Supporting Software Development Tools with An Awareness of Transparent Program Transformations

Myoungkyu Song

Thesis submitted to the faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Master of Science
in
Computer Science and Applications

Eli Tilevich (Chair)
Barbara G. Ryder
Marc R. Fisher

April 24, 2013
Blacksburg, Virginia

Keywords: domain-specific languages, enhancement, program transformation, bytecode engineering, debugging

Supporting Effective Reuse and Safe Evolution in Metadata-Driven Software Development

Myoungkyu Song

ABSTRACT

Programs written in managed languages are compiled to a platform-independent intermediate representation, such as Java bytecode. The relative high level of Java bytecode has engendered a widespread practice of changing the bytecode directly, without modifying the maintained version of the source code. This practice, called *bytecode engineering* or *enhancement*, has become indispensable in transparently introducing various concerns, including persistence, distribution, and security. For example, transparent persistence architectures help avoid the entanglement of business and persistence logic in the source code by changing the bytecode directly to synchronize objects with stable storage. With functionality added directly at the bytecode level, the source code reflects only partial semantics of the program. Specifically, the programmer can neither ascertain the program's runtime behavior by browsing its source code, nor map the runtime behavior back to the original source code.

This research presents an approach that improves the utility of source-level programming tools by providing enhancement specifications written in a domain-specific language. By interpreting the specifications, a source-level programming tool can gain an awareness of the bytecode enhancements and improve its precision and usability. We demonstrate the applicability of our approach by making a source code editor and a symbolic debugger enhancements-aware.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Managed languages, including Java and C#, reduce the complexity of software construction by providing portability, type safety, and automated memory management. A Gartner report predicts that by 2010, as much as 80% of new software will be written in C# or Java [FL05]. Another reason for the widespread popularity of managed languages is that they feature multiple standard libraries and portable frameworks, whose use improves programmer productivity. In fact, it has been observed that the third-party libraries and frameworks of a typical commercial enterprise application commonly constitute the majority of the codebase [DRS07].

Object-oriented frameworks have become an integral part of enterprise software development, as their reusable designs and predefined architectures streamline the software construction process. A major draw of modern enterprise frameworks is that the developer can write business logic components using a Plain Old Object Model (e.g., Plain Old Java Objects (POJOs) and Plain Old Common Language Runtime Objects (POCOs)), business-level application objects that do not implement special interfaces or call framework API methods. Among Plain Old Object Models, POJO-based frameworks have become mainstream in the enterprise computing community, as they improve separation of concerns, speed up development, and improve portability [Ric06].

To provide services to application objects, a framework employs bytecode engineering to enhance their intermediate representation (i.e, bytecode or CLR), commonly at runtime. Figure 1.1 demonstrates the main steps of such framework-based development. First, the source code is compiled to an intermediate representation. Then an enhancer uses bytecode engineering [Dah99] to add new functionality to the compiled bytecode as guided by the corresponding custom metadata. Each framework uses different metadata formats, commonly expressed using XML files or Java 5 anno-

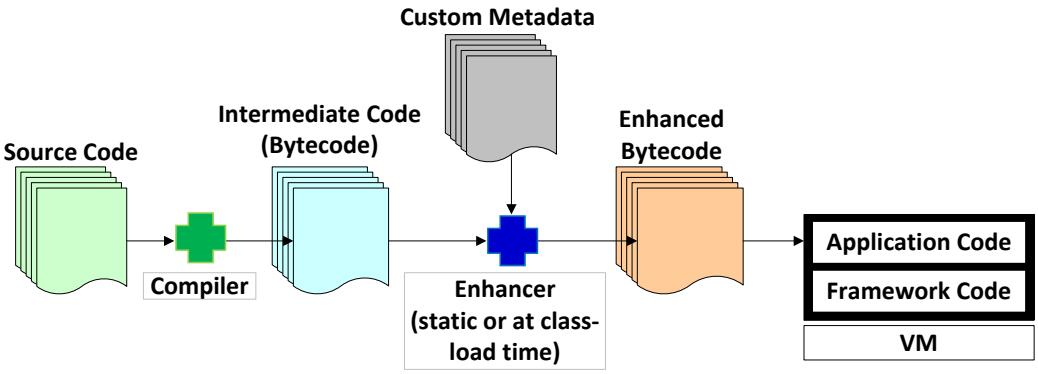

Figure 1.1: Enhancing intermediate code via bytecode engineering.

tations, to mark framework-related program constructs. Further, the enhancer can run as a separate tool or be integrated with a class loader. Finally, the enhanced bytecode, which differs in functionality from the original source code, is executed by the framework.

Although intermediate code enhancement has entered the mainstream of enterprise software development due to the widespread use of frameworks, no standard technique has been introduced to capture and document the enhancements. Metadata guiding the enhancement process not only is custom for each framework, but also specifies *what* program constructs are relevant for a particular framework (e.g., which fields are persistent) rather than *how* the intermediate code should be enhanced. As a result, the enhancements are, at best, documented through an informal narrative (i.e., documentation) or treated as a black box.

Lacking any formal description of bytecode enhancements, source code does not faithfully reflect the full semantics of an enhanced program. For source-level programming tools, this inability to express all the functionality of a program in source code leads to reducing the utility of those programming tools that rely on the one-to-one correspondence between the running version of a program and its source-level representation. The presence of bytecode enhancements not only hinders the ability to understand the real behavior of a program by browsing its source code, but also makes it non-trivial to map the execution of a program to its source code. The programmer

may need to understand the observed behavior of a program, while relating the observed behavior of the enhanced intermediate code back to the original source and vice versa.

Although optimizing compilers have long changed intermediate representations to improve performance, the techniques developed for dealing with such optimized code (i.e., debugging optimized code) are unsuitable for dealing with enhanced bytecode. While optimizations are always semantics-preserving transformations (sometimes under certain input), enhancements change the semantics of a program in custom and difficult-to-generalize ways.

The optional Java class file's attribute `LineNumberTable` provides little value as a mechanism for mapping enhanced bytecode to the original source code. An enhancer cannot adjust the `LineNumberTable` of an enhanced class, as the enhancements are expressed only in bytecode and have no representation in source code.

Even though Aspect Oriented Programming (AOP) [KLM+97] is often used for introducing cross-cutting concerns, the AOP languages such as AspectJ [KHH+01] do not capture all the enhancements commonly introduced directly at the bytecode level, which include changing program construct names, removing program constructs, and generating new classes.

This paper presents an approach that improves the utility and precision of source-level programming tools by capturing and documenting bytecode enhancements. First, we have classified commonly-used bytecode enhancements by examining the API of two widely-used bytecode engineering toolkits and by reverse-engineering the enhancements performed by two commercial enterprise frameworks. Based on our classification, we have created a declarative, domain-specific language for describing bytecode enhancements. To demonstrate the expressiveness of our language, we used it to describe the enhancements performed by several industrial and research systems that use bytecode engineering. Finally, we have used our approach to make a source code editor and a symbolic debugger enhancement-aware, thereby improving their precision and utility.

1.1 Research Agenda

Since enterprise software development has already relyed on bytecode enhancement, emerging enhancers are more likely to transform programs in even more complex ways. As a result, programmers have to understand source code with less sufficient in the presence of a practical view about the program execution behavior. Our approach of enabling source-level programming tools aware of bytecode enhancements can address this problem, and make for programmers it possible to develop the programs transparently transformed with full advantage of its functionalities. This thesis studies about several problems of transparent program transformations, with emphasis on enhancement specifications. This work examines why the existing state of the art in source-level development poorly support tools in programming and debugging programs transparently transformed at the bytecode level; it then proposes novel solution to these problem.

How are software development and maintenance improved in enterprise software development?

In enterprise software development, the framework enhancer commonly adds the required functionalities to the bytecode of a program, a practice that occurs in compile time or runtime execution whose semantics changes against that represented by the source code. Therefore, the programmer cannot directly aware of the program execution by reading its source code, nor associate its functionality with the source code.

1.2 Major Research Contributions

In the followings, we present an outline of the major contributions made by this thesis, especially an approach to understanding transparent bytecode transformations.

- A novel approach to improving the precision and utility of source-level programming tools

in the presence of intermediate code enhancements.

- The Structural Enhancement Rule (SER) language, a Domain-Specific Language (DSL) for concisely expressing structural enhancements that modern enterprise frameworks commonly apply to intermediate code.
- A debugging architecture that enables symbolic debugging of programs whose intermediate code has been transparently enhanced.

1.3 Broader Impacts

This research can help software developers who can be equipped with powerful software methodologies, techniques, and tools for understanding transparent program transformation. The benefits include (1) increasing programmer productivity, as the efforts expended on authoring programs by the enhancements-aware code editor can be significantly reduced, and (2) reducing enhanced programs debugging effort, as source-level debugging of enhanced program's code can be available by making a symbolic debugger aware of the enhancement.

1.4 Structure of This Document

The rest of this thesis is structured as follows. Chapter 2 compares this research with the existing state of the art. Chapters 3 and 4 covers motivations, design, and implementation of Structural Enhancement Rule and its utility of source-level programming tools. Chapter 5 reflects on our experiences of implementing and evaluating this research through case studies. Chapter 6 presents concluding remarks, the contributions of this research, and discuses future work directions.

Chapter 2

Literature Review

The purpose of this chapter is to put the research described in this thesis into perspective by demonstrating how it relates to existing work and practice. First, we discuss directly related work as pertaining to each of the components explored by this thesis. Then we clarify how this work on understanding of transparent program transformation utilizes approaches and techniques from different research areas. Finally, we summarize how this work could benefit or influence existing areas of research.

Our approach to augmenting source-level programming tools with an awareness of intermediate code enhancements is related to several research areas including the debugging of transformed code, program transformation languages, and structural program differencing.

2.1 Debugging Transformed Code

Debugging Optimized Code. Modern compilers have powerful code optimization capabilities. Compilers optimize code via performance-improving transformations. However, because compilers transform an intermediate representation of a program, the relationship between such a transformed representation and the original source code of a program becomes obscured. Specifically, performance-improving transformations reorder and delete existing code as well as insert new code. Thus, debugging optimized code is hindered by the changes in data values and code location. A significant amount of research aims at the problem of debugging optimized code

[Hen82, HCU92, BHS92, KCS05, Fai98, Cop94, Fri83]. The proposed solutions enable source-level debuggers to display the information about an optimized program, as if the original (unoptimized) version of the code was being debugged.

The techniques for debugging optimized code deal with the challenges arising as a result of optimizing compilers transforming intermediate code to improve performance. While these transformations can be quite extensive, they are usually confined to method bodies and do not alter the debugged program's semantics. By contrast, structural enhancement aims at larger-scale program transformations that can add new classes and interfaces to a program. Thus, source level debugging of structurally enhanced programs requires different debugging techniques such as the presented symbolic undo.

