second form can be used if the programmer wants to change the class of an instance to new version of C or to a subclass of the latter. The third form permits a replacement class that is a superclass of C or that just has some common superclass with C. Both of these methods should explicitly call the new operator to create a new instance and then explicitly copy all of the necessary data from c to the new instance.

The second form of convertInstance is guaranteed to be type safe. This means that if there are any objects in the store that refer to instances of C, for example there are instances of class CRef that declares a "C cref" field, then after conversion all references from instances of CRef remain valid, although now some or all of them can point to instances of C's subclasses. That's because Java, as any other object-oriented language, allows a class type variable to refer to an instance of a class that is a subtype of the type of this variable.

However, the third form of convertInstance method can produce an instance of any of the C's superclasses or any class that extends some superclass of C. Since the ultimate superclass of any Java class is Object, this

means that in the extreme case, an instance of *any* class can be returned by such a method. Therefore the third form of `convertInstance` method is type unsafe, and it is the programmer's responsibility to arrange that there are no illegal references after conversion is complete.

The reason for introduction of this method is that it gives the programmer more freedom in restructuring the persistent data. Its use is justified (and safe without any additional measures) if it is necessary to migrate some of the instances of class `C` to another class `D`, which has a common superclass with `C` called `S`, and all of the references to instances of `C` in the store are already only through fields of type `S`. For example, assume that we have a hierarchy of persistent classes with the common abstract superclass `Car` and its subclasses `SportsCar`, `Lorry`, `Van` etc. In the persistent objects, all references to instances of these classes are through fields of type `Car`. If we make some change to class `Lorry`, that will also require creation of its sibling class `Truck`, we can safely use the third form of `convertInstance` method to convert some of the instances of `Lorry` to class `Truck`.

A special case of class modification is when `C` is replaced (i.e. modified and renamed simulatneously). Let us denote `C`'s new name `NewC`. Being informed by the programmer, the persistent build tool knows that `C` and `NewC` are really the old and new names of the same evolved class. Therefore semantically exactly the same set of conversion methods can be used in this case:

```
public static void convertInstance(C c, NewC nc)
public static NewC convertInstance(C c)
public static C_and_NewC_super convertInstance(C c)
```

4.5.2 Class Deletion

If class `C` that has some persistent instances, is to be deleted from the class hierarchy, its "orphan" instances must migrate to other classes. The following methods can be used to perform migration:

```
public static void migrateInstance(C c0, Csuper_sub c1)
public static Csuper migrateInstance(C c)
```

Class `Csuper` may not be nominated for deletion itself. `Csuper_sub` is a class which has a common (undeleted) superclass with `C`.

As before, the first form of `migrateInstance` method receives an initialised instance of the replacement class from the evolution system. The values of all fields with the same name and compatible types in this instance are already copied from the old instance. The second method should call the `new` operator and copy the necessary data between the instances explicitly. This form of the `migrateInstance` method is also type unsafe, and its existence is justified in the same way as the third form of `convertInstance` method (see the previous section).

4.5.3 Conversion for Subclasses of Evolving Classes

If some class evolves such that the format of its instances changes, this will affect the format of instances of all of its subclasses. It is often the case that the conversion procedure for a subclass should be exactly the same as for its evolved superclass. To avoid unnecessary manual replication of conversion methods for every subclass of an evolved class, PJama supports automatically calling an appropriate conversion method for instances of a class that doesn't have its own conversion method. The conversion method defined for the nearest superclass is called, if one exists.

This rule, however, is applicable only to the first form of `convertInstance` method. In other words, only if we have a `void convertInstance(C$$_old_ver_, C)` conversion method for a changed class `C`, this method will be called for instances of `C`'s subclasses. In all other cases the system would expect the developer to provide separate conversion methods for subclasses of `C`, or will perform only default conversion for them.

This restriction is due to both semantical and safety issues. Indeed, it is both "natural" and safe to have symmetric calls of the first form of `convertInstance` method for subclasses of an evolved class `C`. When such a method is called for an instance of any `C`'s subclass `Csub$$_old_ver_`, the system creates and passes to it as a second parameter an instance of the corresponding class `Csub`. This is what the programmer would expect, and this also guarantees the safety in the sense that no references to instances of `Csub` can become invalid.

On the other hand, in most cases the code in assymmetric forms of conversion methods depends significantly on the actual class of the passed instance. For example, an instance of an appropriate class should be created with `new` operator and returned if we want to preserve type safety. In our early experiments, application of the same assymmetric conversion method to instances of initially unanticipated classes appeared to be a source of errors. It either worked incorrectly or required to create cumbersome conversion code that would analyse the actual class of the passed instance — something that, as our experience has shown, is better done by implementing multiple conversion methods for different classes. For these reasons, the described facility was eventually turned off for all but the symmetric form of the `convertInstance` method.

4.5.4 Semi-automatic Copying of Data Between "Old" and "New" Instances

As mentioned above, conversion methods that get an "old" instance as a single argument should create a replacement instance explicitly and they are fully responsible for copying data from one instance to another. However, even though the classes of these instances are most likely to be different and the class of the replacement instance may change from one invocation of the method to another, there can still be many fields with the same name and compatible types in both instances. To facilitate copying of such fields between instances, the following method is available in PJama standard class `org.opj.utilities.PJEvolution`:

public static void `copyDefaults(Object oldObj, Object newObj)`

This method copies the values of all fields that have the same name and compatible types (as defined in section 4.1), from `oldObj` to `newObj`, irrespective of their actual classes. The method uses Java reflection

to find all such pairs. To speed up copying, it caches the results (mappings between fields) for each pair of classes it comes across.

4.5.5 `onConversionStart()` and `onConversionEnd()` Predefined Methods

The developer may include the following two methods with predefined names and signatures in every conversion class:

```
public static void onConversionStart()
public static void onConversionEnd()
```

If `onConversionStart()` method with the above signature is defined in a conversion class, it will be called before any conversion method is called, but after all static conversion methods (see Section 4.7) are called. `onConversionEnd()` method is called after all objects have been converted.

There can be as many `onConversionStart()` (as well as `onConversionEnd()`) methods as there are conversion classes. The order of execution inside both groups is undefined.

4.5.6 An Example – an Airline Maintaining a "Frequent Flyer" Programme

After presenting all available predefined conversion methods, we will finally illustrate their use in a simple example. Imagine an airline that maintains a database of frequently flying customers. Each person is represented as an instance of class `Customer`. Every time a customer flies with this airline, miles are credited to their account. When a sufficient number of miles has been collected, they can be used to fly somewhere for free.

Consider the case where the application developer wants to modify the definition of class `Customer` to represent postal address data more conveniently, as shown in Figure 4.4

```
class Customer {  // Old
  String name;
  String address;
  int milesCollected;
  ...
}
```

```
class Customer {  // Revised
  String name;
  int number;
  String street, city, postcode, country;
  int milesCollected;
  ...
}
```

Figure 4.4: Two versions of evolving class `Customer`.

The single field `address` is replaced with several fields: `number`, `street`, etc., while other fields remain

the same and should retain the same information[3]. In order to convert data, the programmer can write the conversion class presented in Figure 4.5.

```
class CustomerConverter {  // The name is arbitrary
  public static void convertInstance(Customer$$_old_ver_ oldC,
                                     Customer newC) {
    newC.number = extractNumber(oldC.address);
    newC.street = extractStreet(oldC.address);
    newC.city = extractCity(oldC.address);
    newC.postcode = extractPostCode(oldc.address);
    newC.country = extractCountry(oldc.address);
  }

  ... // Methods extractXXX not shown
}
```

Figure 4.5: A simple conversion class for evolving class `Customer`.

In the sole conversion method of this class, it is sufficient to deal only with the fields that have been replaced and added. The values of those that are unchanged, such as `name` and `milesCollected`, are copied from the `oldC` object to the `newC` automatically.

Now imagine that the airline decides to change its policy and divide its customers into three categories: Gold Tier, Silver Tier and Bronze Tier, depending on the number of collected miles. In the new design, class `Customer` becomes an abstract superclass of three new classes, and each `Customer` instance should be transformed into an instance of the appropriate specialised class. In order to perform such a transformation, we have to use a conversion method that can create and return an instance of more than one class. The solution may look like the one presented in Figure 4.6

4.6 Fully Controlled Conversion

The mechanism of fully controlled conversion can be used if the programmer wants to convert instances of the evolved class in a predefined order (which is likely to be different from essentially unpredictalbe order of physical placement of objects in the store), considerably restructure the data in addition to conversion, get rid of some objects instead of converting them, and so on. To run fully controlled custom conversion, the programmer simply declares a method called `conversionMain()` in one of the conversion classes (there should be only one such method). If this method is present, the evolution system will call it, ignoring any other conversion methods that might be present. No automatic linear scan of the store will be performed, therefore it is solely the programmer's responsibility to ensure that all instances of all of the modified classes are converted or made unreachable.

There are three differences between this and normal PJama application execution.

[3]Note that all evolvable classes referencing class `Customer` are checked to ensure that they are simultaneously transformed to use the new class definition (see Chapter 3).

```
import org.opj.utilities.PJEvolution;

class CustomerConverter {
  public static Customer convertInstance(Customer$$_old_ver_ oldC) {
    Customer newC;
    if (oldC.totalMiles > 50000)
      newC = new GoldTierCust();
    else if (oldC.totalMiles > 20000)
      newC = new SilverTierCust();
    else newC = new BronzeTierCust();

    PJEvolution.copyDefaults(oldC, newC);  // Explicit copying of contents
    return newC;
  }
```

Figure 4.6: A conversion method returning instances of multiple types.

1. Temporary class renaming can be used.
2. Support is provided for arranging that new instances assume the identity of old instances (see below).
3. The checkpoint operation may not be used.

In case of bulk conversion, correspondence between an old and the corresponding new instance is established automatically by the underlying evolution system that calls a conversion method on the old instance. After a conversion method returns, the evolution takes the control and puts an "old instance - replacement instance" pair into a system table. Later the identity of the old instance is conferred on the new one, that is, all objects that pointed to the old instance now point to the new one instantaneously (the implementation details are explained in Chapter 6). Simultaneously the old instance vanishes from the computation space. If, however, the programmer opts for fully controlled conversion, they need to explicitly notify the system about every "old instance - replacement instance" pair. For that, there is a special method in the PJama standard class `org.opj.utilities.PJEvolution`:

public static native void `preserveIdentity(Object oldObj, Object newObj);`

We illustrate the usage of fully controlled conversion by continuing the airline example. Assume that, in addition to sorting customers into three categories, an application developer also decides to save them into three separate collections instead of one array. Furthermore, at the same time the developer wants to get rid of those instances for which the collected miles have expired. In order to do that, a method presented in Figure 4.7 can be added to the conversion class in addition to the already existing `Customer convertInstance(Customer$$_old_ver_ oldC)` method.

This example illustrates that fully-controlled custom conversion is most likely to be of use when an application is maintaining an extent (directly or indirectly) of all of the instances evolving.

```
public static void conversionMain() {
  // "Airline" is our main persistent class
  Customer$$_old_ver_ allCustomers[] = Airline$$_old_ver_.allCustomers;
  // Initialize the new copy of "Airline", creating empty collections, etc.
  Airline.initialize();

  for (int i = 0; i < allCustomers.length; i++)
    if (! milesHaveExpired(allCustomers[i])) {
      // This instance is valid, so we convert it
      Customer c = convertInstance(allCustomers[i]);
      // Preserve the identity explicitly
      PJEvolution.preserveIdentity(allCustomers[i], c);
      // Put new instance into the appropriate collection
      if (c instanceof GoldTierCust)
        Airline.goldC.add(c);
      else if (c instanceof SilverTierCust)
        Airline.silverC.add(c);
      else Airline.bronzeC.add(c);
    }

  ...
}
```

Figure 4.7: An example of fully controlled conversion implementation

4.7 Copying and Conversion of Static Variables

PJama supports persistence of static variables unless they are marked `transient`. This sets PJama apart from most of the other known persistence solutions for the Java platform, which treat static variables as implicitly transient [JA98]. Therefore the evolution system has to provide support for the conversion of static variables.

When the new version of class `C` is loaded, its static fields are initialised with their initialisers as usual. After the change validation phase of evolution is complete, for each class that will actually be replaced, the values of all of its static fields that have same names and compatible types in both versions are by default copied from the old version of class to the new one. The value of the field `f` is not copied, however, if `f` is `static final` in either the old or the new version of class `C`.

This is because ordinary static fields often hold some information that is obtained during program execution. Such information is preserved between executions of a persistent program and, similarly, across subsequent evolved versions of the class. In contrast, `static final` fields typically serve as constants, not accumulating any information during runtime. However, if in some new version of a class such a constant has a different value, it is most likely that this change is intentional and should be propagated into the store. For example, the programmer might want to modify a message that the program prints, or change some numeric constant due to the change from one measure system to another.

The programmer can override the above default rule for non-final statics of some class using a special command line option of the persistent build tool (see Appendix A). In that case the static variables of this class will have the values that were assigned to them by the static initialisers of the new class version. Similarly, copying of final static variables between the versions of a class can be enforced.

If simple copying of statics is not enough, a conversion method for statics can be used. This method's signature is:

public static void `convertStatics()`

If a method with this signature is present in a conversion class, it is called immediately after the default copying of statics, but before instance conversion starts (and therefore before any of `onConversionStart()` (see Section 4.5.5) methods are called). The code in this method can deal with all evolving classes and can refer to their old versions as usual, i.e. using the `$$_old_ver_` suffix.

4.8 Access to Non-Public Data Fields and Methods

Class evolution and subsequent object conversion have nothing to do with the normal execution of a persistent application. Therefore the conversion code often needs access to private fields or methods of evolving classes. The problem is partially remedied by default conversion, which copies all fields with same names and compatible types between object versions, irrespective of their protection modifiers. Also, the programmer can get access to non-public fields and methods by making conversion classes subclasses of evolving classes, putting them into the same package and even making a conversion class an inner class of an evolving class. However, these are pretty artificial measures that make the actual conversion code logic less clear, and they might not work if, for example, a single piece of conversion code needs to access several classes from different packages. Therefore we opted for allowing the conversion code to access data with any protection modifiers from any place in the conversion code.

Giving the application developer unconditional access to private members obviously violates the normal Java security rules. However, without this evolution facilities in many cases can become useless. Evolution is essentially a non-trivial operation, and it should be performed by a developer who is well aware of the persistent application structure. So at present we favour taking away access restrictions in the conversion code. To maintain security in future multi-user systems running on PJama platform, it would be more reasonable to permit access to the evolution facilities only to selected users or system administrators, or to refer an evolution to the security manager before it starts.

Currently the programmer gets access to non-public members that are not accessible from the given conversion class, via Java reflection facilities, i.e. methods implemented by classes in standard Java package `java.lang.reflect` (see Section 4.4 which presents an example of using reflection). This is not the most convenient way from the programmer's point of view, but it is at least consistent with the rest of the Java language. The latter, starting from JDK1.2, allows programs to suppress access checks performed by reflected objects at their point of use. PJama evolution systems does that automatically for the programmer, thus giving them free, though not the most convenient, access to all non-public members.

4.9 Related Work

We now switch to considering instance conversion support in a number of commercial and experimental systems, contrasting their evolution facilities with those of PJama. Recognising that many of such products will release new versions with different properties, we have used the latest information available (usually from Web sites), and annotated the bibliography with the data that was obtained.

We have already mentioned in Section 2.6 that PJama is one of the very few (and the only one of those we managed to survey in detail and present here) systems that preserves methods and static variables of classes in the persistent store. Thus it is the only system that performs behavioural consistency checks. For this reason, we simply don't mention support for static variables preservation and behavioural consistency in the discussion that follows, implying that it is not implemented.

We start the discussion with describing the object conversion support in Java Object Serialization (JOS) which is a standard persistence mechanism for Java. We then proceed to contemporary commercial OODB systems, and finally describe several experimental OODBMS.

4.9.1 Java Object Serialization

Since the JDK version 1.1, *Java Object Serialization* (JOS) [Sun00j] has been considered to be the default persistence mechanism for the Java language. JOS is a part of the standard `java.io` package and provides facilites to write/read (serialise/ deserialise) object graphs to and from bytestreams. Bytestream is an abstract representation, to which a disk file or a network connection may physically correspond. Sending objects over the network, specifically those that are arguments of remotely called methods, was actually the original use for JOS.

For a number of reasons, JOS can generally be considered only for persistence of relatively small amounts of non-critical data. It does not scale well: there is no way to change objects on the bytestream individually, and the entire bytestream should be read into memory and then written back to make any changes. It is also not orthogonal: for reasons of security, an instance is allowed to be serialised only if its class implements the standard `java.io.Serializable` interface. Many of the Java core classes do not do this. This can also preclude successful re-use of classes for which source code is not available. No standard transaction facilities of any kind are provided for JOS.

By default JOS would not allow the developer to serialise objects using one class definition and then read them back using a changed class. This is done by calculating a special value, a 64-bit "fingerprint" for the class, when it is serialised. This fingerprint, which is called the `serialVersionUID`, is based on several pieces of class data, including all of the serialisable[4] fields. When an incompatible change to a class is made, for example a serialisable field is added or deleted, instances of this class can no longer be deserialised (an exception is thrown), since the old (saved in the byte stream) and the new `serialVersionUIDs` of the class do not match. However, the developer may override this protection mechanism by explicitly assigning the old `serialVersionUID` (which can be obtained for a class using special utility) to the new class version. This is done simply by defining a `static final long` variable `serialVersionUID` in the

[4]All non-static fields of class are serialisable, except those marked with a Java `transient` modifier.

new class version and assigning it the old value. After that, a byte stream can be deserialised. However, all fields added in the new class definition will be assigned default values and all deleted fields will be lost.

The above problem can be overcome by adding a special `readObject(ObjectInputStream)` method to the changed class (similarly, a `writeObject(ObjectOutputStream)` method may be used to customise the output, e.g. to encrypt some data fields as they are written to disk). `readObject()` method overrides the default functionality used to read an object from the byte stream. Inside this method, the developer may use an instance of special inner class called `ObjectInputStream.GetField`, that allows to read the object into a "black box" and then obtain its fields using a reflection-type API, i.e. by passing field names to methods of `ObjectInputStream.GetField`. Suppose that in the old definition of class `Person` we had two `String` type fields `firstName` and `lastName`, which in the new class definition are replaced with a single field `fullName`, containing essentially a concatenation of the first two. We can then add a `readObject` method similar to one shown in Figure 4.8 to the new version of class `Person`.

```
private void readObject(ObjectInputStream ois)
      throws IOException, ClassNotFoundException {
  ObjectInputStream.GetField gf = ois.readFields();

  // Check just in case if we have the new version of Person in the stream
  fullName = (String) gf.get("fullName", null);

  if (fullName == null) {
    // This is the old version. Calculate fullName.
    String lastName = (String) gf.get("lastName", null);
    String firstName = (String) gf.get("firstName", null);
    fullName = lastName + ", " + firstName;
  }

  ... // Set other fields
}
```

Figure 4.8: An example of method performing conversion in Java Serialization

The main disadvantages of the reflection API is that the code employing it is cumbersome and can not be statically type-checked. Reflection is also likely to slow down conversion, since each access to a field of the old object requires a lookup in some kind of internal dictionary.

We have tested the behaviour of this mechanism if an object being converted refers to another object, also belonging to a changed class. In the above example, it can be done e.g. by adding a field `Person next` to class `Person`. If we then add the following line

```
Person p = (Person) gf.get("next", null);
```

to the above code, it turns out that the conversion mechanism calls the `readObject()` method recursively and returns the converted instance of `Person` in this statement. Therefore the conversion method cannot obtain instances in the old format except the current one, or any data from these instances that is specific to their old format. Such a mechanism, though it is type safe, reduces the capabilities of conversion and may

cause scalability problems. For example, if it is used to convert instances organised in a structure such as a linked list, it will inevitably require a chain of recursive calls to set all the `next` fields, which may lead to VM runtime stack exhausting and application crash.

4.9.2 Commercial OODBMS

In this section we survey object conversion facilities provided in a number of contemporary commercial systems, and one (actually half-commercial, half-experimental) past system (O_2).

4.9.2.1 Objectivity/DB

Evolution support and object conversion facilities in the commercial OODBMS Objectivity/DB [Obj99a, Obj99b, Obj99d, Obj99c], that supports C++, Java and Smalltalk bindings, are the most sophisticated among the systems that we managed to survey. Changes to classes allowed by this system can be arbitrary. Objectivity supports three kinds of object conversion: eager (immediate in their terminology), lazy (deferred) and what is called on-demand, which basically means eagerly triggering previously defined lazy conversion on selected subsets of evolving objects at convenient times. Both default conversion and custom conversion functions are supported.

However, there are a lot of constraints imposed on either kind of conversion. Custom lazy conversion in Objectivity/DB is allowed to set only primitive fields of objects. The documentation does not explain the reasons for this constraint, nor does it say whether it is verified in any way by the system or what the consequences of breaking it can be. In addition, when lazy conversion is applied to an evolving object, the new shape of the latter is made permanent only if the object is accessed in an update transaction. If an object is accessed in a read-only transaction, the converted object will not be saved, and conversion will be repeated next time the object is accessed. This can lead to a significant overhead due to repeating conversions of the same object in an application which mostly reads data, e.g. a geographical information system, and that's where the manual suggests using on-demand conversion. Sequential "overlapping" lazy conversions are highly discouraged — it is recommended that all of the evolving objects are first converted, using e.g. "on demand" method, before the next lazy conversion is applied to the database. Finally, lazy conversion can not be combined with class changes that affect the class hiearchy, e.g. moving persistent-enabled[5] classes up or down the inheritance graph or deleting a persistence-capable[6] class.

Custom conversion code in Objectivity/DB can be written in C++, but there is no sign in the documentation that custom conversion is supported in the Java binding. In C++, the application programmer has to use a kind of custom reflection API to access fields of *both* old and new copies of an evolving object. The documentation does not say anything about calling methods of evolving objects. It also recommends limiting the use of conversion functions to objects being converted, that is, not access other persistent objects from within a conversion function. Thus, complex conversion is not supported or at least discouraged.

[5] *Persistent- enabled* class, in the terminology of Objectivity and other OODBMS, is a class whose objects can't be persistent, but which can read and write persistent objects.

[6] *Persistent-capable* class is a class whose instances can persist, and which can read and write persistent objects.

4.9.2.2 GemStone/J

The commercial system GemStone/J [Gem98, Gem00] is based on the Java language, and many of its features are similar to the existing features of PJama. Its evolution facilities are, however, relatively limited. Access to them is mostly through API calls. Although GemStone/J supports concurrent access of multiple VMs to the same store, class evolution cannot be reliably performed in concurrent fashion. The manual recommends termination of all applications before starting transformation, and shutting down and restarting of the server VM after it is finished. Obvious complexity of implementation of the shared object cache mechanism that provides concurrent access to the same store, probably justifies these limitations.

Classes representing objects in GemStone/J database can be evolved arbitrarily, as in PJama. However, classes themselves (i.e. methods and static variables) are not saved in the database. Instead, essentially descriptors of their instances are saved. Therefore, very little consistency validation or safety checks can be performed.

As far as instance conversion (transformation, in GemStone's terminology) is concerned, the flexibility is quite limited. No user-defined conversion functions are available. The only way of converting data is to specify mapping between old and new data fields. The programmer should create an object of the `java.util.Dictionary` standard Java class containing pairs of objects of the `java.reflect.Field` class. These pairs correspond to fields of old and new classes that should be copied between old and new instances. The programmer passes the map, together with old and new class objects, to the special method that creates a specification to transform the old class to new class. A method is also available, that creates a specification in the default way, where values of the fields with the same name and compatible types are preserved between versions. However, when this method is used, the information from deleted fields is lost and new fields are assigned equal default values. The transformation itself is performed eagerly, by calling yet another method. Many operations before and after it, such as loading new classes, should be performed manually.

4.9.2.3 Versant Developer Suite

Versant Developer Suite [Ver00] is a commercial OODBMS offering multiple language bindings, that include C/C++ and Java. Its evolution facilities allow any changes to classes except adding and dropping non-leaf classes to/from the class hiearchy. It supports both lazy conversion and a variant of eager conversion. However, neither of them are sophisticated. Lazy conversion can perform only default transformations. As for the eager conversion, it has to be a combination of the following steps:

1. Having class `C` which we want to change such that the format of its instances changes, first create a new class `D` with the same definition. Then run a program which would create an instance of `D` for each instance of `C`, copy information between them and "repair" all references to instances of `C` from other objects, so that they now point to the respective instances of `D`. All these operations should be encoded "manually" by the developer.
2. Delete old class `C` — this would also delete all of its instances.
3. Create a new class `C` with new definition. Run a program which would create an instance of `C` for each instance of `D`, copy information from the former instance to the latter and perform the necessary

transformations, and "repair" all references to instances of `D` from other objects, so that they now point to the respective instances of `C`.

4. Delete class `D` and all of its instances.

Thus, there is practically no dedicated support of the conversion process: the developer has to implement a one-off evolution system on each occasion. Such a technology looks very cumbersome, error-prone and hardly scalable.

4.9.2.4 POET Object Server Suite

POET [POE00] is another commercial OODBMS, also offering C++ and Java bindings. Its evolution support is based on explicit maintenance of class versions. This means that every time a class is added or class definition is changed, the developer has to register the new class definition in the *class dictionary*, which is a secondary database of classes that accompanies the "normal" database containing data. However, again, not the classes themselves, but essentially descriptors of their instances, along with the timestamp to distinguish class versions, are saved in the class dictionary. Objects are always accessed using the latest class version. Changes to classes can be arbitrary, but not all of them give the developer equal freedom in choosing conversion type (see below).

Object conversion can be performed in two ways. In the first scenario, an object is physically stored in the format corresponding to its creation time class version. However, it can be read and written using the latest class version — in that case default on-the-fly conversion is performed every time the object is accessed. In the second scenario, all objects can be eagerly converted to the latest format using an database administration tool or an API call. There is a category of changes to classes that can be handled only by using eager conversion, i.e. a database would not open after such a change is made if eager conversion has not been performed. The changes that require eager conversion are called *complex* and *very complex* in POET's terminology. Those that are not implementation-specific (e.g. related to database indexing) are changes to class hierarchy, such as class deletion or insertion.

Conversion capabilities are limited to default functionality. There seems to be a possibility for changing the default values assigned to added fields (by overriding a callback method called during eager conversion), but the documentation is inconsistent in this respect, so we were unable to find exact information on this subject. Data contained in deleted fields is inevitably lost during conversion.

4.9.2.5 ObjectStore PSE

ObjectStore PSE (Personal Storage Edition) [Exc99] from Excelon Corporation (formerly known as Object-Design, Inc.), is a persistent storage engine for Java. It is available in two variants (what would otherwise be called "editions"), named PSE and PSE Pro. As if to increase the confusion with naming, there is also a third product from this family, called Java Interface to ObjectStore Development Client and officially referred to as "ObjectStore". The latter is essentially a collection of Java APIs supposed to be used with

third-party databases. In the following discussion we consider ObjectStore PSE/PSE Pro only and refer to them collectively as "PSE".

PSE and PSE Pro are designed for single-user applications and provide transparent persistence in much the same way as PJama does. The difference between the PSE editions is in the maximum database size, degree of concurrency, support for recovery and disk garbage collection (available in PSE Pro only), etc.

Schema evolution and conversion support in PSE is quite limited. The developer has to use Java Object Serialization ([Sun00j], see also Section 4.9.1) to handle evolution. The manual suggests that the developer dumps the database into a file (serialised byte stream), modifies the classes and then re-creates the database. For database to be serializable, all of the classes in it should implement `java.io.Serializable` interface. Fortunately, adding an implemented interface to a class, as well as certain other changes, such as changes to methods, static fields, transient fields, and adding classes to hierarchy (the documentation does not explain whether only leaf classes are meant, or non-leaf classes can be added as well, as far as this does not change the format of instances of subclasses), do not require database rebuilding.

The first, and very serious problem with the above approach to conversion, is that it does not scale. A database which is to be evolved should be small enough to fit into heap space. Otherwise the documentation suggests to "customise the code that dumps and loads the database". This can be very hard, if, say, the whole database contains a single graph structure with arbitrary connections between nodes. Then, it may not be possible to change all of the classes in the database to implement `java.io.Serializable`, since for some classes, e.g. Java core classes or third-party libraries, the source code may not be available. Finally, as was shown in Section 4.9.1, the conversion facilities in JOS are not very sophisticated. To perform any non-trivial conversions, the developer has to rely on various reflection mechanisms very heavily. This makes conversion code cumbersome, difficult to read and error-prone, since little or no type checking can be done at compilation time.

4.9.2.6 O_2

One of the most sophisticated evolution support technologies we have discovered was once developed for the O_2 [FFM+95, FL96] OODBMS. Its primary mode of use was with applications in C and C++. Any schema modifications except adding or dropping non-leaf classes could be performed either incrementally, using primitives such as adding or deleting attributes to a class specified in a *change definition language*, or by redefining a class as a whole in the schema definition language. Structural consistency of changes was verified by the schema compiler.

Conversion of instances could be performed eagerly or lazily. Both default conversion and user-defined conversion functions were available. Migration of objects of a deleted class to its subclass was supported. Conversion functions (an example taken from [FFM+95] was presented in Figure 4.1) used a special syntax where the type for the old version of evolving class was defined "in place". In Section 4.4 we expressed our concerns about such a syntax.

Versioning has also been implemented for O_2 [FL96], making it the only system to support both adaptational and schema versioning approaches to schema evolution. This means that the schema and the underlying database could either be transformed completely, or new schema versions could be created and coexist si-

multaneously with the older ones. In the latter case, the identity of an object whose class had been evolved remained the same, but there was a separate physical record for such an object, corresponding to each class version. To maintain consistency between these records, the programmer had to provide a *forward conversion function* and a *backward conversion function* for each version of the evolved class. These functions were pieces of code responsible for updating a field in one representation whenever the corresponding field in the other representation was updated. For example, the field `temperature` in the old version of a class may encode temperature in Fahrenheit and in the new version in Celsius. The corresponding conversion functions containing the formulae to convert the temperature would have been invoked automatically every time this field is updated in either representation.

4.9.3 Experimental OODBMS

A number of experimental OODBMS that have evolution facilities were implemented in the last decade. An exhaustive review of them can be found in Rashid's PhD thesis [Ras00]. The latter, however, concentrates mostly on the higher-level aspects of their evolution models and does not tell much about the technical way of implementing instance conversion (updating). When we studied these systems, e.g. Orion [BKKK87, Ban87], F2 [AJEFL95, AJL98], CLOSQL [MS92, MS93, Mon93], Odberg's approach [Odb95], etc., we found that they generally do not provide non-trivial programmer-defined object transformations. Some of the systems use only default conversion or its slightly modified variants (F2), for some others the examples of conversion code are hard to find or understand. Where examples of conversion (*update/backdate*, *mutation propagation*, etc. are essentially the synonyms of conversion) code are presented (CLOSQL, Odberg's work), the code can access only the fields (attributes) of the copies of the current evolving (versioned) object. In Odberg's work, where explanations are the most clear and to which we therefore refer, a separate *mutation propagation function* should be defined for each attribute, say `x`, of one class version, which is different in another class version. This function executes in the context of the other class version. This means that it can access all attributes of the other version, but can set only attribute `x` in the object of class version for which it is defined. In Figure 4.9 we provide an example of such a pair of functions.

This approach has a number of serious limitations. A conversion function can not access any objects outside a single version of the current evolving object, and it can only return a single value. The first constraint limits the possibilites for more general database reorganisation, e.g. converting a field containing the head of a list of objects to an array of this list's elements. We found that such changes are sometimes required in practice. The second constraint, as we suspect, may in certain cases result in a large amount of redundant code, since different attributes may be computed using similar code.

Looking at the practical side of the things, mutation propagation functions are just pieces of C++ code, so to compile them in the correct context, the evolution system would have to extract them out of the above change specification file, combine them with some additional code and pass them to a C++ compiler.

The potential importance of being able to combine schema evolution with more general database reorganisation has been properly recognised only in one work [LH90]. A system called OTGen (Object Transformer Generator) is described in this paper, which assists the database administrator in understanding the effects of changes to class definitions, provides the default transformations, and provides the mechanism to allow overriding of the default transformations. The latter is performed using a tabular notation, i.e. special tables which are first initialised by OTGen to carry out default transformations. These tables are relatively

```
// Initial class version defined in Data Definition Language
CLASS Person {
  string name;
  int age;
  int height;    // Expressed in cm
  ...            // Method headers, etc.
}

// Changed class Person, defined in Change Specification Language
CLASS Person {
  REM height;      // Attribute height is removed
  ADD int meter;   // And replaced with two attributes that
  ADD int cm;      // directly correspond to it

  // backward and forward mutation propagation functions
  // (each function is in square brackets)
  height [return meter*100 + cm]
   <->
  meter [return floor(height/100)]
  cm [return height - 100*floor(height/100)]
}
```

Figure 4.9: An example of conversion code from Odberg's work

expressive: they allow the administrator to assign a number of different values, listed below, to a field in the converted object. A constant value different from the default one can be assigned; a context-dependent transformation, i.e. an assignment depending on the result of a boolean expression, can be made; a value can be copied from an arbitrary field of the corresponding unconverted object, or from a field of an object reachable from the unconverted one; a new object can be created and a reference to it assigned to a field in the converted object; and finally it is possible to specify that such a new object is to be shared between multiple converted objects. Still, this table-driven conversion allows the developers to move and copy data only between the old and new copies of an evolving objects, and objects reachable from the old one through a chain of fields. Methods can not be executed during such a conversion, and we understand that no real multi-step computations that transform information, can be performed.

4.10 Future Work

Several directions of future work on object conversion support can be envisaged, and they are presented in the following subsections.

4.10.1 Dictionary-Assisted Conversion

At present conversion in PJama is supported in two ways: the primitive default object conversion, and the sophisticated custom conversion. These methods of conversion cover the broad range of changes that can be made. There are, however, two kinds of evolutionary change, which are not supported adequately by these mechanisms, since no real conversion of data contained in instances is required, but default conversion does not work either. This happens when:

1. Instance data fields of an evolving class are renamed without changing their types;
2. The type of a field is changed such that implicit type cast can not be performed. For a primitive data field it happens when precision can be lost, e.g. when type `int` is changed to type `byte`. For a class type field it happens when a new type is not a super-type of the old type, e.g. type `java.lang.Object` is changed to type `java.lang.String`.

Presently in both of these cases the developer has to write custom conversion code in Java, which simply performs field assignments combined with explicit type casts. Execution of Java code, however, slows down conversion. For this reason, it might be worthwhile having an additional conversion option, something similar to dictionaries used in Gemstone/J (see Section 4.9.2.2), which, given a simple field mapping, would provide fast automatic conversion. This option was not implemented in PJama primarily because it is not very interesting research. However, we think that in a commercial system it would be useful. This option (at least for the case of simple field renaming, where no runtime errors due to runtime type incompatibility can happen) can also be used with lazy conversion, increasing its value if custom lazy conversion mechanism is not available.

4.10.2 Usability Aspects

The instance conversion technology that we have developed was so far used mostly by the author, and mostly in relatively small-scale experiments. Therefore, one of the most important directions of future work, if adoption of such a technology for more serious usage is considered, would be observation of independent professional developers using it over an extended period. This can help tease out conceptional, usability and engineering issues.

4.10.3 Concurrency and Scalability Issues

In Section 2.5.2 we have explained why lazy conversion is not implemented in PJama at present, and why we consider such a feature as *lazy custom* conversion quite controversial. However, a sensible alternative to it may be *eager concurrent* conversion — provided that an application(s) may temporarily abstain from using the objects that are being converted, or can tolerate blocking on access to an object which is not yet converted. It looks as if this research direction has not yet been explored.

There is also an issue of *conversion scalability*, which is quite important for eager conversion, and can be illustrated on an example of ObjectStore system (see Section 4.9.2.5). Objectivity/DB system (Section 4.9.2.1) also seems to scale poorly if eager conversion is used. The solution to this problem in PJama is discussed in Chapter 6.

4.11 Summary

In this chapter, we have described how conversion of persistent objects, that is, updating them to make them comply with the new definitions of their classes, is implemented in PJama. We explained what kinds of changes to classes result in the change of format of their instances, and thus require conversion (this is partially an implementation-dependent issue). We have then described two major kinds of conversion available in PJama: default (automatic) and custom (programmer-defined). It was explained what kinds of changes can be handled successfully by default conversion, that does not require any intervention from the developer. We then concentrated on describing our support for custom conversion, which is available in two flavours: bulk and fully-controlled. The issue of referencing old and new classes in conversion code, which is common to both of these sub-types of custom conversion, was explained, several alternatives were discussed, and our final solution was presented. The core of this solution is a convention about referring to old versions of evolving classes by specifically mangled names. The PJama system (both the compiler and the VM) recognises such names and treats them specially, loading the respective classes from the persistent store, whereas classes with ordinary names, representing new versions, are loaded from the CLASSPATH.

