*ReBA: Re*factoring-aware *B*inary *A*daptation of Evolving Libraries

Danny Dig, Stas Negara, Ralph Johnson
University of Illinois at Urbana-Champaign, USA
[dig, snegara2,johnson]@cs.uiuc.edu

Vibhu Mohindra
ACL Wireless Limited, India
vibhu@wispym.com

ABSTRACT

Although in theory the APIs of software libraries and frameworks should be stable, they change in practice. This forces clients of the library API to change as well, making software maintenance expensive. Changing a client might not even be an option if its source code is missing or certain policies forbid its change. By giving a library both the old and the new API, clients can be shielded from API changes and can run with the new version of the library.

This paper presents our solution and a tool, ReBA, that automatically generates compatibility layers between new library APIs and old clients. In the first stage, ReBA generates another version of the library, called adapted-library, that supports both the old and the new APIs. In the second stage, ReBA shrinks the adapted-library into a minimal, client-specific compatibility layer containing only classes truly required by the client. Evaluations on controlled experiments and case studies using Eclipse core libraries shows that our approach effectively adapts clients to new library versions, and is efficient.

Categories and Subject Descriptors D2.7 [Software Engineering:] Distribution, Maintenance, and Enhancement; D.2.13 [Software Engineering:] Reusable Software

General Terms Design, Management.

Keywords Refactoring, libraries, component reuse, API compatibility.

1. INTRODUCTION

Software libraries expose a set of *Application Programming Interfaces (APIs)* that allow client applications to interact with the library. A library API consists of a set of public methods and classes that are meant to be used by clients of that API. Sometimes new versions of library APIs are backwards compatible, but often they are not. Our previous study [7] of five mature, widely used Java libraries and frameworks, reveals a large number of API changes that are not backwards compatible. For example, Struts [18] had 136 API changes over a period of 14 months and Eclipse [8] had 51 API changes over a period of one year. In each of the five systems, more than 80% of API-changes that are not backwards compatible are caused by refactorings.

Refactorings [10] are program transformations that improve the structure of the code without changing the external behavior. Examples of refactorings include renaming classes and members, moving methods between classes, and encapsulating fields by replacing direct references with accessor methods.

Although refactorings make libraries easier to understand and use, refactorings often change library APIs. Library clients either stick with the old, increasingly obsolete versions of libraries, or must be upgraded to use the latest version. Upgrading the clients is traditionally done manually, which is error-prone, tedious, and feels disruptive in the middle of development. This makes maintenance expensive.

Previous solutions to automate upgrading of clients advocate that (a) source code of clients [3, 4, 12, 16] or (b) bytecode [13, 20] of clients be changed in response to library API changes. However, this is not always possible. For example, Eclipse IDE [8] ships with many 3rd party clients that extend the functionality of the IDE. These clients are distributed in bytecode format only, without source code, and have software licenses that prohibit changing the bytecodes of clients for legal reasons.

Rather than waiting for 3rd party client developers to update their clients whenever the library APIs change, we propose that clients are automatically *shielded* from the API changes in the library. Our solution restores the *binary compatibility* of the library: older binary clients continue to link and run against the new libraries without changing or recompiling the clients. Our solution automatically generates a *refactoring-aware compatibility layer* without requiring any annotations from library developers. This compatibility layer gives library developers the freedom to improve the library APIs without breaking older clients.

This paper presents *ReBA*, our refactoring-aware binary adaptation tool. ReBA has several distinguishing features that make it practical for modern large software systems comprised of core libraries and many 3rd party clients. We designed ReBA to meet the criteria that we believe are required for modern systems:

- **Binary Clients:** the solution should work with the binary version of the clients. This is desirable because large systems (e.g., IDEs) often do not own the source code of 3rd party clients that extend the functionality of the system.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ICSE'08, May 10–18, 2008, Leipzig, Germany.
Copyright 2008 ACM 978-1-60558-079-1/08/05 ...$5.00.

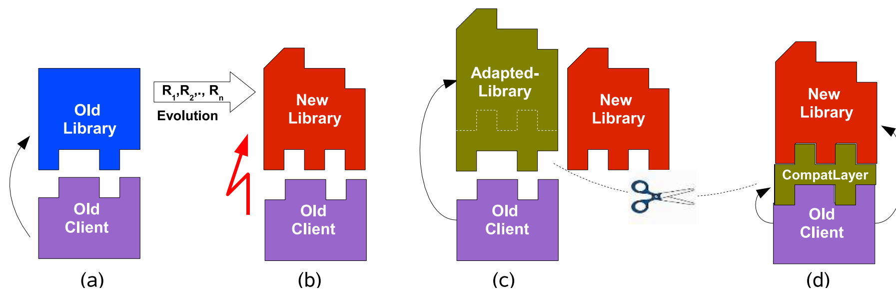

Figure 1: Overview of our approach (arrows show dependences between modules). (a) Library and client code are compatible. (b) API changes in library break compatibility. (c) Library developers use ReBA to generate an adapted-library that supports both the old and the new APIs. (d) Given the adapted-library, client developers use ReBA to carve out a client-specific compatibility layer. The old client code loads the restored APIs from the compatibility layer and the APIs not changed from the new library.

