A Dissertation Submitted to Department of Mathematical and Computing Sciences, Graduate School of Information Science and Engineering, Tokyo Institute of Technology
In Partial Fulfillment of the Requirements for the Degree of Doctor of Science in Mathematical and Computing Sciences

A Study of Dynamic Weaving for Aspect-Oriented Programming

Yoshiki Sato

Dissertation Chair:
Shigeru Chiba

December 2005,

Abstract

The previous implementations of dynamic weaving in AOP (Aspect-Oriented Programming) systems have drawbacks although it is receiving growing interests in both the academia and the industry. Typical dynamic weaving techniques are divided into two kinds of levels in terms of granularity of aspect weaving: a *class-level* and an *object-level* weaving. Either of weaving techniques also has each problem: serious performance penalties and the version barrier, respectively.

To provide an efficient, fine-grained, dynamic weaving mechanism at the class-level and object-level by addressing these problems, this thesis proposes new two mechanisms at each weaving level. One is a selective, just-in-time, aspect weaving mechanism for efficient class-level dynamic AOP, which seeks the performance improvement of weaving an aspect, executing an advice code, and the performance of normal execution. The other is the new concept for loosely-separated namespaces to enabling fine-grained object-level dynamic weaving, which can securely and efficiently relax the version barrier between namespaces.

Acknowledgments

I would like to express my deep gratitude to my supervisor, Shigeru Chiba (Tokyo Institute of Technology). I also profoundly thank Naoki Kobayashi (Tohoku University), Atushi Igarashi (Kyoto University), and Michiaki Tatsubori (IBM Tokyo Research Laboratory), who greatly helped me write this thesis. Hidehiko Masuhara (University of Tokyo) and Kenichi Kourai (Tokyo Institute of Technology) gave me valuable comments. Akihiko Tozawa (IBM Tokyo Research Laboratory) and Akira Koseki (IBM Tokyo Research Laboratory) gave me a plenty of advice when I belonged to IBM Tokyo Research Laboratory in 2003. The thesis committees reviewed the submitted version of this thesis and the final version reflects their comments. The committees are organized by Professor Shigeru Chiba, Professor Satoshi Matsuoka, Professor Etsuya Shibayama, Professor Masataka Sassa, Professor Ken Wakita, and Professor Takuo Watanabe.

I also deeply thank Takayasu Ito (now Ishinomaki Senshu University), who was my supervisor at Tohoku University from 1998 to 2000, Norio Shiratori (Tohoku University) and Tetsuo Kinoshita (Tohoku University), who were my supervisors at Tohoku University from 2000 to 2002, and Nobuyuki Ichiyoshi (Mitsubishi Research Institute) and Yasuyuki Shirai (Mitsubishi Research Institute), who gave support to me for writing this thesis at Mitsubishi Research Institute in 2005.

My colleagues in CSG (Chiba Shigeru Group) at Tokyo Institute of Technology, Daisuke Yokota (now Sony Corporation), Muga Nishizawa, Kiyoshi Nakagawa (now NEC Corporation), Masahiro Matsunuma (now NTT Comware Corporation), Yutaka Sunaga (now NTT Data Corporation), and Yoshisato Yanagisawa, colleagues at Tohoku University, Takuo

Suganuma, Shigeki Sugiura (now Tohoku Gakuin University), Tatsuhiko Sugiyama (now Unicus Corporation), Gen Kitagata, Masahiro Watanabe (now NTT Comware), Masataka Motegi (now Fujitsu Laboratory), Takahiro Uchiya, Kuniaki Kawabata (now IBM Japan), Satoshi Utsumi (now Advantest Corporation), Hidenari Ohwada (now NTT Corporation), Masahiro Shimada (now Fujitsu Corporation), Kenji Ejima (now Matsushita Electric Industrial Corporation), Hiroshi Hamada (now Fujitsu Corporation), Nobuyasu Saito (now Oki Electric Industry Corporation) must have given me an indirect but great influence on my way of thinking.

Finally, from the bottom of my heart, I want to thank my family, my sister, Terumi Sato, and friends, especially Mayuko Sasaki, for support and care during my years of study.

This thesis was financially supported by the CREST program of Japan Science and Technology Corp.

Yoshiki Sato
December 2005

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In computer science, the concept of *Separation of Concerns* [25] proposed by Dijkstra in the early 1970's has still been a long-cherished desire, or dream, for both developers and researchers. Underlying this concept is a general one that enables us to break the complexity of a problem into loosely-coupled, easier to solve, subproblems. These subproblems can be composed relatively easier to yield a solution to the original problem. In developing a software system, the separation of concerns enables us to focus our attention upon some *subprograms* from one aspect's point of view without caring much about other irrelevant aspects. Modularizing software systems could enhance their quality in terms of understandability, reusability, maintainability, and availability. Such modularized subprograms are called subroutines, procedures, functions, methods, modules, structures, classes and components, which depend on a programming dialect, the granularity of a subprogram, and the role of a subprogram against the whole program. The research history about programming languages can be seen as a perpetual quest for ideal modularization. There have been a number of programming paradigm, to say nothing of Object-oriented Programming (OOP), aimed at concrete separation of concerns, including Subject-oriented [39] Programming, Generative Programming [13], Adaptive Programming [68], Feature-oriented Programming [73], Intentional Programming [17], and Aspect-oriented Programming [49].

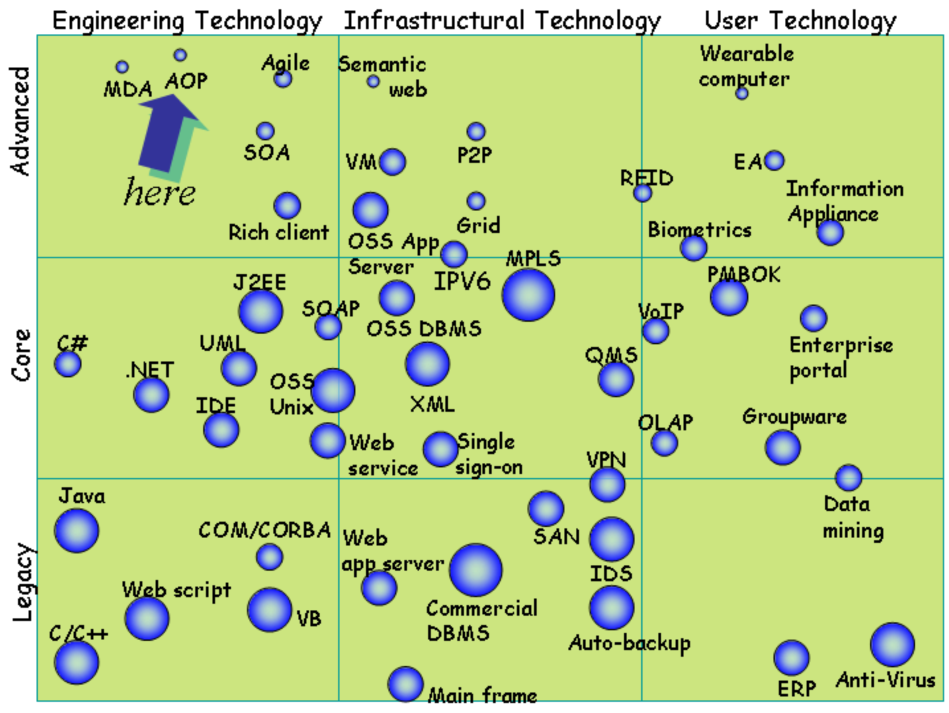


Figure 1.1. An information technology map in 2004 published by JISA (Japan Information Technology Services Industry Association). The circle size represents how much the system integrators have adopted or will adopt the corresponding technology.

Aspect-oriented Programming (AOP) [49] has been proposed especially for practicing the concept of *Advanced Separation of Concerns*, i.e. *Separation of Crosscutting Concerns*, for ever-complicating enterprise software systems used for infrastructual business operations and services. AOP allows us to modularize crosscutting concerns as *aspects* although widely spreading programming language and techniques such as OOP and the design patterns does not elegantly modularize them. Regardless of low attention in the computer industry from the information technology map illustrated in Figure 1.1, AOP is actually becoming an indispensable technology in developing software systems without informing most green developers. From the standpoint of this thesis, the essential property of AOP is inserting pieces of code into software systems externally although advanced mechanisms such as the reusability of aspects, the independent scope of crosscutting concerns, or reflection functionalities are preferably needed. In that standpoint, a great number of recent enterprise systems support an AOP functionality in various forms (e.g., pointcut/advice, XML/interceptor, or mouse click/scribbled code) to provide a disparate range of functionalities. These software systems contain various crosscutting concerns and expose them , such as logging, debugging and profiling information about particular intended parts and testing them, accessing databases, communicating with other systems over networks, performing data caching and prefetching for performance tune-up, and autonomically following user and environmental preferences. These functionalities normally scatter a bunch of tangled code concerning each function and crosscut the boundary between modules. Separating crosscutting concerns in AOP can give quality modules that are independent from the rest of modules and thus can be easily reused for other purposes. These reusable modules include not only a refined existing module but also a crosscutting concern. Currently, AOP is mostly realized as an extension of OOP, and implemented by using a program transformation technology that can compose an aspect into the program statically.

Motivating Problem

More recently, practical demands are being made of dynamic weaving in aspect-oriented programming systems [68, 69, 6, 72, 71]. Unlike ordinary AOP, an AOP system that supports the dynamic weaving mechanism allows dynamically *weave* (compose) and *unweave* (decompose) an aspect into/from a program. Therefore, *pointcuts* and *advice* are changeable during runtime. According to the AOP alliance [2], pointcuts and advice are essential and indispensable elements for AOP, and an aspect is composed of a set of combinations of these elements. The former is used for identifying the set of operations in the program, called *join points*, such as method calls, field accesses, instance creation, and exception handling, the latter is a piece of code executed at the identified join points.

Dynamic weaving techniques are typically classified at two kinds of levels: "class-level" weaving and an "object-level" weaving, each of which is used for a different purpose. A class-level dynamic weaving technique facilitates the behavior of whole application software changing globally and continuously when we attempt to update, develop, and maintain that software. This is because that system composes an aspect with classes involving their all present and future instances. On the other hand, an object-level dynamic weaving technique can easily change the behavior of a part of application software locally and temporarily at runtime. This system allows for aspects that can adapt the behavior of an instance of that application to follow the specific runtime environment and requirements.

The most typical technique for implementing class-level dynamic weaving is based on static code translation although it is not efficient. This approach statically inserts pieces of code, which we call hooks, into all join points, and these hooks determine at runtime whether or not there is associated advice to be activated at each join point, in contrast to the static weaving approach like AspectJ [48]. These runtime checks imply serious performance overhead although they are necessary since dynamic weaving allows turning advice on and off during runtime.

Another technique employing class loaders is a valuable for object-level dynamic weaving but a sweeping approach. The functionalities of object-level AOP are provided by most of today's Java-based application middleware such as J2EE servers, O/R mapping frameworks, and DI (Dependency Injection) containers [35]. A simple implementation of such an AOP system would use multiple class loaders, each of which loads a different version of a class woven with a different aspect. Creating a new class loader, and thereby loads a program woven with a new aspect, we might as well dynamically weave that aspect into the program as deploy (install) that program into the application middleware without restarting them. However, this implementation approach does not work because instances of those versions of a class type are not compatible because of the *version barrier*. The version barrier is a mechanism that prevents an instance of a version of a class type from being assigned to a variable of another version of that class type. In Java, if a class definition (i.e. class file) is loaded into different *namespaces* created by different class loaders, different versions of the class type are created and regarded as distinct types. If two class definitions with the same class name are loaded by different loaders, two versions of the class type are created and they can coexist while they are regarded as distinct types. Therefore, the running application including all related classes have to be discarded with the class loader that loads the application, and thus the service would once be suspended. The jobs that the old application did may also be discarded. This coarse-grained aspect weaving is practically inconvenient.

Solution by this thesis

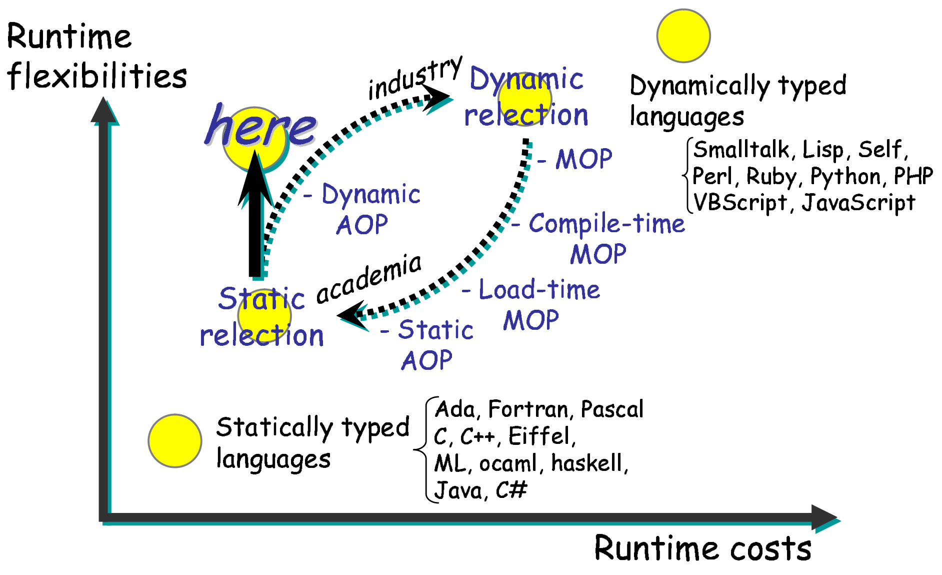
To provide an efficient, fine-grained, dynamic weaving mechanism at the class-level and the object-level by addressing each problem mentioned above, this thesis proposes new two mechanisms respectively. The former one is a "pure" Java and efficient class-level aspect weaving mechanism that seeks the improvement of weaving an aspect, executing an advice code, and the performance of normal execution. The latter one is a "dirty" Java but efficient object-level aspect weaving mechanism including an enhancement of the JVM (Java Virtual Machine). This mechanism lifts the fetters of the Java's portability, however, it allows us to enjoy dynamic weaving functionalities under the existing popular frameworks of class loaders and Java Reflection APIs.

The proposition for the class-level aspect weaving is a selective, just-in-time, aspect weaver for Java called *Wool* [75], which exploits our new implementation technique for addressing the performance problem of static code translation. Wool inserts hooks into loaded classes at runtime *just in time* when the programmer directs the program to start using an aspect. All existing instances follows the change of their class type. Wool allows the programmers to select from two implementation techniques the best one for each join point. The first one is to insert the hooks as breakpoints handled through the debugger interface of the JVM. The other one is to produce a program in which the hooks are embedded as method calls and reload that new program into the JVM.

For enabling fine-grained object-level aspect weaving, the next mechanism proposed in this thesis includes a novel concept and design of *loosely-separated* namespaces in Java called *sister* namespaces [74]. Sister namespaces can relax the version barrier between applications loaded by different class loaders. An instance can be carried beyond the version barrier between sister namespaces even if the class type of that instance is composed with a different aspect between these namespaces.

Position of this thesis

The position of this thesis in the research history of programming languages is that it provides the mechanisms for elevate the runtime flexibility of statically typed languages without forgetting the research results



Runtime flexibilities
Runtime costs
here
industry
Dynamic relection
Dynamically typed languages
Smalltalk, Lisp, Self, Perl, Ruby, Python, PHP VBScript, JavaScript
- MOP
- Dynamic AOP
- Compile-time MOP
academia
- Load-time MOP
Static relection
- Static AOP
Statically typed languages
Ada, Fortran, Pascal C, C++, Eiffel, ML, ocaml, haskell, Java, C#

Figure 1.2. Trade-off between runtime flexibilities and costs in programming languages. This thesis is positioned as the breakthrough for that trade-off.

on the implementation techniques of *reflection* and *MOP (Meta Object Protocol)*. The programming languages have trade-off between runtime flexibilities and costs as illustrated in Figure 1.2. Typically, statically typed languages have performance advantages compared to dynamically typed languages since the necessary runtime type checks are statically done in statically typed languages. Due to that property, statically typed languages have less runtime flexibility than dynamically typed languages. In the research domain of reflection and MOP, there were numerous researchers trying to add runtime power to the statically typed languages as much as dynamically typed languages have. However, most of real applications of those days did not require that much runtime power, and to make matters worse, it causes non-negligible runtime overheads. The static implementation of AOP was born in academia against the background of that history. Despite of that historical background, static implementation of AOP becomes a target of criticism especially from industry. Some developers claim that static AOP is too strict and not enough for developing modern application software as composed of distributed and dynamically plugged software components such as J2EE. Then, most dynamic implementations of AOP appeared in industry are going back to the old days in disregard of history. This thesis takes into account the lessons learned from the past and present cases and then realizes dynamic AOP with deep attention on the runtime costs.

The structure of this thesis

From the next chapter, this thesis presents pragmatic and technological background, and details of the proposed mechanisms. The structure of the rest of this thesis is as follows:

Chapter 2: Dynamic Weaving

This chapter discusses real needs and benefits of a dynamic weaving mechanism in AOP environment. We illustrates that dynamic weaving features are significantly helpful in both development and operation phases.

Also, this chapter overviews existing techniques that can enable dynamic weaving mechanisms. We introduce several programming techniques and languages that can dynamically extend programs. Then, we introduce some pure Java approaches such as static code translation. Finally, we introduce dirty Java approaches that extends the existing JVM for enabling dynamic weaving features.

Chapter 3: A Selective, Just-in-Time, Weaver

To address the performance problem discussed in the previous chapter, we first propose a selective, just-in-time, aspect weaver called Wool. We explain the design, implementation issues, and performance measurements of Wool.

Chapter 4: Loosely-separated Namespace

Then, we propose a novel concept and design of namespaces called sister namespaces for addressing the problem of the version barrier, which prevents us from developing a fine-grained dynamic weaving mechanism in Java. We also show performance measurements and a model of the sister namespace.

Chapter 5: A Dynamic Aspect Injection Container

We introduce a dynamic aspect injection container called Wooler, which enables object-level dynamic weaving mechanism. This container is based on the idea of the DI (Dependency Injection) container. A major breakthrough in injecting aspects into the running program can be realized by employing the mechanism of the sister namespace.

Chapter 6: Conclusion

Finally, we conclude this thesis in Chapter 6. We present contributions and limitations of this thesis and future directions.

Chapter 2

Dynamic Weaving in AOP

Dynamic weaving is not a special concept but a form of the practice of AOP. A dynamic weaving mechanism just allows an aspect being bound to the program at runtime. That mechanism is similar to the dynamic method binding mechanism of OOP, which is also called polymorphism. Dynamic binding in the object-oriented world has two forms; static and dynamic. Statically typed dynamic binding is found in languages such as Java (abstract and interface methods), C++ (virtual functions), and Eiffel (redefinition). An actual method is dynamically selected among the methods that override the virtual methods in each derived classes depending on the dynamic type of the object. The runtime selection of methods among all classes is another case of dynamic binding, found in dynamically typed languages such as CLOS, Self, and Smalltalk. Any operations on an object are dynamically bounded to that object. Dynamic aspect weaving (binding) is considered to bind the crosscutting operations to objects at the same time dynamically.

Due to the historical reasons, dynamic weaving mechanism tends to be seemed a special concept now. In fact, it is not surprising because most of current AOP systems for statically typed languages such as C++ and Java compose an aspect with the program statically; they are based on *static weaving*. The fundamental design for implementing AOP have been inspired by the results of a study on reflection, or MOP (Meta Object Protocol) [50], for statically typed languages. These researches contributed to enhance statically typed languages as flexible as the dynamically typed language. However, then scientists noticed the fact that most applications at that time did not require such runtime flexibilities, and then they inclined toward the improvement of a runtime efficiency in return for a flexibility [78, 92, 63, 61, 59]. Consequently, the developers of AOP systems for statically typed languages select a static design aimed to improve runtime performance rather than the flexibility.

The age changes, and a more dynamic and flexible mechanism has come to be requested from various applications. In the rest of this chapter, we first show motivating examples that require a more flexible, *dynamic*, AOP system. Then, we describe existing techniques for enabling dynamic weaving and problems of them.

2.1 Practical Benefits

Dynamic weaving is not just a mechanism that sounds fascinating but useless in practice. It is a necessary mechanism especially, if an aspect implements a non-functional concern cutting across several modules and the requirement of the functionality dynamically changes at runtime. Non-functional concerns are additional features such as transactions, distribution, security, and logging. On the other hand, the role of functional concerns is to implement a solution independently of the specific characteristics of the execution environment. Thus, non-functional concerns are not directly involved with the core logic of the application and thus they are not mandatory for the application software to provide the minimum service.

2.1.1 Class-level and Object-level Dynamic Weaving

Typically, there are two levels of approaches in employing dynamic weaving mechanisms for a real application, which levels are *class-level* and *object-level.* Each of these approaches has a different benefit and thereby mostly applies a different purpose. A class-level dynamic weaving allows composing an aspect with a set of *classes.* Since it affects all present and future objects, it can dynamically change the behavior of whole application software globally and continuously (Figure 2.1). It is useful for an application that requires several runtime changes spreading among that application. On the other hand, an object-level dynamic weaving allows composing an aspect with a particular set of *objects*. This kind of dynamic systems can change the behavior of a part of application software locally and temporarily at runtime (Figure 2.2). It is useful for an application, in which various crosscutting concerns are switched to follow the specific runtime environment and requirements.

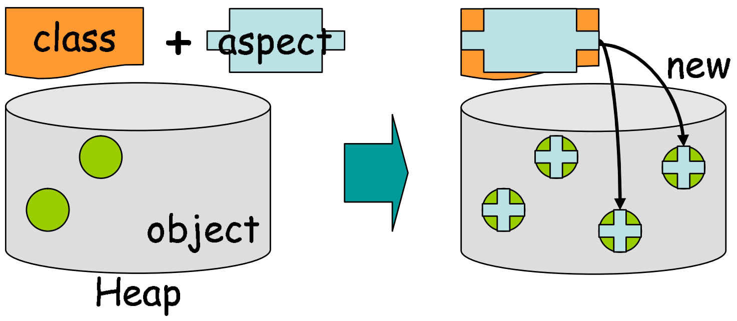


Figure 2.1. Class-level dynamic aspect weaving.

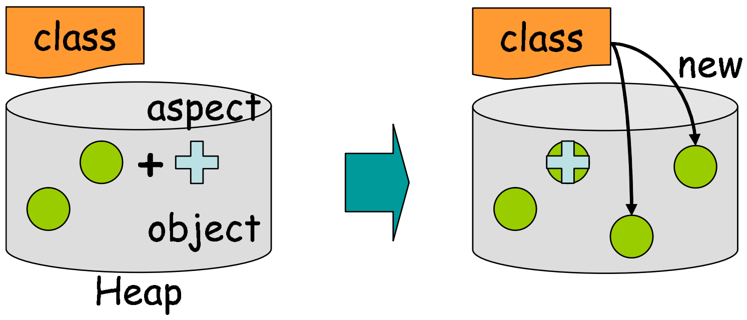


Figure 2.2. Object-level dynamic aspect weaving.

The rest of this section shows practical benefits using either or both levels of dynamic weaving mechanism for separating/coupling non-functional concerns from/with running application programs.

2.1.2 Rapid Prototyping — Logging, Debugging, and Profiling

The first benefit of a dynamic weaving mechanism, especially class-level, is that a developer can partially change an executed program during debugging without a time-consuming process of halting the runtime system, recompiling the program and restarting the system again. Since the developing iteration is annoying (Figure 2.3), this functionality especially provided by class-level systems is useful for rapid prototyping in the development process. This can shorten the lead time of the edit-deploy-run cycle of software development. Given a large amount of incremental changes and bug fixes which often need to be done during large software system development, this feature can dramatically improve the efficiency of development comparing to development with a static AOP system.

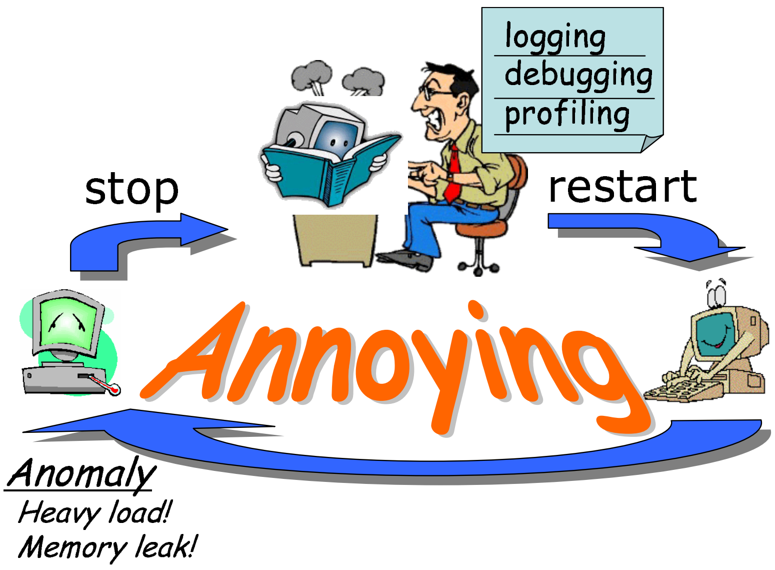


Figure 2.3. A typical iteration on developing and maintaining software.

Profiling a performance of software (or logging, debugging) is a good example showing that a dynamic weaving mechanism is useful. It is recognized as a non-functional concern that can be well modularized using AOP [24][31]. Since the code fragments for collecting profiling information tend to be spread over the whole program, they should be modularized into an aspect. As an example of aspects, this chapter presents a sample profiling aspect described in AspectJ: an implementation of AOP as an extension to Java. The `ProfileOperation` aspect:

```
public abstract aspect ProfileOperation {
    protected Timer timer = new Timer();
    protected int count = 0;

    abstract pointcut operation ();

    pointcut timedentry():
            operation() && ! cflowbelow(operation());

    before(): timedentry() {
        timer.start();
        count++;
    }

    after(): timedentry() {
        timer.stop();
    }

    public Time elapsedTime() {
        return timer.getTime();
    }

    public int operationCount() {
        return count;
    }
}
```

is defined as an abstract aspect. This aspect is a reusable module that encapsulates profiling concerns by defining *pointcuts* and *advice*. There are two advice, `before()` and `after()`, declared in the aspect for turning on and off the `timer`. To profile the elapsed time of an operation, these two advice are called at *join points* identified by the pointcut named `timedentry()`. Join points in AOP are execution points such as method calls, field accesses, instance creation, and exception handling. The `timedentry()` pointcut specifies a set of join points identified by the operation() pointcut defined as an abstract pointcut. Implementing the operation() pointcut in the subaspect of the ProfileOperatin aspect as follows:

```
public aspect ProfilePaintOperation
                    extends ProfileOperation {
    pointcut operation ():
        call(public void Figure+.paint(..)) ||
        call(public void Figure+.repaint(..)));
}
```

we can easily profile the elapsed time of painting operations performed by calling the methods paint() and repaint() declared in the Figure class as following:

```
ProfilePaintOperation.aspectOf().elapsedTime();
```

Note that adding the identifier + at the end of a class name makes the ProfilePaintOperation aspect weave with all subclasses of the Figure class. If we want to change the profiling points to other operations such as moving operations, the ProfilingOperation aspect can be reused as follows:

```
public aspect ProfileMoveOperation
                    extends ProfileOperation {
    pointcut operation ():
        call(public void Figure+.move(..));
}
```

Consequently, profiling concerns are well modularized into an aspect in AOP although they cut across the boundary of modules in OOP.

However, the performance profiling implemented on static AOP systems is not useful from the programmatic viewpoint. Suppose that the software is a Web-based business application, which must run 24 hours a day. Our scenario is that we first run the software without profiling code and, once it shows performance anomaly, perhaps under heavy load, we insert profiling code. The profiling code should be inserted without shutting down the software since the anomaly may be due to the workload up to that point. If the software is restarted, all the internal data structures are reset and hence the information necessary for analyzing the anomaly would be lost. Furthermore, we would need to interactively plug and unplug various kinds of profiling code until solving the anomaly. Each profiling code would cut across different execution points for collecting different profiling information. We thus need dynamic weaving mechanism. Although we could use large profiling code that collects all the information, it would imply serious performance impacts. We should use minimal profiling code at a time for reducing performance impacts. To satisfy these requirements, dynamic weaving is a good solution.

