The following paper was originally published in the
Proceedings of the USENIX Annual Technical Conference (NO 98)
New Orleans, Louisiana, June 1998

Automatic Program Transformation with JOIE

Geoff A. Cohen and Jeffrey S. Chase
Duke University
David L. Kaminsky
IBM Application Development Technology Institute

For more information about USENIX Association contact:

1. Phone: 510 528-8649
2. FAX: 510 548-5738
3. Email: office@usenix.org
4. WWW URL: http://www.usenix.org/

Automatic Program Transformation with JOIE

Geoff A. Cohen*
Jeffrey S. Chase
Department of Computer Science
Duke University
{gac,chase}@cs.duke.edu

David L. Kaminsky
Application Development Technology Institute
IBM Research Triangle Park
dlk@us.ibm.com

Abstract

While the availability of platform-independent code on the Internet is increasing, third-party code rarely exhibits all of the features desired by end users. Unfortunately, developers cannot foresee and provide for all possible extensions.

In this paper, we describe load-time transformation, a stage in the program development lifecycle in which classes are modified at load time according to user-supplied directives. This allows the users to select transformations that add new features, customize the implementation of existing features, and apply the changes to all classes in the environment.

The Java Object Instrumentation Environment (JOIE) is a toolkit for constructing transformations of Java classes. An enhanced class loader calls user-supplied *transformers* that specify rules for transforming target classes. We describe some applications of load-time transformation, including extending the Java environment, integrating classes with specialized environments, and adding functionality directly to classes.

1 Introduction

The accelerating use of Java [GJS96] to enable transportable code over the Internet has created both the need and an opportunity to take a larger view of program development. The possibility that programs will consist of components loaded dynamically, possibly from multiple diverse sources, creates challenges for program developers: How can third-party code be adapted to local environments? How do we add functionality to "shrink-wrapped" code? Is it possible to insert new behaviors, such as recoverability, caching, or visualization, into existing implementations?

Fortunately, Java bears within itself the seeds of the solution. Java is an ideal environment for *load-time transformation*, a powerful technique in which user-specified *transformers* (possibly supplied by a third party) add, remove, or change fundamental details of transportable code as it is imported into the local Java Virtual Machine (JVM)[LY97]. Java has several properties that assist load-time transformation. Transportable Java code arrives from the network as compiled *classfiles* containing procedures (methods) and related data definitions for an object type (class): these classfiles retain a great deal of symbolic information, allowing the receiver to determine the structure of the class and to modify it on-the-fly. Secondly, methods are represented as JVM bytecodes: since bytecodes are stack instructions, it is relatively easy to splice new code into existing methods. Finally and most importantly, the JVM uses a user-extensible *class loader* to locate and load new classes on demand: the class loader can be modified to apply load-time transformations to every classfile brought into the local environment.

Load-time transformation has far-reaching implications for the balance of responsibility between class authors and users. In the traditional model, users run programs whose attributes are statically determined by the original authors. Load-time transformation enables a new model, one in which end users assemble and customize applications by chaining together combinations of original code and third-party transformers. The role of the transformers is to implement class features or extensions that the authors did not foresee or chose not to support directly in the original class. Our hypothesis is that many program behaviors are best applied by generic class transformers as needed, rather than hard-wired into the class source. Broadly, transformers are useful for implementing any behavior that is orthogonal to the purpose of the class and can be specified indepen-

*This work is supported by the National Science Foundation under grants CCR-96-24857 and CDA-95-12356. Geoff Cohen is supported in part by an IBM Cooperative Graduate Fellowship. Portions of the work described in this paper were done at IBM Research Triangle Park.

dently. Moreover, support for transformers can improve reuse of existing code by providing a means to adapt it to local needs.

This paper describes load-time transformation using the Java Object Instrumentation Environment, or JOIE[1], a toolkit for specifying transformations for Java classes. JOIE transformers are written in Java and use JOIE primitives to analyze and modify classes. The JOIE toolkit includes an enhanced class loader that invokes transformers at load time. JOIE works with any JVM, and is available for download at `http://www.cs.duke.edu/ari/joie/`.

The remainder of the paper addresses four main questions that arise from the introduction of load-time transformation and its use in JOIE. What are its capabilities? How is it implemented? What are the useful applications? Can it compromise existing guarantees of security and safety?

Section 2 introduces load-time transformation within the context of the program lifecycle. Section 3 presents the environment and the capabilities JOIE offers to enable transformations. Section 4 examines some implementation details for Java transformations. Section 5 outlines some application areas for JOIE, and Section 6 presents in more detail the implementation of a specific transformer, Automatic Observable, which adds to classes the ability to detect state changes and report them to registered observers. Section 7 discusses issues arising from load-time transformation, including security, safety, debugging complexity, and legal concerns.

2 Program Transformation

In this section, we take a broad look at the stages of the program lifecycle, and examine in detail the loading stage as a point for transformation. We discuss Java in particular, but the principles apply generally to many other languages and environments.