- **Dual Support:** consider a large system like the Eclipse IDE that ships with core libraries and 3rd party clients. Some clients are updated regularly to use the new versions of libraries, but others are not. A solution must allow both old and updated clients to simultaneously use the new version of the libraries and be included in the shipped product.
- **Preserve Edits**: the proposed solution should preserve the edits (e.g., performance improvements or bug fixes) of the library. This is important because old clients can benefit from the enhancements in the library.
- **Preserve Type Information**: a known problem with compatibility layers using the Adapter pattern [11](also known as Wrapper) is that object identity breaks, since adapter and the adaptee are two different classes. We want to ensure that old clients that use type, identity, or equality comparisons (e.g., `instanceof`, `==`, or `equals`) would still work.
- **Use Only Standard Compilers**: this is desirable because many developers do not trust modified compilers or class loaders.

Figure 1 shows an overview of our solution. ReBA works in two main stages. First, library developers use ReBA to create a version of the library, called *adapted-library* (details in Section 4). The adapted-library contains the same code as the new version of the library, but it supports both the old and the new APIs. Second, given this adapted-library, client developers use ReBA to carve a custom-made, client-specific *compatibility layer* (details in Section 5) using a static analysis technique based on *points-to* analysis [21].

This technique greatly reduces the size of the compatibility layer, while our conservative static analysis ensures that an old client loads at runtime only classes that are backwards compatible. The compatibility layer allows both old and new clients to run simultaneously with the refactored library. In addition, since the compatibility layer uses the same class names that the old client expects, the layer preserves the type information and identity of the adapted classes.

This paper makes three major contributions:

1. An innovative solution. Our solution automatically generates a compatibility layer for a library that allows both the old and new interface to be used in the same system. It does not require programmers to annotate their library, but builds the compatibility layer using information recorded by a refactoring tool. It uses points-to analysis in a new way to minimize the size of the layer.

2. Implementation. ReBA, our concrete implementation, works for Java programs. ReBA is implemented as an Eclipse plugin, therefore it is conveniently integrated into the widely used Eclipse IDE. To minimize the size of the compatibility layer, we implemented the points-to analysis using WALA [19], an analysis library from IBM Research.

3. Evaluation. We used ReBA to generate compatibility layers in different scenarios (see Section 7). First, we evaluated ReBA on the data obtained from a controlled experiment with 10 different developers. Second, we evaluated ReBA on three Eclipse libraries which were refactored by Eclipse developers. For both scenarios we used ReBA to generate and apply compatibility layers to a comprehensive test suite created for a pre-refactoring version of the library. After applying our compatibility layers, the tests ran successfully. Profiling shows that the memory and running overhead imposed by our solution is small.

ReBA and our experimental results are available online: `http://netfiles.uiuc.edu/dig/ReBA`

2. MOTIVATING EXAMPLES

We show three examples of API changes that cause problems for clients. These examples are taken from real world case-studies, namely libraries that make up the official distribution of the Eclipse IDE [8].

With respect to how ReBA generates the compatibility layers, API changes can be separated into three categories which are mutually exclusive and cover all cases: deletions of APIs, method-level API refactorings, and type-level API refactoring. Each of the examples below belongs to one of the three categories. The first three examples illustrate the need for the adapted-library, the fourth example illustrates the need to reduce the size of the adapted-library.

Deletion of APIs. In library `jface.text` version 20060926 (format yearmonthday), class `DiffApplier` is deleted. There is at least one client, `workbench.texteditor.tests` that uses this class. As a result of the deletion of `DiffApplier`, any version of this client prior to 20060926 no longer works with the

```
// version 20070329 before refactoring
class RefactoringExecutionStarter {
  ...
  void startCleanupRefactoring(ICompilationUnit[] cus,
              boolean showWizard, Shell shell) {

    ICleanUp[] cleanUps= CleanUpRefactoring.createCleanUps();
    for (int i= 0; i < cleanUps.length; i++)
       refactoring.addCleanUp(cleanUps[i]);
    ...
  }
}
```

```
// version 20070405 after refactoring
class RefactoringExecutionStarter {
  ...
  void startCleanupRefactoring(ICompilationUnit[] cus,
          boolean showWizard, Shell shell,
          ICleanUp[] cleanUps, String actionName) {

    for (int i= 0; i < cleanUps.length; i++)
       refactoring.addCleanUp(cleanUps[i]);
    ...
  }
}
```

Figure 2: Example of ChangeMethodSignature refactoring in plugin `jdt.ui`. Two new arguments were added: `cleanUps` was previously a local variable and is now extracted as an argument, `actionName` is a brand new argument.

newer version of the library. To make this change backwards compatible, ReBA creates a version of the library, which we call adapted-library, where `DiffApplier` is added back. Other than adding a dependency between the test client and the adapted-library, ReBA requires no changes to the client.

Method-level refactorings. In library `jdt.ui` version 20070405, the signature of method `RefactoringExecutionStarter.startCleanupRefactoring` changed from three arguments to five arguments (see Fig.2). The library developers provided default values for the two new arguments when they performed the refactoring. Calling the 5-argument method with the default values behaves like the previous 3-argument method.