2.1.3 Adaptive Software — Caching and Prefetching of Query Results

The next benefit of dynamic weaving mechanisms, both class-level and object-level, is that we can develop services dynamically adapting in response to changes in the environment and requirements of clients. Since various degrees of runtime adaptabilities are needed, both class-level and object-level dynamic weaving mechanisms are useful for modularizing and switching these cross cutting concerns about runtime changes. For example, a security concern on which generality leads to fragile scatters related operations around the whole program. An adaptable aspect including various levels of security operations provides the flexible application, which can dynamically change its security policies depending on the client. Moreover, they allows an adaptable distributed GUI (Graphical User Interface) application, which can changes its configuration such as a deployment, a resolution, a bit rate, to suit it to each environment (e.g., DesktopPC, Laptop, PDA, i-mode).

Adaptable response caching of query results for a network and a database accesses in a web application is also a good example to show the usefulness of dynamic weaving. The implementation of the response cache includes not only caching the results of method calls (Figure 2.4) but also invalidating the cached results (Figure 2.5) that scatter in the software.

Since the response cache is a non-functional and crosscutting concern, it cannot be modularized with object-oriented programming; AOP is

```
 1: public String getQuote(String name) {
 2:     Cache caches = session.getCaches();
 3:     if (caches.containsKey(name)) {
 4:         result = (String) caches.get(name);
 5:     } else {
 6:         result = db.query("getQuote", name);
 7:         caches.put(result);
 8:     }
 9:     return result;
10: }
```

Figure 2.4. A method including codes for caching the query results in database accesses (line 2–5, 7–8).

```
1: public void updateQuotePrice(String name) {
2:     invalidate(name);
3:     Object result = db.query("updateQuote", name);
4: }
```

Figure 2.5. A method including codes for invalidating the cached data (line 2).

necessary [76].

Dynamic weaving enables efficient implementation of adaptable cache, and the class-level one suits for determining the global policy; the object-level one, for the local adaptation. To make the response cache adaptable, the software must be able to dynamically switch a number of aspects at the class-level or the object-level, in which various strategies are modularized, as the runtime environment changes (Figure 2.6). Yagoub *et al.* reported that there is no universal caching strategy that is optimal for all web applications and all the configurations [93]. For example, if the cache provided by an aspect shows a low hit ratio, the software should switch that aspect to another. If only part of the cache shows a high hit ratio, the software should remove the aspects that do not provide that part of the cache. The traditional object-oriented techniques like Design Patterns never modularize such a crosscutting concern, and still less switch it at runtime. Also, static weaving does not even work in this example. If we use static weaving, all the caching aspects must be statically woven in advance. Note that they are woven at different join points and hence, whenever the program execution reaches one of the join points, they must dynamically examine whether every cache is turned on or off. This runtime check causes a serious performance overhead. On the other hand, if we use dynamic weaving, only the activated aspects can be woven to avoid the runtime check.

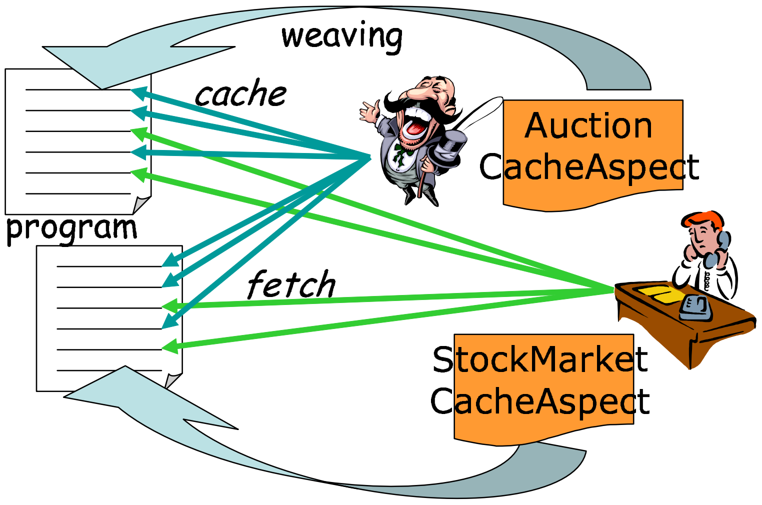


Figure 2.6. Appropriate aspects, in which various caching strategies are modularized, are woven into the running program as the client changes.

2.1.4 DI + AOP — Dynamic Aspect Injection

At last, object-level dynamic weaving mechanism could well work with a lightweight application container framework such as Spring [23], Aspectwerkz [44], HiveMind [4], PicoContainer [70], and Seasar [88]. These lightweight containers do not provide enterprise services but manage only dependencies of a collection of objects. A key feature of these containers is dependency injection (DI), or IoC (Inversion of Control) [35]. It is a programming technique for reducing the dependency among components and thereby enables loosely-coupled components, which reusability are highly improved. If a component includes sub-components, reusing only that component as is independently of those sub-components is often difficult. The idea of dependency injection is to move the code for instantiating sub-components from the program of a component to a component framework, which makes instances of sub-components specified by a separate configuration file (usually an XML file) and automatically stores them in the component. For example, suppose that the program of that component is as following:

```
public class BusinessTask {
    Database db;

    public void setDB(Database _db) {
      db = _db;
    }

    public Object query(String sql) {
        return db.query(sql)
    }
    .....
}
```

Note that this component contains an object of the `Database` interface as a sub-component. Using a lightweight container, the `db` field need not to be initialized in the constructor of the `BusinessTask` class as follows:

```
    public BusinessTask(int type) {
        switch (type) {
        case ORACLE: db = new Oracle(); break;
        case MySQL:  db = new MySQL(); break;
        case PgSQL:  db = new PgSQL(); break;
        }
    }
```

It is initialized (*or injected*) by a factory method provided by the lightweight container supporting dependency injection. Thus, a `BusinessTask` object must not be constructed by the `new` operator. Dependency injection loosens the connection between `BusinessTask` and database classes. It enables us to reuse `BusinessTask` without modification even if we must switch a database accessor to `MockDB` for unit testing. For example, the code snippet below constructs a `MyBusinessTask` object:

```
Resource res = new ClassPathResource("task.xml");
BeanFactory f = new XmlBeanFactory(res);
BusinessTask task = (BusinessTask)f.getBean("task");
```

Here, `ApplicationContext` is provided by a component framework. The `getBean` method constructs an instance of `BusinessTask` and initializes the value of the `db` field. It constructs an object implementing the `Database` interface and assigns it to the `db` field. This initialization is executed according to an XML configuration file `task.xml`.

The so-called DI + AOP containers generate synergistic effect for the developers of component-based applications since both reduce dependency among components. It is natural because DI and AOP share the same goal, which is to reduce dependency among components for better reusability [19]. The developers can inject dependency and compose an aspect in the same manner and obtain the initialized components from the container, specifying dependency and aspects between components by a separate configuration file (usually an XML file).

2.2 Problems of Ordinary Techniques

This section explains problems of ordinary techniques for implementing dynamic weaving in the AOP system. Typical implementations of object-based AOP systems, including both static and dynamic AOP, insert *hooks* at a number of join points such as method calls, field access, instance creation, and exception handling. If the program execution reaches join points, the inserted hook intercepts it and executes a piece of advice code if it is included in a set of join points identified by pointcuts.

In most AOP systems that provide static weaving mechanism, a hook is usually implemented as inlined hooking code, in which pieces of aspects are directly embedded into a base program by static translations of source code or bytecode. However, several join points cannot be uniquely determined by the conditional pointcuts, such as *this*, *target*, *args*, and *cflow*. Such a set of join points depends on the current execution context and changes dynamically. Thus, hooking code must be embedded into potential join points with conditional statements, which examine if the advice should be executed in the execution context.

Generally, an AOP system that provides dynamic weaving mechanism must examine whether any advice should be executed at every join point when the execution of a program passes that point. For dynamic weaving, all the join points are dependent on the execution context, since the set of join points are specified at runtime. Furthermore, the set of join points changes dynamically. Thus, the check whether or not the system should execute advice must continue after the join point has been specified.

In this section, the following three techniques are explained:

- Static code translation.
- Hot deployment for class-level weaving.
- Hot deployment for object-level weaving.

The first two techniques are mainly used for class-level dynamic weaving. Static code translation is a naive approach and has a performance drawback. Hot deployment is a popular but a sweeping approach, since dynamic weaving is done at the component level. The hot deployment mechanism is also used for enabling object-level dynamic weaving. It has a drawback caused by the *version barrier* explained deeply in the last of this section.

2.2.1 Static Code Translation

There exists a well-known approach that enables every join point in a class or an object to be checked at runtime, and which is supported by static code translation of application programs. For example, JAC [69] and Handiwrap [6] support the dynamic weaving mechanism using a static code translation approach, in which a compiler (or a translator) inserts minimal hooks for all potential join points (Figure 2.7). They translate the code of a program to a version with inserted hooks. The translation is performed by the source-to-source or binary-to-binary, during compilation or class loading. Most AOP systems also use static code translation and this is more or less appropriate to their purpose because most intercepted join points are identified statically.

Static code translation does not cause much of a performance penalty in advice execution, while involving some overhead in normal operations with no woven aspect. The execution of advice is fast since an inserted

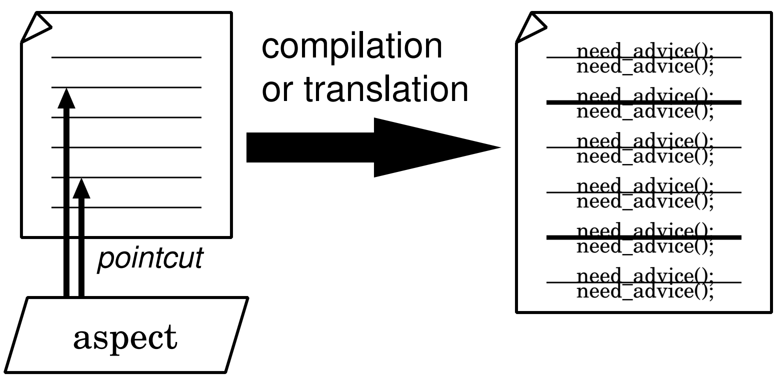


Figure 2.7. Static code translation.

hook is represented as just a method call. However, even if no aspects are woven, all checks whether or not the system should execute advice is performed. This results in unnecessary method calls or verbose indirection of object references, which involves overhead in normal operations that cannot be ignored.

Popovici *et al.* [71] and Bockisch *et al.* [9] have implemented a Just-In-Time (JIT) aspect compiler based on the IBM Jikes RVM [5]. Their JIT compiler inserts hooks at all potential join points only at the time of the first just-in-time compilation. Thus, their work can be regarded as a static code translation approach mentioned above. They avoided adding options to the JIT compiler that could recompile bytecode since that would increase the complexity of the JIT compiler support too much. They reported they could limit the overhead due to the hooks since their hooks are implemented using native code, not Java byte code. Unfortunately, the JIT compiler approach is irreconcilable with recent high-performance runtime technologies like Sun's HotSpot(TM) technology or the IBM JIT compiler [90], which involves the mixture of a JIT compiler and interpreter.

2.2.2 Hot Deployment

2.2.2.1 Hot Deployment for Class-level Weaving

Most of application servers, either commercial (e.g., Websphere [8], Weblogic [43]) or open-source (e.g., JBoss [56], Tomcat [14], Geronimo [3]), provide a kind of the dynamic weaving functionality based on Java class loaders. Some AOP environments such as Aspectwerks [44], Spring [23], and Seasor2 [88] also adopt it. Although class loaders prevent loaded classes being *reloaded*, they allow *redeploying* an application component on the same process of the JVM. This mechanism is especially called the *hot deployment* functionality in the application server domain, which enables software components to be plugged and unplugged without restarting application servers. An application component can be dynamically customized with an aspect on a per-component basis. If a code translator composes an aspect with the application program during deployment, loading, or compile time, loading a modified version of that application component by using a new class loader seems to enable dynamic aspect weaving at the class-level.

However, some existing objects are discarded if the old application component is redeployed by using the new class loader. These old objects can not keep existing in the process of the JVM as well as not being composed with an aspect (Figure 2.8). This is because the old classes are discarded together with that class loader. This is remarkably inconvenient in practice. For example, we can not profile a performance of application software once it runs. Furthermore, it is considered a *component-level* approach rather than class-level. We can not easy-to-use that component-level dynamic weaving mechanism for shortening the lead time of the edit-deploy-run cycle of software development.

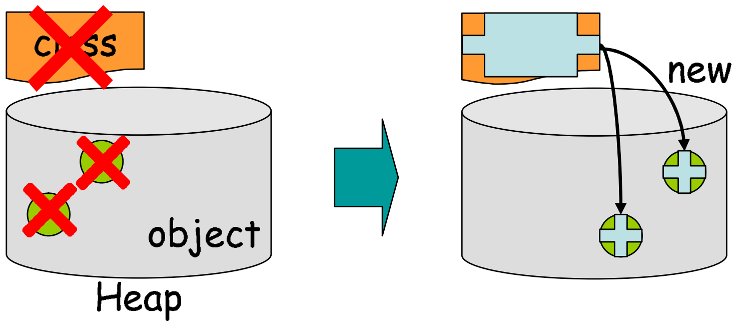


Figure 2.8. Redeploying a component by a class loader does not care about existing objects.

2.2.2.2 Hot Deployment for Object-level Weaving

With all coarseness of component-level aspect weaving, some AOP systems adopt the hot deployment mechanism for object-level dynamic weaving. Unlike the case of class-level weaving, a class loader is used as just a factory of the objects that are composed with an aspect. For example, the following pieces of code show the weaving process:

```
ClassLoader loader = new CustomClassLoader(...);
byte[] modifiedClass
    = weaver.compose("BookMall","SessionCache");
Class c = loader.load(modifiedClass);
BookMall mall = (BookMall) c.newInstance();
```

Creating a custom class loader, putting the caching concern on the `BookMall` class using an aspect weaver, loading the modified version of the `BookMall` class into the different namespace from that of the running application, the users can get an object of the new `BookMall` object, which contains a session caching concern. A namespace is a map from the class names to the class definitions. A set of classes included in the same component *joins* [1] its own namespace and thus naming conflicts between components can be avoided. Moreover, a component can be dynamically and individually updated to be composed with an aspect without restarting the whole execution environment.

However, the running application can not deal with the `mall` object due to the strict separation of namespaces in Java, ironically called the version barrier (Figure 2.9). The version barrier is a mechanism that prevents an object of a version of a class from being assigned to a variable of another version of that class. In Java, if a class definition (i.e. class file) is loaded by different class loaders, different versions of the class are created and regarded as distinct types. If two class definitions with the same class name are loaded by different loaders, two versions of the class are created and they can coexist while they are regarded as distinct types. The version barrier is a mechanism for guaranteeing that different versions of a class are different types. Regarding two versions as distinct types is significant for performance reasons. If not, advantages of being a statically typed language would be lost. Therefore, Java's class loaders,

[1]A class joining a namespace means it is being loaded by the class loader that creates the namespace.

which play a central role in runtime flexibility of Java, does not readily underlie the object-level dynamic weaving mechanism in AOP.

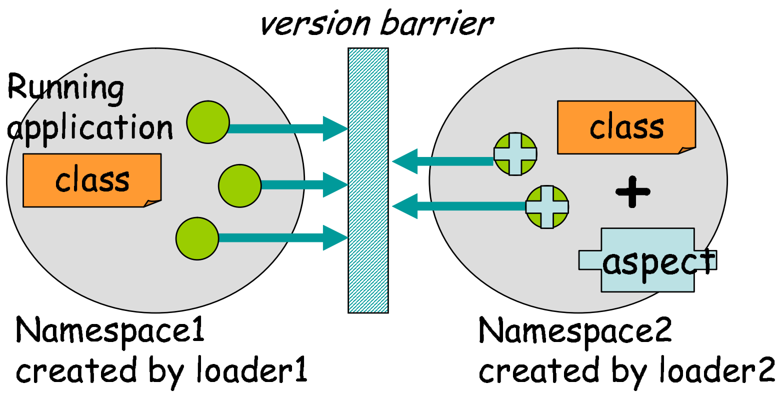


Figure 2.9. The version barrier.

Then, the DI + AOP containers recommend us to use an interface type as the type of a variable that can refer to instances of multiple versions of a class even if each version is composed with a distinct aspect. The underlying mechanism is especially called the *AOP proxy* in the container framework community. The sequence diagram for the AOP proxy is shown in Figure 2.10 and the intuitive class diagram is shown in Figure 2.11. In fact, the `AopProxy` class implements the `IBusinessTask` interface and dispatches all method calls to the actual `BusinessTask` object. This mechanism is based on the STRATEGY PATTERN and the TEMPLATE METHOD PATTERN. The key technique is to load the `BusinessTask` class and the same name class but composed with `CacheAspect1` into separate namespaces. The latter version of that class can be dynamically loaded, and thus the cache strategy can be dynamically changed. In fact, the AOP proxy class works between the `IBusinessTask` interface and the `BusinessTask` classes. In Java 1.3, the Dynamic Proxy API [80] has been introduced to facilitate users employing this technique easily. However, this technique requires programmers to define an interface type for every multi-versioned class and access instances of the class through the interface type.

Some of the DI + AOP containers do not force us to define a component class as an interface type, however, the component becomes a non-POJO (Plain Old Java Object). These containers implicitly gen-

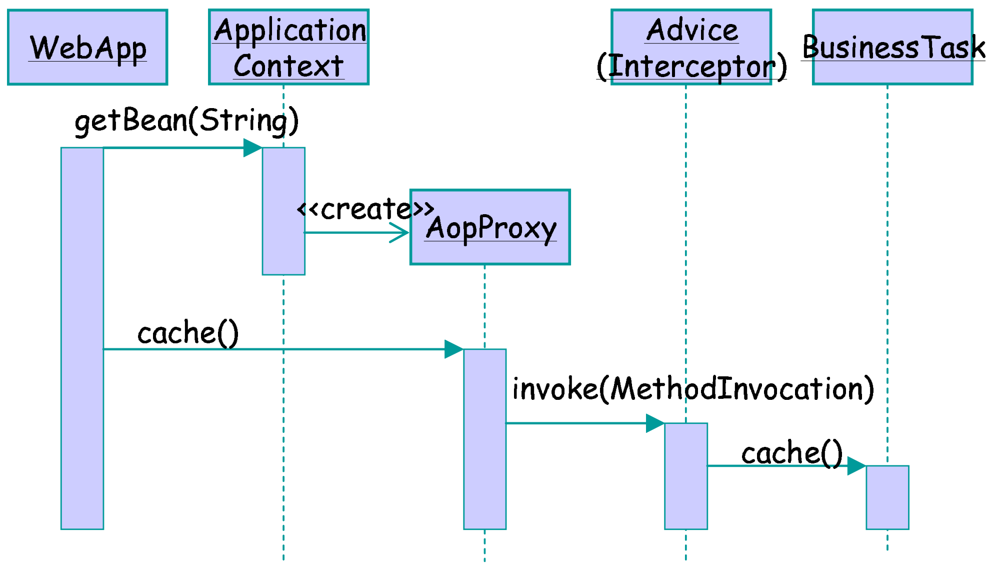


Figure 2.10. The `AopProxy` object intercepts all method calls to the `BusinessTask` component.

erates a subclass of that component using a bytecode engineering tools such as CGLIB [15], BCEL [57], and Javassist [18] as shown in Figure 2.12. The generated subclass is composed with an aspect and loaded into the container framework at runtime. For example, a subclass of the `BusinessTask` class, the `BusinessTask$1` class, is dynamically composed with the `CacheAspect1` class. Thus, the component needs not to be declared as an interface type. For example, even if the `BusinessTask` component is declared as a regular class type in the code snippet as shown below, the container framework can generate the multiple version of the `BusinessTask` class while composing various aspects with the generated subclasses of it:

```
BeanFactory factory
  = new ClassPathXmlApplicationContext("task.xml");
BusinessTask cs
  = (BusinessTask)factory.getBean("task");
```

However, the generated subclass must override all methods declared in the extending component. It means that the component must not be

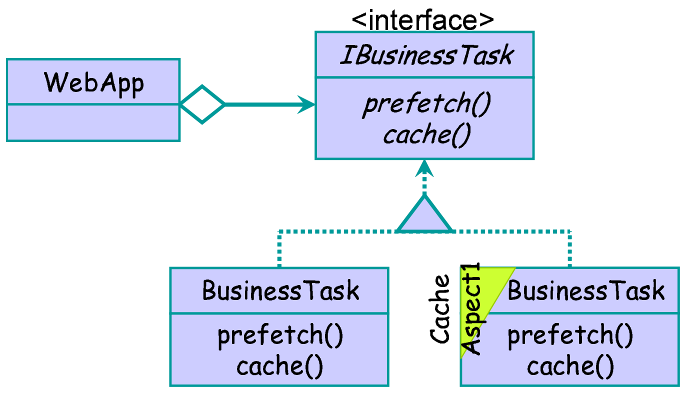


Figure 2.11. The `BusinessTask` component composed with `CacheAspect1` can be used as the `IBusinessTask` type.

declared as a final class and each declared method must not be declared as a private method. That component is no longer a POJO.

2.2.2.3 The Version Barrier

The rest of this section presents two example problems actually caused by the version barrier, which developers often encounter when developing a component-based application in Java. In fact, a number of Java developers have reported a problem imposed by the version barrier.

J2EE components Most J2EE platforms, either commercial (e.g., Websphere, Weblogic) or open-source (e.g., JBoss, Tomcat), support both the development and the deployment of pluggable component archives (EJB-JARs, WARs, and EARs). A Web Application Archive (WAR file) is used to deploy a web-based application. This file can contain servlets, HTML files, Java Server Pages (JSPs), and all associated images and resource files. An Enterprise Application Archive (EAR file) may contain one or more Enterprise JavaBeans (EJBs) and WARs. The functionality of the so-called hot deployment enables such J2EE components to be plugged and unplugged at runtime without restarting the application servers. Thus, a J2EE application can be dynamically customized on a per-component basis. This dramatically improves the productivity of software development. For enabling hot deployment, each component joins a distinct namespace, loaded by a distinct class loader.

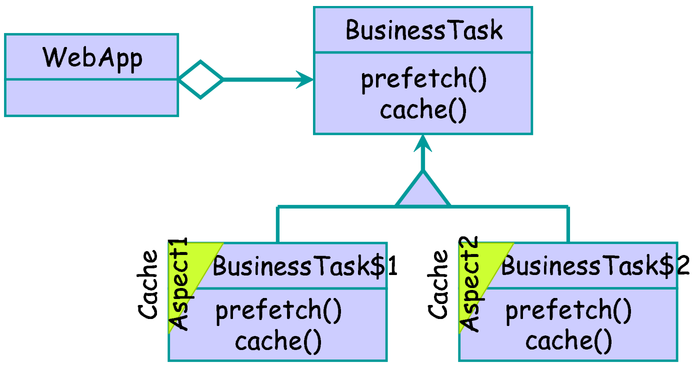


Figure 2.12. A subclass of the `BusinessTask` class such as the `BusinessTask$1` class and the `BusinessTask$2` can be dynamically generated with an aspect such as the `CacheAspect1` aspect and the `CacheAspect1` aspect.

However, the version barrier makes it impossible to pass instances of each version of a class across the boundary of J2EE components, or namespaces. Such instances are typically caches, cookies, or session objects or beans. For example, consider the following scenario. An instance of the `Cart` class must be passed between servlets included in different web application archives (that is, from the `BookMall` included in one web archive, WAR1, to the `OnlineBank` in another web archive, WAR2). The class file of the `Cart` class is packaged into a Java Archive (JAR) file, and identical copies of that JAR file reside in the `WEB-INF/lib` directories in each web archive. Thus, each class loader loads the `Cart` class separately. Figure 2.13 illustrates the implementation of these servlets: `BookMall` puts an instance of `Cart` into the session cache and `OnlineBank` pulls that instance out of the cache. When casting it from `Object` to `Cart`, the JVM will throw a `ClassCastException`. Since the `Cart` class referenced by the `BookMall` class is a distinct type from the type referenced by the `OnlineBank` class, the version barrier prevents assignment of that instance to the variable `cart` in the `OnlineBank` class by throwing a cast error in advance.

Some readers might think the delegation model of class loaders in Java is a solution to the problem above. These WAR components can share the same version of a class if they delegate the loading of that class to their common parent, such as the `EAR` class loader (Figure 2.14). In fact, the typical J2EE platform has such a common parent loader. Child class

BookMall.select()

```
SessionCache cache
   = session.getCache();
cart = new Cart();
cart.put(item);
cache.add("cart", cart);
```

runs in a servlet in WAR1.

$\Longleftrightarrow$

OnlineBank.estimate()

```
SessionCache cache
   = session.getCache();
Object object = cache.get("cart");
cart = (Cart) object;
```

runs in a servlet in WAR2,
and throws a `ClassCastException`.

Figure 2.13. Passing the session cache from one WAR component to another.

loaders can have their parent loader load a class if they want to share the same version of that class. In the case of J2EE, the `SystemClassLoader` is the parent of all EAR class loaders and an EAR class loader is, in turn, the parent of all WAR class loaders included in that EAR. However, the solution using a parent class loader tightly couples several irrelevant J2EE components together. Such coarse-grained composition decreases the maintainability and availability of all related software components. For example, consider the two components `DVDStore` and `Pizzeria`: the former models an online 24-hour DVD store and the latter models an online home delivery pizzeria available from hours 10 to 21. If both of these components share the above-mentioned application component including `Cart` and if this component is packaged into `Pizzeria`, then undeploying `Pizzeria` for maintenance stops the service by `DVDStore`. Since `DVDStore` must run 24 hours a day, it is almost impossible to decide the maintenance schedule of `Pizzeria`.

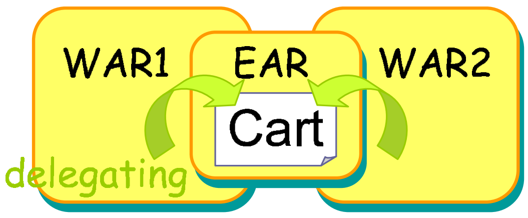


Figure 2.14. A parent EAR class loader is used for sharing class types between WAR1 and WAR2. The rounded box represents a namespace for the J2EE component. The overlapping part means the overlapped namespace.

To solve this problem, the JBoss application server provides the unified class loader (UCL) architecture [55] for sharing across components across the J2EE components. A collection of UCLs acts as a single class loader, which places into a single namespace all the classes to be loaded. All classes are loaded into the shared repository and managed by this repository. However, this architecture disables different J2EE components with the same name (Figure 2.15).

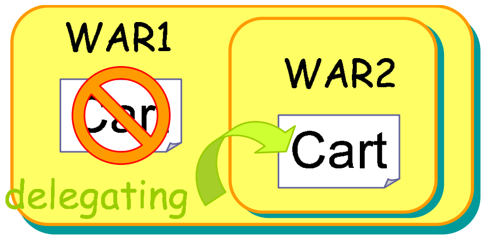


Figure 2.15. The JBoss application server based on the unified class loader architecture makes a parent-child relationship between the communicating components.

Another technique, considered a last resort, is using the Java Serialization API to exchange objects between different J2EE components through a byte stream, which is the referred to as *Call-by-Value* (Figure 2.16). Typical J2EE platforms adopt this approach for inter-EAR communications. However, even if an EAR wants to transfer an object to another EAR deployed in the same container (or JVM), it must execute a remote call. This remote call is a waste of I/O resources and it decreases the overall performance. Although the Local Interface mechanism introduced in EJB2.0 allows communications between components without remote calls (*Call-by-Reference*), these components must be packaged together in the same archive.

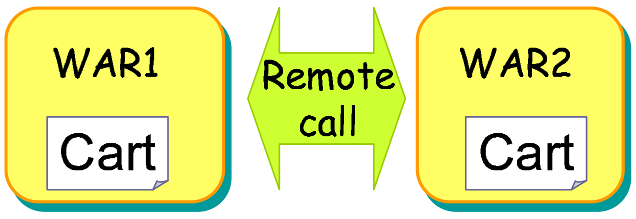


Figure 2.16. All inter-component communications are realized by a remote call.