We then described a number of methods with predefined signatures, that can be used in bulk conversion. Methods are available both for the case when a class is changed, and when a class is deleted (then we are talking about *migrating* its "orphan" instances to other classes). Some of the presented methods are type unsafe, but in our opinion this unsafety is justified, which is confirmed by examples of evolutionary changes to classes.

Support for fully-controlled conversion, allowing the developers to convert instances of changed classes in non-default order and combine conversion with arbitrary data restructuring, was described then. Both bulk and fully-controlled conversion technologies were illustrated with a simple continuing example.

Relatively minor issues of static variable conversion and of access to non-public members were discussed, and were followed by the survey of the related work. The survey has shown that PJama's conversion facilities are the most advanced among the systems considered. It also looks as if PJama is presently the only system supporting custom instance conversion programming in Java. Other systems, supporting multiple language bindings, seem to allow non-trivial conversions to be encoded only in C++.

Considering the future work, it was shown, that it might be worth equipping PJama with a third type of conversion, called dictionary-assisted, which is intermediate between the default and the custom conversion. We have also pointed out that the value of our technology would be really proved only if it is evaluated by independent software developers over extended period of time. Finally, we mentioned concurrency and scalability issues as possible directions of future research.

Lack of powerful and convenient evolution facilities, which we have observed in most of the presently

available commercial OODBMS, may be one of the reasons why they have not yet achieved the same level of commercial success as relational DBMS. The capability of storing very complex data structures in a persistent object system should, in our opinion, be backed by the capability to evolve these structures in arbitrary ways, otherwise the usability of such a system degrades over time. We hope that our contribution to the research in evolution of persistent objects may help to stimulate adoption of more powerful solutions in commercial OODBMS.

Finally, we expect that a conversion technology similar to the one described in this chapter, can be also applicable in the context of run-time evolution for Java applications (see Chapter 7).

Chapter 5

Conversion Code Semantics and Internals

So far we were illustrating the conversion mechanism of PJama with very simple examples. However, when this mechanism is applied to some less trivial cases, e.g. when several classes are co-evolving in one atomic evolution step, evolving classes are used by other evolving classes, and so on — a number of additional issues become evident. The challenge here is to provide a conceptual framework which provides both freedom and safety, is readily comprehensible to the developer, and yet avoids commitments which over-extend the engineering requirements.

We describe these issues and the conceptual view of how they are resolved, in the following sections 5.1 - 5.4, and some important implementation details — in section 5.5. In Section 5.6 we demonstrate how we evaluated our conversion facilities. Section 5.7 presents an alternative design of conversion code support mechanism, that, in contrast with the one implemented in PJama, does not require changes to the Java language — at a price of providing somewhat less convenience to the developer.

5.1 Stability of the "Old" Object Graph during Conversion

An important feature of the conversion mechanism implemented in PJama is the stability of the source ("old") data. During conversion, newly-created instances are not automatically made reachable from any persistent data structure. "Old instance – new instance" reference pairs are kept in a special system table instead, and the source object graph remains unaffected. All class type fields of an existing persistent object will continue to point to the "old" instances throughout the conversion code execution, irrespective of whether the referenced instance has already been converted or not. Equally, when class type fields of freshly created "new" instances are initialized by the default conversion mechanism, they also point to "old" objects, as illustrated in Figure 5.1.

This stability is essential for comprehensible conversion semantics — the programmer always knows unambiguously which version of an object they are going to obtain by following a reference. Thus, Java's type safety is preserved. Two separate worlds of objects — the "old" and the "new" one, with only logical links between some objects belonging to them, are maintained until conversion is finished. Then, the "old" objects

are atomically "swapped" with their "new" counterparts, making the physical "old" instances unreachable and passing their identity to "new" objects. This has an effect of an instant "flip" that transforms the old object graph into the new (see Chapter 6 for the implementation details). The "old" instances can eventually be reclaimed by the disk garbage collector.

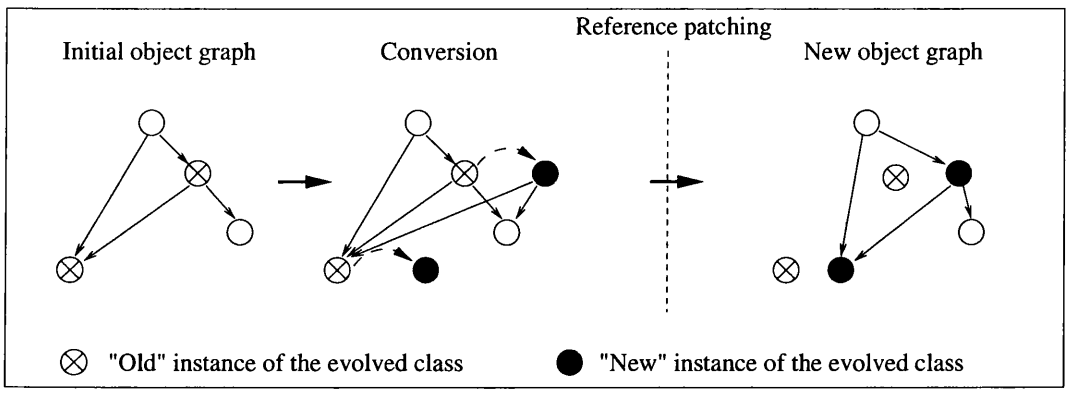

Figure 5.1: Management of references during conversion

The fact that the old object graph remains stable during conversion and is visible to conversion methods in its entirety gives the programmer free access to all of the data in the unconverted format at any moment during conversion. We believe that this is very important, since, for example, in a complex conversion the data from an object other than the current converted one might be needed, which is deleted from the latter object during conversion. Thus, our semantics allows developers to "move" data between different evolving objects during conversion, irrespective of the order in which these objects are transformed. It also allows conversion code to collect summaries, e.g. sets, sums or averages, iterating over objects in the "old world".

A new version for an "old" converted object, if it already exists, can be obtained using yet another method declared in the `PJEvolution` class called `getNewObjectVersion(Object)`. To illustrate this, let's again consider the example from the previous chapter, at a moment when the airline decides to divide its customers into three tiers. Imagine that there is a field `reference` of type `Customer` in both old and new versions of class `Customer`. This field points to a person that has once referred this customer to the airline. The airline decides that if the customer goes to the Gold Tier, then the one who has referred them gets bonus miles. The code that implements this logic is presented in Figure 5.2

Note that during conversion `reference` field of `oldC` variable continues to point to an instance of `Customer$$_old_ver_` irrespective of whether that particular instance has already been converted or not.

5.2 Loading Classes Referenced from Conversion Code

Conversion code is simply one or more normal Java classes. But a conversion class usually references classes with mangled names, such as `C$$_old_ver_`. Our modified Java compiler and the PJama VM in the evolution mode, both interpret the "old version" suffix specially. When the conversion class is resolved, every class that it references under a mangled name is loaded from the store, whereas a class with the same (modulo special suffix) name is loaded from the CLASSPATH.

```
public static Customer convertInstance(Customer$$_old_ver_ oldC) {
  Customer newC;
  if (oldC.totalMiles > 50000) {
    newC = new GoldTierCust();

    // Give bonus to the person who referred this customer
    Customer$$_old_ver_ ref = oldC.reference;  // It's old instance
    ref.totalMiles += BONUS_MILES;

    // See if "ref" has already been converted,
    // and if so, update its new version
    Customer refNew = (Customer) PJEvolution.getNewObjectVersion(ref);
    if (refNew != null)  // ref has already been converted
      refNew.totalMiles = ref.totalMiles;
  }
  ...
  return newC;
}
```

Figure 5.2: Referencing both "old" and "new" instances in the conversion code

However, the conversion class may reference many more classes than just those for which two versions are explicitly declared in the conversion code. Every class referenced from the conversion class, should be loaded either from the persistent store or from the file system (the latter is also called CLASSPATH). Since such classes may be evolving themselves (not being mentioned under a mangled name does not mean that a class is not evolving), it does matter where they are loaded from. It is crucial to define this unambiguously and such that it is intuitively clear to the application developer.

The above rule about classes that are referenced both under a mangled and an original name is intuitive enough. Naturally, if some class is referenced only under a mangled name, the programmer would expect it to be loaded from the persistent store. If some class referenced under a mangled name is not found in the store, the evolution system throws an error and stops at the change validation stage.

However, for classes that we would like to reference by "ordinary", non-mangled names, the situation does not look so unambiguous. Simple symmetry suggests that we load all classes with "normal" names from the CLASSPATH. But there is one complication here which is due to the "human factor". A piece of conversion code would almost always reference a number of classes that are not evolving. These are first of all *Java core classes* (see Section 2.5.3 for details of why we can't evolve these classes). In addition, many persistent application classes are usually involved, but do not evolve themselves, in a particular evolution transaction. To reduce the probability of errors and to make the conversion code less clogged with mangled names, we would like to avoid using mangled names for all these *stable* classes.

After careful consideration, we decided that stable classes can be named with ordinary, non-mangled names and the system should always load such classes from the persistent store. The primary argument for the second part of this decision is that a stable persistent class may contain static variables which were set during previous executions of the persistent application, and the values of these variables may be required

by the conversion code.

To determine if some class referenced from the conversion class is stable or not, we apply the standard change validation procedure described in Section 3.3 to all such classes at evolution time. In this case, we are interested just in the "same - not same" answer. Java core classes can be detected immediately by their names and do not require the change validation procedure.

Thus, now we can formulate the precise rules defining where classes that are referenced from the conversion class are loaded from:

1. If the class has an "old version" name, it is loaded from the persistent store.
2. Otherwise, if the class is a Java core class, an attempt is made to load it from the persistent store. If it is not found there, the class is loaded from the CLASSPATH.
3. Otherwise, if the class is a stable user class, it is loaded from the persistent store.
4. Otherwise, the class is loaded from the CLASSPATH.

5.3 Programmer's View of the Old Class Hierarchy

5.3.1 Uniform Renaming of Old World Classes

Evolving classes can be used by other classes. This fact, unfortunately, leads to a mismatch between the formal, declared type of a class type variable and its actual, runtime type. A simple example presented in Figure 5.3 illustrates this problem.

```
// Old version                    // New version
class LinkedListElement {         class LinkedListElement {
  LinkedListElement next;           LinkedListElement next;
  ...                               Object addedField;
                                    ...
}                                 }

// Conversion method
public static void convertInstance(LinkedListElement$$_old_ver_ ol,
                                   LinkedListElement nl) {
  LinkedListElement$$_old_ver_ ol_next = ol.next;  // Illegal?
  ...
}
```

Figure 5.3: An evolving class referencing an evolving class

What is the type of the field `next` of the old version of class `LinkedListElement`? We can only answer that it was formally defined as `LinkedListElement`, but when the above conversion code starts to run, its actual

runtime type should be `LinkedListElement$$_old_ver_`. This follows from our decision to maintain two separate worlds of old and evolved classes and instances during conversion, as explained in section 5.1. However, this contradicts the normal rules of the Java language. The above conversion code is what the programmer would expect to see, but it would not be compiled by the ordinary Java compiler, even if the latter loads classes with mangled names from the persistent store. Clearly, an extension to the Java language specification is needed to resolve this mismatch, and some additional modifications to the compiler and the VM are required.

```
// Old version of class Csuper
class Csuper {
  void m1() {...}
  void m2(Csuper cs) {...}
  ...
}

// Old version of class C
class C extends Csuper {
  C c;
  int i;
  boolean mb() {
    return D.m();
  }
  ...
}
```

```
// New version of class Csuper
class Csuper {
  void m2(Csuper cs) {...}
  void m3() {...}
  ...
}

// New version of class C
class C extends Csuper {
  C c;
  int i, isquare;
  boolean mb() {
    return D.m();
  }
  ...
}
```

```
// Conversion class
class ConvertC {
  public static void convertInstance(C$$_old_ver_ c_old, C c_new) {
    if (c_old.mb())
      c_old.m1();
    c_old.m2(c_old);
    c_new.m2(c_new);
    c_new.isquare = c_old.i * c_old.i;
    c_new.m3();
  }
}
```

Figure 5.4: An example of conversion code referencing old and new class versions implicitly.

There is also another source of similar type mismatches. So far we were talking just about "classes referenced by the conversion class". Now let us consider the kinds of these references at the Java source code level. We suggest dividing all references from the class source code to other classes into *explicit* and *implicit*. Explicit referencing means that a class is explicitly named somewhere in the Java code, as a type of a variable or a formal parameter, in an explicit type cast, etc. However, in addition to that a class typically references a number of classes implicitly. For example, if we mention the field of an instance of some class `inst.f` in the source code of class `C`, we may not name the type of `f` at all. However, the class representing the formal type of `f` will nevertheless be referenced from the compiled bytecode of `C`. If `C` references some

method defined in another class, simply as `inst.m()`, the reference to `m`'s defining class will necessarily be included in `C`'s bytecode. Needless to say, some classes can be referenced both explicitly and implicitly.

In addition, a class can reference many classes recursively — via other classes.

It turns out that implicit links to classes can produce type mismatch problems during evolution. To illustrate them, consider the common case where several classes are being evolved in one go. These classes are often hierarchically related. Assume that class `C` and its superclass `Csuper` are being evolved simultaneously. Furthermore, `C` evolves such that its instances require conversion, whereas for `Csuper` this is not required (see Figure 5.4). But in the new version of class `Csuper` we delete an old method `m1()` and add a new one — `m3()`. These methods are inherited by the old and the new versions of class `C`, and we need to call each of them during conversion, as shown in Figure 5.4.

`ConvertC` class references class `Csuper` implicitly, because methods `m1()`, `m2()` and `m3()` of that class are referenced. But the conversion code presented above contains no hint that there are actually two versions of class `Csuper`. The unchanged Java compiler will look up the name of the superclass in classes `C$$_old_ver_` and `C`, find the name `Csuper` in both, and link both classes to physically the same superclass. The latter will be loaded from the CLASSPATH, since it is evolving and its name (no special suffix) does not suggest that it should be loaded from the store. The new version of `Csuper` does not define method `m1()`, therefore the compiler will issue an error message and stop. If `m1()` was defined in the new version of `Csuper`, but differently, the code would compile, but then would work incorrectly.

In the above example there is also a recursive reference from `ConvertC` to `D` — via the old version of class `C` that calls method `D.m()`. It would be natural to expect the old and new versions of `C` to call the appropriate versions of `D`.

```
// Old version of class Csuper
class Csuper$$_old_ver_ {
  void m1() {...}
  void m2(Csuper$$_old_ver_ cs) {...}
  ...
}

// Old version of class C
class C$$_old_ver_ extends Csuper$$_old_ver_ {
  C$$_old_ver_ c;
  int i;
  boolean mb() {
    return D$$_old_ver_.m();
  }
  ...
}
```

Figure 5.5: Old class versions definitions as viewed by the evolution system

To solve all the above problems, we have chosen the following strategy. We arrange that both during compilation and execution of the conversion code the illusion is maintained that *all* persistent classes (except

Java core classes) are given old-version names. Therefore, in the above example the application developer should write the conversion code as if the old classes were defined as in Figure 5.5

The classes are renamed *irrespective* of whether they are really evolving or not. This is because at the stage of conversion code compilation the compiler has not determined which classes are evolving. Thus to be on the safe side we assume that any class can be evolving. This means that every class loaded from the persistent store would have old-version name, at least at the compile time. Further details of how this is implemented are presented in section 5.5.2.

5.3.2 Supporting the Naming Arrangement for Stable Classes

The uniform renaming performed by the compiler, that was presented in the previous section, has one drawback — it conflicts with the naming arrangement for stable classes. In other words, in the conversion code, how can we denote a stable class S its ordinary, non-mangled name, as specified in Section 5.2, if type S is referenced from an old-version class? The example in Figure 5.6 illustrates this problem.

```
// Old version of class E as perceived by the programmer
// Class S is unchanged (stable) in reality
public class E$$_old_ver_ {
  S$$_old_ver_ sField;

  S$$_old_ver_ retSTypeResult() {
  ...
  }
}

// Conversion method
public static void convertInstance(E$$_old_ver_ e_old, E e_new) {
  S s = e_old.retSTypeResult();  // Illegal?
  ...
}
```

Figure 5.6: An example of referencing a stable class in the conversion code

The actual runtime class of both sides of the assignment expression in the conversion method will be the version of class S loaded from the store. This is achieved by the mechanism described in Section 5.2, after verifying that S has not changed. However, at the compile time the compiler doesn't know if S is evolving or stable, and assumes that it is evolving[1]. Consequently, at the compile time the type for the left-hand side of the above expression will be S, that is, the version of class S loaded from the CLASSPATH, and the type for

[1]In principle, it could have been possible to re-organise the evolution process, so that compilation of conversion class could be performed only in combination with checking all evolvable classes and then evolving the store. Thus it would have been known precisely which classes have been changed and which are stable. Further, evolution happening immediately after that would not give the developer a chance to change more classes and thus make conversion code incorrect. However, this would require further changes to the compiler and make it very tightly connected with the evolution tool. Our present solution, in our opinion, is more general and thus, we believe, more optimal than this one.

the right-hand side will be class S$$_old_ver_— the version of class S loaded from the store. The formal types, therefore, will be different and incompatible.

The solution to this problem, as well as to a number of others, is the provision of *extended implicit type casts*, described in the next section. The compile-time types in expressions similar to the above remain different, but they are made compatible (only one way, however — from the old class to the new one). This allows the above code to compile successfully. At run time only one version of class S exists — that is the version loaded from the store, and its real name is not mangled. How this is achieved is described in the section 5.5.3. As a result, if we add the additional Java statements shown in Figure 5.7 after the first line in the body of the above conversion method, they will produce the output presented in the same figure.

```
System.out.println("The class of sField is " + e_old.sField.getClass());
s = new S();
System.out.println("The class of s is " + s.getClass());
System.out.println("These classes are equal? : " +
                   e_old.sField.getClass().equals(s.getClass()));


The class of sField is S
The class of s is S
These classes are equal? : true
```

Figure 5.7: Addition to the conversion method in Figure 5.6 and the resulting output

5.4 Extended Implicit Type Casts

The practice of writing conversion code has shown that sometimes the programmer needs to explicitly assign a value of an old-version class to a variable of the new-version class type. A classic example is evolving class C which has a field of type D (D is also evolving), and this field is renamed in the new version. If we have two versions of class C as presented in Figure 5.8, the default conversion will not copy the value between dfield and d_field because of the different names of these fields. Thus we have to do it explicitly, in the conversion method for class C. What we would like to be able to write is the conversion code presented in the same figure.

The previous section has explained that both the compile-time types and the runtime classes of these variables are different, and under ordinary Java rules they are not assignment compatible. We should, however, mention one possibility to perform such an assignment. The actual instance to which c_old.dfield refers at the moment when the above method is called could have been already converted or not. So, if we really want, we can determine this using the API of the org.opj.utilities.PJEvolution class (see section 5.1). Furthermore, if it hasn't been converted yet, we can explicitly call the conversion method for class D, get the new object for c_old.dfield and finally assign it to c_new.d_field. But all this is rather complex and inconvenient and works only if there is a conversion method for class D. So, ironically, if D is stable, which is not known at the moment of compilation, there would be no workaround and the code would not compile. What we really need is, therefore, the compiler and interpreter support for appropriate type casts in cases similar to the above.

```
// Old version of class C          // New version of class C
class C {                          class C {
  D dfield;                          D d_field;
  ...                              ...
}                                  }

public static void convertInstance(C$$_old_ver_ c_old, C c_new) {
  c_new.d_field = c_old.dfield;
  ...
}
```

Figure 5.8: Example where explicit "old-to-new" assignment is required, and desirable conversion code

Such a support has been implemented, so now the above conversion code is absolutely valid. It is, however, unsafe by definition. The compiler and the interpreter allow the programmer to assign to the variable a value of a type which can have a totally incompatible instance format. If class `D` is evolving, the value of `c_old.dfield` assigned to `c_new.d_field` will *always* be a reference to the instance of the old class, because our non-ambiguous conversion code semantics requires this. It will be changed to the reference to a "new" class instance only after all instances of all evolved classes have been converted. Therefore, if at any moment the conversion code tries to interpret `c_new.d_field` according to its formal definition, i.e. as a value of a new version of class `D`, it is likely to cause an error, which can't be prevented by the Java interpreter. The safety of the Java language is violated here rather explicitly and brutally. It is, however, not a bigger violation than the one which we would get automatically if the field `dfield` retained its name in the new class, and thus the default conversion mechanism copied the reference to the "old" instance into it.

The rules for extended implicit type casts make not only direct counterparts like `D` and `D$$_old_ver_` compatible, but also superclasses of evolved classes and their old counterparts, e.g. `Dsuper` and `D$$_old_ver_`. If in the assignment "`a = b`" both `A` and `B` are "normal" classes, the standard Java language procedure checks if either `A` and `B` are the same (trivial case), or if `B` is a subclass of `A`. We have extended this check in the following way: if `B` is found to be an old-version class (has a mangled name), then its new counterpart `Bnew` is looked up (if `B` is actually to be deleted, a transient counterpart is looked for its superclass, and so on). Then the standard validation procedure is performed for classes `A` and `Bnew`.

Coercions leading to lack of type safety, that we have introduced, are in conflict with the general design of Java, and we consider this unfortunate. In Section 5.7 we present an alternative design, which is free from most of the above problems, but at a price of decreased convenience for the developer, more cumbersome conversion code and probably less effective implementation. In the summary for this chapter we present further thoughts on this subject.

5.5 Implementation Issues

This section presents the most important (and not immediately obvious) issues in the implementation of the evolutionary instance conversion and class replacement mechanisms for PJama. Section 5.5.1 explains how the system implements fetching of old-version classes with mangled names such that name mangling

does not have undesirable effects on the normal persistent application execution. Section 5.5.2 describes the compiler and runtime mechanisms that support the illusion that all old classes have old-version mangled names, and load all involved classes from correct sources. The next section explains how we arrange that stable classes are always loaded from the persistent store, no matter what kind of reference from what (old or new) class is followed. Finally, in Section 5.5.4 some issues related to using the Java class loader mechanism during conversion are explained.

5.5.1 Fetching Classes with Mangled Names

At present the PJama evolution technology supports only immediate, or eager object conversion. This, in particular, means that two versions of an evolving class should exist only during the limited time interval, while an evolution transaction is in progress. Within that interval the VM can be queried for a class with an old-version name at any moment. After transaction termination, there should be no accessible classes with old-version names in the store.

We modified the Persistent Class Directory code to handle mangled class names. This data structure maps the names of classes to the respective persistent class objects, and the PJama VM always queries it first when it resolves a symbolic reference to a class. PCD treats mangled class names specially only if the PJama VM is in the "evolution mode", i.e. a special internal flag is set. This happens if our persistent build tool is running over a non-empty persistent store. In all other cases the PCD code will simply try to look up a class with the given name, without any specific name interpretation.

In order to prevent propagation of mangled class names into the store, the name of the class is never mangled in the array of class name strings contained in the PCD. Instead, the PCD dynamically interprets the given class name every time it is queried. Only the class name string contained in the main-memory copy of the class object is mangled. Class names contained in class objects are not subject to the PJama VM update tracking mechanism, therefore a mangled name can never propagate into the persistent store.

5.5.2 Implementation of Uniform Renaming of Old Classes

As we explained in Section 5.3.1, the modified Java compiler in the evolution mode gives the programmer the illusion that all persistent classes (except Java core classes) have old-version mangled names. To maintain such an illusion by eagerly renaming all persistent classes everywhere in the store would obviously be too expensive. Instead, on-demand renaming of classes actually involved in the given evolution is performed. We will now discuss how this is implemented.

As any Java class, a compiled conversion class references other classes symbolically via its *constant pool.* In Section 3.3.2 we have presented the constant pool structure and explained that a class can be referenced from the constant pool of another class both *directly* and *indirectly.* That is, essentially a class name can be a separate entry in the constant pool of another class, or can be only included into an entry corresponding to a signature of a member of this or another class. In order to maintain consistency, we should achieve a state where a name of an old-version class, wherever in the constant pool of a conversion class it is referenced, is always mangled. Looking at the example from Section 5.3.1 again, we observe that we should, in particular,

create two separate entries in the conversion class constant pool for class `Csuper`. One entry with the name `Csuper$$_old_ver_` represents an old class, that defines method `m1()`, and also `m2()` applicable to `c_old` variable. Another entry, with `Csuper` name, should represent a class defining the method `m3()`.

In order to support uniform renaming of old classes, the compiler was modified in the following way. During parsing of a conversion class, as soon as the compiler comes across a class name for the first time, it loads this class. Normally any class is loaded from the CLASSPATH, however in our implementation a class with a mangled name is loaded from the store. The compiler then marks this class as an old-version one. Subsequently, whenever a symbolic entry from the constant pool of such a class is requested, the compiler immediately mangles all class names in this constant pool entry (excluding the names of Java core classes) and returns the result. So, uniform class renaming happens lazily. The compiler code that requested a constant pool entry, will load all classes referenced in this entry from the store, because of the mangled names. The process will continue recursively. Thus at compile time every class reached from the explicit old-version class will be also old-version-named and loaded from the store. On the other hand, every class with non-mangled name will be loaded from the file system.

This uniform and consistent renaming allows the compiler to load classes from correct sources from wherever they are referenced, and to have precisely defined ("old" or "new") class type for every field, formal parameter or local variable involved in the conversion code.

When the conversion class `ConvertC` is compiled, the mangled symbolic entries will be copied into its constant pool. As a result, in our example the constant pool of `ConvertC` will contain, in particular, the following symbolic entries: class names `C`, `C$$_old_ver_`, `Csuper`, `Csuper$$_old_ver_`; method signatures `m1()` with a reference to `Csuper$$_old_ver_` as a defining class, `m2(Csuper$$_old_ver_)` with the same defining class, `m2(Csuper)` and `m3()` with `Csuperas` a defining class.

At conversion run time, during conversion class resolution, all classes that the latter references will be loaded either from the store or from the CLASSPATH according to the rules given in Section 5.2, i.e. on the basis of their names plus their actual stable or evolved state. In turn, classes reachable from these classes will be loaded from the same source, i.e. either a persistent store (for old-version persistent classes) or the CLASSPATH (for the new-version classes). However, stable classes will always be loaded from the store (see next section). Thus the described technique of uniform class name mangling guarantees that during evolution, irrespective of the length of the path the given class is reached from a known "old" or "new" class, it will be loaded from the correct source.

To complete the picture, we have to mention that signatures of fields and methods that include mangled class names, such as `m2(Csuper$$_old_ver_)`, need additional attention at run time, during conversion code linking, when the respective *field* and *method blocks* are looked up in the actual persistent classes. Field and method blocks are essentially records containing signatures and other information and grouped into tables [LY99]. Every class has its own tables of field and method blocks. Lookup in these tables is performed by symbolic names. When a persistent class is loaded, these tables are created by the standard VM mechanisms, and the names that field and method blocks are assigned at that time are never mangled. Therefore to be able to link mangled signatures (in the conversion class constant pool) with appropriate blocks (in the actual referenced classes), we had to patch the field and method block lookup routines of the VM to take the special suffix into account. A similar modification has been performed on the Java bytecode verifier to prevent it from issuing "Field not found" and "Method not found" error messages when it comes across mangled signatures.

5.5.3 Management of Stable Classes

As mentioned in Section 5.3.2, at conversion run time we want only one version for every stable class to be visible to the conversion code. It should be the version loaded from the persistent store, and it should have a non-mangled name. However, at conversion code compile time it is not known which classes are stable. Therefore class renaming described in the previous section is applied to all classes referenced from classes with explicitly mangled names. One consequence of this is that the compiler may create two symbolic references for a stable class in the conversion class constant pool — one with an original and one with a mangled name. In the example given in Figure 5.6, the conversion class will have references to both `S` and `S$$_old_ver_` class names in its constant pool. When `opjb` is invoked, all class type references from a conversion class are resolved eagerly by our change validation code. As a result, the VM will load two identical, but physically separate versions of class `S` — one from the CLASSPATH and another from the store, and will create two physical pointers to them from the conversion class. We can't prevent creation of these two class copies, since for some other technical reasons (actually an obscure problem in the JVM itself that we can't fix) resolution of references from the conversion class must happen before any classes are checked and possibly found stable.

Two copies of a stable class are also created if we have an evolving class which references a stable class, and this reference is resolved in the old and the new copies of this class before we have information on stable/evolving status of the referenced class. Unlike the case of conversion classes, references from evolving classes are generally not resolved eagerly. However, a reference from a class to its superclass is always resolved when the class is loaded into the VM memory, and some other references may become resolved as a side effect of reflection operations performed during substitutability checks.

Therefore, if we have a stable class `S`, then, for various reasons, by the time all classes are checked and determined to be stable or changed, there can exist a "wrong" copy of class `S`, loaded from the CLASSPATH, and referenced by conversion classes and evolving classes. All these references should be replaced with a reference to the persistent copy of `S`. Measures should also be taken to prevent loading stable classes from the CLASSPATH during conversion code execution, and to prevent mangling the names of stable classes. Here is the list of all stable class management operations performed by the evolution system after substitutability checks are complete and before conversion starts:

1. The pointers from the constant pool of each conversion class to transient versions of stable classes are switched to their counterpart persistent versions.
2. The pointers from new versions of evolving classes to transient versions of stable classes are switched to their counterpart persistent versions.
3. Transient versions of stable classes are deleted from internal JVM class tables to prevent the VM from accidental linking them to evolving classes.
4. The class names inside the stable class objects loaded from the store are changed back to the original, non-mangled form.
5. Further attempts to load stable class objects from the CLASSPATH, e.g. during resolution of new versions of evolving classes when conversion code is executed, are prevented. This is achieved by making our custom *class loader* (see next section) that normally loads classes from the CLASSPATH, aware of stable classes, and making it load them from the store.

5.5.4 Dealing with Multiple Class Loaders

Every Java class loaded into the VM has two class loaders associated with it: the *initiating class loader* and the *actual class loader* (see [LY99], Chapter 5). The initiating class loader is the one that initiated loading of this class. However, it may delegate class loading to another class loader. For example, when class `String` is loaded by an application class loader, the resulting class in fact has `NULL`[2] as its class loader. The application class loader will be its initiating class loader, and `NULL` will be its actual class loader. Technically it means that the class object will be placed into that application (initiating) class loader's class cache, but will have `NULL` in its `classloader` field.

For application classes loaded in the ordinary way (i.e. not using any custom class loader) the initiating and the actual class loaders are always the same single dedicated class loader, called the Application Class Loader. In PJama we currently can evolve only such classes, the main reason for this being the implementation difficulties of dealing with multiple "incarnations" of the same-named (but maybe not physically the same) class, where each "incarnation" is a class object loaded by a separate class loader. It is also not quite clear what the operation semantics should be in this case, e.g. whether all such classes should be evolved simultaneously or not.

Any JVM maintains the constraint that there should not exist two or more classes with the same name and the same actual class loader, as well as no two classes with the same name and initiating class loader, but different actual class loaders.

When a class is promoted into the store, the class loader that it references directly, i.e. its actual class loader, is also promoted. On the other hand, its class loader's class cache, essentially an array of class objects present in every class loader object, is not used in PJama at all, so no link back from the class loader to the class is saved into the store. This patch was initially introduced to avoid problems in other parts of the PJama VM, but as we will see, it also simplifies evolution.

When we evolve classes, we load all of the substitute classes from the CLASSPATH with our own class loader. The fact that the substitute classes may have the same names as the original ones, but always a different class loader, helps to manipulate all of these classes within the VM memory easily. However, when the system replaces the original classes in the store, the substitute classes should be assigned the same "original" class loader to be compatible with all other non-evolved persistent classes when the persistent application resumes. We do not need to do any class replacements in class caches of class loaders, since these caches are not persistent.

Care has to be taken over when to patch class loaders in the new versions of evolved classes. If we do this in the VM memory, before promoting a class into the store (which is simple to implement), we will not be able to run conversion code after that. This is because it will reveal that we violated the JVM internal constraint that does not allow two physically separate classes with the same name and (now) the same class loader to exist in the VM memory simultaneously. However, we need to be able to promote an evolved class and then continue conversion, because this is required by our scalable conversion mechanism implemented in Sphere (see Chapter 6), which reads objects from the store, converts them and promotes the new objects back into the store one partition at a time. Therefore, patching the class loader of an evolved class is done at a low

[2] `NULL` denotes a special *bootstrap class loader*, which is not a Java object, but effectively a piece of internal JVM code that is used to load the minimum necessary set of classes (including class `java.lang.ClassLoader` itself) upon JVM startup.

level, in the promotion routine of the PJama VM, so that the classes in the store are patched as they are promoted, but not changed in the main memory. During conversion, at the VM level different versions of an evolving class appear to have different class loaders. However, the new versions of classes already promoted into the store, reference the original, persistent class loader. After conversion finishes, the VM is shut down. Next time it is resumed to run a persistent application, it finds a set of classes with the same persistent class loader in the store.

It is worth noting that so far we were implying that there is only one "original" class loader for all changing classes. This constraint holds in persistent applications that don't have user-defined class loaders. In such applications user classes have the same application class loader as both originating and actual class loader. Some Java core classes (which we can't evolve so far anyway) may have a `NULL` bootstrap classloader. Otherwise, i.e. if user-defined class loaders exist, both change validation and class substitution will become much more complex, because there may be more than one persistent class with the same name in the store, etc. The validation part of the PJama evolution technology has some partially implemented provision for supporting that. But so far everything works on the assumption that all classes that are received by the substitution code within one evolution have the same original class loader.

5.6 Evaluation

During development and testing of the PJama's evolution facilities, we have created more than 20 regression tests. Most of these tests exercise various aspects of object conversion, since this is the most innovative, and also technically the most difficult area. The tests were created to evaluate and validate the new concepts, and as response to discovered bugs, to prevent their re-appearance in future. Typically, each bug discovered after initial testing would have resulted in a new regression test or in adding theF code that would exercise this bug in one of the existing tests.

All of the tests are synthetic, essentially to allow their automatic execution and validation inside a general test suite which was developed for PJama. However, we also used a real-life application called GAP (Geographical Application of Persistence), which is being developed at the University of Glasgow by a succession of student projects (see e.g. [Jap00]). GAP is a relatively large system, currently consisting of more than 100 classes, of which about 30 are persistent. The size of the main persistent store containing the map of the UK as polylines composed into geographic features, is nearly 700MB. It will also load US Bureaux of Census TIGER geographical data. This gave us larger examples, e.g. the State of California maps occupy 2.75GB. So far the application's facilities focus mainly on displaying maps at various scales (with user-controlled presentation) and finding requested objects. The author made a number of improvements to this system, using the evolution facilities of PJama whenever possible. This lead to discovering of a number of bugs, provoked several new ideas, and inspired a number of regression tests that imitate data structures and evolution steps in GAP.

In Appendix B one of the regression tests from the actual test suite of PJama, that imitates some aspects of GAP, is presented. We hope that it can serve as an illustration to this and previous chapters.

At present we are concerned by the fact that our concepts have not been evaluated by really independent software developers. Such an evaluation would be very valuable.

5.7 Future Work — an Alternative Design Investigation

The need for a convenient instance conversion mechanism for the Java language is not unique to PJama platform. A similar mechanism would be desirable for any persistence solution for Java, e.g. Java Serialization, OODBMS with Java binding, or Java runtime evolution. However, our present mechanism of instance conversion, more precisely, its part that deals with class renaming, is unlikely to be universally adopted. The reason is that this mechanism introduces extensions to the Java language, and, therefore, requires modification of the Java compiler and the core of the JVM (e.g. the code that verifies class files and checks compatibility of types in assignments at run time). Despite all of the useful features that these language extensions bring, Sun Microsystems, which defines the standard for the Java language, is reluctant to adopt such extensions. People who work on the Java specification believe that the language is already overloaded and further extensions should be adopted only if the demand for them is really compelling and there are no conventional alternatives to them in the present Java implementation [Bra00]. At least the first part of this requirement is probably not true about instance conversion, at least presently. Therefore it makes sense to think about a different mechanism, that might be less convenient, but does not require changes to Java and as such can be easier to implement and adopt for various platforms.

5.7.1 Changes to Java with the Present Mechanism

The changes to the Java language that our present conversion mechanism introduces are:

1. A special way of loading classes with specifically mangled names. A mangled name signals to the compiler and then the VM to load a class with the respective ordinary name from the persistent store, instead of the normal CLASSPATH, and then to mangle its name.
2. A mechanism for lazy uniform renaming of old classes, described in Sections 5.3.1 and 5.5.2.
3. Extension of the rules for implicit type casts of the Java language, described in Section 5.4.
4. Special management of stable classes, described in Sections 5.3.2 and 5.5.3.