There is at least one client broken because of this API change, the testing client `jdt.ui.tests`. To shield this client from the change, ReBA generates an adapted-library containing a version of `jdt.ui` where the original method with 3 arguments is restored. The method merely delegates to the new version with 5 arguments, and passes the default values for the extra arguments (see Fig.3). Because the adapted-library delegates to the latest implementation of the method, old clients can benefit from the improvements in the library. When supplying the compatibility layer to the old client, all tests run successfully.

```
//Adapted Library Class
class RefactoringExecutionStarter {
  ...
  // ReBA adds this method with the old signature
  void startCleanupRefactoring(ICompilationUnit[] cus,
              boolean showWizard, Shell shell) {
    return startCleanupRefactoring(cus, showWizard, shell,
          CleanUpRefactoring.createCleanUps(),"Clean Up");
  }

  //new method with 5 arguments
  void startCleanupRefactoring(ICompilationUnit[] cus,
          boolean showWizard, Shell shell,
          ICleanUp[] cleanUps, String actionName) {
    //same implementation as in the
    //new version of the library
    ...
  }
}
```

Figure 3: Compatibility layer generated for ChangeMethodSignature refactoring in library `jdt.ui`

Type-level refactorings. In library `workbench.texteditor` version 20060905, class `Levenstein` was renamed to `Levenshtein` (notice an extra "h" character). There is at least one client, `workbench.texteditor.tests` that is broken because of this change. ReBA generates an adapted-library where the class name `Levenshtein` is reverted back to `Levenstein` (all other code edits in `Levenshtein` are preserved). Because the adapted-library consistently uses the same class names that the old client expects, client code that uses type checking (e.g., `instanceof`) or makes use of object equality (`equals`) or object identity (`==`), still runs as before, without requiring any changes to the client.

The need for carving. After creating the adapted-library, in order to enable both upgraded and old clients to run simultaneously, one needs to keep both the adapted-library and the new version of the library. This can result in doubling the memory consumption.

To alleviate this problem, once it generated the adapted-library, ReBA carves out a client-specific compatibility layer which is much smaller than the adapted-library. Recall our previous example where ReBA generated the adapted-library by reverting the renamed class `Levenshtein` back to `Levenstein`. Figure 4 shows some classes in the adapted-library that use `Levenstein`. A client can use `Factory` to create instances of `Levenstein` class. `Factory` has a direct source reference to `Levenstein`, therefore `Factory` needs to be in the compatibility layer. If it was not in the compatibility layer, the client would load the version of `Factory` from the new library where `Factory` creates instances of `Levenshtein`, thus breaking the client.

```
class Factory {
  Levenstein create(){
    return new Levenstein();
  }
 }

class Indirect {
  Object m() {
    Factory factory = new Factory();
    return factory.create();
  }
 }
```

Figure 4: Library classes that use `Levenstein`

The other library class, `Indirect`, does not appear to have a reference to `Levenstein`. Does `Indirect` needs to be put in the compatibility layer? Yes. Although the source code of `Indirect` does not have a reference to `Levenstein`, its bytecode does. This is because the method call to `create` in the bytecode of `m` is replaced by the method descriptor for `create`. Bytecode method descriptors contain not just the method name and arguments, but also the return type [14]. Therefore, in order to restore the backward compatibility, the version of `Indirect` that is compiled with `Levenstein` needs to be put in the compatibility layer, otherwise a runtime error "method not found" is thrown. ReBA uses a points-to analysis (details in Section 5) to find all classes in the adapted-library that can refer to `Levenstein`, thus

it finds both `Factory` and `Indirect` and adds them to the compatibility layer. The carved compatibility layer contains only `Levenstein`, `Factory`, and `Indirect`.

3. OVERVIEW

3.1 Background information

During library evolution, most changes add new API methods and classes that do not affect old clients. Therefore, we distinguish between API changes that are backwards compatible (e.g., additions of new APIs), and those that are not backwards-compatible (e.g., deletion of API methods, renaming API classes). Since only the latter category affects old clients, from now on, by API changes we refer to those changes that are not backwards compatible.

Besides refactorings, libraries evolve through edits. We distinguish between *API edits* (e.g., edits that change the APIs) and *code edits* including all remaining edits (e.g., performance improvements, bug fixes). Code edits in general have less defined semantics, making it harder for tools to automatically reason about them. ReBA preserves in the adapted-library all code edits from the latest version of the library, and adapts the API deletions and API refactorings. Refactorings preserve the semantics, but edits do not. Although ReBA intends to ensure that old binary clients run "correctly" with the new versions of the library, because of the edits, we cannot give such strong semantic guarantees.

In a statically-typed language like Java, an API consists of two things: (i) the type/class of an object, represented by a fully qualified (i.e., package.class) name along with a set of inherited classes and interfaces and (ii) the message protocol, or the set of methods and public fields that it supports. Thus we further classify API refactorings based on what aspect of the API is changed (type or method protocol).

In the current implementation, ReBA supports supports the following API changes. (i) *Type-changing refactorings* include rename class, move class to different package, rename package, and move package (these refactorings change the fully qualified name of a type). (ii) *Method-level refactorings* include rename method, move method, change method signature, encapsulate field with accessor methods, rename field. (iii) *API deletions* include delete class, delete method, delete package. These API changes occur most frequently in practice [7].

3.2 High-level Overview

This section gives an overview of each of the two stages of our solution.

Creating the Adapted Library.

One approach to creating a backwards-compatible, adapted-library, is to start from the new version of the library and undo all API changes, in the reverse order in which they happened. Although simple, this approach has two limitations. First, one needs to selectively undo refactorings while ignoring API additions. There can be dependences between refactorings and API additions, so undoing one without the other is hard. Second, from a practical point of view, tools that record API changes might not store all the information required to undo an API change. For example, Eclipse refactoring logs do not store enough information to undo deletions.