2.1 Stages of the Program Lifecycle

There are a number of stages in the program lifecycle during which a program author or user can specify the functionality of a class or set of classes. Some examples of tools used at different stages are detailed in Table 1. Originally, of course, the base functionality is declared by the class author in source code, and that source code is translated into an executable by a compiler.

Authors or users can employ post-processors such as instrumentation tools to insert new method calls into an existing executable image. A popular example of this is the tool ATOM [SE94], which works on executable images for the Alpha processor; similar functionality is available for Java with BIT [LZ97]. Most often, this instrumentation is used for performance analysis or as an interface to platform simulation. An important guarantee typically made by instrumentation tools is that the semantics of the original program are not changed. However, Shasta [SG97] processes executable images to run on distributed shared memory systems. Object Design Incorporated's ObjectStore PSE [Obj98] also uses a post-processor, to insert persistence methods into existing code. Rational Software Corporation's tool Purify [Rat98] changes code to detect memory leaks.

Multiple third-party components (classes or more often collections of interacting classes) are integrated during *application composition*. In Java, these components are known as Beans and are often handled in visual builders. This composition allows consumers of code—either end-users or programmers using components in their own application—to modify certain properties of the component. However, users can only modify those properties foreseen by the original author; they cannot independently add features except through the basic object-oriented techniques of inheritence.

After application composition, the classes are eventually loaded into the environment. During execution, the bytecodes can be translated into native local platform instructions by a Just-In-Time compiler (JIT). JITs only reimplement the bytecodes in a different language; they do not add new functionality (although JITs may transform the code for optimization, for example unrolling loops or reordering instructions).

2.2 Load-time Transformation

The architecture of the JVM, in which classes are loaded on demand by a user-extensible class loader, offers a complementary alternative to the previous steps: load-time transformation, in which the loader is responsible not only for locating the class, but for transforming it in ways specified by the user.

Load-time transformation is precisely late enough that the transformation cannot burden other users (as it would if it were performed at, say, component integration), and yet early enough that the JVM is unaware that any transformation has taken place, and the transformed class is still verified by the JVM before it is accepted.

A transformation registered with a class loader can be applied to all classes—or some specific subset—that are eventually loaded into the machine.

[1] Proper pronunciations include *joy*, *joey*, or *zhwa*, depending on your cultural preference.

Stage	Example Use	Example Tool
Pre-processor	macros or conditional compilation	cpp
Compiler	translation from source to classfile	javac
Post-processor	Instrumentation	ATOM, BIT, ObjectStore
Component Integration	Setting text, color	BeanBuilder
Load-time	User-supplied transformation, templates	ClassLoader, JOIE
Just-in-time compilation	Compilation to native code	JIT

Table 1: Stages in the program development life cycle.

2.3 Other Related Work

The idea of load-time transformation itself is not an especially new one: for example, many operating systems and programming environments support dynamic linking, which binds references to library routines at load time. However, these transformations do not alter semantics of the programs themselves (although the semantics of the routines could vary from library to library), and users cannot specify the type or scope of the transformation.

The power of load-time transformation in Java has been recognized by other groups recently. An implementation of parameterized types (i.e. templates) in Java [AFM97] uses the loader to instantiate an appropriate new class. Binary Component Adaptation [KH97] uses load-time transformation to ensure compatibility between third-party components, by performing such symbolic manipulations as renaming methods, marking classes as implementing common interfaces, and restructuring class hierarchies and relationships.

Two other research projects deal with late binding of functionality to objects and classes. "Subject-Oriented Programming" [OHBS94] is a model of object-oriented programming that allows late composition of components from multiple, independently-developed groups of classes. SOP allows new fields and methods to be added, possibly overriding old definitions. "Aspect-Oriented Programming" [KLM+97] emphasizes that different *aspects* of a class (such as the base algorithm, desired precision, data structure, etc.) should be written seperately, to be woven together later into a single class; this simplifies the programming model for modules that contain different aspects.

3 The JOIE Environment

This section outlines how load-time transformation works in the JOIE environment. We first describe how transformers are installed and invoked using the JOIE class loader. We then present the key facilities of the JOIE toolkit that allow transformers to parse, analyze, and modify the transformed classes.

3.1 The JOIE ClassLoader

Before a class can execute in a JVM it must be loaded by a class loader. The JVM invokes a class loader to resolve a reference to a class that has not yet been loaded. The class loader is responsible for locating the missing class file, fetching it from the file system or a network server, and returning it to the JVM. The JVM then *verifies* the new class to ensure that it is semantically valid and safe. To allow for flexibility in loading and instantiating classes, the Java environment allows users to define new class loaders as subclasses of `java.lang.ClassLoader`.

JOIE supports load-time transformation in a special subclass of the ClassLoader. The JOIE ClassLoader exports methods for registering transformers, and it applies registered transformers to each loaded class after fetching the class file into memory but before submitting it to the JVM for verification. The JVM has no way to determine that any changes were made from the original class (although we have adopted the convention of marking transformed classes as implementing an interface `Transformed`, and recording the transformer responsible).

To expose the class internals to the transformers, the JOIE ClassLoader creates a `joie.ClassInfo` object for each class as it is loaded. The transformers access the features of the JOIE toolkit by invoking methods of `ClassInfo` and related classes, as described below. Section 4.1 deals with the internals of the JOIE `ClassInfo` class in more detail.