The similar problem occurs in the case of object-level dynamic weaving in the J2EE platform. For example, the following aspect:

```
public aspect SessionCache {
    Cart Customer.cart = new Cart();

    pointcut select() :
      call(public void BookMall.select(Item));

    before(Item item) : select() && args(item) {
        cart.put(item);
    }

    public Cart get(String key) {
        return cart;
    }
}
```

represents the caching concern, which is removed from the `BookMall` class in figure 2.13. For example, the following pieces of code seem to allow object-level dynamic weaving::

```
ClassLoader loader = new CustomClassLoader(...);
byte[] modifiedClass
    = weaver.compose("BookMall","SessionCache");
Class c = loader.load(modifiedClass);
BookMall mall = (BookMall) c.newInstance();
```

Creating a custom class loader, putting the caching concern on the `BookMall` class using an aspect weaver, loading the modified version of the `BookMall` class into the different namespace from that of the running application, the users can get an object of the new `BookMall` object. However, the running application can not deal with the `mall` object due to the version barrier.

Eclipse plug-in framework The Eclipse platform [86], an integrated development environment for Java, can be considered as a component system due to its advanced plug-in framework. A plug-in module can contain all sorts of resources, including code and documentation. A plug-in module must also contain sufficient information for the platform to incorporate the code or documentation into itself. The plug-in framework allows us to easily add, update and remove discrete portions of the contents. In addition, since a separate class loader (called a plug-in class loader) is created for each plug-in module, each plug-in module has its unique namespace and is dynamically deployable.

However, the Eclipse plug-in framework has a structural problem due to the version barrier. For example, consider the Eclipse help system plug-in module [38]. It is a useful plug-in module that allows users to develop and deploy professional-quality, easy-to-use, and searchable online documentation. The Eclipse help system can be used as an infocenter, which is an application implemented as a web component and accessible from a web browser. However, to be used as an infocenter, the current Eclipse help system needs to run on a separate process from the process of the web server (Figure 2.17). The web server must make new processes for the help system and the minimum Eclipse system, and then the web server must dispatch all requests to the help system. Thus, every communication for dispatching requests from the web server to the help system is a remote call, which involves marshalling all passed instances.

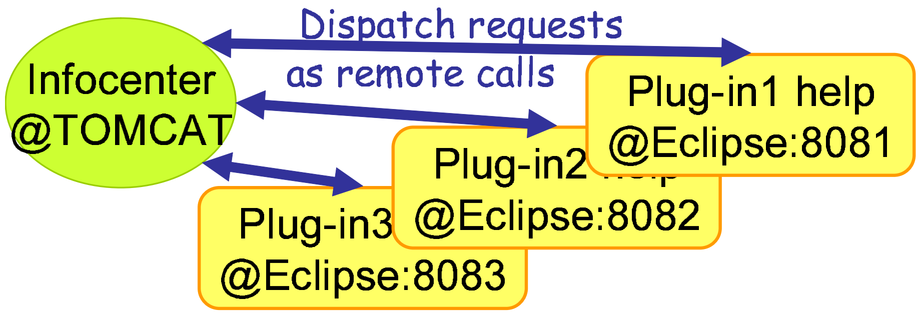


Figure 2.17. The Eclipse help system must run as a separate process.

A real problem of the example above is that, no matter which namespace the help system joins, all instances must be marshaled and unmarshaled with performance penalties to avoid trouble due to the version barrier when they are passed between the web server and the help system. This is true even if the help system is run on the same process as the web server. Suppose that the help system runs on the same JVM as the infocenter, and both the help system and the infocenter use the Apache Xerces [85] archive, which contains an XML parser in the `WEB-INF/lib`

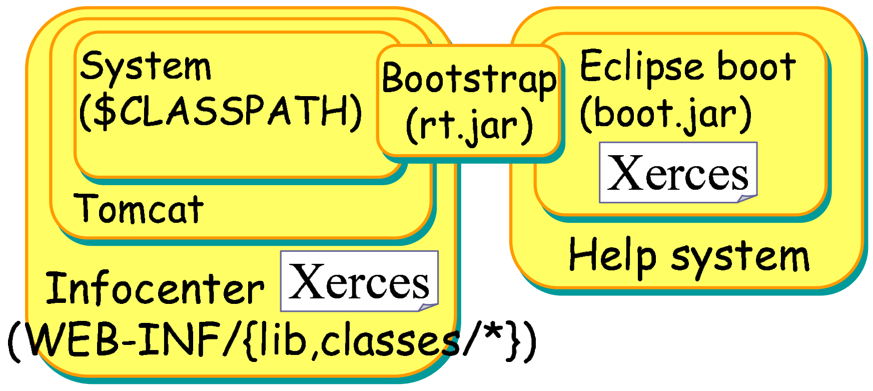


Figure 2.18. The Xerces archives are loaded in duplicate for the Eclipse help system and the infocenter.

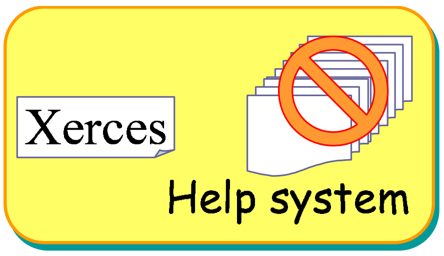


Figure 2.19. Loading all components by a class loader breaks the isolation of each namespace.

directory. If the help system joins a namespace independent of the namespace of the infocenter (Figure 2.18), the version barrier does not allow the instances of an XML parse tree to be exchanged between the help system and the infocenter, since the copies of the Xerces archive are loaded in duplicate and then different versions of the tree-node class types are created for each archive. If the help system joins the same namespace as the infocenter by deploying as a WAR file into the `WEB-INF/lib` directory (Figure 2.19), the XML parse tree can be exchanged between the two components. However, this obviously breaks the isolation of the help system from the infocenter. For example, several core components of the Eclipse platform must also be loaded together with the help system, and these core components cause naming conflicts with the infocenter. Furthermore, all the components must be redeployed together when some

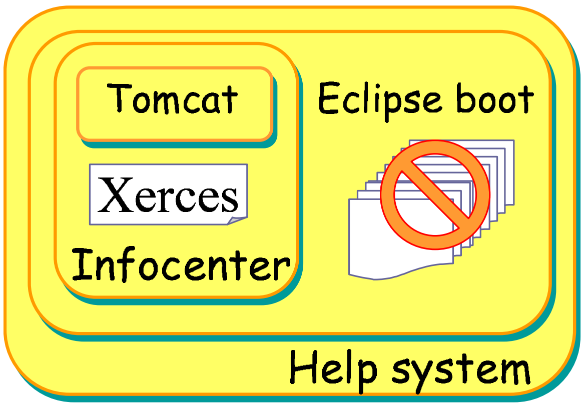


Figure 2.20. Delegating the Xerces archives to the web component class loader breaks the isolation of the help system.

of the components are redeployed for maintenance. Finally, if the help system joins a descendant namespace of the infocenter (Figure 2.20), delegating the Xerces archives to the parent class loader also allows sharing the Xerces archives. However, it ends up breaking separated namespaces, too.

Extending assignment compatibility The problems illustrated above can be solved if the algorithm for computing *assignment compatibility* in the Java programming language is extended to include version conversions between different versions of a class type. Here, the version conversion means a conversion from a version of a class type to any other version of that class type. If this conversion is chosen in the context of assignment, casting, and method invocation conversions such as widening and narrowing conversions, instances could be easily passed across the version barrier [2]. For example, this extension of assignment compatibility would allow assignments between different versions of a class type. Thus, a component would be able to pass instances into and from another component, even if both components load and define that class type separately. The `OnlineBank` class in Figure 2.13 would not throw a cast error. Moreover, the Eclipse platform would not need to care about where and how many Xerces libraries are available in the current execution environment.

[2]If two class types have assignment compatibility with each other, one type can be converted to the other type in the context of not only assignment conversions but also casting and method invocation conversions.

However, naively relaxing the version barrier by extending the assignment compatibility causes a serious security problem. For example, a program may access a non-existing field or method and then crash the JVM. In fact, the version barrier of Sun JDK 1.1 was wrongly relaxed, and thus it had a security hole known as the type-spoofing problem, first reported by Saraswat [91]. This security hole had been solved by the loader constraint scheme [51], which rather strengthens the version barrier. To avoid this security problem while relaxing the version barrier, it would be necessary to have runtime type checking, as is found in dynamically typed languages such as CLOS, Self, and Smalltalk. In such languages, since a variable is not statically typed, any type of instance can be assigned to it. For security, several interpreters for dynamically typed languages perform runtime type checks, called guard tests, so that an exception can be thrown at runtime if a non-existing method or field is accessed. A drawback of this approach is that it requires frequent runtime type checks, which implies non-negligible performance degradation, whereas the JVM performs these runtime type checks. Another technique is to perform runtime type checks at every assignment operation, such as the `aastore` Java bytecode instruction, which is used for storing an object reference in an array object. This operation verifies that the stored object is type-safe. However, this approach also causes performance degradation, since the JVM must perform a type-check for not only `aastore` but also for a large number of other assignment instructions.

2.3 Summary

This chapter illustrated practical benefits of dynamic weaving in AOP and currently known techniques for enabling it. We showed three motivating examples, rapid prototyping, adaptive software, and aspect injection, and two well-known techniques, static code translation and hot deployment.

Through this chapter, we have seen that the class-level and the object-level dynamic weaving mechanisms are effectively used for a different purpose. The class-level dynamic weaving mechanism is useful for an application that requires several runtime changes spreading among that application; the object-level dynamic weaving mechanism, for an application that contains various crosscutting concerns to be switched to follow the specific runtime environment and requirements.

Table 2.1 summarizes the features of the two techniques. Both of them allow implementing the class-level and the object-level dynamic weaving mechanisms. However, the static code translation technique causes non-negligible performance penalties while the regular execution with no woven aspect. The overheads become more serious in the case of object-level dynamic weaving mechanisms due to the heavyweight hooks involving look-ups of appropriate objects. The hot deployment technique employing class loaders is a valuable but sweeping approach for the class-level dynamic weaving mechanism. It requires all classes and those existing objects within a woven component are once abandoned even for one woven class. Furthermore, the version barrier forces developers to follow the complicate manner while adopting this technique for object-level dynamic weaving mechanisms.

Table 2.1. Static code translation and hot deployment.

	Class-level aspect weaving	Object-level aspect weaving
Static code translation	○	△
Hot deployment	△	○

In the following chapter, we propose new two mechanisms for addressing efficient, fine-grained, dynamic weaving mechanisms at the class-level and the object-level, respectively. The former allow selective, just-in-time, aspect weaving for enabling efficient class-level dynamic weaving, the latter one can relax the version barrier between namespaces for efficient object-level dynamic weaving.

Chapter 3

A Selective, Just-in-Time, Weaver

This chapter presents Wool [75], which is a selective, just-in-time, aspect weaver for Java. The previous implementations of class-level dynamic weaving techniques suffered from serious performance penalties. This chapter presents our new efficient class-level dynamic weaving technique in Java for addressing the underlying problem. This system called Wool is a hybrid of two approaches. When a new aspect is woven in, the programmers can select to reload into the JVM a modified class file in which hooks for executing advice are statically embedded, or they can insert hooks as breakpoints in the JVM. Since the two approaches have different performance characteristics, the programmers can select the best one for each join point.

Recently, practical demands are being made of dynamic aspect-oriented programming (AOP [49]) systems [68, 69, 6, 72, 71]. Unlike static weaving in AOP, a dynamic weaving mechanism allows dynamically weaving and unweaving an aspect into/from a program. Moreover, advice and pointcuts are changeable during runtime. These dynamic features extend the application domains of aspect-oriented programming. Dynamic weaving can make development cycles shorter [24] and it allows for aspects that can adapt the behavior of application software at runtime to follow the changes of the runtime environment and requirements [7, 32, 76].

The most typical technique for implementing class-level dynamic weaving mechanisms is based on static code translation although it is not efficient. This approach statically inserts pieces of code, which we call hooks, into all join points, and these hooks determine at runtime whether or not there is associated advice to be activated at each join point, in contrast to static weaving in AspectJ [48]. These runtime checks imply serious performance overhead although they are necessary since dynamic weaving allows turning advice on and off during runtime.

This chapter presents our Java-based dynamic AOP system called *Wool*, which exploits our new implementation technique for addressing the performance problem mentioned in Chapter 2. Wool inserts hooks into a program at runtime *just in time* when the programmer directs the program to start using an aspect. Wool allows the programmers to select from two implementation techniques the best one for each join point. The first one is to insert the hooks as breakpoints handled through the debugger interface of the Java virtual machine (JVM). The other one is to produce a program in which the hooks are embedded as method calls and reload that new program into the JVM. These two techniques do not require a custom JVM, but work with the standard JVM.

The rest of this chapter is organized as follows. Section 3.1 presents our new implementation technique for class-level dynamic weaving. It also shows an overview of the current implementation of Wool. Section 3.2 compares Wool to other AOP systems. Section 3.3 presents the results of our experiments. We conclude this chapter in section 3.4.

3.1 Wool

We developed Wool, which inserts hooks into the program on demand, in Java. Since the hooks are inserted after all of the intercepted join points are specified, Wool does not insert unnecessary hooks. This section presents the details of our new dynamic AOP system Wool and shows how it enables efficient class-level dynamic weaving.

3.1.1 An overview of Wool

Wool is implemented as a Java library that provides dynamic weaving functionality, consisting of APIs to write aspects, a weaver to compose aspects with programs, and a subsystem for accepting a request for weaving from the outside of the running program.

Wool allows the aspect to be woven either locally, from within an application running on the same JVM, or remotely when sent to the subsystem of Wool. The following code shows how the aspect is woven in by Wool.

```
WlAspect azpect = WlAspect.forName("ProfileAspect");
Wool wool = Wool.connect("localhost", 5432);
wool.weave(azpect);
```

In a locally woven case, the aspect instance `azpect` is created in the running program. The weaver instance `wool` is connected to the subsystem of Wool. Weaving runs immediately after the method `weave()` is called. Alternatively in a remotely woven case, the aspect instance is actually created and recomposed outside of the JVM in which it will be woven. It is then serialized and sent over the network to the subsystem of Wool in the target JVM.

3.1.2 Just-in-time hook insertion

Wool adopts a hybrid approach so that the programmers can choose a suitable hook at a join point considering the entire cost, and which hooks are breakpoints or method calls. In Wool, just-in-time hook insertion is done in two time-frames at runtime, as shown in Figure 3.1.

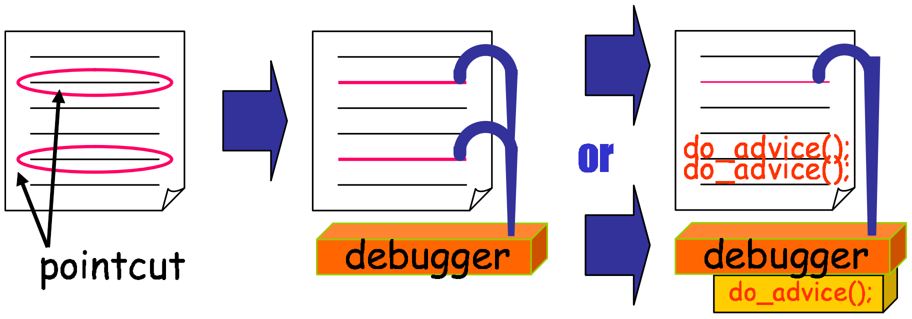


Figure 3.1. Two time-frames for hook insertion.

The strategy for deciding at the hooked join point whether advice is executed or embedded into the program is simple. All of the hooks are represented as breakpoints first. At each hooked join point, there are alternative ways, one is executing pieces of advice by using the debugger and the other is embedding hooks into the program using dynamic code translation. If the hooked join point using a breakpoint is judged likely to be intercepted again and again in the future, and if the degradation it causes is estimated to be higher than that caused by dynamic code translation, such a hook should be embedded into the program instead of executing the advice by using the debugger. After the hook is embedded, the breakpoint at the join point is removed.

A comparison of the implementation techniques of dynamic weaving mechanisms is shown in Table 3.1. Wool is a hybrid of the last two techniques. Unlike static code translation, both of the two techniques that Wool adopts do not insert any unnecessary hooks (Column 1 in Table 3.1).

Table 3.1. Comparison of the three approaches. Wool is a hybrid of the last two techniques, which are using breakpoint-based execution and dynamic code translation. Each column indicates the degree of the efficiency of using that approach.

	Static code translation	Breakpoint-based execution	Dynamic code translation
frequently executed advice	○	×	○
rarely executed advice	×	○	×
hook insertion	○ (statically)	△	×
normal operation	×	○	×
without aspect	×	○	○

3.1.2.1 As a breakpoint

The first hook insertion method, which we call breakpoint-based execution, where all the hooks are inserted as breakpoints, which are set at runtime through standard debugger interface in Java called JPDA (Java

Platform Debugger Architecture) [82]. The JPDA allows a programmer to register requests for execution events inside a JVM and controls execution for each event notification. These breakpoints are set for all join points specified by a pointcut. If the thread of control reaches one of the breakpoints, it switches to the debugger thread and the advice associated with that join point (breakpoint) is run. Using JPDA doesn't require the modification of the runtime system.

The execution overhead due to breakpoint is not a serious problem since the HotSpot (TM) VM that comes with the Java 2 SDK 1.4 runs a program together with a just-in-time compiler even if any breakpoints are set. In addition, hooks in the form of breakpoints can be inserted into programs so quickly (Column 3 in Table 3.1). Although programs must be run in a debug mode, it doesn't cause much performance penalty under normal operations without active advice (Column 4 in Table 3.1).

For frequently executed advice, the overheads for breakpoint-based execution are not negligible (Column 1 in Table 3.1). The large number of context switches to execute the advice causes the overhead, since advice has to be executed separately in the debugger process (Figure 3.2).

3.1.2.2 As a method call

The second hook insertion method, which we call dynamic code translation: To reduce the overhead caused by context switches, a frequently invoked join point expressed as a breakpoint is replaced with a modified method in which the hooks are directly embedded as shown in Figure 3.3. The method body is modified at the bytecode level so that a bytecode sequence for executing the advice is embedded at the join points contained in the method body. At the breakpoint, all join points specified by the pointcut are identified, so hooks can be statically embedded into the programs without garbage (unnecessary) hooks as in other static-code-translation-based dynamic weaving mechanisms.

The runtime replacement of bytecode is done using the hotswap mechanism [26] of the JPDA. The hotswap mechanism allows a new class to be reloaded at runtime while under the control of a debugger. The actual reloading isn't performed immediately when the static code translation is completed, because the cost of such a translation is very large. If there is a method that should be replaced with a hook embedded method, dynamic code translation is forked, the breakpoint-based execution continues until the translation is finished. Therefore, the dynamic code translation

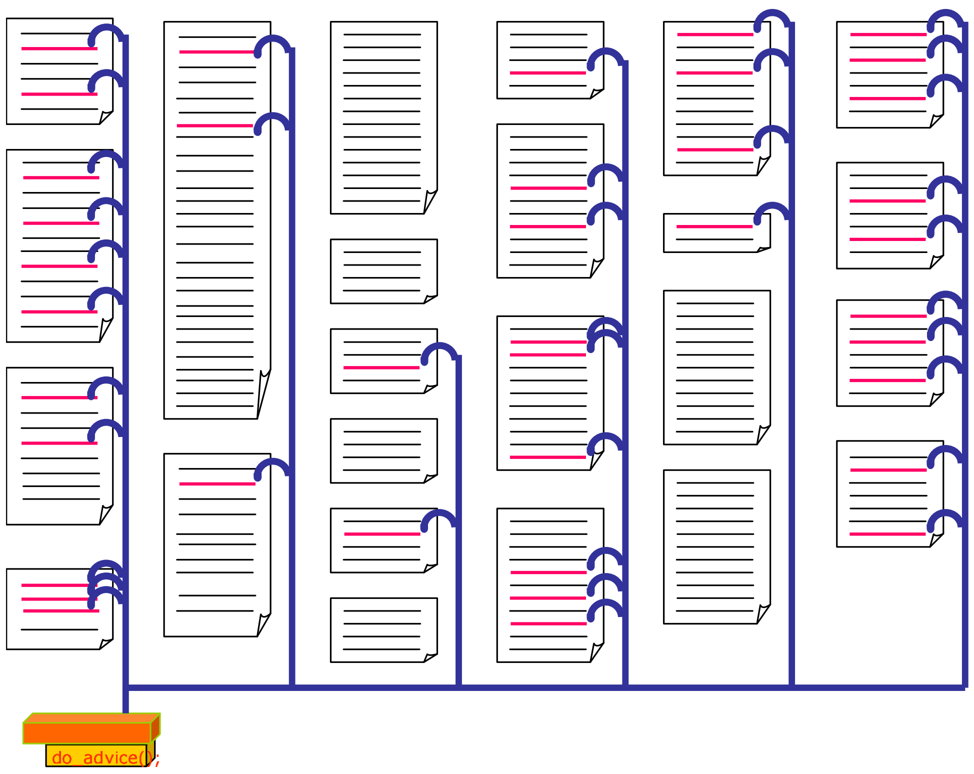


Figure 3.2. Tons of context switches are caused for frequently executed advice.

stops the application thread for a short time and uses the translation time effectively. After replacing the method, the thread of control does not stop at the join points contained in the method body. The hooks are embedded into the program as simple method calls, and therefore the advice execution is much faster than using the debugger (Column 1 in Table 3.1).

Dynamic code translation is not efficient under certain circumstances. It causes only a single context switch to embed hooks into the program. However, the cost of the translation and the hotswap performed for every crosscut class is relatively high if advice is rarely executed (Column 2 and 3 in Table 3.1). In this case, dynamic code translation is just unnecessary as most of hooks are in the static code translation approach in section 2.2.

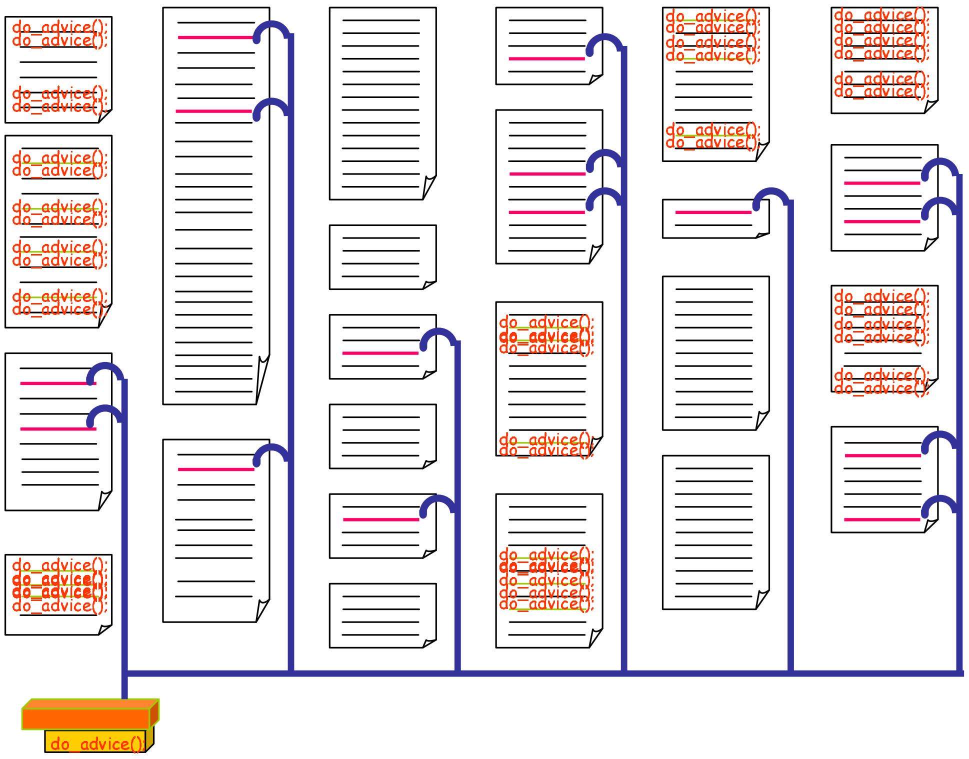


Figure 3.3. Well-balanced hooks as a breakpoint and a method call.

3.1.3 Aspect in Wool

Wool provides the programmers with APIs to define an aspect in Java. It does not provide a special aspect language for easily writing an aspect, which is different from the languages such as AspectJ or any others that are intended to enhance flexibility and abstraction. Using these APIs, an aspect can be instantiated in the Java program. Therefore, the aspect can be composed and changed by a program dynamically. This means that pointcuts and advice can be reconstructed while the target program runs.

The following fragment of a program is a sample profiling aspect described in Java with Wool APIs:

```
public class ProfileAspect extends WlAspect {
    Timer timer = new Timer();
    int count = 0;

    public void pointcut_paintcall() {
      Pointcut p
      = Pointcut.methodCall("*","Figure+","paint","*");
      associate(p, "enter", WlAdvice.BEFORE);
      associate(p, "exit", WlAdvice.AFTER);
    }

    public void enter() {
      timer.start();
      count++;
    }

    public void exit() {
      timer.stop();
    }
}
```

Here, the class `ProfileAspect` inherited from `WlAspect` is used for profiling the bottleneck of a program. In particular, the above example is specified by the `Pointcut` object for profiling the method call that belongs to the subclasses of the `Figure` class and named `paint`. A method prefixed with `pointcut_` like `pointcut_paintcall()` is called by the weaver at the time an aspect is actually woven into a program. This aspect inserts *before* advice and *after* advice to measure the elapsed time and the number of the method calls. The `enter()` method and the `exit()` method are inserted as `before` and `after` advice, respectively. Advice in Wool is associated by using the method `associate()` inherited by the class of `WlAspect`.

3.1.3.1 Aspect

The first step in the use of Wool is to create a `WlAspect` object representing an aspect defined by programmers. This step is for creating the aspect and makes it accessible from a program. In an aspect of Wool, the programmers can define it in the following two ways:

- Define the subclass of `WlAspect`, or

- Add advice to the scratch object of `WlAspect`.

The subclass of `WlAspect` represents an encapsulation of crosscutting concerns. Programmers can define aspect variables in it, which are accessed from advice or aspect methods or inter-type declarations[1] such as `timer` and `count` shown in the above example. In addition, it contains initial weaving advice described in the method `weave()`, inherited from `WlAspect`. It is called on the return from the weaver at the time an aspect is actually woven into a program. It is only by using the method `weave()` that the programmers can insert advice into a program in the subclass of `WlAspect`.

To construct and reconstruct an aspect object dynamically, Wool provides another way to create it from scratch. This feature is useful because an aspect, which is the intercepted join point identified by the pointcut or the operation defined by advice, can be formed according to the behavior of the running program. To do this, a `WlAspect` object must be created as follows:

```
WlAspect azpect = WlAspect.scratchAspect();
azpect.associate(pointcut, advice, WlAdvice.BEFORE);
```

The created object `azpect` represents an empty aspect that has no advice or inter-type declaration although the method `associate()` associates advice to the aspect later. If new advice is added to a non-empty aspect like the class `ProfileAspect`, advice inserted by using the `associate()` method is left as it is, and the new advice is just added as extra advice.

The added advice is not immediately reflected in the program. In Wool, advice is synchronized with the program only by the method `weave()` or `unweave()`. Thus, the behavior of a running program is changed only when those methods are called.

3.1.3.2 Pointcut

Wool provides several methods for identifying the set of join points by using the `Pointcut` class. The `Pointcut` class has some static methods to identify a set of join points and some methods to be used for some

[1]An *Inter-type declaration* (formerly called the introduction) is an element for AOP, which cut across classes and their hierarchies.

logical operations. For example, the method `methodCall()` identifies a call to the method with four `String` arguments. Those arguments are used for indicating a modifier, a method name, a declared class, and a signature. Table 3.2 lists several methods in `Pointcut`.

Table 3.2. Methods in `Pointcut` for identifying a set of join points.

`static Pointcut methodCall(String, String, String, String)`	identify a call to the method.
`static Pointcut methodExecute(String, String, String, String)`	identify an execution of the method.
`static Pointcut fieldGet(String, String, String)`	identify a read of the field.
`static Pointcut fieldSet(String, String, String)`	identify a write of the field.
`static Pointcut instanceCreate(String, String, String)`	identify a creation of the instance.
`static Pointcut exceptionHandle(String)`	identify a handling of the exception.
`static Pointcut within(String)`	identify any join point defined in the class.
`Pointcut and(Pointcut)`	perform an AND operation.
`Pointcut or(Pointcut)`	perform an OR operation.