Debugging for AOP. Aspect-Oriented Programming (AOP) [KLM+97] can be viewed as an enhancement technology, and several approaches aim at debugging support for AOP [EAH+07, PT08, IKI04]. However, debugging aspect-oriented programs is different from debugging transparently enhanced programs. AOP provides a domain-specific language for programming enhancements such as AspectJ [KHH+01], and an AOP debugger traces the bytecode generated by AspectJ to its aspect source file. By contrast, transparent bytecode enhancements have no source code representation. Thus, the approaches to debugging AOP software cannot be applied to debugging transparently enhanced programs.

In addition, AspectJ as a language cannot express all the structural enhancement transformations. For example, it is impossible to express in AspectJ that the name of a program construct (e.g., class, field, or method) be changed. AspectJ does not provide facilities for removing a program construct, something that program enhancers need to do on a regular basis. For example, the split class enhancement moves fields from a class to another class, thus removing them from the original class. Finally, program enhancers often generate new classes and interfaces and reference them in the enhanced program. AspectJ is not designed for expressing how to generate a new class, whose

structure is based on some existing program construct. Thus, we could not have used AspectJ instead of SER to represent structural enhancements.

2.2 Program Transformation Languages

SER is a domain-specific language for expressing how bytecode is enhanced structurally. Other domains also feature domain-specific languages for expressing program transformations.

JunGL [VEdM06], a domain-specific language for refactoring, combines functional and logic query language idioms. JunGL represents programs as graphs and manipulates refactorings via graph transformations. The language provides predicates to facilitate the querying of graph structures. JunGL uses demand-driven evaluation to prevent scripts from becoming prohibitively complex.

Sittampalam *et al.* [SdML04] specify program transformations declaratively and generate executable program transformers from specifications. The Prolog language is augmented with facilities for incremental evaluation of regular path queries. The augmented language is used to specify program transformations by expressing a program and its transformed counterpart. The transformations can be combined with a strategy script, based on Stratego [VeABT99], to specify the traversals of a program and the order of the transformations.

Whitfield and Soffa's code-improving transformation framework consists of a case tool and a specification language [WS97]. The specification language, called Gospel, declares program transformations; the case tool, called Genesis, generates a program transformer given a Gospel specification. A Gospel script consists of a declaration section, containing variables declarations, a precondition section, containing code pattern descriptions and control dependence conditions, and an action section, containing a set of transformation operations.

The Coccinelle [SHL+07] tool introduces the SmPL language for locating and automatically fixing bugs. SmPL programs specify how in response to some runtime condition, a program should be transformed to fix a bug and log the changes.

FSMLs [Cza06] is a domain specific modeling language for describing the framework-provided knowledge, including framework instantiation, procedures for implementing interfaces, and proper usage of framework services. FSMLs bears similarity to our approach in its ability to provide a bi-directional mapping between framework features and their abstract representation.

JAVACOP [ANMM06] introduces a declarative rule language for expressing programmer-defined types, called *pluggable types*. The types are described as user-defined rules, which JAVACOP uses for transforming the abstract syntax tree of a program.

Compile-Time Reflection (CTR) [FCL06] enables generative programming without the intricacies of the reflection API in the mainstream managed languages such as Java or C#. To that end, CTR extends C# with the ability to inspect and generate code using templates and pattern matching. CTR generates code at compile time, thus ensuring that the generated code is statically checked. The purpose of CTR is to express how source code should be generated, rather than how bytecode should be enhanced.

Because these program transformation languages were designed specifically for their respective domains, we could not have used any of them for our purposes. The main design goal of our SER language was to be able to express structural enhancements used by bytecode enhancers declaratively and to provide special constructs for domain-specific transformations such as adding getter/setter methods.

2.3 Program Differencing

SER scripts describe generalized structural program differences. Program differencing is an active research area and several new differencing algorithms have been proposed recently.

Previtali *et al.*'s technique [PG06] generates version differences of a class at bytecode level. Their algorithm produces the information on added, removed, or modified classes. Dmitriev incorporates program change history into a Java make utility that selectively recompiles dependent source files [Dmi02]. The JDIFF [AOH04] algorithm identifies changes between two versions of an object-oriented program using an augmented representation of a control-flow graph. M. Kim *et al.* [KNG07] infer generalized structural changes at or above the level of a method header, represented as first-order relational logic rules. These techniques could be leveraged to generate SER scripts automatically by generalizing the differences between the original and enhanced versions of multiple classes.

Our own Rosemari system [TT08] generalizes structural differences between two versions of a representative example. Such examples are usually supplied by framework vendors to guide the developers in upgrading their legacy applications from one framework version to another. Rosemari features a DSL for describing program transformations, but can be retargeted to present structural changes in SER instead.

2.4 Symbolic Execution

The idea of symbolic undo, used in implementing our debugger, is influenced by symbolic execution [Kin76], which analyzes a program by executing it with symbolic inputs, but without actually running it. By analogy, our symbolic undo technique enables source level debugging of enhanced

bytecode by mapping it back, via symbolic operations, to its original source code, also without affecting the program's execution.

Chapter 3

Background

In modern enterprise software development, the programmer expresses business logic components using application classes that do not inherit from special framework types or have any other functionality besides business logic. Then extra functionality is added to the intermediate code[1] of the application classes via bytecode enhancement to enable enterprise frameworks to provide services, thereby implementing various concerns, including persistence, transactions, and security. This development model relieves the programmer from the burden of having to implement these important concerns by hand. The programmer simply uses metadata (such as XML files or Java 5 annotations) to designate specific application objects as interacting with a framework, and all the tedious details of enabling the interaction happen entirely behind the scenes. Despite the convenience afforded by the use of such intermediate code enhancement, as we demonstrate next, its use compromises the utility and precision of source-level programming tools.

3.1 Transparent Persistence Frameworks

The following example comes from the domain of transparent persistence, which is used by several persistence frameworks in industrial software development, including Hibernate [BKN05] and JDO [Rus07]. A transparent persistence architecture combines features of both orthogonally persistent languages [ADJ+96, LAC+96, MBMZ01, Atk88] and data access libraries, such as Java Database Connectivity (JDBC)[Sunc] and Open Database Connectivity (ODBC)[Mic].

[1] In the rest of the manuscript, we use the terms *intermediate code* and *bytecode* interchangeably.

When program data outlives the program's execution, the data is said to be *persistent*. Transparent persistence architectures provide a software framework for managing the program data marked as persistent by the programmer. The management entails synchronizing the persistent data and its stable storage representation.

A representative of a transparent persistence infrastructure for Java is Java Data Objects (JDO) [Rus07]. JDO uses static post-compile enhancement to enable Java objects with persistence capabilities. The programmer writes Java objects to be persisted as regular Java Beans [Sunb]. A separate JDO metafile (in XML) specifies which fields of a class should be persisted and how they map to stable storage. Finally, as specified by the metadata, the JDO enhancer adds persistence-specific methods and fields to each persistent class, enabling its instances to interact with the JDO runtime.

In particular, the enhancer changes a persistent class to implement interface `PersistenceCapable` and wraps all read and write accesses to a persistent field with special methods that interact with the JDO runtime. Thus, before the value of a persistent field is retrieved or modified, an appropriate JDO-specific action is triggered, thereby ensuring that a fresh copy of the data is retrieved from stable storage and all the changes in the application space are properly persisted.

The design of JDO satisfies the stated goal of introducing persistent capabilities transparently. JDO enables rank and file programmers to focus on *what* data is being persisted and treating *how* the data is persisted as a black box. Enterprise programmers may be aware that some bytecode enhancement is taking place, but the specific enhancements are not relevant to their primary concern—expressing the required business logic.

3.2 Example: Mortgage Authorization Application

Consider a mortgage authorization application used by a bank to calculate the maximum amount of mortgage eligibility, according to a set of business rules that use a customer's salary and credit score. The application uses several transparently persistent classes, including `SalaryLevel` and `CreditLevel`, whose objects are persisted in a relational database such as MySQL or Oracle using the JDO framework.

Consider the code listing in Figure 3.1, showing a method `displayMaxMortgageEligibility`. The method displays the amount of maximum mortgage eligibility given a display object and a projected salary increase amount. As is usually the case, the method manipulates different concerns of the application: business logic and graphical user interface. Assume that the GUI part of the method contains a bug: method `getMortgageField` returns `null`, thereby causing a `NullPointerException` to be thrown in the next statement. Although the bug is in the GUI logic of the method and has nothing to do with persistence or enhancements, the JDO bytecode enhancements complicate source level debugging of the code. As an illustration, consider the enhanced bytecode of the method displayed in Figure 3.2. The programmer stepping through `displayMaxMortgageEligibility` with a standard debugger will encounter the enhancements, including various new methods, including `jdoGetSalaryLevel` and `jdoGetCreditLevel`, which can be misleading, obfuscating the location of the bug. Although tracing the enhanced bytecode with a standard debugger may accidentally lead the programmer to discover a suspect bytecode instruction, matching the instruction to the corresponding statement in the original source code may quickly turn nontrivial. Bytecode-only enhancements do not have any source code level representation.

From a different perspective, a programmer who first encounters this application with the goal of adding new features or fixing a bug would not get a realistic picture of the application's behavior by

browsing the source code. In particular, the source code contains no information pertaining to the intermediate code enhancements introduced to enable transparent persistence. Thus, any change to the code could potentially lead to an unrelated change in the persistence functionality, leading to difficult-to-find errors. The metadata used to designate persistent classes only marks classes as such, but does not describe how exactly they will be enhanced.

This simple but realistic example demonstrates how transparent enhancement hinders the effectiveness of source-level programming tools. Because intermediate code enhancement has become an indispensable part of enterprise software development, new approaches are required to make source level programming tools enhancements-aware.

```
void displayMaxMortgageEligibility (Display display, double projectedIncrease) {
    double newSalary = salaryLevel.getSalary() + projectedIncrease;
    salaryLevel.setSalary(newSalary);
    double maxMortgage = calcMaxMortgage(salaryLevel,creditLevel);
    FrameField mortgageField = display.getMortgageField();
    mortgageField.setVal(maxMortgage);
    ...
}
```

Figure 3.1: Original source code.

```
void displayMaxMortgageEligibility (Display, double);
    Code:
    aload_0
    invokestatic  //jdoGetsalaryLevel;
    invokevirtual
    i2d
    dload_2
    dadd
    dstore 4
    aload_0
    invokestatic  //jdoGetsalaryLevel;
    dload  4
    invokevirtual
    aload_0
    aload_0
    invokestatic  //jdoGetsalaryLevel;
    aload_0
    invokestatic  //jdoGetcreditLevel;
     ...
```

Figure 3.2: Bytecode enhanced by the JDO enhancer.

Chapter 4

Understanding and Expressing Bytecode Enhancement

Because intermediate code is enhanced using special-purpose libraries, typically at class load time, the enhancements are poorly understood and not well-documented, if at all. The problem stems from a lack of the right expression medium for such enhancements. If intermediate code enhancements could be expressed in regular source code, there would be no need to manipulate intermediate code directly, such as with bytecode engineering. Conversely, bytecode is too low level a representation to be useful for most programming tools.