INPUT: $Library_{New}$, $Library_{Old}$, $\Delta_{Library} = (R_1, R_2, ..., R_n)$
OUTPUT: *AdaptedLibrary*
Creating AdaptedLibrary **begin**
1 $AdaptedLibrary = Library_{New}$
2 **forEach**(Operation op: $\Delta_{Library}$)
3 Operation $\widetilde{op}$ = `preserveOldAPI(op)`
4 $Library_{Old}$ = `replayOperation`($\widetilde{op}$, $Library_{Old}$)
5 **endForEach**
6 $AdaptedLibrary$ += `copyRestoredElements`($Library_{Old}$)
7 $AdaptedLibrary$ = `reverseTypeChanges`($AdaptedLibrary$)
end

Figure 5: Overview of creating the Adapted-Library

Therefore, ReBA does not undo API changes on the newer version of the library, but it replays the (modified) API changes on the old version of the library. When replaying the API changes, ReBA alters them such that they preserve the backwards compatibility (e.g., a delete operation is not replayed, and a rename method leaves a delegate).

Figure 5 shows an overview of creating the adapted-library. The algorithm takes as input the source code of the new and old versions of the library, and the trace of API changes that lead to the new version. These API changes can be retrieved from an IDE like Eclipse (version 3.3) that automatically logs all refactorings and deletions. Alternatively, API changes can be inferred using our previous tool RefactoringCrawler [6].

ReBA starts from the old library version and processes each library API change in the order in which they happened. For each API change *op*, ReBA creates a source transformation, $\widetilde{op}$, that creates the refactored program element and also preserves the old element. Recalling our example in Fig.2, ReBA creates another change signature refactoring which keeps both the old and the new method (the old method merely delegates to the new method). Then ReBA replays the $\widetilde{op}$ transformation on the old version of the library.

Once it processes all API changes, ReBA copies the changed program elements from $Library_{Old}$ to *AdaptedLibrary*. Although the *AdaptedLibrary* starts as a replica of the new library version, with each copying of program elements from the modified $Library_{Old}$, it supports more of the old APIs.

The compatibility of classes whose type was changed cannot be restored by copying, but only by restoring their types. ReBA restores the types by reversing refactorings that changed the types. For example, ReBA restores the API compatibility of `Levenshtein` in our motivating example by renaming it back to `Levenstein`.

At the end, the adapted-library contains all the APIs that the old client requires. The algorithm processes refactorings from different categories differently. Section 4 presents one example from each category.

Carving the Compatibility layer.

To optimize for space consumption, ReBA constructs a client-specific compatibility layer. The customized compatibility layer consists of classes whose API compatibility was restored in the adapted-library, as well as those classes that are affected by this change. Some classes are *directly affected*, for example, a class that has a reference to a renamed class. Other classes are *indirectly affected*, for example classes that do not have a source reference, but only a

INPUT: *AdaptedLibrary*, *client*, $\Delta_{Library} = (R_1, R_2, ..., R_n)$
OUTPUT: *CompatLayer*
Carving the Compatibility Layer **begin**

```
1  CompatLayer = ∅
2  Graph pointsToGraph =
3    buildPointsToGraph(AdaptedLibrary, client)
4  forEach(Operation op: Δ_Library)
5    Class[] classes= getChangedClassesFrom(op)
6    Set reachingClasses =
       getReachingNodes(pointsToGraph, classes)
7    CompatLayer = append(CompatLayer, {classes, reaching-
Classes})
8  endForEach
end
```

Figure 6: Overview of carving the compatibility layer

bytecode reference to the restored class. Recalling our example from Fig. 4, ReBA needs to put in the compatibility layer both `Factory` (directly affected) and `Indirect` (indirectly affected), in order to restore the binary compatibility.

To find both directly and indirectly affected classes, and to keep only those classes that a specific client can reach to, ReBA uses a *points-to* analysis. Points-to analysis establishes which pointers, or heap references, can point to which variables or storage locations. Figure 6 gives an overview of how we use points-to analysis to determine the direct and indirect affected classes, the details of the analysis are found in Section 5.

Starting from the client, ReBA first creates a directed graph (see Fig.6) whose nodes are library classes reached from the client and whose edges are points-to relations. Then ReBA iteratively processes each API change and determines the classes that are changed due to restoring their backwards compatibility. ReBA traverses the *pointsToGraph* and gets all other library classes that can reach to the restored classes. For each refactoring, the compatibility layer grows by adding the classes that are changed as well as their reaching classes.

Putting it all together.

Next we show how an old client is using the compatibility layer along with the new version of the library. First we give a gentle introduction to how classes are loaded in Java. ReBA does not require any special class loading techniques, thus it enables anybody who uses the standard class loading to benefit from our solution.

In Java, classes are loaded lazily, only when they are needed. Classes are loaded by a *ClassLoader*. When running a client, one has to specify the *ClassPath*, that is the places where the bytecodes of all classes are located. The ClassPath is a sequence of classpath entries (e.g., Jar files, folders with bytecode files), each entry specifying a physical location. When loading a class, the ClassLoader searches in each entry of the classpath until it can locate a class having the same fully qualified name as the searched class. In case there are more than one class that match the fully qualified name, the ClassLoader loads the first class that it finds using the order specified in the class path entries.

After creating the compatibility layer, ReBA places it as the first entry in the ClassPath of the client. Thus, at runtime, when loading a class, the ClassLoader searches first among the classes located in the compatibility layer.

Suppose that the old client asks for a class whose API compatibility was restored by ReBA. The ClassLoader finds this class in the compatibility layer and it loads it from there. Even though a class with the same name (but which is not backwards compatible) might exist in the new library, because the ClassPath entry for the compatibility layer comes before the entry for the new library, the ClassLoader picks the class from the compatibility layer.