3.1.3.3 Advice

Advice is represented as an ordinary method in Wool. In AOP, advice consists of a pointcut and an advice body. The *associate()* method in the

`WlAspect` class associates a `Pointcut` object with a method as advice. Thus, the advice is easily changed and modified at runtime. In addition, that method takes as a parameter a kind of advice, such as *before* and *after* advice.

The following method shows how to obtain reflective information about the current join point for the advice to use:

```
public void exit(Figure $target) {
    timer.stop();
    out.println("Painting " + $target.getName() +
                " takes " + timer.getElapsedTime() +
                "[ms]");
}
```

We can access the `this` object for the current join point by naming a method parameter `$this`. Since the hooks are specialized for the type of each joinpoint, the runtime context bound to the advice body can be minimum. This optimization is derived from the study of optimization techniques in [20] and [12]. Similar to *this()* pointcut of AspectJ, advice is not invoked if the type of a method parameter is different from the object at the identified join point. We can also use all reflective information by using `$joinpoint` as a parameter. Mainly, this object is used to obtain certain dynamic information such as the currently executing object or the target object or the arguments. The current version of Wool doesn't support obtaining more reflective information such as data structures of the class for the sake of efficiency. However, such an optimization technique as partial evaluation [58] offers the possibility of efficiently providing rich reflective information for programs, since it can statically pack that information only into the advice that requires them. Table 3.3 lists several parameter names for accessing the context of the current join point.

3.1.3.4 Inter-type Declaration

Although the limitations of the JPDA prevent Wool from implementing an inter-type declaration directly, it is easy to implement it indirectly. When a class is replaced with a new one, the JPDA restricts the new one to changing the schema like fields and the hierarchy like subclasses or

Table 3.3. Parameter names for accessing the context of current join point.

`$this`	access the `this` object in the current join point.
`$target`	access the target object in the current join point.
`$arg1, $arg2, ...`	access the argument object in the current join point.
`$joinpoint, $jp`	access the execution context in the current join point. The type of that parameter must be `Joinpoint`.

the interfaces and class modifiers and method modifiers, and to deleting methods. Thus, the inter-type declaration itself is restricted with the JPDA. However, the introduced method or field is actually referred to only from the advice code. Therefore, by adding a hidden map or a list for the inter-type declaration to all of the classes at load-time, then making the advice code use the hidden variable, Wool can allow for the addition of class elements.

3.1.4 Control of the weaver

Wool provides an optional function for programmers to control the behavior of a weaver. This function operates at the time when an aspect actually weaves the program, in other words, when the effect of an aspect appears in the running program. In AOP systems that support the dynamic weaving mechanism, the timing of the weaving is important because there is a non-determinacy when an aspect is woven from a remote JVM and there is a necessity to care for a paired advice in relation to the activation frames.

This function is implemented by delegating methods related to the weaving operation from Wool to the programmer. A programmer can control Wool by overriding the methods of `WlAspect`, specifically `hook()` and `initWeave()`. The object of `Wool` is passed to the programmer through those two methods as a parameter. Thus, by implementing the weaving operation by hand with several provided methods, the programmer can control Wool and take care of paired advice using dynamic information. Again, the programmer can select the method of hook insertion as described below in detail. Table 3.4 lists the available methods through the object of `Wool`.

Table 3.4. Available methods in `Wool` for the control of Wool.

`void advice(Joinpoint)`	execute advice associated with the join point.
`void embedHook(Joinpoint, Pointcut or String,` *optional* `boolean)`	embed hooks into the program by using dynamic code translation. Second optional parameter triggers undocking the translation thread.
`int countActivationFrame(String)`	count the number of activation frames in the context the intercepted program is running in.
`void filterClass(String, boolean)`	restrict the loaded classes to be affected by an aspect.

3.1.5 Implementation of just-in-time hook insertion

We present the implementation issues of just-in-time hook insertion by describing the details of weaving step-by-step. The order of the weaving process in Wool is:

(1) Scan classes.

(2) Insert hooks as breakpoints.

(3) The programmer selects the most suitable method.

(4)-1 Execute using the debugger, or

(4)-2 Embed the hook and call the advice.

Following are the details of each step.

3.1.5.1 Scan classes

After Wool is attached to the target program, the application threads except for the threads like the garbage collection and JIT compiler threads are suspended for a while. Wool scans all of the loaded classes and finds out the join points specified by any pointcut. The method `initWeave()` is called just before this scan. For example, if some classes are filtered by the method `filtering()` in `initWeave()` as follows:

```
public class ProfileAspect {
   public void initWeave(Wool wl)
                              throws WoolException {
      wl.filterClass("^java.*|^sun.*", false);
   }
}
```

those classes are excluded from the scanning.

3.1.5.2 Insert hooks as breakpoints

Wool sets breakpoints to specify each join point in a set of filtered classes. In order to set the breakpoint, Wool use the subclasses of `Hook` (`CallHook`, `GetHook`, etc.) included in the `wool.hook` package that is implemented using the class `BreakpointRequest` in JPDA. At the same time, pieces of any advice represented as a closure is associated with the join point through the objects of `Hook`. Finally, all of the threads that Wool has suspended are resumed.

3.1.5.3 The programmer selects the most suitable method

When any thread of the target program reaches the first join point, it is intercepted by Wool. Wool calls the method `hook()`. A programmer can avoid the executing advice that join point for the paired advice by overriding the method `hook()` in the subclass of `WlAspect`. Wool gives programmers dynamic information about the join point through the object of the functions using `Joinpoint`(`CallJoinpoint`, `GetJoinpoint`, etc.) included in the `wool.joinpoint` package, which are all implemented using the class `BreakpointEvent`. At the same time, the programmer can select whether to activate dynamic code translation by the method `embedHook()` or to execute the advice by the method `advice()`:

```
Pointcut p = Pointcut.methodCall(
              "public","FigureElement","paint","*");
public void hook(Wool wl, Joinpoint $jp)
                            throws WoolException {
  if (wl.countActivationFrame("main") > 0) {
    wl.advice($jp); // breakpoint-based execution
  } else {
    wl.embedHook($jp, p); // dynamic code translation
  }
}
```

This fragment of a program means that if there is no activation frame at the join point on the thread named `main`, the advice associated with the joinpoint `joinpoint` is activated. Otherwise, dynamic code translation is performed. The method `embedHook()` takes the object of `Pointcut` or the name of the class as a parameter.

3.1.5.4 Execute using the debugger

There are two cases when the debugger executes advice. One is that the method `hook()` is not overridden, which is the default case. The other is that the method `advice()` is called in an overriding `hook()`. Just by calling the method `advice()`, the appropriate advice associated with that join point is executed.

3.1.5.5 Embed the hook and call the advice

When the method `embedHook()` is called, Wool creates a hook for the class to be installed using Javassist [18], which is a load-time bytecode modification tool, and calls the method `redefineClass()`, which is declared in the class `VirtualMachine` in JPDA, to replace it with the new one. During the translation and replacement, the intercepted program allowed to resume execution. The advice is executed by the debugger substituting the advice as required until the replacement is completed.

Once dynamic code translation has been executed, the control of Wool will not return to the aspect program for the sake of efficiency. Not to adopt dynamic code translation or to insert hooks per thread it is better to continue breakpoint-based execution because hooks can be embedded anytime under the control of Wool.

3.1.6 Taking care of activation frames

Using just-in-time hook insertion, there is an exceptional case that we have to treat in a special way when substituting a method in which hooks are embedded. This is when the execution of some advice involves a join point contained in the method currently being executed. For example, suppose that a draw method in a rectangle class is currently being executed and the activation frame associated with that method is on the execution stack. After the class file of the rectangle class is reloaded with the hotswap mechanism, however, the execution of the draw method with that activation frame on the stack is still being performed according to the definition of the draw method given by the old class file. Thus, the hooks contained in the new class file are not effective for that execution. The hooks are effective only for the execution of the draw method started after the reloading as shown in Figure 3.4. However, the draw method might recursively call itself after the class file. To avoid this problem, dynamic code translation is automatically delayed, instead breakpoint-based execution is performed on the activation frame until the activation frame is popped from the stack.

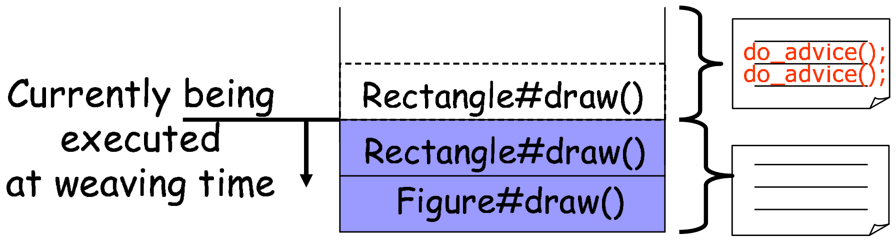


Figure 3.4. An aspect does not affect activation frames.

We also have to be careful with the execution of a pair consisting of before and after advice woven at the same join point. If that pair is woven accidentally while the method containing that join point is executed, only the after advice will be executed at the end of that execution. The before advice will not be executed since the method execution had already been started as shown in Figure 3.5. This behavior might cause a problem if the after advice depends on the results of the before advice. For example, the before advice might record the current time and the after advice can use that value to compute the elapsed time. In this case, after advice must not be executed if the corresponding before advice was not executed. To solve this problem, our technique allows the programmers to select the behavior in that case using the dynamic information at the join point.

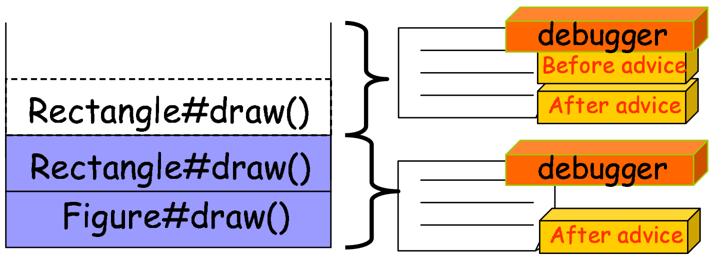


Figure 3.5. An aspect does not affect activation frames.

3.2 Related Work

First, the mechanism of our just-in-time weaver is much similar to that of the HotSpot VM [81]. Both mechanisms allow switching two kinds of the execution mode considering the whole execution cost, the HotSpot VM has the interpreter and the just-in-time compilation mode; our just-in-time weaver the debugger and the dynamic code translation mode. However, the HotSpot VM does not need to take care of the semantics of a running application such as activation frames and statically typed operations. It would not be changed before and after just-in-time compilations. On the other hand, we think the just-in-time aspect weaver should take care of activation frames and perform type-checks equal to the bytecode verifier for statically typed programs including advice code, and then provide the mechanism for it to the programmers. Wool provides it as mentioned in section 3.1.6 and section 3.1.3.

In the rest of this section, we discuss some AOP implementations related to our work, and compare them to Wool. Most current AOP implementations are based on code translation performed by a preprocessor at compile-time or by an extended classloader at load-time of the classes. Two extreme dynamic weaving mechanisms have already proposed exceptions to static code translation, where hooks consist of all-breakpoints or all-methodcalls. Both of these systems have drawbacks in their program execution performance. Wool can avoid these performance penalties by taking a suitable approach for each join point according to the programmer's specification.

An earlier version of AspectJ [48] pre-processes the source code of the aspects and produces a base Java program used to generate a pure Java program that includes woven aspects within it. Even though it only supports static weaving, AspectJ is a typical compiler-based AOP system. Since it is a static AOP system, whether to weave the advice at a join point is determined at compilation time. Also, the advice activity never changes during the runtime in AspectJ. This is sometimes a problem for faster development cycles [24] and for adaptable aspects [7, 32, 76].

Several researchers have addressed the problem of compile-time weaving by shifting the timing of aspect weaving to later stages. Approaches using bytecode-modification tools such as BCA [47] and Javassist [18] use a customized Java class loader to allow weaving at load-time. Extensions of a just-in-time (JIT) compiler like OpenJIT [66] allow weaving at the time of dynamic compilation by the JIT compiler. These are useful for faster development cycles. With these approaches, however, the chance of composition of an aspect with a program is restricted to only one time, at load-time or at dynamic compilation. In order to allow the dynamic activity of advice code, we need some tricks like runtime class evolution [34] to decompose the aspects from a program. We employed the hotswap mechanism of the JPDA for that in Wool.

PROSE [72] uses the JVM debugger interface called JPDA to insert a hook as a breakpoint, which is same as Wool when it inserts only hooks as breakpoints. They report that the execution of advice is too slow in their system to be acceptable. However, we think this approach is useful in limited cases. For example, when a system administrator must recover from system failure as soon as possible, a lightweight diagnosis aspect could be helpful. Meanwhile, when Wool inserts all of the hooks as method calls, this is the same as our previous work [21]. Our experiment has shown that dynamic code translation and class hotswapping impose heavy costs in execution time. However, dynamic compilation may amortize such costs in the long term. Although Steamloom [9] supports replacement of a loaded class with the aspect woven class by modifying the IBM Jikes RVM, it does provide selective hooks to programmers.

3.3 Experimental Results

This section first shows the result of our preliminary experiments validating the fundamental of Wool approach basing in a debug mode and combining two hooking means in Wool. After that, it reports the result of our application benchmark which compares Wool to other implementation approaches to dynamic AOP systems. We performed all the experiments on the Sun Java 2 SDK v1.4.2 HotSpot^{TM} Client VM with Linux kernel 2.4.25, which were running on a Pentium4 1.9GHz processor with 1GB memory.

3.3.1 Preliminary Experiment

3.3.1.1 Debug mode

Wool forces application programs run in the debug mode but it is not a major problem with Java 2 SDK 1.4. Although [72] reported that this overhead is too large to use the JPDA for implementing a dynamic weaving mechanism, this overhead has been significantly reduced by using Java 2 SDK 1.4. We measured the overhead incurred by a debug mode to show that Wool adopts a realistic method. Table 3.5 summarizes the relative execution time of the SPECjvm98 [79] benchmarks in the debug mode of Sun Java 2 SDK 1.4. The observed performance loss is less than 5%.

3.3.1.2 Two kinds of hooks

To demonstrate the differences of the two kinds of hooks, the breakpoint and the method call, we compared the performance of a join point hooked by a breakpoint with the same one using a method call, both using Wool. In these measurements, the join point was an empty method call, and the advice was empty. These measurements involved 10,000 iterations. The results of these micro-measurements are shown in Table 3.6.

Table 3.5. The overhead for SPECjvm98 in the debug mode of Sun Java2 SDK 1.4.

Benchmark	overhead
_200_check	103.52 %
_201_compress	99.18 %
_202_jess	104.64 %
_209_db	101.54 %
_213_javac	100.82 %
_222_mpegaudio	101.33 %

Breakpoint hooking takes approximately 700 times longer than method-call hooking on average. The elapsed time for breakpoint hooking varies widely depending on the implementation of the process scheduler used in the experimental environment because a breakpoint must be intercepted by a debugger process. Consequently, once a hook is inserted as a method call, it brings about a large performance improvement. The average time of the hook as a method call shown in Table 3.6 does not include the time elapsed during dynamic code translation in order to measure the pure elapsed time for the hook as a method call.

Table 3.6. Hooks as breakpoints and method calls in Wool.

Measurement	Average	Minimum	Maximum	Hook insertion
breakpoint	9.956[ms]	9[ms]	103[ms]	
method call	14.3[us]			435[ms]

3.3.2 Wool measurements

To demonstrate the effectiveness of the proposed just-in-time hook insertion, we compared the overhead of Wool with other techniques. We picked the `jess` benchmark program from the SPECjvm98 benchmarks and measured the execution time of the program with one of the input data called monkey banana. The `jess` benchmark is the Java Expert Shell System based on NASA's CLIPS expert shell system, which has over 10,000 lines of code and 140 classes.

We provided a before advice code which does nothing and let it woven into all the `public` method bodies in the `jess` program. The methods woven an advice code exists 163 and totally called 87,457 times. For comparison, we measured the execution time of the program with the advice woven varying the underlying systems to the one with static code translation stated in Chapter 2, the one only with dynamic code translation, the one only with breakpoint-based hooks, and Wool. For making use of Wool's hybrid approach, we implemented a simple profiler using Wool APIs as follows:

```
public void hook(Wool wl, Joinpoint $jp)
                           throws WoolException {
 wl.advice($jp);
 Class clazz =
   ((ExecutionJoinpoint)$jp).method().declaringType();
 if (map.increment(clazz) > 10)
     wl.embedHook($jp, clazz.getName());
}
```

This means that if a class is being frequently intercepted, the hooks are embedded into this class dynamically. Using this simple profiler and adjusting the threshold, the method for hook insertion was automatically and suitably selected without requiring in-depth knowledge of the application.

Table 3.7. Elapsed time [ms] of `jess`. The result in AspectJ is 114.8 ms just for reference.

	Static code trans.	Dynamic code trans.	Breakpoint-based exec.	Wool (hybrid)
pointcut	0	3,454.8	3,468.0	3,452.0
hook insertion	0	1,961.8	0	1,065.4
execution	4,488.8	580.2	140,945.0	783.8
elapsed time	4,488.8	5,995.8	144,413.0	5,301.2

Table 3.7 lists the results of the benchmark execution. The total time consists of the pointcut time (elapsed time for scanning classes), the hook insertion time (elapsed time for runtime code translation and

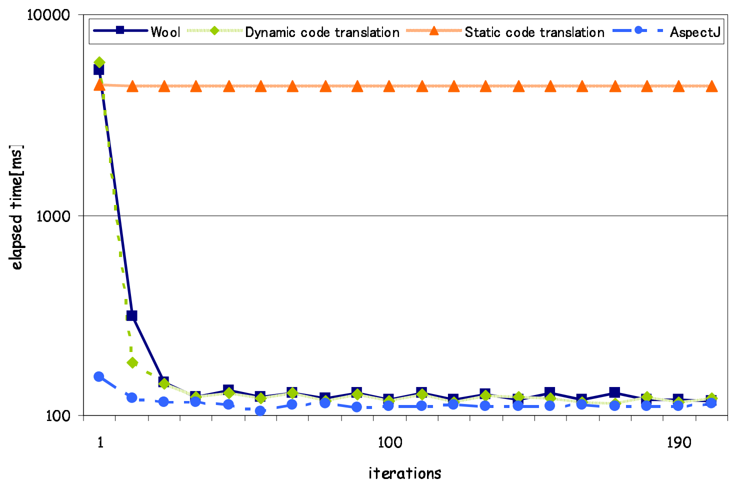


Figure 3.6. Elapsed time [ms] of jess. The results show the averages of every 10 times. The first results are the same as shown in table 3.7. Although the results in AspectJ include the time of aspect weaving (8,180 [ms]) and restarting the JVM, these are illustrated just for reference.

hotswapping) and the execution time (the rest of the elapsed time). Figure 3.6 illustrates the results of the averages on every 10 times execution. Table 3.8 lists the numbers of translated classes, inserted hooks, and executed pointcut tests, for each hook implementation approach. These results show that Wool ran long-lived dynamic AOP application as fast as AspectJ after all hooks are embedded into the application. This is because Wool avoided inserting unnecessary hooks. The static code translation inserted hooks into the program 17 times as many as Wool, and thus resulted in 12 times more pointcut tests. Thereby causes the static code translation 18% faster than Wool at the first execution, although it resulted in 38 times slower than Wool before long. In the case of the first benchmark execution, Wool was about 12% faster than dynamic code translation and about 96% faster than breakpoint-based execution. This is because Wool allowed switching the breakpoint and method call implementations at every join point. The results for dynamic code translation show that compiling extra 61 (76 - 15) classes did not improve the performance against Wool. The compilation cost 896.4 (1961.8 - 1065.4) msec. whereas the execution time was reduced by only

Table 3.8. The numbers of translated classes, inserted hooks, and pointcut test. The numbers in parenthesis represents the comparison to AspectJ.

	Static code trans.	Dynamic code trans.	Breakpoint-based exec.	Wool (hybrid)
translated classes	149 (196%)	76 (100%)	0	15 (20%)
inserted hooks	2,815 (1727%)	163 (100%)	163 (100%)	163 (100%)
execution times	1,077,338 (1231%)	87,457 (100%)	87,457 (100%)	87,457 (100%)

203.6 (783.8 - 580.2) msec. The breakpoint-based execution caused very large performance degradations because of over 87,000 context switches.

3.4 Summary

This chapter presented a new dynamic aspect weaver called Wool, which makes it possible to implement efficient, class-level, dynamic weaving mechanisms. Wool is implemented in Java without modifying the existing runtime system. It integrates a technique using breakpoints provided by the debugger interface of the JVM and a technique using the hotswap mechanism, which allows us to reload a class file that has already been loaded. This selective functionality is delegated to programmers with dynamic information about the target program. Furthermore, it provides a framework for taking care of activation frames by controlling the timing of the aspect weaving.

Our experiments showed Wool runs dynamic AOP application 38 times faster than an AOP system using static code translation approach under a certain circumstance. This is because Wool avoids inserting unnecessary hooks. Moreover, the experiment showed. Wool is about 12% faster than dynamic code translation, and about 96% faster than breakpoint-based execution. This is because Wool allows programmers to select the most suitable hooking means at each joinpoint from breakpoint or method call implementation.

Chapter 4

Loosely-separated Namespaces

This chapter introduces a novel concept for Java namespaces, called *sister namespaces* [74], to address the problem of object-based dynamic weaving techniques due to the version barrier. The version barrier is a barrier that prevents Java class loaders from being a feasible method for implementing object-level dynamic weaving mechanisms. Sister namespaces can relax the version barrier between components even if each of these components is composed with a distinct aspect. The main purpose of this chapter is to provide a mechanism for relaxing the version barrier, while still allowing type-safe instance accesses between components with negligible performance penalties in regular execution.

Practically all modern programming environments allow developers to utilize some kind of component system (e.g., JavaBeans [33], EJB [60], CORBA [67], .NET/DCOM/ActiveX [64], Eclipse plug-ins [77]). A component system allows programmers to develop a component-based application, which can be developed and then deployed per component. Most of the component systems for Java adopt a single class loader per component, and thereby create a unique namespace for each application component. A namespace is a map from the class names to the class definitions. A set of classes included in the same component *joins* [1] its

[1]A class joining a namespace means it is being loaded by the class loader that

own namespace and thus naming conflicts between components can be avoided. Moreover, a component can be dynamically and individually updated without restarting the whole execution environment.

One significant drawback of such component systems for Java is the difficulty for components to communicate across class loader boundaries in the Java Virtual Machine (JVM) [40, 53, 30, 41]. In fact, such communication is well known to frequently cause a cast error `ClassCastException` or a link error `LinkageError`. Most of the link errors are just bugs; an error is caused when a class is wrongly loaded by both parent and child loaders [45]. These bugs can be easily avoided if developers are careful. However, cast errors are extremely difficult to avoid since this problem is caused by the strict separation of namespaces, ironically called the *version barrier*. The version barrier is a mechanism that prevents a version of a class type from being converted to another version of that specific class type. For instance, it restricts an instance of the former type to be assigned to the variable of the latter type.

In Java, a class type is uniquely identified at runtime by the combination of a class loader and a fully qualified class name. If two class definitions with the same class name are loaded by different loaders, two versions of that class type are created and they can co-exists, although they are regarded as distinct types. This mechanism allows Java applets, Java Servlets, and EJBs working on the same process of the JVM without naming conflicts 4.1. The version barrier is a mechanism for guaranteeing that different versions of a class are regarded as different types. This guarantee is significant for performance reasons. If different versions of a class were not regarded as different types, the advantages of being a statically typed language would be lost. Moreover, if the same class definition (i.e., class file) is loaded by different class loaders, different versions of that class are created and regarded as distinct types. Therefore, if two components load the same class file individually, one component cannot pass an instance of that class type to the other.

This chapter presents our novel concept of namespaces in Java, which we call *sister namespaces*, and the design of that mechanism. Sister namespaces can relax the version barrier between application components. An instance can be carried beyond the version barrier between sister namespaces if the type of that instance is compatible between these namespaces. The mechanism of sister namespaces is implemented by ex-

creates the namespace.

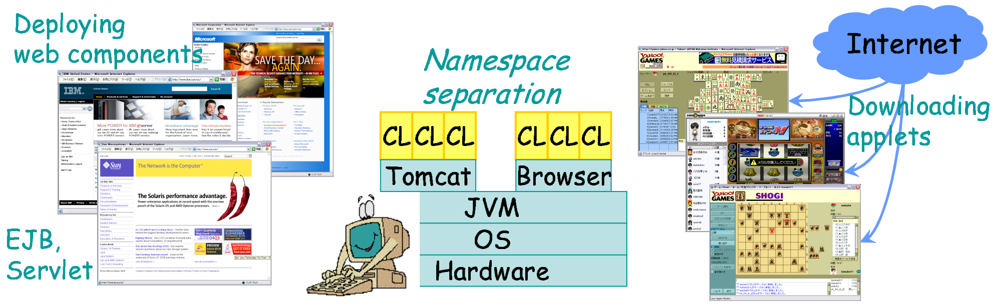


Figure 4.1. Downloaded applets, deployed servlets and EJBs can work on the same process of the JVM without violating with each other.

tending the type checker and the class loader of the JVM.

The rest of this chapter is organized as follows. Section 4.1 presents the design and implementation of the sister namespace. Section 4.2 discusses a few implementation issues. Section 4.4 presents the results of our experiments. Section 4.5 compares the sister namespace mechanism to other related work. Section 4.6 concludes this chapter.

4.1 Sister Namespaces

We propose *sister namespaces*, which can relax the version barrier between namespaces. Different versions of a class type that join sister namespaces can be assignment compatible with each other if these versions have differences while still preserving the *version compatibility*. Our challenge is to relax the version barrier while keeping type-safe instance accesses efficient. In this section, we first define extended assignment compatibility, which is based on Java binary compatibility [29] (Section 4.1.1). Next, we show the sister-supported type checker, which takes the central role in relaxing the version barrier for sister namespaces (Section 4.1.3). The type checker blocks illegal objects when they move across the version barrier, and thus no subsequent extra check is needed for these objects. This is enabled because it is prohibited for a namespace to become a sister of its parent or child namespace. In addition, we present the sister loader constraint (Section 4.1.4) and then the schema class loading scheme (Section 4.1.5). They prevent eager class loading and type inconsistencies, respectively.

We implemented the sister namespaces on the IBM Jikes Research Virtual Machine (RVM) [5]. The extensions to the Jikes RVM are only the sister-namespace API, a sister-supported class loader, and a sister-supported type checker. The Java API is provided as an extension to the existing `java.lang.ClassLoader` in the GNU Classpath libraries. These extensions consist of several core classes of the Jikes RVM such as class and object representations.

4.1.1 Version Compatibility

This section provides the definition of version compatibility, which securely extends the assignment compatibility between different versions of a class type. We define two versions of a class type, C_{ver1} and C_{ver2}, as assignment compatible with each other if C_{ver1} is version compatible with C_{ver2} and vice versa. A class type C_{ver2} is version compatible with C_{ver1} if all the class types that could previously link with C_{ver1} and work with an instance of C_{ver1} without errors are able to also correctly work with instances of C_{ver2} without other assignment compatibility rules such as a subtyping relation. Thus, if C_{ver1} and C_{ver2} are version compatible, then an instance of C_{ver1} can be securely converted to the type C_{ver2} when it is assigned to a variable of C_{ver2} and vice versa. Here, being secure means that every operation on C_{ver2} is applicable to the instance of C_{ver1} without errors; any method call, field access, or type casting applied to the variable does not fail.

Figure 4.1 shows the differences that programmers are permitted to make between two versions of a class while preserving version compatibility between two versions. The following summarize the differences:

- Differences of declared static members such as a static field, a static method, a constructor, or an initializer.
- Differences of the implementation of instance members, such as an instance method.

The differences are derived from the study of the binary compatible changes mentioned in the Java language specification [37].

