Revision Classes for Explicit Versioning

Susanne Cech Previtali
Department of Computer Science
ETH Zurich
Switzerland

Michele Schäuble
Department of Computer Science
ETH Zurich
Switzerland

Thomas R. Gross
Department of Computer Science
ETH Zurich
Switzerland

ABSTRACT

The source code of software is typically managed by version control systems that keep track of the different versions of *files* over time. As versioning is associated with a file and not a class, the versioning mechanism is semantically detached from the actual source code. This paper introduces the concept of *revision classes*. Revision classes provide an explicit versioning mechanism for classes that, similar to inheritance, allows the developer to redefine existing class members and add new class members in a new version. This explicit versioning mechanism allows the developer to explicitly declare the deltas to a previous version. The developer may reflect on the necessity of updates and thus errors can be avoided that sneak into the source code by inconsiderate changes.

Categories and Subject Descriptors

D.2.7 [**Distribution, Maintenance, and Enhancement**]: Version control; D.3.2 [**Language Classifications**]: Object-oriented languages; D.3.3 [**Language Constructs and Features)**]: Modules, packages

1. INTRODUCTION

Object-oriented programming languages are recognized to produce more reusable, extensible, robust, and correct programs than earlier programming languages and are used today for many software projects. Software quality factors do not include maintainability, although estimated 70% of the software cost is devoted to maintenance ([8], page 17). The source code of software is typically managed by version control systems (e.g., SVN, CVS). A version control system associates versions with *files*. Files are checked out of a repository, edited, and checked in again with a comment that documents the modifications, thus creating the next version of the files. The file versions are implicit, only text comparison tools make the differences explicit. By working on the granularity of files, version control systems are general tools for any kind of file-based versioning and, in particular, are independent from programming languages.

When working on a checked-out version, a developer modifies not only the methods intended (e.g., the method containing the bug), but also other unrelated methods (e.g., code refactoring). When several developers work on the same project, comments within the code or from the versioning system serve as documentation on what was changed. In addition, a "commit" is not necessarily connected to a logical update. Changes that belong together logically may be split over several commits. As versioning is associated with a file and not a class, the versioning mechanism is semantically detached from the actual source code. By changing a file and not the structural elements of a class, information on where the class is changed is lost. In addition, an implicit versioning mechanism may be responsible for errors that are introduced by modifying the code directly.

Inheritance promotes better reuseability by factoring out common behavior to superclasses. Subclasses are derivations of their respective superclasses and thus provide a *vertical* class extension mechanism. Instead of defining data-type abstractions, subclassing is often used as a means of implementation inheritance. Changes are specified explicitly as deltas to a superclass, code duplication is avoided, and type safety and polymorphism ensured. Inheritance can be seen as a versioning mechanism, as each subclass represents a more specialized version of the superclass.

In this paper, we introduce the concept of revision classes, an explicit class versioning mechanism. Revision classes introduce a notion of time and thus support a *horizontal* class extension mechanism. Revision classes take the idea of inheritance to define extensions (i.e., additions of class members) and specializations (i.e., redefinitions of class members). The developer explicitly declares changes and thus inconsiderate changes may be avoided. The introduction of revision classes promises further improvements regarding the robustness and correctness of software written in object-oriented programming languages.

We make the following contributions: (1) We introduce an explicit version mechanism for classes. (2) We describe a language extension for Java which we implemented by modifying an existing Java compiler. (3) We validated the approach by a small case study and modified three versions of an existing Java program to exploit revision classes.

The paper is structured as follows. Section 2 discusses related work. Section 3 introduces the concept of revision classes, the Java language extension, and our compiler called *Crema*. Section 4 presents the case study as first validation of the concept. Section 5 discusses the presented approach and Section 6 summarizes and concludes.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
RAM-SE '09, July 7, 2009, Genova, Italy
Copyright 200X ACM 978-1-60558-548-2/09/07 ...$10.00.

2. RELATED WORK

Robbes and Lanza [11] collect high-level change data from IDEs to model software evolution. The developer's intention about the changes to be applied to a program is tracked by tools that watch the interaction of the developer with the IDE. The authors model semantic changes (e.g., the addition or redefinition of a method), which are stored to reproduce an arbitrary program version. Our approach supports the same level of granularity of explicit changes. Instead of tracking the IDE interaction, we request the programmer to explicitly *state* the necessary changes.