Now suppose that the client searches for a class that is backwards compatible even in the new version of the library and is not present in the compatibility layer. The ClassLoader searches for this class in the compatibility layer; it is not found here, and the ClassLoader continues searching in the new library where it finds the required class.

Now suppose that we have both an old client and a new client running together. The compatibility layer allows the old client to load classes that are backwards compatible as described above. As for the new client, since it does not have the compatibility layer among its ClassPath entries, it can only load classes from the new library.

4. CREATING THE ADAPTED LIBRARY

This section describes how the adapted-libraries are generated by presenting one example from each of the three categories of API changes. We use the same examples as in our motivating section (Sec. 2).

4.1 Deletion of APIs

For operations belonging to this category, function `preserveOldAPI` (line 3 in Fig.5) returns a `NOP` operation. For example, when processing the operation that deletes class `DiffApplier`, ReBA does not replay the deletion, thus it keeps `DiffApplier` in the old version of the library. Later, function `copyRestoredElements` copies `DiffApplier` from the old version of the library to the adapted-library.

Function `reverseTypeChanges` does not process operations belonging to this category, since their API compatibility was restored previously by function `copyRestoredElements`.

4.2 Method-level Refactorings

For operations belonging to this category, function `preserveOldAPI` returns a new operation. The new operation is the same kind of refactoring as the original refactoring. However, in addition to replacing the old method with the refactored method, it creates another method with the same signature as the old method, but with an implementation that delegates to the refactored method.

For our motivating example of ChangeMethodSignature (Fig. 2) that adds two arguments to `startCleanupRefactoring`, the new operation produced by `preserveOldAPI` is another ChangeMethodSignature refactoring. Because ReBA is implemented on top of Eclipse, it uses the Eclipse representation to create new refactorings. In Eclipse (version > 3.2), when playing a refactoring, the refactoring engine logs each refactoring and it produces a refactoring descriptor. This refactoring descriptor is a textual representation of the refactoring and contains information such as: the type of refactoring, the program element on which the refactoring operates (identified via a fully qualified name), and arguments supplied by the user through the UI (e.g., the default values for added method arguments).

Function `preserveOldAPI` takes a refactoring descriptor and it creates another refactoring descriptor. In our example, the new refactoring descriptor specifies the same properties as the old descriptor: the type of refactoring, the input element `startCleanupRefactoring`, the default values for the two extra arguments (the String literal "Clean Up" and "CleanUpRefactoring.createCleanUps()"), etc. In addition, ReBA adds another tag specifying that the original method should be kept and delegate to the refactored method. Out of the new descriptor, ReBA creates a refactoring object and replays this refactoring on the old version of the library.

Figure 7 shows that for the motivating example (Fig. 2), function `copyRestoredElements` copies only the restored method with the old signature from the refactored version of the old library to the adapted-library. ReBA does not copy the method with the new signature from the old library. Recall that by construction, the adapted-library contains all code edits from the new version of the library. Thus, the adapted-library uses the same implementation of this method as the new version of the library. After copying the old method in the adapted-library, because the copied method delegates to the new implementation of the method (located in the same class), ReBA allows clients to benefit from the improvements (e.g., bug fixes, performance improvements) in the new version of the library.

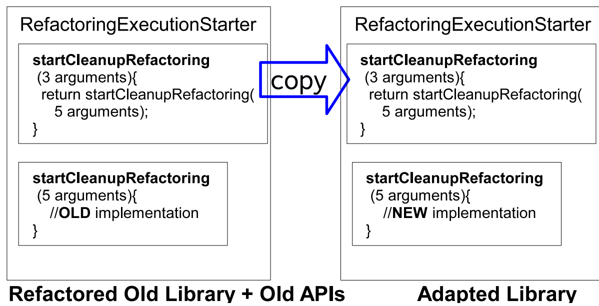

Figure 7: Using the ChangeMethodSignature motivating example (Fig. 2), function `copyRestoredElements` copies only the restored API elements.

Function `reverseTypeChanges` does not process operations from this category since their API compatibility was restored previously.

4.3 Refactorings that Change Types

Operations in this category include all those that change the fully qualified (e.g., package.class) names of classes. Fully qualified names play a central role in how classes are loaded at runtime; any changes to these names would cause older clients to fail to load the class. Since a class cannot have two names, restoring the compatibility of a class whose name has changed, means restoring the old name of the class. This restoration happens in two steps.

For refactorings belonging to this category, function `preserveOldAPI` is the identity function, e.g., returns back the original refactoring. Although this operation does not restore the backwards compatibility, ReBA still replays it on the old library. This is needed because later refactorings could depend on the names changed by refactorings in this category. More specifically, later refactoring cannot be executed unless the previous refactorings were executed. The reason for the dependences between refactorings is the fact that refactoring engines identify program elements by their fully qualified names (e.g., package.class.methodName for a method). For example, given the sequence:

$\Delta_{Library} = \{op_1 = \text{RenameClass}(A \rightarrow B);$
$op_2 = \text{MoveMethod}(B.m \rightarrow C)\}$

Refactoring op_2 could not discover method `B.m` unless it looks into class `B`, thus op_1 must take place before op_2 could proceed.

Function `copyRestoredElements` does not copy elements affected by this kind of refactorings, because these elements are not yet backwards compatible. Function `reverseTypeChanges` is the one that restores the compatibility of these program elements by creating a *reverse* refactoring and applying it on the adapted-library. In our motivating example, function `reverseTypeChanges` reverts `Levenshtein` back to `Levenstein` by applying a reverse rename class refactoring.