Version compatibility is based on the idea of binary compatibility; it means that an instance rather than a class can work with the binary of another version of the class type. Java binary compatibility defines a set of changes that developers are permitted to make to a package or to a class

Table 4.1. A set of version compatible changes derives from two-way binary compatibility

changes	class	interface	field
abstract	×	n/a	n/a
final	×	n/a	×
static	n/a	n/a	×
			(○ private)
transient	n/a	n/a	○
synchronized	n/a	n/a	n/a
accessibility	×	×	×
adding/removing	n/a	n/a	×
including static			(○ private)
body	n/a	n/a	n/a

	method	constructor	initializer
abstract	×	n/a	n/a
final	×	n/a	n/a
static	×	n/a	○
	(○ private)		
transient	n/a	n/a	n/a
synchronized	○	n/a	n/a
accessibility	×	×	n/a
adding/removing	×	×	○
including static	(○ private,native)	(○ private)	
body	○	○	○

or interface type while preserving the compatibility with the preexisting binaries (Table 4.2). A change to a class type is binary compatible with preexisting binaries if preexisting binaries that previously linked without errors will continue to link without recompiling. Version compatibility defines differences between two versions of a class type that preserve the binary compatible property between an instance of one version and the binary of the other version. Unlike the original binary compatibility, the version compatibility allows any change to static members since static member accesses are irrelevant to instances. Version compatibility deals with the compatibility between an instance and the binary of another version of that class type. Therefore, to be version compatible, two versions of a class type must have the same set of private members, although the implementations of those members may differ. This is a difference from the binary compatibility, which allows the two versions to have a different set of private members. Since a private member can be accessed from not only `this` instance but also from other instances of another version of that class type, version compatibility requires that the two versions have the same set of private members.

Table 4.2. Java binary compatibility

changing to	class	interface	field	method	constructor	initializer
abstract	×[1]	n/a	n/a	×[2]	n/a	n/a
non-abstract	○	n/a	n/a	○	n/a	n/a
final	×[3]	n/a	×[4]	×[3]	n/a	n/a
non-final	○	n/a	○	○	n/a	n/a
static, non-static	n/a	n/a	×[5]		n/a	○
transient, non-transient	n/a	n/a	○	n/a	n/a	n/a
synchronized, non-synchronized	n/a	n/a	n/a	○	n/a	n/a
strengthening accessibility.	×[6]					n/a
weakening accessibility.	○	○	○	○	○	n/a
addition, or override,overload, changing throws clause.	n/a	n/a	○	○	○	○
removal, or changing parameters, returntypes.	n/a	n/a	×[7]	×[8]		○
method body	n/a	n/a	n/a	○	○	○
changing type hierarchy.	Reordring type hierarchy with no member lost.[9]					

4.1.2 Creating Sister Namespaces

A sister namespace is a first-class entity, but it is created implicitly when a class loader is instantiated with a class loader given as a parameter. The `ClassLoader` class provides the new constructor as follows:

```
protected ClassLoader(ClassLoader parent,
                      ClassLoader sister)
```

The class loader obtained from this constructor becomes a *sister class loader* of the class loader specified by the parameter `sister`. The latter class loader also becomes a sister of the former one. These two sister class loaders construct their own sister namespaces; the version barrier between them is relaxed if the version compatibility is satisfied. The sister class loaders must not have a parent-child relationship. This rule is significant for the efficient type checking we describe later. If the sister class loaders have such a relationship, the construction of the sister namespaces fails. In this chapter, if a version of a class type is loaded earlier (or later) than other versions, it is called a *younger* (or *older*) sister version of that class type. This young-old relationship is independent of the creation order of the sister class loaders.

In the case described in Section 2.2.2.3, the application programmers can exchange instances between two namespaces for WAR1 and WAR2 if they are sister namespaces. Each namespace can contain a different version of the type of the exchanged instance. WAR1 and WAR2 must be loaded by the class loaders created as follows:

```
ClassLoader ear = new EARClassLoader();
ClassLoader war1 = new WARClassLoader(ear);
```

[1]InstantiationError

[2]AbstractMethodError

[3]VerifyError

[4]IllegalAccessError

[5]IncompatibleClassChangeError, but ○ if the member is private.

[6]IllegalAccessError, but ○ if the class is loaded by the bootstrap or the system class loader.

[7]NoSuchFieldError, but ○ if the field is private.

[8]NoSuchMethodError, but ○ if the field is private or native.

[9]VerifyError, if some of the members are lost.

```
ClassLoader war2 = new WARClassLoader(ear, war1);
```

The `ear`, `war1`, and `war2` are instances of the `ClassLoader` class. The third `new` operation creates sister namespaces for WAR1 and WAR2. Both `war1` and `war2` have the same parent class loader `ear`. In general, application programmers of components, such as Applets, Servlets, Eclipse plug-ins, and EJB, do not have to be aware of namespaces or class loaders. These are implicitly managed by the application middleware. Creating sister namespaces by using the `ClassLoader` constructor above is the work of middleware developers. A sister namespace can make another plain namespace its sister on demand. Since the sister relationship is transitive, if a namespace becomes a sister of a namespace and then it becomes a sister of another namespace, all three namespaces become sisters of each other. Programmers can incrementally create a new namespace and make it another sister of the other sister namespaces. This feature would be useful in cases of incremental development processes and routine maintenance work.

Note that all class types *defined* by a sister class loader can be version compatible with the corresponding sister version of that class type, even if the loading of these class types are *initiated* by the child class loaders. An *initiating* class loader, which initiates the loading of a class type, does not have to actually load a class file. Instead, it can delegate to the parent class loader. The class loader that actually loads a class file and defines that type is called a *defining* class loader of that type. This delegation mechanism is used for sharing the same version of class type between the initiating and defining class loaders. In Figure 4.2, if two sister namespaces are created between class loaders L3 and L3', the classes F and H can be version compatible with F' and H', respectively. The pairs E and E' or G and G' are not compatible with each other since they are defined by other class loaders.

4.1.3 Sister-supported Type Checking

The version barrier is relaxed by a type checker that considers the sister namespaces. In Java programs, most bytecode instructions such as the method invocation instructions `invokevirtual` and `invokenonvirtual`, and field access instructions such as `getfield` and `putfield` are statically typed. These instructions do not perform dynamic type checking.

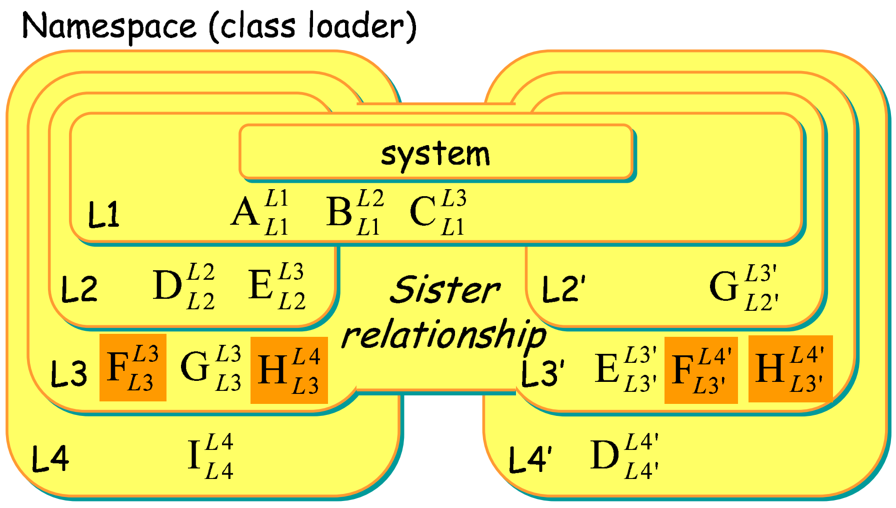

Namespace (class loader)
system
L1 A_{L1}^{L1} B_{L1}^{L2} C_{L1}^{L3}
L2 D_{L2}^{L2} E_{L2}^{L3}
Sister relationship
L2' $G_{L2'}^{L3'}$
L3 F_{L3}^{L3} G_{L3}^{L3} H_{L3}^{L4}
L3' $E_{L3'}^{L3'}$ $F_{L3'}^{L4'}$ $H_{L3'}^{L4'}$
L4 I_{L4}^{L4}
L4' $D_{L4'}^{L4'}$

Figure 4.2. The notation $C_{L_d}^{L_i}$ represents a class type, where C denotes the name of the class, L_d denotes the class's defining loader, and L_i denotes the loader initiated class loading. An inclusion relation represents a parent-child relationship. For example, the class loader L1 is a parent of both L2 and L2'. And the classes A, B, C, and the system classes are visible in the namespaces L2, L2', L3, L3', L4, and L4'. In this figure, the sister namespace L3 and L3' have a sister relationship.

Therefore, these instructions *as they are* can work correctly with any version of class type if they are version compatible. On the other hand, several instructions such as `instanceof`, `checkcast`, `invokeinterface`, `athrow`, and `aastore` entail dynamic type checking. The type checking by those instructions must be enhanced if the version barrier is relaxed so that version compatible instances can be passed between sister namespaces. The algorithm of enhanced type checking for sister namespaces is shown in Figure 4.3. After the regular type checks are performed, and if they fail (line 2), the extra checks are executed (lines 3–6). First, a sister relationship is examined (line 3). If the left-hand side class type (`LHS`) and the right-hand side class type (`RHS`) have a sister relationship, then the type checker determines whether one class type has undergone the schema compatible loading process against the other type (line 4). Schema compatible loading is introduced later.

Note that these extra checks for sister namespaces are executed only after the regular type checks fail. Since typical programs do not frequently cause type errors, this enhancement for the built-in type checker implies no performance penalties as long as instances are not passed between sister namespaces.

```
1: if LHS is a subtype of RHS then true
2: else if LHS is not a subtype of RHS then
3:     if LHS is a sister type of RHS &&
4:           LHS is version compatible with RHS
then true
5:     else false
6:     end
7: end
```

Figure 4.3. Pseudo code for enhanced type checking for sister namespaces. A type check is the determination of whether a value of one type, hereafter the right-hand side (RHS) type , can legally be converted to a variable of a second type, hereafter the left-hand side (LHS) type. If so, the RHS type is said to be a subtype of the LHS type and the LHS type is said to be a supertype of the RHS type.

The sister-supported type checker only prohibits a version incompatible instance from being passed between sister namespaces. A version incompatible class can join each of the sister namespaces if an instance of that version stays within the namespace. To avoid the security problem described in Section 2.2.2.3 (naively relaxing the assignment compatibility), sister namespaces must detect a version incompatible instance being passed between sister namespaces. This detection is executed by only the `checkcast` instruction. In other words, the detection is not executed by other instructions for method invocation, field access, and assignment. This is mainly due to the design of sister namespaces, which must not have a parent-child relationship between them. This rule brings the *bridge-safety* [91] property to all classes included in the sister namespaces. This property guarantees that an instance of a class type is always examined by the `checkcast` instruction when it is passed between sister namespaces. It must be first upcast to a type loaded by the common parent class loader of the two sister class loaders, and then it must be downcast before it is assigned to the class type loaded by the sister class loader at the destination. For example, when an instance of `Cart` is passed, it will be first upcast to a super class of `Cart`, such as the `Object` class, and then downcast to another version of the `Cart` class (Figure 4.4). Therefore, the `checkcast` instruction is always executed when the instance is downcast to `Cart`.

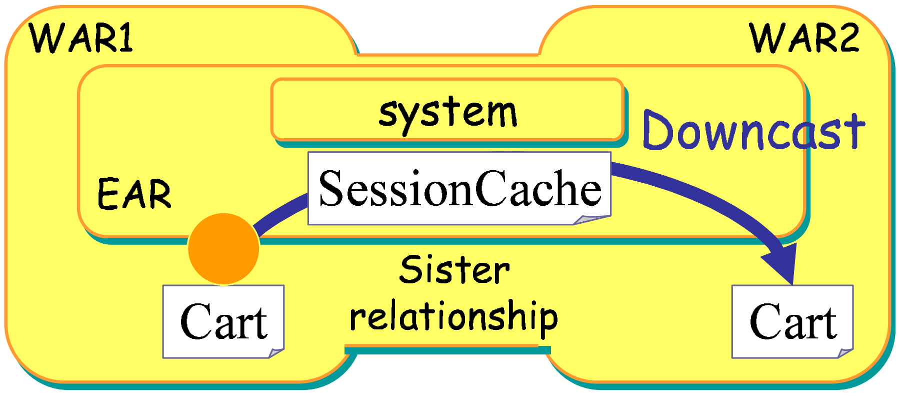


Figure 4.4. Downcast enforced by the bridge-safety property satisfied between namespaces.

To implement the sister-supported type check, we modified the `VM_DynamicTypeCheck` class in the Jikes RVM. We extended that class and the TIB (Type Information Block) for fast type checking to consider sister relationships. The original TIB holds several arrays of type identifiers. For example, the arrays of extended superclass types and of implemented interface types are stored in the TIB for fast type checking without looking up the whole type hierarchy [10][11]. Similarly, the extended TIB holds two arrays of `sid`s. The `sid` is the identifier of a sister relationship. The two arrays are of the `sid`s of the extended superclasses and the `sid`s of the implemented interfaces. The `sid` of a class can be obtained from a `VM_Class` object representing that class. We extended the `VM_Class` class to hold the `sid` of the class.

4.1.4 Sister Loader Constraint

A straightforward implementation of the sister-supported type checker requires eager class loading. Even if the sister-supported type checker verifies that the type of an instance is version compatible, that instance cannot be fully *trusted*. The instance may contain a version incompatible instance as a field value or return it as a result of a method execution. That is, the *untrusted* instance may *relay* an incompatible instance. Since an instance is type checked only when it is downcast, the types of the instance that may be relayed must also be type checked at the same time.

Therefore, the type checker verifies all the class types occurring in the class definition of that instance, such as parameter types[2], return types, and field types. It also recursively verifies the class types occurring in the definitions of those types. However, if this recursive type check is naively implemented, all the related classes would have to be eagerly loaded. This eager loading is practically unacceptable, since the advantages of the dynamic features of Java would be lost. The sister-supported type checker must be able to work with the scheme of lazy class loading. Note that the original class loading mechanism of Java is based on lazy class loading.

To examine version compatibility while enabling lazy class loading, the JVM maintains a set of *sister loader constraints*, which are dynamically updated when the sister-supported type checker works. If the type checker finds a class type that must be verified but has not been loaded yet, the JVM does not eagerly load that class; instead, it records a sister loader constraint. For example, if the type checker attempts to verify that a version of class C is version compatible with another version C', but C or C' has not been loaded yet, the JVM records as a constraint that C must be version compatible with C'. This constraint is later verified when C or C' is loaded. If the type checker detects that this constraint is not satisfied, it throws a `LinkageError`. While the type checker is verifying that constraint, if it finds another class type that must be verified but is not loaded, a new sister loader constraint is recorded. If the type checker finds a class type that must be verified and has been already loaded, it recursively verifies that class type at the same time. Note that every constraint is verified only once. The result of the verification is recorded to avoid further verification.

In summary, the JVM needs to maintain the invariant: *Each class type co-existing in the namespace satisfies all the sister loader constraints.* The invariant is maintained as follows:

Every time a new class joins a sister namespace, the JVM verifies whether that class type will violate an existing sister loader constraint.

If the class type being loaded violates an existing sister loader constraint, loading that class type fails since that class type is un-

[2]If a parameter type is not version compatible, an incompatible instance may be sent back without type checking to the namespace that has sent the instance of the class type including that parameter type.

trusted in the namespace. If there is no constraint referring to that class type, the JVM loads that class type, although that class type might be version incompatible. It is verified later when a new constraint referring to that class type is recorded.

Every time a new sister loader constraint is recorded, the JVM verifies whether that constraint is satisfied with the class types that have been already loaded.

If a class type that has already been loaded does not satisfy a newly recorded constraint, loading the class type that starts the type checking process producing the new constraint is untrusted and hence the loading is aborted. If any class types needed for verifying that constraint have not been loaded, the verification is postponed until those classes are loaded. Otherwise, if all the class types needed for the verification have been loaded and the constraint is successfully verified, the constraint is removed from the record.

For efficient verification of constraints, we added an array of flags to `VM_Class`. Each flag indicates whether the version of the class type represented by a `VM_Class` object has been recursively type checked with another sister version. The flag is true only if the two versions of the class type are version compatible and if the type checker has verified that those two versions never *relay* a version incompatible instance. Since there might be multiple sisters, the `VM_Class` object holds an array of the flags, each of which indicates the result of the type check with each sister version. The JVM uses these flags for executing a recursive type check only once.

4.1.5 Schema Compatible Loading

Even if two versions of a class type satisfy the version compatibility, these instances may have schema incompatibility. This means that the layout of the internal type information blocks (TIBs) may not be identical between the two versions of the class type. The TIB holds fields and function pointers to a corresponding method body. The order of the TIB entries depends on the JVM or compilers; it does not depend on the order of the member declarations in a source file or a class file. Thus, even if two versions of a class type have version compatibility, the layout of the TIBs may not be identical.

The sister-supported class loader guarantees that layouts of the TIBs are identical between two versions of a class type if the class types are version compatible (Figure 4.5). Since the JVM uses a constant index into the TIB when it accesses a field or a method, the JVM cannot correctly execute the bytecode if the layouts of the TIBs are not identical between compatible versions of the class type. Therefore, when the class loader loads a younger version of a class type, the JVM constructs the TIB of that version of the class so that the layout of the TIB is identical to that of the TIB of an older version of the class. This loading process is called *schema compatible loading*. Note that this process is given up against the incompatible class type that has no binary compatibility with the older sister version of that class type. This result is employed by the JVM to quickly examine whether a class type is trusted or not.

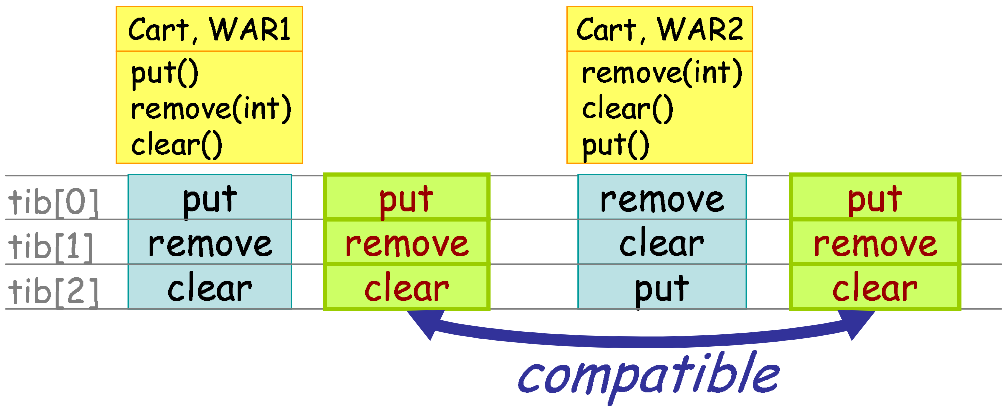


Figure 4.5. Because of Schema compatible class loading, both of the sister version of a class have the same layout of the TIB.

In the Jikes RVM, a TIB is constructed during the execution of the `resolve` method in `VM_Class`. The `resolve` method is invoked during the class resolution process by the `VM_ClassLoader` class, an instance of which represents a class loader. The `resolve` method has been extended to perform the schema compatible loading.

4.2 Discussion

4.2.1 Canceling JIT Compilations

Just-In-Time (JIT) compiled code sometimes needs to be canceled since a devirtualized method call does not correctly refer to a method declared in a sister version of the class type. Recent optimizing JIT compilers [42, 81] perform the devirtualization optimization that transforms not only a final and a static method but also a virtual method call to a static method. For a given virtual call, the compiler determines whether or not the call can be devirtualized by analyzing the current class hierarchy. If the method can be devirtualized and its code size is small enough, the compiler inlines the method. Therefore, if the type of an instance is converted to a sister version of that class type, the JVM would continue to invoke the original inlined code instead of the real method of that instance. This is because the JIT compiler does not consider sister namespaces; method bodies might be different among sister versions of the same class type.

To avoid this problem, the JIT compiler must cancel devirtualization when a new sister version of a class type is loaded. Fortunately, most optimizing JIT compilers have an efficient cancellation mechanism for dynamic class loading. Since a whole class hierarchy cannot be statically determined in Java, JIT compilers can dynamically replace [81] or rewrite [42] inlined code. This is performed when a new subclass is loaded and the subclass overrides a method that has not been overridden by the other subclasses. The JIT compiler that supports sister namespaces also cancels devirtualization when version compatibility is verified and a new sister version of a class type is available.

4.2.2 Eager Notifications of Version Incompatibility

Version incompatibility checks between sister versions of a class type may eagerly throw a cast error before any incompatible class types actually co-exist in one namespace. For example, if the type checker detects that a class type may relay a version incompatible instance, it throws a cast error. This eager notification strategy is similar to the loader constraint scheme [51]. The JVM prohibits different versions of a class type from even being loaded if the JVM encounters an operation that relays instances from one namespace to the other. If the JVM has already loaded these versions of the class, the JVM throws a link error. However, except for the loader constraint scheme, verification of compatibility and error notification are not performed at loading time but are done later by the linker, while the linker resolves the constant pool items (e.g., `NoSuchMethodError`, `IllegalAccessError`, `IncompatibleClasChangeError`). If a Java program includes binary incompatibility, it continues to run until it actually executes an illegal operation caused by that incompatibility. This lazy verification and notification are useful in practice.

However, we have adopted the eager strategy for avoiding performance penalties due to version compatibility checks. To delay notifications of incompatibility as long as the program continues to run without errors, a number of guard tests must be embedded into the incompatible class. In Java, once the JVM executes a method invocation or a field access, the operation is linked with the call site and replaced with efficient code that does not perform type checking anymore. Therefore, the guard tests must be embedded for verifying version compatibility after the version incompatibility is found. It requires the JIT compiler to recompile the code that refers the incompatible class type. Moreover, the guard tests imply the non-negligible performance overhead mentioned in Section 2.2.2.3; thus, we do not delay the notifications of incompatibility.

4.3 An Abstract Model of the Sister Namespace

This section extends a model of Java dynamic linking and verification [27]. We consider language $\mathcal{P}$, which stands for loaded, verified and prepared programs. We describe execution in terms of expressions e, states σ, loaded code P[3], version compatible sisters V, a set of trusted sisters D, and a set of trusted but not loaded sisters C. It has the general form:

$$\mathsf{e}, \sigma, \mathsf{PP'}, \mathsf{V}, \mathsf{C}, \mathsf{D} \rightsquigarrow \mathsf{e'}, \sigma', \mathsf{PP'}, \mathsf{VV'}, \mathsf{CC'}, \mathsf{CD'}$$

thus describing that the expression may be rewritten, the state may be modified, code may be loaded, the version compatible sisters may be added, and some of the set of loaded and non-loaded sisters may be trusted – the terms $\mathsf{PP'}$, $\mathsf{VV'}$, $\mathsf{CC'}$, $\mathsf{DD'}$ indicate concatenation of P, V,C, D.

[3]We do not describe verification and preparation of the loaded code, and thus the loading process includes verification and preparation in our model.

4.3.1 The languages $\mathcal{P}$

The language $\mathcal{P}$ presents an abstract view of the Java bytecode. For simplicity, we only consider classes, subclasses, assignment, method overloading and inheritance, field inheritance and hiding. All methods have one argument – multiple arguments can be encoded through objects.

Expressions Figure 4.6 containes the syntax of expressions in $\mathcal{P}$ program. Field acceses and instance or class method calls, corresponding to bytecode instructions such as getfield, putfield and invokevirtual, has the form $\mathsf{e}_1[\mathsf{t}_1,\mathsf{t}_2].\mathsf{f}$, where t_1 is the class containing the field definition, and t_2 the type of that field. Instance method calls have the form $\mathsf{e}_1[\mathsf{t}_1,\mathsf{t}_2,\mathsf{t}_3].\mathsf{m}(\mathsf{e}_2)$, where t_1 is the class containing the method definition, t_2 is the type of the method's argument, and t_3 is the result type.

The only types we consider are classes and `int`. We do not deal with subtyping introduced through interfaces. Values are either integers, or addresses of objects. Addresses are represented by positive numbers and are denoted by $\alpha, \alpha' etc$; the null pointer is denoted by 0. Values, whether they stand for addresses or for integers, are denoted by $\beta, \beta' etc.$

Language for loaded, verified and prepared code, $\mathcal{P}$ We describe the program $\mathcal{P}$ through functions that lookup the superclasses, fields and methods of a class.

Definition 1 A tuple ($\mathcal{P}, \mathcal{T}, \mathcal{M}_{ff}, \mathcal{F}_{ff}, \mathcal{M}_e$, $\mathsf{P}_\emptyset$) is a language for loaded, verified and prepared code, iff

- *$\mathcal{P}$ is a set.*
- *$\mathcal{T}$ is a function,* $\mathcal{T}$: $\mathsf{Id} \times \mathcal{P} \longrightarrow \mathsf{Id} \cup \{\epsilon\}$.
- *$\mathcal{M}$ is a function,* $\mathcal{M}$: $\mathsf{Id} \times \mathsf{Id} \times \mathsf{Typ} \times \mathsf{Typ} \times \mathcal{P} \longrightarrow \mathsf{Expr} \cup \{\epsilon\}$.
- *$\mathcal{F}_{ff}$ is a function,* $\mathcal{F}_{ff}$: $\mathsf{Id} \times \mathsf{Id} \times \mathsf{Typ} \times \mathcal{P} \longrightarrow \mathsf{Offs} \cup \mathsf{ErrOffs}$.
- *$\mathcal{M}_{ff}$ is a function,* $\mathcal{M}_{ff}$: $\mathsf{Id} \times \mathsf{Id} \times \mathsf{Typ} \times \mathsf{Typ} \times \mathcal{P} \longrightarrow \mathsf{Offs} \cup \mathsf{ErrOffs}$.

$e \in \text{Expr}$	$::=$	e[t,t,t].m(e)	method call
	$\mid$	$v = e$	assignment
	$\mid$	`new` c	object creation
	$\mid$	`this`	receiver
	$\mid$	v	variable
	$\mid$	β	integer value
	$\mid$	$(t)\ e$	cast
	$\mid$	NllPErr $\mid$ LoadErr	null-pointer err., load err.
	$\mid$	IncVerErr	incompatible class err.
	$\mid$	CasErr $\mid$ VerCasErr	cast err., version cast err.
v	$::=$	$e[t,t].f$	field access
	$\mid$	z	parameter
$t \in \text{Typ}$	$::=$	$c \mid$ `int`	class, integer
$\phi \in \text{Offs}$	$::=$	$1 \mid 2 \mid \ldots$	offsets
$\chi \in \text{ErrOffs}$	$::=$	-1	member undefined
	$\mid$	-2	type of wrong kind
	$\mid$	-3	type undefined
$\alpha \in \text{Addr}$	$::=$	$0 \mid \phi$	address
$\beta \in \text{Val}$	$::=$	$\alpha \mid -1 \mid -2 \mid \ldots$	value
$c \in \text{Id}$			c class names
$m, f, z \in \text{Id}$			m method names, f field names

Figure 4.6. The syntax of expressions

- $\mathcal{M}_e$ *is a function,* $\mathcal{M}_e : \mathsf{Offs} \times \mathsf{Id} \times \mathcal{P} \longrightarrow \mathsf{Expr} \cup \{\epsilon\}$.

- $\mathsf{P}_\emptyset \in \mathcal{P}$, $\forall \mathsf{t} \in \mathsf{Id} : \mathcal{T}\ (\mathsf{t},\mathsf{P}_\emptyset) = \epsilon$.