As a means of understanding and expressing intermediate code enhancements, we have introduced Structural Enhancements Rules (SER), a special purpose language for documenting bytecode enhancements. Figure 4.1 demonstrates how using SER can improve the precision and utility of source level programming tools. Specifically, the upper part of the figure shows a SER interpreter helping inform the programmer about how various program constructs will be enhanced at the bytecode level, and can be integrated with a source code editor. The lower part of the figure shows a SER interpreter being used to map the enhancements to the original source code at runtime, and can be integrated with a symbolic debugger.

Next we first explain our analysis and classification of structural enhancements. Then we describe the design and implementation of SER.

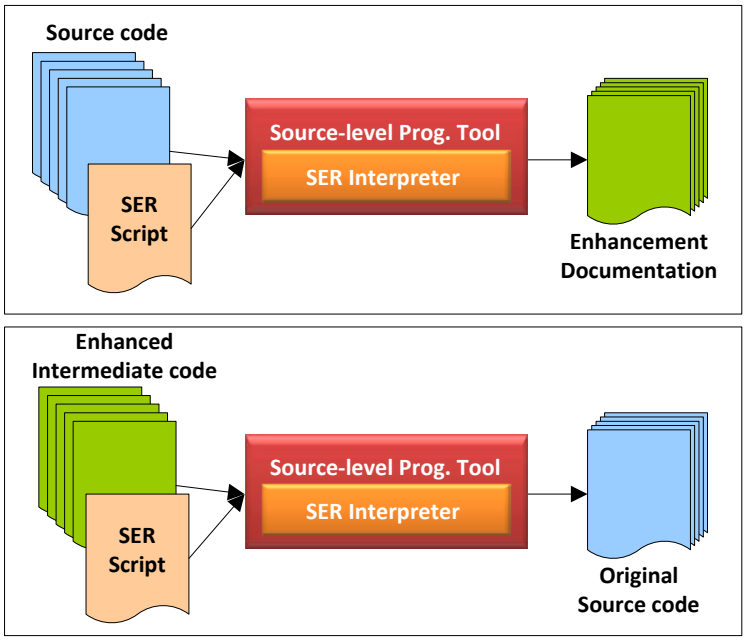

Figure 4.1: Source level programming tools with intermediate code enhancement information.

4.1 Structural Enhancements

Intermediate code enhancement is concerned with structural changes, which are large scale program transformations. The purpose of SER is to document structural enhancements in sufficient enough detail to improve the precision and utility of source-level programming tools that manipulate the original source code of an enhanced program. To ensure that SER provides the requisite facilities for expressing common intermediate code enhancements, we first catalog and classify common structural enhancements.

To determine what constitutes a structural enhancement, we have reverse-engineered several industrial and research systems that use bytecode enhancement.[1] In addition, we have also examined the capabilities of two major Java bytecode engineering libraries Apache BCEL[Apa] and Javassist

[1]Reverse-engineering these systems is perfectly legal, as they follow an open-source development model. Reverse-engineering enhanced bytecode turned out to be more effective than understanding the source code of the enhancers.

[Shi].

Structural enhancements are the subset of general program transformations that affect the structure of an object-oriented program, including classes, methods, and fields, as well as limited changes to method bodies. Structural enhancements to method bodies are primarily confined to replacing direct field accesses with setter and getter methods as well as with other wrapper methods.

A more strict definition of structural enhancement is as follows. Modify a program, confining the set of changes to the following operations:

- Adding a new class or interface.
- Changing the type of a class or an interface (i.e., changing the parent interfaces and/or classes)
- Adding a new method or field
- Removing a method or field
- Changing the signature of a method (e.g., adding or removing parameters or changing the return type).
- Changing the type of a field
- Replacing direct field accesses with setter/getter methods

4.2 Semantics of Structural Enhancements

Next we provide a more formal treatment of the program transformations that constitute the structural enhancement operations commonly applied to intermediate code.

A set of original classes, $C = \{c_1, c_2, \ldots, c_n\}$
A set of added classes, $C^+ = \{c_1^+, c_2^+, \ldots, c_n^+\}$
A set of original interfaces, $I = \{i_1, i_2, \ldots, i_n\}$
A set of added interfaces, $I^+ = \{i_1^+, i_2^+, \ldots, i_n^+\}$
A set of original methods, $M = \{m_1, m_2, \ldots, m_n\}$
A set of replicated methods, $M' = \{m'_1, m'_2, \ldots, m'_n\}$
A set of added methods, $M^+ = \{m_1^+, m_2^+, \ldots, m_n^+\}$
A set of removed methods, $M^- = \{m_1^-, m_2^-, \ldots, m_n^-\}$
A set of original fields, $F = \{f_1, f_2, \ldots, f_n\}$
A set of added fields, $F^+ = \{f_1^+, f_2^+, \ldots, f_n^+\}$
A set of removed fields, $F^- = \{f_1^-, f_2^-, \ldots, f_n^-\}$
$\langle c \rangle p$ denotes a **class** c of program p
$\langle C \rangle p$ denotes a set of classes C of program p
$\langle i \rangle c$ denotes a **interface** i implemented by **class** c
$\langle I \rangle c$ denotes a set of interfaces I implemented by **class** c
$\langle m \rangle c$ denotes a method m of **class** c
$\langle M \rangle c$ denotes a set of methods M of **class** c
$\langle f \rangle c$ denotes a field f of **class** c
$\langle F \rangle c$ denotes a set of fields F of **class** c
ext denotes **class** inheritance relationship
$impl$ denotes **interface** inheritance relationship

Table 4.1: Syntax definitions of SER

Table 4.1 lists the symbols that we use in describing the enhancement operations, with the sets of transformed program constructs appearing first. The original program consists of a set of classes. During the enhancement, new classes can be generated and added to the program. In the original program, a class can implement some interfaces. An enhancer can add new interfaces to the set of implemented interfaces and can also change the super class. Methods and fields can be added to and removed from a class. Finally, existing methods can serve as templates for other methods. We refer to this operation as "replication." For example, a new wrapper method could be created based on some existing method–the new method will have the same signature, but a different name. There are no explicit constructs for changing a field or a method–this transformation can be expressed by

$\texttt{AddClass}(\langle C\rangle p, \langle C^{+}\rangle p) = \cup_{i \in C}\langle i\rangle p \ \cup \ \cup_{j \in C^{+}}\langle j\rangle p$
$\texttt{AddSuperClass}(\langle C\rangle p\ ,\ \langle C^{+}\rangle p) = \cup_{i \in C}\langle i\rangle p \ ext \ \cup_{j \in C^{+}}\langle j\rangle p$
$\texttt{RemoveSuperClass}(\langle C\rangle p, \langle C^{-}\rangle p) = \cup_{i \in C}\langle i\rangle p \ \setminus \ \cup_{j \in C^{-}}\langle j\rangle p$
$\texttt{AddSuperInterface}(\langle I\rangle c\ ,\ \langle I^{+}\rangle c) = \cup_{i \in I}\langle i\rangle c \ impl \ \cup_{j \in I^{+}}\langle j\rangle c$
$\texttt{RemoveSuperInterface}(\langle R\rangle c, \langle R^{-}\rangle c) = \cup_{i \in R}\langle i\rangle c \ \setminus \ \cup_{j \in R^{-}}\langle j\rangle c$
$\texttt{AddMethod}(\langle M\rangle c, \langle M^{+}\rangle c) = \cup_{i \in M}\langle i\rangle c \ \cup \ \cup_{j \in M^{+}}\langle j\rangle c$
$\texttt{RemoveMethod}(\langle M\rangle c, \langle M^{-}\rangle c) = \cup_{i \in M}\langle i\rangle c \setminus \cup_{j \in M^{-}}\langle j\rangle c$
$\texttt{ReplicateMethod}(\langle M\rangle c) = \langle M\rangle c \mapsto \langle M'\rangle c = \cup_{i \in M}\langle i\rangle c \ \cup \ \cup_{j \in M'}\langle j\rangle c$
$\texttt{AddField}(\langle F\rangle c, \langle F^{+}\rangle c) = \cup_{i \in F}\langle i\rangle c \ \cup \ \cup_{j \in F^{+}}\langle j\rangle c$
$\texttt{RemoveField}(\langle F\rangle c, \langle F^{-}\rangle c) = \cup_{i \in F}\langle i\rangle c \setminus \cup_{j \in F^{-}}\langle j\rangle c$
$\textbf{Field}[Get \mid Set]\texttt{Replacer}(\langle F\rangle c) = \langle F\rangle c \mapsto \langle M'\rangle c = \cup_{i \in F}\langle i\rangle c \ \cup \ \cup_{j \in M}\langle j\rangle c \ \cup \ \cup_{k \in M'}\langle k\rangle c$

Table 4.2: Structural Enhancement Operations.

removing the old version and subsequently adding a new one.

Table 4.2 demonstrates the semantics of structural enhancement operations using set operations. Adding new program elements to existing ones is described using $\cup$, the set union operator. Removing program elements is described using $\setminus$, the set difference operator. Replicating program elements is described using $\mapsto$ and $\cup$, which designate a new element being created based on some existing element and added to the set, but the existing element still remaining in the set.

4.3 SER Language Design

The Structural Enhancement Rules (SER) language is a declarative, domain-specific language that was designed to be easy to learn, use, and understand. To show how SER can serve as an effective medium for expressing intermediate code enhancements, we introduce it by example.

```
1  Program JDO Using SUPER_JDO_SER
2  Begin
3      EnhClass.Name = OrgClass.Name
4      EnhClass.AddInterface("javax.jdo.spi.PersistenceCapable")
5      Var Pattern fieldP
6          Begin
7              modifiers = "private"
8          End
9      Var Iterator fieldIter = OrgClass.Fields(fieldP)
10
11     -- add method that'll have prefix, "jdoSet" or "jdoGet".
12     EnhClass.AddMethod(FieldSetReplacer("jdoSet", fieldIter))
13     EnhClass.AddMethod(FieldGetReplacer("jdoGet", fieldIter))
14 End
```

Figure 4.2: SER Script for the JDO Enhancement.

Describing the JDO Enhancement

Figure 4.2 shows the SER script, which documents the enhancements performed by the JDO enhancer, discussed in Section 3. The script starts with the keyword `Program` followed by the name of the script. SER scripts can use each other by means of the keyword `Using`. The body of the script is delineated by the `Begin` and `End` keywords rather than curly braces. We have deliberately made SER look different from the C family of languages. Another distinctive feature of SER is not using semicolons to terminate program statements–each statement is expected to start from a new line. Each script makes available to the programmer two reflective objects, `OrgClass` and `EnhClass`, representing the original class and the enhanced class, respectively. This design decision is guided by the observation that bytecode enhancement either modifies an existing class or generates a brand new class, using some original class as a template. In both cases, the program in the enhanced class can be expressed as a function of the corresponding constructs in the original class. Each reflective class object has built-in attributes `Name` and `Type`, which represent their name and class type (i.e., class or interface), respectively. The statement on line 3 expresses that the enhancer will modify the original class rather than generate a brand new one.

```
1  Program REMOTING
2  Begin
3      Module REMOTING_PROXY
4          Begin
5              Var Pattern fieldP
6              Begin
7              modifiers = "private"
8              type = REMOTING_IFACE.EnhClass.Name
9              name = "obj"
10         End
11     EnhClass.AddField(fieldP)
12     .......
13     End
14
15     Module REMOTING_IFACE
16     Begin
17     .......
```

Figure 4.3: SER script for the Remoting enhancement.

