Recombining Concerns: Experience with Transformation

Geoff A. Cohen[*]
Department of Computer Science
Duke University
gac@cs.duke.edu

Abstract

A key technology to enable separation of concerns is the final step of reintegrating multiple, possibly independently-developed modules. We argue for a technique of *transformation*, where the process of modifying a class to exhibit new behavior is controlled by a user-supplied program. Additionally, we argue that performing such a process as a program is loaded offers benefits in flexibility, performance, and reusability. We refer to this as *load-time transformation*. We describe the advantages of such a technique, present the implementation of our prototype, JOIE, and discuss our experience in developing a transformation that adds the capability of running consistent replicas over multiple machines.

1 Introduction

At some point in the lifecycle of software developed with a separation-of-concerns methodology, the different decomposed modules must be combined (or recombined) to produce a whole program.

The semantics and power of separation will be greatly affected by three choices made in determining how the recombination is performed:

- When in the software development lifecycle does the recombination occur?
- How are recombinations specified?
- How are conflicts between multiple modules resolved?

In this paper we discuss *load-time transformation*, a general technique that features a great deal of user control over the process of recombining modules. This process of modifying classes occurs as the program is laoded into the user's environment.

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

We have a working and publically available implementation of load-time transformation for Java, and we discuss our experience using it, in particular to develop a facility to enable automatic replication facilities to programs.

2 Load-time Transformation

In this section we present the structure and benefits of performing the recombination of concerns as software is loaded into a user's runtime environment.

2.1 Choosing a Time

Is the recombination of concerns a tool for software developers, or for software users? Phrased differently, what is the end product of software developed using a separation methodology?

As a tool for program development, producing a complete program, separation of concerns offers significant benefits. Developers may be more productive by exploiting increased modularization, and be able to produce higher-quality code in less time. This role of the separation methodology has the advantage that it does not require software users to change their own methodology or even be aware of a more sophisticated development system.

However, by delaying the recombination of concerns (or allowing new concerns to be recombined much later), we can reap a number of benefits:

- Users (or more likely, deployers) can customize software to their own environments and needs. This may even include removing concerns that have been added by developers but are not desired.
- Modules, if written in an independent fashion, can be applied to many different programs, increasing code reuse.
- Code when delivered is slimmer, especially noticeable if software is downloaded over a network.

The Java Virtual Machine's architecture, which offers dynamic and user-extensible loading, is an excellent opportunity to accomplish late, user-controllable binding of concerns into programs. By subclassing the class java.lang.ClassLoader, we provide a nearly transparent system that operates on standard Java classfiles, but can dynamically modify the program as it is loaded, adding code fragments throughout the class to deal with additional concerns.

2.2 Transformers

A key ingredient in this architecture is that the process itself of modifying the code (or *transformation*) is user-programmable. Merely combining different classes (as one does in mixins) is sufficient for many important applications, but fails when multiple concerns must be integrated, or when the implementation of concern is highly dependent on the existing implementation it is affecting.

Our design of load-time transformation relies on user-provided *transformers*, Java classes that take a Java class as input and produce an appropriately modified class as output. Transformers can introduce new methods and fields, add code fragments to existing methods, edit the existing code, or modify how the class interacts with other classes.

By implementing these changes in Java, transformers can be intelligent about how these modifications are made. Rather than blindly adding the same mixin to all classes, the mixin could be dynamically generated or adapted to specific circumstances. For example, a transformation that adds a capability security system to a class can examine all methods of that class, determine which are leaf methods and which may call other methods, and optimize the placement of capability checks.

3 JOIE

JOIE [CCK98] is our prototype for load-time transformation. It supports the addition of mixins, method prefixes and suffixes, instrumentation, the creation of proxy classes, the redirection of method calls to new call sites, and general bytecode editing.

It is freely available for research and academic purposes. It has been used in a number of projects, including *security-style passing* from Princeton's Safe Internet Programming Team [WF98]; Naccio, a code-safety system from MIT[ET99]; the Correlate parallel and distributed programming project from K.U. Leuven[RJM$^+$98]; the Kava metaprogramming system from the University of Newcastle-upon-Tyne; Ivory, a distributed replica system from Duke[BCC$^+$99], and others.

What these programs have in common is that they specify a facet of a program, such as the security architecture, a method reification system, a distributed naming system or whatever, as a transformation from a set of code that does not include this concern into a program that includes the new and modified code.

3.1 Experience

We used JOIE in our implementation of Ivory[BCC$^+$99], a system that adapts dynamic web services to run on multiple replicas, while keeping shared data consistent. Here, we briefly describe the transformers used in Ivory to render programs replicable.

There is a suite of (currently) four transformers that are run on applications, enabling those applications to be cacheable in an Ivory proxy server. The transformers fall into two main categories: data transformers, which edit elements of the application's data, and application transformers, which edit the logic of the application itself.

1. DirtyTransform. This transformer performs two main functions: first, it mixes in the (slightly misleadingly named) class ConsistentImpl into the target. This includes the addition of a private transient boolean field, Ivory-dirty, and a public method Ivory-clear(), which sets that field to false. This allows Ivory to monitor whether objects have been modified since it last sent a copy of that object to a replica.