- *for any* P_1, $\mathsf{P}_2 \in \mathcal{P}$, their concatenation, $\mathsf{P}_1\mathsf{P}_2$, gives a further element of $\mathcal{P}$, with:
  - $\mathcal{T}(\mathsf{t},\mathsf{P}_1\mathsf{P}_2) = \mathcal{T}(\mathsf{t},\mathsf{P}_1)$ *if* $\mathcal{T}(\mathsf{t},\mathsf{P}_1) \neq \epsilon$, $\mathcal{T}(\mathsf{t},\mathsf{P}_2)$ *otherwise.*
  - $\mathcal{M}(\mathsf{m},\mathsf{c},\mathsf{t}_2,\mathsf{t}_3,\mathsf{P}_1\mathsf{P}_2) = \mathcal{M}(\mathsf{m},\mathsf{c},\mathsf{t}_2,\mathsf{t}_3,\mathsf{P}_1)$ *if* $\mathcal{T}(\mathsf{c},\mathsf{P}_1) \neq \epsilon$, $\mathcal{M}(\mathsf{m},\mathsf{c},\mathsf{t}_2,\mathsf{t}_3,\mathsf{P}_2)$ *otherwise.*
  - $\mathcal{F}_{ff}(\mathsf{f},\mathsf{c},\mathsf{t},\mathsf{P}_1\mathsf{P}_2) = \mathcal{F}_{ff}(\mathsf{f},\mathsf{c},\mathsf{t},\mathsf{P}_1)$ *if* $\mathcal{T}(\mathsf{c},\mathsf{P}_1) \neq \epsilon$, $\mathcal{F}_{ff}(\mathsf{f},\mathsf{c},\mathsf{t},\mathsf{P}_2)$ *otherwise.*
  - $\mathcal{M}_{ff}(\mathsf{m},\mathsf{c},\mathsf{t}_2,\mathsf{t}_3,\mathsf{P}_1\mathsf{P}_2) = \mathcal{M}_{ff}(\mathsf{m},\mathsf{c},\mathsf{t}_2,\mathsf{t}_3,\mathsf{P}_1)$ *if* $\mathcal{T}(\mathsf{c},\mathsf{P}_1) \neq \epsilon$, $\mathcal{M}_{ff}(\mathsf{m},\mathsf{c},\mathsf{t}_2,\mathsf{t}_3,\mathsf{P}_2)$ *otherwise.*

– $\mathcal{M}_e(\phi,\mathsf{c},\mathsf{P}_1\mathsf{P}_2) = \mathcal{M}_e(\phi,\mathsf{c},\mathsf{P}_1)$ *if* $\mathcal{T}(\mathsf{c},\mathsf{P}_1) \neq \epsilon$,
$\mathcal{M}_e(\phi,\mathsf{c},\mathsf{P}_2)$ *otherwise.*

$\mathsf{P}_\emptyset$ indicates the empty program in $\mathcal{P}$, and ϵ indicates lookup of a non-existing entity. $\mathcal{T}(\mathsf{t},\mathsf{P})$ is intended to return the direct superclass of t, if t is declared as a clsas in P. $\mathcal{M}(\mathsf{m},\mathsf{c},\mathsf{t}_2,\mathsf{t}_1,\mathsf{P})$ is intended to return the body of method m defined in class c, with result type t_1 and argument type t_2, or ϵ if no such method is found. $\mathcal{F}_{ff}(\mathsf{f},\mathsf{c},\mathsf{t},\mathsf{P})$ and $\mathcal{M}_{ff}(\mathsf{m},\mathsf{c},\mathsf{t}_2,\mathsf{t}_1,\mathsf{P})$ are intended to return the offset of field f defined in class sf c with type t, or the offset of method m defined in class c with argument type t_2 and return type t_1. $\mathcal{M}_e(\phi,\mathsf{c},\mathsf{P})$ looks up the method body in class c using offset ϕ. Note, that $\mathcal{M}(\mathsf{m},\mathsf{c},\mathsf{t}_2,\mathsf{t}_1,\mathsf{P})$ does not contain ($\not\supset$) $\mathcal{M}(\mathsf{m},\mathsf{c}',\mathsf{t}_2,\mathsf{t}_1,\mathsf{P})$, $\mathcal{F}_{ff}(\mathsf{f},\mathsf{c},\mathsf{t},\mathsf{P}) \not\supset \mathcal{F}_{ff}(\mathsf{f},\mathsf{c}',\mathsf{t},\mathsf{P})$, $\mathcal{M}_{ff}(\mathsf{m},\mathsf{c},\mathsf{t}_2,\mathsf{t}_1,\mathsf{P}) \not\supset \mathcal{M}_{ff}(\mathsf{m},\mathsf{c}',\mathsf{t}_2,\mathsf{t}_1,\mathsf{P})$, $\mathcal{M}_e(\phi,\mathsf{c},\mathsf{P}) \not\supset \mathcal{M}_e(\phi,\mathsf{c}',\mathsf{P})$, if $\mathcal{T}(\mathsf{c},\mathsf{P})=\mathsf{c}'$.

We define functions to collect all fields and methods in $\mathcal{P}$.

Definition 2 *For* $\mathsf{P}\in\mathcal{P}$, $\mathsf{c}\in\mathsf{Id}$ *with* $\mathcal{T}(\mathsf{c},\mathsf{P})=\mathsf{c}'$, *we define*:

- $\mathcal{T}_s(\mathsf{P}) = \{\mathsf{t} \mid \mathcal{T}(\mathsf{t},\mathsf{P}) \neq \epsilon\}$.
- $\mathcal{S}_s(\mathsf{c},\mathsf{P}) = \mathsf{c}',\mathcal{S}_s(\mathsf{c}',\mathsf{P})$.
- $\mathcal{F}_{sig}(\mathsf{c},\mathsf{P}) = \{(\mathsf{t}\ \mathsf{f}) \mid \mathcal{F}_{ff}(\mathsf{f},\mathsf{c},\mathsf{t},\mathsf{P})>0\}$.
- $\mathcal{F}_{lay}(\mathsf{c},\mathsf{P}) = \{(\mathsf{t}\ \mathsf{f}\ \phi) \mid \mathcal{F}_{ff}(\mathsf{f},\mathsf{c},\mathsf{t},\mathsf{P})=\phi\}$.
- $\mathcal{M}_{sig}(\mathsf{c},\mathsf{P}) = \{(\mathsf{m}\ \mathsf{t}_1\ \mathsf{t}_2\ \mathsf{e}) \mid \mathsf{e} = \mathcal{M}(\mathsf{m},\mathsf{c},\mathsf{t}_1,\mathsf{t}_2,\mathsf{P}) \neq \epsilon\}$.
- $\mathcal{M}_{lay}(\mathsf{c},\mathsf{P}) = \{(\mathsf{m}\ \mathsf{t}_1\ \mathsf{t}_2\ \phi) \mid \mathcal{M}_{ff}(\mathsf{m},\mathsf{c},\mathsf{t}_1,\mathsf{t}_2,\mathsf{P})=\phi\}$.

$\mathcal{T}_s(\mathsf{P})$ is intended to return the all classes loaded, verified and prepared in P. $\mathcal{S}_s(\mathsf{c},\mathsf{P})$ is intended to return the all super classes of class c as a list. $\mathcal{F}_{sig}(\mathsf{c},\mathsf{P})$ and $\mathcal{M}_{sig}(\mathsf{c},\mathsf{P})$ are intended to return a set of the tuple of field f defined in class c with type t, or a set of the tuple of method m defined in class c with argument type t_1, return type t_2 and body e. $\mathcal{F}_{lay}(\mathsf{c},\mathsf{P})$ and $\mathcal{M}_{lay}(\mathsf{c},\mathsf{P})$ are intended to return a set of the tuple of field f defined in class c with type t and offset ϕ, or a set of the tuple of method m defined in class c with argument type t_1, return type t_2 and offset ϕ.

Lemma 1 *For programs* L, L_1, L_2,$\in \mathcal{P}$:

- $\mathcal{T}_s(\mathsf{P}_1) \cap \mathcal{T}_s(\mathsf{P}_2) = \emptyset \Rightarrow \mathsf{P}_1\mathsf{P}_2 = \mathsf{P}_2\mathsf{P}_1$
- $\mathcal{T}_s(\mathsf{P}_2) \subseteq \mathcal{T}_s(\mathsf{P}_1) = \emptyset \Rightarrow \mathsf{P}_1\mathsf{P}_2 = \mathsf{P}_1$

4.3.2 Execution

This section describes the execution model defined in terms of a rewriting relationship, consisting of expression e, store σ, loaded code P, version compatible sisters V, a set of trusted sisters D, and a set of trusted but not loaded sisters C. Thus, execution has the form $\mathsf{e}, \sigma, \mathsf{PP}', \mathsf{V}, \mathsf{C}, \mathsf{D} \rightsquigarrow \mathsf{e}', \sigma', \mathsf{PP}', \mathsf{VV}', \mathsf{CC}', \mathsf{CD}'$. In order to define the concept of a sister relationship, version compatibility and a trusted relationship between class types, we describe some judgements and functions:

- $\mathsf{P} \vdash \mathsf{c} \leq_{clss} \mathsf{c}'$
  – c is a subclass of c' in context of P.
- $\vdash \mathsf{P} \diamond_a$
  – The subclass relationship in P is not cyclic.
- $\vdash \mathsf{P} \diamond_{sups}$
  – P contain all supertypes of types defined in P.
- $\mathcal{S} \subset \mathsf{Id} \times \mathsf{Id}$
  – t_1 S t_2 means t_1 and t_2 have a sister relationship with each other.
- $\mathsf{t}_1 \overset{vc}{=} \mathsf{t}_2$
  – t_1 and t_2 are version compatible with each other, and thus, the layout of both classes are identical.
- $\mathsf{P},\mathsf{V},\mathsf{C},\mathsf{D} \vdash \mathsf{t}_1 \underset{tr}{\sim} \mathsf{t}_2 \rightharpoonup \mathsf{C}',\mathsf{D}'$
  – t_1 and t_2 are trusted with each other in context $\mathsf{P},\mathsf{V},\mathsf{C},\mathsf{D}$, that is, both classes never return instances of a version incompatible class.

So as to give a more concise description of the rewrite semantics, and also, to distinguish between routine rewrite rules, and those particular to Java implementation, we introduce three kinds of contexts in Figure 4.7. Expression contexts, $\sqsubset\ .\ \sqsupset^{exp}$, are filled with a sub-expression; their execution propagates execution to this sub-expression, as in rule PROPAGATE. Null-contexts, $\sqsubset . \sqsupset^{nll}$, when filled with 0, raise an exception when executed as in rule NULL POINTOR. Type contexts, $\sqsubset . \sqsupset^{typ}$, are filled with a type name; their execution causes the type to be loaded, verified and prepared if the type is not part of the loaded code, as in rules LOAD, LOAD ERROR and LOADING VIOLATION.

$\sqsubset . \sqsupset^{exp}$	$::=$	$\sqsubset . \sqsupset [t,t,t].m(e)$	the receiver of a method call
	$\mid$	$\alpha[t,t,t].m(\sqsubset . \sqsupset)$	the argument of a method call
	$\mid$	$\sqsubset . \sqsupset [t,t].f$	the receiver of a field access
	$\mid$	$\sqsubset . \sqsupset = e$	the left-hand side variable
	$\mid$	$v = \sqsubset . \sqsupset$	the right-hand side variable
	$\mid$	$(t) \sqsubset . \sqsupset$	the casting variable
$\sqsubset . \sqsupset^{nll}$	$::=$	$\sqsubset . \sqsupset [t,t,t].m(e)$	the receiver of a method call
	$\mid$	$\sqsubset . \sqsupset [t,t].f$	the receiver of a field read
	$\mid$	$\sqsubset . \sqsupset [t,t].f = \beta$	the receiver of a field write
$\sqsubset . \sqsupset^{typ}$	$::=$	$\alpha[\sqsubset . \sqsupset, t, t].m(\beta)$	the type of a calling method
	$\mid$	$\alpha[\sqsubset . \sqsupset, t].f$	the type of a accessing field
	$\mid$	$\texttt{new} \sqsubset . \sqsupset$	the type of a creating instance
	$\mid$	$(\sqsubset . \sqsupset)\alpha$	the type of a casting instance

Figure 4.7. Contexts

We now study the five components of execution. Note, that the five components are disjoint, if a rule from one component is applicable, then no rule from another component is applicable.

4.3.2.1 Evaluation

Evaluation is the part of execution, and it comprises:

PROPAGATE – Propagate execution at the receiver and then the argument of a method call, at the receiver of a field access and to the left hand and right hand sides of an assignment.

$$\frac{\mathrm{e}, \sigma, \mathrm{P}, \mathrm{V}, \mathrm{C}, \mathrm{D} \rightsquigarrow \mathrm{e}, \sigma', \mathrm{P}', \mathrm{V}', \mathrm{C}', \mathrm{D}'}{[\mathrm{e}]^{exp}, \sigma, \mathrm{P}, \mathrm{V}, \mathrm{C}, \mathrm{D} \rightsquigarrow [\mathrm{e}']^{exp}, \sigma', \mathrm{P}', \mathrm{V}', \mathrm{C}', \mathrm{D}'}$$

VARIABLE ACCESS – Accessing variable or addresses.

$$\frac{\text{z a variable}}{\mathrm{z}, \sigma, \mathrm{P}, \mathrm{V}, \mathrm{C}, \mathrm{D} \rightsquigarrow \sigma(\mathrm{z}), \sigma, \mathrm{P}, \mathrm{V}, \mathrm{C}, \mathrm{D}}$$

Variable Assignment – Assigning to variables.

$$\frac{}{\mathrm{z} = \beta, \sigma, \mathrm{P}, \mathrm{V}, \mathrm{C}, \mathrm{D} \rightsquigarrow \beta, \sigma[\mathrm{z} \mapsto \beta], \mathrm{P}, \mathrm{V}, \mathrm{C}, \mathrm{D}}$$

Null Pointer – Throwing the NllPErr exception when attempting to call a method, access field, or assign to field of 0.

$$\frac{}{[0]^{nll}, \sigma, \mathrm{P}, \mathrm{V}, \mathrm{C}, \mathrm{D} \rightsquigarrow \mathrm{NllPErr}, \sigma, \mathrm{P}, \mathrm{V}, \mathrm{C}, \mathrm{D}}$$

New – Creating new objects of already loaded class c (c $\in \mathcal{T}_s$(P)), initializing the declared fields with 0.

$$\frac{\begin{array}{l} \mathrm{c} \in \mathcal{T}s(\mathrm{P}) \\ \alpha \text{ new in } \sigma \\ \mathcal{F}_{lay}(\mathrm{c}, \mathrm{P}) = \{(\mathrm{t}_1\ \mathrm{f}_1\ \phi_1), \ldots, (\mathrm{t}_\mathrm{n}\ \mathrm{f}_\mathrm{n}\ \phi_\mathrm{n})\} \\ \sigma' = \sigma[\alpha \mapsto \mathrm{c}, \alpha + \phi_1 \mapsto 0, \ldots, \alpha + \phi_\mathrm{n} \mapsto 0] \end{array}}{\text{new c}, \sigma, \mathrm{P}, \mathrm{V}, \mathrm{C}, \mathrm{D} \rightsquigarrow \alpha, \sigma', \mathrm{P}, \mathrm{V}, \mathrm{C}, \mathrm{D}}$$

4.3.2.2 Resolution

Resolution describes the process of resolving references to fields or methods. It corresponds to the bytecode instructions such as getfield, putfield, invokevirtual.

Field Access – Field access has the form $\alpha[t_1, t_2].f$. The offset of that field is determined using $\mathcal{F}_{ff}$(f,t_1,t_2,P), and if found ($\mathcal{F}_{ff}$(f,t_1,t_2,P)=ϕ), then it is used to calculate the address of that field ($\alpha + \phi$). – here the actual class of the receiver is used ($\sigma(\alpha)$)

$$\frac{\mathcal{F}_{ff}(\mathrm{f}, \mathrm{t}_1, \mathrm{t}_2, \mathrm{P}) = \phi}{\begin{array}{l} \alpha[\mathrm{t}_1, \mathrm{t}_2].\mathrm{f}, \sigma, \mathrm{P}, \mathrm{V}, \mathrm{C}, \mathrm{D} \rightsquigarrow \sigma(\alpha + \phi), \sigma, \mathrm{P}, \mathrm{V}, \mathrm{C}, \mathrm{D} \\ \alpha[\mathrm{t}_1, \mathrm{t}_2].\mathrm{f} = \beta, \sigma, \mathrm{P}, \mathrm{V}, \mathrm{C}, \mathrm{D} \rightsquigarrow \beta, \sigma[\alpha + \phi \mapsto \beta], \mathrm{P}, \mathrm{V}, \mathrm{C}, \mathrm{D} \end{array}}$$

Method Call – Method call has the form $\alpha[t_1, t_2, t_3].m(\beta)$. The offset is determined using $\mathcal{M}_{ff}$(m,t_1,t_2,t_3,P), and if found

($\mathcal{M}_{ff}$(m,t_1,t_2,t_3,P)=ϕ), then it is used to select the method body from the lookup table of the class of α through $\mathcal{M}_e$(ϕ,$\sigma(\alpha)$,P). – here the actual class of the receiver is used ($\sigma(\alpha)$).

$$\frac{\begin{array}{l}\mathcal{M}_{ff}(\mathrm{m}, \mathrm{t}_1, \mathrm{t}_2, \mathrm{t}_3, \mathrm{P}) = \phi \\ \mathcal{M}_e(\phi, \sigma(\alpha), \mathrm{P}) = \mathrm{e} \\ \mathrm{y}_1, \mathrm{y}_2 \text{ are fresh variables in } \sigma \\ \mathrm{e}' = \mathrm{e}[\mathrm{y}_1/\mathrm{x}, \mathrm{y}_2/\mathrm{this}] \\ \sigma' = \sigma[\mathrm{y}_1 \mapsto \beta, \mathrm{y}_2 \mapsto \alpha]\end{array}}{\alpha[\mathrm{t}_1, \mathrm{t}_2, \mathrm{t}_3].\mathrm{m}(\beta), \sigma, \mathrm{P}, \mathrm{V}, \mathrm{C}, \mathrm{D} \leadsto \mathrm{e}', \sigma', \mathrm{P}, \mathrm{V}, \mathrm{C}, \mathrm{D}}$$

CAST – Cast has the form (t)α. The apparent type of the casting object changes to the target type t, if it is already loaded and the actual type of the casting object ($\sigma(\alpha)$) is a subclass of the target type t.

$$\frac{\begin{array}{l}\mathrm{t} \in \mathcal{T}s(\mathrm{P}) \\ \mathrm{P} \vdash \sigma(\alpha) \leq \mathrm{t}\end{array}}{(\mathrm{t})\alpha, \sigma, \mathrm{P}, \mathrm{V}, \mathrm{C}, \mathrm{D} \leadsto \alpha, \sigma, \mathrm{P}, \mathrm{V}, \mathrm{C}, \mathrm{D}}$$

CAST ERROR – Throwing the CasErr exception if the actual type of the casting object ($\sigma(\alpha)$) is not a subclass of the target type t and has no sister class.

$$\frac{\begin{array}{l}\mathrm{t} \in \mathcal{T}s(\mathrm{P}) \\ \mathrm{P} \nvdash \sigma(\alpha) \leq \mathrm{t} \\ \mathrm{t} \not\mathcal{S}\ \mathrm{t}'\end{array}}{(\mathrm{t})\alpha, \sigma, \mathrm{P}, \mathrm{V}, \mathrm{C}, \mathrm{D} \leadsto \mathrm{CasErr}, \sigma, \mathrm{P}, \mathrm{V}, \mathrm{C}, \mathrm{D}}$$

VERSION CAST – This rule describes the version checking situations when CAST and CAST ERROR rule are not applicable: if the actual type of the casting object ($\sigma(\alpha)$) is a subclass of the target's sister class t' and these sister classes are trusted with each other.

$$\frac{\begin{array}{l}\mathrm{t} \in \mathcal{T}s(\mathrm{P}) \\ \mathrm{P} \nvdash \sigma(\alpha) \leq \mathrm{t} \\ \mathrm{t}\ \mathcal{S}\ \mathrm{t}' \\ \sigma(\alpha) \leq \mathrm{t}' \\ \mathrm{P}, \mathrm{V}, \mathrm{C}, \mathrm{D} \vdash \mathrm{t} \underset{\mathrm{tr}}{\sim} \mathrm{t}' \rightharpoonup \mathrm{C}', \mathrm{D}'\end{array}}{(\mathrm{t})\alpha, \sigma, \mathrm{P}, \mathrm{V}, \mathrm{C}, \mathrm{D} \leadsto \alpha, \sigma, \mathrm{P}, \mathrm{V}, \mathrm{C}', \mathrm{D}'}$$

Version Cast Violation – Throwing the VerCasErr exception if the actual type of the casting object ($\sigma(\alpha)$) is a subclass of the target's sister class t' but these sister classes are not trusted with each other.

$$\frac{\begin{array}{l} \mathrm{t} \in \mathcal{T}s(\mathrm{P}) \\ \mathrm{P} \nvdash \sigma(\alpha) \leq \mathrm{t} \\ \mathrm{t}\ \mathcal{S}\ \mathrm{t}' \\ \sigma(\alpha) \leq \mathrm{t}' \\ \mathrm{P}, \mathrm{V}, \mathrm{C}, \mathrm{D} \nvdash \mathrm{t} \underset{\mathrm{tr}}{\sim} \mathrm{t}' \rightharpoonup \mathrm{C}', \mathrm{D}' \end{array}}{(\mathrm{t})\alpha, \sigma, \mathrm{P}, \mathrm{V}, \mathrm{C}, \mathrm{D} \leadsto \mathrm{VerCasErr}, \sigma, \mathrm{P}, \mathrm{V}, \mathrm{C}', \mathrm{D}'}$$

4.3.2.3 Loading

Load – Loading is required when a type context, $\sqsubset t \sqsupset^{typ}$, is executed for a type t which has not been loaded yet. That is, when a new object of class t is create, or a when a field of class t is accessed, or when a method from class t is called. For the sake of simplicity, the loading process includes verification and preparation of loaded classes. If loading is successful (P'=$load$(t, P) $\neq$ P$_\emptyset$), then execution continues with the loaded code augmented by P'. A load function $load$(t, P) returns class definitions for t and all its superclasses except for those already defined in P, provided that no class circularity was encountered; otherwise it returns P$_\emptyset$. A set of version compatible sister classes in the program including the newly loaded classes is recorded (V' = {(c$_1$, c$_2$) | c$_1 \in \mathcal{T}_s$(PP'), c$_2 \in \mathcal{T}_s$(P'), c$_1$ $\mathcal{S}$ c$_2$, c$_1 \overset{vc}{=}$ c$_2$ }). Note, that the version incompatible classes are just recorded, then verified while determining the trusted relationship. Then, the sister loader constraints for the loaded sister classes are determined (P,V,C$_i$,D$_i$ $\vdash$ c$_{i1} \underset{tr}{\sim}$ c$_{i2} \rightharpoonup$ C$_{i+1}$,D$_{i+1}$).

$$\frac{\begin{array}{l} \mathrm{e} = \sqsubset \mathrm{t} \sqsupset^{typ} \\ \mathrm{t} \notin \mathcal{T}_s(\mathrm{P}), \ \vdash \mathrm{P}\diamond_{\mathrm{sups}} \\ \mathsf{P'}{=}load(\mathsf{t}, \mathsf{P}) \neq \mathsf{P}_\emptyset \\ \mathrm{V}' = \{(\mathrm{c}_1, \mathrm{c}_2) | \mathrm{c}_1 \in \mathcal{T}_\mathrm{s}(\mathrm{PP}'), \mathrm{c}_2 \in \mathcal{T}_\mathrm{s}(\mathrm{P}'), \mathrm{c}_1\ \mathcal{S}\ \mathrm{c}_2, \mathrm{c}_1 \overset{\mathrm{vc}}{=} \mathrm{c}_2\} \\ \{(\mathrm{c}_{11}, \mathrm{c}_{12}), \ldots, (\mathrm{c}_{\mathrm{n}1}, \mathrm{c}_{\mathrm{n}2})\} = \mathrm{C} \cap \{\mathcal{T}\mathrm{s}(\mathrm{PP}') \times \mathcal{T}\mathrm{s}(\mathrm{PP}')\}^4 \\ \mathrm{P}, \mathrm{V}, \mathrm{C}_\mathrm{i}, \mathrm{D}_\mathrm{i} \vdash \mathrm{c}_{\mathrm{i}1} \underset{\mathrm{tr}}{\sim} \mathrm{c}_{\mathrm{i}2} \rightharpoonup \mathrm{C}_{\mathrm{i}+1}, \mathrm{D}_{\mathrm{i}+1} \ \ \forall \mathrm{i} \in 1 \ldots \mathrm{n} \end{array}}{\mathrm{e}, \sigma, \mathrm{P}, \mathrm{V}, \mathrm{C}_1, \mathrm{D}_1 \leadsto \mathrm{e}, \sigma, \mathrm{PP}', \mathrm{VV}', \mathrm{C}_{\mathrm{n}+1}, \mathrm{D}_{\mathrm{n}+1}}$$

Determining version compatibility, through the judgement $\mathsf{t}_1 \overset{vc}{=} \mathsf{t}_2$, and a trusted relationship, through the judgement $\mathsf{P,V,C,D} \vdash \mathsf{t}_1 \underset{tr}{\sim} \mathsf{t}_2 \rightharpoonup \mathsf{C',D'}$ are described below:

$$(1)\quad \frac{}{\mathrm{t} \overset{\mathrm{vc}}{=} \mathrm{t}}$$

$$(2)^5\quad \frac{\begin{array}{l} \mathcal{T}(\mathrm{c}_1, \mathrm{P}) = \mathrm{c}'_1 \quad \mathcal{T}(\mathrm{c}_2, \mathrm{P}) = \mathrm{c}'_2 \\ \mathrm{c}'_1 \overset{\mathrm{vc}}{=} \mathrm{c}'_2 \\ \mathcal{F}_{lay}(\mathrm{c}_1, \mathrm{P}) = \{(\mathrm{t}_1\ \mathrm{f}_1\ \phi_1), \ldots, (\mathrm{t}_\mathrm{n}\ \mathrm{f}_\mathrm{n}\ \phi_\mathrm{n})\} \\ \mathcal{F}_{lay}(\mathrm{c}_2, \mathrm{P}) = \{(\mathrm{t}'_1\ \mathrm{f}_1\ \phi_1), \ldots, (\mathrm{t}_\mathrm{n}\ \mathrm{f}_\mathrm{n}\ \phi_\mathrm{n})\} \\ \mathcal{M}_{lay}(\mathrm{c}_1, \mathrm{P}) = \{(\mathrm{t}_{11}\ \mathrm{t}_{12}\ \mathrm{m}_1\ \phi_1), \ldots, (\mathrm{t}_{\mathrm{m}1}\ \mathrm{t}_{\mathrm{m}2}\ \mathrm{m}_\mathrm{m}\ \phi_\mathrm{m})\} \\ \mathcal{M}_{lay}(\mathrm{c}_2, \mathrm{P}) = \{(\mathrm{t}'_{11}\ \mathrm{t}'_{12}\ \mathrm{m}_1\ \phi_1), \ldots, (\mathrm{t}'_{\mathrm{m}1}\ \mathrm{t}'_{\mathrm{m}2}\ \mathrm{m}_\mathrm{m}\ \phi_\mathrm{m})\} \end{array}}{\mathrm{c}_1 \overset{\mathrm{vc}}{=} \mathrm{c}_2}$$

$$(3)\quad \frac{}{\mathrm{P, V, C, D} \vdash \mathrm{t} \underset{\mathrm{tr}}{\sim} \mathrm{t} \rightharpoonup \mathrm{C, D}}$$

$$(4)\quad \frac{(\mathrm{c}_1, \mathrm{c}_2) \notin \mathcal{T}_\mathrm{s}(\mathrm{PP}') \times \mathcal{T}_\mathrm{s}(\mathrm{PP}')}{\mathrm{P, V, C, D} \vdash \mathrm{c}_1 \underset{\mathrm{tr}}{\sim} \mathrm{c}_2 \rightharpoonup \mathrm{C} \cup \{(\mathrm{c}_1, \mathrm{c}_2)\}, \mathrm{D}}$$

$$(5)\quad \frac{(\mathrm{c}_1, \mathrm{c}_2) \in \mathrm{D}}{\mathrm{P, V, C, D} \vdash \mathrm{c}_1 \underset{\mathrm{tr}}{\sim} \mathrm{c}_2 \rightharpoonup \mathrm{C, D}}$$

$$(6)\quad \frac{\begin{array}{l} (\mathrm{c}_1, \mathrm{c}_2) \in (\{\mathcal{T}_\mathrm{s}(\mathrm{PP}') \times \mathcal{T}_\mathrm{s}(\mathrm{PP}')\} \backslash \mathrm{D}) \cap \mathrm{V} \\ \mathrm{C}_1 = \mathrm{C} \backslash \{(\mathrm{c}_1, \mathrm{c}_2)\}, \mathrm{D}_1 = \mathrm{D} \cup \{(\mathrm{c}_1, \mathrm{c}_2)\} \\ \mathcal{S}_s(\mathrm{c}_1, \mathrm{P}) = \mathrm{t}_1, \ldots, \mathrm{t}_\mathrm{p} \quad \mathcal{S}_s(\mathrm{c}_2, \mathrm{P}) = \mathrm{t}'_1, \ldots, \mathrm{t}'_\mathrm{p} \\ \mathrm{P, V, C_i, D_i} \vdash \mathrm{t}_\mathrm{i} \underset{\mathrm{tr}}{\sim} \mathrm{t}'_\mathrm{i} \rightharpoonup \mathrm{C}_{\mathrm{i}+1}, \mathrm{D}_{\mathrm{i}+1} \quad \forall \mathrm{i} \in 1 \ldots \mathrm{p} \\ \mathcal{F}_{sig}(\mathrm{c}_1, \mathrm{P}) = \{(\mathrm{t}_1\ \mathrm{f}_1), \ldots, (\mathrm{t}_\mathrm{q}\ \mathrm{f}_\mathrm{q})\} \quad \mathcal{F}_{sig}(\mathrm{c}_2, \mathrm{P}) = \{(\mathrm{t}'_1\ \mathrm{f}_1), \ldots, (\mathrm{t}'_\mathrm{q}\ \mathrm{f}_\mathrm{q})\} \\ \mathrm{P, V, C_{p+j}, D_{p+j}} \vdash \mathrm{t}_\mathrm{j} \underset{\mathrm{tr}}{\sim} \mathrm{t}'_\mathrm{j} \rightharpoonup \mathrm{C}_{\mathrm{p+j+1}}, \mathrm{D}_{\mathrm{p+j+1}} \quad \forall \mathrm{j} \in 1 \ldots \mathrm{q} \\ \mathcal{M}_{sig}(\mathrm{c}_1, \mathrm{P}) = \{(\mathrm{t}_{11}\ \mathrm{t}_{12}\ \mathrm{m}_1), \ldots, (\mathrm{t}_{\mathrm{r}1}\ \mathrm{t}_{\mathrm{r}2}\ \mathrm{m}_\mathrm{k})\} \\ \mathcal{M}_{sig}(\mathrm{c}_2, \mathrm{P}) = \{(\mathrm{t}'_{11}\ \mathrm{t}'_{12}\ \mathrm{m}_1), \ldots, (\mathrm{t}'_{\mathrm{r}1}\ \mathrm{t}'_{\mathrm{r}2}\ \mathrm{m}_\mathrm{k})\} \\ \mathrm{P, V, C_{p+q+k}, D_{p+q+k}} \vdash \mathrm{t}_{\mathrm{k}1} \underset{\mathrm{tr}}{\sim} \mathrm{t}'_{\mathrm{k}1} \rightharpoonup \mathrm{C}_{\mathrm{p+q+k+1}}, \mathrm{D}_{\mathrm{p+q+k+1}} \quad \forall \mathrm{k} \in 1 \ldots \mathrm{r} \\ \mathrm{P, V, C_{p+q+r+h}, D_{p+q+r+h}} \vdash \mathrm{t}_{\mathrm{h}2} \underset{\mathrm{tr}}{\sim} \mathrm{t}'_{\mathrm{h}2} \rightharpoonup \mathrm{C}_{\mathrm{p+q+r+h+1}}, \mathrm{D}_{\mathrm{p+q+r+h+1}} \quad \forall \mathrm{h} \in 1 \ldots \mathrm{s} \end{array}}{\mathrm{P, V, C, D} \vdash \mathrm{c}_1 \underset{\mathrm{tr}}{\sim} \mathrm{c}_2 \rightharpoonup \mathrm{C}_{\mathrm{p+q+r+s+1}}, \mathrm{D}_{\mathrm{p+q+r+s+1}}}$$