In addition, to the attributes, the reflective class objects make available to the programmer both accessor and modifier methods. The accessor methods of SER mirror the ones in the Java Reflection API [Gle], with two notable exceptions. First, SER accessor methods return declarative iterators, which can only be passed as parameters to SER modifier methods as we discuss later. Second, accessor methods can accept as parameters `Pattern` objects, which describe properties of program constructs. For example, the `Pattern` starting on line 5 is named `fieldP`, and it describes all the fields or methods that are `private`. The statement on line 9 uses the `fieldP` `Pattern` to obtain an `Iterator` of all the `private` fields in the original class. SER Patterns provide records describing every existing property of a program construct, including its name, type, modifiers, and signature. A `Pattern` can include any combination of records, as necessary.

The SER modifier methods can only be applied to the `EnhClass` reflective object. SER provides modifier methods for adding and deleting all the language constructs, including constructors, fields, and methods. Changing a construct can be expressed by removing it first and then adding a new construct back. The statement on line 4 documents the enhanced class modified to implement interface `PersistenceCapable`. Every modifier method can accept an iterator as a parameter,

```
1  Program Hibernate Using SuperHibernateSer
2  Begin
3      -- class name that'll have new class name containing
4      -- "EnhancerByCGLIB".
5      EnhClass.Name = OrgClass.Name + "*EnhancerByCGLIB*"
6      EnhClass.AddInterface("org.hibernate.proxy.HibernateProxy")
7      EnhClass.AddInterface("net.sf.cglib.proxy.Factory")
8      EnhClass.AddSuperclass(OrgClass.Name)
9
10     Var Pattern publicP
11         Begin
12             modifiers = "public"
13         End
14     Var Itererator methodIter = OrgClass.Methods(publicP)
15     Var Pattern nameP
16         Begin
17             name = "CGLIB*" + name
18         End
19     -- add method that'll have new method name
20     -- containing "CGLIB"
21     EnhClass.AddMethod(methodIter.CreateNewIterator(nameP))
22 End
```

Figure 4.4: SER Script for the Hibernate Enhancement.

and then every element represented by the iterator will be added. Methods and fields must be explicitly added to the `EnhClass` object, even if the enhancer modifies an existing class represented by `OrgClass`. A shorthand notation `EnhClass = OrgClass` represents copying all the language constructs of the original class to the enhanced class. This shorthand is useful when the enhanced class differs from the original class only by a small margin.

SER also provides special constructs for replacing direct field accesses with setter and getter methods. To that end, SER provides methods `FieldSetReplacer` and `FieldGetReplacer`, respectively, which can be taken as parameters to the modifier method `AddMethod`. These methods take a prefix for the getter or setter method names and an iterator representing the fields, accesses to which are replaced. The statements on lines 12 and 13 express the rule that JDO replaces direct field accesses with setter and getter methods, whose names start with “jdoSet” and “jdoGet”, respectively.

Describing the Remoting enhancement

A SER program can be comprised of multiple modules, which can refer to each other's reflective objects. A standard SER program is likely to contain multiple modules, each representing the enhancements applied to a different class.

Figure 4.3 shows a fragment of an enhancement that transforms direct references into proxy references in order to enable their execution on a remote machine. In this enhancement, used in several research systems [ORH02, OK05, TS02], for each original class, proxy, implementation, and interface classes are generated. The `Pattern` starting on line 5 refers to the type of the `EnhClass` of `REMOTING_IFACE`, which is another module in the same script. In this case, the `Name` of the `EnhClass` in the other module represents the type of the field added on line 11.

Describing the Hibernate Enhancement

As an example of a more advanced feature of SER, consider the script that appears in Figure 4.4. This description captures how Hibernate [BKN05], another widely-used commercial persistence architecture uses bytecode enhancement. Unlike JDO, Hibernate does not modify persistent classes; instead, it creates proxy classes that extend the original persistent classes, as is expressed by the statement on line 8. Furthermore, the exact name of the created proxy classes as well as the names of their methods start with hard-coded prefixes ("EnhancerByCGLIB", "CGLIB"), appended with randomly-generated integers.

To express that the names of proxy methods are based on the names of the corresponding methods in the original persistent classes, SER introduces the ability to create a derivative iterator, given an iterator and a pattern. On line 14, an `Iterator` describing the public methods in the `OrgClass` is obtained. Then on line 15, a new `Pattern` describes adding the prefix `CGLIB*` to the name of

```
1  Program SPLIT_CLASS
2  Begin
3      -- field to be split.
4      Var Pattern _tmp
5          Begin
6              name = "y"
7          End
8      Var Iterator field2SplitIter = OrgClass.Fields(_tmp)
9
10     Module SPLIT_MAIN_PARTITION
11     Begin
12         EnhClass.Name = OrgClass.Name
13         -- add all fields from the original class
14         Var fieldIter1 = OrgClass.Fields()
15         EnhClass.AddField(fieldIter1)
16         -- remove the fields that'll go to
17         -- the secondary partition.
18         EnhClass.RemoveField(field2SplitIter)
19         ...
```

Figure 4.5: SER script for 'split a class to minimize the amount of network transfer' enhancement.

a program construct. Finally, on line 21, the `AddMethod` method takes as its parameter the return value of method `CreateNewIterator`, which creates a new iterator by applying a pattern, `nameP`, to an iterator, `methodIter`. In effect, the one-liner on line 21 declaratively expresses that the public methods in the newly-generated `EnhClass` will have names that start with "CGLIB", followed by some random number, and ending with the corresponding method's name in the `OrgClass`. For example, if a persistent class has a method named `foo`, the generated proxy's corresponding method could be named `CGLIB123foo`.

In essence, SER provides programming support for manipulating sets of methods, constructors, and fields. An iterator representing a set of class constructs can be obtained based on a pattern. SER has a functional feel in that SER does not allow changing iterators, but rather makes it possible to derive new iterators by applying a pattern to an existing iterator. Also, program constructs can only be added to or removed from `EnhClass`, thus simplifying the API.

Describing the Split Class enhancement

Consider the script depicted in Figure 4.5, which describes one of the enhancements required to split a class into partitions, so that only the fields used by a remote computation be transferred across the network. This enhancement entails selecting a subset of fields of the original class and placing them into a newly created class, representing the primary partition, which will be sent across the network. In SER, the selection of the required fields is accomplished by first adding to `EnhClass` all the fields contained in the original class (Line 15). And then the fields intended for the secondary partition are removed (Line 18).

SER language summary

Table 4.3 summarizes the SER programming constructs. The language follows a minimalistic design, introducing new constructs only if necessary, with the goal of making it easier to learn and understand. SER conveys the enhancements declaratively and does not have explicit conditional or looping constructs. As a result, a SER script does not contain a sufficient level of detail to be used as input for a bytecode enhancer, but it is descriptive enough to document the enhancements for source-level programming tools.

In validating the usability of SER, we have documented four enhancements used in production and research systems: JDO [Rus07], Hibernate [BKN05], Remoting [TS02], and Split Class [CDL99, TS05]. The scripts describing these enhancements in their entirety can be downloaded from the project's website [Son]. In the discussion above, we have presented only fragments of these scripts to demonstrate various language features of SER. SER is an interpreted language, and its interpreter can be integrated into existing programming tools. We discuss the SER interpreter's design In Section 4.4, and how we used the interpreter to enhance two existing source-level programming

Program *script-name1* [**Using** *script-name2*]
Begin
[**Module** *module-name* **Begin**...**End**]
...
End
A SER script can include other scripts and can be divided into modules.

OrgClass / **EnhClass**
Reflective class objects: original and enhanced class.

Var Pattern *pattern-name*
Begin
property="*value*"
...
End
Patterns for matching program constructs, based on their properties.

Var Iterator *iterator-name*
An iterator for a collection of program constructs.

Constructors/**Methods**/**Fields** ([*pattern-name*])
Return a collection of program constructs, [possibly matching pattern-name]

Field[**Get**/**Set**]**Replacer**(*prefix, iterator-name*)
Replace direct accesses to the fields specified by *iterator-name* with *getter*/*setter* methods starting with *prefix*, and return them as an iterator

iter-name.**CreateNewIterator**(*pattern-name*)
Create a new iterator by applying *pattern-name* to *iter-name*

[**Add**|**Remove**]**Interface**/**Superclass**/**Method**/**Field** ([*iterator-name*|*pattern-name*])
Add or remove program elements specified by *iterator-name* or *pattern-name* to or from **EnhClass**

Table 4.3: SER language constructs.

tools in Sections 5.1 and 5.2.

4.4 SER language interpretation

The purpose of documenting enhancements with a SER script is to provide a bi-directional mapping between the source code of a class and its enhanced bytecode representation. The SER interpreter can take either a Java source file or a bytecode class file as parameters, corresponding to

the original or enhanced versions of a class, respectively. Another required parameter is the SER script associated with the input file. Although it would be possible for the interpreter to search for SER scripts based on the input files' names, in the current implementation it uses a configuration file that specifies which SER files work with which Java or class files. If the SER file cannot be located, the interpreter signals an error.

When processing a SER script, the interpreter parses the main script and all the included scripts specified with the `Using` keyword. Figure 4.6 demonstrates the interpreter's process flow, which differs depending on the type of the input file used to parameterize the interpreter.

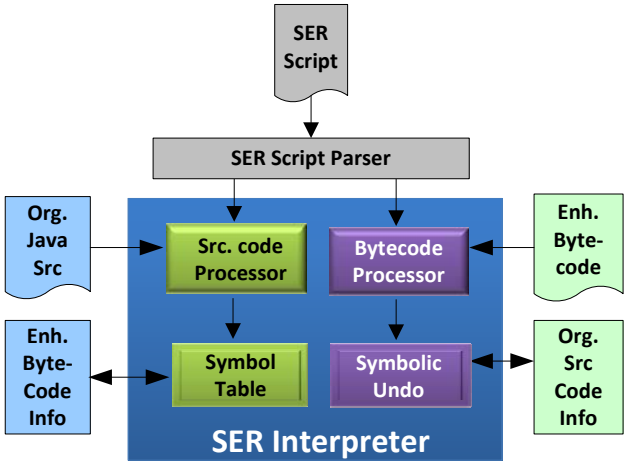

Figure 4.6: SER Interpreter.

If a Java source file, containing the original source code, is passed to the interpreter, the file is processed using the Eclipse JDT API [EF08]. Then the enhancement information processed by the SER parser is combined with the one of the original Java source. The result is stored into a symbol table module that implements a quick lookup mechanism capable of retrieving, in constant time, all the enhancements applied to a given program construct. Finally, an external API to the symbol table makes it possible to retrieve the enhancement information. In summary, the API uses JDT AST constructs as lookup keys to retrieve the enhancements associated with them. For example,

sending a class object as a parameter will return a list of lists, containing the methods, constructors, and fields that the bytecode enhancer adds to this class. Of course, some of these lists could be empty.