We have also introduced type unsafety to Java, which contradicts with the language specification. Type unsafe assignments can be explicitly made, since they are allowed under item 3 above. In addition, the default conversion mechanism combined with our treatment of two separate object worlds (Section 5.1) is another source of type unsafety. For example, assume that we have evolving classes `C` and `D` and the conversion method for `C`, shown in Figure 5.9. When conversion method is called, the runtime class of the value that has just been automatically assigned to `c_new.d_field` variable by the default conversion, will be `D$$_old_ver_`. Consequently, the assignment in the conversion code will yield an incorrect result, with the compiler unable to recognise the problem.

A solution that has more chance to be widely adopted than our present one, should avoid any of the listed changes to the Java language and, ideally, should eliminate the above type unsafety problem. Yet it would be very desirable to retain the main feature of our present mechanism, that is, refer to old classes by mangled names instead of using reflection. One such solution is presented in the next section.

```
class C {      // Common part of the old and new versions
  D d_field;   // of the evolving class C
  ...
}

class D {  // old version          class D {  // new version
  int x;                            double y;
  ...                               ...
}                                  }

void convertInstance(C$$_old_ver_ c_old, C c_new) {
  double y = c_new.d_field.y;  // Compiles, but wrong value at runtime!
  ...
}
```

Figure 5.9: Example of type unsafety in conversion code.

5.7.2 A Solution That Does not Change the Java Language

5.7.2.1 Class Naming and Stable Class Issues

The main difference between our present mechanism and the proposed one is that in the latter, class renaming at compilation time should be performed eagerly and statically. Before conversion class compilation, copies of all of the old classes should be automatically extracted from the persistent store and renamed (mangled). Class file names themselves, and the names of all old persistent classes in any context inside these class files, should be eagerly mangled. After such a preprocessing, the old classes should be placed onto the standard CLASSPATH, where their new versions are already residing. Therefore, a standard unchanged Java compiler can pick them up and thus compile a conversion class.

At run time, we should take care of class renaming and of binding the existing (persistent) instances to classes with mangled names. This can be done in different ways depending on the concrete context:

- In an OODBMS with multiple possible language bindings, or in case of Java Serialization, i.e. when Java class objects are not stored in the database (serialized stream), we can just re-use the mangled classes located on the CLASSPATH.
- In PJama or a similar persistent platform, we can use a mechanism similar to the present one (described in Section 5.5.1), that dynamically renames every class loaded from the store during conversion code execution. It should also preprocesses each class file, mangling symbolic references to other classes inside it once and forever (in contrast with the present mechanism).
- In case of runtime evolution, all old classes are already in memory and all objects are bound to them. We suspect that we will have to rename all of these classes eagerly rather than lazily, e.g. because of various internal naming consistency checks and assertions that the VM might have. In presence of such assertions, we might be unable to have some old classes renamed and some not, as is the case with lazy renaming.

There is a certain tension in this model considering the treatment of stable classes. If we eliminate their special handling and mangle all extracted persistent classes absolutely uniformly, the result will be that during conversion we may get two versions for each stable class — one reached via an old class and loaded from the store, and another reached via a new class and loaded from the CLASSPATH. These classes, however, are internally equivalent. Presence of two differently named copies of the same class will result in formal incompatibility between existing persistent instances of this class and new instances created during conversion (extended implicit type casts, that are used in our present model to eliminate this problem, contradict with the Java language specification and should not be used in the new model). If these new instances are then promoted into the store, it will lead to promotion of the transient copy of the stable class, thus creating two persistent copies of the same unchanged class — unless some very special measures are taken.

Therefore, it seems more reasonable to combine extraction and mangling of old class versions with their preliminary analysis. The latter would determine which classes have actually been modified compared to the new counterparts, and it is important that the situation does not change by the time the conversion starts (perhaps the system should verify that). Having the information on changed classes, we avoid mangling the names of stable classes in references from old class versions, and, therefore, will need only one (persistent) copy of each stable class during conversion code compilation and run time. However, these copies of stable classes can contain references to old versions of evolving classes, which we will necessarily have to mangle. Thus, stable classes will not look "entirely stable" anymore, since some of their fields, etc. may have explicitly old-version types, and the developer will have to take that into account when writing conversion code. The system will also have to take care about restoring the original, non-mangled version of a stable class on the CLASSPATH once conversion is complete. But otherwise, this model of stable class handling looks feasible.

Since extended type casts mechanism is eliminated, something has to be done in order to handle cases such as the one described in Section 5.4, i.e. to support copying the contents of a renamed field of evolving type, between the old and the converted object. A mechanism of conversion dictionaries that was proposed in Section 4.10.1 can be used in this situation, if such a transformation should be performed in the same way for all instances of the evolving class. However, sometimes more flexibility may be required. That is, for some evolving instances, the value should just be copied from the old field to the new one, and for some others it should be, say, copied from a different field or set to `null`. To handle this (probably quite infrequent) case, a special reflection API, which is aware of correspondence between old and new types, can be used. To make it more type safe, actual assignments that it performs can be deferred until the conversion method returns, so that inside the conversion method it would not be possible to have an object field holding a value of an incompatible class type.

5.7.2.2 Type Unsafety Issues

The design presented above solves all of the "practical" problems of flexible conversion without the need to change the Java language, the compiler and the interpreter part of the JVM. It will allow the developers to achieve the same results as with the present mechanism, at a price of some increase in the complexity of the conversion code. E.g. look at the functionally equivalent pieces of conversion code presented in Figure 5.10.

```
// Old version                    // New version
class LinkedListElement {         class LinkedListElement {
  LinkedListElement nxt;            LinkedListElement next;
  ...                               Object addedField;
                                    ...
}                                 }

// Conversion method in the old model
public static void convertInstance(LinkedListElement$$_old_ver_ ol,
                                   LinkedListElement nl) {
  LinkedListElement$$_old_ver_ ol_next = ol.nxt;
  if (isValidElement(ol_next))
    nl.next = ol_next;
  else
    nl.next = null;
  ...
}

// Conversion method in the new model
public static void convertInstance(LinkedListElement$$_old_ver_ ol,
                                   LinkedListElement nl) {
  LinkedListElement$$_old_ver_ ol_next = ol.nxt;
  if (isValidElement(ol_next)) {
    try {
      PJEvolution.set_field(nl, "next", ol_next); // Calling reflection
    } catch (Exception e) {
      ...
    }
  } else
    nl.next = null;
  ...
}
```

Figure 5.10: Conversion code in old and new models.

The only remaining problem with this design is that it does not eliminate type unsafety in cases similar to the one that was presented in Figure 5.9. This unsafety arises in two cases:

1. Default conversion assignments to the fields of the converted object, that are performed before the conversion method is called for this object.
2. Conversion code operating on one evolving object and trying to access the new copy of another evolving object.

It may well be that the best solution will be a "pragmatic" one, in which these problems are simply ignored. After all, conversion is always a non-trivial operation, which is supposed to be performed by a developer

who understands very well the implications of what they are doing. Some security mechanism could verify that the agent provoking the evolution is acceptable.

However, to eliminate at least the safety breach numbered 1 above, we can change the way the default part of conversion is performed during custom conversion. Instead of copying the fields with the same names and compatible types *before* the conversion method is called, we can do it *after* that, at least for fields of non-primitive types. There is one implication here, which is due to the fact that the conversion method may override some of the default assignments. If on entry into the conversion method, all fields in the "new" object have `null` values, and on the exit some have non-`null`, the default conversion code may itself assign the default values only to those fields which are `null`. But what happens if the programmer wants to explicitly assign the `null` value to some field?

To solve this problem, the conversion mechanism may, instead of `null`, assign all of the class type fields in the new copy of the evolving object to special "dummy" objects. These may be objects of the respective class type, just one instance of each class for the whole evolution transaction, created only once without using a constructor (to avoid any side effects) and destroyed in the end. On the exit from a conversion method, only the fields referring to such known objects will be subject to default assignments. Assignments using the conversion dictionary and the "deferred assignment" reflection mechanism will be performed at the same time.

It remains not entirely clear how to deal with the "loophole" numbered 2 above (and whether it is worth dealing with it at all). Default conversion assignments and dictionary assignments to the fields of the new object can be deferred until all objects have been explicitly converted. This will lead to a second pass over all evolving objects — which we effectively have in PJama already (see Chapter 6), but which may not necessarily be practical in other systems. Then, it is not a problem to defer until very late default and dictionary-assisted assignments, since they are performed in the same way for all instances. But what to do with reflection-assisted assignments, which can vary from one instance to another? If we want to defer them too, we will have to store the information about such assignments for each instance, probably causing a substantial memory overhead. Alternatively, it may be worth thinking about implementing a totally different technique, which will analyse the bytecodes of the conversion method and forbid operations such as access to evolving-type fields of the new copies of evolving objects.

5.8 Summary

In this chapter we have first explained the semantics underlying execution of PJama conversion code. According to this semantics, the complete and unchanged view of the "old object world" is provided by the system to the developer until all objects are converted. We have then explained the issues related to loading of old versions of changed classes during conversion, mangling their names, and handling classes that do not evolve (stable classes), for which only one version should exist during conversion. These issues have been resolved by introducing the mechanism of lazy name mangling and by extending the Java implicit type casting rules to allow casts between the versions of evolving classes.

The above type casts, and also assignments that in some cases can be made by default conversion when it is combined with custom conversion, are type unsafe. As such, these extensions contradict with the

specification of the Java language, break one of the cornerstones of its ideology, and require changes to a Java compiler and to the core of the JVM. On the other hand, they provide substantial convenience to the developer who writes conversion code. Nevertheless, we realise that the above negative properties make the wide adoption of these extensions to Java unlikely. For this reason, we have also proposed an alternative design of instance conversion mechanism for Java, in which most of the unsafety problems are eliminated. This, however, is achieved at a price of making conversion code somewhat harder to write and more cumbersome. Furthermore, eliminating some "last bits" of type unsafety may require substantial increase of evolution mechanism design complexity.

In general, class evolution and language type safety seem to be fundamentally in conflict. On the other hand, the very process of co-ordinated code and data evolution is outside the scope of semantic definitions for Java and most other languages. Given the presence of long-lived and long-running systems that inevitably need to evolve, this appears to be an issue for language designers to consider further.

Chapter 6

Store-Level Support for Evolution

High-level support for evolution in a persistent object system — that is, language extensions and VM runtime mechanisms — are only a part of the evolution story. The main reason why special store-level support is also needed, is that a persistent object system can typically store many more objects in the persistent store than can fit into a computer's main memory, even virtual memory. This makes effective (in terms of memory and disk space required and performance) and safe conversion implementation an issue, since:

- We support (only) eager object conversion, so all evolving objects should be converted in one go, independent of their total size;
- Conversion process should be safe, i.e. if a Java runtime error or a system crash occurs mid-way, objects must not be left in half-transformed, and thus effectively unusable state. Furthermore, unless all of the objects have already been converted on the JVM level, the process should be rolled back (not forward) by the system recovery mechanism, since a runtime error can manifest a deeper semantic problem, and eventually a change to the developer-supplied conversion procedure may be required.

There are also some less significant, but still important concerns, which will be explained throughout this chapter. Fortunately, the present persistent store sub-system of PJama called Sphere was designed from the beginning with many of these concerns in mind. Therefore, when it came to store-level implementation of evolution support, our work was significantly facilitated. Still, Sphere is a quite complex system; therefore to speed-up the work, a large part of it was performed by Craig Hamilton, who by that time had worked on Sphere for two years. The author is deeply grateful to him, since without his help this work would definitely have taken much more time and effort. Craig and Tony Printezis, the main designer of Sphere, were also involved in obtaining the experimental performance results presented in this chapter.

The structure of this chapter is as follows. Section 6.1 presents an overview of Sphere, with the emphasis on its features important for evolution. Section 6.2 formulates precisely the goals that we wanted to achieve. The next section considers several variants of store-level evolution algorithms, explains the reasons for choosing the present algorithm and discusses its tradeoffs. Finally, in Section 6.4, the initial performance results are presented.

Parts of the work in this chapter have been previously published as papers [DH00, ADHP00].

6.1 Sphere Persistent Store Overview

The architecture of Sphere is detailed in [PAD98, Pri99, Pri00]. This system, among other merits, has a number of features that facilitate evolution. These features are listed below:

- The store is divided into *partitions*. This was introduced to permit incremental store management algorithms, such as disk garbage collection and evolution. Partitions can be moved and resized within the store, while retaining their identity. Objects can be moved within partitions without changing their identity, i.e. *PID*s.
- An object *PID* in Sphere is logical. This means that instead of a physical offset of the object in the store (as it used to be in PJama Classic), a *PID* contains a combination of the logical ID of the store partition where the object resides, and the index of the object's entry in the *indirectory* of this partition (Figure 6.1).

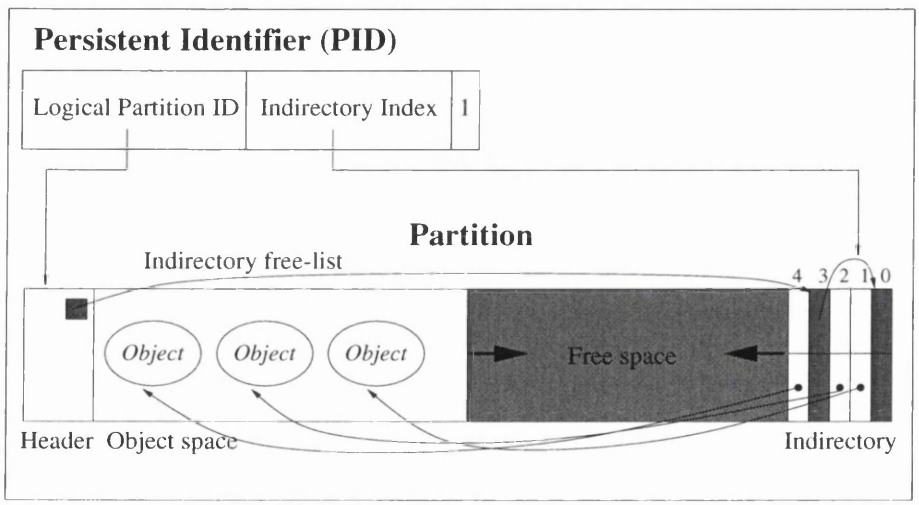

Figure 6.1: *PID* format and partition organisation.

- A single partition can contain only certain *kinds* of objects. For example, instances and large scalar arrays are always kept in separate partitions. This can greatly speed up the lookup of evolving objects, since partitions containing unsuitable object kinds simply do not need to be inspected.
- Each Sphere object that corresponds to a Java instance holds a reference to a *descriptor*. Descriptors are internally used objects that describe the layout (e.g. location of pointer-type fields) of objects of the same type. A descriptor is lazily replicated in each partition that contains objects that need to be described by it. If one or more instances of the same class reside within a partition, then a copy of their class's descriptor must be present in that partition. Conversely, if no instances of a class exist in a partition, no descriptor for that class will be present. There is a descriptor index in each partition.

Using these properties of descriptors we can rapidly discover whether any instances of a class exist within a partition, without the need to maintain exact class extents.

- Sphere is fully transactional. If a computation that runs on behalf of some transaction modifies the contents of the store and then crashes before committing the updates, the store layer will automatically perform *recovery* on restart, returning its contents to the initial state, as they were before the failed transaction started. Transactional behaviour is achieved by *logging* operations which change the contents of the store (see [Ham99, MHL+92]).

With such a rich infrastructure available to us, we could set up ambitious goals for our store-level evolution support mechanism. These goals are presented in the next section.

6.2 Goals for the Evolution Platform

Due to Sphere's descriptor mechanism, replacement of classes in the persistent store is not a problem. All objects reference a class via its descriptor, so to successfully replace a class in the store it is enough to promote its new version and then to patch each descriptor (there may be no more than one descriptor for a given class in a partition) to make it point to this new version. Since each partition contains an index of all descriptors residing in it, descriptor lookup is very fast. Furthermore, if a class has changed insignificantly, e.g. just method code has been modified, whereas the all of the other data structures representing this class remain unchanged, then we can apply an optimisation. We simply promote the new bytecode array into the store and patch the pointer to the bytecodes in the class object (see Section 2.1.3.4 for details of disk class object representation details). The *PID* of the class object remains the same then, and we don't need to patch any descriptors.

Therefore, the main concern about evolution support on the persistent store level is the implementation of object conversion mechanism. We have formulated our requirements to this mechanism as follows:

1. It should be scalable. Here we will define scalability as a combination of two sub-requirements. First, the growth of the number of evolving persistent objects should not cause a proportional (or, ideally, any) growth of the amount of main memory required to perform evolution successfully. Second, the performance (time to evolve a single object) should not noticeably depend on the total number of evolving objects.
2. It should be safe (atomic). That is, any failure at any stage of evolution, whether caused by an exception in the programmer-defined conversion code or an OS/hardware problem, must leave the store in either its original or its final (evolved) state.
3. It should be complete and general-purpose, i.e. support the semantic properties of conversion described in Section 5.1.

The next section explains why the above requirements presented a challenge in the context of PJama, and our solutions.

6.3 The Store-Level Evolution Algorithm

6.3.1 Terminology

Throughout this chapter we will use the following terminology. We refer to the persistent objects that are undergoing evolution as *evolving* objects. During the execution of conversion methods and for some period thereafter, these objects exist simultaneously in their old form, *old object*, and their new, converted form - *converted object*. An unpredictable number of *new* objects can also be produced by the conversion code, e.g. if we replace an integer field `date` in an evolving object with a new instance of class `Date`.

6.3.2 Temporary Storage for Converted Objects

The fundamental consequence of the requirement 3 in the previous section is that the store-level evolution mechanism (*evolution foundation*, as we will denote it for short) is not allowed to replace any old objects in the store with their converted counterparts until all objects have been evolved. In other words, whenever an object with a certain *PID* is requested from the store, and this object is an evolving object, the old object should be returned. Therefore we have to keep both old and converted objects alive and accessible (though not necessarily resident in main memory) until the end of the evolution transaction.

This presents a challenge to the evolution foundation. The simplest solution — to keep converted objects in main memory until all evolving objects are converted, and then to put them into the store in place of old objects — does not scale, since it does not satisfy the requirement 1 in the previous section. We will now consider the alternatives.

Two places suitable for temporary storage of converted objects can be envisaged: some kind of custom virtual memory and the persistent store itself. The first variant, however, can be ruled out almost immediately, the main reason being that Sphere stores, and thus the space that evolving objects may occupy, can grow to sizes much larger than the virtual memory limit on the 32-bit hardware, which is most popular at present.

If we try the second approach, we can sketch the following general design. Evolving objects are processed in portions small enough to fit into the main memory. As soon as all of the objects in that portion are processed, the converted objects are saved (promoted) into the store. Both the old and the converted objects can then be evicted from the main memory, either immediately or lazily, as further portions of objects are processed. This is repeated until all of the evolving objects are converted, and has the effect of producing the two "object worlds" - the old and the new one. Then these worlds are *collapsed*: each old object is substituted by the corresponding converted one.

6.3.3 Associating and Replacing Old and Converted Objects

The presented design addresses the most fundamental issue of low-level evolution support — scalability. However, looking at it in more detail reveals a number of other issues. The first of them is the question of how to store the association between old and converted objects, which, when the objects are promoted

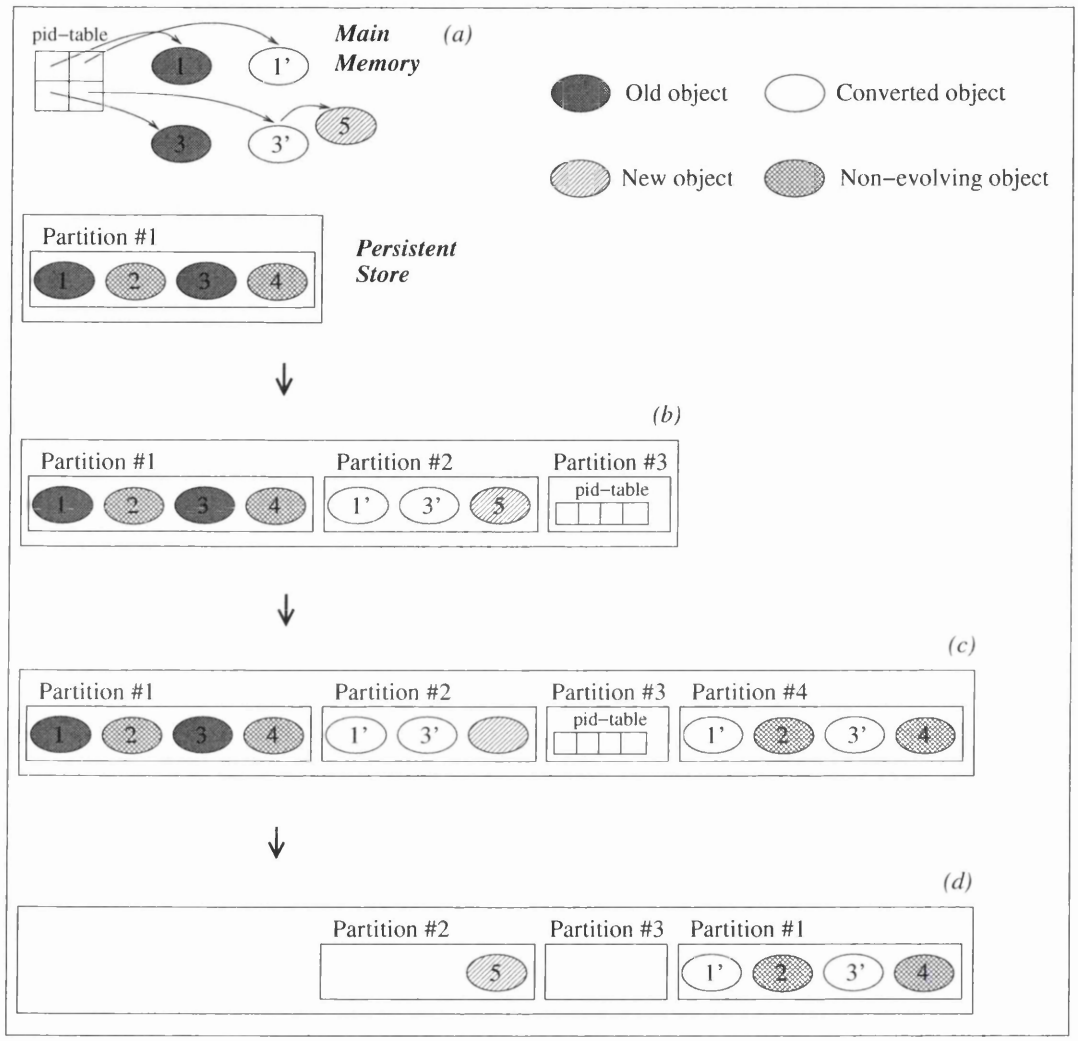

Figure 6.2: A hypothetical low-level conversion implementation using global *PID*-table

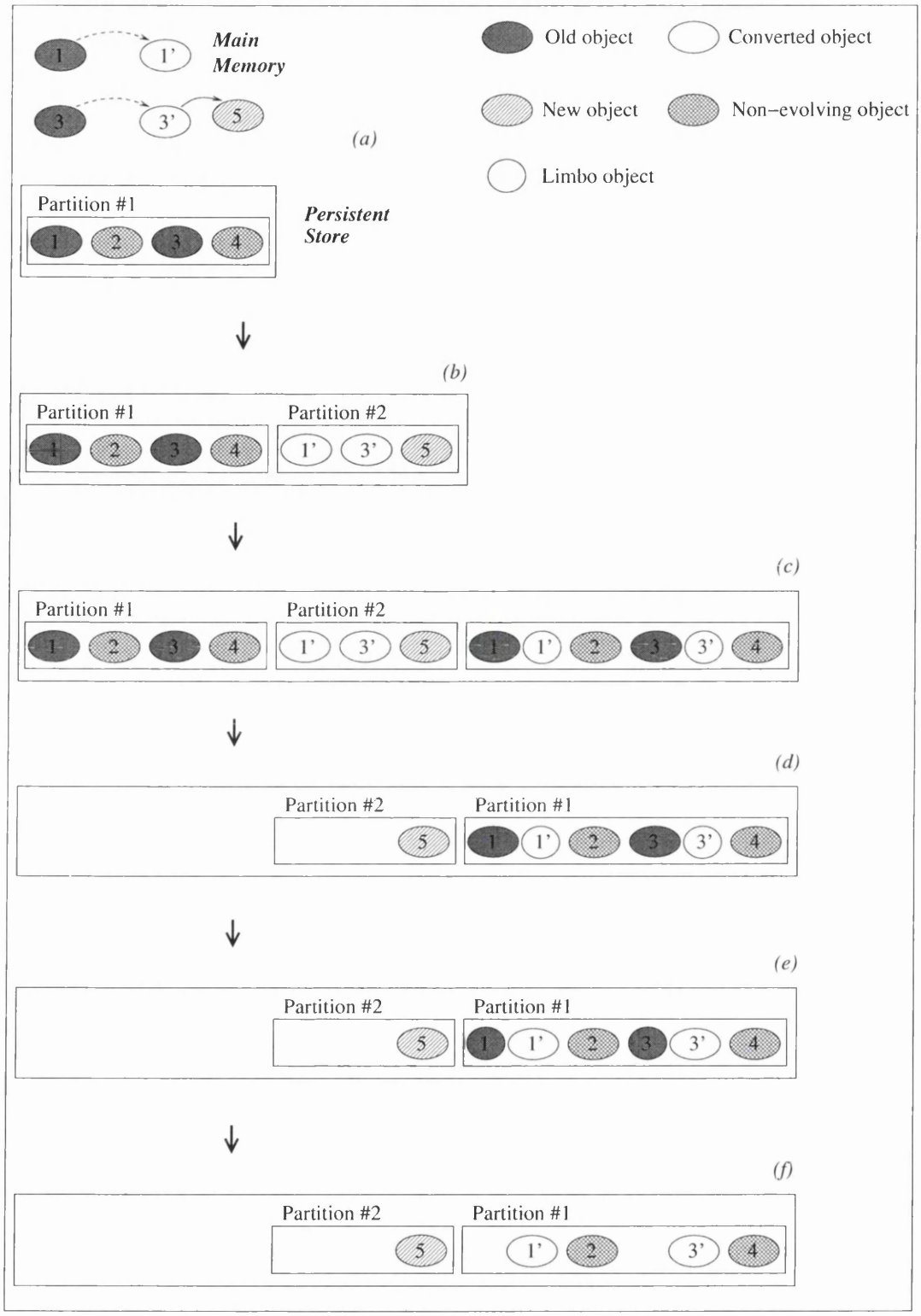

Figure 6.3: Low-level conversion implementation using limbo objects

into the store in a usual way, will be a table of "old *PID*, new *PID*" pairs. This association should be preserved throughout the conversion process, until collapsing starts, which needs this information. Once again, the simplest solution — a main memory table containing *PID* pairs (PID-*table*, as we will denote it), can, unfortunately, cause scalability problems. For example, assume that we need to convert 2^{29} objects (Sphere allows us to save more in a single store, but let us consider a realistic case when just a part, not all of the objects in the store, are being converted). An object *PID* is currently 32-bits, therefore a pair of *PID*s will occupy 8 bytes. So we will need 2^{32} bytes to store the *PID*-table — the whole virtual memory available on a 32-bit machine. This is unacceptable, so we need to think of how to save parts of this table to disk during conversion, or how to preserve the association between objects in some other way.

Saving parts of the *PID*-table to the store is possible if the table itself is made a persistent object. Further, the associated overhead can be made relatively low if this table is constructed as a linked list of arrays (sub-tables). Every time a portion of objects is converted and promoted into the store, the next sub-table is also promoted. The disadvantage of this approach is that the table itself occupies space in the store, which may be comparable to the space occupied by the evolving objects, as we have just seen. Managing this persistent table also increases log traffic.

There is also an issue of how to replace old objects with the respective converted ones in the store. The main problem here is that a converted object can be, and in practice almost always is, physically larger than the old object, since changes to classes tend to add, rather than delete information in them. Therefore it is generally not possible to overwrite an old object with a converted one, unless all objects that follow the old object are "shifted" within their partition. However, the size of the partition may be insufficient to accommodate the total difference in size between the converted and old objects. And to change partition size in Sphere, another partition would have to be created and the information from the old one copied to the new one. Therefore, the only way to substitute old objects with the converted ones will be to create a new partition of sufficient size and copy all objects from the old partition into it, replacing old versions of evolving objects with converted ones. Then the logical ID of the old partition should be transferred to the new one. After that, the old partition can be discarded. Even if a crash occurs at some stage of this process, it is always possible to resume partition transformation after system re-start, since all information needed to complete it successfully is in the persistent store.

The above algorithm is depicted in Figure 6.2, where the store containing a single partition with evolving objects is shown. During phase *(a)* old objects from the evolving partition are read into main memory, then the corresponding converted objects, one new object and a *PID*-table are created. During phase *(b)* all of this data is promoted into the persistent store. In phase *(c)* the contents of the original evolving partition are copied into a new partition, simultaneously replacing each old object with the corresponding converted one. Finally, in phase *(d)* the new partition takes the identity of the old one, and the old partition is discarded, as well as all temporary objects. Lack of space does not allow us to illustrate the algorithm on a store with two or more evolving partitions. However the reader should be aware, that none of the old partitions can be discarded until all objects are evolved, otherwise recoverability of the process will be impacted.

To summarise, the solution that involves a global persistent *PID*-table, solves the problem of scalability, but results in a significant overhead in the terms of persistent store usage. Additional space in the store is required for the *PID*-table. Additional temporary partitions should be allocated, which can not be released until evolution completes. And, finally, the number of *PID*s needed temporarily is equal to the number of evolving objects. For this reason, in the worst case we would be unable to convert a store that contains an amount of objects greater than a certain threshold, since we would simply run out of *PID*s.

Therefore a solution which we have eventually adopted does not involve a global *PID*-table and uses a different technique for associating old and converted objects. It is presented in the next section.

6.3.4 The Store-Level Instance Conversion Algorithm

To avoid both allocation of a new *PID* for each converted object and the need for a global *PID*-table, we decided to save converted objects in the *limbo* form. A limbo object does not have its own *PID* and physically resides immediately after the live object to which it corresponds. "Old object - converted object" pairs packed into "live object - limbo object" conglomerates consume no additional *PID*s and the lowest possible additional space, and exist until all of the evolving objects are converted. If a crash occurs in the middle of the evolution transaction, limbo objects are discarded automatically, since they are "visible" only to the evolution system. No additional actions to recover the system in this case are required. After all evolving objects are converted, objects in each "live-limbo" pair are swapped, so that each converted object becomes "live". Subsequently, old objects which now are marked as empty gaps, can be reclaimed by the disk garbage collector.

We will now present the detailed description of the main steps of store-level instance conversion:

1. *Preparation for Conversion: Marking phase* of Sphere's off-line cyclic garbage collector for the whole store (SGGC, see [Pri00]) is carried out. This ensures that all of the garbage objects are marked so that they will not participate in evolution and hence be resuscitated by being made reachable.

2. *Conversion of all of the Evolving Instances:* The store is traversed, one populated partition, `p`, at a time, visiting only partitions with relevant regimes (so that, for example, partitions containing only scalar arrays need not be visited). For each class `C` whose instances are evolving, `C` is looked up in `p`'s descriptor table. If no descriptors are found, we skip to the next partition. Conversely, if a descriptor for `C` is found, objects in `p` (partition #1 in Figure 6.3) are scanned sequentially. For each non-garbage evolving object, default and/or custom conversion is performed. The net effect of all transformation operations is that a converted instance is created, and possibly some new objects (Figure 6.3(a)). The "old instance *PID*, converted instance memory address" pair is recorded.

   When all of the evolving objects in `p` have been processed, all of the converted objects are promoted into a new partition (partition #2 in Figure 6.3(b)).

   After that the most important sub-phase of low-level conversion begins. Using a slightly customised copying garbage collector, the contents of `p` are copied into a completely new partition. Whenever the old copy of an evolving object is transferred, the corresponding converted object is fetched and placed directly after it in the limbo form (Figure 6.3(c)). On completion of this partition sweep, the logical ID of partition #1 is transferred to the new partition, and the original partition is returned to free space. Thus in the resulting partition (physically new partition with logical ID 1, Figure 6.3(d)), the old form of each evolving instance is directly referenced by its unchanged *PID*, and the limbo form resides in the bytes that follow it, no longer directly referenced by a *PID*. *Temporary converted* objects (objects $1'$ and $3'$ in partition #2) are marked free, so that the space in this partition can be re-used in future.

   If phase 3 does not occur, e.g. due to a crash, the limbo objects will be reclaimed by the next garbage collection. Hence the reachable store currently has not changed, it has only expanded to hold unreachable limbo objects not visible to applications, and new instances reachable only from them.

3. *Switch from the Old World to the New World:* At the end of the previous phase all of the new format data are in the store but unreachable. Up to this point recovery from failure, e.g. due to an unhandled exception thrown by a custom conversion function, would have rolled back the store to its original state. We now switch to a state where recovery will roll forward to complete the evolution, as, once we have started exposing the new world, we must expose all of it and hide the replaced parts of the old world.

   A small internal data set is written to the log, so that in the event of system or application failure, there is enough information to roll forward, thus completing evolution. This set essentially records all of the partitions that contain limbo objects. Each partition in this set is visited and all "live object - limbo object" pairs are swapped, i.e. the old object is made limbo (marked as re-usable space), and the converted limbo object is made live (Figure 6.3(e)). The next garbage collection will reclaim the re-usable space (Figure 6.3(f)).

   Once this scan has completed, the new world has been merged with the unchanged parts of the old world.

4. *Commit Evolution:* Release the locks taken to inhibit other use of the persistent store and write end-of-evolution record to the log.

6.3.5 Tradeoffs

The advantages of the above solution are the very low disk space and log traffic overheads. Indeed, use of limbo objects to arrange that both old and converted objects and their dependent data structures can co-exist in the store avoids writing images of evolving objects into the log. Also, only two extra partitions are needed to complete evolution, and a small number of extra *PID*s. The implementation of this functionality was relatively simple, since the code of the disk garbage collector was largely reused. Having these properties, the implementation also performs quite well (see Section 6.4).

However, the drawback of this solution is that during evolution, once a converted object is turned into limbo state, it becomes inaccessible. That's because such an object does not have its own *PID* and is not recognised as a first-class store object. On the PJama VM level, once a converted object is promoted into the store and made a limbo object, its *PID* (which is now meaningless) is discarded, and any updates to this object will not propagate into the store.

Our present experience shows that in most cases the conversion code does not need to revisit a converted copy of an evolving object after the latter is created. However, the conversion code will need to do this if, for example, during conversion we want to create a forward chain of objects.

The limbo object solution also does not give any advantages in the case of fully controlled conversion, where the order of faulting of evolving objects can be arbitrary. For this reason, our present implementation of fully controlled conversion, though operational, does not scale. In this conversion mode, all evolving objects are loaded into memory on demand. Only after developer's conversion code terminates, evolved objects are sorted by partitions and the standard conversion operation described above is applied to each group, sequentially.

A solution which permits revisiting the converted objects can be envisaged, that still exploits limbo objects.

For it, Sphere primitives that read and update a limbo object, given the *PID* of the corresponding live object, will be required. However, the main challenge will be to design and implement the management of limbo objects in the Java heap. Since in the PJama VM two objects can never have the same *PID*, limbo objects will have to be treated not as normal persistent objects. Probably a separate table that maps *PID*s to memory objects, equivalent to the main PJama ROT (Resident Object Table), will be required for them. The *PID*s of the corresponding live objects will be re-used for limbo objects. The garbage collection, eviction and update tracking/saving code of the PJama VM will have to be modified. But, provided that these changes are made in a comprehensible way and do not compromise the performance of ordinary PJama applications, this approach seems reasonable.

However, the problem of how to make fully controlled conversion scalable still remains. The only solution to it that we can currently see uses periodic saving of converted objects to the store, plus a persistent *PID*-table, essentially as described in the beginning of Section 6.3.3.

6.4 Initial Performance Evaluation

We have measured the time that it takes for our system to evolve a store with a certain number of evolving objects. The number of factors on which this may depend is large, so we concentrated on the following issues:

- Verify that the time grows linearly with the number of evolving objects.
- Explore the impact of non-evolving objects in the store on the performance, particularly when their number is much greater than that of those evolving.
- Explore how the complexity of the objects and of the conversion code affects the evolution time.
- Validate synthetic tests with some real-life applications.

6.4.1 Benchmark Organisation

Our benchmark consisted of three synthetic tests, the main properties of which are summarised in Table 6.1.