[1]Actually $\{(c_{11}, c_{12}), \ldots, (c_{n1}, c_{n2})\} = \{(c_1, c_2) | c_1\ \mathcal{S}\ c_2\} \cap \mathrm{C} \cap (\mathcal{T}s(\mathrm{PP}') \times \mathcal{T}s(\mathrm{PP}'))$ but $C \subset \{(c_1, c_2) | c_1\ \mathcal{S}\ c_2\}$.

[5]The model in this section does not support overloading. All class members are uniquely identified only by their names.

The rule (2) describes version compatibility between classes. Classes c_1 and c_2 are version compatible if the both superclasses are version compatible, and then those class layouts are compatible with each other. It means that schema compatible class loading is performed against both c_1 and c_2. The constraint for c_1 and c_2 are recorded when either or both of the classes are not loaded as in rule (4), and if both have been loaded, those classes are trusted with each other without recording the constraint as in rule (5). The rule (6) describes that the verification of the trusted relationship for c_1 and c_2 propagates over all types of the superclasses, fields, method arguments, and method results. When either of those classes has not been loaded, the constraint of that pair is recorded, if they are not version compatible with each other, they can trust each other.

Loading determines the object layout (TIB) including method lookup tables. We describe the requirements of the *loading* function as following definition, *i.e., load*(t,P) is a *loading function* if (2a) it allocates distinct offsets to fields, (2b) preserves field offsets from superclasses, (2c) preserves method offsets from superclasses, (2d) all valid offsets lead to a method body either defined for that class in P, or inherited from a superclass.

Definition 3 *A function load* : $\mathsf{Id}\times\mathcal{P} \longrightarrow \mathcal{P}$ *iff:*
$\vdash \mathsf{P}\diamond_a$, $load(\mathsf{c,P}) = \mathsf{P'} \neq \mathsf{P}_\emptyset \Rightarrow$

1. $\mathsf{c} \in \mathcal{T}_s(\mathsf{P'}) \setminus \mathcal{T}_s(\mathsf{P})$
2. $\mathcal{T}(\mathsf{c,P}) = \mathsf{c'} \Rightarrow$
   $\forall$ f,f',t,t',t_1,t_2,m :
   (a) $\mathcal{F}_{ff}(\mathsf{f,c,t,P'}) = \mathcal{F}_{ff}(\mathsf{f',c,t',P'}) > 0 \Rightarrow \mathsf{f{=}f',t{=}t'}$.
   (b) $\mathcal{F}_{ff}(\mathsf{f,c',t,PP'}) > 0 \Rightarrow \mathcal{F}_{ff}(\mathsf{f,c',t,PP'}) = \mathcal{F}_{ff}(\mathsf{f,c,t,P'})$.
   (c) $\mathcal{M}_{ff}(\mathsf{m,c',t_2,t_1,PP'}) > 0 \Rightarrow \mathcal{M}_{ff}(\mathsf{m,c',t_2,t_1,PP'}) = \mathcal{M}_{ff}(\mathsf{m,c,t_2,t_1,P'})$.
   (d) $\mathcal{M}_{ff}(\mathsf{m,c,t_2,t_1,P'}) = \phi \Rightarrow \mathcal{M}_e(\phi,\mathsf{c,P'}) \neq \epsilon$, *or* $\mathcal{M}_{ff}(\mathsf{m,c',t_2,t_1,PP'}) = \phi$.

LOAD ERROR – Throwing the LoadErr exception if loading is not successful.

$$\frac{\mathrm{t} \notin \mathcal{T}s(\mathrm{P})}{\sqsubset \mathrm{t} \sqsupset^{typ}, \sigma, \mathrm{P}, \mathrm{V}, \mathrm{C}, \mathrm{D} \rightsquigarrow \mathrm{LoadErr}, \sigma, \mathrm{P}, \mathrm{V}, \mathrm{C}, \mathrm{D}}$$

Loading Violation – Throwing the IncVerErr exception if the sister loader constraints for the loaded sister classes are not satisfied, that is, the sister classes are not trusted with each other.

$$\frac{\begin{array}{l}\mathrm{t} \notin \mathcal{T}_s(\mathrm{P}),\ \vdash \mathrm{P}\diamond_{\mathrm{sups}} \\ \mathsf{P}'{=}load(\mathsf{t},\mathsf{P}) \neq \mathsf{P}_\emptyset \\ \{(\mathrm{c_{11}}, \mathrm{c_{12}}), \ldots, (\mathrm{c_{n1}}, \mathrm{c_{n2}})\} = \mathrm{C} \cap \{\mathcal{T}\mathrm{s}(\mathrm{PP}') \times \mathcal{T}\mathrm{s}(\mathrm{PP}')\} \\ \neg\exists \mathrm{C_{i+1}}, \mathrm{D_{i+1}}.(\mathrm{P}, \mathrm{V}, \mathrm{C_i}, \mathrm{D_i} \vdash \mathrm{c_{i1}} \underset{\mathrm{tr}}{\sim} \mathrm{c_{i2}} \rightharpoonup \mathrm{C_{i+1}}, \mathrm{D_{i+1}})\end{array}}{\sqsubset \mathrm{t} \sqsupset^{\mathrm{typ}}, \sigma, \mathrm{P}, \mathrm{V}, \mathrm{C_1}, \mathrm{D_1} \rightsquigarrow \mathrm{IncVerErr}, \sigma, \mathrm{P}, \mathrm{V}, \mathrm{C_{n+1}}, \mathrm{D_{n+1}}}$$

4.4 Experimental Results

This section reports the results of our performance measurements. We performed all the experiments on the IBM Jikes Research Virtual Machine 2.3.2 with Linux kernel 2.4.25, which were running on a Pentium4 1.9GHz processor with 1GB memory. Both the Jikes RVM and our modified RVM were compiled to use the baseline compiler for building the boot image with the semi-space garbage collector.

4.4.0.4 Baseline performance

To measure the baseline performance, we ran the SPECjvm98 [79] benchmarks on both our JVM and the unmodified JVM. The problem sizes of all the benchmarks were 100 (maximum). Table 4.3 lists the results. The numbers are the average execution time for 20 repetitions. The baseline overhead due to the sister namespace was negligible.

4.4.0.5 Cost of loading classes into sister namespaces

We measured the time for loading classes with a plain class loader or a sister class loader. This experiment shows the performance penalty incurred by the sister class loader, which executes schema compatible loading and verifies the version compatibility between classes. We took nine application programs, listed in Table 4.4, to measure the total loading time. The loading process includes delegating to the parent class

Table 4.3. SPECjvm98 benchmark results on both our JVM and the unmodified JVM.

Benchmark Program	*Jikes RVM (JRVM)*	*Sister-supported JRVM (SVM)*	*SVM /JRVM*
_201_compress	47.293 ms	46.218 ms	97.7%
_202_jess	40.258 ms	38.726 ms	96.2%
_205_raytrace	22.704 ms	23.404 ms	103.1%
_209_db	65.628 ms	67.075 ms	102.2%
_213_javac	54.698 ms	57.759 ms	105.6%
_222_mpegaudio	29.344 ms	29.210 ms	99.5%
_227_mtrt	25.812 ms	24.563 ms	95.2%
_228_jack	28.372 ms	28.047 ms	98.9%
Total	314.109 ms	315.002 ms	100.3%

loader, searching for a class file in a specified classpath, and resolving, initializing, and instantiating that class type in the JVM. All loading processes are iterated 20 times. The results show that the performance penalty varied among those applications from around 14% to 67%. The penalties mostly depended on the number of declared methods and fields. Thus, the largest application showed the largest overhead.

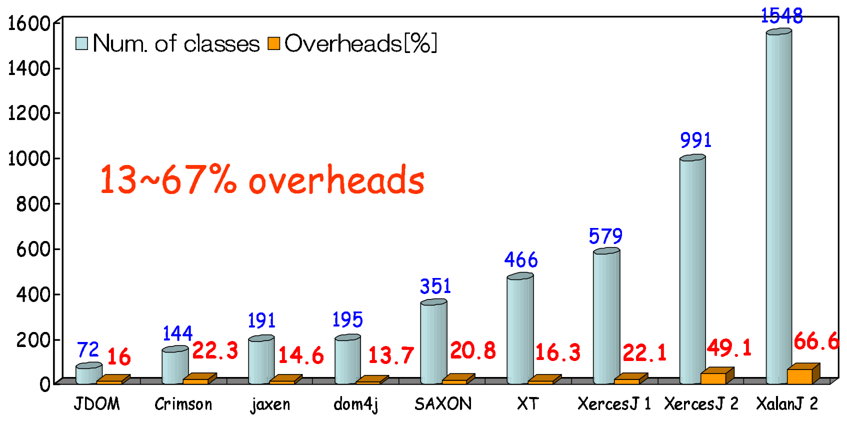
4.4.0.6 Cost of the `checkcast` instruction

Finally, we measured the execution time for version checking compared with the ordinary type checking. The sister-supported type checking includes not only the ordinary `checkcast` operation but also the checking of trusted instances. We ran a program that executes the `checkcast` instruction for every class included in a given application, and then we measured the total execution time of all the `checkcast` instructions. Both experiment programs ran after all the classes had been loaded and then the version compatibility of all the classes was verified. We successively ran the program twice; the execution time of the second run indicates the execution time of `checkcast` after the version compatibility of all the possibly relayed classes is verified during the first run. Some of the first checks also make use of the results of previous verifications.

Table 4.5 lists the results. The results are the average of 10,000 iterations. The total execution time of the first checks was from about

Table 4.4. Total loading time using an ordinary class loader and a sister class loader. All classes are sequentially loaded by the `loadClass()` method.

Program *(No. of classes)*		*Total loading time*		*sister*
		plain namespace	*sister namespace*	*/plain*
JDOM	(72 classes)	328 ms	382 ms	116.5%
Crimson	(144 classes)	569 ms	696 ms	122.3%
jaxen	(191 classes)	802 ms	919 ms	114.6%
dom4j	(195 classes)	1,308 ms	1,487 ms	113.7%
SAXON	(351 classes)	1,749 ms	2,113 ms	120.8%
XT	(466 classes)	1,223 ms	1,422 ms	116.3%
XercesJ 1	(579 classes)	2,495 ms	3,046 ms	122.1%
XercesJ 2	(991 classes)	4,144 ms	6,177 ms	149.1%
XalanJ 2	(1,548 classes)	12,884 ms	15,290 ms	166.6%



JDOM [87] : A simple Java representation of an XML document, version 1.0
Crimson [83] : A Java XML parser included with JDK 1.4 and greater, version 1.1.3
jaxen [89] : An XPath engine, version 1.0.
dom4j [62] : The flexible xml framework for Java, version 1.5.2
SAXON [46] : An XSLT and XQuery processor, version 6.5.3
XT [52] : A fast, free implementation of XSLT in java, version 20020426a
XercesJ 1 : The Xerces Java Parser 1.4.4.
XercesJ 2 : The Xerces2 Java Parser 2.6.2.
XalanJ 2 [84] : An XSLT processor for transforming XML documents, version 2.6.0.

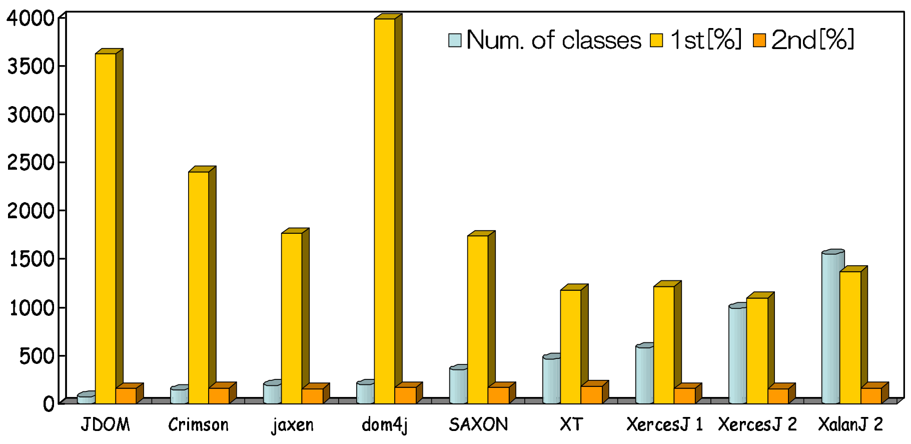
10 to 40 times slower than the ordinary `checkcast` operation. This is because the sister-supported type checker traverses all possibly relayed class types. Since each application has a different number of possibly relayed classes, the relative performance varies for each application. On the other hand, the second checks included only around 160% overhead compared to the ordinary `checkcast` operation. Note that this overhead is incurred only when `checkcast` examines the type of an instance coming from another sister namespace. The overhead is negligible in regular cases. It is the case that we use an interface type for adopting the delegation technique as described in section 2.2.2.3 (Figure 2.14 and 2.18). Note that adopting the delegation technique decreases the maintainability and availability of all related software components as mentioned that section.

We also compared the execution time of the type check with the time for marshalling and unmarshalling several XML data objects. Remember that the most harmless practice for the inter-component communication described in Chapter 2 is using a remote call, which passes an object by means of the call-by-value. This practice lets us avoid the problem of the version barrier, but it implies overhead due to the marshalling and unmarshalling for parameter passing. On the other hand, sister namespaces also let us avoid the problem, and it implies extra overhead only due to the type check. We measured the execution time for marshalling and unmarshalling DOM objects created by XercesJ 2.6.2 from 33 XML files taken from the Eclipse help system, which includes the Platform, Workbench, JDT, Plug-in and PDE document plug-ins. The overall file size was about 400KB. The measured execution time was 3 million times larger than the execution time of the ordinary `checkcast` operation. Of course, actual inter-component communication using a remote call would spend much more time for the network data transportations. Therefore, this result shows the sister namespace is a significantly faster solution compared to the solution of passing objects by a remote call.

[1]It is the case that we delegate an interface type to the parent class loader for exchanging the instances of two versions of a class type. This approach requires the passing objects to be cast to the target type from that interface type.

Table 4.5. Total execution time of the type check by `checkcast`

Program *(No. of classes)*	`checkcast`[1] [us]	*Sister namespaces* *1st* [us]	*2nd* [us]	*Relative performance* *1st*	*2nd*
JDOM (72 classes)	33.3	1,205.7	53.7	3,721%	261.3%
Crimson (144 classes)	69.0	1,659.7	112.6	2,505%	263.1%
jaxen (191 classes)	89.2	1,573.1	139.8	1,864%	256.7%
dom4j (195 classes)	109.7	4,371.7	185.8	4,085%	269.3%
SAXON (351 classes)	295.7	5,141.3	499.8	1,839%	269.0%
XT (466 classes)	381.5	4,505.8	698.7	1,281%	283.1%
XercesJ 1 (579 classes)	644.4	7,824.3	1,041.3	1,314%	261.5%
XercesJ 2 (991 classes)	1,158.6	11,534.6	1,798.0	1,096%	255.1%
XalanJ 2 (1,548 classes)	1,650.6	22,627.8	2,696.2	1,471%	263.3%



4.5 Related Work

In the object database community, several schema evolution techniques such as schema or class versioning [22, 1] have been studied. These techniques allow multiple co-existing versions of a schema or a class. Instances are evolved when passing through the version barrier into the modified application or other applications. Using such evolvable object databases is a workable alternative for component-based applications. However, our work concentrates on programming environments, especially where runtime overheads due to schema evolution must be severely minimized.

There have been other research activities tackling the version barrier problem in the programming language and environment community. Most of the previous research regarded the version barrier as a temporal boundary between *old* and *new* components and thus focused on dynamic software updates or evolution. That is, multiple versions of the same class type could not simultaneously co-exist in the running program. Therefore, our problems were not directly addressed. In this research area, the main topic is which existing object should be adapted to an updated version of the class type and how and when. For example, the work on the hotswapping of classes falls into this category. Malabarba *et al.* [53] modified the JVM to make a class reloadable at runtime so that all existing objects can be updated incrementally. The JPDA (Java Platform Debugger Architecture) [82] and the `java.lang.instrument` package of the Java2 SDK5.0 provide the restricted hotswap functionality, whereby existing instances can be considered as the new version of the class type without being updated. Hjalmtysson and Gray [40] implemented dynamic classes in C++ by using templates. Users can selectively update some but not all objects with the help of wrapper (or proxy) classes and methods. Hnětynka *et al.* [41] proposed the renaming approach using a bytecode manipulation tool. A class loader using this renaming approach allows the reloading of a class, although it renames that class.

Our work mainly focuses on the spatial version barriers among multiple components. An older version of a class type remains after a new version is loaded. Dynamic type changes such as predicate classes [16], reclassifying objects [28], and wide classes [54] may allow relaxing of the spatial version barrier, since multiple class members can be implicitly merged by the explicit composition operation. Type-based hotswapping proposed by Duggan *et al.* [30] is similar to our work but classified into the same category as the above systems. The .NET counterparts of the Java class loaders are *application domains*, which are used to load and execute assemblies and can run in a single process. However, they adopt the call-by-value semantics on inter-component communication between application domains using the .NET Remoting Framework. Of course, dynamically typed languages, such as CLOS, Self, and Smalltalk, provide more flexible mechanisms for allowing types to be changed at runtime. However, our challenge is relaxing only the version barrier in the strictly typed object-oriented world with negligible performance penalties. Our contribution is that we have provided a simple mechanism for relaxing the version barrier, which has been confusing Java programmers because of the complicated semantics.

4.6 Summary

This chapter presented the design and implementation of loosely-separated sister namespaces in Java. Combining multiple namespaces as sister namespaces can relax the version barrier between them. It thereby allows an instance to be assigned to a different version of that class type in that sister namespace. This mechanism was implemented on the IBM Jikes RVM for evaluation of the performance overhead. Our experiment showed that, once an instance passes into the sister namespace across the version barrier, all instances of that class type can go back and forth between the sister namespaces with significantly low performance overhead. Our experiment also demonstrated that the execution performance has only negligible overhead unless an instance is passed across the version barrier.

In the previous chapter, we have explained a class-level dynamic AOP system named Wool. It allows weaving aspects with a program at runtime by using the hotswap mechanism of the standard debugger interface called JPDA (Java Platform Debugger Architecture) [82]. The version compatible changes shown in this chapter are almost the same as that supported by the JPDA. Therefore, using sister namespaces will provide an object-level dynamic system while keeping the equivalent flexibility to Wool.

Chapter 5

A Dynamic Aspect Injection Container

This chapter describes a dynamic aspect injection container based on sister namespaces, which enables object-level dynamic weaving. Object-level dynamic weaving mechanisms can dynamically compose an aspect with a particular set of objects, not classes. This kind of dynamic weaving mechanisms can change the behavior of a part of application software locally and temporarily at runtime. The mechanism of the sister namespace can allow an instance being carried beyond the version barrier between namespaces even if each version of that class type is woven with a distinct aspect. Thus, an aspect can be dynamically *injected* into components across the namespace by passing an instance of the woven version of a class. The injection mechanism can be realized by an extension to a sort of DI *(Dependency Injection)* containers, which can reduce the dependency among components and thereby improving the reusability of components.

Key features of recent modern component frameworks such as Spring [23], Aspectwerkz [44], HiveMind [4], PicoContainer [70], and Seasar [88] are DI [35] and AOP. In other words, most of the DI containers support AOP. The so-called DI + AOP containers generate synergistic effect for the developers of component-based applications since both reduce dependency among components. The developers can inject dependency and compose an aspect in the same manner and obtain the initialized components from the container, specifying dependency and aspects between components by a separate configuration file (usually an XML file).

Although the component frameworks, mostly in Java, adopt a sophisticated mechanism for supporting dynamic weaving, it is not perfect. In the most container frameworks, an aspect is not composed with a component itself but the subclass of that component, especially called the *AOP proxy*. The developers can deal with the AOP proxy as the apparent component. Since the class file of the AOP proxy is secretly generated by using a bytecode engineering tools such as CGLIB [15], BCEL [57], and Javassist [18], the actual component can be dynamically replaced with the new proxy composed with a different aspect [1]. However, this mechanism requires the developers to design the apparent component class to be implemented or extended in the future. For example, if the apparent component class is declared as a regular class type, it must not be final and the declared methods must not be private for being overridden by the AOP proxy. On the other hand, if the apparent component is declared as an interface, the developers have to define an interface type for every component and access instances of the class through the interface type. That component is no longer a POJO (Plain Old Java Object).

This chapter introduces a dynamic aspect injection container named Wooler, which enables object-level dynamic weaving. Wooler can dynamically inject aspects into components by passing woven objects across the sister namespace. The rest of this chapter explains the design and implementation of the dynamic aspect injection container Wooler.

5.1 An Aspect in Wooler

An aspect in Wooler is represented as a regular component, which is composed of an XML configuration and an advice class. The XML configuration is used for describing dependency among components including aspects. The advice class includes advice methods that are invoked at the specified join points of components.

[1]The AOP proxy may just redirect method calls for the apparent component to the actual component.

5.1.1 An XML configuration

To describe dependency among components including aspects, an XML configuration is a better approach than providing a specific language extension. It is not used just for connecting components from outside the component-based application. It can work as keys of components including aspects. The following is an example XML configuration for dependency injection:

```
<beans>
  <bean id="taskfactory" class="wooler.ProxyFactoryBean">
    <property name="proxyTargetClass">
      <value>true</value>
    </property>
    <property name="target">
      <ref local="task"/>
    </property>
  </bean>
  <bean id="logaspect" class="LogAspect" method="log"/>
</beans>
```

An application reads this configuration when it creates application components:

```
Resource res = new ClassPathResource("task.xml");
BeanFactory f = new XmlBeanFactory(res);
BusinessTask task
= (BusinessTask)f.getBean("taskfactory", "logaspect");
```

The XML configuration includes no dependency between the `task` component and the `logaspect` aspect. Instead, the application explicitly takes the names of the component and the aspect. Thus, the `logaspect` can be easily composed with a different component:

```
BusinessTask newtask
= (BusinessTask)f.getBean("newtask", "logaspect");
```

The `task` component and the `newtask` component compose with the same `logaspect` component. It means that the `logaspect` aspect can modularize the cross cutting concern lying between the `task` component and the `newtask` component. The developers need not manage the map of the aspect objects including aspect fields. Wooler, of course, provides the way for explicitly describing dependency between a component and an aspect:

```
<bean id="taskfactory" class="wooler.ProxyFactoryBean">
  <property name="interceptor">
      <value>logaspect</value>
  </property>
</bean>
```

To filter some messages into the aspect, the developers can specify a message pattern as follows:

```
<bean id="logaspect" class="LogAspect" method="log">
  <property name="pattern">
    <value>.*get.*</value>
  </property>
</bean>
```

This property can serve as pointcut designators in AspectJ. Then, the method calls against the `task` component and beginning with `get` are dispatched to the `log` method of the `logaspect` component.

5.1.2 An advice class

An advice body is described as a method body declared in an advice class. The advice class in Wooler is the same as that of Wool mentioned chapter 3, which is a normal class as following:

```
class LogAspect {
  public void log(MethodInvocation mi) {
    if (mi.getMethod().getName().startsWith("get")) {
      ...
    }
  }
}
```

The `log()` method is invoked at the execution points specified by the XML configuration as mentioned before. Moreover, the following method is also supported to efficiently obtain reflective information about the current join point for the advice to use:

```
public void log(BusinessTask $this) {
    out.println("This␣object␣is␣" + $this);
}
```

We can access the `this` object for the current join point by naming a method parameter `$this`. Similar to *this()* pointcut of AspectJ, advice is not invoked if the type of a method parameter is different from the object at the identified join point. We can also use all reflective information by using `$thisJoinpoint` as a parameter. Mainly, this object is used to obtain certain dynamic information such as the currently executing object or the target object or the arguments. Table 5.1 lists several parameter names for accessing the context of the current join point.

Table 5.1. Parameter names for accessing the context of current join point.

`$this`	access the `this` object in the current join point.
`$target`	access the target object in the current join point.
`$arg1, $arg2, ...`	access the argument object in the current join point.
`$thisJoinpoint`	access the execution context in the current join point. The type of that parameter must be `Joinpoint`.

In some DI containers, for intercepting the execution of a program, the aspect has to override or implement the interceptor class such as

MethodInterceptor and FieldInterceptor provided by the container framework:

```
class LogInterceptor implements MethodInterceptor {
  public Object invoke(MethodInvocation mi)
    throws Throwable {
    if (mi.getMethod().getName().startsWith("get")) {
      ...
    }
  }
}
```

Since this interceptor model allows us a sort of meta programming using plenty of the dynamic information, it may be enough in practice. However, this interceptor model is an old-fashioned approach. The complicate meta programming has annoyed a number of developers up to now. And it is not good in terms of the execution performance for reifying an execution context at every interception.