3.2 Transformers and the ClassLoader

JOIE transformers are written in Java. This design choice offers three important benefits. First, transformers have access to the full power of the Java language, including procedures and variables, conditional logic, and arbitrary control flow. Second, Java vastly simplifies JOIE's design, as no special interpreter for transformers

is needed. Finally, since the transformers are executed by the JVM, they are subject to the same safety checks as any Java program.

A JOIE transformer is simply a Java class implementing the interface `joie.ClassTransformer`. To install a transformer, the user installs a bootstrap wrapper that instantiates the JOIE ClassLoader and the desired transformer classes, and then registers the transformer objects with the ClassLoader. The wrapper then requests the ClassLoader to load the main class of the application to execute; the ClassLoader automatically invokes each registered transformer for each class subsequently loaded by the application. The current version of the JOIE ClassLoader rejects attempts to register new transformers once the first class is loaded. This effectively prevents untrusted application code from installing transformers. This issue is discussed in more detail in Section 7.2.

The core of each transformer is implemented in its `transform(ClassInfo)` method, which is invoked by the ClassLoader to apply the transformation. Multiple transformers can be chained together by passing the `ClassInfo` object for each target class to each transformer in sequence. The ClassLoader invokes the transformers in order of priorities specified at registration time. Within the method `transform`, transformers may call upon the JOIE toolkit routines for reflection, class modification, or bytecode modification, as described in the following subsections.

3.3 Load-time Reflection

A transformer uses the *reflection* portion of the JOIE API to expose the structure of the target class, including symbolic information, fields, methods, interfaces, and attributes. Using the API, a transformer can determine the name, signature, and modifiers (public, synchronized, static, etc.) for any class member. Given a class, it can identify the superclass and any implemented interfaces. Most importantly, a transformer can directly access and browse the class methods and their bytecode instructions.

These reflection features of the JOIE toolkit are similar to those present in a number of languages to allow the discovery of the structure of classes at run time. Java added runtime reflection functionality in the 1.1 release of the Java Developer's Kit (JDK). However, the Java reflection API is available only after the class has been loaded into the JVM. This is too late for load-time transformation, which seeks to transform the class before loading takes place. Additionally, reflection was not designed to extend functionality, and so does not make available the implementation of class methods. Method implementations are accessible through the `javap` disassembler included in the standard Java Developer's Kit (JDK), but `javap` runs from the shell and prints to its standard output; it is not integrated into the Java reflection API, nor does it produce a data structure that can be manipulated by the program.

JOIE's reflection features are useful for a wide variety of analyses, both in their own right as program analysis tools and as the basis for more powerful transformations. For example, JOIE can be used to construct the class hierarchy, build a call graph, or create interprocedural control-flow or dataflow graphs.

3.4 Class Modification

Transformers call upon the JOIE API not only to discover elements of classes through reflection, but also to change or modify those elements. That is, given access to a class (`ClassInfo`) or a member of a class, a transformer can change aspects of the class implementation. For example, transformers can: set or unset modifiers; add, remove, or rename fields or methods; change method signatures or field types; adjust the list of interfaces implemented by the class; adjust references to fields or methods to point to new fields or methods; adjust the value of embedded constants; or manipulate the inheritance hierarchy.

The structure of Java classfiles allows the JOIE toolkit to implement a wide range of class modifications without the need to modify other classes that refer to the target class. This is because all references to class members, including external references, are symbolic and stored in a *Constant Pool*. Section 4.1 discusses implementation issues for class modification in JOIE.

3.5 Bytecode Modification

The power of JOIE stems primarily from its ability to modify the bytecode instructions as well as the fields and properties of a class. Given a `ClassInfo`, the JOIE reflection API allows the transformer to iterate through each `Method` of the class. Given a `Method`, the API exports an array of `Instruction` objects along with ancillary information such as parameters to the method, maximum stack and frame size, exception handling information, etc. The transformer can reorder or replace instructions, insert new instructions, alter the stack depth, reassign values to different frame locations, or modify exception handling by inserting new handlers or modifying the range of instructions protected by a handler. One important use of these features is to *splice* new code into the existing implementation, changing the behavior of the method without affecting the code that is already there.

Class	Method	Explanation
ClassInfo	getMethods	returns array of Methods
ClassInfo	getFields	returns array of Fields
ClassInfo	addField	inserts new Field in class
Field	set	sets a flag (such as private)
Method	getCode	returns Code for Method
Code	getInstructions	returns array of Instructions
Code	getExceptionTable	returns exception-handler table
Code	insert	inserts new Instructions into Code
Instruction	getMnemonic	returns string mnemonic
Instruction	getOp	returns first operand of instruction

Table 2: Selected methods from JOIE API.