Test No.	Description	Objects	Change
1	Simple class	simple	simple
2	001 benchmark	complex	simple
3	001 benchmark	complex	complex

Table 6.1: Benchmark organisation

In all three tests we varied the number of evolving objects (denoted n) in the store between 20,000 and 200,000. The second varying parameter was the number of non-evolving objects per evolving object, denoted by g for *Gap*. This varied from 0 (all of the objects in the store are evolving) to 9 (9 non-evolving

objects per one evolving object). The objects were physically placed in the store such that evolving and non-evolving objects were interleaved. This is illustrated in Figure 6.4. From the evolution point of view, this is the worst possible store layout, since we have to scan and evolve all of the partitions.

All of the tests were run with a constant Java heap size of 24MB (that is the default size for PJama), which, in the worst case, is an order of magnitude less than the space occupied by all the evolving objects. The Sphere disk cache size was set to the default value of 8MB.

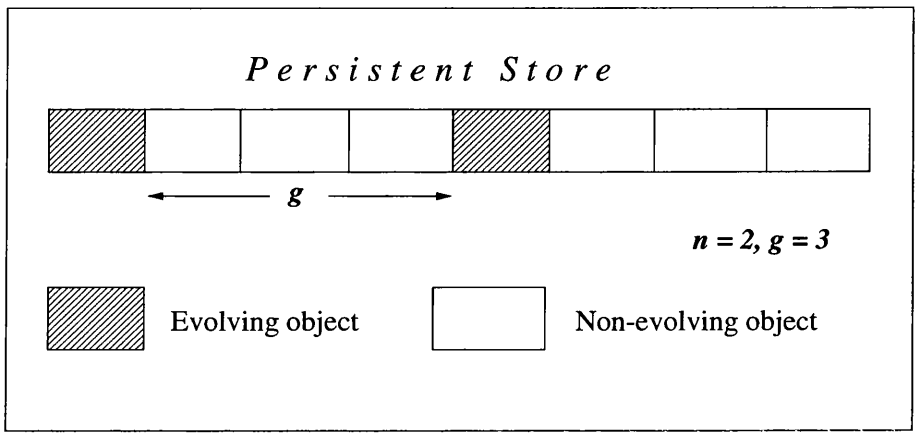

Figure 6.4: Test store layout example

In test 1 the old and the new versions of the evolving class were defined as presented in Figure 6.5. Default conversion was applied to instances of C. According to the rules of default conversion, the values of fields i and j were automatically copied between the old and the converted object, and the k field was initialized to 0.

```
// Old version          // New version
public class C {        public class C {
  int i, j;               int i, j;
                          int k;
...                     ...
}                       }
```

Figure 6.5: Old and new versions of the evolving class in test 1.

Tests 2 and 3 were performed over stores populated with instances of the class called Part from the adapted version of 001 benchmark, which we have taken from Chapter 19 of [ZCF+97]. The initial Java version of this class is presented in Figure 6.6. As defined in the 001 benchmark, an object of class Part contains a unique id and exactly three connections to other randomly selected parts. Instances referenced from the given Part instance p are stored in the p's to list. In turn, all instances that reference p in their to lists are stored in the p's from list, allowing a reverse traversal. The values of other fields are selected randomly from a given range.

In test 2 the only change in the new version of class Part was a different type of its id field — it was changed to long. Java types int and long are logically, but not physically compatible. This means that the values of the former type can be safely assigned to the fields of the latter, but the size of fields of these

```
import java.util.LinkedList;

public class Part {
  int id;
  String type;
  int x,y;
  long build;
  LinkedList to;
  LinkedList from;

  // Methods to set/get fields, etc.
}
```

Figure 6.6: The initial version of the evolving class in test 2.

types are different (32 bits and 64 bits respectively). So object conversion is required in this case, but default conversion is enough to handle information transfer correctly.

In test 3 a more complex change was applied: the type of `to` and `from` fields was changed to `java.util.Vector`. The objects contained in the list can't be copied into another data structure automatically, so the conversion class presented in Figure 6.7 was written. As a result of conversion, for each `Part` instance, two new objects are created, and six objects are discarded.

```
import java.util.LinkedList;
import java.util.Vector;

public class ConvertPart {
  public static void convertInstance(
                Part$$_old_ver_ partOld,
                Part partNew) {
    int toSize = partOld.to.size();
    partNew.to = new Vector(toSize);
    for (int i = 0; i < toSize; i++)
      partNew.to.add(partOld.to.get(i));

    int fromSize = partOld.from.size();
    partNew.from = new Vector(fromSize);
    for (int i = 0; i < fromSize; i++)
      partNew.from.add(partOld.from.get(i));
  }
}
```

Figure 6.7: Conversion class used in test 3.

In each test run we were invoking our build tool that analyses and recompiles classes and then initiates instance conversion. It was only the instance conversion phase which we measured. In each test we measured

the *Total Time*, defined as the time elapsed between the start and the end of conversion. We also measured the *Sphere Time*, defined as the time spent within the Sphere calls corresponding to step 2, part 2 and step 3 of the evolution algorithm (see Section 6.3.4). The difference between Total Time and Sphere Time was called the *Mutator Time*.

Every test run with the same values of n and g parameters was repeated ten times, and the average time value was calculated after discarding the worst case. All experiments were run on a lightly-loaded Sun Enterprise 450 server with four[1] 300MHz UltraSPARC-II CPUs [Sun00q], an UltraSCSI disk controller, and 2GB of main memory. The machine runs the Sun Solaris 7 operating system. The Sphere configuration included a single 1GB segment and a 150MB log. The store segment and the log resided on the same physical disk (9.1GB Fujitsu MAB3091, 7,200rpm [Fuj00]). Both the store and the log file were placed on the same disk as a pessimal arrangement to accentuate effects due to log writes.

6.4.2 The Experimental Results

In tests 1 and 2 we observe completely uniform behaviour, characterised by almost perfectly linear growth of the evolution time with both n and g. In test 1 the minimum and maximum total time values were 1.25 and 57.38 sec, whereas in test 2 they were 2.42 and 75.10 sec, respectively.

The 3-dimensional figures 6.8 and 6.9 show the total time taken during the evolution phase, with a further breakdown indicated at the extremes of both axes. For test1, Figure 6.10 shows the breakdown in more detail for a fixed $g = 0$, varying n. Figure 6.11 shows the same breakdown, this time for a fixed $n = 200,000$, varying g.

Graphs for all experiments at each value of n and g were generated, yielding the same typical set of results; namely that as the number of evolving objects increases (Object Gap decreases), less of the total time is spent within the Sphere kernel, than within the Mutator.

Linear growth of the time with n means that the scalability requirement for evolution technology is satisfied withing the range explored. Despite the fixed Java heap size, the time grows proportionally with the number of evolving objects.

The growth of evolution time proportionally with the object gap (this parameter can also be interpreted as the total number of objects in the store), illustrates a trade-off we have made. When a partition is evolved, all of the objects contained in it are transferred into a new partition, thus a comparable amount of time is spent handling both evolving and non-evolving objects. The alternatives are to explicitly maintain exact extents or to segregate classes. Either would impact normal executions, we believe significantly[2].

On the other hand, the current implementation's results are quite acceptable: the time it takes to convert a store in which only 1/10th of the objects are actually evolving is only about 4 times greater than the

[1] The evolution code makes very little use of multi-threading.

[2] With an explicit extent, every object creation has to perform an insert and additional space is required. Such extents increase the complexity of garbage collection, which has to remove entries. Regime scheme implemented in Sphere already provides as much segregation as the mutator chooses. We currently segregate into four regimes: large scalar arrays, large reference arrays, large instances and small instances. As segregation is increased, clustering matching access patterns is reduced.

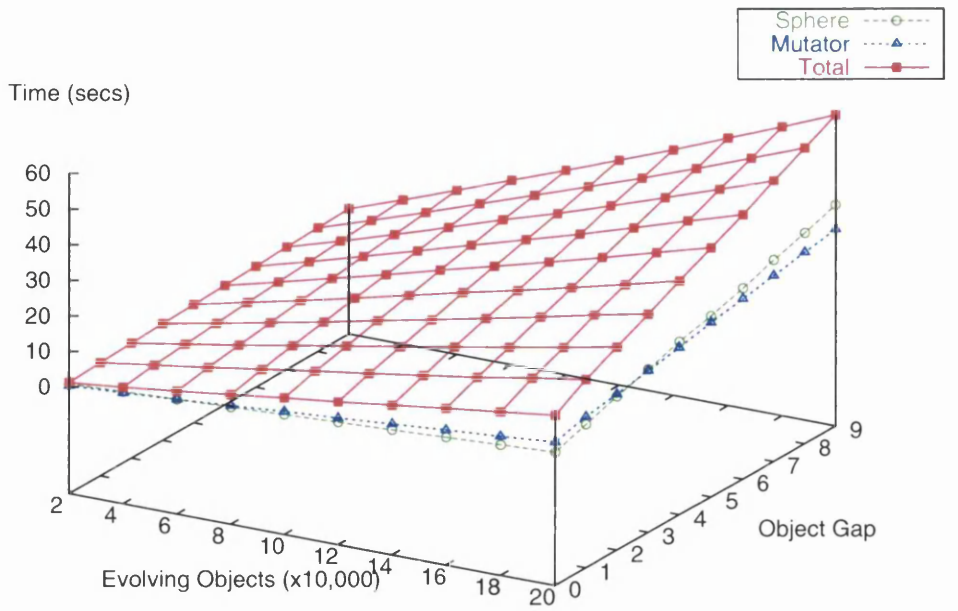

Figure 6.8: Test 1 results

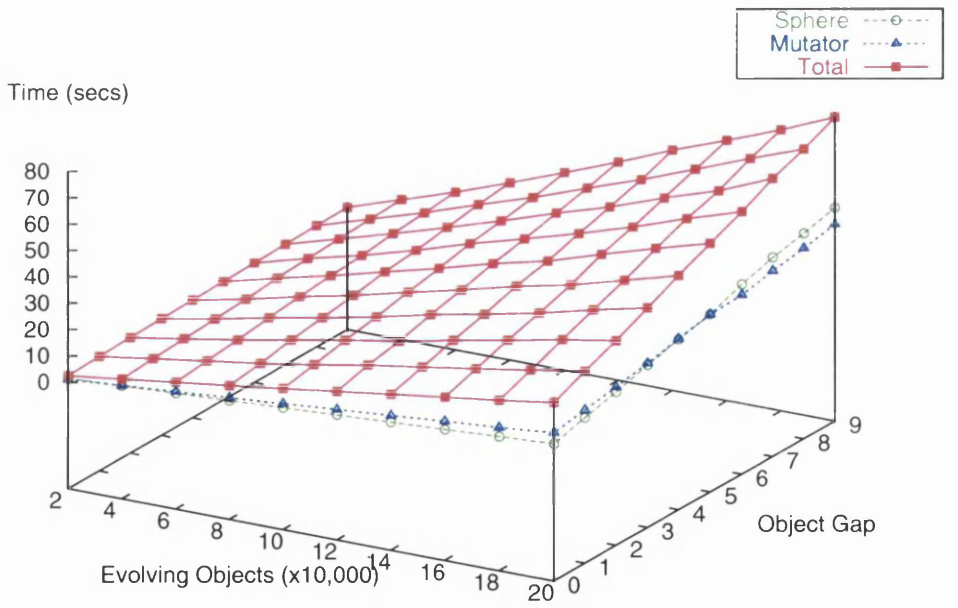

Figure 6.9: Test 2 results

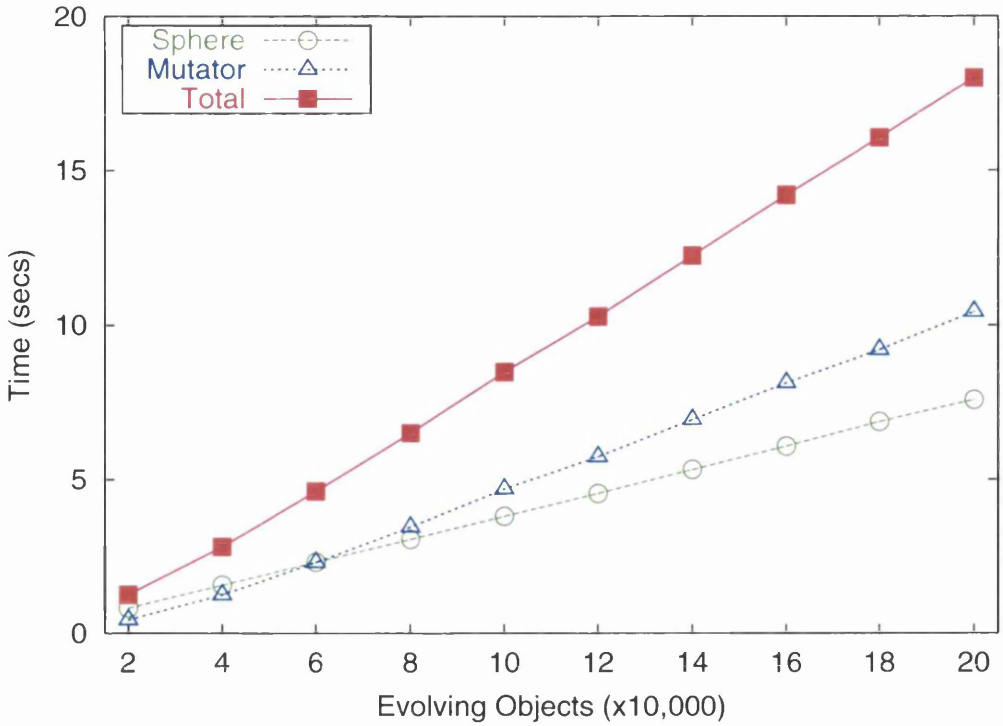

Figure 6.10: Test 1 results – fixed $g = 0$

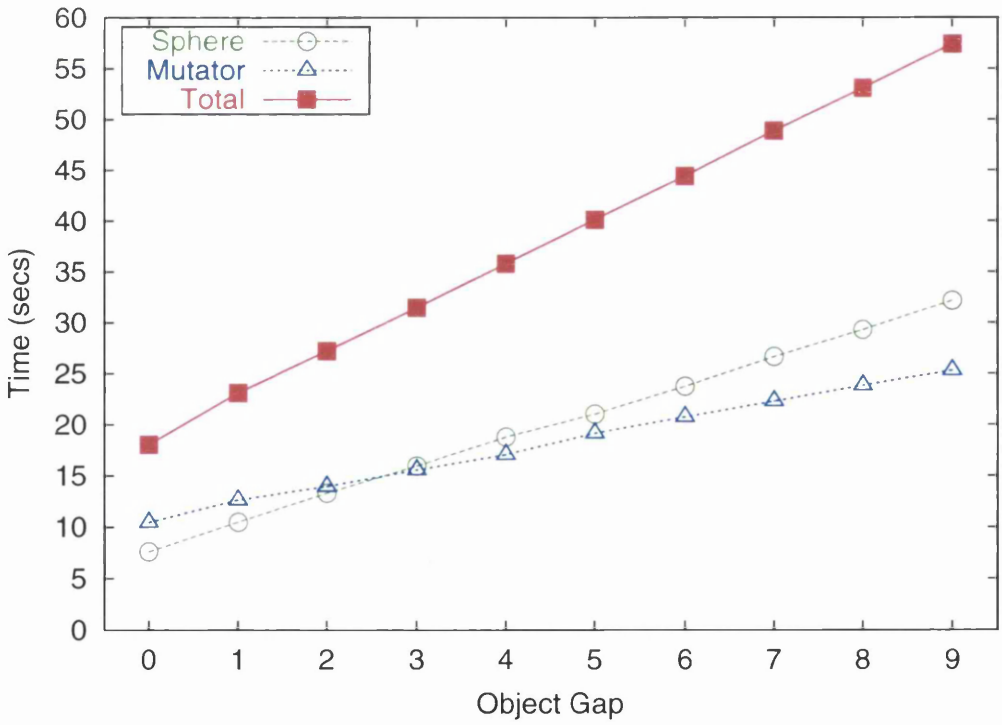

Figure 6.11: Test 1 results – fixed $n = 200,000$

time it takes to evolve the store containing only evolving objects. We also performed experiments with the stores where evolving and non-evolving objects were laid out in the store in two solid blocks, i.e. optimally clustered. On average, the slowdown for the store with $g = 9$ compared to the store with $g = 0$ was only about 5%.

In test 3 (Figure 6.12) we observe the same linear behaviour of Sphere, however the total time demonstrates a strange "quirk" in the part of the graph, where the number of objects is the greatest and they are packed densely. The cross-section of this graph at the constant value of $n = 200,000$ is presented in Figure 6.13. The behaviour is caused by a pathology in the upper software layer, i.e. JVM and Object Cache[3].

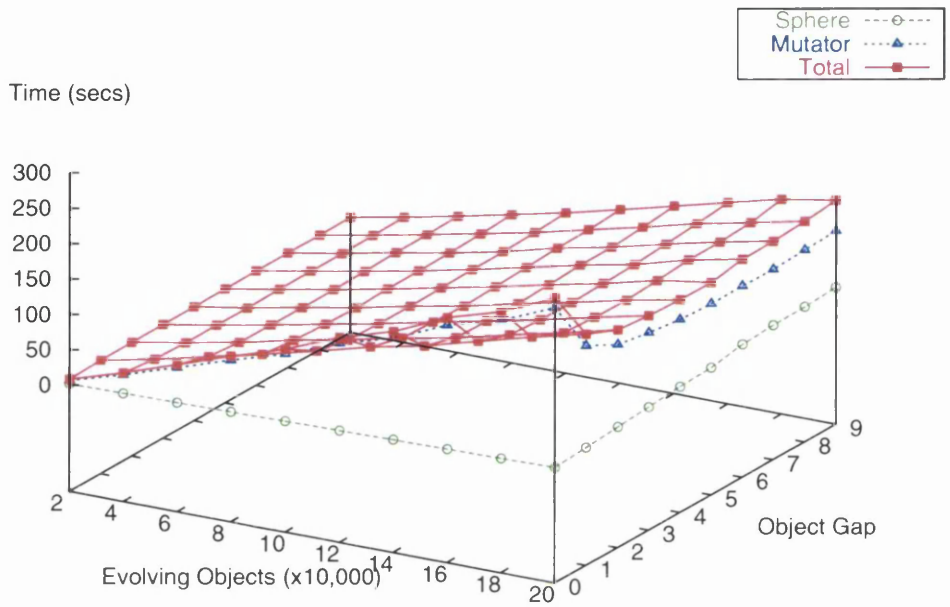

Figure 6.12: Test 3 results

To verify the linearity of our system's behaviour for much larger number of objects, we have performed the same evolution as in test 1, but with fixed $g = 5$ and with the number of objects varying between 100,000 and 2,000,000. The results are presented in Figure 6.14. At the highest point in the graph, the store contains approximately 12,000,000 objects of which 2,000,000 interleaved objects are evolved.

In all of our tests we measured the amount of log traffic generated as part of evolution. Building limbo evolved objects generates no additional log traffic, as this step is performed as part of disk garbage collection, which itself has been optimised for very low log traffic (see [HAD99, Pri00]). Evolution only requires the generation of a log record for every evolving object at commit time i.e. when swapping the state of limbo objects to make them live (step 3). Each log record is of a fixed size of 64 bytes (of which 40 bytes are system overhead), regardless of object size. In real terms 200,000 evolving objects generates approximately 16MB of log traffic. We anticipate that a further reduction in log traffic is possible by optimising the log

[3]This problem was investigated and it was found that its source is PEVM memory management, specifically the persistent object eviction mechanism.

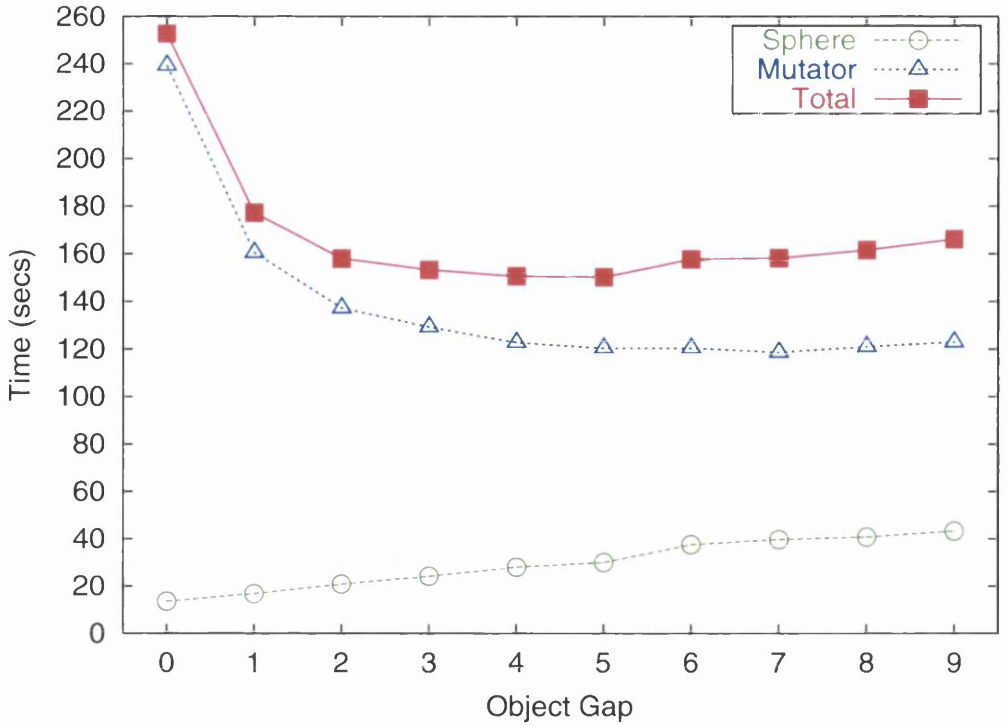

Figure 6.13: Test 3 graph cross-section at $n = 200000$

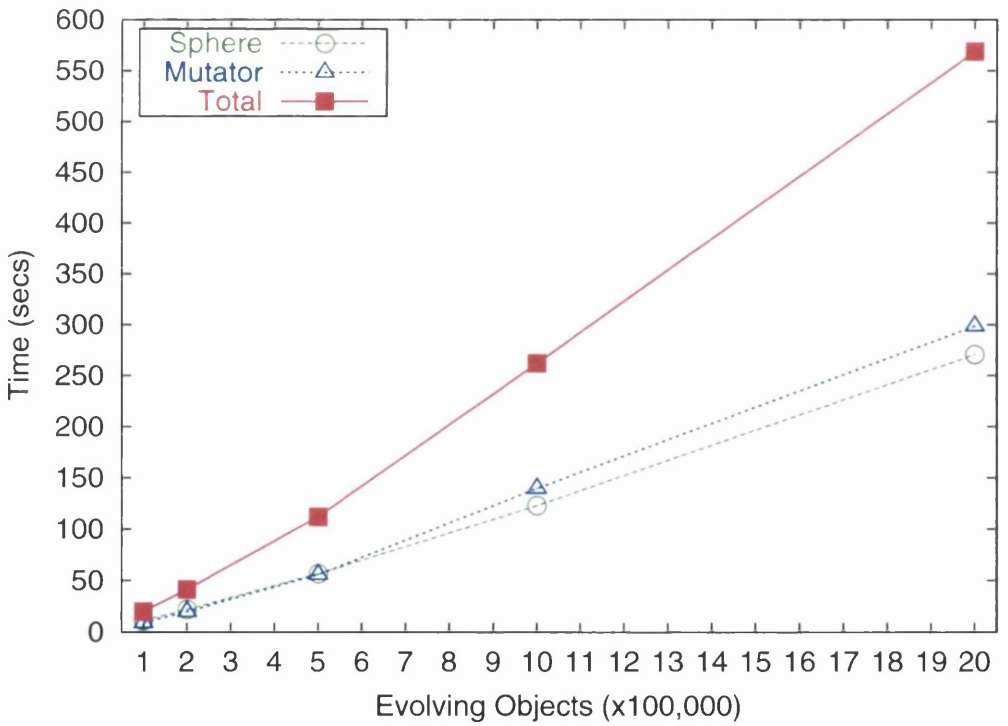

Figure 6.14: Test 1 with large number of objects, fixed $g = 5$

records associated with swapping the limbo states. We believe we can cache the information, generating log records which represent a vector of swapped states, rather than the current one-per-object.

In all of the above tests we observed the time per evolving object. For each test we have calculated average total and average Sphere time per object (in test 3 not taking into account the pathologically behaving part of the graph). The results are summarised in Table 6.2.

Test	Test 1	Test 2	Test 3
Object size (words)	2 – 3	7 – 8	7 – 7
Average total time per object (ms)	0.174	0.182	0.624
Average Sphere time per object (ms)	0.102	0.130	0.143

Table 6.2: Summary of the experimental results obtained in synthetic tests.

Comparing the time for test 1 and test 2, we observe relatively small change of time (30% for Sphere and almost 0% for total time), whereas the object size has grown about three times. We can conclude that at least for small objects and simple conversions the number of evolving objects matters much more than their size. Consequently, effects such as significant difference in conversion time for stores of the same size are possible.

6.4.3 A Real-Life Application

To validate our synthetic tests with a real-life application, we performed several experiments with GAP (see Section 5.6 for more details). In our evolution experiments we were changing persistent instances of "geographical line" class, subclasses of which represent such geographic features as roads and rivers. The form of vector data storage for such a feature can be in two forms shown on Figure 6.15.

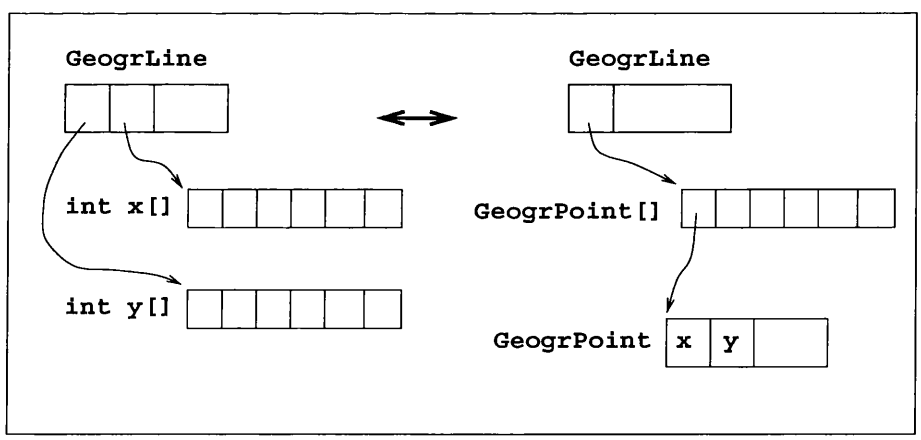

Figure 6.15: Representations of Geographical Lines

During GAP development, both of these representations were tried, so we decided that conversion between them is a good example of a "real-world" schema change. We converted successfully about 900,000 line

objects (complete UK data store, about 700MB in size), which took about 30 minutes. We also performed several experiments with smaller numbers of objects and observed practically linear time growth.

6.4.4 Related Work

To our surprise, we have managed to find very few research works that deal with scalability and recoverability of evolution and its performance. On the other hand, the documentation on the commercial systems which we could get hold of seems to generally ignore this problem. However, at least in one case an independent report reveals it, as we will see. Below is what we were able to find out.

There is one work where the O_2 system is benchmarked, which was published in 1994 [FMZ94]; an extended version of this appears in [ZCF+97]. In that work the authors concentrate on measuring and comparing the performance of immediate and deferred updates to variable size object databases. Since in PJama we have currently implemented only immediate (eager) conversion facilities, this work is of no direct relevance to us. It is also not clear, whether the requirements of scalability and safety were considered at this stage in the experimental design or product prototype. It is hard to compare the performance results, since in this work the authors did not specify exactly the changes they were making, and the hardware they used is almost obsolete by today's standards, so that equivalent times cannot be calculated.

The RD45 project at CERN has focused on the problems of providing persistent storage for vast quantities of data generated in physics experiments since 1995. At present the main system selected for use in this project is Objectivity/DB [Obj99a, Obj99b, Obj99d, Obj99c]. Its evolution facilities are quite sophisticated, as was discussed in Section 4.9.

Two CERN internal reports [Eur97a, Eur97b] contain some performance measurements for evolutionary object conversion with this system. Unfortunately, they mainly cover lazy conversion performance. Again, the hardware configuration used in the experiments (90MHz Pentium PC with 64MB memory and 1GB hard disk), does not allow us to compare absolute performance values. The report also shows that the time was linear in the number of objects evolving, and not particularly sensitive to object size.

As for the scalability and recoverability requirements, it looks as if Objectivity/DB satisfies the latter but does not satisfy the former, at least when eager conversion is applied. The documentation does not say anything, but according to [Eur97b], during evolution this system gradually loads all the evolving objects into RAM and keeps them there until the end of the transaction. This limits the amount of objects that can be converted in a single transaction to the amount that can fit into main memory and effectively prevents complex conversion of large databases.

6.5 Summary

In this chapter we have established the goals for our design of the low-level conversion support mechanism for the PJama system (scalability, safety and completeness). We then explained the issues that we encountered due to these requirements. We have presented one possible design of such a system, explained its drawbacks (high consumption of persistent store resources), and then presented our current design, which is

much more economic in this respect. The evaluation of this system has confirmed that it scales very well and that the performance is as expected, i.e. the time to convert objects depends linearly on their number. The absolute performance figures look good, but the virtual absence of similar published data for other systems makes comparisons difficult.

At present the importance of scalability and safety of eager object evolution is probably underestimated by researchers and developers. This may be due to the fact that, as we explained in this chapter, this becomes a real issue only if the higher-level conversion technology is sophisticated and provides the developer with the way to access data beyond the current object being converted. Some of the well-established modern commercial systems restrict evolution to avoid complex conversions. Nevertheless, even with these constraints, some of their conversion implementations do not appear to scale at this time.

There is also very little information in the literature on the performance of eager conversion. Again, it is more likely to be an issue if higher-level mechanism is non-trivial.

Chapter 7

Runtime Evolution of Java Applications

So far we were talking about off-line evolution of persistent classes and data. In this chapter, the technology of runtime evolution (dynamic class redefinition) of Java applications is described. The author started to work on it at Sun Microsystems Laboratories in August 2000 as an intern, and is continuing at present as an employee. Runtime evolution technology allows developers to change class definitions while the program containing these classes is active.

Runtime evolution has a lot in common with persistent class evolution. We believe that the experience gained during the work on persistent application evolution can be re-used in the following aspects:

- We believe that all possible safety checks and guarantees are of special importance for runtime evolution, since this technology is being designed specifically to prevent interruptions, especially the abrupt ones (such as a runtime error), of target applications. An approach equivalent to the one taken in the `opjb` (see Chapter 3), which involves class compatibility checks and specific actions in response to incompatible changes, should be followed to ensure that an intended change is type safe in the context of the current application.
- As with persistent application evolution, we will need to be able to convert existing instances if the new version of their class defines a different format for them (Chapter 4). Similar kinds of conversion, i.e. default and custom, would be desirable to implement.
- Problems of scalability and durability of conversion similar to those identified in Chapter 6 may arise, and perhaps the existing experience may be utilised in solving them.

On the other hand, runtime evolution is inherently more difficult than persistent class and data evolution. In case of persistent application evolution as it is implemented in PJama and other persistent systems known to us, an application being evolved is shut down, and thus what is actually evolving is a relatively loose collection of persistent classes and their instances. A running application, in contrast, has additional state, the main part of which is the contents of the stack(s) of its thread(s). Thus, to evolve an application dynamically, we have to somehow match its current code and the current state of its stack with the new code (and possibly new stack format that it defines). It is hard to see how this can be performed in an arbitrary case,

and thus our current goal is to identify possible specific cases, when this transition can be performed such that its effects are predictable.

Technically, supporting runtime evolution is also more difficult. One reason is that when it comes down to implementation of class replacement, whether it is a persistent store or an active VM, a major part of the operations that the evolution technology performs is switching and fixing various links between classes, their instances, and classes and internal VM structures. However, when a Java application is running, the number and complexity of such links is an order of magnitude greater than when an application is saved as a quiescent collection of classes in the persistent store.

For these reasons, the decision was taken to implement runtime evolution for Java in a number of stages. On the first stage, a very limited set of evolution facilities should have been implemented, which would allow only very restricted changes. On the other hand, it would have required a minimum implementation effort. Restrictions also mean that changes that are allowed are simple and thus their consequences can generally be predicted and controlled by the developer who initiates the change.

On the following stages, the number of kinds of allowed changes and, consequently, implementation complexity, should gradually increase. We believe that in future, additional functionality would be also required, that would help the developer to understand the consequences of changes, or e.g. apply these changes only when certain conditions are met, such that their results become predictable.

Thorough testing and evaluation of mechanisms developed at a previous stage should be performed before proceeding to the next stage, possibly causing adjustment of the next stage goals. This should prevent sudden emergence of unforeseen fundamental problems.

This chapter is structured as follows. We first describe our implementation platform — Sun HotSpot Java VM. Implementation stages as we currently see them are presented in Section 7.2. Section 7.3 describes the first stage, which has already been accomplished, on which only changes to method bodies are allowed. In Section 7.4 the second implementation stage, which is now close to completion, is discussed. At this stage, we impose fewer restrictions on the allowed changes. However they still should be binary compatible, and we don't support modifying the instance format. Section 7.5 discusses the directions for the future work, including one interesting application of class redefinition technology — dynamic fine-grain profiling. We then present the review of the related work and our conclusions.

In this chapter, in addition to the JLS (Java Language Specification, [GJSB00]), we will often reference the JVMS, which is short for Java Virtual Machine Specification [LY99].

7.1 HotSpot Java VM

The HotSpot VM is Sun's present Java VM, available in the JDK releases starting from JDK1.3. It was first officially launched in April 1999. In this section, we first present a historical note on this system, and then describe its features that are important in the context of our work.

7.1.1 A Historical Note

The following information was kindly provided by Robert Griesemer from JavaSoft, to whom the author is sincerely grateful.

Many of the ideas that eventually ended up encoded in the Java HotSpot Virtual Machine have their roots in much earlier systems, specifically the implementations of Smalltalk and SELF programming languages. While dynamic code generation has been used before for a variety of applications, the seminal work by Deutsch and Schiffman for the Smalltalk-80 system [DS84] is a key piece in the puzzle. Their system dynamically compiled individual methods into native machine code to speed up an otherwise comparatively slow Smalltalk bytecode interpreter. Their system influenced many other Smalltalk systems. Later, both the Smalltalk language and its implementation inspired the pure object-oriented language SELF by Ungar and Smith [US87]. A straightforward SELF interpreter runs even slower than a comparable Smalltalk interpreter and thus more sophisticated compiler technology was required. Significant performance improvements were achieved by Hoelzle in 1994 with his adaptive compilation system for SELF [Hol95]. In Hoelzle's system, frequently used methods were recompiled and inlined into the caller methods; in addition more sophisticated object-oriented optimisations (such as the elimination of block instantiation via block inlining) were introduced. The system also featured a mechanism called *deoptimisation*, which allowed the replacement of an optimised method activation by its corresponding interpreted equivalent (or several of them, if inlining occurred). Deoptimisation allows source-level (i.e., bytecode level) debugging of optimised code and also enables more sophisticated optimisations in the compiled code. The reader can find more information on the SELF project, including a number of publications, on the SELF Web site [Sun00r].

In 1994, two key engineers from the SELF group joined a small startup company called LongView Technologies LLC, commonly known as Animorphic Systems, in Palo Alto, California. At Animorphic Systems, many ideas found in the original SELF implementation were refined, and combined with state-of-the-art memory management and a high-performance interpreter. Animorphic's goal was to evolve the SELF technology to product quality and (re-)apply it to a new Smalltalk implementation called Strongtalk [BG93]. In 1995, with the advent of Java, the Animorphic team refocused and applied the same, somewhat modified, technology to their own clean room implementation of a Java virtual machine. Both Animorphic's Smalltalk and the Java HotSpot technology were advertised at the OOPSLA'96 conference exhibition. In 1997, Animorphic Systems was acquired by Sun Microsystems. The complete Animorphic engineering team continued with the further development of their Java virtual machine, which replaced Sun's Classic VM in 2000, with JDK1.3 release. Sun is continuing to evolve the HotSpot technology which by now includes significant improvements in compilation technology, garbage collection, thread synchronisation, internal data structures, and is covered by many patents.

7.1.2 The Features of the HotSpot JVM

The following discussion is based on the Sun's white paper [Sun99b], certain materials available internally at Sun, studying the source code of HotSpot VM, and the author's personal communication with the developers of this system.