To restore the old types, ReBA creates reverse refactorings for all refactorings that changed the object types. Then ReBA applies the reverse refactorings backwards, from the last operation in $\Delta_{Library}$ toward the first operation.

In the `Levenshtein` example, ReBA creates the reverse refactoring by constructing a new refactoring descriptor where it swaps the old and new names of the class. ReBA passes this refactoring descriptor to the Eclipse refactoring engine to apply the refactoring.

5. CARVING THE COMPATIBILITY LAYER

From the adapted-library, ReBA carves a smaller, client-specific compatibility layer. This layer contains only the classes whose API compatibility was previously restored and those classes from which a client could reach/load the restored classes. All the remaining classes need not be present in the compatibility layer, and can be loaded from the new version of the library.

5.1 Points-to analysis

The heart of this stage is the use of *points-to* analysis to find the set of classes, *Reaching*, that can "reach" to classes whose API compatibility is restored. Points-to analysis establishes which pointers, or heap references, can point to which variables or storage locations. Figure 8 illustrates the creation of the points-to graph for a simple program.

In Fig. 8 allocated objects are marked with squares ($<s_i$:T$>$ where i shows the line number where the object is created, T represents the type). Pointers to objects (i.e., references in the Java terminology) are denoted by named circles, having the same name as the pointer. Directed edges connect pointers to the objects they point to, or to other pointers (e.g., field f1 points to pointer u1). Fields are also connected to their objects by directed edges.

We use a fast points-to analysis that is flow-insensitive: it does not take into account the order of statements (e.g., line 5 does not "kill" the previous points-to relation $u1 \rightarrow s_4 : U$). Being a static analysis, the points-to analysis is conservative, meaning that at runtime some pointers "might" point to the computed allocated objects. Having false positives means that our analysis can add unnecessary classes to the compatibility layer. However, it is much more important that the analysis does not miss cases when a class should be added to the compatibility layer.

To build the points-to graph, we use WALA [19], a static analysis library from IBM Research. The example we illus-

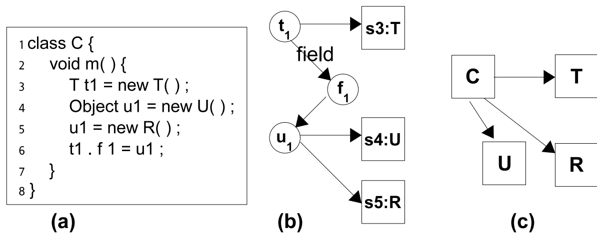

Figure 8: An example points-to graph at the end of method `m`. (a) shows the source code, (b) shows the points-to relations, (c) shows the points-to relations between classes.

trated shows the points-to graph for an *intraprocedural* analysis. When the function has calls to other functions, WALA does an *interprocedural*, context-sensitive analysis (i.e., it takes the calling context into account when analyzing the target of a function call.) To compute an interprocedural analysis, WALA first creates the call graph using the *entry-points* (e.g., main methods) from the client. WALA does not need the source-code of the client or library, it only needs the bytecode of client and library. Using the information from the call graph, the interprocedural analysis expands function calls. In other words, it builds the points-to graph by traversing "through" function calls.

From the points-to graph created by WALA, ReBA creates a customized points-to graph where it represents points-to relations at a higher granularity. Specifically, the custom points-to graph represents points-to relations only among classes. First, it short-circuits the original points-to graph such that each pointer points to a leaf object (e.g., it changes $t_1 \rightarrow f_1 \rightarrow u_1 \rightarrow s_5 : R$ to $t_1 \rightarrow s_5 : R$). Next, for each points-to relation in the short-circuited graph it creates a points-to relation between the class where the pointer is defined and the class of the pointed object. For the graph in Fig. 8.b, ReBA creates the points-to graph in Fig. 8.c whose nodes are classes and the edges are relations between classes.

Next, for each API change, ReBA determines the set of classes changed in order to restore the API compatibility. Then ReBA searches in its points-to graph for all classes that can "reach" to these changed classes through the points-to relations. Finally, ReBA puts the *Reaching* classes and the changed classes in the custom-made compatibility layer.

6. DISCUSSION

During the first stage when ReBA generates the adapted-library by replaying the modified API changes, ReBA only needs access to the source code of the library, not the client. Since the library developers use ReBA to generate the adapted-library, ReBA has no problem accessing the library source code. With each new library version, library developers can ship a signed, trusted adapted-library that is backwards compatible. Client developers can use ReBA to carve out a custom compatibility layer. Additionally, library developers could also use ReBA to generate custom compatibility layers for clients for which they do not have access to the source code. For example, large frameworks like Eclipse ship with 3rd party clients for which the framework providers do not have access to source code. Eclipse developers could use ReBA to accommodate 3rd party clients that make up the official distribution of Eclipse.

In Java, the bytecode of the client contains as much information as the source code. In cases when the source code of clients is missing, one could ask why not change the bytecodes of clients directly? There is no technical reason why one could not refactor the bytecodes of the client in response to library refactorings. However, there might be several other reasons why changing the bytecodes is not an option. First, for legal reasons, several proprietary companies and software licenses forbid any changes in the bytecodes of shipped clients. Even some open-source licenses only allow changes in the source code. Second, even the smallest change in the source code of clients might require recompilation, thus rendering the previous bytecode patches invalid. Having a bytecode patch that needs to be recreated and reapplied at each recompilation of clients can easily lead to expensive maintenance. Third, the bytecodes could be intentionally obfuscated for privacy reasons.