Transformers can use JOIE's bytecode modification features for a variety of purposes. For example, transformers may instrument classes by inserting static method calls to global analysis routines, or instructions to increment per-instance counters. Another use is to add exception handling code to allow classes to run in an environment that can produce exceptions not anticipated or handled by the original class author. Such a transformer would contain a list of unsafe operations together with a body of code to deal with exception types that could be raised by those operations. The transformer installs the code as an exception handler in each transformed class, and updates the classfile's table of exception handlers to reflect which regions of code are protected by that handler.

Splicing can be combined with class modification to enable a powerful new way to extend functionality. The transformer can add new generic methods that implement some functionality within the class, then splice in calls to those methods at appropriate points. For example, a transformer that adds persistence methods for moving objects to and from a persistent store could also splice calls to those methods in constructors and finalizers. Section 5 outlines potential applications in more detail, and Section 6 presents a case study of such a transformation.

4 Implementation of the JOIE Toolkit

This section outlines the implementation of the JOIE toolkit in more detail, and addresses several implementation issues for transformation in the Java environment.

4.1 ClassInfo

JOIE transformers operate on `ClassInfo` objects, which encapsulate and export all of the information present in class files. The JOIE ClassLoader creates a `ClassInfo` object for the class before invoking the first transformer, then passes the `ClassInfo` to each registered transformer in sequence. After the last transformer completes, the ClassLoader converts the `ClassInfo` into a transformed classfile byte array in memory, and submits it to the JVM for verification.

To improve performance, the JOIE toolkit methods parse the elements of the target classfile lazily as they are requested by transformers. For example, if a transformer simply adds a new interface to the list of interfaces implemented by the class, then `ClassInfo` need not parse the list of fields, methods, or bytecodes. Since each `ClassInfo` instance preserves parsing state across transformers, each element of a transformed class is parsed at most once.

Java classfiles retain a great deal of symbolic information in a data structure known as the *Constant Pool*, including variable-length symbol entries for all methods, fields, and constants defined or referenced by the class. Entries in the Constant Pool may address other entries by index. For example, an entry for a method `Banana.peel()` contains the index of the entry for class `Banana` and the index of an entry holding the name and type of the method. The entry for class `Banana` in turn contains the index for the UTF-8-encoded string "Banana." The name and type are also stored as indices that point to encoded strings for the name ("peel" in our example) and the descriptor, which encodes parameter and return types.

All references in instructions — including field accesses and method invocations — are represented as indices pointing to symbolic names stored in the Constant Pool. The JVM resolves external references by symbolic name as the program executes.

This structure allows the JOIE toolkit to implement a wide range of load-time class modifications easily. In particular, it is rarely necessary to update classes that reference the transformed class, since all references into the

target class are resolved by symbolic lookup in their own Constant Pool. Moreover, many class modifications can be implemented simply by adding or manipulating individual entries in the Constant Pool. In particular, it is unnecessary to modify the bytecode instructions in the target class unless the transformer explicitly requests it. A change to the Constant Pool affects method instructions only if it changes the index of an entry in the Constant Pool. Fortunately, there are no ordering constraints on the Constant Pool, so any new entries can be appended rather than inserted in the middle, preserving the indices of existing entries. Unreferenced entries are simply left in place.

As the JOIE toolkit parses the Constant Pool, it creates a list for each type of entry, storing each symbol value and its index. This allows us to more quickly add new entries without duplication. For example, to insert a new entry for a Field we first search the list of field entries to ensure that an entry with the requested class, name, and type is not already present. If not, we construct a new field entry, and must then search for or create appropriate class, name, and type entries.

4.2 Splicing and Inserting Bytecodes

In a typical JVM, each method invocation runs with an operand stack and a single index-addressable *frame* whose size is statically determined. JVM bytecodes operate on elements at the top of the operand stack or move operands from the frame to the operand stack or vice versa. For example, the `iadd` instruction pops the top two elements off the stack as integers, adds them, and pushes the result onto the stack. Similarly, a method invocation pops the target of the method and the appropriate number of parameters, and places the result (if any) onto the stack.

This architecture has interesting implications for bytecode modification. First, being stack instructions, bytecodes are sensitive to placement and ordering. For example, simply inserting a method call instruction into a sequence might consume the wrong value off the stack, and leave an unexpected value on the top. In general, splices must be *stack-neutral*, i.e., the spliced code must leave the depth and types of the stack unchanged. However, the values of the entries in the stack may be changed, and the splice may have other side effects that affect the rest of the method, such as modifying the frame or some object.

A second issue is that all branches are relative, so inserting instructions between a branch and its destination will make the destination field incorrect. To preserve the original control flow, JOIE must correctly update the destination field. JOIE solves this by applying modifications to collections of Instruction objects linked to the `ClassInfo`. The Instruction objects represent branch targets as pointers to the destination Instruction. A similar approach is used to update the exception handler table, to preserve the binding of exception handlers to ranges of instructions. In each case, JOIE regenerates the relative addresses when it linearizes the `ClassInfo` to a classfile. For load-time transformation, the JOIE ClassLoader generates the transformed in-memory classfile with the correct relative offsets before submitting it to the JVM for verification.

4.3 Modifying the Frame

As described above, each method invocation places the method's arguments and local variables in the frame. Thus any method transformers that change the number of arguments or local state of the method must modify the method's frame.