7.1.2.1 Mixed Mode Application Execution

Straight bytecode interpretation is too slow for most industrial-strength Java applications. Therefore these days all of the industrial JVMs include compilation of the bytecodes to platform-specific native code to achieve reasonable program execution speed. Several strategies have been developed to address this need. They can be categorised broadly by whether the compilation to native code takes place before run time (static compilation) or during runtime (dynamic compilation). Static compilation for Java is presently supported by several systems, e.g. Visual Cafe [Web00] and JOVE [Ins00]. In contrast, HotSpot, as well as previous Sun's JVMs (Classic VM and EVM [WG98]) implement dynamic compilation. IBM's Jalapeño JVM [AAB+99, ACL+99] does not have an interpreter at all, and dynamically compiles the Java code.

HotSpot VM is equipped with both a dynamic compiler and an interpreter, and runs applications in *mixed mode*. This means that only parts of an application are compiled, whereas the rest is interpreted. Such a strategy is used to minimise total execution time, since it is observed that most of the applications spend most of their time in only a small portion of their code. HotSpot identifies at run time those parts of an application's code that are most critical for performance. The easiest way to do that is to associate a counter with each method entry and each loop of the application and increment this counter every time the given method is called or the loop is re-iterated in the interpreter. Those methods which are found to be called most, or which contain the most intensive loops ("hot spots", hence the name of this VM) are compiled and optimised, without wasting time and space compiling seldom-used code.

A discussion comparing the merits of static and dynamic compilation can certainly be very interesting (e.g. such issues as code portability, support for dynamic class loading and reflection, etc. are quite hard in case of static compilation), but it is out of scope of this thesis. What is relevant for this work is the ability of one approach or another to support runtime evolution of applications. From this point of view, there are two extremes — a purely interpretative VM, and a native compiler that creates a monolithic native executable file for a Java application. In the first case, runtime evolution implementation is technically the easiest, whereas in the second, we believe, it is hardly possible at all. This is due to the difficulties of inspecting and changing the internal structure of an application once it is loaded and running. It turns out that HotSpot VM with its mixed mode application execution and a number of additional facilities, originally designed for different purposes, (see further discussion) provides a reasonable compromise, allowing dynamic evolution with a moderate investment of technical effort.

We will now discuss the implementation of the interpreter and the compiler in HotSpot in more detail, with the emphasis on how certain features facilitate or make difficult the implementation of runtime evolution.

7.1.2.2 The Fast Interpreter

HotSpot is presently equipped with a fast interpreter, the design of which is different from that in previous Sun's VMs. The earliest implementation of the interpreter in the Classic VM followed a simple pattern, the main element of which was a loop combined with a `switch` statement, as sketched in Figure 7.1.

Such a design is not speed-optimal, for several reasons. First, the `switch` statement performs repeated compare operations, and in the worst case it may be required to compare a given command with all but one bytecodes to locate the required one. Second, it uses a separate software stack to pass Java arguments, while

```
while (thereAreMoreBytecodes()) {
  b = nextBytecode();
  switch (b) {
  case instr_x : do_this(); break;
  case instr_y : do_that();
                 ...
                 break;
  ...
  }
}
```

Figure 7.1: The core of a simple interpreter.

the native C stack is used by the VM itself. A number of JVM internal variables, such as the program counter or the stack pointer for a Java thread, are stored in C variables, which are not guaranteed to be always kept in the hardware registers. Management of these software interpreter structures consumes a considerable share of total execution time.

The interpreter in HotSpot is designed in a radically different way, in order to minimise the gap between the VM and the underlying hardware. Omitting some second-order technical details, its architecture looks as follows. Instead of using a `switch`, addresses of the pieces of the VM code responsible for interpretation of individual bytecodes are looked up by the interpreter in a special table. In other words, for a particular bytecode with the value, say, 5, the interpreter just goes to the fifth table entry, gets the address of the code and jumps to it. However, to be able to create such a table with pointers to code sections (not procedures, for which a high-overhead `call` instruction instead of a `jump` would have to be used), these code sections have to be generated at interpreter start-up. This is the only way (in standard C/C++) to obtain the runtime address of a code section and place it into the instruction lookup table. Therefore, a considerable part of the HotSpot interpreter code looks like quasi-assembler, but is actually a C++ code that, upon the VM startup, generates native machine code, which then works as a part of the interpreter. An example of a function that generates interpreter code for Java bytecode `pop` instruction (for Intel architecture) is presented in Figure 7.2.

```
void TemplateTable::pop() {
  // Some assertions here are omitted by us for clarity
  __ popl(eax);
}
```

Figure 7.2: An example of interpreter code generation

This remarkably short example also illustrates that a native hardware stack is used by the interpreter, i.e. all Java arguments (along with the VM internal information passed to methods) are put onto it. Certain dedicated registers are used to hold Java program counter, stack pointer, and other internal variables.

It is worth noting, that in the last releases of Sun's Classic JVM the main interpreter loop was also rewritten in assembler for speed-up, and table-driven lookup of code sections that interpret particular bytecodes was also used. However, instead of dynamic generation of the interpreter code, that can be supported by any

standard C/C++ compiler, a non-standard mechanism available in GNU C/C++ was used, that allows the programmer to manipulate source code labels in the same source code. Using this mechanism, it is possible to fill in a table with the addresses of static sections of code. However, in other aspects this interpreter implementation was much less sophisticated than in HotSpot. In EVM it was eventually abandoned, for the sake of code portability and maintainability, once the measurements had shown that the real speed-up due to the implementation in assembler, was rather small, of the order of 5-10 per cent. In the presence of the fast enough dynamic compiler, this speed up was not worth the maintenance effort.

Overall, the gap between the VM and the real machine is significantly narrowed by the HotSpot interpreter, which makes the interpretation speed considerably higher. This, however, comes at a price of e.g. large machine-specific chunks of code (roughly about 10 KLOC (thousand lines of code) of Intel-specific and 14 KLOC of SPARC-specific code). Overall code size and complexity is also significantly higher, since e.g. the code supporting dynamic code generation is needed. Obviously, debugging dynamically generated machine code is significantly more difficult than static code. These properties certainly do not facilitate implementation of runtime evolution, but they don't make it infeasible either.

7.1.2.3 The Compilers and Decompilation

At present, there are two native compilers available in HotSpot. The first of them is a relatively lightweight one (informally known as *C1*), oriented towards client-side applications, where fast startup due to fast dynamic compilation, and small footprint is more important than the quality and thus the speed of the code produced. The second compiler, *C2*, is relatively slow, but produces high-quality native code. It is thus more appropriate for server-side applications, where compilation time is almost irrelevant compared to the importance of the ultimate execution speed. Only one compiler can be used during a single VM session, and it can be chosen through a command-line flag. This is due to the fact that e.g. the stack formats for Java method calls defined by the two compilers are different, which makes dynamic switching from one compiler to another impractical.

One optimisation (it will be explained later why it is important for our work) supported by the compilers[1] is aggressive method inlining. "Aggressive" here means that a larger number of methods can be inlined that would be possible to if compilation was purely static (more precisely, if its results could not be changed later). If compilation is static in this sense, a method can be inlined only if the compiler can determine that a certain call cite is strictly *monomorphic*, i.e. only one method implementation can be called from it. Practically in Java it means that a method is either `static`, or can not be overridden in a subclass (i.e. `private` or `final`), or there is a special case such as the one below, where it is possible to determine through static analysis that the call is monomorphic:

```
int res = new C().m();
```

However, in certain cases static analysis can not guarantee that a call site is monomorphic, whereas the situation at run time suggests that it is, at least for some period of time. For example, if we have a piece of code similar to the one presented in Figure 7.3, and observe that `C` is a leaf class (more precisely, that there

[1] Actually, at present only by the `C2` compiler, though the implementation of the same optimisation in the `C1` is under way.

are no subclasses of this class *currently* loaded), we can inline the above call[2]. The problem is, what to do if later a subclass of `C` that overrides `m()` is loaded?

```
void m(C c) {
  c.m();
  ...
}
```

Figure 7.3: An example of a non-guaranteed monomorphic call site.

In the HotSpot compilers this problem was solved with the help of the so-called *deoptimisation* technology (initially introduced in the SELF system [Hol95]). Initially, the compiler uses runtime analysis to perform inlining aggressively. When each particular inlining is performed, the information about the method that inlines another one at a non-monomorphic call site, essentially a "caller method - callee method" pair, is recorded. Once a class is loaded, it is checked if it overrides any previously inlined methods, and if it does, all dependent methods are deoptimised. The latter means that the VM switches from executing their compiled code (containing inlinings that became incorrect) back to interpretation (which always calls methods via the dynamic dispatching mechanism). Later the deoptimised methods may be recompiled.

Technically deoptimisation is very hard, since the VM has to determine precisely the point in the interpreted code that corresponds to the point in the compiled code, create an interpreter-specific stack frame from the corresponding stack frame of the compiled method, and replace this frame on the hardware stack. This is done eagerly for all of the method activations currently on stack.

For runtime evolution support in production mode, when the Java code is compiled and optimised for high performance, the presence of the deoptimisation mechanism is very valuable. Methods of classes that are being dynamically redefined could be previously inlined, so once a class is redefined, all methods that depend on its methods in this way, should be deoptimised. Note that `static`, `final` and `private` methods, whose call sites are always strictly monomorphic, can be redefined too. Therefore, even if HotSpot supported only limited, non-aggressive inlining, we would have to develop some sort of deoptimisation technology for our own purposes, or disable inlining of methods (even of the `private` ones, since in Java they can be called not only from their defining class, but also from the inner classes of this class). Fortunately, we could re-use the existing technology, and had to only slightly customise it to support recording information about inlining of otherwise irrelevant `static`, `final` and `private` methods. This information is recorded in a special way, so that it is not taken into account by the "normal", non-evolutionary deoptimisation mechanism.

7.1.2.4 Memory Management

The first Sun JVM, the Classic VM, employed *object handles* to simplify implementation of garbage collection. A handle was a small fixed-size data area associated with a Java object and containing a pointer to the latter. All Java objects pointed to each other indirectly, via their handles. Thus during garbage collection objects could be moved around freely without changing pointers to other objects inside them — only the

[2]Compilation in HotSpot happens only after the VM realises that the given method is used heavily enough. If a method was called many times and the given call site in it is still monomorphic, it is unlikely that it will become polymorphic in the near future. Thus this kind of optimisation is justified.

pointer to each object in its handle needed to be patched.

However, it appears that accessing objects via handles slows down execution of Java applications significantly (this problem was first observed, and the VM without handles implemented, for Smalltalk by Ungar [Ung86]). Therefore the next generation Sun's VMs, i.e. EVM and HotSpot, are both "handleless". Java objects in these VMs point to other objects directly. Other modern JVMs, e.g. IBM Jalapeño, are also handleless.

Handles would simplify runtime evolution in cases when it is required to replace the internal class object for an evolving class, or to convert objects, creating their new versions on the heap. In both cases, we need to patch pointers: from instances to their class or from non-evolving instances to the evolving ones. If handles exist, it is enough to scan and patch just them; without handles, scanning the heap and patching direct pointers in objects themselves is required. However, this is unlikely to be especially difficult, since similar heap scanning and pointer patching are performed during the last phase of garbage collection. Thus we believe that the HotSpot VM garbage collector and the infrastructure for it which can be used separately (e.g. there is a readily available C++ class in HotSpot that allows iteration over all of the objects in the Java heap), is likely to satisfy our needs. Most importantly for us, this GC is *fully accurate*, which means that when it runs, it knows the location of all of the pointers to Java objects. This is generally not an easy thing to do, e.g. special measures are required to locate pointers that reside on native execution stacks or created in the native methods. For these reasons, garbage collectors of earlier JVMs, e.g. Sun's Classic VM, were *non-accurate* or *conservative* [BW88]. In some cases they had to consider as a putative pointer some memory words that looked "suspicious", i.e. contained a value that could be interpreted as a pointer. Since it was not known for sure whether or not such a word is really a pointer, its contents could not be changed and thus the contents of the memory area at which it was pointing could not be relocated. In contrast, accuracy, or *exact memory management* of the HotSpot garbage collector allows it to reclaim reliably all inaccessible object memory, and to relocate any object. In our case it also guarantees, that if we convert an object and thus create a new copy for it (effectively relocating this object), we will be able to patch all of the pointers to it reliably.

The HotSpot VM garbage collector is also incremental, significantly reducing user-detectable garbage collection pauses. The incremental collector scales smoothly, providing non-disruptive, relatively constant pause times, even when extremely large object data sets are being manipulated. This guarantees good behaviour for high-availability server applications, highly interactive applications, and applications manipulating very large sets of live objects. Some parts of the garbage collection functionality are inherited by HotSpot from EVM [AD97, ADM98, WG98], where they were first proved highly effective.

7.1.2.5 Internal Data Structures

The initial design of the internal representation of Java objects and classes in the VM memory affects the implementation of runtime evolution quite significantly. Comparing how similar things were implemented in different VMs, we come to the conclusion that variations in their design, even relatively small ones (which, nevertheless, are very hard to change once the system is mature, can lead to dramatic variations in the evolution implementation complexity.

Consider the internal representation of a class object in the HotSpot VM memory, a fragment of which is

presented in Figure 7.4. A tag on the top of each box on this figure denotes the type (the real internal HotSpot's C++ class names are used) of the respective object. Those types whose names end with `..Oop` are "quasi-Java" objects: they are invisible to Java applications that the VM runs, but are allocated on the Java heap. Thus, the garbage collector treats them as Java objects, so they can be moved to compact the heap and garbage collected once they become unreachable[3]. Pointers from one object to another are usually `private` fields of the respective C++ classes, accessible through accessor methods, names of which are printed over the corresponding arrows.

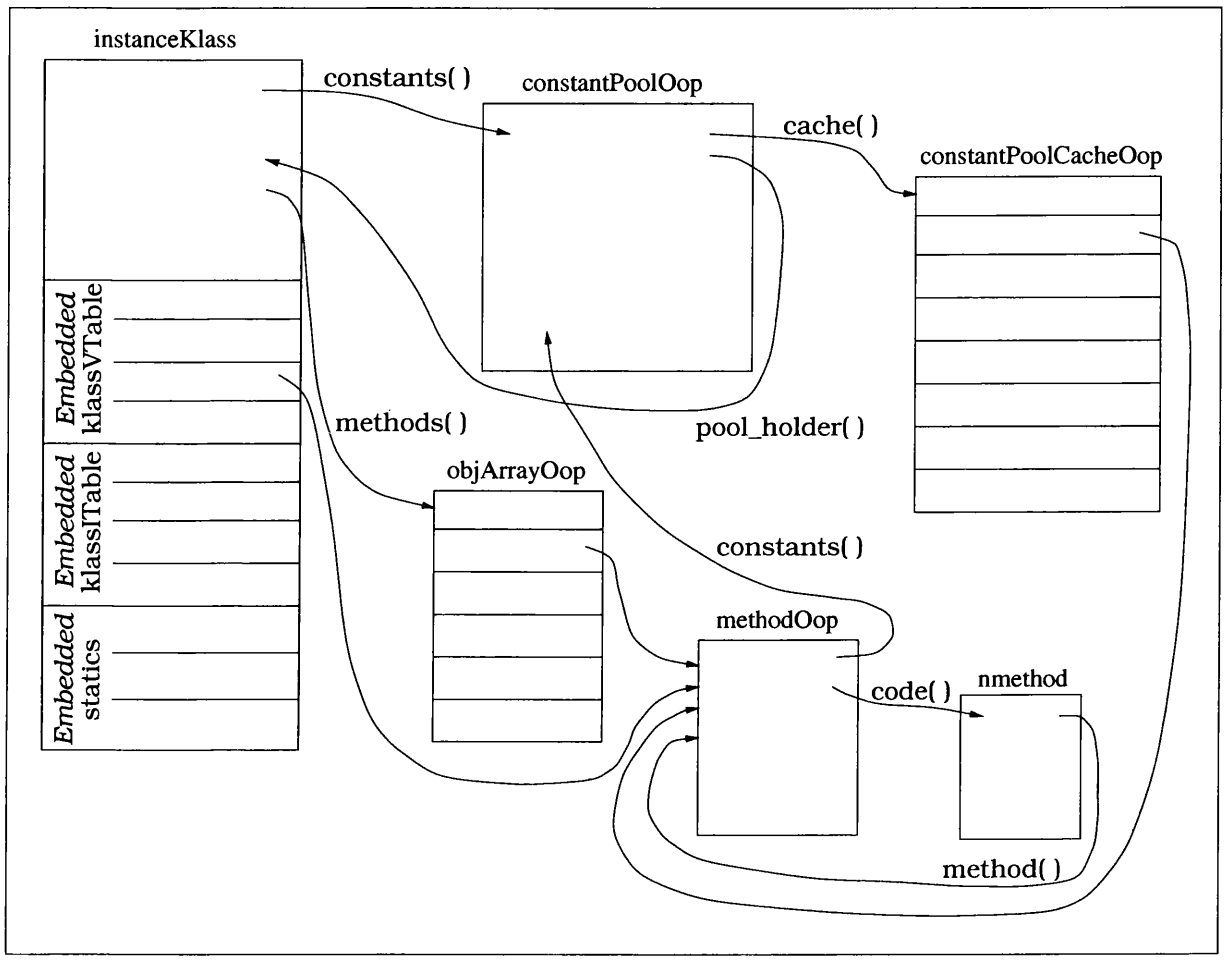

Figure 7.4: Internal representation of a class object in HotSpot VM

The main part of the class object has type `instanceKlass`, to which (more precisely, to its embracing quasi-Java object) instances of this class point directly[4] Two objects to which an `instanceKlass` points in turn, are most relevant for our work. One is the array of methods, which contains pointers to method objects for all methods defined in this class. The second object (actually a pair of objects) represent the constant pool (see JVMS, §4.4 and Section 3.3.2) of the class.

[3]Actually, the main part of the class object, `instanceKlass`, is embedded itself inside such a quasi-Java object — we don't show this embracing object just for the sake of clarity. Such a form of embedding is used in internal HotSpot types very often.

[4]In contrast, in EVM instances pointed to a special data structure called *near class*, and the latter pointed to the main class object. A near class contained, in particular, the pointers to the instance layout map and the VMT for the given class.

The constantPoolOop type object directly follows the constant pool structure in the class file, i.e. maps indexes to the respective constants, allowing to store (at class creation time), and then fetch, constants of types that correspond literally to those defined in the specification for Java class files. E.g. having an index for a method entry, one will obtain two other indexes and eventually the symbolic names of this method and its defining class. For any but a very primitive interpreter, this level of service is unacceptably slow. For example, what an interpreter really requires when it comes across a virtual method call, is an index into the virtual method table (VMT) for this method. This index, combined with the actual class of the instance on which a method is executed, will yield the correct Java method object with the bytecodes to execute.

In order to cache information such as the above, for method calls and field reads/writes, an additional structure called *constant pool cache* is employed in HotSpot. constantPoolCacheOop is a quasi-Java array, entries of which, however, point to non-Java objects of ConstantPoolCacheEntry type. There is one-to-one correspondence between each such object and an entry of either CONSTANT_Fieldref, CONSTANT_Methodref, or CONSTANT_InterfaceMethodref (see Section 3.3.2) type in the class's constant pool. A ConstantPoolCacheEntry object contains information relevant for an entry type (field or method). For example, for a static method entry it contains a direct pointer to the respective method object, whereas for a virtual method it contains its VMT index. Some less essential information, such as Java access codes for fields and methods, is also stored in these objects. When a Java class is being linked after loading, the VM parses and *rewrites* (in HotSpot's terminology) the bytecodes of its methods. This means that for each bytecode that calls a method or reads/writes a field, the index into the constant pool is replaced with the index into the constant pool cache. The essential contents of constant pool cache entries are, however, initialised lazily, i.e. when a method is first called or a field is first accessed.

A method object (methodOop) contains the bytecodes of the respective Java method and all other information necessary to run this method, e.g. the exception table, number of parameters/local variables, etc. Bytecode array, line number table (for exception handling/debugging), local variable table, and exception table are embedded into a methodOop object. However, the compiled code, created only for "hot" methods, is placed into a separate object of type nmethod.

Deoptimisation mechanism, first briefly described in Section 7.1.2.3, which is vital for our evolution technology, works as follows. If it has been found that certain method inlines contained in the compiled code of method m() became incorrect, we have to do two things: switch from the compiled code to interpretation of m() immediately, and prevent other methods from calling and executing the (now incorrect) compiled code of m(). Since calls to compiled methods from other compiled methods are direct, i.e. performed with a "call *addr*" processor instruction, we can't simply delete a compiled method. Instead, the link between methodOop and nmethod is cut, and a software trap (effectively a jump to the special handler function) is placed in the beginning of the compiled code for m(). Once this nmethod is called, a handler takes control, goes to the calling site and essentially overwrites it. The VM is then forced to re-resolve a call at this site through the constant pool cache, and will eventually get either the interpreted code, or the new nmethod, which by that time may have already been created.

The last part of the class representation which is relevant for our work is the virtual method table (VMT, or klassVTable as the respective HotSpot type is called), the interface method table (klassITable, the same purpose structure, but for interface methods) and static fields representation. Unfortunately, in this respect the HotSpot VM is not ideal for our purposes. In order to speed up access to methods and static variables, both of the above tables and the static fields are embedded directly into the instanceKlass object. This is in contrast with e.g. earlier Sun's JVMs (see e.g. [DA97]), where these were separate array objects,

referenced from the main class object. Embedding means that if we add a virtual method or a static variable to the class, its original `instanceKlass` object will become unusable[5]. A new class object has to be created for this class, and, in case of added virtual method, the same thing should be done for all of its subclasses. Then all of the pointers to this class object must be patched, so that they point to the new class version.

7.1.2.6 Code Size and Quality

HotSpot JVM source code currently comprises approximately 250 KLOC of processor- and OS-independent code, about 35 and 44 KLOC of, respectively, Intel- and Sparc-specific code (primarily used by the interpreter and the two compilers), and about 5 KLOC of code for each OS supported (currently Solaris, Linux and Windows). A relatively small amount of code, 1-2 KLOC in each case, is both processor- and OS-specific (there are just four of such cases, since only one OS, namely Solaris, is available for both Sparc and Intel processors). Thus, in general, HotSpot is a fairly large system.

Of the three JVMs whose source code the author worked with: Sun's Classic JVM, EVM, and HotSpot — the latter has the best code quality. In part this is due to the C++ implementation language (the other two VMs are written in C). Despite the lack of safety of C++ inherited from C, it is still an object-oriented language, and therefore coding in it provides a better code organisation almost for free. Indeed, just the fact that data and code which operates on it are grouped together in classes, greately improves code clarity and facilitates understanding it. Hierarchical class organisation, that leads to grouping more general and more specific functionality in different classes, also helps in understanding and extending the code. It also facilitates organising code in such a way that it can be effectively re-used, even in unanticipated ways. HotSpot code follows this discipline consistently enough, providing in many cases good and easily accessible infrastructure for those who wish to extend it. This was supported much less well in Sun's previous JVMs.

One other advantage of C++ relevant for a VM implementation is a certain level of memory management automation available in it. For example, in any JVM we have to manipulate Java objects in the VM runtime code. When this code runs on behalf of a normal application Java thread, GC can happen at any time. Locations of all pointers to Java objects should be known to the GC, therefore pointers to Java objects within the JVM code can rarely be passed as such. Instead, we have to hide them behind *handles*, which are pointers to some locations, essentially pools of object pointers, managed by the VM. Once garbage collection happens, the pointers in the pools are updated, but the values of handles (which at this time can be contained in hardware registers or C stack frames) need not be changed. Thus, we ensure GC safety, but get a problem of management of handles themselves. Namely, we should deallocate a handle once we don't need the Java object that it references anymore. Manual management of explicit deallocations of handles in EVM (which is written in C) was a headache, whereas in HotSpot handles are implemented as C++ class instances. Thus, if a handle is declared as a local variable of some function, a destructor for it, which will do all the necessary cleanup, is executed automatically once the handle goes out of scope. The same happens with some other structures implemented as C++ classes, e.g. various locks. Overall this noticeably improves code maintainability.

The coding style of HotSpot follows reasonable discipline. For example, the majority of data fields in

[5]To "expand" the class object in place would require sliding all of the objects that follow it in the heap, and patching the addresses — similar to what happens during mark-and-compact GC. This is even more expensive than allocating a new class object and patching pointers to it.

classes are private, with names starting with underbar, e.g. `int _some_field`. This allows a programmer to distinguish object fields from method local variables immediately when looking at an arbitrary piece of code. Fields that are required outside the class are accessible via accessor methods that also follow the naming convention. In the above case two accessor methods would be called `int some_field()` and `void set_some_field(int)`. Though this might not seem a great breakthrough, following this discipline saves somebody who tries to understand, extend and debug the code, innumerable lookups in a large number of files just to find out the most elementary things.

Unlike the two earlier Sun's JVMs, HotSpot does not suffer from excessive use of C macros, especially nested ones, which were debugging nightmare when working with those VMs.

The amount and quality of comments in HotSpot sources is much greater than that in the Sun's Classic JVM, and also looks greater than in the EVM. Unfortunately, however, very little alternative documentation exists for HotSpot (though the source code of its earlier versions is freely available, see [Sun00i]). Absence of the alternative documentation leads to the fact that the knowledge of many important internals of this project exists essentially only in the heads of its developers at JavaSoft. This is in sharp contrast with e.g. IBM's Jalapeño project with its large number of publications [IBM01]. The latter is primarily a research project (unlike HotSpot, which is a product), but papers also appear on the IBM Development Kit (IBM DK), which is IBM's rival product (see e.g. [GBCW00, SOT+00]).

7.1.3 Summary

HotSpot is a high-performance industrial Java VM that can be tuned at start-up time for both client-side and server-side optimal performance. The price of its high performance is increased code size and complexity, which makes changes and extensions that were not originally anticipated harder to perform. Nevertheless, for the purposes of this work HotSpot proved to be malleable enough. Certain properties of this system, such as support for compiled method deoptimisation, the infrastructure for iterating over Java objects and stack frames, and quite good code quality, facilitate implementation of runtime evolution a lot. Others, like the specific design of the interpreter, are almost neutral. Finally, the design of some internal data structures makes our work harder, and may also affect the performance of the resulting evolution sub-system. Overall, however, our impression is that HotSpot, at least compared to Sun's previous JVMs (Classic and EVM), is more suitable for experimental work, particularly for adding runtime evolution support.

7.2 Staged Implementation

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

1. **Changing method bodies only.** At this stage, only one kind of change to classes is allowed and supported, and these are changes to method bodies (code) only. Everything else in the new class

version should remain exactly as it was in the old class version. This stage is essentially complete now, the code is being tested and should eventually be included in the Sun's JDK1.4 release, scheduled for November 2001.

2. **Binary (source) compatible changes.** At this stage, only binary (source) compatible changes to classes (see JLS, Chapter 13) should be allowed. The reason for this to be a separate implementation stage is that incompatible changes are much more difficult to handle in the context of the running VM than in the context of the evolution of a loose collection of classes. The reader may recall (Chapter 3) that in the latter context, if we detect an incompatible change to a class, we just ensure that the new set of classes is mutually consistent. We do that by selective recompilation of classes that depend on the changed class. In the context of the running VM, we can also feed to it the new, consistent set of classes. The problem is, what shall we do in certain cases, e.g. when a class is made `final` while there are still some loaded subclasses of it. We currently believe that eventually (during the subsequent stages) the VM should check all such cases and abort the redefinition operation if any internal invariant (type safety) violation is detected.

   It is also not entirely clear yet, what kind of class compatibility — binary or source — should be checked. If only the VM side of evolution mechanism (see Section 7.3.2) is considered, we would rather talk about binary compatibility verification only. One reason is that in several cases source compatibility can be ensured only if the source code for a class is available. For the VM it seems inappropriate to deal with the source code for classes. On the other hand, an evolution tool which we eventually would like to develop, may (mandatorily or optionally) check source compatibility as well.

3. **Arbitrary changes, except the changes to instance format.** At this stage, any changes to classes should be allowed and supported, with the exception of those that affect the format of their instances. In principle, however, even the latter kind of changes may be allowed, provided that at the moment of transition there are no live instances of the modified class.

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

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

Most of the code supporting the evolution functionality is currently being written in C++, as a part of the JVM. That is because most of the operations it is currently performing are low-level and JVM-internal. In future certain higher-level operations may be more adequately implemented in Java, as it was in PJama (see Section 2.4. This will also prevent bloating the core JVM code.

In the following sections, we explain why the presented levels of functionality were chosen for different stages and describe the accomplished work, as well as design ideas not yet implemented.

7.3 Stage 1: Support for Changing Method Bodies

Stage 1 goals were chosen as they are essentially for two reasons. First, intuitively, only allowing changes to method bodies suggests that resulting disruption to the internal JVM structures is going to be minimal, and so is the implementation effort and the probability of serious unforeseen problems. Such changes can not be unsafe in the sense of binary or source incompatibility, thus there is no need to develop a substantial piece of functionality that deals with this aspect of change safety. On the other hand, this level of functionality looks useful and general enough (unlike, for example, the capability to patch only individual bytecodes in methods, without changing method size, as implemented in e.g. [Chi00]).

Another reason was that this level of class modification functionality is typically supported by advanced debuggers, e.g. the one implemented in IBM's VisualAge IDE [IBM00b]. In the debugger which has this facility, the developer can change the code of any method of the target application and resume debugging of the modified code. If the changed method is active (is on stack), the debugger would typically pop all of the active frames for this method. Support for more complex changes in the debugger does not make much sense: in this case it is cheaper to modify, recompile and restart the whole target application. Sun's nearest practical goal is, therefore, to equip HotSpot with the support for changing methods in the debugging mode, so that it matches or is ahead of other JVMs on the market.

Despite the fact that the goals at this implementation stage are quite limited, there are still a number of issues, both conceptual and technical, that should be resolved to make this technology work. In the following subsections, we describe these issues and our solutions.

7.3.1 Dealing with Active Methods

A class that we are replacing can have active methods. The new version of such a method may be different, but in the VM running in production mode we in general can't pop frames of arbitrary active methods, since this may lead to incorrect behaviour of the target application. Thus, the biggest problem of runtime application evolution — of matching the old and the new program code and state when the execution is at an arbitrary point — comes to the fore. If we change only methods, several solutions to this problem can be considered:

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

2. Currently active calls to old methods complete, all new calls go to new methods. This solution is technically the easiest.

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

4. Identify a point in the new method that corresponds to the current execution point in the old method, and switch execution straight from the old method code to the new one. In certain cases, e.g. when a method being evolved never terminates, and the changes are purely additive and free of side effects (for example, trace printing statements are added), this can be the desired and useful semantics. However, in more complex cases it may be very hard for the developer to understand all of the implications of a transition from one code version to another at some arbitrary point. One other application of the mechanism developed for this policy may be for dynamic fine-grain profiling (see Section 7.5.4).

It looks as if none of these solutions is a "single right" one. Rather, they are different policies, and each of them may be preferable in certain situations. However, solution number 2 looks much easier to implement than the others, and thus it is the only one which is currently supported. Our present design ideas for other method switching policies are discussed in detail in Section 7.5.3.

7.3.2 Interface to the Runtime Evolution Functionality

At present the core of the runtime class evolution functionality is implemented as two calls, `RedefineClasses()` and `PopFrame()` in the *Java VM Debugging Interface (JVMDI)* [Sun00l], which in turn is a part of *Java Platform Debugger Architecture (JPDA)* [Sun00k]. At the moment of writing the description of these calls has not yet been finalised and included in the publically available copy of the JVMDI Specification. For this reason and for convenience, we present the initial specification of the above two calls in Appendix C.

Note that the `RedefineClasses()` call accepts the actual class object to redefine, not the class name. This, in particular, means that the issue of dealing with multiple classes with the same name, loaded by different class loaders, should be addressed by the evolution tool, rather than the VM. We hope that the GUI tool with sufficient visualisation facilities would allow the user to conveniently specify which class(es) should be actually redefined.

JPDA is the open standard, to which a JVM and/or a debugging tool of any vendor may conform. This should allow a programmer to debug an application in the VM supplied by one vendor using a debugger tool from another vendor. The reader can find the complete description of the JPDA design and APIs in the documentation referenced above. Below we present a quick overview.

In JPDA (see Figure 7.5) the debugging tool (debugger) and the target JVM (debuggee) run in separate OS processes, on the same or different machines. A relatively compact debugger back end is a part of the JVM, activated if the latter is started in debugger-enabled mode, using several special command line options. It runs in a dedicated thread and provides a way both to inspect the state and to control the execution of the target application. The native C interface that the back end implements (JVMDI) is a two-way interface. A JVMDI client can query and control the target application through many different functions in this interface. On the other hand, the client can be notified of interesting occurrences through events that the VM generates in the debugger-enabled mode.

Running the debugger back end in the same virtual machine as the target application, and the front end — in a separate process, provides maximal control over the target application with minimal intrusion on the part of the debugger. The code of the back end is relatively compact and consumes a minimum amount

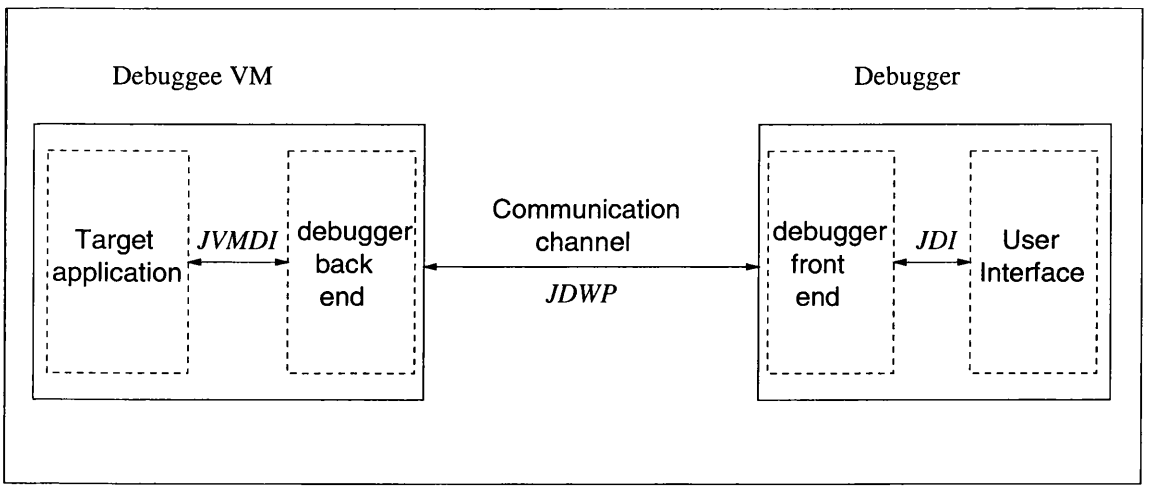

Figure 7.5: Java Platform Debugger Architecture (JPDA).

of resources. Since it is the native C code, it will not struggle with the target application for Java-specific resources, such as Java heap space, so the overhead and the intrusion it introduces are minimal. On the other hand, the debugger may be implemented as a portable Java application, free to use GUI APIs and other code that consumes significant amounts of RAM and/or processor cycles.

Debugger back end and front end communicate with each other over a communication channel, using *Java Debugging Wire Protocol* (JDWP). This protocol defines the format of information and requests transferred between the debuggee VM and the debugger front-end. JDWP itself does not define the transport mechanism. What is presently supported by the HotSpot JVM and Sun's debugging tools are sockets and (under Windows only) shared memory. The specification of the protocol allows the debuggee and the debugger to run under separate VM implementations and/or on separate hardware/OS platforms. It does not specify the programming language for either side, which means that e.g. the debuggee VM can be written in Java itself.

Information and requests specified in JDWP are roughly at the level of the JVMDI, but also include additional information and requests necessitated by bandwidth and signal rate issues, e.g. information filtering and batching.

For the debugger front end, Sun provides its own Java class library called *Java Debugging Interface* (JDI) which defines a relatively high-level interface to the debugging functionality in Java. It therefore can be easily integrated into a portable standalone debugging tool or an IDE written in Java.

To support runtime class redefinition, JDWP was extended to include the commands corresponding to `RedefineClasses` and `PopFrame`. To make this functionality available for target applications running in production mode[6], the format of the debugging-related command line options of the JVM was extended. It is now possible to start the VM in the mode with the thread listening to the JDWP channel, as well as the class redefinition functionality, enabled, and the rest of the debugging functionality disabled.

[6]Currently, HotSpot provides the complete debugging functionality only in purely interpreter mode. Furthermore, in the standard debugging-enabled mode, an additional overhead is introduced by e.g. the interpreter making JVMDI calls on method entry/exit, class loading, etc.

7.3.3 New Class Version Loading and Validation

In order to redefine a class at run time, if only its method bodies have been changed, we first perform the following steps (see Section 7.1.2.5 for relevant terminology):

1. Create an internal class object (`instanceKlass`) out of the bytecodes of the new class version, using standard JVM class file parsing and class object creation mechanisms, and parts of the standard class linking mechanism. Such a strategy allows us to re-use the existing JVM code and create the necessary internal JVM structures that will be needed to correctly support dynamic class redefinition. The following operations are performed:

   (a) Parse the passed class file, creating `instanceKlass` object. The whole operation is aborted if any of the usual problems, such as unsupported class version or class format error, are detected.