Rather than using the points-to analysis, another approach is to search in the bytecode for those classes that were changed indirectly when restoring the compatibility of other classes. For the example presented in Fig. 4, class `Indirect` would be such a candidate. Although this approach is sufficient for regular Java libraries and clients, it is not adequate for other toolkits that extend the class lookup mechanism. For example, in Eclipse framework, each library or client (e.g., plugin) that makes up the IDE needs to specify its own Classpath. Rather than using one single ClassLoader, Eclipse creates individual ClassLoaders for each library or client. When restoring the compatibility of an API class, ReBA needs to put in the compatibility layer all classes that can transitively reach to the restored class, regardless of whether or not these classes have been changed. Failure to do so will result in runtime linkage errors because the "reaching" class and the restored class are not loaded by the same ClassLoader.

Our approach to adapt libraries is not limited to two consecutive versions of the library, but works for any two versions of a library. Given three different library versions, V_1, V_2, and V_3, the adapted-libraries/compatibility layers generated are not stacked on top of each other. To enable a V_1-compatible client to work with the V_3 library, ReBA generates one single V_1-V_3 compatibility layer. Since this avoids delegation between several layers, our approach results in a smaller performance overhead for the client. The downside of this approach is that the analysis for creating the V_1-V_3 layer does not reuse the analysis for creating the V_1-V_2 layer, but starts all over. However, this analysis is fast.

Although ReBA does not currently handle all refactorings, we examined all refactorings supported by the Eclipse IDE and they all could be adapted in a similar way. Some Eclipse refactorings are composed of smaller refactorings. For example, ExtractClass refactoring which introduces another class in the type hierarchy and moves methods in the introduced class can be implemented by combining refactorings from our second category (method-level) and refactorings from the third category (type-level). Changes to the type inheritance hierarchy need to be reverted the same way how we revert type names.

Although our solution is not language-specific, its implementation is. In order to adapt our solution to other programming languages, these are the areas which will need to be changed. First, class lookup mechanism is language-

specific. In Java, we place the compatibility layer as the first entry in the Classpath. For example, Smalltalk uses a dictionary to lookup classes, thus the adapted classes need to be placed in the class dictionary. Second, the points-to analysis uses the WALA library for Java, and an equivalent library will be needed for the new programming language. A language-specific points-to analysis can be used to ensure that all class libraries that can "escape" the adapted-classes are placed in the compatibility layer in order to preserve the type and information identity of objects. Third, refactoring engines are language-specific, so Eclipse's refactoring engine needs to be replaced by another engine.

Strengths. Our solution meets all four practical criteria outlined in Section 1: it does not need access to the source code of clients, nor does it require any changes to the clients. Second, it supports both old and new clients to run against a new version of the library. Third, it preserves the edits in the library. Fourth, it preserves the type and identity of the adapted classes.

The current state-of-the-practice in library design favors deprecating the old APIs, and removing them after several iterations. However, deprecated APIs are rarely removed. For example, Java 1.4.2 runtime library has 365 deprecated methods. Our solution enables libraries to evolve without resulting in the proliferation of APIs. ReBA enables library developers to maintain crisp APIs at the source code level, while at the binary level, the old APIs are added back automatically for compatibility reasons.

Limitations. The key aspect of our solution is that instead of putting a wrapper around the library, we give it two interfaces, an old API and a new API. This only works when the two interfaces are compatible. It is fine for the interfaces to intersect, but they can not contradict each other. For example, if library developers delete method `A.m1`, and then rename `A.m2` → `A.m1`, it is not possible to support both the old and the new interface of class `A`. Although this might seem like a severe limitation of our approach, in practice we never met such conflicting changes.

Our adaptation approach adds some memory and CPU overhead. However, the evaluations show that this overhead is small enough to be considered acceptable.

7. EVALUATION

In the evaluation, we want to answer these questions:

- **Q**1(effectiveness): Does our generated compatibility layer allow older clients to run with newer versions of libraries?
- **Q**2(efficiency): Is the generated compatibility layer only as large as a concrete client would need? What is the performance overhead imposed by our solution?

To answer these questions we conducted different studies. The first study is a controlled experiment of 10 developers evolving a LAN library [5] independently. The second study is a suite of case-studies using ReBA to restore the API compatibility of Eclipse core libraries. We used comprehensive JUnit test suites developed by the library provider. If the old tests succeed with the new version of the library, it implies a strong likelihood that a regular client application would succeed too.

Table 1: Demographics of the participants.

	Mean	Std.Dev.	Min.	Max
Years Programming	8.35	1.97	5	12
Years Java Programming	4.7	1.72	2.5	7.5
Years Using Eclipse	2.1	1.24	0.5	4

7.1 Controlled experiment

We asked 10 developers to independently evolve a version of a LAN simulation library [5]. The library code was accompanied by an automated JUnit test suite. Each developer had to implement one of two features. How they implemented the feature was their choice. In addition, developers had the freedom to refactor any APIs in the library, though we did not influence their choices. We requested that the developers not work for more than one hour.

Demographics. Although the developers were graduate students, they were mature programmers with several years of programming experience. Most of them had worked in industry previously (two of them had extensive consulting experience, two others were active Eclipse committers). Each developer implemented successfully the required feature. Table 1 shows the demographics of our population.

Experimental treatment. We took their solutions, and used ReBA to generate a compatibility layer for each of their solution. Then we ran the original test suite with the compatibility layer and the new library versions.