If a binary class file, containing the enhanced bytecode, is passed to the interpreter, the file is processed using the Javassist library [Shi]. Then the enhancement information processed by the SER parser is combined with the one of the enhanced bytecode. The result is stored in a format that we call *Symbolic Undo*, which is a collection of instructions that can be used to map every enhancement back to the original source code. We will detail the exact format of Symbolic Undo when we discuss a symbolic debugger for enhanced programs that we implemented as a case study.

Chapter 5

Applicability and Case Studies

This chapter argues that this dissertation explores algorithms, techniques, and tools for supporting software development environment that can be a valuable to increase the programmer's productivity as well as save their efforts and time.

To evaluate the applicability of this research, we have augmented two existing source-level programming tools with an awareness of bytecode enhancements. Specifically, we have added a new browsing view to a widely-used source code editor to present the bytecode enhancements applied to the program constructs of the displayed compilation unit. We have also created a new debugging architecture that enables a symbolic debugger running an enhanced program to display the original source code, undoing the enhancements at runtime.

To check the effectiveness of the enhancements-aware tools, we have applied both of them to four different applications, each using a different bytecode enhancement scheme. The first two applications use transparent persistence architectures, albeit with drastically different enhancement strategies. While JDO modifies persistent classes, Hibernate generates proxy classes that inherit from them. Another difference between JDO and Hibernate is the time when the enhancement takes place: while JDO enhances persistent classes as a static post-compilation step, Hibernate generates proxy classes at class load time.

Two other applications come from the domain of distributed computing. The first application performs an RMI Remoting enhancement, in which the original, local class is rewritten into a remote implementation class, and new proxy and RMI remote interface classes are generated. This

rewrite is performed by several systems designed to make distributed computing in Java more intuitive, both at the source level such as JavaParty [PZ97], and transparently at the bytecode level such as J-Orchestra [TS02].

Another enhancement from the distributed computing domain modifies the structure of a data class to optimize the network transfer of its instances. Class fields are divided into a set that is used by a remote server and the one that is not, and the class is rewritten into two partitions containing those sets, using the Split Class binary refactoring [TS05]. Finally, the partition to be transferred across the network is made to implement `Serializable` to enable the marshaling of its instances. Because the rewrite adds a new capability, it is classified as an enhancement.

We expressed each enhancement scheme in SER. For the transparent persistence architectures, we reverse-engineered the enhancements by comparing the original and enhanced versions of multiple application classes. In the distributed application cases, we simply documented in SER the transformations informally described in the respective research publications [TS02, TS05].

5.1 Source Editor with Zoom-in-on-enhancements View

Intermediate code enhancement introduces new functionality behind the scenes, transparently to the programmer. Furthermore, leaving the programmer unaware of the specific changes applied directly to the bytecode has been recognized as a key benefit of the technique—it improves separation of concerns, with the programmer being responsible for business logic only. Nevertheless, as we argue next, making the bytecode enhancement information accessible to the programmer can yield software engineering benefits. For example, bytecode enhancers follow a certain convention in choosing the names of program constructs that they add. In particular, JDO starts the names of all the added setter/getter methods with prefix `jdoSet/Get`. There is nothing, however, that would prevent the programmer from writing a method starting with these prefixes, thus creating a diffi-

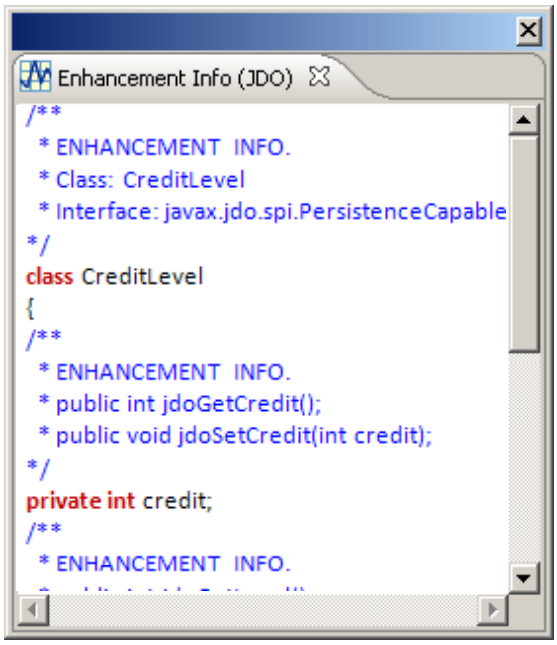

Figure 5.1: Documentation for the JDO Enhancement.

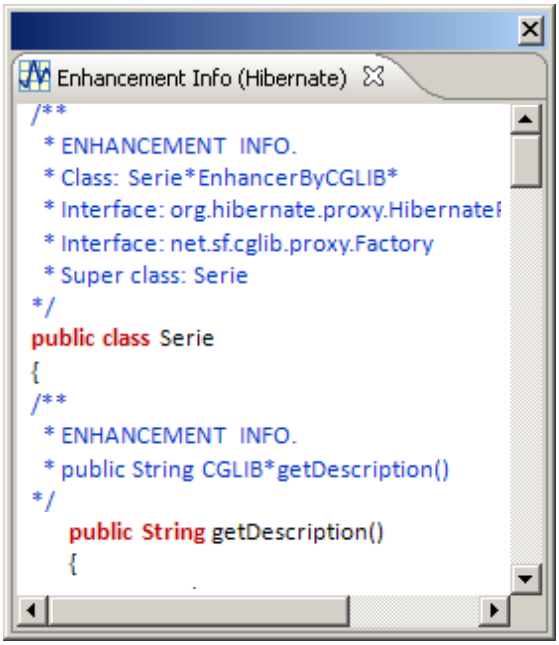

Figure 5.2: Documentation for the Hibernate Enhancement.

cult to diagnose name clash. By examining the enhancements that will be applied to a class, the programmer can quickly identify such inappropriately-named program constructs. As another example, consider the restriction of Hibernate of not being able to persist instances of `final` classes and any classes that have `final` methods. This restriction seems completely arbitrary, unless the programmer can examine how Hibernate enhances the persistent classes. This restriction becomes immediately apparent, as soon as the programmer observes that persistence-enabling proxies generated by Hibernate extend the persistent classes. As yet another example, certain performance bottlenecks in the enhanced code can be identified by examining the enhancements. For example, the use of RMI in remoting enhancements can be detrimental to performance in some networking environments. As a final example, some bugs in metadata, which describes which program constructs are to be enhanced, can be more easily identified if the enhancement code is visible.

In this case study, we have integrated the SER interpreter with an Eclipse Java code editor. Whenever the programmer selects a class whose bytecode is subject to enhancement, a new view pops up

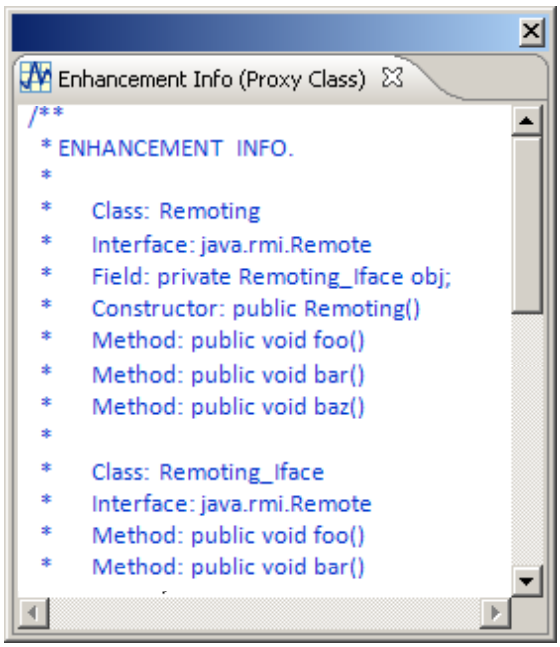

Figure 5.3: Documentation for the Proxy class Enhancement.

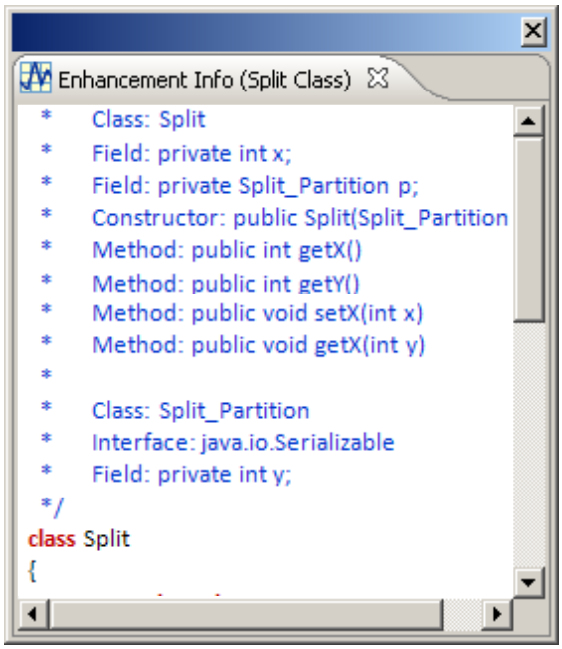

Figure 5.4: Documentation for the Split class Enhancement.

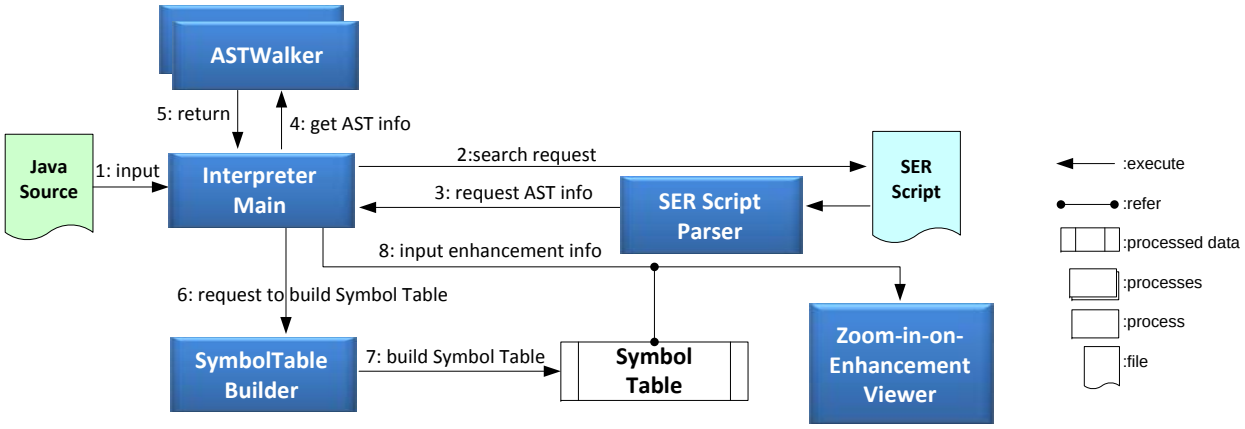

Figure 5.5: The zoom-in-on-enhancement view collaboration diagram.

displaying an abbreviated description of the enhancements. We call this view window a *zoom-in-on-enhancements view*. Following the Eclipse tooling strategy, the view is visible only if activated, so if they so choose, the programmers are free to remain oblivious about the nature and specifics of enhancements. In the case if the original class is modified at the bytecode level, the enhance-

ments are shown as special comments in the main editor. Since not all the information about the enhancements in known at source edit time, the enhancements cannot be expressed in source code form. If a single class is associated with several classes created during an enhancement, each of the created classes is displayed in a separate view.