   (b) Verify the bytecodes, using standard Java verification mechanisms. Again, the whole operation is aborted if problems are detected.

   (c) Rewrite the bytecodes, so that they work with the constant pool cache (see Section 7.1.2.5) rather than the standard constant pool.

   The following operations, which are standard for normal class loading and linking, are *not* performed:

   (a) Adding the new class to the JVM internal class dictionary (a structure mapping class name and loader to the class object) and to the *loader constraints* (see JVMS, §5.3.4). The new class object is not treated as a separate class; it is just a placeholder, pointing to other objects such as methods and constant pool, which we will then link to the original class object.

   (b) Recursive linking of superclasses and superinterfaces of the loaded class. Not needed, since changes to class hierarchy are not allowed, thus all of the superclasses and superinterfaces of the changed class are already loaded and linked.

   (c) Initialisation of `klassVTable` and `klassITable` of the new class. That is not required, because new `instanceKlass` will not be used as a real working class object.

   (d) Execution of static initialisers of the new class version. This is in accordance with the specification, which shares the view according to which reflective changes should not cause any otherwise "normal" initialisations.

2. Validate the changes, i.e. compare the old and the new class versions. If any of the following is not true, abort the whole operation:

   (a) The superclasses for the original and the new class objects are the same.

   (b) The number, names and order of directly implemented interfaces are the same. The order of interfaces is important since it affects the format of `klassITables`.

   (c) The number, names, type and order of the fields declared in the old and the new class versions are the same. The order is important as it affects the format of instances and of the static variable block in the `instanceKlass`.

   (d) The number, names, signatures and the order of methods declared in these classes is the same. The order is important as it affects the `klassVTable`.

After all new classes are loaded and the changes are validated, each new class object is appended to the class cache of the class loader of the original class. Each Java class loader has such a class cache (which is just a Java `Vector` type object in class `java.lang.ClassLoader`), to which class objects can be appended, but can never be removed. It exists solely to prevent unloading of otherwise unreachable classes, since the JLS specifies that a class becomes unreachable only if its class loader becomes unreachable. Why we need to perform this operation will be explained below, when we discuss the operations during actual class transformation.

7.3.4 Class Transformation inside the JVM

Note that the above operations were performed within a standard Java thread, that runs concurrently with other threads. This normal execution mode reduces latency and enables garbage collection. The latter may be required, since when classes are loaded, a number of object allocations on the Java heap are performed. On the other hand, at the time of actual class transformation we should exclude any interference from other application threads. Otherwise, such a thread may attempt to access a class in the middle of transition, when it is in half-transformed state, and the consequences will be unpredictable. We also may need to inspect the stacks of Java threads and possibly patch them, which also requires all these threads to be blocked. Therefore, class transformation happens inside a special HotSpot *system thread*, which first suspends all other threads and prohibits garbage collection.

Technically, such a specialised system thread can be implemented in HotSpot by simply deriving a subclass from a special `VM_Operation` class and overriding its designated methods. The functionality implemented in the base `VM_Operation` class guarantees that the supplied code will be executed in a system thread, whereas all application threads will be blocked. Furthermore, each of these threads will be at a *safe point*, i.e. a state when the location of all of the pointers to Java objects is known[7]. The latter property allows us to walk stacks of these threads and patch them. In future, this should also allow us to perform object conversion, that requires updating all pointers to converted objects.

So, we invoke a method of our subclass of `VM_Operation`, and class transformation starts. How pointers in the two class objects are patched during this operation is presented in Figure 7.6. Those links which we switch from the structures of the original class to the counterpart structures of the new one and vice versa, are shown as red arrows that cross the boundary between the class objects. Objects that become unused either immediately after transformation (the new `instanceKlass` and its array of method pointers), or as soon as no old methods are executed anymore, are depicted as shaded.

Links from `klassVTable` and `klassITable` of the new `instanceKlass` object to the virtual methods are simply not set, since the corresponding initialisation function is not executed for the new `instanceKlass`. Other pointers are set or cleared during the following essential steps that our code performs:

1. If working in the debugger, remove all of the breakpoints in the methods of the class being replaced. This specification requirement was introduced to support both sophisticated and naive debugging agents. The latter may not distinguish between methods that have really been changed and that have

[7]This is required, in particular, to perform exact garbage collection. HotSpot's garbage collector entry code is also implemented as a `VM_Operation` subclass.

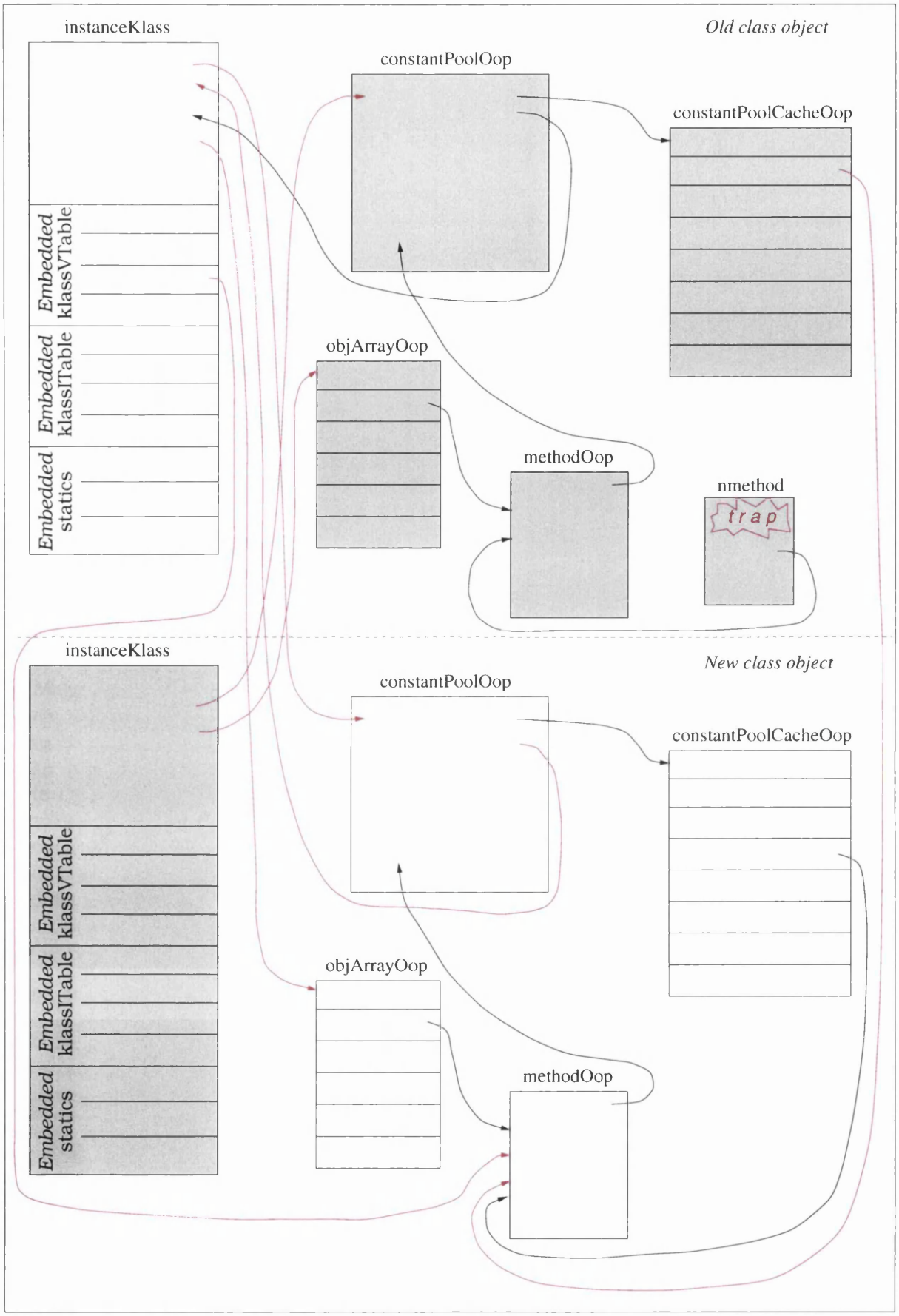

Figure 7.6: Class transformation when method bodies only are changed.

not, while breakpoints in the changed methods should be necessarily deleted. A sophisticated client can re-instantiate the breakpoints in the methods that have not actually been changed (see Section 7.3.5) after redefinition is complete.

2. If the class being redefined has any compiled methods (`nmethods`), deoptimise them, i.e:
   - for those of them which are on stack, switch to interpretation of their (still old) code;
   - make each such `nmethod` *non-entrant* (unexecutable) by replacing its first instruction with a trap, and cutting the link between the `methodOop` and `nmethod`.

   The effect of these operations is that some previously compiled code that calls a (now non-entrant) compiled method `n` would get into a trap, and the trap handler would eventually redirect it to the new version of this method through the patched constant pool cache (see below).

3. Deoptimise all the compiled code that depends on the class being redefined (i.e. inlines any of its methods). Execution of dependent methods will continue in the interpreted mode, in which the calls to the methods of the modified class will bind correctly to the new versions of these methods[8].

4. Patch the indexes into the constant pool from the array of fields (not shown on the figure) of the original `instanceKlass`. This is required, since the layout of the new constant pool can be different, so the old indexes in the array of fields, that correspond to field names and signatures, can become invalid.

5. Make the original `instanceKlass` point to the new constant pool object, and make the latter point back to the original `instanceKlass`. This is required, since in the new class version the contents of the constant pool may be different.

6. Make the new `instanceKlass` point to the old constant pool object and the old array of method pointers. We have previously protected the new class object from being garbage collected by its inclusion in the class loader's class cache. This was done just in order to protect the above objects. Otherwise, i.e. if all old methods become unreachable, it may happen that the garbage collector attempts to collect those of them which still have an activation on the stack after class redefinition is complete. Our solution here is of course quite conservative; on the other hand, the space overhead due to presence of multiple old class versions in memory looks small compared with the total space occupied by objects in large Java applications. Modifying the GC so that it scans stacks for method activations and treats those as references to their method object, i.e. as roots, could be a better approach; however we are not sure that this would not break other VM invariants. Another argument for our present solution is that it should allow to perform a class replacement "undo" quite easily.

7. Replace the methods, i.e. make the original `instanceKlass` point to the new method pointer array.

8. In the production mode, mark each old method with a special "obsolete" flag. This is done to prevent repeated compilation of old methods and subsequent calls of this, effectively old code, in certain situations.

[8]Note that this may not necessarily work for an arbitrary Java interpreter implementation. Practically all of the Java interpreters optimise certain bytecodes by physically replacing them with their "quick" versions. A quick bytecode references a resolved field or method in a shorter way than the standard bytecode, e.g. it may contain just the number of the virtual method in the VMT, instead of its constant pool index. The problem for us arises when this index is made embedded in the method body, and no way is provided to trace back to the real defining class and method behind this index (as it was the case in Sun Classic JVM). Fortunately, in HotSpot quick bytecodes refer to constant pool cache, which contains all the necessary information about the referenced fields and methods.

9. In the debugger mode, compare methods within each "old – new" pair, and mark obsolete only those which have been really changed. "Obsolete" flag will be used by JVMDI functions that inspect the stack on the debugger's request and return IDs for methods on stack. According to the specification, a universal "obsolete ID" should be returned for all methods that have been changed and are still active. Methods that have not really been changed in the new class version are handled in a special way (see Section 7.3.5 for details), so that eventually valid IDs can be returned for them.

10. Re-initialise the `klassVTable` and `klassITable` of the original `instanceKlass` tables, so that the pointers from them are redirected to the new methods.

11. Iterate over all classes currently loaded, except Java core classes, which are guaranteed to not reference our evolving application classes. For each class `C`, scan its constant pool cache and replace all pointers to old methods of the evolving class with the pointers to the corresponding new methods. If `C` is a subclass of the evolving class, scan and do the same patching for its `klassVTable` and `klassITable`.

7.3.5 Methods, Equivalent Modulo Constant Pool

The semantics of changing methods in the debugging and in the server contexts are different. What the user would expect during debugging, is the capability to change methods individually. From the user's point of view, they just fix a method, after which the debugger pops all of its stack frames, so that this method's old code will never be re-entered. Other methods of the same class remain as they were. In other words, after replacement has happened, there is still effectively a single version for each method in a class to which the change was applied.

The semantics for the change in the server context that we currently support, is different: for a changed method both the old and the new version are allowed to coexist until the old version code terminates. In fact, for an unchanged method two versions (though they are functionally absolutely equivalent) may coexist in the same way as well.

The reader may have noticed that the transformation technique depicted in Figure 7.6 reflects the second type of semantics. We will now describe how we support the first one.

What we need is, first of all, to be able to distinguish between changed and unchanged methods. It turns out that simple literal comparison of bytecodes for each "old – new" pair of methods will not always work. That is because some method that has been really changed may, for example, define new constants in the constant pool, or not use some constants that its old version used. As a result, the layout of the constant pool in the new class version may change, the indexes for the same constants may become different in the old and the new class versions, and eventually the bytecodes for two methods that are functionally equivalent may now look different. To reliably compare two method versions we, therefore, have to parse their bytecodes, comparing them one after another. For those bytecodes which have a constant pool index argument, we compare the actual values of constants at the two indexes, and if they are the same (though the indexes themselves may not be the same), we consider the pair of bytecodes equivalent. If two methods are compared using this method and found equivalent, we say that they are *equivalent modulo constant pool.*

Now that we know which methods have been actually changed and which not, we can pop the frames of the changed methods, and thus effectively a single version for each such method will remain alive. But

we can't do the same with unchanged methods, and if such a method has some invocations on stack, we end up with effectively two functionally equivalent versions (`methodOops`) of such a method: the "old" one referenced from the stack frames, and the "new" one referenced from the class object. In JVMDI, methods are referenced through special objects (`jmethodIDs`), that internally consist of a pointer to the `instanceKlass` object and an index into its method array. So, all `jmethodIDs` will now automatically point to the new code, whereas the VM will be executing the old one. Consequently, if we try to set a breakpoint in the method (this operation also goes through its `jmethodID`), it will be set in the new code and have no effect in the old (and still running) code.

To fix this problem, we scan the stacks of the Java threads and patch all the pointers to old method versions, so that they now point to the respective new method versions. An interpreter stack frame also contains a pointer to the constant pool cache of the method's class — this also needs to be updated. Finally, we update the value of the "current bytecode pointer" in each patched stack frame — this pointer is a machine address, so it has to be updated once a method is effectively relocated.

If any breakpoints existed in the unchanged methods, they should be re-instantiated by the debugging client in the new method version after redefinition is complete (this, as well as the other aspects of breakpoint handling, is a specification requirement imposed by Sun and beyond the author's control).

7.3.6 Summary

At the implementation stage 1, our runtime evolution technology supports changes to method bodies only. The implementation is now fully operational, and the code is the part of the HotSpot JVM scheduled for release in the end of the year 2001. Changes to classes are supported via a single C call in the JVMDI (JVM Debugging Interface). This facility works in both the debugging and the production JVM modes. In the latter, however, it is supported only for one of the two HotSpot dynamic compilers, namely the "server compiler" (`C2`). This temporary restriction is due to the fact that so far another compiler, `C1`, does not yet support compiled code deoptimisation. Considering that runtime evolution in production mode is targeted at long-running server-side applications, for which `C2` is used, this restriction is not a serious problem.

7.4 Stage 2: Support for Binary Compatible Changes

The reasons for choosing this stage's goals as they are, were first explained in Section 7.2. By allowing binary compatible changes only, we avoid the difficulties of verifying that the new class version does not break certain internal JVM invariants. For example, if we allow binary incompatible changes, we should immediately reject a class that became `final`, if it already has loaded subclasses. We discuss this in more detail in Section 7.5.1. Here we will just note that our informal experience is that most of the changes performed during application evolution, tend to extend the application by adding new classes, fields and methods. Most of such changes are binary compatible. We therefore believe that support for binary compatible changes will satisfy a significant part of the demand for runtime evolution.

The interface to runtime evolution functionality and the policy of dealing with active methods remains the same at this implementation stage. Therefore we will proceed straight to the technical details of class

redefinition. With the changes allowed at this implementation stage, this procedure becomes more complex. It now includes the following phases:

1. New class versions loading;
2. Change validation and replacement method determination;
3. New class version unregistering and re-registering;
4. Subclass expanding;
5. Class replacement.

Not all of these stages are likely to be applicable in the "general case", i.e. for any other JVM, though we believe that for any JVM complying with the JVMS, loading of new classes and change validation will have to be performed in essentially the same way. Class unregistering and re-registering probably have to be very similar too. On the other hand, the need to use one or another replacement method depending on the nature of changes to the class, and the need to expand, as we call it, the subclasses of some of the changed classes, is dictated solely by the fact that in HotSpot VM virtual method tables and static variables are embedded inside class objects, rather than stored in separate arrays. This optimisation was originally introduced to speed up execution by getting rid of several intermediate instructions that are required to follow the pointer from the class object to the separately stored VMT or a static variable. Though other alternatives, that provide the same speed-up without compromising "serviceability", might be possible in principle, at present we are unable to change this arrangement, since a vast amount of the VM code depends on the present class structure.

In the following sections we explain why each of the above stages of class redefinition is necessary in HotSpot and describe how it is implemented.

7.4.1 Loading New Class Versions

Allowing less restrictive, though binary compatible, changes means, among other things, that new classes can be inserted into the class hierarchy. This fact leads to a significant change to the class loading procedure for the implementation stage 2. At the previous stage, it was enough to just parse the bytecodes for the new class version, passing the parsing code the same class loader that loaded the old class version. The result was that the new class version would always link to the old superclass, as shown in Figure 7.7 (a), even if several hierarchically related classes were replaced. It worked like that since the old class loader assigned to the new class version would always pick up the old, already loaded superclass for this class. This behaviour was absolutely correct at this stage, since no changes to the class hierarchy were allowed, the old class object was re-used for the new class version, and class interface remained the same — thus verification of the new class version against the old and the new superclass would yield the same result.

However, none of the above is true once less restrictive changes are allowed. New classes may be inserted into the class tree. If new methods and static variables are added, the internal class object for the new class version may become larger than the old one (since virtual method tables and static variable slots are

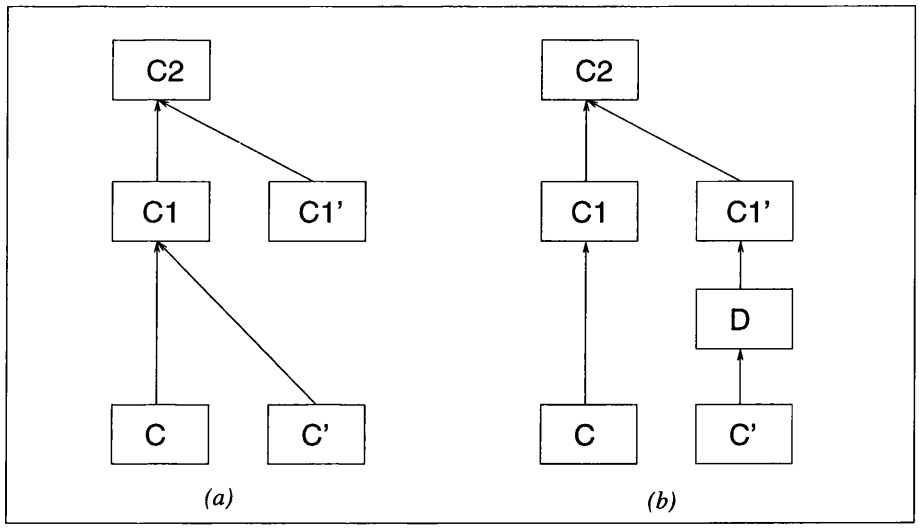

Figure 7.7: Linking hierarchically related classes during redefinition.

embedded inside the class object), and thus it would not be possible anymore to re-use the old class object. Finally, since the interface and the layout of the superclass may change, the code verifying its changed subclasses will need the link to the new, not the old superclass version. Therefore the links between classes in the hierarchy should look as shown in Figure 7.7(b)[9].

To support this kind of class loading and linking, we had to implement our own class loader (see the specification for the `ClassLoader` class in [Sun00e]), called `HotSwapClassLoader`. This class loader is quite simple: it has a constructor and just two methods. The constructor takes an array of classes to redefine (i.e. class objects for their old versions) and the bytecodes of their new versions. The second method, `defineStartingClass(Class C)`, is called by our JVM-internal code to notify `HotSwapClassLoader` about the old class `C`, whose new version we are going to load. This is done to make `HotSwapClassLoader` remember `C`'s original class loader `CL`, which should be used to locate and load any of the `C`'s superclasses that are not being redefined themselves. The last method of `HotSwapClassLoader` is a standard method for all class loaders called `loadClass(String name)`. We call it explicitly for `C`'s new version, and then the JVM calls it to load all of `C`'s superclasses recursively, until an already loaded superclass with the same defining class loader is found. Our implementation of this method compares the given class name with the names of the classes nominated for redefinition. If a match is found, the corresponding new bytecodes are passed to the standard `defineClass()` method, and the resulting class object is returned (class `C1` in Figure 7.8(a)). If a match is not found, it means that `loadClass()` was called recursively for some superclass of the new class version, and this superclass is not nominated for redefinition. In this case, `HotSwapClassLoader` calls the `loadClass()` method of the original class loader `CL` (see above). This method will either return the already loaded common superclass of both the old and the new class versions (class `C2` in Figure 7.8(b)), or will load the completely new class, that was inserted into the hierarchy (class `D` in Figure 7.8(c)). In the latter case, loading will continue recursively until a common superclass for both versions is reached (class `C2` in Figure 7.8(d)).

[9]Note, however, that to optimise the performance and continue to provide support for debuggers, we still use the technology of method only replacement, as implemented at stage 1, if only the bodies of a class's methods are actually changed. Therefore, the eventual shape of the class hierarchy may be a combination of those presented in Figure 7.7 (a) and (b). But while we are loading the classes, we don't know exactly how each class was modified, so we have to use the conservative strategy (b).

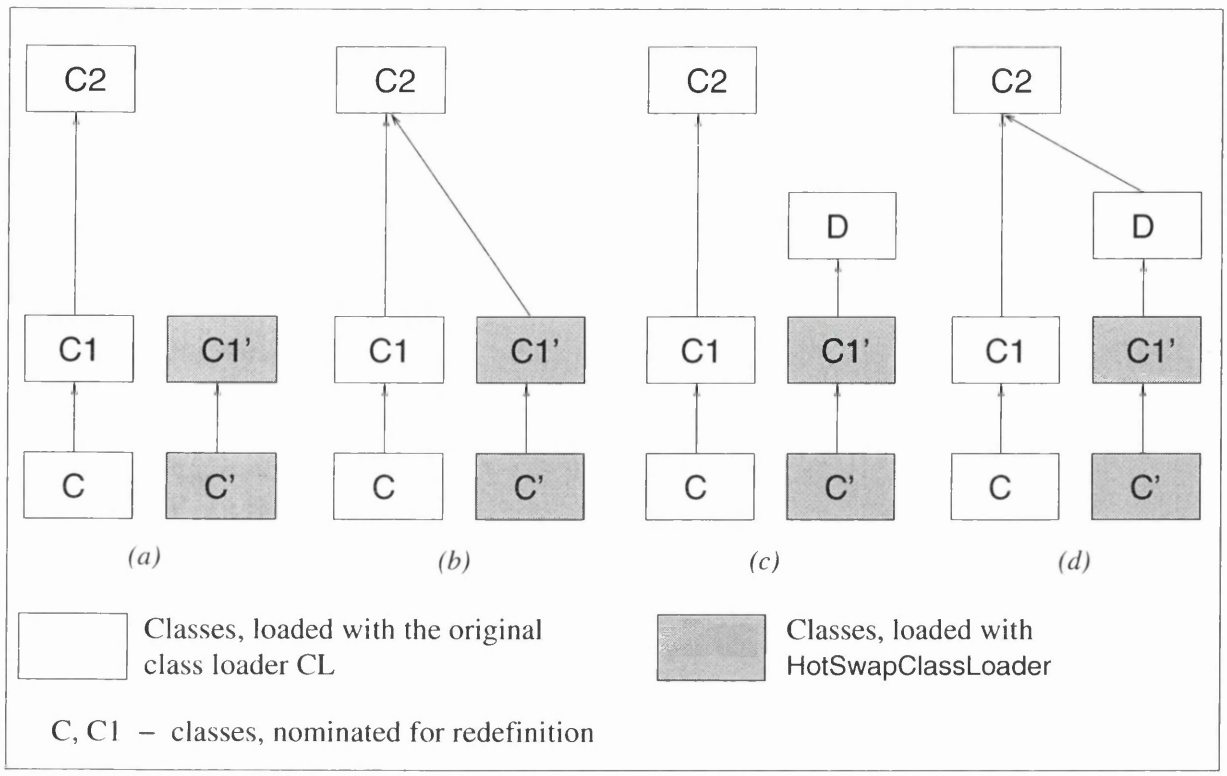

Figure 7.8: Loading new class versions.

7.4.2 Change Validation and Replacement Method Determination

After all of the new class versions are loaded, we compare each pair of class versions. This procedure has two goals:

1. Validate the changes, ensuring that they are binary compatible. If any change is incompatible, abort the whole operation, i.e. do not change any classes and return the error code.
2. Determine the replacement method to be used for each pair of class versions.

The validation procedure checks that all of the changes to the class are binary compatible, as specified in the JLS, Chapter 13. In addition, we should check that the instance format has not changed, since at present we don't support instance conversion. The change validation algorithm takes the following form:

1. Compare the class modifiers:
   (a) Check that access modifiers for the new class version are not more restrictive than for the old one;
   (b) If the new class is `final`, check that the old class is also `final`. The same with the `abstract` modifier.

2. Check the superclasses: make sure that the old superclass is one of superclasses of the new class version. Class names instead of class objects themselves are compared, since an arbitrary mix of old/new/unchanged classes is possible at this time.

3. Check the interfaces. New class version should implement, directly or indirectly, all of the interfaces that the old one implements directly. Again, we compare the names of the interfaces rather than the class objects for them.

4. Check the fields defined in the class versions:

   (a) For the instance fields, the number, names, signatures and offsets within a class instance should be exactly the same in both versions. Access modifiers can be changed to be less restrictive. If some field is `final` in the new class version, it should also be `final` in the old class version.

   (b) For the static fields, the order is not important. `private` fields can be deleted freely, and new fields can be added. Each non-private field in the old class version should have a counterpart with the same name and signature in the new class version. For such fields, access modifiers can be changed to be less restrictive. If some field is `final` in the new class version, it should also be `final` in the old class version.

5. Check the methods and constructors (the latter are treated internally as static methods with a same special name). The order of methods is not important. Private methods can be deleted, and new methods can be added. Each non-private method in the old class version should have a counterpart with the same name and signature in the new class version. For such methods, access modifiers can be changed to less restrictive. If some method is `final` in the new class version, it should also be `final` in the old class version. The same with `abstract` modifier.

In parallel with change validation, we collect information that determines the replacement method to be used for each pair of class versions. Currently two such methods are available. The first is the one which we implemented at the previous implementation stage, and is described in Section 7.3.4. It is used if we determine that only the method bodies of the given class have been changed. Otherwise, the second method is used, which involves complete replacement of the class object. It is described in Sections 7.4.4 and 7.4.5.

7.4.3 Unregistering and Re-registering New Classes

After all of the new classes are loaded and the changes are validated, we need to assign the original class loader to each new class version, so that it can correctly substitute the old class version. This operation requires manipulations with certain internal JVM structures, and can lead to creation of new Java objects, so it can not be performed in the system thread, where garbage collection is prohibited.

When a new class version is loaded with `HotSwapClassLoader`, it is registered with this class loader in the internal JVM dictionary that maps "class name – class loader" pairs to class objects. In addition, the class object points to the class loader, and the latter points to the class from its class cache. Therefore to make new class versions valid substitutes for the original classes, we do the following:

1. Remove the original and the new classes from the dictionary;

2. Switch the pointer in the new class version from `HotSwapClassLoader` to the original classloader for the old class version `CL`;

3. Re-register the new class version in the system dictionary, with `CL`. This operation will also add the new class to `CL`'s class cache.

4. Check the superclass `S` of the new class. If its actual class loader is `HotSwapClassLoader`, there are two possibilities: this is a new version for a class that is itself nominated for redefinition, or it is a new class that has just been inserted into the class hierarchy. If among the classes nominated for redefinition we find a class with the same name `S`, we stop this local class loader fix up recursion, since `S` will be dealt with in its own time. Otherwise we go to Step 1 (though there will be no "original" class on the subsequent iterations), and repeat this and the following steps recursively until the class loader for some `S` is not `HotSwapClassLoader`.

Note that we do not delete the old class version from the `CL`'s class cache. This is done, as in the alternative procedure where only methods are replaced, to prevent garbage collection of methods of the old class version which may potentially remain active forever, and also to facilitate the "undo" operation.

7.4.4 Subclass Expanding

For the same reason as class unregistering, this operation is also performed in a Java thread, i.e. with garbage collection enabled and other threads running in parallel.

Special treatment of subclasses is required because each class has at least as many entries in its VMT as its superclass, and because HotSpot VM stores VMTs inside class objects (see Figure 7.4). Therefore, once the VMT becomes larger in some class, we have to replace the whole class object for this class, and we then have to do the same for all of its subclasses. The difference between class redefinition and subclass expanding is that for a class being redefined the standard mechanism of class creation out of bytecodes is used, whereas for its subclasses, to save time and memory, we use our own non-standard "class object expansion" mechanism.

This mechanism works as follows. We first recursively collect all of the subclasses of each class nominated for redefinition. These subclasses are placed in a special array. The collection procedure is organised such that classes appear topologically sorted in this array, i.e. a superclass always occupies a position with a smaller number than its subclass. We then scan this array sequentially, and for each class (old class version) in it calculate the `klassVTable` and `klassITable` (see Section 7.1.2.5) sizes for the new class version. We then allocate memory for the new class version object and copy most of the information into it from the old class version. Virtual method tables, however, are filled in based on the information in the superclass new version (which is already available by that time, since the array is topologically sorted).

Finally, we unregister the old class version and register the new one with the class dictionary, similar to the way in it was done with the classes nominated for redefinition.

7.4.5 Class Replacement

As already mentioned, class replacement at implementation stage 2 can be performed according to one of two methods, depending on the nature of changes to the class. If the changes involve only method bodies, class transformation is performed exactly as described in Section 7.3.4. Otherwise, the operations described below are performed.

1. Deoptimise any compiled methods of the class being redefined (exactly as in item 2, Section 7.3.4).
2. Deoptimise all of the compiled code that depends on the class being redefined (exactly as in item 3, Section 7.3.4).
3. Determine the common (for the old and the new class version) subsets of static variables and methods
4. Copy the values of the common static variables from the old class version to the new one.
5. Iterate over all of the classes currently loaded, except Java core classes, which are guaranteed to not reference our evolving application classes. For each class C, scan its constant pool cache and replace all pointers to old methods of the evolving class with the pointers to the respective new methods. Do the same with the pointers to the static fields of the evolving class.

Once the above procedure is complete for all redefined classes, we perform the last phase of class replacement — instance rebinding. We iterate over all of the objects in the Java heap, and for each of them that happens to be an instance of a class nominated for redefinition and replaced as above, or a subclass of such a class, we switch the class pointer to the new class version.

Note that the described complex procedure can in principle be avoided in the intermediate case, i.e. when changes to a class involve more than just the bodies of its methods, but the number of its virtual methods and the number of the static variables it defines remains the same. In that case, we can avoid creating a new class object for the new class version, subclass expanding and instance rebinding for all of these classes. We will have to replace the method array and the constant pool for the class, as we do in the method only class transformation, and we will have to modify in place the VMT and the array of static variables for this class and its subclasses. This optimisation may be implemented in future.

7.4.6 Summary

At the implementation stage 2, our runtime evolution technology should support any binary compatible changes to classes, excluding those that affect the format of class instances. The implementation is now complete, and we are testing it, currently using synthetic tests. Once it is tested properly, we plan to start experimenting with "real-life" applications.

7.5 Future Work

7.5.1 Stage 3: Support for Binary Incompatible Changes

We have already mentioned in Section 7.4 that binary incompatible changes can potentially break a number of internal JVM invariants, e.g. the rule that a `final` class should not have subclasses or that there should be no calls to `abstract` methods. Invariants like these are normally checked by the JVM during class loading and linking (see JVMS, §2.17.2 and §2.17.3, respectively), and these checks will automatically be performed for a new version of a class being redefined, once this new version is loaded. However, since the old version of this class has been inside the JVM for some time, there may be other classes that use it, i.e. have symbolic links or already resolved, physical pointers to this class or its members. Binary incompatible changes mean by definition that some of these links may now become invalid, so we need to verify if any constraint is actually violated, and abort the whole operation if this is the case.

The reader can observe that this is problem is very similar to the one which we encountered in PJama, and which we solved by using selective class recompilation (see Section 3.1 for details). However, there are a number of differences between the PJama context and the dynamic class redefinition, which do not allow us to re-use the same solution, at least for all possible change policies. The most fundamental difference is that in PJama we operated on a quiescent store, with the persistent application shut down, and could thus replace one consistent set of classes with another consistent set, and then re-start the application. When we dynamically replace a class, we, in the general case, still have some old code that has not yet completed execution (and may never terminate). So, in the general case our new class versions have to be consistent not only between themselves, but also with the old class definitions. There are also less fundamental considerations, e.g. about whether or not the JVM can, in an arbitrary case, trust the evolution tool (which may come from a different vendor) in its ability to provide a consistent set of classes, etc. Therefore, it currently seems that the safest option is to repeat the standard checks performed during linking for all of the classes that use the class being redefined, to make sure that none of the binary incompatible changes to this class introduces any actual problem.

This is unlikely to be an easy task. To write our own verification code, which does essentially the same job as the already existing standard code, is obviously not a good solution. On the other hand, if we try to re-use the standard JVM class verification code, we can potentially expect problems that are due to the difference in class representation at the time when this code is normally invoked and at the time when we want to re-invoke it. In the latter case, method bytecodes, for example, are already rewritten for speed-up, whereas the verification code might need to analyse the original bytecodes. Another problem is that we have to avoid making any classes point to the new copy of the redefined class until all of the verification procedures have completed successfully — but these procedures themselves normally require the class being verified to point directly to other classes (at least to its superclass). Therefore, it looks as if in order to implement support for binary incompatible changes, we will have to carefully inspect, and possibly patch, the standard verification code, to make it work in non-standard circumstances.

There is also a question of when to report any detected problems. According to the JVMS, §2.17.3, if an error occurs during resolution, an exception is thrown at the first point in the program that actually uses a symbolic reference that caused the problem. So, for example, if the code contains a `new` operator for class `C` that was made `abstract`, it will start and run without problems up until the moment when this operator

is to be executed. Only at that point an `InstantiationError` will be thrown. If `C` had originally been non-abstract and was changed to `abstract` by our redefinition mechanism before a particular `new C()` operator was reached, the JVMS-compliant behaviour looks consistent. But when should the VM report the problem with the `new C()` operator for class `C` redefined to `abstract`, if this operator was already executed successfully one or more times? Clearly, this is out of scope of the JVMS, and it is yet to be specified what should happen in this case. The author's opinion is that the problems like this one should be checked and reported eagerly, to ensure maximum change safety — which will again require re-using the existing verification code in a non-standard way, or writing our own code.

7.5.2 Instance Conversion

Once we allow the instance format of a class to change, we would have to implement instance conversion. We believe that the same kinds of conversion as in PJama, i.e. default and custom conversion (see Section 4.2), should eventually be implemented. At present we can hardly imagine making as complex changes to classes at run time as in the case of persistent application evolution, since making a complex change at run time would require the engineers to take many more possible effects into account. Thus it will probably be enough to implement only bulk (Section 4.5) and/or dictionary-assisted (Section 4.10.1) forms of custom conversion.

Lazy conversion implementation, which is probably even more desirable in the context of runtime evolution, since it minimises latency, seems very problematic for a handleless VM such as HotSpot. If the new object version is larger than the old one, the object has to be relocated. Since the pointers to this object are direct, all of them have to be located and patched at once. This operation would have a prohibitive cost if it is applied to each object individually, over and over again. An alternative solution may be to put a forwarding pointer to the new object at the old object location, and modify the VM such that on every object access it checks for a forwarding pointer and follows it. The GC code will have to be modified too, so that it takes forward pointers into account. But even if we find a convenient way of distinguishing between the normal object and the forward pointer, this solution will still have a major drawback of increasing the cost of each object access and thus the (probably significant) VM slowdown. Thus, it may be more feasible to implement eager concurrent conversion, which would work in much the same way as incremental concurrent garbage collection (and can actually re-use the code of the latter). Such a mechanism will convert objects in parallel with the normal execution of the evolved application, and will only block an application thread if it tries to access an object which has not been converted yet.