Modifying the frame is somewhat trickier than modifying the Constant Pool. In particular, the JVM specification requires that the arguments to the method appear in order at the low end of the frame, before local variables held in the frame. This means that adding a new argument to a method may displace existing local variables or even other arguments. In these cases, the transformer must update the method's instructions to redirect loads and stores from one frame location to another.

Relocating references to frame locations is simple in concept but more complex in implementation. There are twenty-five separate instruction types responsible for loading operands from the frame to the stack, and twenty-five complementary instructions for storing from the stack to the frame. Since the instruction set is strongly typed (for ease of verification), each type (integer, long, float, double, and reference) has its own family of loads and stores. Moreover, JVM load and store instructions encode their operands in any of several different ways. For example, references to frame locations 0, 1, 2, and 3 may use efficient "shortcut" instructions with the location encoded in the opcode. If the frame has more than 256 locations, load and store instructions that reference locations at the high end of the frame must be prefixed by a "wide" bytecode, indicating that the frame offset is specified as two bytes rather than one. In these cases, changing the frame offset of the operand may force a change to the opcode itself, or make it necessary to insert or delete a "wide" prefix. For example, the instruction could be a shortcut that references frame location 3; if the new target is location 5 then the opcode must be changed to take an explicit operand, since there is no shortcut for frame location 5. JOIE correctly generates the correct opcode and operand in the face of such changes.

4.4 Performance Issues

Delays for loading classes are already substantial, given the high latency of loading the class from disk or from across the network, and the time required to verify the class. Even so, the performance of load-time transformation is a critical concern, since the overhead of transformation may be perceived by the user.

Despite the incremental parsing optimization, the class parsing and modification code in our current prototype is fairly slow. While we have put little effort into performance tuning at this point, initial experiments with simple JOIE transformers show that it takes milliseconds or even tens of milliseconds to parse and transform typical classes. We did these experiments on 167 MHz Sun Microsystems Ultra 1 systems with Solaris 2.4, and using Sun's JDK 1.2 beta 3 and JIT compiler.

We plan to improve our current JOIE prototype in several important ways. `ClassInfo` currently represents all internal data structures (lists of methods, fields, interfaces, and instructions) as Java arrays. This makes access fast (as compared to Vectors, for example), but it also makes insertion extremely expensive. In a future release we plan to instead use the Java Collection interface available in JDK 1.2. In addition to improving performance, this will allow transformers to use the iterators and update primitives in the Collection interface to access the Methods, Fields, and Instructions of each target class, streamlining the JOIE interface and implementation.

Adding fields or methods to a class often requires repeated searches of the Constant Pool, since we are forbidden from repeating any entry. Currently these searches are linear; an index structure would significantly reduce the cost of adding and accessing entries in the Constant Pool during transformation.

5 Example Applications

There is a vast array of different types of applications of JOIE. In this section we consider three broad categories: extending Java, integrating classes with a system infrastructure, and adding functionality to classes. These examples are meant to be suggestive, not exhaustive. Other uses of JOIE include instrumentation, program analysis, and optimization.

5.1 Extending Java

There is always a demand for new features in Java, but it is difficult to extend the environment while preserving Java's "Write Once, Run Anywhere" promise. New features for Java could be implemented at a number of different levels: the source language, the compiler, the JVM, or the JIT compiler. JOIE adds a new level where extensions to the environment can be prototyped easily and safely. Once the extensions prove useful, they may be implemented at other levels, or perhaps remain implemented as a JOIE transformer. More generally, transformers can be used to run non-conforming code on standard JVMs, or conforming code on non-standard JVMs.

For example, consider the demand for support for complex numbers as primitives. While a developer could add such a feature to a compiler and matching JVM, classfiles produced by the compiler would not run correctly on any standard JVM. Alternatively, one could use a Complex class, with appropriate methods for adding, multiplying, etc. This solution is portable but inefficient.

However, a transformer could recognize the instantiation of Complex, and replace each instance with a pair of floats, and similarly replace method invocations with inline floating point operations. The transformed code will incur lower overhead than the relatively expensive method invocation, while conforming to the JVM standard. More importantly, the classfile running in a system without JOIE will still run correctly, if more slowly. Adding such a feature to Java does not require the transformer author to rewrite or even look at JVM source code.

A different style of extension could add new bytecodes to the JVM implementation. For example, the performance of numerical analysis programs may be improved by augmenting the instruction set with vector bytecodes implementing matrix operations such as sum, dot product, min, max, etc. A compiler could easily generate code using the new bytecodes, but the code would not run on any JVM that does not implement them. Assuming that a transformer could recognize these constructs, it could replace the standard long version of the code with the new bytecodes (or vice versa).

Parametric types are another example. The Thor group proposes an implementation of parametric types for Java [MBL97] that requires a modified JVM and compiler. A load-time transformer could translate a less efficient but standard implementation to use special new instructions.

More ambitious language extensions could add pre- and postconditions to methods [Mey92], continuations, closures, or multimethods. An implementation of "security-style passing" at Princeton [WF98] uses JOIE to achieve a feature similar to continuations, making some subset of the state of the computation available at all times.