5.2 Aspect weaving across sister namespaces

Although Wooler compose an aspect class with the AOP proxy as usual, the proxy class is not a subclass or a subtype of the component class but a *sister* class of that one. Wooler first creates a sister class loader of the class loader of the target component, and then generates the AOP proxy including some advice invocations following the XML configuration. The proxy class, generated by using a bytecode engineering tool, are loaded into the sister namespace at runtime. Since the AOP proxy becomes a sister of the component class, an object of the AOP proxy can be injected to the running application and regarded as that component mentioned in chapter 4. Pseudo code for aspect weaving in Wooler is following:

```
BusinessTask task
= (BusinessTask)f.getBean("taskfactory", "logaspect");

public Object getBean(String prop, String aspect)
   throws Exception {
```

```
  ...
  loader = Thread.currentThread.getContextLoader();
  sister = new ClassLoader(loader.getParent(),loader);
  byte[] proxy = translator.weave(prop, xml, aspect);
  Class c = sister.load(proxy);
  return c.newInstance();
}
```

Figure 5.1 illustrates the AOP proxy in the Wooler compared to the ordinary designs.

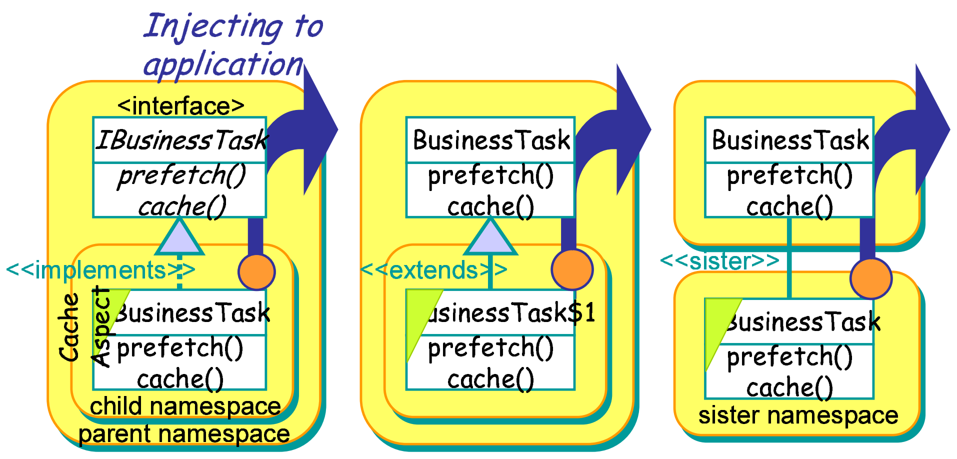


Figure 5.1. The AOP proxy is defined as a subtype implementing the component interface in the left, as a subclass extending the component class in the center, and as a sister class of the component class in the right.

Unlike ordinary container frameworks, Wooler imposes no restriction on the component class. This is because the sister classes have only to satisfy version compatibility. The version compatible changes include differences of static members such as static methods, constructors, and initializers and differences of instance members such as instance methods. Therefore, the component class need not be defined as an interface or a non-final class, and the declared methods can be defined as static or private ones. Of course, Wooler can replace the AOP proxy at runtime by recreating the sister namespace against the target component as long as the version compatibility between sister classes is maintained.

Chapter 6

Conclusion

This thesis has discussed dynamic AOP weaving mechanisms, focusing on efficiency and granularity. The mechanisms presented in this thesis enable efficient class-level dynamic weaving and fine-grained object-level dynamic weaving. Thus, it will propel AOP to the realistic option for the today's software developers, who have a strong desire for a solution to create high quality software.

Contributions

The contributions by this thesis are summarized as follows:

- This thesis proposed a selective, just-in-time, aspect weaving mechanism for enabling efficient class-level dynamic weaving. This mechanism allows an aspect weaver and its user to insert suitable hooks into the running program in consideration of the overall performance of the application. This will contribute to the designers of virtual machines and just-in-time compilers for supporting AOP, or for a pure aspect-oriented programming language.
- Then this thesis proposed the novel concept and design for loosely-separated namespaces to address the problem of the version bar-

rier, and enabling fine-grained object-level dynamic weaving. This mechanism can relax the version barrier between namespaces while keeping type-safety, efficiency, and laziness of the Java class loader. This also brings a brilliant settlement to component frameworks seeking loosely-coupled components. This will also contribute to the designers of a new specific programming language aimed for loosely-coupled components.

- Furthermore, this thesis proposed a dynamic aspect injection mechanism for realizing an object-level dynamic weaving based on the previous two mechanisms. It emerged that the existing lightweight container frameworks does not excellently support dynamic weaving only by exploitation of subtyping or subclassing. And this strongly demonstrated usefulness of the loosely-separated namespace in practice.

Limitations

The limitations of this thesis and dynamic AOP itself are summarized as follows:

Inter-type declarations Aspects in AspectJ can declare members (fields, methods, and constructors) that are owned by other types. These are called inter-type members (formerly called the introductions). The mechanisms proposed in this thesis do not support this language constructs. Since the JVM does not allow changing the class schema at runtime, inter-type declarations are not directly realized. However, the inter-type members are always accessed from advice code and thus these members actually need not to be declared other types. Therefore, the inter-type members are enough to be declared in an aspect if the scope of these members is properly managed. For example, field access code described in an inter-type method need to be properly redirected to a field declared the corresponding owned type.

Aspects in AspectJ can also declare that other types implement new interfaces or extend a new class. Wool and Wooler do not support this mechanism. It is also difficult for dynamic AOP itself, since implementing this kind of runtime type changes in statically typed object-oriented languages causes large performance penalties.

Around advice Current implementation of Wool does not support dynamic weaving of around advice. When Wool runs at the breakpoint-based execution mode, the running program always returns to the intercepted execution point after the inserted advice code is executed. That is, Wool can not skip or cancel the currently executed method by popping the corresponding execution frame. To realize around advice in Wool, we must enhance the JPDA implementation to be able to control the executing program with keeping the semantics of that program.

On-time weaving The dynamic weaving processes depend on the thread or the OS scheduler, since dynamic weavers always begin working after the target program runs. Suppose that multiple weavers are trying to compose distinct aspects with the identical class in parallel, the order of woven aspects depends on the thread scheduler although AspectJ provides the `precedence` mechanism to control the order of advice executions. Suppose that dynamic weavers affect multiple processes as well as threads, which enables so-called inter-process aspects [65]. It is difficult to compose an aspect with all threads or processes following universal conveniences. If do that, it may cause a deadlock, and the weaving process may never start. Therefore, dynamic AOP systems including Wool and Wooler need to provide high abstractions to monitor the target program and control the weaver.

Future directions

Possible future directions of this thesis are following:

Adaptive aspect weaving Our first version of Wool requires the programmers to make decisions about the hooks. This manual selection has a high probability of producing good results. However, sometimes the programmer does not know the best combination of hooks as breakpoints and as method calls. In the future, we will implement a sophisticated profiler like that of the HotSpot VM [81] to automatically select the most appropriate hooks.

Soundness of the sister namespace Currently, we just formalized an abstract model of the sister namespace. We need to explore soundness of the sister namespace. Then, we also examine this issue with respect to the Java security architecture [36].

Apply to the production VM The mechanisms proposed by this thesis need to apply the production VM and work in the real world. Although our just-in-time aspect weaver is implemented on the Sun JVM, it accesses the JVM indirectly using that debugger interface. Meanwhile loosely-separated namespaces are implemented on the research VM of IBM. Most of the production VMs equip various kinds of optimization mechanisms in the crown of information science, toward just-in-time compilation including inlining and devirtualization, lock and memory management. Our mechanisms must not forbid those optimizations. We should confirm it by exposing them to the real world such as the BEA's JRockit VM cite-jrockit.

Implement dynamic aspect injection container Unfortunately, a dynamic aspect injection container named Wooler has not been implemented yet because it is easy-to-prepare. However, container frameworks require not only high throughput for processing requests but also high scalability. In fact, modern component frameworks such as a J2EE server deal with a vast number of clients simultaneously. On the other hand, we did not assume that an enormous number of sister namespaces is co-exists at the same time. Therefore, Wooler should be implemented, widely-used and enhanced; if it has improper scalability in practice.

Bibliography

[1] Andrea H. Skarra and Stanley B. Zdonik. The Management of Changing Types in an Object-Oriented Database. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'86)*, volume 11 of *SIGPLAN Notices 21*, pages 483–495, Portland, Oregon, nov 1986.

[2] AOP Alliance. Aop alliance (java/j2ee aop standards). `http://aopalliance.sourceforge.net/`, 2002.

[3] Apache Software Foundation. Apache Geronimo. available at: `http://geronimo.apache.org/`, 2003.

[4] Apache Software Foundation. HiveMind. available at: `http://jakarta.apache.org/hivemind/`, 2005.

[5] B. Alpern and C. R. Attanasio and J. J. Barton and M. G. Burke and P Cheng and J.-D. Choi and A. Cocchi and S. J. Fink and D. Grove and M. Hind and S. F. Hummel and D. Lieber and V. Litvinov and M. F. Mergen and T. Ngo and J. R. Russell and V. Sarkar and M. J. Serrano and J. C. Shepherd and S. E. Smith and V. C. Sreedhar and H. Srinivasan and J. Whaley. The Jalapeno Virtual Machine. *IBM System Journal*, 39(1):211–238, feb 2000.

[6] Jason Baker and Wilson Hsieh. Runtime Aspect Weaving Through Metaprogramming. In *1st International Conference on Aspect-Oriented Software Development*, pages 86–95, 2002.

[7] Bo Noerregaard Joergensen and Eddy Truyen and Frank Matthijs and Wouter Joosen. Customization of Object Request Brokers by Application Specific Policies. In *Middleware 2000 conference*, 2000.

[8] Bob Cancilla and Kevin Postreich. *IBM Websphere Application Server Express: Pathways to Success on the Web.* Midrange Computing, 2004.

[9] Christoph Bockisch, Michael Haupt, Mira Mezini, and Klaus Ostermann. Virtual Machine Support for Dynamic Join Points. In *International Conference on Aspect-Oriented Software Development*, 2004.

[10] Bowen Alpern and Anthony Cocchi and David Grove. Dynamic Type Checking in Jalapeño. In *Java Virtual Machine Research and Technology Symposium*, 2001.

[11] Bowen Alpern and Anthony Cocchi and Stephen J. Fink and David Grove and Derek Lieber. Efficient Implementation of Java Interfaces: Invokeinterface Considered Harmless. In *Proceedings of the 2001 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA 2001)*, number 11 in SIGPLAN Notices, vol.36, pages 108–124, Tampa, Florida, USA, nov 2001. ACM.

[12] Mathias Braux and Jacques Noyé. Towards Partially Evaluating Reflection in Java. In *Proceedings of the 2000 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM '00)*, 2000.

[13] Johan Brichau, Kim Mens, and Kris De Volder. Building Composable Aspect-Specific Language with Logic Metaprogramming. In *Generative Programming and Component Engineering - Generative Programming and Component Engineering*, LNCS 2487, pages 93–109. Springer-Verlag, 2002.

[14] Budi Kurniawan and Paul Deck. *How Tomcat Works: A Guide To Developing Your Own Java Servlet Container.* Brainysoftware.Com, 2004.

[15] cglib. Code Generation Library. available at: `http://cglib.sourceforge.net/`, 2002.

[16] Craig Chambers. Predicate Classes. In *ECOOP'93 - Object-Oriented Programming, 7th European Conference*, volume 707 of *Lecture Notes in Computer Science*, pages 268–296, Kaiserslautern, Germany, jul 1993. Springer-Verlag.

[17] Charles Simonyi. The Death of Computer Languages, The Birth of Intentional Programming. In *Technical Report MSR-TR-95-52, Microsoft Research*, 1995.

[18] Shigeru Chiba. Load-time structural reflection in Java. In *ECOOP 2000*, LNCS 1850, pages 313–336. Springer-Verlag, 2000.

[19] Shigeru Chiba and Rei Ishikawa. Aspect-oriented programming beyond dependency injection. In *Proceedings of the ECCOP 2005 - Object-Oriented Programming: 19th European Conference*, volume 3586 of *Lecture Notes in Computer Science*, pages 121–143, Glasgow, UK, jul 2005.

[20] Shigeru Chiba and Muga Nishizawa. An Easy-to-use but Efficient Java Bytecode Translator. In *Second International Conference on Generative Programming and Component Engineering (GPCE'03)*, Erfurt Germany, September 2003.

[21] Shigeru Chiba, Yoshiki Sato, and Michiaki Tatsubori. Using HotSwap for Implementing Dynamic AOP Systems. In *1st Workshop on Advancing the State-of-the-Art in Run-time Inspection, July, 2003, Darmstadt, Germany held in conjuction with ECOOP 2003*, July 2003.

[22] Stewart M. Clamen. Type Evolution and Instance Adaptation. Technical Report CMU-CS-92–133, Carnegie Mellon University School of Computer Science, Pittsburgh, PA, 1992.

[23] Craig Walls and Ryan Breidenbach. *Spring in Action.* Manning Publications, 2005.

[24] Jonathan Davies, Nick Huismans, Rory Slaney, Sian Whiting, Matthew Webster, and Robert Berry. Aspect Oriented Profiler. In *2nd International Conference on Aspect-Oriented Software Development*, 2003.

[25] Edsger Wybe. Dijkstra. *A Discipline of Programming.* Prentice Hall (Sd), 1976.

[26] Mikhail Dmitriev. Towards flexible and safe technology for runtime evolution of java language applications. In *Proceedings of the Workshop on Engineering Complex Object-Oriented Systems for Evolution, in association with OOPSLA 2001 International Conference*, pages 14–18, Tampa Bay, Florida, USA, October 2001.

[27] Sophia Drossopoulou. An Abstract Model of Java Dynamic Linking and Loading. In *Types in Compilation*, volume 2071 of *Lecture Notes in Computer Science*, pages 53–84, Montreal, Canada, September 2000. Springer.

[28] Sophia Drossopoulou, Ferruccio Damiani, Mariangiola Dezani-Ciancaglini, and Paola Giannini. Fickle : Dynamic Object Reclassification. In *ECOOP 2001 - Object-Oriented Programming, 15th European Conference*, volume 2072 of *Lecture Notes in Computer Science*, pages 130–149, Budapest, Hungary, jun 2001. Springer.

[29] Sophia Drossopoulou, David Wragg, and Susan Eisenbach. What is Java Binary Compatibility? In *Proceedings of the 1998 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications (OOPSLA '98)*, pages 341–361, Vancouver, British Columbia, Canada, oct 1998.

[30] Dominic Duggan. Type-Based Hot Swapping of Running Modules. In *Proceedings of the Sixth ACM SIGPLAN International Conference on Functional Programming (ICFP '01)*, volume 10 of *SIGPLAN Notices 36*, pages 62–73, Florence, Italy, oct 2001. ACM.

[31] Easy Software Foundation. ajProfiler - easy java profiler. `http://http://ajprofiler.sourceforge.net/`, 2002.

[32] Eddy Truyen and Bo Norregaard Jrgensen and Wouter Joosen. Customization of Component-based Object Request Brokers through Dynamic Configuration. In *Technology of Object-Oriented Languages and Systems*, 2000.

[33] Robert Englander. *Developing Java Bean.* O'Reilly and Associates, Inc., 1997.

[34] Huw Evans and Peter Dickman. Zones, Contracts and Absorbing Changes: An Approach to Software Evolution. In *Proceedings of the 1999 ACM SIGPLAN Conference on Object-Oriented Programming*

Systems, Languages & Applications (OOPSLA), number 10 in SIGPLAN Notices vol.34, pages 415–434, Denver, Colorado, USA, nov 1999. ACM.

[35] Martin Fowler. Inversion of Control Containers and the Dependency Injection pattern. `http://www.martinfowler.com/articles/injection.html`, 2004.

[36] Li Gong, Gary Ellison, and Mary Dageforde. *Inside Java2^{TM} Platform Security: Architecture, API Design, and Implementation 2nd Edition.* Addison-Wesley, Boston, Mass., 2003.

[37] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification Second Edition.* Addison-Wesley, Boston, Mass., 2000.

[38] Kari L. Halsted and James H. (Jamie) Roberts. Eclipse help system: an open source user assistance offering. In *Proceedings of the 20st annual international conference on Documentation, SIGDOC 2002*, pages 49–59, Toronto, Ontario, Canada, oct 2002. ACM.

[39] William Harrison and Harold Ossher. Subject-Oriented Programming (A Critique of Pure Objects). In *Object-Oriented Programming, Systems, Languages, and Applications.*, pages 411–428, 1993.

[40] Gísli Hjálmtyśson and Robert Gray. Dynamic C++ Classes: A lightweight mechanism to update code in a running program. In *Proceedings of the USENIX Annual Technical Conference*, New Orleans, Louisiana, 1998. USENIX.

[41] Petr Hnětynka and Petr Tůma. Fighting Class Name Clashes in Java Component Systems. In *Modular Programming Languages, Joint Modular Languages Conference, JMLC 2003*, volume 2789 of *Lecture Notes in Computer Science*, pages 106–109, Klagenfurt, Austria, aug 2003. Springer.

[42] Kazuaki Ishizaki, Motohiro Kawahito, Toshiaki Yasue, Hideaki Komatsu, and Toshio Nakatani. A study of devirtualization techniques for a Java$^{\text{TM}}$ Just-In-Time compiler. In *Proceedings of the 2000 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications (OOPSLA 2000)*, number 10 in

SIGPLAN Notices, vol.35, pages 294–310, Minneapolis, Minnesota, USA, oct 2001. ACM.

[43] Jon Mountjoy and Chugh Avinash and Brett McLauglin. *Weblogic: The Definitive Guide.* Oreilly & Associates Inc, 2004.

[44] Jonas Bonér, Alexandre Vasseur. AspectWerkz - Plain Java AOP. available at: `http://aspectwerkz.codehaus.org/`, 2002.

[45] JUnit FAQ. Why do I get an error (ClassCastException or LinkageError) using the GUI TestRunners? available at: `http://junit.sourceforge.net/doc/faq/faq.htm`, 2002.

[46] Michael Kay. SAXON The XSLT and XQuery Processor. available at: `http://saxon.sourceforge.net/`, 2001.

[47] Ralph Keller and Urs Hëlzle. Binary Component Adaptation. In *ECOOP'98 - Object-Oriented Programming*, LNCS 1445, pages 307–329. Springer-Verlag, 1998.

[48] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. In *ECOOP 2001*, LNCS 2072, pages 327–353. Springer-Verlag, 2001.

[49] Gregor Kiczales, John Lamping, Anurag Menhdhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *Proceedings European Conference on Object-Oriented Programming*, volume 1241, pages 220–242. Springer-Verlag, Berlin, Heidelberg, and New York, 1997.

[50] Gregor Kiczales, Jim Des Rivieres, and Daniel Bobrow. *The Art of the Metaobject Protocol.* The MIT Press, 1991.

[51] Sheng Liang and Gilad Bracha. Dynamic Class Loading in the Java Virtual Machine. In *Proceedings of OOPSLA'98, Proceedings of the 1998 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications*, number 10 in SIGPLAN Notices, vol.33, pages 36–44, Vancouver, British Columbia, Canada, oct 1998. ACM.

[52] Bill Lindsey. XT. available at: `http://www.blnz.com/xt/`, 2002.

[53] Scott Malabarba, Raju Pandey, Jeff Gragg, Earl Barr, and J. Fritz Barnes. Runtime Support for Type-Safe Dynamic Java Classes. In *Proceedings of ECOOP 2000 - Object-Oriented Programming, 14th European Conference*, volume 1850 of *Lecture Notes in Computer Science*, pages 337–361. Springer-Verlag, jun 2000.

[54] Manuel Serrano. Wide Classes. In *Proceedings of the ECCOP'99 - Object-Oriented Programming, 13th European Conference*, volume 1628 of *Lecture Notes in Computer Science*, pages 391–415, Lisbon, Portugal, jun 1999. Springer-Verlag.

[55] Francisco Reverbel Marc Fleury. The JBoss Extensible Server. In *ACM/IFIP/USENIX International Middleware Conference*, volume 2672 of *Lecture Notes in Computer Science*, pages 344–373, Rio de Janeiro, Brazil, jun 2003. Springer.

[56] Mark Fleury and Scott Stark and Norman Richards. *JBoss4.0 - The Official Guide.* Sams, 2005.

[57] Markus Dahm. Byte Code Engineering with the BCEL API. In *Technical Report 8-17-98, Freie Universit at Berlin, Institut f ur Informatik*, 2001.

[58] Hidehiko Masuhara, Gregor Kiczales, and Chris Dutchyn. A Compilation and Optimization Model for Aspect-Oriented Programs. In *Compiler Construction, 12th International Conference, CC 2003, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2003, Warsaw, Poland, April 7-11, 2003, Proceedings*, volume 2622 of *Lecture Notes in Computer Science*, pages 46–60. Springer, apr 2003.

[59] Hidehiko Masuhara and Akinori Yonezawa. Design and partial evaluation of meta-objects for a concurrent reflective language. In *Proceedings of the ECCOP'98 - Object-Oriented Programming, 12th European Conference*, volume 1445 of *Lecture Notes in Computer Science*, pages 418–439, Brussels, Belgium, jul 1998.

[60] Vlada Matena and Beth Stearns. *Applying Enterprise JavaBeans^{TM}: Component-Based Development for the J2EE^{TM} Platform.* Pearson Education, 2001.

[61] Mathias Braux and Jacques Noyé. Towards Partially Evaluating Reflection in Java. In *Proceedings of the 2000 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM '00)*, number 11 in SIGPLAN Notices vol.34, pages 2–11, Boston, Massachusetts, USA, jan 2000. ACM.

[62] Metastaff, Ltd. dom4j: the flexible xml framework for Java. available at: `http://www.dom4j.org/`, 2001.

[63] Michael Golm and Jürgen Kleinöder. Jumping to the Meta Level: Behavioral Reflection Can Be Fast and Flexible. In *Proceedings of Meta-Level Architectures and Reflection, Second International Conference, Reflection'99*, volume 1616 of *Lecture Notes in Computer Science*, pages 22–39. Springer-Verlag, 1999.

[64] Adam Nathan. *.NET and COM: The Complete Interoperability Guide.* Sams, 2002.

[65] Muga Nishizawa, Shigeru Chiba, and Michiaki Tatsubori. Remote pointcut: a language construct for distributed AOP. In *Proceedings of the 3rd International Conference on Aspect-Oriented Software Development, AOSD 2004, Lancaster, UK, March 22-24, 2004*, pages 7–15. ACM, 2004.

[66] Hirotaka Ogawa, Kouya Shimura, Satoshi Matsuoka, Fuyuhiko Maruyama, Yukihiko Sohda, and Yasunori Kimura. OpenJIT Frontend System: an implementation of the reflective JIT compiler frontend. In *ECOOP 2000*, LNCS 1850. Springer-Verlag, 2000.

[67] OMG. *The Common Object Request Broker: Architecture and Specification. Revision 2.0.* OMG Document, 1995.

[68] Doug Orleans and Karl Lieberherr. DJ: Dynamic adaptive programming in Java. In *In Reflection 2001: Meta-level Architectures and Separation of Crosscutting Concerns*, LNCS 2192, pages 73–80. Springer-Verlag, 2000.

[69] Renaud Pawlak, Lionel Seinturier, Laurence Duchien, and Gèrard Florin. JAC: A flexible framework for AOP in Java. In *Reflection 2001*, pages 1–24, 2001.

[70] PicoContainer Organization. PicoContainer. available at: `http://www.picocontainer.org/`, 2003.

[71] Andrei Popovici, Gustavo Alonso, and Thomas Gross. Just in Time Aspects: Efficient Dynamic Weaving for Java. In *2nd International Conference on Aspect-Oriented Software Development*, pages 100–109, 2003.

[72] Andrei Popovici, Thomas Gross, and Gustavo Alonso. Dynamic Weaving for Aspect-Orinented Programming. In *1st International Conference on Aspect-Oriented Software Development*, pages 141–147, 2002.

[73] Christian Prehofer. Feature-Oriented Programming: A Fresh Look at Objects. In *ECOOP'97 - Object-Oriented Programming, 11th European Conference, Jyväskylä, Finland, June 9-13, 1997, Proceedings*, volume 1241 of *Lecture Notes in Computer Science*, pages 419–443. Springer, Jun 1997.

[74] Yoshiki Sato and Shigeru Chiba. Loosely-separated "Sister" Namespaces in Java. In *Proceedings of the ECCOP 2005 - Object-Oriented Programming: 19th European Conference*, volume 3586 of *Lecture Notes in Computer Science*, pages 49–70, Glasgow, UK, jul 2005.

[75] Yoshiki Sato, Shigeru Chiba, and Michiaki Tatsubori. A Selective, Just-In-Time Aspect Weaver. In *Second International Conference on Generative Programming and Component Engineering (GPCE'03)*, pages 189–208, Erfurt Germany, September 2003.

[76] Marc Segura-Devillechaise, Gilles Muller Jean-Marc Menaud, and Julia L. Lawall. Web Cache Prefetching as an Aspect: Towards a Dynamic-Weaving Based Solution. In *2nd International Conference on Aspect-Oriented Software Development*, pages 110–119, 2003.

[77] Sherry Shavor, Jim D'Anjou, Scott Fairbrother, Dan Kehn, John Kellerman, and Pat McCarthy. *The Java Developer's Guide to Eclipse.* Addison-Wesley, 2003.

[78] Shigeru Chiba. A Metaobject Protocol for C++. In *Proceedings of the OOPSLA'95, Tenth Annual Conference on Object-Oriented Programming Systems, Languages, and Application*, number 10 in SIGPLAN Notices, vol.30, pages 285–299, Austin, Texas, USA, oct 1995.

[79] Spec - The Standard Performance Evaluation Corporation. SPECjvm98. `http://www.spec.org/osg/jvm98/`, 1998.

[80] Sun Microsystems. Dynamic Proxy Classes. available at: `http://java.sun.com/j2se/1.3/docs/guide/reflection/proxy.html`, 1999.

[81] Sun Microsystems. The Java HotSpot Performance Engine Architecture. available at: `http://java.sun.com/products/hotspot/whitepaper.html`, 1999.

[82] Sun Microsystems. Java^{TM} Platform Debugger Architectuer. available at: `http://java.sun.com/j2se/1.4/docs/guide/jpda`, 2001.

[83] The Apache XML Project. Crimson Java Parser. available at: `http://xml.apache.org/crimson`, 2000.

[84] The Apache XML Project. Xalan Java XSLT Processor. available at: `http://xml.apache.org/xalan-j`, 2002.

[85] The Apache XML Project. Xerces2 Java Parser. available at: `http://xml.apache.org/xerces2-j`, 2002.

[86] The Eclipse Foundation. Eclipse.org. homepage: `http://www.eclipse.org/`, 2001.

[87] The JDOM^{TM} Projec. JDOM. available at: `http://www.jdom.org/`, 2000.

[88] The Seasar Foundation. Seasar - DI Container with AOP. available at: `http://www.seasar.org/`, 2004.

[89] The Werken Company. jaxen: universal java xpath engine. available at: `http://jaxen.org/`, 2001.

[90] Toshio Suganuma and Takeshi Ogasawara and Mikio Takeuchi and Toshiaki Yasue and Motohiro Kawahito and Kazuaki Ishizaki and Hideaki Komatsu and and Toshio Nakatani. Overview of the IBM Java Just-In-Time Compiler. *IBM Systems Journals*, 39(1):175–193, 2000.

[91] Vijay Saraswat. Java is not type-safe. `www.research.att.com/~vj/bug.html`, 1997.

[92] Ian Welch and Robert Stroud. Kava - Using Bytecode Rewriting to add Behavioural Reflection to Java. In *USENIX Conference on Object-Oriented Technology*, 2001.

[93] Khaled Yagoub, Daniela Florescu, Valerie Issarny, and Patrick Valduriez. Caching Strategies for Data-Intensive Web Sites. In *Proceedings of the 24th International Conference on Very Large Databases (VLDB)*, Cairo Egypt, sep 2000.