Figures 5.1, 5.2, 5.3, and 5.4 show the screen-shots of the *zoom-in-on-enhancements views*, which document the enhancements used in the example applications. The views have been integrated with Eclipse.

Figure 5.5 presents a collaboration diagram that shows the backend processing triggered by the source editor to launch *a zoom-in-on-enhancements view*. When the SER interpreter's main module receives a Java source file as input, the corresponding SER script is identified (using a configuration file), loaded, and parsed. The interpreter employs several abstract syntax tree walkers (using Visitors) to traverse the Java program, collecting the information about how the general enhancement instructions in the SER script will affect the specific program constructs (e.g., fields, methods, constructors, etc.). The collected enhancement information is stored in a symbol table for fast searching and retrieval. Finally, the interpreter compiles a complete documentation of the enhancements, which it uses to parameterize the zoom-in-on-enhancement viewer.

5.2 A Symbolic Debugger for Enhanced Intermediate Code

Because intermediate code enhancements are not represented at the source code level, source-level debugging of such enhanced bytecode is nontrivial. Application code is enhanced to be able to interact with a framework, and the enhancements cannot be simply turned off to facilitate debugging. Thus, tracing, analyzing, and fixing flawed programs whose bytecode has been enhanced with a standard debugger is misleading—the debugger will show all the enhanced program's code faithfully, both the original logic and the transparently introduced enhancements. From the debugging

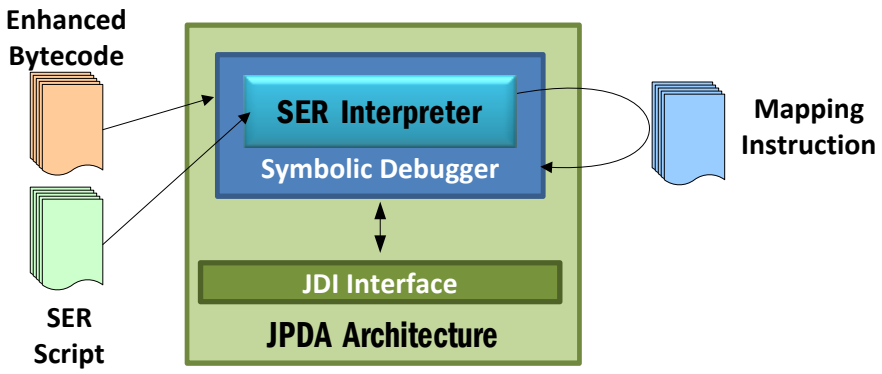

Figure 5.6: Debugging transparently enhanced programs.

perspective, enhancements obfuscate the original source code's logic.

The debugging of transparently enhanced programs can be facilitated by making a symbolic debugger aware of the enhancements. The debugger could execute an enhanced program, but report the source code information pertaining to the original source code. As our proof of concept, we have created a new debugging architecture that leverages the facilities offered by the Java Platform Debugger Architecture (JPDA) [Suna]. We then applied the new architecture to augment the capabilities of the standard debugger distributed with Sun's JDK with an awareness of the enhancements in debugged programs.

Figure 5.6 demonstrates our new debugging architecture, which integrates a SER interpreter. When debugging enhanced bytecode, the debugger also takes the SER description of the enhancements as input. The integrated SER interpreter then computes *symbolic undo instructions* that map the enhanced bytecode to the original source code.

To reverse the enhancements that add and change program constructs, our debugging architecture introduces the *skip* and *reverse* symbolic undo instructions, which can be applied to classes, methods, fields, and entire packages. The debugger organizes the symbolic undo instructions as a hierarchical collection through the "contains" relationship (i.e., packages contain classes, classes contain methods, etc.). All the instructions are sorted in the order of their qualified full names,

so that they could be efficiently located through binary search in logarithmic time. The debugger executes the symbolic undo instructions on the encountered enhancements, so that the information reported to the user pertains to the original programmer written code.

The symbolic undo instructions work as follows. The *skip* instruction suppresses the output of those program elements that, in the original version, have not been represented in source code. Skip operations raise a special purpose debugging event whose semantics is similar to the regular debugger's *step* event; however, the output associated with handling the event is suppressed. The *reverse* instruction changes the name of a program construct displayed through the standard debugging output. For example, if an enhancer has changed the name of a class, a reverse instruction will direct the debugger to report the class's original name.

From the user's perspective, our symbolic debugger is a plug-in replacement for the standard JDK command-line debugger, providing the capabilities to step through the code, set breakpoints, print variable values, etc. The implementation leverages the Java Platform Debugger Architecture (JPDA)[Suna], which consists of several layers of protocols and interfaces provided by the Java Virtual Machine. The functionality required for symbolically undoing transparent enhancements is implemented by inserting additional translation logic to the standard operations used by debuggers based on JPDA. JPDA includes a client interface for accessing the debugging services, which are connected to the JVM through an event queue. To receive events from the queue, the client must set a breakpoint by calling an API method. Triggering a breakpoint delivers a debugging event to an event handler method that can access all the breakpoint's information, including its location, value of variables,etc.

Events are also triggered when the *step* command moves the debugger to the next source code line. The values of member and local variables can be printed at any point after a breakpoint has been triggered or the step command has been executed. Our symbolic debugger intercepts each debugging event and executes a corresponding symbolic undo instruction, thus mapping the

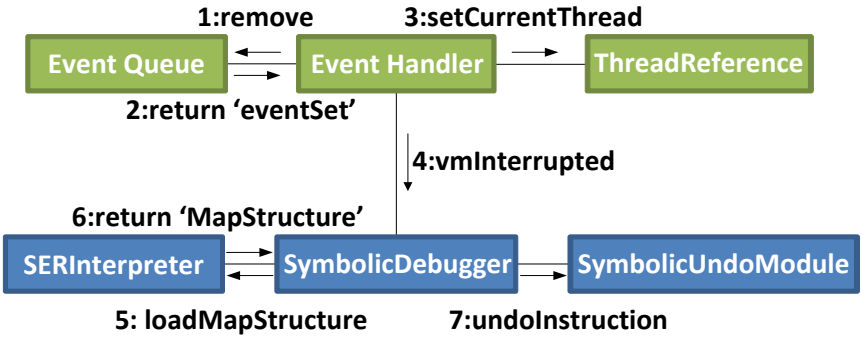

Figure 5.7: The symbolic debugger's collaboration diagram.

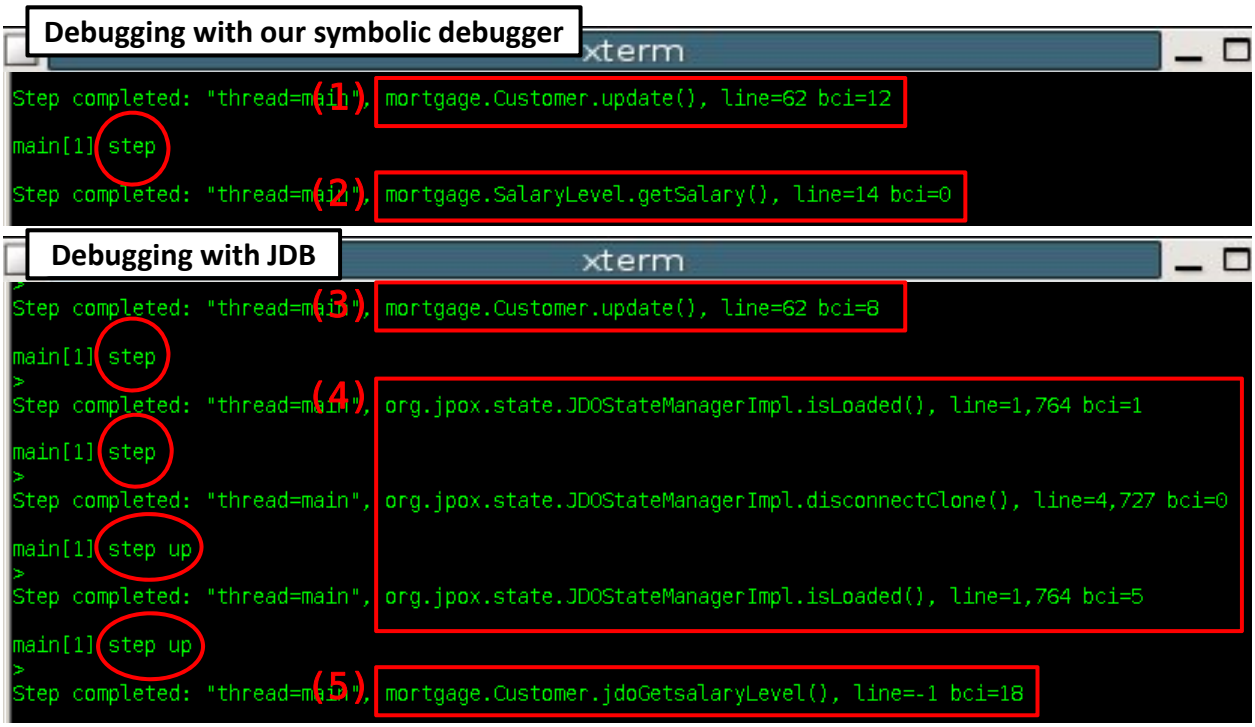

Figure 5.8: Debugging enhanced code: Debugger with an enhancements awareness vs. JDB

enhanced bytecode back to the original version of the code.

Figure 5.7 shows a collaboration diagram that details the runtime architecture of our symbolic debugger. The diagram depicts the main events driving the execution of our symbolic debugger. The main module of the debugger, *SymbolicDebugger*, manages the symbolic undo information,

using the services of the *EventHandler*. The *EventHandler* receives debugging events from the JPDA *EventQueue* and delegates them to *SymbolicDebugger* by calling `vmInterrupted`. *SymbolicDebugger* then evaluates the received event against the symbolic undo information, which can be generated on demand, and symbolically undoes any encountered enhancement. By using JPDA, which is thread-aware, our debugger can effectively handle multi-threaded programs.

As a demonstration of the utility of the new debugger, we inserted a bug to the example mortgage eligibility application presented in Section 3. We then traced the bug using both our augmented debugger and JDB. In our experiences, the enhancements introduced by the JDO framework complicate the debugging process. Figure 5.8 shows two screen shots, corresponding to debugging with our symbolic debugger and JDB, respectively. Points marked as (1) and (3) mark the start of the traced method, in their respective debugger's displays. Individual debugging steps are circled. Point (2) shows the original code as displayed by our debugger. Points (4) and (5) show the bytecode instructions that would be skipped and reversed by our debugger, respectively. Our experience suggests that our debugger has the potential to become an effective aid in locating bugs in enhanced programs. Nevertheless, only a controlled user study can confirm the veracity of this conjecture. We plan to conduct such a study as future work.