Bierman et al. [2] propose language support and a type system for explicit version control by the developer. UpgradeJ allows the developer to define three kinds of class *upgrades* with different update possibilities. Classes have explicit version numbers; types declare the version numbers they accept. The developer must specify a possibly open range of types a variable can accept. UpgradeJ provides for incremental type checking of version classes. Revision classes also introduce explicit version control in classes. In contrast to UpgradeJ, not every revision class is a type: The fields and methods of *all* versions of a class are merged into a class equivalent to a conventional class; this *merged* class yields a type. Such a typing mechanism allows the coexistence of conventional classes and revision classes and thus the usage of existing libraries.

Bergel et al. [1] introduce special Java packages called *Classboxes* that enable the refinement of existing classes by method additions and redefinitions. Classboxes provide their own scope at run-time, in which the refinements are visible and as a consequence, multiple versions of classes may be active at run-time. While our approach supports the expression of similar changes, each class version *updates* the previous class versions and thus possibly invalidates prior definitions. Classboxes do not describe explicit *versions* of a program and thus the same maintenance problems as for conventional classes exist.

Denker et al. [5] present *Changeboxes* as means of defining an explicit snapshot as a version of a program. Changeboxes encapsulate the specification of changes to be applied to a set of prior Changeboxes. Every new program version is defined by a Changebox, and several versions can be merged into a new version of a Changebox. Merging several Changeboxes may result in conflicting change specifications, which must be resolved by a conflict resolution strategy (e.g., redefinition of a removed method). As multiple Changeboxes can be executed by a virtual machine at the same time, multiple versions of classes can co-exist. Our approach discards old class versions and only implicitly supports the removal of class members.

Mezini et al. [10, 9] introduce Caesar, a component model that groups a set of collaborating classes. A component is a virtual class that defines a *class-family* [6], its element classes are inner classes. By extending a component, the inner classes can be extended (and implemented partially) as well, either as implementation or binding. Binding components encapsulate also pointcuts and advice declarations. Caesar components promote a better modularization and extensibility of classes and modules. Caesar components are not explicit *versions*, versioning must be integrated with conventional version control systems.

3. REVISION CLASSES

In this section, we first present revision classes. We then introduce the language extension we have designed for Java and finally discuss our compiler.

3.1 Concept

Revision classes introduce an explicit versioning mechanism for classes. The concept of revision classes is similar to the concept of inheritance: A subclass extends a superclass and specifies the differences to the superclass by overriding existing methods or adding additional methods. A *revision class* extends (i.e., *revises*) another revision class and specifies the differences to the revised class. Similar to a subclass, a revision class cannot stand on its own and always relates to the revised class.

A *revision class* has an explicit version number. The revision class that defines the initial version is referred to as the *base revision class*. A base revision class can be defined in any version and never revises another revision class. A base revision class always conforms to a conventional class. A revision class can *revise* another revision class. Revision classes with the same name are referred to as a *series of related revision classes*.

Figure 1 illustrates the presented concept. The *vertical axis* denotes the inheritance and thus the subtyping relationship. The *horizontal axis* describes the development over time and thus the versioning relationship. There are two base revision classes in Version 1: P#1, Q#1, with Q extending P. In Version 2, a new base revision class R#2 is added that extends Q. Revision class P#2 revises P#1. In Version 3, all existing revision classes are revised.

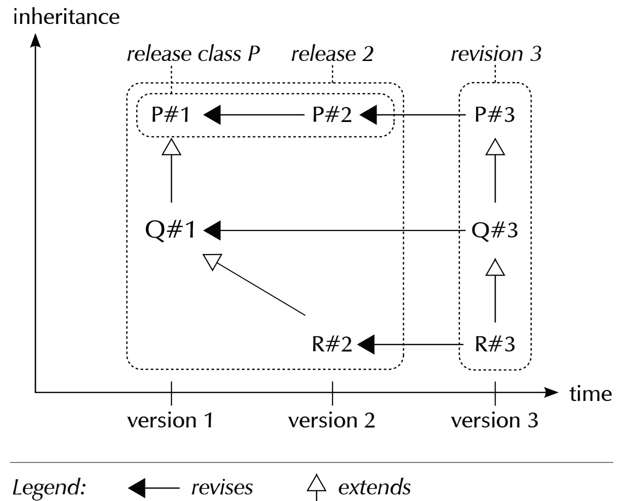

Figure 1: The concept of revision classes.