Eager conversion implementation seems to be easy enough, provided that we find a way to re-use the existing GC code that relocates objects. Whether conversion scalability problems, which required a significant effort to overcome in PJama (see Chapter 6), are going to be an equally serious issue in case of runtime evolution, is to be investigated.

7.5.3 Implementing Multiple Policies for Dealing with Active Old Methods of Changed Classes

Several policies for dealing with active methods of old class versions, that we can think of, were first briefly presented in Section 7.3.1. We will now discuss possible ways of implementing these policies.

7.5.3.1 On-the-fly Method Switching

This policy means that we identify a point in the new method that "corresponds" to the current execution point in the old method, and then continue execution from this point in the new method. The main conceptual issue here is how to define the above "correspondence". The answer that we suggest is to compare the bytecodes of the old and the new methods, in much the same way as utilities like Unix `diff` compare texts. We should, of course, take into account (as we do when we compare methods for equivalence, see Section 7.3.5) that the constant pool layouts and thus indexes into the constant pool may be different in method versions. Textual comparison of the bytecodes will identify the minimum differences and, consequently, the matching segments. If the current execution point is in such a segment, we can find the matching point in the new bytecode and switch the execution to it.

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

The above method of establishing matching points in bytecode versions does not, of course, guarantee the "correct" execution of the program being transformed — as our technology in general does not guarantee it. However, we currently can identify one case when such a transition is likely to be harmless. That is the case when the changes to the bytecodes in the new version are purely additive, and the added code does not produce any known harmful side effects. Examples of such modifications are adding code that prints some tracing information, e.g. to help identify a bug, or supports profiling (see Section 7.5.4). Other cases, where an experienced developer is likely to be able to predict the effects of a transition, may also be possible. In any case, in the end we have to trust the skill of the software engineer initiating the change, but we would like to provide as much support for them as possible.

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

Identifying common fragments in method versions is not the only problem with implementing this mechanism. The new method code will inherit the current method frame (JVMS, §3.6) from the old method, that is already populated with local variables and the operand stack (JVMS, §3.6.2) contents. Obviously, the execution can continue correctly after the transition only if the old, inherited frame layout in the transition point is the same as the new code expects (or we can somehow "re-shuffle" the frame contents to make them consistent with the new code). This layout includes the number, names, types and order (i.e. slots allocated) for the local variables, and the number and types of the operands currently in the operand stack. The max-

imum depth (JVMS, §3.6.2) of the operand stack may also be important, if a JVM allocates Java operand stacks as fixed-size frames[10]. Therefore, if the maximum stack depth changes for the substitute method, we should either reject it or take care about expanding the window for the operand stack.

Furthermore, to support following the pointers from the Java stacks during GC, HotSpot prepares stack maps for each method, that contain pointer locations. They may also need to be updated for the substitute method.

Whether all of the details of the frame layout should be checked in the general case, how to do that, and whether supporting this general case is worth the effort, remains a question. However, temporarily leaving most of the above safety issues behind, we have, as a matter of experiment, implemented the functionality that can substitute methods and switch execution on-the-fly. In the simple tests, where we were just adding `System.out.println()` statements to our code, and did not introduce any changes to the stack layout, it worked well for us. Thus, we believe that at least limited applications of this technology guaranteed to not violate safety restrictions, e.g. the one discussed in the next section, and also dynamic fine-grain profiling discussed in Section 7.5.4, can be utilised almost immediately.

7.5.3.2 Wait Until No Active Old Methods

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

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

Method entries and exits can be tracked using, for example, the already available mechanism of debugging events generation. However, this mechanism, as well as the rest of the debugging functionality, works only in the purely interpreted JVM mode, while we want our technology to work for the compiled code of server-type applications. Debugging events generation is also expensive, since the event is generated upon entry into and exit from *every* method. One possible alternative may be to implement this policy using the mechanism of on-the-fly method switching discussed in the previous section. This way, method redefinition will consist of the following two stages. On the first stage, we patch the code of the old methods, adding to them the calls to two methods predefined in our own special class. These calls increment and decrement a counter, which is initially set equal to the total number of activations of the old methods. The method that decrements the counter should check if it becomes equal to zero. When this happens, it should call the standard class redefinition procedure, that will redefine the classes as originally requested by the user.

Alternatively, at least in the case of a single-threaded application it is possible to patch only the outermost

[10]Fortunately, it is not the case with HotSpot, which allocates Java operand stacks on the top of the native stack, thus allowing the former to grow freely.

activation of the old method `m()`, replacing on the stack the original return address in this activation with the address of our own code. The latter will perform class redefinition and then resume the normal execution from the saved original address. This way we will avoid the overhead of the calls that increment and decrement the counters.

7.5.3.3 Old Threads Call Old Code, New Threads Call New Code

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

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

It is currently unclear to us how, if possible at all, to determine whether or not an arbitrary thread may call the given class. Perhaps the developer can specify this explicitly, or perhaps the overhead due to call dispatching will be tolerable, and thus the synthetic class can be used as an ultimate new class version.

7.5.4 Dynamic Fine-Grain Profiling

One interesting application of class redefinition technology and the on-the-fly method switching policy may be dynamic fine-grain profiling of Java applications. The standard profiling mechanism presently available in HotSpot JVM is called Java VM Profiling Interface (JVMPI) [Sun00o], and its architecture has much in common with JPDA (see Section 7.3.2). This mechanism has a granularity limited to a method, since it only generates an event on method entry and exit. This mechanism is also somewhat expensive, since it is only possible to configure it to generate an event on entry into/exit from *each* method. Thus, if the user wants

to profile just one method, it is the profiling tool's responsibility to extract the required information out of the flow of incoming events. To the best of the author's knowledge, no other JVM at present implements anything more sophisticated. Therefore, if the user wants to measure the time spent in a code section smaller than a method, the only solution is to explicitly bracket the measured code with the user's own measurement code, e.g. calls to `System.currentTimeMillis()`, recompile and re-run the application. Obviously, this is a very tedious procedure, especially when it is used to gradually narrow down the suspicious code area in order to find out which piece is a bottleneck.

Our class redefinition mechanism can be used to implement cheap, dynamic fine-grain profiling by essentially automating the above procedure[11]. Calls to an equivalent of `System.currentTimeMillis()`[12] can be inserted into the original method bytecode, around the code section that the user wants to measure. The `return` statements inside this code section, which is a headache if this procedure is performed manually, can be handled automatically. Then, an original method can be replaced with the patched one, such that, if necessary, the execution switches to it on the fly, as discussed in Section 7.5.3.1. This can be repeated an arbitrary number of times during the same session, allowing the user to change the profiled code section(s) and observe the results without interrupting the running application.

7.6 Related Work

A number of techniques for dynamic evolution of software applications exist at present. In the following discussion, we loosely classify them according to the semantics of changing code and the programming interface.

7.6.1 Dynamic Class Versioning

In in the work by Hjalmtysson and Gray [HG98], dynamic classes for C++ are described. A dynamic class is a class, written in accordance with the special rules, whose implementation can be changed dynamically during program execution. However, its external interface may not change. Furthermore, the link between an existing object and its defining class may not be changed. This leads to multiple versions of the same class coexisting simultaneously, with objects being partitioned between them.

This technique intentionally does not involve any modifications to the runtime system or language extensions. Instead, a proxy (wrapper) class, associated with every dynamic class, dispatches method calls to an appropriate class version depending on the object on which a method is executed. A proxy class itself is an instance of a generic template class. Instances of a dynamic class are created using the latest version of this class, by invoking a constructor of the proxy class, which in turn calls a factory method of the dynamic class (which should necessarily be provided by the programmer).

This technique has a number of drawbacks. It requires that the application itself provides the mechanism, e.g.

[11] The author originally discovered this solution himself — only to find out later that some Smalltalk system(s) used to use the same mechanism.

[12] For more precise measurements, and to avoid the overhead of extra Java calls, we can actually insert a special bytecode, that would directly call an appropriate C function.

an external event handler, that would respond to some particular user action and load the new class version on demand. It requires any dynamic class to be written as two classes: an abstract one that defines the interface, and its descendant(s) that implement this interface and are effectively dynamic class versions. Only a single constructor is allowed for a dynamic class, and there are certain complications with inheritance from such a class. Static method redefinition cannot be supported by this mechanism. Two-level call dispatching by the proxy class imposes an overhead on each dynamic class method call, though the authors claim that this problem is not going to be serious, since, at least in their application domain (programmable networks), most of the dynamic classes are not the finest-grained classes in the system, and their methods are called relatively rarely. If a similar system were implemented for Java (there should be no difficulties with this), the performance penalty would likely be higher, since the cost of an additional method call is much higher in Java than in C++.

Overall, this technique is probably the best solution for C++ as a compiled language with inherently little or no possibility to inspect and modify a running application. However, in our opinion, it is too primitive for Java, considering what can be achieved by modifying the JVM.

7.6.2 Load-Time Transformation

Several projects, e.g. Binary Component Adaptation [KH98], JOIE [CCK98], Javassist [Chi00], and OpenJIT [OSM+00] exist, that support modification or generation of classes at load time (before or during class loading). The primary use of this technique is to optimise or reconfigure applications by generating specialised classes, or instrumenting the existing ones. For example, a system for load-time transformation of Java classes may allow the developer to change the class name, add an implemented interface, add a method, or perform certain modifications to an existing method. This functionality may be useful in some situations, for example to facilitate evolution of Java interfaces. Adding a method to an interface is a binary incompatible change, and all classes that implement this interface should be changed at once to implement the added method. If at least the default implementation of the added method is simple enough, e.g. can be written using only the methods previously defined in the same interface, a load-time class transformation system can solve the problem of interface evolution by synthesising, at load time, such a default method for each class implementing the modified interface. Furthermore, this can be the only solution if the source code for classes is not available.

However, this method's fundamental limitation is that it is not possible to modify classes and objects already present in the VM. Also, to modify a method, such systems typically require the developer to specify changes at the bytecode level. Alternatively, some of them provide an inevitably limited number of higher level primitives, such as "change a field access expression to access a different field" or "replace a `new` expression with a `static` method call".

7.6.3 Dynamic Typing

Smalltalk [GR83, Gol84] and CLOS [Sla98] support dynamic typing, that allows both the class itself and the pointer from an instance to the class to be changed freely at run time. Data fields and methods may be added or removed, method code can be changed, and so on. This evolution mechanism can be used both internally

from within the program (i.e. the code can modify itself), or externally (e.g. from an IDE), as in our system. There are virtually no constraints on the operations that can be performed. This, however, means that no type safety checks of any kind are performed by the system when a change is made, and any unsafe changes, such as deleting a data field or method which is still used by the code, can be made. For these languages this is not considered a problem, since, in contrast with Java, they are type unsafe from the beginning, and their runtime system supports complete runtime type checking (with all the associated overhead). The latter, however, guarantees just that an error made during class redefinition will manifest itself later as a "graceful" runtime exception, rather than strange program behaviour or runtime system crash.

As for Java, so far very little has been done in the area of truly runtime (not class load time or link time) class redefinition. Presently we are aware of one example [MPG+00]. The system was implemented for Sun Classic JVM, JDK1.2. It supports only binary compatible changes to classes, but also allows the developer to modify classes such that the format of their instances changes. The latter is backed by the mechanism of essentially lazy conversion, which is facilitated by the presence of handles in the Classic JVM. Thus, instance conversion happens in two phases. On the first phase, all of the instances of the modified class are eagerly located, a special flag is set in the instance header to indicate that the instance should be converted, and the class pointer in this instance's handle is switched to the new class version. The second phase is lazy: an instance is converted (using default rules) and relocated when the JVM tries to access it. Upon an instance relocation, only the pointer to this instance from its handle needs to be changed.

When reported, this system had some serious limitations. Classes that have methods active at the moment of evolution, simply can not be evolved: the system throws an exception and it is up to the developer to handle it. Evolution support for compiled methods is not available, thus the dynamic native compiler in the JVM is disabled. Only classes loaded by a special class loader (which also provides an API for class redefinition) can be redefined. Though the latter design feature is intentional, we suspect that it may not be very practical if the system is used for existing applications.

7.7 Summary

In this chapter, we have described the technology for runtime evolution of Java applications (dynamic class redefinition), that allows developers to modify running Java applications. The technology is being developed for the HotSpot JVM, Sun's main production JVM at present. The most important features of this JVM, and their effect on the runtime evolution implementation, were discussed in Section 7.1. We then introduced our plan of staged implementation of our technology, where each stage corresponds to some, relatively consistent, level of functionality. Since our technology allows developers to evolve classes whose methods are currently active, we have also suggested several possible policies for dealing with such methods. These policies are independent of the functionality levels corresponding to the above implementation stages. The same is true about the support for instance conversion, which can be introduced at any stage except the first one.

In the first stage we allow the developers to modify only the method bodies of classes. We support only one policy for dealing with active methods: "active calls to old methods complete, new calls go to new code". This functionality is completely operational now, and will be included in the forthcoming release of HotSpot/JDK (JDK1.4), scheduled for the end of year 2001.

In the second stage, which is now close to completion, we support less restrictive, though only binary compatible, changes to classes. In the third stage, we are planning to support all possible changes, including the binary incompatible ones, provided that they are type safe in the context of the current running application. We expect that it will not be easy to implement type safety verification, since in the context of the running application we will not be able to re-use our solution with smart recompilation (Chapter 3), which worked well for us in PJama.

In Section 7.5 we presented our thoughts on the following aspects of the future work: support for binary incompatible changes, instance conversion, and implementation of the remaining policies for dealing with active methods. We also suggest that the mechanisms developed for dynamic class redefinition can be re-used to implement dynamic fine-grain profiling of Java applications.

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

Chapter 8

Review, Conclusions and Future Work

In this chapter, we first review the thesis (Section 8.1). In the next section, we present our thoughts on the future work. Its directions that we propose include runtime evolution of Java applications (this work is actually in progress now), design of VMs supporting runtime evolution, and further development of ideas related to smart recompilation and object conversion. Finally, we present conclusions

8.1 Thesis Review

This thesis addressed three issues, that are all relevant to management of changes to large and long-lived, or enterprise, Java applications: evolution of classes, evolution of persistent objects, and runtime evolution of applications. Our goal was to design, implement and assess powerful and flexible technologies that would:

- Allow the developers to perform any kinds of changes to applications and persistent objects.
- Scale well.
- Guarantee maximum possible level of safety of changes.

The following aspects of change safety were addressed:

- Type safety. Any change to a Java class which may potentially violate Java's type safety invariants, should be checked to ensure that it is deleterious in the context of the particular application being evolved.
- Persistent store (database) transformation safety. The underlying support mechanism should guarantee that links between objects can not be re-arranged such that type safety may be violated. If a Java runtime error or a system crash occurs mid-way through the conversion process, objects must not be left in half-transformed, and thus effectively unusable state.

The work that has led to solutions addressing evolution of Java classes and persistent objects, was performed in the context of the PJama orthogonally persistent platform. For it, we have implemented the following:

- The technology which we call *persistent build*, that supports evolution of persistent classes (that is, their physical replacement in the persistent store), and ensures type safety of changes with the help of smart recompilation. For PJama, it was implemented in the form of a standalone "build and evolve" tool called `opjb`. However, our technology of smart recompilation for Java is general and independent of any particular Java compiler. Thus, it can be easily re-used with any IDE for Java, or in a standalone Java-specific utility similar to `make`.
- The technology for powerful, flexible, scalable and recoverable conversion of persistent objects. Power and flexibility are achieved by providing several conversion types: *default* and *custom*, and within the latter — *bulk* and *fully controlled*. While default conversion completely automates simple object transformations, advanced custom conversion forms support arbitrarily complex changes to persistent objects and their collections. This is achieved first of all by using a special mechanism of temporary renaming of old versions of evolving classes. It allows the developers to write conversion code in Java, that can manipulate with an arbitrary mix of old and new classes and their instances. While the presently implemented variant of this mechanism introduces certain non-standard extensions to the Java language, we also suggest an alternative design where the language modification is avoided — at a price of somewhat reduced developer's convenience. We believe that a similar mechanism can be very useful, and not difficult to implement, for other persistent object solutions for Java.

  Scalability and recoverability properties of our system, that were implemented in the underlying persistent store system, Sphere, allow the developers to convert persistent stores of arbitrary size with limited size main memory, and guarantee that evolution either completes successfully or is rolled back.

PJama evolution system was evaluated on a large number of synthetic tests and on one relatively large persistent application, a geographical system called GAP. This work resulted in a better understanding of the evolution problems and in a number of improvements to our system. The experiments we performed have confirmed the scalability of our system, proved the conversion time to depend linearly on the number of evolved objects, and proved the ability of the system to evolve large amounts of data successfully.

The work on runtime evolution of Java applications was performed in the context of the HotSpot JVM, which is the present production JVM of Sun Microsystems, Inc. The problems of class and object evolution, such as type safety preservation and low-level link rearrangement, become much more complex once they are considered in the context of a running application, that has active methods of old class versions, multiple physical links between classes, objects, and the VM internal structures, etc. Therefore, a plan of staged development was devised, where each stage corresponds to some, relatively consistent functionality level, that can be included in a release of the JVM. From the beginning, this functionality should allow engineers to change Java applications that run in production (as opposed to debugging) mode, and may have active methods for classes that are being redefined. We have suggested several policies for dealing with active old methods, each of which may be more suitable for a particular class of applications.

The access to this functionality is currently through a single call in the JVM Debugging Interface (JVMDI) C language API. It is planned that in future a special GUI evolution tool, similar to existing debugging tools,

will be developed. As with the debugging tools, it would communicate with the JVM through a remote connection, allowing the developers to connect to a running JVM and modify the application that is running on-the-fly.

In the first implementation stage, which is now complete and operational, our technology supports changes only to method bodies, and provides a single policy for dealing with active methods, that allows all of the calls to old methods to complete, while dispatching all of the new calls to new methods. In the second implementation stage, which is now close to completion, we will be supporting any binary compatible changes to classes, that do not require instance conversion.

8.2 Future Work

8.2.1 Runtime Evolution — Work in Progress

The work on runtime evolution continues. After finishing and testing the implementation stage 2, we will consider several possible directions of future work.

- Proceeding straight to stage 3, in which the system should support any changes to classes, including those that are binary incompatible, provided that they do not violate type safety invariants for the particular application being evolved (Section 7.5.1).
- Implementing object conversion in the context of runtime evolution, thus making the system support changes to classes that affect the format of their instances (Section 7.5.2).
- Implementing additional policies for dealing with active old methods (Section 7.5.3).
- Re-using the existing and planned technology to implement dynamic fine-grain profiling for Java applications (Section 7.5.4).

To determine the highest priority direction from the above, it would be desirable to first experiment with the technology that is already available. Obtaining an application that can be considered "real-life", and trying to improve it using this technology, may be very effective in determining what needs to be improved, as it was the case for PJama evolution technology, when we used it in the course of our work on improving the GAP system.

We have already experimented with small changes to applications, mostly those that were free of deleterious side effects, and believe that even this level of functionality can be quite useful in certain situations, e.g. when it is necessary to diagnose a bug without stopping a running application, or tune it for better performance. However, so far it is difficult for us to imagine how serious changes to running applications can be made — mainly because we don't know how a developer can predict and take into account all of the possible effects that a change at an arbitrary moment of time can produce. Thus, we believe that in the long-term prospective it would be very important to think about some mechanism that would automate the prediction of a useful proportion of the effects of runtime evolution (at least the known undesirable ones) and possibly help to eliminate them.

8.2.2 Development of VMs that Support Runtime Evolution

At present we are working with the JVM which was not originally designed to support runtime evolution. It seems primarily a lucky coincidence (but also perhaps its developers' foresight due to their experience with SELF) that it has some properties, such as the support for deoptimisation of compiled code, which make runtime evolution in production mode possible. On the other hand, other properties, such as the monolithic internal representation of class objects, makes runtime evolution implementation more difficult, and will inevitably affect its performance (see Section 7.1.3 for details).

Currently we can formulate some requirements which are worth taking into account when starting to develop a new VM, if it is planned that it is to support runtime evolution. Much the same requirements can also be applied to design of persistent stores which may contain classes for persistent objects.

Our most important observation is that, whenever possible, monolithic and hence non-malleable internal representations of classes should be avoided. Data structures that represent classes occupy a small amount of space compared to instances, and access to most of them (except VMTs and structures similar to constant pool caches in HotSpot), happens relatively rarely. Therefore space and performance gains due to their tight packing is likely to be negligible, whereas the general maintainability of the system, and its ability to accommodate extensions that support evolution, can suffer significantly. This was our experience with representation of classes in the original persistent store for PJama (see Section 2.1.3.4), and then with HotSpot (see Section 7.1.2.5). Once a class is represented as a collection of separate entities that can be replaced individually, evolution implementation becomes much simpler. Its performance also improves significantly, since many extensive pointer rearrangement operations become unnecessary. For the last point, it is most important to design the class representation such that it is not required to patch pointers from instances to their class when the latter is changed. An exception to this rule may be the case when a change to the class requires instance conversion — then the instances typically need to be relocated anyway. In Sphere persistent store this was achieved with the help of descriptors (see Section 6.1), and in a VM for an object-oriented language it may be solved by using data structures such as *near classes* used in EVM [WG98].

Another requirement is that whenever certain optimisations to the VM, such as replacing some bytecodes with the "quick" versions or replacing symbolic links with direct pointers, are introduced, an easy way to find out what the initial representation for an optimised item looks like, should be provided. In HotSpot the presence of such a support in the constant pool cache facilitated our work a lot (see Section 7.3.4), whereas, for example, in the Sun's Classic JVM it was impossible to find out to what original constant pool item an argument of a "quick" bytecode corresponded.

More requirements will probably be formulated in the course of future work on runtime evolution support in HotSpot.

Considering support for evolution of persistent objects, one problem at present is evolution of Java core classes (see Section 2.5.3). These classes may have persistent instances, however their evolution is beyond the control of the platform's users. Thus, it should be the Java platform vendor's responsibility to provide the conversion code, or at least the specification of how instances of these classes should be converted, if definitions of any of them are changed.

8.2.3 Other Directions of Future Work

Other directions of future work may include:

- Work on smart recompilation for Java and other languages. A short-term goal may be to develop a standalone `Javamake` utility (see Section 3.5.2) that uses a method for determining classes to recompile which is similar to the one presented in this thesis. Other methods, e.g. the one presented in Section 3.5.1, may be subsequently considered for implementation, depending on the performance of the initial implementation.

  A long-term goal may be to try to devise a formal proof of completeness and correctness of the data contained in our tables of source incompatible changes (Section 3.5.3). It would be most valuable if the resulting apparatus could be then used for other languages, and/or a formal methodology for determining incompatible changes and affected programming modules in any language could be developed.

- Work on persistent object conversion. In the short-term prospective, it would be very interesting to try to implement a solution for Java object conversion that we proposed in Section 5.7.2. This solution exploits our idea of old class version renaming, which is a key to powerful and flexible custom conversion mechanism. Yet it does not use any extensions to the Java language (unlike our current solution), and therefore has much more chance to gain acceptance with the other persistent object solutions for Java.

  In the longer-term prospective, exploration of alternative conversion strategies, e.g. custom lazy conversion or eager concurrent conversion (see Section 4.10), as well as the evaluation of the usability aspects of various conversion methods, can be an interesting research topic.

- So far our evolution technology is oriented more towards development-time evolution, rather than deployment-time (see Section 2.2). An industrial-strength persistent platform will need support for the latter, since for it the developers and users would certainly be different people, and the users may not be as skilled and knowledgeable about the system internals as the developers. Therefore the facilities would be required, that would automate higher-level aspects of evolution procedures and help the users to plan and conduct changes.

8.3 Conclusions

Long running applications and applications combining software with long-lived objects are now common place. All of these applications require better support for change. The evolution technology pioneered in this thesis is, we believe, a significant step in the long journey of accommodating change in software systems.

Appendix A

Command Line Options of the PJama Persistent Build Tool

The persistent build tool for PJama, called `opjb`, is invoked as follows:

```
opjb [options] [.java files] [classes] [@files]
```

The arguments can be in any order, except that the `-store` option, if present, should always be the first (this is the requirement of the underlying PJama system itself). The tool always runs agains a persistent store, either explicitly specified with the `-store` option (see below), or implicitly specified through the environment variable, `PJAVA_STORE`. To shorten or simplify the `opjb` command, one or more files may be specified that themselves contain one argument per line, using the `@` character plus the filename.

If any `.java` files are specified, they will be compiled, and, if compilation is successful, the resulting classes will be added to the Evolvable Class Directory (ECD) of the current persistent store. If any classes are specified, they will also be added to the ECD.

`-store <store>, -config <store>`	specifies the persistent store to be used (full path required)
`-delete <class>, -del <class> [-mig <other_class>]`	specifies a persistent class to delete, and, optionally, a class to which to migrate the 'orphan' instances
`-replace <old class> <new class>, -rep <old class> <new class>`	specifies a persistent class to replace with a new class

`-sns`	a flag that can be used after the name of a class that is being substituted or replaced, to indicate that values of its `static` non-final variables should not be copied from the old class to the new one
`-scf`	a flag that can be used after the name of a class that is being substituted or replaced, to indicate that values of its `static final` variables should be copied from the original class to the substitute one
`-convclass <conversion class>`	specifies a conversion class
`-classpath <directories separated by colons>`	specifies the path in which to look up the `.class` files. Otherwise the current value of the `CLASSPATH` environment variable is used.
`-d <directory>`	specifies the directory in which to place generated class files
`-sourcepath <directories separated by colons>`	specifies the path from which to look up the `.java` files
`-vonly`	verify only mode - no changes to the will be made
`-nosources <package name>`	specifies a package for which there are no `.java` sources
`-verbose`	output messages about what the tool is doing
`-lpclasses`	print a list of all user persistent classes in the given store
`-g, -g:none, -g:{lines,vars,source}, -O, -nowarn, -deprecation, -encoding`	these are the options of the underlying `javac` Java compiler that make sense in the context of `opjb` and therefore accepted by it. See the online Java SDK Tools documentation [Sun00f] for their explanation
`-cverbose`	output messages about what the compiler is doing (equivalent to the `-verbose` option of the `javac`

Appendix B

Conversion Code Examples

In this appendix, we present one of the tests from our regression test suite. It consists of a number of classes, resembling those used in the GAP geographical information system. Classes have undergone two evolutions, resulting in three versions of code and two conversion classes.

B.1 Version 1 of the Code

```
package geogrdata;

// An interface to represent an object that has name (geographic identity)
public interface NamedGeogrObject {
  public String getName();

  public void setName(String name);
}


// ---------------------------------------------------------------------
package geogrdata;

// Geographical point object, that has location and geographic identity
public class GeogrPoint implements NamedGeogrObject {
  private int X, Y;
  private String Name;

  private GeomLine[] links;

  public GeogrPoint(int X, int Y) {
    this.X = X;
```

```
    this.Y = Y;
  }

  public String getName() {
    return Name;
  }

  public void setName(String Name) {
    this.Name = Name;
  }

  public GeomLine[] getLinks() {
    return links;
  }

  public void setLinks(GeomLine links[]) {
    this.links = links;
  }

  public int getX() { return X; }

  public int getY() { return Y; }

  void setX(int X) { this.X = X; }

  void setY(int Y) { this.Y = Y; }
}


// ---------------------------------------------------------------------
package geogrdata;

// Abstract geometric polyline - an object without its own geographic identity
public class GeomLine {
  private GeogrPoint allPoints[]; // (Non-economic) representation of coordinates

  public GeomLine(GeogrPoint allPoints[]) {
    this.allPoints = allPoints;
  }

  public GeogrPoint getStartPoint() {
    return allPoints[0];
  }

  public GeogrPoint getEndPoint() {
    return allPoints[allPoints.length-1];
  }
```

```
  public GeogrPoint[] getPoints() {
    return allPoints;
  }
}


// ---------------------------------------------------------------------
package geogrdata;

// Geographic polyline - an object with coordinates and geographic identity
public class GeogrLine extends GeomLine implements NamedGeogrObject {
  private String name;

  public GeogrLine(GeogrPoint allPoints[], String name) {
    super(allPoints);
    this.name = name;
  }

  public String getName() {
    return name;
  }

  public void setName(String name) {
    this.name = name;
  }
}


// ---------------------------------------------------------------------
package geogrdata;

// Abstract geometric area - an object without geographic identity
public class GeomArea {
  private GeomLine allLines[];

  public GeomArea(GeomLine allLines[]) {
    this.allLines = allLines;
  }

  public GeomLine[] getLines() {
    return allLines;
  }
}


// ---------------------------------------------------------------------
```

```
package geogrdata;

// Geographic area - an object with coordinates and geographic identity
public class GeogrArea extends GeomArea implements NamedGeogrObject {
  private String name;

  public GeogrArea(GeomLine allLines[], String name) {
    super(allLines);
    this.name = name;
  }

  public String getName() {
    return name;
  }

  public void setName(String name) {
    this.name = name;
  }
}


// ---------------------------------------------------------------------
package geogrdata;

import java.util.Hashtable;

// Geographic area with some additional attributes (an attribute can be any object)
public class AttrGeogrArea extends GeogrArea {
  private Hashtable attributes;

  public AttrGeogrArea(GeomLine allLines[], String name) {
    super(allLines, name);
    attributes = new Hashtable();
  }

  public void setAttribute(Object attr, Object value) {
    attributes.put(attr, value);
  }

  public Object getAttribute(Object attr) {
    return attributes.get(attr);
  }
}


// ---------------------------------------------------------------------
package geogrdata;
```

```
import org.opj.OPRuntime;

// Class representing complete geographic database
public class GeogrBase {
  static public GeogrLine[] geogrLines;
  static public GeogrPoint[] geogrPoints;
  static public GeogrArea[] geogrAreas;

  static {
    OPRuntime.roots.add(GeogrBase.class);
  }
}
```

B.2 Version 2 of the Code

Only changed and added classes are presented. The main difference between Version 1 and Version 2 is that in the latter, representation of coordinates of geographic polylines is changed. Also, class `GeomPoint` is added to represent an abstract point object, eliminating the initial inconsistency in the design.

```
package geogrdata;

// Abstract geometric point - an object with location but no geographic identity
// This is a new class.
public class GeomPoint {
  protected int x, y;

  public GeomPoint(int x, int y) {
    this.x = x;
    this.y = y;
  }

  public int getX() { return x; }

  public int getY() { return y; }

  void setX(int X) { this.x = x; }

  void setY(int Y) { this.y = y; }
}
```

```
// ---------------------------------------------------------------------
package geogrdata;

// Geographic point object - now a descendant of GeomPoint
public class GeogrPoint extends GeomPoint implements NamedGeogrObject {
  private String name;  // This field was called "Name"

  private GeomLine links[];

  public GeogrPoint(int x, int y, String name) {
    super(x, y);
    this.name = name;
  }

  public String getName() {
    return name;
  }

  public void setName(String name) {
    this.name = name;
  }

  public GeomLine[] getLinks() {
    return links;
  }

  public void setLinks(GeomLine links[]) {
    this.links = links;
  }
}


// ---------------------------------------------------------------------
package geogrdata;

// Abstract geometric polyline - an object without its own geographical identity
public class GeomLine {
  // New, more economic representation of coordinates - was GeogrPoint allPoints[]
  public GeomPoint startPoint;
  public GeomPoint endPoint;
  private int allPointsX[];
  private int allPointsY[];

  // Constructor and methods interface have been change to optimally work with
  // new representation of coordinates.
  public GeomLine(int allPointsX[], int allPointsY[],
                  GeomPoint startPoint, GeomPoint endPoint) {
```

```
    this.allPointsX = allPointsX;
    this.allPointsY = allPointsY;
    if (startPoint == null)
      this.startPoint = new GeomPoint(allPointsX[0], allPointsY[0]);
    else
      this.startPoint = startPoint;
    if (endPoint == null) {
      int endInd = allPointsX.length - 1;
      this.endPoint = new GeomPoint(allPointsX[endInd], allPointsY[endInd]);
    } else
      this.endPoint = endPoint;
  }

  public GeomPoint getStartPoint() {
    return startPoint;
  }

  public GeomPoint getEndPoint() {
    return endPoint;
  }

  public int[] getXCoords() {
    return allPointsX;
  }

  public int[] getYCoords() {
    return allPointsY;
  }
}


// ---------------------------------------------------------------------
package geogrdata;

// Geographic polyline - an object with coordinates and geographical identity.
public class GeogrLine extends GeomLine implements NamedGeogrObject {
  private String name;

  // Constructor was fixed according to the change in its superclass
  public GeogrLine(int allPointsX[], int allPointsY[], String name,
                   GeomPoint startPoint, GeomPoint endPoint) {
    super(allPointsX, allPointsY, startPoint, endPoint);
    this.name = name;
  }

  public String getName() {
    return name;
```

```
  }

  public void setName(String name) {
    this.name = name;
  }
}
```

B.3 Conversion Class 1

Here is the conversion class used to convert the instances of the classes changed from Version 1 to Version 2.

```
import geogrdata.GeogrPoint$$_old_ver_;
import geogrdata.GeomLine$$_old_ver_;
import geogrdata.GeogrPoint;
import geogrdata.GeomLine;
import geogrdata.GeomPoint;

import org.opj.utilities.PJEvolution;
import java.lang.reflect.Field;

public class ConvertGeogrBase {
  static Field allPointsX, allPointsY;

  public static void onConversionStart() {
    // This method is executed once, before conversion methods for
    // instances. We use it for obtaining field objects. We can use
    // reflection freely to access private data in conversion code.
    Class classGeomLine = GeomLine.class;
    try {  // Get reflection objects for private arrays of coordinates
      allPointsX = classGeomLine.getDeclaredField("allPointsX");
      allPointsY = classGeomLine.getDeclaredField("allPointsY");
    } catch (Exception e) {
      System.out.println(e);
      System.exit(-1);
    }
  }

  public static GeomPoint convertInstance(GeogrPoint$$_old_ver_ oldp) {
    if (oldp.getName() != null || oldp.getLinks() != null) {
      GeogrPoint newp = new GeogrPoint(oldp.getX(), oldp.getY(), oldp.getName());
      PJEvolution.copyDefaults(oldp, newp);
      return newp;
    } else {
```

```
      // GeogrPoints which in reality were just geometric points, should
      // be converted to instances of GeomPoint
      GeomPoint newp = new GeomPoint(oldp.getX(), oldp.getY());
      return newp;
    }
  }

  // This conversion method will be used for instances of both class GeomLine
  // and GeogrLine, which is GeomLine's descendant
  public static void convertInstance(GeomLine$$_old_ver_ oldl, GeomLine newl) {
    GeogrPoint$$_old_ver_ points[] = oldl.getPoints();
    int len = points.length;
    int x[] = new int[len];
    int y[] = new int[len];
    for (int i = 0; i < len; i++) {
      x[i] = points[i].getX();
      y[i] = points[i].getY();
    }

    try {
      allPointsX.set(newl, x);
      allPointsY.set(newl, y);
    } catch (Exception e) {  // Reflection can produce exceptions
      System.out.println(e);
      System.exit(-1);
    }
    newl.startPoint = points[0];
    newl.endPoint = points[points.length-1];
  }
}
```

B.4 Version 3 of the Code

Here we have found that the class `AttrGeogrArea` is inadequate. The only attribute that a geographic area can have is `SOLID`, equal to either `TRUE` or `FALSE`. Therefore to have a hash table of attributes in each geographic area object is wasteful. We delete the `AttrGeogrArea` class, at the same time adding the `solid` field to class `GeogrArea`. All instances of `AttrGeogrArea` are migrated to class `GeogrArea`, with the `solid` field set to an appropriate value.

```
package geogrdata;

// Geographic area - an object with coordinates and geographic identity
public class GeogrArea extends GeomArea implements NamedGeogrObject {
```

```
  private String name;
  private boolean solid;  // The only possible attribute of geographic area

  public GeogrArea(GeomLine allLines[], String name, boolean isSolid) {
    super(allLines);
    this.name = name;
    this.solid = isSolid;
  }

  public String getName() {
    return name;
  }

  public void setName(String name) {
    this.name = name;
  }

  public boolean isSolid() {
    return solid;
  }
}
```

B.5 Conversion Class 2

Here is a conversion class with a method to migrate instances of deleted class `AttrGeogrArea` to class `GeogrArea`.

```
import geogrdata.GeogrArea$$_old_ver_;
import geogrdata.AttrGeogrArea$$_old_ver_;
import geogrdata.GeogrArea;

import org.opj.utilities.PJEvolution;
import java.lang.reflect.Field;

public class ConvertGeogrAreas {
  static Field solidField;

  public static void onConversionStart() {
    // Since the "solid" field is private, we have to use reflection to set it
    Class classGeogrArea = GeogrArea.class;
    try {
      solidField = classGeogrArea.getDeclaredField("solid");
    } catch (Exception e) {
```

```
      System.out.println(e);
      System.exit(-1);
    }
  }

  public static void migrateInstance(AttrGeogrArea$$_old_ver_ oldA, GeogrArea newA) {
    boolean solid = ((String) oldA.getAttribute("SOLID")).equals("TRUE");
    try {
      solidField.setBoolean(newA, solid);
    } catch (Exception e) {
      System.out.println(e);
      System.exit(-1);
    }
  }
}
```

Appendix C

Specification of JVM Calls Supporting Runtime Class Evolution

In this appendix, we present the specification of the two calls that presently implement runtime class evolution functionality, and which are a part of the Java Virtual Machine Debug Interface [Sun00l].