5.3 Discussion

We demonstrate how these transformations can be perused by the programmer in the zoom-in-on-enhancements view and can also enrich the functionality of bytecode-level programming tools in the symbolic debugger. Specifically, our SER interpreter automatically derives the transformations between the original and enhanced program versions and vice versa. Using a SER script, our interpreter calculates a precise mapping between a given piece of source code and its enhanced bytecode representation. This mapping can be effectively utilized by programming tools to improve their

precision and utility.

The main contribution of this research is the SER language and its interpreter, whose expressiveness we validate by building an enhanced source editor and a symbolic debugger. The fact that we were able to build functional source level tools that work with four different enhancement strategies, in our view, validates the power of SER and the effectiveness of its interpreter. Although user studies with real programmers (as part of future work) are likely to reveal interesting insights, giving us direct feedback from programmers, these user studies will only be tangentially related to the main contributions of this research.

SER scripts are the responsibility of framework developers, who will have to update the scripts to reflect the latest version of the enhancement strategy in place. This does require an extra maintenance effort, which we argue should be insignificant due to the declarative nature of SER.

Troubleshooting and fine-tuning programs often requires that the programmer know exactly which parts of the programs runtime representation come from programmer written code and which ones were introduced through bytecode enhancement. Existing programming tools fail to provide this information.

Our experiences with several commercial frameworks (JDO, Hibernate, JSecurity) and research projects indicate that bytecode enhancers avoid changing method bodies with the exception for replacing field accesses with setter/getter methods. Our choice of the structural changes to support is thus based only on empirical evidence. Our hypothesis is that changing bytecode on a larger scale safely requires expensive program analysis, which can be infeasible to perform at class load time without imposing a significant performance overhead. If bytecode enhancers start making larger-scale changes in the future, SER will have to be expanded accordingly.

Chapter 6

Conclusions and Future Work

This chapter has argued about the value of enhancing source-level programming tool with an awareness of transparent program transformations. To enable such awareness, we have introduced SER, a declarative language that concisely describes structural enhancements. To validate our approach, we have augmented two existing source-level programming tools with an awareness of bytecode enhancement. We have also expressed in SER four enhancement strategies followed by different enhancers and used our augmented programming tool to handle the enhanced programs. As part of enhancing a source level debugger, we have introduced a new debugging architecture that makes it possible to calculate reverse-mapping instructions based on a SER specification. Bytecode enhancement has already entered the mainstream of enterprise software development, and future enhancers are likely to transform programs in even more complex ways. As a result, source code alone will become even less sufficient in presenting a realistic picture about the functionality of a program. Our approach of making programming tools aware of bytecode enhancements has the potential to address this problem, and help programmers enjoy the benefits of transparent bytecode enhancement without suffering any of its disadvantages.

The algorithms, techniques, and tools for supporting reuse and evolution in metadata-driven software development, explored by this dissertation, provide rich possibilities for future work. Each software technology developed for this research can be further improved for the perspective of its functionalities and applicability. Additionally, the general idea can be discussed for its applicability to other domains. We next outline several possible future work directions.

As future work, we plan to conduct user studies to evaluate the value of our approach for programmers with different levels of expertise. One such study could evaluate whether a symbolic undo debugger is more effective than a regular debugger in helping the programmer to locate and fix bugs. Another study could evaluate the value of integrating the enhancement information with a programming editor. We plan to create a debugger for SER scripts. Although the declarative nature of SER scripts makes it easier to ensure their correctness, bugs still can be introduced, particularly if a SER script is developed by someone other than a framework developer. Having a SER debugger is likely to improve the usability of our approach. We plan to extend our approach to programs that have been transparently enhanced more than once, possibly by different enhancers. For example, the same application can use multiple frameworks, each enhancing intermediate code in its own way. Bytecode enhancement could add functionality in languages other than the host language, including query languages such as SQL or Datalog.

Our approach would have to be extended to add the enhancements-awareness to programming tools handling multi-language applications. There could be potential benefit in synthesizing the source code representation of bytecode-only enhancements. To that end, the SER interpreter would have to be integrated with a decompiler that is enhancements-aware. The success of this approach would mainly depend on the decompiler's efficiency and precision. Generating SER scripts automatically will reduce the framework programmer's effort and make our methodology more appealing to the average programmer. A promising approach would require generalizing program differencing, a target of several recent research efforts [KNG07]. Finally, we would like to make our symbolic debugger available as an Eclipse IDE plug-in.

Appendices

Appendix A

The Structural Enhancement Rules (SER) Language's EBNF Grammar

A.1 Base syntax

A.1.1 Variant Attributes

```
String Char = Printable - ["]
StringLiteral = '"'String Char*'"'
IdLetter = Letter
IdAlphaNumeric = Alphanumeric
Identifier = IdLetterIdAlphaNumeric*
IndirectCharLiteral = QuoteCharSign1Quote
‘‘Case Sensitive’’ = ‘True’
‘‘Start Symbol’’ = <CompilationUnit>
Comment Line = ‘--’
<Literal>
  ::= StringLiteral
<Modifiers>
        ::= <Modifier>
```

```
          |  <Modifiers> <Modifier>
<Modifier>
         ::= 'public' | 'protected'
          |  'private' |  'static'
          |  'abstract' |  'final'
          |  'native' |  'synchronized'
<ClassAccess>
  ::= <SERClass Type> '.' <ClassAttribute>
<ClassAttribute>
  ::= 'Name' | 'Type'
<ModuleAccess>
  ::= Identifier '.' <ClassAccess>
<AssignOptr>
  ::= '='
   |  '+='
<NamingConvention>
  ::= <Literal>
<Prefix>
  ::= <Literal>
<Name>
  ::= <SimpleName>
   |  <QualifiedName>
<SimpleName>
  ::= Identifier
<QualifiedName>
  ::= <Name> '.' Identifier
```

A.1.2 Type and Signature syntax

```
<PatternVarType>
  ::= ‘Pattern’
<IteratorVarType>
  ::= ‘Iterator’
<Type>
  ::= ‘Class’ | ‘Interface’
<SERClass Type>
  ::= ‘OrgClass’ | ‘EnhClass’
<Type>
  ::= <PrimitiveType> | <ReferenceType>
<PrimitiveType>
  ::= <NumericType> | boolean
<NumericType>
  ::= <IntegralType> | <FloatingPointType>
<IntegralType>
  ::= byte | short | int | long | char
<FloatingPointType>
  ::= float | double
<ReferenceType>
  ::= <ClassInterfaceType> | <ArrayType>
<ClassInterfaceType>
  ::= <ClassType> | <InterfaceType>
<ClassType>
  ::= Identifier
```

```
<InterfaceType>
  ::= Identifier
<ArrayType>
  ::= <Type> [ ]

<SignatureDclr>
  ::= <MethodHeader>
<MethodHeader>
  ::= <Modifiers>? <ResultType>
      <MethodDeclarator> <Throws>?
<ResultType>
  ::= <Type> | void
<MethodDeclarator>
  ::= <Identifier> ( <FormalParameterList>? )
<FormalParameterList>
  ::= <FormalParameter> |
      <FormalParameterList> , <FormalParameter>
<FormalParameter>
  ::= <Type> <VariableDeclaratorId>
<VariableDeclaratorId>
  ::= <Identifier> | <VariableDeclaratorId> [ ]
<Throws>
  ::= throws <ClassTypeList>
<ClassTypeList>
  ::= <ClassType> | <ClassTypeList> , <ClassType>
```

A.1.3 Body syntax

```
<CompilationUnit>
  ::= <ProgramDclr>
   |  <ProgramDclr> | <ModuleDclr>
<ProgramDclr>
  ::= 'Program' <Name>
<ModuleDclr>
  ::= 'Module' <Name>
<ProgramBody>
  ::= 'Begin' 'End'
   |  'Begin' <ModuleBody> 'End'
<ModuleBody>
  ::= 'Begin' 'End'
```

A.2 Iterator syntax

```
<IterVar>
  ::= Identifier
<IteratorVarId>
  ::= 'Var' <IteratorVarType> <IterVar>
<IteratorVarDclr>
  ::= <IteratorVarId> <AssignOptr> <MethodInvocation>
<MethodInvocation>
  ::= <SERClass Type> '.' <Accessors>
```

```
   |  <SERClass Type> '.' <IteratationHandler>
   |  <SERClass Type> '.' <ModificationHandler>
```

A.3 Pattern syntax

```
<PatternVar>
  ::= Identifier
<PatternVarId>
  ::='Var' <PatternVarType> <PatternVar>
<PatternVarDclr>
  ::= <PatternVarId> <PatternDclrBody>
<PatternDclrBody>
  ::= 'Begin' <PatternDclr> 'End'
<PatternDclr>
  ::= <PatternModifierDclr>
   |  <PatternTypeDclr>
   |  <PatternNameDclr>
   |  <PatternSignatureDclr>
<PatternModifierDclr>
  ::= <ModiferAccess> <AssignOptr> <Modifier>
<PatternTypeDclr>
  ::= <TypeAccess> <AssignOptr> <ModuleAccess>
<PatternNameDclr>
  ::= <NameAccess> <AssignOptr> <NamingConvention>
<PatternSignatureDclr>
```

```
  ::= <SignatureAccess> <AssignOptr> <SignatureDclr>
<PatternAccess> ::= <ModiferAccess>
                 |  <TypeAccess>
                 |  <NameAccess>
                 |  <SignatureAccess>
<ModiferAccess>   ::= 'modifiers'
<TypeAccess>      ::= 'type'
<NameAccess>      ::= 'name'
<SignatureAccess> ::= 'signature'
```

A.4 Access and Replication syntax

```
<Accessors>
  ::= <Constructors>
   |  <Methods>
   |  <Fields>
<Constructors>
  ::= 'Constructors' '(' ')'
   |  'Constructors' '(' <PatternVar> ')'
<Methods>
  ::= 'Methods' '(' ')'
   |  'Methods' '(' <PatternVar> ')'
<Fields>
  ::= 'Fields' '(' ')'
   | 'Fields' '(' <PatternVar> ')'
```

```
<IteratationHandler>
  ::= <FieldGetReplacer> | <FieldSetReplacer>
   |  <CreateNewIterator>
<FieldGetReplacer>
  ::= 'FieldGetReplacer' '(' <Prefix> ',' <IterVar> ')'
<FieldSetReplacer>
  ::= 'FieldSetReplacer' '(' <Prefix> ',' <IterVar> ')'
<CreateNewIterator>
  ::= 'CreateNewIterator' '(' <IterVar> ')'
```

A.5 Addition and removal syntax

```
<ModificationHandler>
  ::= <AddInterface>   | <RemoveInterface>
   |  <AddSuperclass>  | <RemoveSuperclass>
   |  <AddConstructor> | <RemoveConstructor>
   |  <AddMethod>      | <RemoveMethod>
   |  <AddField>       | <RemoveField>
<AddInterface>
  ::= 'AddInterface' '(' <IterVar> ')'
<AddSuperclass>
  ::= 'AddSuperclass' '(' <IterVar> ')'
<AddConstructor>
  ::= 'AddConstructor' '(' <PatternVar> ')'
<AddMethod>
  ::= 'AddMethod' '(' <IterVar> ')'
   |  'AddMethod' '(' <PatternVar> ')'
<AddField>
  ::= 'AddField' '(' <IterVar> ')'
  ::= 'AddField' '(' <PatternVar> ')'
<RemoveInterface>
  ::= 'RemoveInterface' '(' <IterVar> ')'
<RemoveSuperclass>
  ::= 'RemoveSuperclass' '(' <IterVar> ')'
<RemoveConstructor>
```

```
  ::= 'RemoveConstructor' '(' <PatternVar> ')'
<RemoveMethod>
  ::= 'RemoveMethod' '(' <IterVar> ')'
   |  'RemoveMethod' '(' <PatternVar> ')'
<RemoveField>
  ::= 'RemoveField' '(' <IterVar> ')'
  ::= 'RemoveField' '(' <PatternVar> ')'
```

Bibliography

[ADJ+96] M. P. Atkinson, L. Daynès, M. J. Jordan, T. Printezis, and S. Spence. An orthogonally persistent Java. *SIGMOD Rec.*, 25(4):68–75, 1996.