5.2 Integrating Classes with a System Infrastructure

Other transformations change classes to integrate them into some runtime environment. This idea generalizes the use of ATOM or BIT to instrument code to drive an on-the-fly simulation, by inserting code to report selected events (e.g., branches or method calls) to the simulator. This technique can be used for other purposes as well. Some possibilities include integrating imported classes with local system environments that support object caching, distribution, debugging, or visualization.

The transformers for these environments have similar structures. First, the transformer must identify the points in the code that handle events of interest to the system environment, e.g., object instantiation or method invocation. The transformer then splices calls to the environment at these points, passing any necessary information as arguments.

For example, a debugging environment may be interested in the values of parameters to methods, return values, etc. A debug transformer would identify areas of interest and insert calls to the environment, allowing a user perhaps to visualize and see results of the computation. The advantage of this approach is that the debugging calls are inserted only at load time; if the user does not require them, they are not inserted at all, reducing the size of the transported class. Also, note that running the debug version of the code requires no special JVM, and only areas of interest are transformed.

5.3 Adding Functionality to Classes

More ambitious transformers seek to extend the functionality of classes themselves as they are loaded. While any sort of functionality could be added in principle, this approach makes most sense when the new functionality is orthogonal to the implementation of the class, such as persistence or logging. This makes the transformer easier to construct, and also creates a logical separation between the code of the original class and the new functionality. Users who are uninterested in that specific function avoid the cost of downloading, verifying, and storing the unwanted code.

An existing approach to the late extension of classes is known as *mixins*, which are implemented in LISP [MSW83] and other languages. Mixins are groups of methods and fields that are added to a class definition sometime after compilation. Traditional mixins are passive: they add methods to the class, but those methods are called only from outside of the class, and never by the original methods.

JOIE can be used to implement a stronger model, *active* mixins. An active mixin is structurally similar to a traditional mixin. In addition to adding fields, methods, or interfaces to an existing class, active mixins parse existing methods and insert or replace instructions. For example, a mixin for versioning would contain methods for reading the appropriate version, and writing changed instances of objects to a storage system. However, these methods must be explicitly called by external code. An active mixin for versioning would not only add the methods, but also insert a read call in the constructor, and a write call in the finalizer.

Recoverability could also be implemented as an active mixin. Methods could be made recoverable by inserting logging code: whenever these methods finish, parameters or return values are written to disk, by calling methods supplied by the mixin author (possibly inserted as methods into the class, or an auxiliary class). With the addition of a mixed-in method to read the log and begin computation at the point of failure, simple recoverability could be added to a class.

6 Case Study: Automatic Observables

In this section, we describe an active mixin called Automatic Observable, and discuss its implementation in JOIE.

6.1 Observer/Observable

The Observer pattern is a well-known design pattern (also known as Model/View), discussed in [GHJV95]. In it, whenever an *Observable* object changes state, its *Observer* classes are notified. This pattern is used when there are common classes that encapsulate state, and other classes that represent a "view" of that state.

Sun's JDK provides an implementation of Observer/Observable in Java. In it, Observer is an interface containing the single method `Update(Observable, Object)`, which is called by an Observable object's `notifyObservers` method whenever it wishes to inform its Observers that its state has changed. User classes acting as Observers need only implement the interface, call the `addObserver` method of an Observable instance, and they will be able to begin observation.

In contrast, Observable is implemented as a class which includes methods for managing Observer lists and notifying Observers, as well as code to check, set, and clear a hasChanged bit. Programmers define Observable classes by subclassing the `Observable` base class. Unfortunately, requiring authors to use a specific base class in this way is often limiting. For example, third-party software cannot be made Observable without changing an existing class hierarchy.

```
class AutoObservable implements ClassTransformer {
    ClassInfo transform(ClassInfo cinfo) {
        cinfo.addField(Type.BOOLEAN, "dirty");
        Instruction[] splice = code.putField("dirty", true);
        Method[] meths = cinfo.getMethods();
        for(int i=0; i<meths.length; i++) {
            if(!meths[i].isSet(Modifier.STATIC)) {
                Code code = meths[i].getCode();
                Instruction[] insts = code.getInstructions();
                for(int j=0; j<insts.length; j++) {
                    if(insts[j].getMnemonic().equals("putfield")) {
                        code.insert(j+1, splice);
                        insts = code.getInstructions();
                        j += splice.length; }   }   }   }   }   }
```

Figure 1: Code Fragment From JOIE Transformer for Automatic Observable

6.2 Observable Mixins

The problems with the Observable base class could be avoided through the use of mixins. An Observable mixin would mix the Observable code, including the hasChanged bit, into an existing class, without modifying the inheritance hierarchy. However, with this approach, the class author is responsible for properly instrumenting the code to manage the hasChanged bit and call notifyObservers at appropriate times. Unfortunately, the traditional mixin model is not capable of changing existing methods to include new method calls.