C.1 RedefineClasses()

```
typedef struct {
    jclass clazz;                /* Class object for this class */
    jint class_byte_count;       /* number of bytes defining class (below) */
    jbyte *class_bytes;          /* bytes defining class */
                                 /* (in Class File Format of JVM spec) */
} JVMDI_class_definition;

jvmdiError
RedefineClasses(jint classCount, JVMDI_class_definition *classDefs);
```

All classes given are redefined according to the definitions supplied. If any redefined methods have active stack frames, those active frames continue to run the bytecodes of the original method. The redefined methods will be used on new invokes. Any JVMDI function or event which returns a `jmethodID`, will return `OBSOLETE_METHOD_ID` when referring to the original method (for example, when examining a stack frame where the original method is still executing) unless it is equivalent to the redefined method (see below). The original method ID refers to the redefined method. Care should be taken thoughout a JVMDI client to handle `OBSOLETE_METHOD_ID`. If reseting of stack frames is desired, use PopFrame to pop frames with `OBSOLETE_METHOD_ID`s.

An original and a redefined method should be considered equivalent if their bytecodes are the same except for indicies into the constant pool and the referenced constants are equal.

This function does not cause any initialization except that which would occur under the customary JVM semantics. In other words, redefining a class does not cause its initializers to be run. The values of preexisting static variables will remain as they were prior to the call. However, completely uninitialized (new) static variables will be assigned their default value.

If a redefined class has instances then all those instances will have the fields defined by the redefined class at the completion of the call. Preexisting fields will retain their previous values. Any new fields will have their default values; no instance initializers or constructors are run.

Threads need not be suspended.

All breakpoints in the class are cleared.

All attributes are updated.

No JVMDI events are generated by this function.

This is an optional feature which may not be implemented for all virtual machines. Examine `can_redefine_classes`, `can_add_method` and `can_unrestrictedly_redefine_classes` of `GetCapabilities` to determine whether this feature is supported in a particular virtual machine.

Parameters:

`classCount` : the number of classes specified in classDefs.
`classDefs` : the array of new class definitions.

This function returns either a universal error or one of the following errors:

`JVMDI_ERROR_NULL_POINTER`
Invalid pointer: `classDefs` or one of `class_bytes` is NULL.

`JVMDI_ERROR_INVALID_CLASS`
An element of `classDefs` is not a valid class.

`JVMDI_ERROR_UNSUPPORTED_VERSION`
A new class file has a version number not supported by this VM.

`JVMDI_ERROR_INVALID_CLASS_FORMAT`
A new class file is malformed (The VM would return a `ClassFormatError`).

`JVMDI_ERROR_CIRCULAR_CLASS_DEFINITION`
The new class file definitions would lead to a circular definition (the VM would return a `ClassCircularityError`).

`JVMDI_ERROR_FAILS_VERIFICATION`
The class bytes fail verification.

JVMDI_ERROR_NAMES_DONT_MATCH
The class name defined in the new class file is different from the name in the old class object.

JVMDI_ERROR_NOT_IMPLEMENTED
No aspect of this functionality is implemented (can_redefine_classes capability is false).

JVMDI_ERROR_ADD_METHOD_NOT_IMPLEMENTED
A new class file would require adding a method, (and can_add_method capability is false).

JVMDI_ERROR_SCHEMA_CHANGE_NOT_IMPLEMENTED
The new class version changes fields (and can_unrestrictedly_redefine_classes capability is false).

JVMDI_ERROR_HIERARCHY_CHANGE_NOT_IMPLEMENTED
A direct superclass is different for the new class version, or the set of directly implemented interfaces is different (and can_unrestrictedly_redefine_classes capability is false).

JVMDI_ERROR_DELETE_METHOD_NOT_IMPLEMENTED
The new class version does not declare a method declared in the old class version (and can_unrestrictedly_redefine_classes capability is false).

JVMDI_ERROR_CLASS_MODIFIERS_CHANGE_NOT_IMPLEMENTED
The new class version has different modifiers (and can_make_binary_compatible_changes capability is false).

JVMDI_ERROR_METHOD_MODIFIERS_CHANGE_NOT_IMPLEMENTED
A method in the new class version has different modifiers than its counterpart in the old class version (and can_make_binary_compatible_changes capability is false).

C.2 PopFrame()

```
jvmdiError
PopFrame(jthread thread);
```

Pop the topmost stack frame of thread's stack. Popping a frame takes you to the preceding non-native frame (popping any intermediate native frames). When the thread is resumed, the thread state is reset to the state immediately before the called method was invoked: the operand stack is restored (objectref if appropriate and arguments are added back), note however, that any changes to the arguments, which occurred in the called method, remain; when execution continues, the first instruction to execute will be the invoke. Note that if there are intervening native frames, the called method will be different than the method of the popped frame.

Between calling PopFrame and resuming the thread the state of the stack is undefined. To pop frames beyond the first, these three steps must be repeated:

1. suspend the thread via an event (step, breakpoint, ...)
2. call `PopFrame`
3. resume the thread

Locks acquired by a popped frame are released when it is popped. This applies to synchronized methods that are popped, and to any synchronized blocks within them, but does not apply to native locks.

`finally` blocks are not executed.

Changes to global state are not addressed.

If this function is called for a thread different than the current thread, the specified thread must be suspended. The thread must be in a Java programming language or JNI method.

All frame IDs for this thread are invalidated.

No JVMDI events are generated by this function.

This is an optional feature which may not be implemented for all virtual machines. Examine `can_pop_frame` of `GetCapabilities` to determine whether this feature is supported in a particular virtual machine.

Parameters:

`thread` : the thread whose top frame is to be popped.

This function returns either a universal error or one of the following errors:

`JVMDI_ERROR_INVALID_THREAD`
Thread was invalid.

`JVMDI_ERROR_NULL_POINTER`
Invalid pointer.

`JVMDI_ERROR_THREAD_NOT_SUSPENDED`
Thread was not suspended or current thread.

`JVMDI_ERROR_NO_MORE_FRAMES`
There are no more Java programming language frames on the call stack.

`JVMDI_ERROR_NOT_IMPLEMENTED`
This functionality is not implemented (can_pop_frame capability is false).

Bibliography

[AAB+99] B. Alpern, C.R. Attanasio, J.J. Barton, A. Cocchi, S.F. Hummel, D. Lieber, T. Ngo, M. Mergen, J.C. Shepherd, and S. Smith. Implementing Jalapeño in Java. In *Object Oriented Programing: Systems, Languages, and Applications (OOPSLA)*, pages 314–324, October 1999. Available at http://www.research.ibm.com/jalapeno/publication.html.

[ABC+83] M.P. Atkinson, P.J. Bailey, K.J. Chisholm, W.P. Cockshott, and R. Morrison. An Approach to Persistent Programming. *The Computer Journal*, 26(4):360–365, November 1983.

[ABDS94] E. Amiel, M.-J. Bellosta, E. Dujardin, and E. Simon. Supporting Exceptions to Behavioural Schema Consistency to Ease Schema Evolution in OODBMS. In *Proceedings of the 20th International Conference on Very Large Data Bases, Santiago, Chilie*, pages 108–119. Morgan Kaufmann, 1994.

[ACC82] M.P. Atkinson, K.J. Chisholm, and W.P. Cockshott. PS-Algol: an Algol with a Persistent Heap. *ACM SIGPLAN Notices*, 17(7):24–31, July 1982.

[ACL+99] B. Alpern, A. Cocchi, D. Lieber, M. Mergen, and V. Sarkar. Jalapeño — a Compiler-Supported Java Virtual Machine for Servers. In *Proceedings of the Workshop on Compiler Support for Software System (WCSSS 99), Atlanta, GA, held in conjunction with PLDI 99*, May 1999. Available at http://www.research.ibm.com/jalapeno/publication.html.

[AD97] O. Agesen and D. Detlefs. Finding References in Java Stacks. In *Proceedings of the OOPSLA'97 Workshop on Garbage Collection and Memory Management, Atlanta, USA*, October 1997.

[ADHP00] M.P. Atkinson, M. Dmitriev, C. Hamilton, and T. Printezis. Scalable and Recoverable Implementation of Object Evolution for the PJama Platform, 2000. Proceedings of the 9th International Workshop on Persistent Object Systems (POS9).

[ADM98] O. Agesen, D. Detlefs, and J.E.B. Moss. Garbage Collection and Local Variable Type-Precision and Liveness in Java Virtual Machines. In *Proceedings of PLDI'98, Montreal, Canada*, pages 269–279, June 1998.

[AJ99] M.P. Atkinson and M.J. Jordan. Issues Raised by Three Years of Developing PJama. In C. Beeri and O.P. Buneman, editors, *Database Theory — ICDT'99*, number 1540 in Lecture Notes in Computer Science, pages 1–30. Springer-Verlag, 1999.

[AJ00] M.P. Atkinson and M.J. Jordan. A Review of the Rationale and Architectures of PJama: a Durable, Flexible, Evolvable and Scalable Orthogonally Persistent Programming Platform. Technical report, Sun Microsystems Laboratories Inc and Department of Computing Science, University of Glasgow, 901, San Antonio Road, Palo Alto, CA 94303, USA and Glasgow G12 8QQ,

Scotland, 2000. TR-2000-90,
http://www.sun.com/research/forest/COM.Sun.Labs.Forest.doc.pjama_review.abs.html.

[AJEFL95] L. Al-Jadir, T. Estier, G. Falquet, and M. Leonard. Evolution Features of the F2 OODBMS. In *Proceedings of the 4th International Conference on Database Systems for Advanced Applications - DASFAA 1995*, pages 284–291, 1995.

[AJL98] L. Al-Jadir and M. Leonard. Multiobjects to Ease Schema Evolution in an OODBMS. In *Proceedings of the 17th International Conference on Conceptual Modelling - ER 1998*, volume 1507 of *Lecture Notes in Computer Science*, pages 316–333. Springer-Verlag, 1998.

[Ala97] S. Alagic. The ODMG Object Model: Does it Make Sense? In *Proceedings of OOPSLA 1997*, volume 32 of *ACM SIGPLAN Notices*, pages 253–270, October 1997.

[Ala99] S. Alagic. Type Checking OQL Queries in the ODMG Type System. *ACM Transactions on Database Systems*, 24(3):212–233, September 1999.

[AM95] M.P. Atkinson and R. Morrison. Orthogonal Persistent Object Systems. *VLDB Journal*, 4(3):309–401, 1995.

[Atk78] M.P. Atkinson. Programming Languages and Databases. *VLDB Journal*, pages 408–419, 1978.

[Atk00] M.P. Atkinson. Persistence and Java — a Balancing Act. In *Objects and Databases. International Symposium, Sophia Antipolis, France, June 2000. Revised Papers*, volume 1944 of *LNCS*. Springer-Verlag, 2000.

[ATW94] R. Adams, W. Tichy, and A. Weinert. The Cost of Selective Recompilation and Environment Processing. *ACM Transactions on Software Engineering and Methodology*, 3(1):3–28, January 1994.

[Ban87] J. Banerjee. Data Model Issues for Object-Oriented Applications. *ACM Transactions on Office Information Systems*, 5(1):3–26, January 1987.

[BG93] G. Bracha and D. Griswold. Strongtalk: Typechecking Smalltalk in a Production Environment. In *Proceedings of OOPSLA'93*, ACM SIGPLAN Notices, pages 215 – 230, 1993.

[BKKK87] J. Banerjee, W. Kim, H. Kim, and H.F. Korth. Semantics and Implementation of Schema Evolution in Object-Oriented Databases. In *Proceedings of ACM SIGMOD Conference*, volume 16, pages 311–322, December 1987.

[Bor00a] Borland Inprise Inc. C++ Builder 5. http://www.inprise.com/bcppbuilder/, 2000.

[Bor00b] Borland Inprise Inc. Delphi 5. http://www.inprise.com/delphi/, 2000.

[Bor00c] Borland Inprise Inc. JBuilder. http://www.inprise.com/jbuilder/, 2000.

[Bra00] G. Bracha. Personal Communication, November 2000.

[Bru91] T. Brunoff. *Makedepend Manual Page*, April 1991. Tektronix, Inc. and MIT Project Athena, University of New Mexico.

[BW88] H. Boehm and M. Weiser. Garbage Collection in an Uncooperative Environment. *Software – Practice and Experience*, pages 807 – 820, September 1988.

[Byo98] J. Byous. Java[tm] Technology: an Early History. http://java.sun.com/features/1998/05/birthday.html, 1998.

[Cat97] R.G.G. Cattell, editor. *The Object Database Standard: ODMG 2.0.* Morgan Kaufmann, 1997.

[CCK98] G.A. Cohen, J.S. Chase, and D.L. Kaminsky. Automatic Program Transformation with JOIE. In *Proceedings of the USENIX Annual Technical Symposium*, 1998.

[Chi00] S. Chiba. Load-Time Structural Reflection in Java. In Elisa Bertino, editor, *ECOOP 2000, European Conference on Object-Oriented Programming, Cannes, France*, volume 1850 of *LNCS*, pages 313 – 336. Springer-Verlag, 2000.

[DA97] L. Daynès and M.P. Atkinson. Main-Memory Management to support Orthogonal Persistence for Java. In *Proceedings of the Second International Workshop on Persistence and Java (PJW2)*, pages 37–60, Half Moon Bay, CA, USA, August 1997.

[DdBF+94] A. Dearle, R. di Bona, J. Farrow, F. Henskens, A. Lindstrom, J. Rosenberg, and F. Vaughan. Grasshopper: An Orthogonally Persistent Operating System. *Computing Systems*, 7(3):289–312, 1994.

[DH00] M. Dmitriev and C.G. Hamilton. Towards Scalable and Recoverable Object Evolution for the PJama Persistent Platform. In *Objects and Databases. International Symposium, Sophia Antipolis, France, June 2000. Revised Papers*, volume 1944 of *LNCS*. Springer-Verlag, 2000. Also available at http://www.disi.unige.it/conferences/oodbws00/.

[DS84] L. Deutsch and A. Schiffman. Efficient Implementation of the Smalltalk-80 System. In *Proceedings of 11th Symposium on the Principles of Programming Languages*, Salt Lake City, USA, 1984.

[DWE98] S. Drossopoulou, D. Wragg, and S. Eisenbach. What *is* Java Binary Compatibility? In C. Chambers, editor, *Object Oriented Programing: Systems, Languages, and Applications (OOPSLA)*, volume 33(10) of *ACM SIGPLAN Notices*, Vancouver, BC, October 1998.

[Eur97a] European Organisation for Nuclear Research (CERN). Object Database Features and HEP Data Management, CERN/LHCC 97/8. http://wwwinfo.cern.ch/asd/rd45/reports.htm, June 1997.

[Eur97b] European Organization for Nuclear Research (CERN). Using an Object Database and Mass Storage System for Physics Analysis, CERN/LHCC 97/9, April 1997. http://wwwinfo.cern.ch/asd/rd45/reports.htm.

[Exc99] Excelon Corp. ObjectStore. http://www.odi.com/products/objectstore.html, 1999.

[Fel79] S.I. Feldman. Make - A Program for Maintaining Computer Programs. *Software – Practice and Experience*, 9(4):255–256, April 1979.

[FFM+95] F. Ferrandina, G. Ferran, T. Meyer, J. Madec, and R. Zicari. Schema and Database Evolution in the O_2 Object Database System. In *Proceedings of the 21st Conference on Very Large Databases (VLDB)*, Zurich, Switzerland, 1995.

[FL96] F. Ferrandina and S.-E. Lautemann. An Integrated Approach to Schema Evolution for Object Databases. In *Proceedings of the 3rd International Conference on Object-Oriented Information Systems (OOIS)*, London, UK, 1996.

[FMZ94] F. Ferrandina, T. Meyer, and R. Zicari. Schema Evolution in Object Databases: Measuring the Performance of Immediate and Deferred Updates. In *Proceedings of the 20th International Conference on Very Large Data Bases, Santiago, Chilie*. Morgan Kaufmann, 1994.

[Fuj00] Fujitsu Ltd. *3.5-inch Magnetic Disk Drives MAB3045/MAB3091*, 2000. http://www.fujitsu.co.jp/hypertext/hdd/drive/overseas/mab30xx/mab30xx.html.

[GBCW00] W. Gu, N.A. Burns, M.T. Collins, and W.Y.P. Wong. The Evolution of a High-Performing Java Virtual Machine. *IBM Systems Journal*, 39(1), 2000. http://www.research.ibm.com/journal/sj/391/gu.html.

[Gem98] GemStone Systems Inc. *GemStone/J Programming Guide*, March 1998. Version 1.1.

[Gem00] GemStone Systems Inc. The GemStone/J iCommerce Platform. http://www.gemstone.com/products/j/main.html, May 2000.

[GJSB00] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification, Second Edition.* Addison-Wesley, June 2000.

[Gol84] A. Goldberg. *Smalltalk-80. The Interactive Programming Environment.* Addison-Wesley, 1984.

[Gos97] J. Gosling. A Brief History of the Green Project. http://java.sun.com/people/jag/green/index.html, 1997.

[GR83] A. Goldberg and D. Robson. *Smalltalk-80: The Language and its Implementation.* Addison-Wesley, 1983.

[GSAW98] S. Grimstad, D.I.J. Sjøberg, M.P. Atkinson, and R.C. Welland. Evaluating Usability aspects of PJama based on Source Code Measurements. pages 307–321. Morgan Kaufmann, August 1998.

[HAD99] C.G. Hamilton, M.P. Atkinson, and M. Dmitriev. Providing Evolution Support for PJama_1 within Sphere. Technical Report TR-1999-50, Department of Computing Science, University of Glasgow, Glasgow G12 8QQ, Scotland, December 1999.

[Ham97] C.G. Hamilton. Measuring the Performance of Disk Garbage Collectors: Garbage Collecting Persistent Java Stores. Master's thesis, Department of Computing Science, University of Glasgow, Glasgow G12 8QQ, Scotland, 1997.

[Ham99] C.G. Hamilton. Recovery Management for Sphere: Recovering a Persistent Object Store. Technical Report TR-1999-51, Department of Computing Science, University of Glasgow, Glasgow G12 8QQ, Scotland, December 1999.

[HG98] G. Hjalmtysson and R. Gray. Dynamic C++ Classes: a Lightweight Mechanism to Update Code in a Running Program. In *Proceedings of the USENIX Annual Technical Conference*, June 1998.

[Hol95] U. Holzle. *Adaptive Optimization for Self: Reconciling High Performance with Exploratory Programming.* PhD thesis, Stanford University, March 1995. Also published as Sun Microsystems Laboratories Technical Report SMLI TR-95-35.

[IBM00a] IBM Inc. The Jikes Open Source Project. http://oss.software.ibm.com/developerworks/opensource/ jikes/project/index.html, August 2000.

[IBM00b] IBM Inc. VisualAge for Java. http://www-4.ibm.com/software/ad/vajava/, 2000.

[IBM01] IBM Inc. Jalapeño Project Publications. http://www.research.ibm.com/jalapeno/publication.html, 2001.

[Ins00] Instantiations Inc. The JOVE 2.0 Optimizing Native Compiler for Java Technology. http://www.instantiations.com/jove/product/thejovesystem.htm, 2000.

[JA98] M.J. Jordan and M.P. Atkinson. Orthogonal Persistence for Java — A Mid-term Report. In M.P. Atkinson, M.J. Jordan, and R. Morrison, editors, *Proceedings of the 3rd International Workshop on Persistence and Java*, pages 335–352. Morgan Kaufmann, 1998.

[JA00] M.J. Jordan and M.P. Atkinson. Orthogonal Persistence for the Java Platform — Specification. Technical report, Sun Microsystems Laboratories Inc, 901, San Antonio Road, Palo Alto, CA 94303, USA, 2000. in preparation.

[Jap00] R.P. Japp. Adding Support for Cartographic Generalisation to a Persistent GIS. BSc Dissertation, University of Glasgow, Department of Computing Science, 2000.

[Jon96] R. E. Jones. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management.* John Wiley and Sons, Ltd., 1996. With a chapter on Distributed Garbage Collection by R.Lins.

[JV95] M.J. Jordan and M.L. Van De Vanter. Software Configuration Management in an Object Oriented Database. In *Proc. of the Usenix Conf. on Object-Oriented Technologies*, Monterey, CA, June 1995.

[JV97] M.J. Jordan and M.L. Van De Vanter. Modular System Building with Java Packages. In *Proc. of the 8th Int. Conference on Software Engineering Environments*, Cottbus, Germany, May 1997.

[KC88] W. Kim and H.-T. Chou. Versions of Schema for Object-Oriented Databases. In *Proceedings of 14th International Conference on Very Large Databases*, pages 148–159. Morgan Kaufmann, 1988.

[KH98] R. Keller and U. Holzle. Binary Component Adaptation. In *Proceedings of ECOOP'98*, volume 1445 of *LNCS*. Springer-Verlag, 1998. Also available at http://www.cs.ucsb.edu/oocsb/papers/TRCS97-20.html.

[LH90] B.S. Lerner and A.N. Habermann. Beyond Schema Evolution to Database Reorganisation. In *Proceedings of ECOOP/OOPSLA 1990*, volume 25 of *ACM SIGPLAN Notices*, pages 67–76, October 1990.

[Lie96] J. Liedtke. *L4 Reference Manual*, September 1996.

[LLOW91] C. Lamb, G. Landis, J. Orestein, and D. Weinreb. The ObjectStore database system. *Communications of the ACM*, 34(10):50–63, October 1991.

[LM99] B. Lewis and B. Mathiske. Efficient Barriers for Persistent Object Caching in a High-Performance Java Virtual Machine. In *Proc. of the OOPSLA'99 w'shop "Simplicity, Performance and Portability in Virtual Machine Design"*, 1999.

[LMG00] B. Lewis, B. Mathiske, and Neal Gafter. Architecture of the PEVM: A High-Performance Orthogonally Persistent Java Virtual Machine. 2000. To be published in the Proc. of the 9th Workshop on Persistent Object Systems (POS9).

[LY99] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification, Second Edition.* Addison-Wesley, 1999.

[MBCD89] R. Morrison, F. Brown, R. Connor, and A. Dearle. The Napier88 Reference Manual. Research Report PPRR-77-89, Universities of Glasgow and St Andrews, 1989.

[Met00] Metrowerks. CodeWarrior for Java, Professional Edition, Version 5.0. http://www.metroworks.com/desktop/java/, 2000.

[MHL+92] C. Mohan, D. Haderle, B. Lindsay, H. Pirashesh, and P. Schwarz. ARIES : A Transaction Recovery Method supporting Fine-granularity Locking and Partial Rollbacks using Write-Ahead Logging. *ACM Transactions on Database Systems*, 17(1):94–162, March 1992.

[Mic00a] Microsoft Corp. Microsoft Visual C++ Home Page. http://msdn.microsoft.com/visualc/default.asp, 2000.

[Mic00b] Microsoft Corp. Microsoft Visual J++ Home Page. http://msdn.microsoft.com/visualj/, 2000.

[Mon93] S. Monk. *A Model for Schema Evolution in Object-Oriented Database Systems.* PhD thesis, Lancaster University, Computing Department, 1993.

[MPG+00] S. Malabara, R. Pandey, J. Gragg, E. Barr, and J.F. Barnes. Runtime Support for Type-Safe Dynamic Java Classes. In Elisa Bertino, editor, *ECOOP 2000, European Conference on Object-Oriented Programming, Cannes, France*, volume 1850 of *LNCS*, pages 337–361. Springer-Verlag, 2000.

[MS92] S. Monk and I. Sommerville. A Model for Versioning of Classes in Object Oriented Databases. In *Proceedings of the 10th BNCOD*, pages 42–58, 1992.

[MS93] S. Monk and I. Sommerville. Schema Evolution in OODBs Using Class Versioning. *ACM SIGMOD Record*, 22(3):16–22, September 1993.

[MV99] T. Murer and M.L. Van De Vanter. Replacing Copies with Connections: Managing Software across the Virtual Organization. In *Proceedings of the 8th IEEE International Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprises*, Stanford University, CA, USA, June 1999.

[Net00] NetBeans Open Source Project. http://www.netbeans.org, 2000.

[Obj99a] Objectivity, Inc. *Objectivity Technical Overview, Version 5*, September 1999. http://www.objectivity.com/Products/TechOv.html.

[Obj99b] Objectivity, Inc. *Schema Evolution in Objectivity/DB*, September 1999. http://www.objectivity.com/WhitePapers.html.

[Obj99c] Objectivity, Inc. *Using for Java Guide. Release 5.2*, September 1999.

[Obj99d] Objectivity, Inc. *Using Objectivity/C++ Supplement. Release 5.2*, September 1999.

[Odb94a] E. Odberg. A Global Perspective of Schema Modification Management for Object-Oriented Databases. In *Proceedings of the 6th International Workshop on Persistent Object Systems (POS-6)*, pages 479–502, 1994.

[Odb94b] E. Odberg. Category classes: Flexible classification and evolution in object-oriented databases. *Lecture Notes in Computer Science*, 811:406–419, 1994.

[Odb95] E. Odberg. *Multiperspectives: Object Evolution and Schema Modification Management for Object-Oriented Databases.* PhD thesis, Department of Computer Systems and Telematics, Norwegian Institute of Technology, February 1995.

[OSM+00] H. Ogawa, K. Shimura, S. Matsouka, F. Maruyama, Y. Sohda, and Y. Kimura. OpenJIT: An Open-Ended, Reflective JIT Compile Framework for Java. In *Proceedings of ECOOP'2000*, volume 1850 of *LNCS*. Springer-Verlag, 2000. See also http://www.openjit.org/.

[Ous90] T. Ousterhout. Tcl: an Embeddable Command Language. In *Proceedings of the USENIX Association Winter Conference*, 1990.

[Ous93] T. Ousterhout. *An Introduction to Tcl and Tk.* Addison-Wesley, 1993.

[PAD98] T. Printezis, M.P. Atkinson, and L. Daynès. The Implementation of Sphere: a Scalable, Flexible, and Extensible Persistent Object Store. Technical Report TR-1998-46, Department of Computing Science, University of Glasgow, Glasgow G12 8QQ, Scotland, May 1998.

[PAJ99] T. Printezis, M. P. Atkinson, and M. J. Jordan. Defining and Handling Transient Data in PJama. In *Proceedings of the 7th International Workshop on Database Programming Languages (DBPL'99)*, Kinlochrannoch, Scotland, September 1999.

[Phi99] G. Phipps. Comparing Observed Bug and Productivity Rates for Java and C++. *Software — Practice and Experience*, 29(4):345–358, April 1999.

[PJa00] PJama Team. PJama — API, Tools and Tutorials, web-based documentation. http://www.sun.com/research/forest/COM.Sun.Labs.Forest.doc.external_www.PJava.main.html, March 2000.

[POE00] POET Software Corp. POET Object Server Suite Documentation. http://www.poet.com/products/doc/index.html, 2000.

[Pri96] T. Printezis. Analysing a Simple Disk Garbage Collector. In *Proceedings of the 1st International Workshop on Persistence and Java (PJW1)*, number TR-96-58 in SMLI Technical Report, Sun Microsystems, MS MTV29-01, 901 San Antonio Road, Palo Alto, CA 94303-4900, 1996.

[Pri99] T. Printezis. The Sphere User's Guide. Technical Report TR-1999-47, Department of Computing Science, University of Glasgow, Glasgow G12 8QQ, Scotland, July 1999.

[Pri00] T. Printezis. *Management of Long-Running, High-Performance Persistent Object Stores.* PhD thesis, Department of Computing Science, University of Glasgow, Glasgow G12 8QQ, Scotland, 2000.

[Ras00] M. Awais Rashid. *A Database Evolution Approach for Object-Oriented Databases.* PhD thesis, Computing Department, Lancaster University, UK, 2000.

[RHJ98] D. Ragget, A. Le Hors, and I. Jacobs, editors. *HTML 4.0 Specification.* April 1998. World Wide Web Consortium (W3C) Recommendation.

[RS99] A. Rashid and P. Sawyer. Evaluation for Evolution: How Well Commercial Systems Do. In *Proceedings of the 1st ECOOP Workshop on Object-Oriented Databases*, 1999.

[Rus00] C. Russell. JSR-12 Java Data Objects Specification (approved for development). http://java.sun.com/aboutJava/communityprocess/jsr/jsr_012_dataobj.html, apr 2000.

[SA97] S. Spence and M.P. Atkinson. A Scalable Model of Distribution Promoting Autonomy of and Cooperation Between PJava Object Stores. In *Proc. of the 13th Hawaii Int. Conf. on System Sciences*, Hawaii, USA, January 1997.

[Sjø93a] D.I.K. Sjøberg. Quantifying Schema Evolution. *Information and Software Technology*, 35(1):35–44, January 1993.

[Sjø93b] D.I.K. Sjøberg. *Thesaurus-Based Methodologies and Tools for Maintaining Persistent Application Systems.* PhD thesis, Department of Computing Science, University of Glasgow, Glasgow G12 8QQ, Scotland, 1993.

[SK88] R.W. Schwanke and G.E. Kaiser. Smarter Recompilation. *ACM Transactions on Programming Languages and Systems*, 10(4):627 – 632, October 1988.

[Sla98] S. Slade. *Object-Oriented Common Lisp.* Prentice Hall, 1998.

[SLU00] Sound Languages Underpin Reliable Programming Project (SLURP), Imperial College, University of London, London, UK. http://www-dse.doc.ic.ac.uk/Projects/slurp/index.html, 2000.

[SMK+94] M. Satyanarayanan, H.H. Mashburn, P. Kumar, D.C. Steere, and J.J. Kistler. Lightweight Recoverable Virtual Memory. *ACM Transactions on Computers and Systems*, 12(1):33–57, February 1994.

[SOT+00] T. Suganuma, T. Ogasawara, M. Takeuchi, T. Yasue, M. Kawahito, K. Ishizaki, H. Komatso, and T. Nakatani. Overview of the IBM Java Just-in-Time Compiler. *IBM Systems Journal*, 39(1):175–193, 2000. http://www.research.ibm.com/journal/sj/391/suganuma.html.

[SP89] R.W. Schwanke and M.A. Platoff. Cross References are Features. In *Software Engineering Notices*, pages 86 – 95, 1989. Proceedings of the Second International Workshop on Software Configuration Management, Princeton, New Jersey, November 1989.

[Spe99] S. Spence. PJRMI: Remote Method Invocation for Persistent Systems. In *Proc. of the Int. Symposium on Distributed Objects and Applications (DOA'99), Edinburgh, Scotland.* IEEE Press, 1999.

[Spe00] S. Spence. *Limited Copies and Leased References for Distributed Persistent Objects.* PhD thesis, Department of Computing Science, University of Glasgow, Glasgow G12 8QQ, Scotland, May 2000.

[Sun99a] Sun Microsystems Inc. Java 2D™ API Specification. http://java.sun.com/j2se/1.3/docs/guide/2d/spec.html, May 1999.

[Sun99b] Sun Microsystems Inc. The Java HotSpotTM Server VM. The Solution for Reliable, Secure Performance for the Enterprise. http://java.sun.com/products/hotspot/docs/general/hs2.html, 1999.

[Sun00a] Sun Microsystems Inc. Enterprise Java Beans™ Technology. http://java.sun.com/products/ejb/, November 2000.

[Sun00b] Sun Microsystems Inc. Forte For Java. http://www.sun.com/forte/ffj/, 2000.

[Sun00c] Sun Microsystems Inc. Java 2 Platform, Enterprise Edition – Compatibility. http://java.sun.com/j2ee/compatibility.html, 2000.

[Sun00d] Sun Microsystems Inc. Java 2 Platform, Standard Edition Version 1.3. http://java.sun.com/j2se/1.3/index.html, 2000.

[Sun00e] Sun Microsystems Inc. Java 2 Platform, Standard Edition Version 1.3 API Documentation. http://java.sun.com/j2se/1.3/docs/api/index.html, 2000.

[Sun00f] Sun Microsystems Inc. Java 2 Platform, Standard Edition Version 1.3 Tools Documentation. http://java.sun.com/j2se/1.3/docs/tooldocs/tools.html, 2000.

[Sun00g] Sun Microsystems Inc. Java 3D^{tm} API Specification, Version 1.2. http://java.sun.com/docs/books/java3d/, April 2000.

[Sun00h] Sun Microsystems Inc. Java Blend. http://www.sun.com/software/javablend/index.html, 2000.

[Sun00i] Sun Microsystems Inc. Java HotSpot Virtual Machine — Sun Community Source Licensing. http://www.sun.com/software/communitysource/hotspot/download.html, 2000.

[Sun00j] Sun Microsystems Inc. Java Object Serialization. http://java.sun.com/j2se/1.3/docs/guide/serialization/index.html, 2000.

[Sun00k] Sun Microsystems Inc. Java Platform Debugger Architecture. http://java.sun.com/products/jpda/doc/architecture.html, 2000.

[Sun00l] Sun Microsystems Inc. Java Virtual Machine Debug Interface Reference. http://java.sun.com/products/jpda/doc/jvmdi-spec.html, 2000.

[Sun00m] Sun Microsystems Inc. JavaServer Pages Technology. http://java.sun.com/products/jsp/, 2000.

[Sun00n] Sun Microsystems Inc. Java^{tm} Remote Method Invocation (RMI). http://java.sun.com/j2se/1.3/docs/guide/rmi/index.html, April 2000.

[Sun00o] Sun Microsystems Inc. Java^{tm} Virtual Machine Profiling Interface (JVMPI). http://java.sun.com/j2se/1.3/docs/guide/jvmpi/index.html, April 2000.

[Sun00p] Sun Microsystems Inc. JDBC^{tm} – Connecting Java and Databases. http://java.sun.com/products//jdk/1.1/docs/guide/jdbc/index.html, November 2000.

[Sun00q] Sun Microsystems Inc. Workgroup Servers, Sun Enterprise^{tm} 450. http://www.sun.com/servers/workgroup/450/, 2000.

[Sun00r] Sun Microsystems Laboratories. SELF Home Page. http://www.sun.com/research/self/index.html, 2000.

[Tic86] W. Tichy. Smart Recompilation. *ACM Transactions on Programming Languages and Systems*, 8(3):273–291, July 1986.

[Ung86] D. Ungar. *The Design and Evaluation of a High Performance Smalltalk System.* PhD thesis, University of California, Berkeley, February 1986. http://sunsite.berkeley.edu/Dienst/UI/2.0/Describe/ncstrl.ucb/CSD-86-287?abstract=smalltalk.

[US87] D. Ungar and R. Smith. SELF: The Power of Simplicity. In *OOPSLA'87 Conference Proceedings*, volume 22 of *ACM SIGPLAN Notices*, pages 227 – 241, Orlando, Florida, December 1987.

[Van98] M.L. Van De Vanter. Coordinated Editing of Versioned Packages in the JP Programming Environment. In *Proc. of the 8th Int. Symposium on System Configuration Management (SCM-8) Brussels*, LNCS. Springer-Verlag, 1998.

[Ver00] Versant Corp. Versant Developer Suite 6.0. http://www.versant.com/us/products/vds/index.html, 2000.

[VM99] M.L. Van De Vanter and T. Murer. Global Names: Support for Managing Software in a World of Virtual Organizations. In *Proc. of the 9th Int. Symposium on System Configuration Management (SCM-9) Toulouse, France*, LNCS. Springer-Verlag, 1999.

[Web00] WebGain. Visual Cafe 4.0. http://www.visualcafe.com/, 2000.

[WG98] D. White and A. Garthwaite. The GC Interface in the EVM. Technical Report TR-98-67, Sun Microsystems Laboratories Inc, 901, San Antonio Road, Palo Alto, CA 94303, USA, 1998.

[ZCF+97] C. Zaniolo, S. Ceri, C. Faloutsos, R.T. Snodgrass, V.S. Subrahmanian, and R.Zicari. *Advanced Database Systems*. Morgan Kaufmann, 1997.