[ANMM06] Chris Andreae, James Noble, Shane Markstrum, and Todd Millstein. A framework for implementing pluggable type systems. *SIGPLAN Not.*, 41(10):57–74, 2006.

[AOH04] T. Apiwattanapong, A. Orso, and M.J. Harrold. A differencing algorithm for object-oriented programs. *Automated Software Engineering, 2004. Proceedings. 19th International Conference on*, pages 2–13, Sept. 2004.

[Apa] Apache Jakarta Project. The Byte Code Engineering Library. `http://jakarta.apache.org/bcel/manual.html`.

[Atk88] M. P Atkinson. The Napier88 persistent programming language and environment, 1988.

[BHS92] Gary Brooks, Gilbert J. Hansen, and Steve Simmons. A new approach to debugging optimized code. In *PLDI '92: Proceedings of the ACM SIGPLAN 1992 conference on Programming language design and Implementation*, pages 1–11, 1992.

[BKN05] C. Bauer, G. King, and I. NetLibrary. *Hibernate in Action*. Manning, 2005.

[CDL99] Trishul M. Chilimbi, Bob Davidson, and James R. Larus. Cache-conscious structure definition. *SIGPLAN Not.*, 34(5):13–24, 1999.

[Cop94] Max Copperman. Debugging optimized code without being misled. *ACM Trans. Program. Lang. Syst.*, 16(3):387–427, 1994.

[Cza06] Krzysztof Czarnecki. Framework-specific modeling languages with round-trip engineering. In *In MoDELS*, pages 692–706, 2006.

[Dah99] M. Dahm. Byte code engineering. *Java Informations Tage*, pages 267–277, 1999.

[Dmi02] Mikhail Dmitriev. Language-specific make technology for the Java programming language. *SIGPLAN Not.*, 37(11):373–385, 2002.

[DRS07] Bruno Dufour, Barbara G. Ryder, and Gary Sevitsky. Blended analysis for performance understanding of framework-based applications. In *ISSTA '07: Proceedings of the 2007 international symposium on Software testing and analysis*, pages 118–128, New York, NY, USA, 2007. ACM.

[EAH+07] Marc Eaddy, Alfred Aho, Weiping Hu, Paddy McDonald, and Julian Burger. Debugging aspect-enabled programs. In *Software Composition*, pages 200–215. 2007.

[EF08] Eclipse Foundation. Eclipse Java development tools, March 2008. `http://www.eclipse.org/jdt`.

[Fai98] Rickard Edward Faith. *Debugging programs after structure-changing transformation*. PhD thesis, 1998. Adviser-Jan F. Prins.

[FCL06] Manuel Fähndrich, Michael Carbin, and James R. Larus. Reflective program generation with patterns. In *GPCE '06: Proceedings of the 5th international conference on Generative programming and component engineering*, pages 275–284, New York, NY, USA, 2006. ACM.

[FL05] J. Flen and A. Linden. Gartners hype cycle special report. Technical report, Gartner Research, 2005. `www.gartner.com`.

[Fri83] Peter Fritzson. A systematic approach to advanced debugging through incremental compilation (preliminary draft). In *Proceedings of the ACM SIGSOFT/SIGPLAN software engineering symposium on High-level debugging*, pages 130–139, 1983.

[Gle] Glen McCluskey. Using Java Reflection. `http://java.sun.com/developer/technicalArticles/ALT/Reflection/index.html`.

[HCU92] Urs Hölzle, Craig Chambers, and David Ungar. Debugging optimized code with dynamic deoptimization. *SIGPLAN Not.*, 27(7):32–43, 1992.

[Hen82] John Hennessy. Symbolic debugging of optimized code. *ACM Trans. Program. Lang. Syst.*, 4(3):323–344, 1982.

[IKI04] Takashi Ishio, Shinji Kusumoto, and Katsuro Inoue. Debugging support for aspect-oriented program based on program slicing and call graph. In *ICSM '04: Proceedings of the 20th IEEE International Conference on Software Maintenance*, pages 178–187, 2004.

[KCS05] Naveen Kumar, Bruce R. Childers, and Mary Lou Soffa. Tdb: a source-level debugger for dynamically translated programs. In *AADEBUG'05: Proceedings of the sixth international symposium on Automated analysis-driven debugging*, pages 123–132, 2005.

[KHH+01] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. In *Proceedings of the 15^{th} European Conference on Object-Oriented Programming (ECOOP)*, pages 327–353, London, UK, 2001. Springer-Verlag.

[Kin76] James C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, 1976.

[KLM+97] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. M. Loingtier, and J. Irwing. Aspect-oriented programming. In *ECOOP*. Springer-Verlag, 1997.

[KNG07] Miryung Kim, David Notkin, and Dan Grossman. Automatic inference of structural changes for matching across program versions. In *ICSE '07: Proceedings of the 29th International Conference on Software Engineering*, pages 333–343, 2007.

[LAC+96] B. Liskov, A. Adya, M. Castro, S. Ghemawat, R. Gruber, U. Maheshwari, A. C. Myers, M. Day, and L. Shrira. Safe and efficient sharing of persistent objects in thor. In *SIGMOD '96: Proceedings of the 1996 ACM SIGMOD international conference on Management of data*, pages 318–329, New York, NY, USA, 1996. ACM.

[MBMZ01] Alonso Marquez, Stephen Blackburn, Gavin Mercer, and John N. Zigman. Implementing orthogonally persistent java. In *POS-9: Revised Papers from the 9th International Workshop on Persistent Object Systems*, pages 247–261, London, UK, 2001. Springer-Verlag.

[Mic] Microsoft. Microsoft Open Database Connectivity. `http://msdn.microsoft.com/en-us/library/ms710252(VS.85).aspx`.

[OK05] Alessandro Orso and Bryan Kennedy. Selective Capture and Replay of Program Executions. In *Proceedings of the Third International ICSE Workshop on Dynamic Analysis (WODA 2005)*, pages 29–35, St. Louis, MO, USA, May 2005.

[ORH02] Alessandro Orso, Anup Rao, and Mary Jean Harrold. A technique for dynamic updating of Java software. *Proceedings of the International Conference on Software Maintenance (ICSM'02)*, October 2002.

[PG06] Susanne Cech Previtali and Thomas R. Gross. Dynamic updating of software systems based on aspects. *Proceedings of the 22nd IEEE International Conference on Software Maintenance*, pages 83–92, September 2006.

[PT08] Guillaume Pothier and Éric Tanter. Extending omniscient debugging to support aspect-oriented programming. In *SAC '08: Proceedings of the 2008 ACM symposium on Applied computing*, pages 266–270, 2008.

[PZ97] M. Philippsen and M. Zenger. JavaParty–transparent remote objects in Java. *Concurrency Practice and Experience*, 9(11):1225–1242, 1997.

[Ric06] Chris Richardson. Untangling enterprise Java. *ACM Queue*, 4(5):36–44, 2006.

[Rus07] Craig Russell. Java Data Objects 2.1, June 2007. `http://db.apache.org/jdo/specifications.html`.

[SdML04] Ganesh Sittampalam, Oege de Moor, and Ken Friis Larsen. Incremental execution of transformation specifications. In *POPL '04: Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 26–38, 2004.

[Shi] Shigeru Chiba. Java Programming Assistant. `http://www.csg.is.titech.ac.jp/~chiba/javassist`.

[SHL+07] Henrik Stuart, René Rydhof Hansen, Julia L. Lawall, Jesper Andersen, Yoann Padioleau, and Gilles Muller. Towards easing the diagnosis of bugs in os code. In *PLOS '07: Proceedings of the 4th workshop on Programming languages and operating systems*, pages 1–5, New York, NY, USA, 2007. ACM.

[Son] Myoungkyu Song. The structural enhancement rules language website. `http://research.cs.vt.edu/vtspaces/ser`.

[Suna] Sun Microsystems. Java Platform Debugger Architecture. http://java.sun.com/javase/technologies/core/toolsapis/jpda/.

[Sunb] Sun Microsystems. JavaBeans Specification. http://java.sun.com/javase/technologies/desktop/javabeans/docs/spec.html.

[Sunc] Sun Microsystems. The Java Database Connectivity. http://java.sun.com/products/jdbc/overview.html.

[TS02] Eli Tilevich and Yannis Smaragdakis. J-Orchestra: Automatic Java application partitioning. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, pages 178–204. Springer-Verlag, LNCS 2374, 2002.

[TS05] Eli Tilevich and Yannis Smaragdakis. Binary refactoring: Improving code behind the scenes. In *Proceedings of International Conference on Software Engineering (ICSE)*, pages 264–273, May 2005.

[TT08] Wesley Tansey and Eli Tilevich. Annotation refactoring: inferring upgrade transformations for legacy applications. In *OOPSLA '08: Proceedings of the 23rd ACM SIGPLAN conference on Object oriented programming systems languages and applications*, pages 295–312, 2008.

[VeABT99] Eelco Visser, Zine el Abidine Benaissa, and Andrew Tolmach. Building program optimizers with rewriting strategies. *SIGPLAN Not.*, 34(1):13–26, 1999.

[VEdM06] Mathieu Verbaere, Ran Ettinger, and Oege de Moor. JunGL: a scripting language for refactoring. In *ICSE '06: Proceedings of the 28th International Conference on Software Engineering*, pages 172–181, 2006.

[WS97] Deborah L. Whitfield and Mary Lou Soffa. An approach for exploring code improving transformations. *ACM Trans. Program. Lang. Syst.*, 19(6):1053–1084, 1997.