However, Observable is easily implemented as an active mixin using a JOIE transformer. Figure 1 shows a fragment of the code for the JOIE transformer for Observable. The implementation resides in the single method `transform` called by the JOIE ClassLoader (of course, transformers may have additional methods). The transformer first adds a new boolean field, "dirty", to the class. It then scans each instruction of each non-static method defined in that class. If the instruction changes an instance field (e.g., the `putfield` instruction), the transformer inserts a code sequence that sets the dirty bit. The counter skips over those new instructions, and the scan continues. Not shown in the example is code that inserts the method call to notify the observers, or to mix in the methods to return the dirty bit, notify observers, or manipulate the list of observers.

This transformer is meant to be illustrative only. A more efficient scheme would build and traverse the control flow graphs, setting the dirty bit at most once per path, and adding the notify call only to exit blocks that could follow a block containing a set to dirty. A more universal solution would also perform an interprocedural search on member objects, and also handle inheritance (only a superclass needs to have the methods and the dirty bit mixed in, but all subclass methods must be scanned and spliced).

7 The Key to Pandora's Box?

Introducing a new phase to specify functionality raises a number of difficult issues. These are generally a result of the fact that traditional tools, from design tools to debuggers and security systems, were not designed to take load-time transformation into account.

We suggest that load-time transformation will be a permanent part of the programming model in the future. However, much work remains to be done to develop a safe, general, and practical methodology for load-time transformation. Here, we touch on four important issues: debugging, security, legal issues, and the need for a framework to guide transformer authors in writing transformers that are general and correct as well as powerful.

7.1 Debugging Transformed Code

When a bug in a transformed application arises, it is often unclear if it is a bug in the original code, a bug in the transformation code, or a bug that arose from the interaction of the two. Debugging JOIE transformers today demands an understanding of the both the original code and the transformer. The problem is particular severe if multiple transformations were active. JOIE currently has no support for controlling interactions among transformers.

Some specific debugging support is essential to facilitate load-time transformation as a useful programming tool. At the very least, the environment must be able to

determine if the failed code was transformed, if transformed code appears in the call chain, and which transformers were responsible for any transformed code. Basic information can be made available to the environment by tagging transformed classfiles in JOIE.

7.2 Security

A natural reaction when presented with JOIE is to worry that it will break the security guarantees made by Java, e.g., the promise that imported code cannot gain unauthorized access to resources or memory. With respect to security, JOIE transformers are no more powerful than any imported code. Put another way, code produced by JOIE is as safe as code produced by an untrusted compiler. However, it is important to develop a rigorous understanding of the effect of transformers on security. For example, while class authors choose their compilers, they do not necessarily choose their transformers; a transformed digitally signed applet may not be as trustworthy as the original.

Java security is supported by two main pillars: the verifier and the SecurityManager. The Java verifier guarantees, among other things, that code is type-safe, and particularly that pointers cannot be created or manipulated. This prevents a malicious or buggy program from walking through memory and reading or writing where it normally would not have access.

The SecurityManager is a user-extensible class that is asked by library code at execution time whether specific classes have permission to access specific resources. For example, a program loaded over the Internet might not be permitted to read from the filesystem, but it could spawn a new thread.

Since transformations are applied before the code is verified, transformed code must still comply with the same type-safety restrictions as non-transformed code. The SecurityManager has the same ability to restrict transformed code's access to system resources as before.

There are also important security policy issues. For example, is it safe for classes to instantiate and register new transformations during the execution of the program? Can transformers be loaded and applied from over the network safely? Currently, JOIE's default policy is to disable registration of transformers as soon the first class is loaded, but this can be overridden by the user. There is also the question of how to determine the access permissions of a transformed class: can classes ever gain or lose permissions due to being transformed?

7.3 Legal Concerns

An additional concern covers legal issues, which are beyond the scope of this paper and the authors' experience. One concern is that JOIE, like many utilities, including decompilers, disk copy routines, and binary editors, could be used to violate copyrights. However (like those other tools), it has legitimate uses that remain within the law.

A more complex and troublesome issue regards liability: is a failure the result of the original class, the transformer, or both? Who has the ultimate responsibility?

7.4 Usability

To be useful, the software should also be convenient to use. Currently, transformer authors must be conversant with the bytecode structure of Java and have a deep understanding of classfile structure and layout. A set of abstractions that sit on top of JOIE might provide a framework in which less experienced programmers can construct useful and correct transformations. We are investigating the development of such a framework.

8 Conclusion

This paper describes JOIE, a toolkit for automatic, user-directed transformation of Java classfiles. Program transformation using JOIE can be the basis for a wide variety of extensions, including instrumentation, mixins, automated inclusion of code for visualization or debugging, and bytecode replacement.

A powerful feature of JOIE is its ability to apply transformations at load time to meet the user's site-specific needs. Load-time transformations can include changing the implementation of the original methods of the class, introducing a new model for extending the functionality of Java classes.

Load-time transformation is well-suited to use with transportable code in the Internet. It allows software consumers to assemble customized applications by combining software modules and transformers obtained from a variety of sources. To illustrate the potential of load-time transformation, we outline several examples of how it can be used to extend the Java programming environment, integrate classes with system facilities in the local environment, and adapt or extend third-party classes. To show how these transformations would work, we described in detail the Automatic Observable transformer, an *active mixin* that adapts existing classes to use the Observer/Observable design pattern.