We distinguish between revisions and releases. A *revision* includes a set of revision classes that describe some update relative to the previous release. In Figure 1, revision classes P#3, Q#3, and R#3 form Revision 3. A *release* includes a selected and all prior revisions. We define a *release class* all revision classes with the same name and thus a series of related revision classes. A release class *merges* all operations and field definitions and conforms conceptually to a conventional Java class. In Figure 1, release class P for Version 2 consists of revision classes P#1 and P#2.

Inheritance enables the dynamic binding of methods, i.e., the determination of the correct variant of a method depending on the type of the receiver object at run-time. For a given release class, all methods of the previous revision classes are merged to yield the latest version of every method. Figure 2 shows some examples when combining revisioning with inheritance. Revision classes written in italics indicate that the existence of the respective revision class is irrelevant for the result. In Figure 2(a), always Revision 2 is selected depending on the target type. In Figure 2(b), Revision 1 is selected for target types C and Revision 2 for target types S. In Figure 2(c), Revision 2 is selected for C and Revision 1 for S.

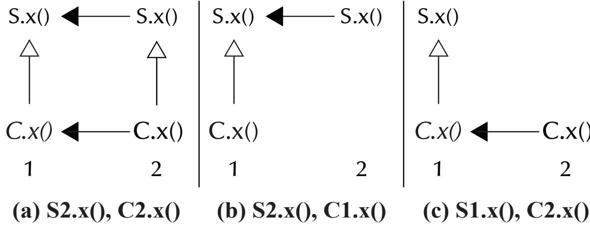

Figure 2: Method dispatch in revision classes.

We have extended the traditional method dispatching algorithm for single inheritance to include revision classes. The algorithm seamlessly includes support for revision classes for a given program version n. Inspired by the notation of Bracha and Cook [3], we define the actual methods for a release class as follows:

$$S(1) = \Delta(S, 1) \quad (1)$$

$$S(n) = \Delta(S, n) \oplus S(n-1) \quad (2)$$

$$C(1) = \Delta(C \rightarrowtriangle S, 1) \oplus S(1) \quad (3)$$

$$C(n) = \Delta(C \rightarrowtriangle S, n) \oplus \Delta(C \rightarrowtriangle S, n-1) \oplus S(n) \quad (4)$$

The binary $\oplus$ operator takes the value from the left argument in case the same method is present in both classes. In this way, the precedence is given to the previous revision class and then to the superclass.

(1) The methods for a release class S of the first version are the methods as specified in the base revision class.

(2) The methods for a release class S in a version n are the methods of the revision class of version n combined with the methods of the previous revisions.

(3) The methods for a release class C with a superclass S in the first version are the methods of the base revision class C combined with the methods of the superclass of the first version.

(4) The methods for a release class C with a superclass S are the methods of the revision class C for version n combined with the methods of the previous revisions and the methods of the superclass.

The concept of revision classes approximates the changes subclasses can specify. Revision classes add new methods and fields, redefine methods, and implement additional interfaces. Every revision class represents a more recent version of its revised class. With revision classes, we can model updates in a similar way as specializations in subclasses. We have extended the dynamic method dispatching algorithm to take into account revision classes. Depending on the required version *and* the type of an object, the proper version of a method can be chosen at run-time.

3.2 Language design

In the following, we present the extension we have designed for the Java language. We describe the language extension with a concrete example that consists of two versions with each three classes: a superclass, a subclass, and a main class. Figure 3 shows the class diagram. Version 1 declares three base revision classes Super, Sub, and Main. Sub contains a method printHi() and two String fields, an id and a msg that are initialized to some values. Main defines the main method. In Version 2, all base revision classes are revised: Super adds a new method printBye(). Sub redefines the initial value of field id and the implementation of method printHi(). Also Main redefines the implementation of the main method.

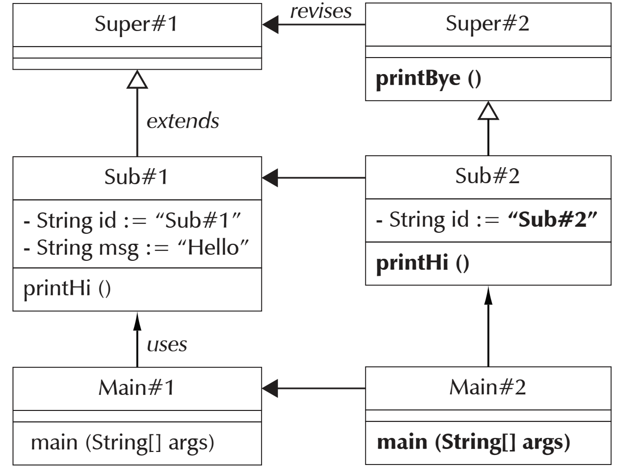

Figure 3: Example class diagram.