   Secondly, the transformer examines the code of the methods of the target class, and places a splice of code after each putfield instruction (the instruction used to set the value of a class's field). The splice checks to see if Ivory-dirty is true. If it is not (i.e. the object is clean), Ivory-dirty is set to true, and the object is placed in the global dirty list.

2. ArrayTransform. This is a special case of DirtyTransform for arrays. Since we cannot transform arrays to have a dirty bit, we insert code in classes that contain arrays as fields. That code performs a complicated stack rotation that places the array reference on top of the stack and sends the array to the global dirty list.

3. AutoWriter. This transformer generates new methods, readExternal and writeExternal, marking the class as implementing the interface java.io.Externalizable. The writeExternal method walks through all of the non-static and non-transient fields of the object, putting the value of that field on the stack and calling the appropriate method of ObjectOutput. readExternal does a similar action but places the values into the fields. It also calls checkcast, to guarantee that the object meets the type-safety guarantees of the Java runtime.

4. CommitTransform. This examines a class, and if it implements an interface that is a subinterface of Consistent, it brackets all Consistent calls with beginTransaction and endTransaction calls to the global state manager. This technique allows the programmer

of a class to easily indicate to the Ivory transformers which methods leave the data structures in a consistent state. This allows us to avoid the problems of shipping possibly inconsistent data to replicas.

4 Issues in Composition

4.1 The Problem

An application which employs multiple transformations may run into difficulty. While the JOIE environment is certainly capable of accepting multiple registrations and applying them in turn, the results may not be the ones desired. For example, consider two transformations, one of which instruments the code of all of the methods of a class with performance-testing code, the other adds new methods to access the performance data. These transformations must be ordered to act correctly! Adding the new methods first will result in presumably undesired performance data being collected from the performance infrastructure itself.

4.2 Relations

We can first examine the attributes of transformers based on what sorts of members of instructions they operate on, and what sort they produce. We refer to the set of elements the transformer operates on as its *domain* and the set of elements that it inserts as its *range*. For example, a transformer that inserts a new method into a class has the class as its domain, and the new method as its range. An instrumenter that adds new method calls after every write has the set of all writes as its domain, and its specific method call as its range.

We can thus begin to establish commutativity relationships between types of transformations based on their domains and ranges. Two transformers, t1 and t2, and associative if there is no intersection between t1.domain and t2.range, and between t2.range and t1.domain (and both are additive). More simply, they are *non-interfering*. For example, a transformer that adds a new method (domain = class; range = that method) is non-interfering with a transformer that retypes a field (domain = field; range = field). However, a transformer that instrumented methods (domain = methods; range = a method call) will interfere as its domain consumes the first's range.

4.3 Strict Ordering

A simple solution that imposes no runtime cost is to require the transformations to be strictly ordered. When transformers are registered with the JOIE ClassLoader, they can be given a priority, and JOIE will run them in the priority order.

This solution is sufficient for those cases in which the order of operations of the transformers can be linearized.

4.4 Invisibility

For transformations that have interfering domains and ranges, a more extreme solution may be necessary, especially in cases with circular dependencies. With invisibility, the results of one transformation (its range) will not be apparant to the introspection of the second. Thus, any possible interference is removed.

This is accomplished simply by attaching tables to the class and its methods, denoting which members or instructions originated from a given transformer. This allows the JOIE environment to, in effect, lie to transformers through the introspection API, failing to report new members or instructions.

4.5 Further Measures

We believe that the previous two solutions should be sufficient for most applications. There remain unsolved problems relating to *detecting* that a conflict is occuring, and ways to handle those less tractible conflicts. The ultimate solution, although in practice it may be prohibitively complex, is to allow the transformers to browse each other, in effect discovering what sorts of changes they may have made. This is a second level of metaprogramming, and may have a great deal of unforeseen consequences.

5 Conclusion

We discussed the advantages of load-time transformation as a technique for integrating separately-developed or cross-cutting modules. We presented our prototype, JOIE, and discussed our experiences using JOIE to implement replication as a cross-cutting concern.

Finally, we discussed some challenges that arise in simultaneously combining multiple transformations, and some techniques that may avoid conflicts in some situations.

JOIE is publically available, and we encourage researchers to download it and play with it. More information can be found at `http://www.cs.duke.edu/ari/joie`.

References

[BCC$^+$99] Geoff C. Berry, Jeffrey S. Chase, Geoff A. Cohen, Landon P. Cox, and Amin Vahdat. Toward Automatic State Management for Dynamic Web Services. In *Proceedings of the Network Storage Symposium*, October 1999.

[CCK98] Geoff A. Cohen, Jeffrey S. Chase, and David L. Kaminsky. Automatic Program Transformation with JOIE. In *USENIX 1998 Annual Technical Conference*, pages 167–178, June 1998.

[ET99] David Evans and Andrew Twyman. Flexible Policy-Directed Code Safety. In *1999 IEEE Symposium on Security and Privacy*, May 1999.

[RJM+98] Bert Robben, Wouter Joosen, Frank Matthijs, Bart Vanhaute, and Pierre Verbaeten. A Metaobject Protocol for Correlate. In *Proceedings of ECOOP '98 Workshop on Reflective Object-Oriented Programming Systems*, July 1998.

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