The introduction of program transformation — and load-time transformation in particular — raises many interesting research issues and practical concerns that must be addressed. A fundamental question is: when is it appropriate to specify functionality as a transformation? Other issues include the problems of debugging

transformations and transformed code, security and legal concerns, and performance. Most of these issues are common to any introduction of a new stage in the program development lifecycle. This paper does not attempt to present complete answers to these questions. Rather, the contributions of the paper are to show the potential of load-time transformation, present a prototype toolkit as a basis for further experimentation, and identify the key issues raised by the technique.

There are a number of features that we plan to add to JOIE, including more advanced dataflow analysis for methods. We are also extending a Java compiler to pass programmer-supplied hints to JOIE. Such hints would be useful, for example, to indicate blocks of code that could be transformed to superoperators[Pro95], such as dot product or a method invocation on a parametric type. Also, JOIE adds functionality at the class level only. We are considering extending this idea to instances, allowing orthogonal features to be added on an instance-by-instance basis.

Our plans for future work with JOIE include exploring new uses for load-time transformations. JOIE is most exciting when viewed as an enabling technology. We believe that JOIE allows a fundamentally more powerful way to express functionality, and intend on exploring the possibilities. The Automatic Observable active mixin suggests that transformers can automate other design patterns [GHJV95] as well. Other promising possibilities include automatic hooks and adaptation for system facilities such as persistence, distribution, recoverability, survivability, caching, and mobility.

9 Acknowledgements

We'd like to thank Mike Fox, Jim Gray, Manoj Kasichainula, R. Adam King, and Zhiyong Li of IBM Research Triangle Park for assistance and guidance from the earliest stages of the project. Thanks to early users Jonathan Seeber and Anya Bilska for putting up with bugs. Siddhartha Chatterjee and Jan Prins of UNC-Chapel Hill, and Owen Astrachan and Robert Duvall of Duke University all made very useful comments on drafts of this paper. Chris Laffra and Lee Nackman of IBM's Thomas J. Watson Research Center gave advice on bytecode manipulation, and the attendees of the IBM Research Java Conference gave many useful comments. Thanks also go to Dan Wallach, the anonymous reviewers, and our shepherd Benjamin Zorn.

References

[AFM97] O. Agesin, S. Freund, and J.C. Mitchell. Adding Type Parameterization to the Java Language. In *Proceedings of the Symposium on Object Oriented Programming: Systems, Languages, and Applications*, pages 49–65, 1997.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley Publishing Company, Reading, Massachusetts, 1995.

[GJS96] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. Addison Wesley Publishing Company, Reading, Massachusetts, 1996.

[KH97] Ralph Keller and Urs Hölzle. Binary Component Adaptation. Technical Report TRCS97-20, Department of Computer Science, University of California at Santa Barbara, December 1997.

[KLM+97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-Oriented Programming. Technical Report SPL97-008 P9710042, Xerox Palo Alto Research Center, February 1997.

[LY97] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison Wesley Publishing Company, Reading, Massachusetts, 1997.

[LZ97] Han Bok Lee and Benjamin G. Zorn. BIT: A Tool for Instrumenting Java Bytecodes. In *The USENIX Symposium on Internet Technologies and Systems*, pages 73–82, 1997.

[MBL97] Andrew C. Myers, Joseph A. Bank, and Barbara Liskov. Parameterized Types for Java. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 132–145, January 1997.

[Mey92] Bertrand Meyer. *Eiffel: The Language*. Prentice Hall, 1992.

[MSW83] David Moon, Richard Stallman, and Daniel Weinreb. *The Lisp Machine Manual*. AI Lab, MIT, Cambridge, Massachusetts, 1983.

[Obj98] Object Design Inc. ObjectStore PSE Resource Center, 1998. `http://www.odi.com/content/products/PSEHome.html`.

[OHBS94] Harold Ossher, William Harrison, Frank Budinsky, and Ian Simmonds. Subject-Oriented Programming: Supporting Decentralized Development of Objects. In *The 7th IBM Conference on Object-Oriented Technology*, 1994.

[Pro95] Todd A. Proebsting. Optimizing an ANSI C Interpreter with Superoperators. In *Proceedings of the 22th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 322–332, January 1995.

[Rat98] Rational Software Corporation. Purify, 1998. `http://www.pure.com/products/purify`.

[SE94] Amitabh Srivastava and Alan Eustace. ATOM: A System for Building Customized Program Analysis Tools. In *Proceedings of the SIGPLAN '94 Conference on Programming Language Design and Implementation*, pages 196–205, June 1994.

[SG97] Daniel J. Scales and Kourosh Gharachorloo. Towards Transparent and Efficient Software Distributed Shared Memory. In *The Sixteenth ACM Symposium on Operating Systems Principles*, 1997.

[WF98] Dan S. Wallach and Edward W. Felten. Understanding Java Stack Inspection. In *1998 IEEE Symposium on Security and Privacy (to appear)*, May 1998.