Revision classes are recognized with a special keyword (i.e., revisionclass). Using a separate keyword provides several advantages. First, a distinct language keyword stresses the fact that revision classes depend on the respective revised classes. Second, the concepts of conventional Java classes and revision classes are cleanly separated, thereby also enabling the usage of both constructs in the same program. As a consequence, existing libraries do not have to be ported to revision classes. Third, the parser can recognize revision classes, which simplifies the separate handling of conventional and revision classes.

Figure 4 lists the source code for the first and the second revision. In the first version, all base revision classes are declared. Apart from the special keyword revisionclass and the explicit version number, the base revision classes conform to conventional Java classes. Revision class Sub#1 extends revision class Super#1. Sub#1 contains two String fields, an id and a msg that are initialized to some values. Sub#1 declares a method printHi() that outputs the fields. Main#1 defines a local variable of type Sub, instantiates Sub, and calls the method printHi().

```
revisionclass Super#1 {
}

revisionclass Sub#1 extends Super {
    private String id = "Sub#1: ";
    private String msg = "Hi!";
    public void printHi() {
        System.out.println(id + msg);
    }
}

revisionclass Main#1 {
    public static void main(String[] args) {
        Sub s = new Sub();
        s.printHi();
    }
}
```

Output: Sub#1: Hi!

```
revisionclass Super#2 revises Super#1 {
    public void printBye() {                    // method added in version2
        System.out.println("Super#2: Bye!");
    }
}

revisionclass Sub#2 revises Sub#1 extends Super{
    private String id = "Sub#2: ";              // redefine private member

    public void printHi() {
        previous;                               // inline code of previous revision
        System.out.println(id + msg);           // access private member
    }
}

revisionclass Main#2 revises Main#1 {
    public static void main(String[] args) {
        Sub s = new Sub();
        s.printHi();                            // callable for versions> 1
        s.printBye();
    }
}
```

Output: Sub#2: Hi!
Sub#2: Hi!
Super#2: Bye!

Figure 4: Programming with revision classes.

In the second version, all base revision classes are revised. Super#2 introduces a method printBye() that consequently is to be inherited by its subclass Sub#2. Sub#2 redefines the initial value of id and changes printHi() using previous to execute the code of its old version. Main#2 does not only say hello, but also says good bye. The example shows important properties of our language extension:

Only release classes provide types. Examples are the extends clause in revision class Sub#1 or the declaration of the local variable s of type Sub in revision class Main#2. As the developer uses release classes (e.g., Sub) and not revision classes (e.g., Sub#1), the developer does not need to track version numbers of revision classes. This typing scheme also enables the coexistence of revision classes and conventional classes.

Revision classes can add/redefine methods. Revision classes define only the delta to the previous class version. Declaring a method that does not exist in a new version semantically *adds* the method to the class (e.g., printBye() in Super#2). The compiler ensures that the method is only called starting from the version that introduces the method. Declaring a method with the same signature as in an earlier revision class *redefines* the code of the method of the old revision. As also the implementation of private methods should be changeable, we have decided that revision classes can also redefine (as well as access) private methods. If the signature is changed, the method is considered to be overloaded and thus added to the new version.

Revision classes redefine fields. The compiler allows to redefine the initial value of fields (e.g., id is redefined), also for private fields. We have not yet implemented type changes, as the semantic analysis must ensure that all methods that use a redefined field are indeed redefined to reflect the type change.

Statement previous inlines the revised method. The statement is similar to the Precursor keyword in Eiffel [8], the special statement previous allows to access the original code of the revised method (see printHi() in codeSub#1). As the compiler inlines the original code, we have chosen not to provide a previous() method that also takes arguments.

3.3 Compiler

We have integrated the language extension in the Espresso Java 1.1 compiler [12]. We call our compiler *Crema* because the support for revision classes is realized as a layer on top of Espresso. Including revision classes in the compiler allows us to perform semantic analysis for revision classes and thus to detect invalid changes or usages of revision classes. For example, a method defined in a new revision must not be used in an older version.

The compiler *Crema* takes as input a set of revision classes (and optionally conventional classes) and the requested version number. *Crema* generates Java bytecode for both revision *and* release classes. The bytecode of *revision classes* conforms to the source code of the revision classes and thus contains only the differences to the revised class. Revision classfiles may be loaded by a custom class loader that effects the merging process of our compiler. The bytecode for *release classes* conform to a conventional classes and can be executed on any standard Java virtual machine [7].

The analysis of *Crema* ensures the semantic correctness of revision classes. For example, access flags of method redefinitions must not vary over the versions. The compiler

Table 1: Evolution of JDNSS. Numbers in parenthesis list classes (fields, methods) added or removed to the given version.

Version	Classes	Fields	Methods
1.0	16	83	114
1.1	16 (+2,-2)	91 (+9,-1)	116 (+7,-9)
1.2	19 (+3)	94 (+6,-3)	136 (+10,-8)

Table 2: Updates in JDNSS. Legend: +: additions, −: removals, Args: argument types.

Versions	Classes		Fields			Methods			
	+	−	+	−	Type	+	−	Args	Body
1.0→1.1	2	2	9	1	0	7	9	4	25
1.1→1.2	3	0	6	3	9	10	8	0	28

merges all related revision classes to yield release classes, thereby updating the member redefinitions and additions. A release class consists of all revision classes from the first to the last revision.

Our plans for future work include adding support for revision classes to the Java VM. A revision class-enabled JVM may load new revision classes at run time and thus achieve dynamic software updating [4].

4. EVALUATION

We have performed a case study to remodel an existing Java application to revision classes using three versions of JDNSS, a Java DNS server (version 1.0, 1.1, 1.2) [13]. Table 1 shows the evolution of JDNSS in terms of classes, fields, and methods per version. The first program version consists of 16 classes, 83 fields, and 114 methods. The numbers in parenthesis also indicates the number of classes added (i.e., +) or removed (i.e., −) from one version to the next.

In Version 1.1, a central logging mechanism is introduced (i.e., Java standard library logging). Classes declare a logger field, and method bodies are adapted to employ the logging functionality. Furthermore, some methods are removed and some parameters are changed. In Version 1.2, the Java logger is replaced with a custom logger implementation. Table 2 shows the number of classes (fields, methods) affected from these updates: 9 fields are added (java/lang/Logger) in version 1.1, and the type of these fields is changed to MyLogger in Version 1.2. The changes of the method bodies (25 in version 1.1, 28 in version 1.2) are mostly because of calls to the logging library. The addition of 7 (respectively 10) methods can be ascribed to refactoring of the code.

Revision classes can declare a well-defined set of updates: Classes (fields, methods) can be added, the implementation of methods can be redefined, the initial value of fields, direct superclasses, and more interfaces can be implemented. Other updates must be composed from the simple ones. For example, methods cannot be explicitly removed. To handle a method removal, we redefine the original method. If the method overrides an inherited method from a parent class, the new version of the method performs a supercall, thus effectively eliminating the overridden method. If there is no superclass, the method is redefined to throw a java/lang/NoSuchMethodError. Changes to argument types of a method include the addition, exchange, or removal of argument types. Such changes can be handled as a combination of method removal and addition. First, the original method is updated as "removed" (as described before) and then, a new method is declared with the new argument types. We modeled the type changes of fields as the addition of a new field, while ensuring that the old field in not accessed anymore. As we combined the small set of changes, we could model all updates of both version steps of JDNSS.

We have never used the previous statement; previous is suitable when additional code is prepended or appended to the original version. Instead, we found most changes in the middle of the code and thus could not apply previous.

Using revision classes for explicit versioning in classes reduces the code size. As revision classes declare the delta to the previous class version, code duplication is avoided. By comparing the different source code sizes, we achieve a reduction of around 60%. We could reach around 50% reduction if the language supported type changes of fields. The more program versions are added, the less code is added or redefined and thus the code sizes of revision classes becomes smaller with every version. In addition, code understanding and debugging is facilitated, as the changes are explicitly specified.

Table 3: Code size (lines of code) of the different JDNSS versions.

Version	Crema compliant	Revision class	Revision class w/type
1.0	2188	2188	2188
1.1	2293	1000	1000
1.2	2471	1120	558
Total	6952	4308	3746
Rel.	100%	62%	54%

The case study has shown that all changes can be explicitly stated and ideally, more complex transformations should be supported. If not only classes, but also methods and fields are explicitly revised, the evolution of single language elements becomes explicit and also deletions can be handled directly.

5. DISCUSSION

The revisioning mechanism explicitly defines changes over time. Analogously to inheritance, extensions (i.e., additions of new fields and methods) and specializations (i.e., redefinitions of fields and methods) can be defined. The developer specifies necessary changes explicitly by providing a delta to a previous version. Explicit versioning has several advantages: Revision classes define explicit changes, and thus errors can be localized and attributed to a particular version. The code size of classes is reduced over time, and thus redundancy as a possible source of errors is avoided. Revision classes provide a basis for a more conscious way of programming, as the developer must explicitly declare where the code is adapted.

Revision classes may add more complexity in code understanding because of the higher number of files. Adequate tool support is necessary: An IDE could provide different views on revision classes. For example, the last full release

can be displayed that shows only the latest class versions, or the concept of browsing can be extended to allow walking back in time while highlighting the changes.

The methods for a given release are determined with a simple algorithm that extends the method calculation of inheritance. Currently, our compiler uses the algorithm to statically determine the methods of a given release. In principal, the algorithm enables to dispatch revision class methods dynamically. As a consequence, we can envision a virtual machine with direct support for revision classes. Such a virtual machine could load revision classes at run-time and thus provide for seamless updating of the executing program at run-time.

The case study has shown that all changes in JDNSS could be modeled with the changing possibilities offered by revision classes. Nevertheless, more elaborate revisioning mechanisms would be necessary, e.g., the possibility to explicitly delete fields and methods. Currently we are investigating how to express even more changes with an explicit class versioning mechanism.

6. CONCLUDING REMARKS

We have presented the concept of revision classes. Revision classes allow the developer to explicitly version classes and thus to explicitly describe the difference to the previous class version. Current version control systems employ file versioning, and thus the semantic information is decoupled from classes. By including a versioning mechanism in classes and not in files, we hope that the developer may consider the necessity and the effect of changes. As a consequence, programming errors that sneak into the code while fixing another bug or adding another functionality may be avoided.

7. ACKNOWLEDGEMENTS

This work was funded, in part, by the NCCR “Mobile Information and Communication Systems”, a research program of the Swiss National Science Foundation.

8. REFERENCES

[1] A. Bergel, S. Ducasse, and O. Nierstrasz. Classbox/J: Controlling the Scope of Change in Java. In *Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 177–189, 2005.

[2] G. Bierman, M. Parkinson, and J. Noble. UpgradeJ: Incremental Typechecking for Class Upgrades. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, pages 235–259, 2008.

[3] G. Bracha and W. Cook. Mixin-based Inheritance. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems Languages and Applications, European Conference on Object-Oriented Programming (OOPSLA/ECOOP)*, pages 303–311, 1990.

[4] S. Cech Previtali. *Dynamic Updating of Object-Oriented Software Systems based on Aspects.* PhD thesis, Laboratory for Software Technology, Department of Computer Science, ETH Zurich, 2009.

[5] M. Denker, T. Gîrba, A. Lienhard, O. Nierstrasz, L. Renggli, and P. Zumkehr. Encapsulating and Exploiting Change with Changeboxes. In *Proceedings of the International Conference on Dynamic Languages (ICDL)*, pages 25–49, 2007.

[6] E. Ernst. Family Polymorphism. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, pages 303–326, 2001.

[7] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification.* Addison-Wesley Publishing Company, Inc., Reading, Massachusetts, 2nd edition, 1999.

[8] B. Meyer. *Object-Oriented Software Construction.* Prentice Hall, 2nd edition, 1997.

[9] M. Mezini and K. Ostermann. Integrating Independent Components with On-demand Remodularization. In *Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 52–67, 2002.

[10] M. Mezini and K. Ostermann. Conquering Aspects with Caesar. In *Proceedings of the International Conference on Aspect-Oriented Software Development (AOSD)*, pages 90–99, 2003.

[11] R. Robbes and M. Lanza. A Change-based Approach to Software Evolution. *Electronic Notes in Theoretical Computer Science (ENTCS)*, 166:93–109, 2007.

[12] Java Espresso compiler. *http://types.bu.edu/Espresso/JavaEspresso.html.*

[13] JDNSS—Java DNS Server. *http://jdnss.sourceforge.net/.*