Results. Table 2 shows the number of compile errors and test failures when putting together the old JUnit test suite with the new library. Each row displays the data for a particular developer solution: the number of errors before and after placing the ReBA-generated compatibility layer. After using the ReBA-generated compatibility layers, there were no problems for any the 10 solutions. The last row shows a solution that did not contain any refactoring nor other API-breaking changes - as a consequence, ReBA generated an empty layer and the original test ran successfully.

To answer the efficiency question, with regards to the minimality of our compatibility layers, we manually examined the generated compatibility layers and they contain exactly the classes that the old tests need. Removing any class from the compatibility layer would break the API compatibility.

We used the TPTP [eclipse.org/tptp] profiler to measure the memory and the runtime overhead of using ReBA compatibility layers. To measure the overhead, for each solution we manually upgraded the client test suite and used it as the base case. The overhead of using the generated layer instead of manually upgrading the client is small. In cases that involve the encapsulation of fields with accessor methods, the runtime overhead is negative because the adapted version directly references the fields, whereas the refactored version uses the accessor methods.

7.2 Case studies

We took three core Eclipse libraries. As API clients we used older versions of JUnit test suites created by the Eclipse developers. Each row in Table 3 shows the versions of the library and client. The client `workbench.texteditor.tests` is an older version of an Eclipse test suite comprising 106 tests. The third row shows an example of an Eclipse library, `core.refactoring` breaking an Eclipse UI client, `ui.refactoring`.

Table 4 displays the effectiveness and efficiency of the compatibility layer generated by ReBA. After using the compat-

Table 2: The effectiveness and efficiency of ReBA compatibility layers for the controlled experiment before/after applying the compatibility layer

Developer Solution	API-Breaking Changes	CompileErrors Before	CompileErrors After	TestFailures Before	TestFailures After	MemoryUsed[B] Before	MemoryUsed[B] After	Memory Overhead	RunningTime[ms] Before	RunningTime[ms] After	Running Overhead
#1	1 ChangeMethodSignature, 6 EncapsulateField	34	0	3	0	1418312	1532640	8.06%	14.37	14.02	-2.39%
#2	6 EncapsulateField	29	0	3	0	1417896	1532216	8.06%	14.98	14.35	-4.21%
#3	2 ChangeMethodSignature	4	0	2	0	1414864	1533584	8.39%	9.65	10.03	3.93%
#4	1 RenameMethod, 2 ChangeMethodSignature	2	0	1	0	1418648	1535296	8.22%	8.7	8.9	2.65%
#5	2 RenameType, 1 RenameField	19	0	3	0	1420400	1526888	7.49%	7.86	7.99	1.56%
#6	3 RenameMethod	5	0	2	0	1422376	1538616	8.14%	10.0	10.1	1.29%
#7	6 EncapsulateField	29	0	3	0	1419296	1532056	7.94%	14.1	13.7	-2.33%
#8	1 DeleteMethod, 1 EncapsulateField	6	0	3	0	1421480	1532968	7.84%	11.4	11.0	-2.98%
#9	3 ChangeMethodSignature	2	0	2	0	1426664	1534736	7.57%	7.94	8.26	4.01%
#10	0	0	0	0	0	1358884	1358884	0	7.62	7.62	0

Table 3: Eclipse plugins used as case studies

Library	LibrarySize[LOC]	client
`workbench.texteditor` v20060905	16842	`workbench.texteditor.tests` v20060829 (106 tests)
`jface.text` v20060926	31551	`workbench.texteditor.tests` v20060829 (106 tests)
`ltk.core.refactoring` v20060228	8121	`ltk.ui.refactoring` v20060131

ibility layer generated by ReBA, all but one tests passed for the first two case studies. However, the single failing test is a test that was failing even when library and tests in Eclipse are compatible (both have the same version number).

In the `core.refactoring` study, since Eclipse CVS repository does not contain any tests for this library, we could not run any automated test suite before/after applying the compatibility layer. However, since the `ui.refactoring` is a graphical client, we exercised the UI manually, after applying the ReBA-generated compatibility layer. All usage scenarios reveal that the client is working properly.

To answer the efficiency question, we examined the compatibility layer and found that it contains only the classes that are truly needed by the client. To measure the memory and running overhead, we manually upgraded the source code of the client to make the client compilable; we used this upgraded client as the base for comparison with the compatibility layer. The memory overhead is much smaller in the case studies because the compatibility layer is very small compared to the large number of classes in each library.

All experiments were performed on a laptop with Intel Pentium M, 1.6GHz processor, and 512MB of RAM. The analysis done by WALA is scalable. WALA takes 2 minutes to create the points-to graph for `jdt.ui`, a relatively large library of 461 KLOC.

8. RELATED WORK

Restoring Source Compatibility Upgrading the source code of clients in response to library changes has been a long time research topic [3, 4, 7, 12, 16]. However, these solutions require that source code of clients be available. This is not always the case, especially in a modern large system comprised of a multitude of clients, some of these being open-source, others proprietary.

Restoring Binary Compatibility. There are several previous solutions [9, 13, 15, 17, 20] that aim to restore the binary compatibility. Purtilo and Atlee [15] present a language, Nimble, that allows one to specify how the parameters of a function call from an old module can be adapted to match the new signature of the function. Nimble handles a rich set of changes in function signatures: reordering of parameters, coercing primitive types into other primitive types, etc. Using Nimble, programmers write the mapping between the two versions of the function, then the system generates a function with the old signature that delegates to the function with the new signature. Nimble works for procedural languages, whereas ReBA is an approach for OO languages. The function signature changes in Nimble are similar with the ChangeMethodSignature refactorings. In addition, ReBA supports other types of API changes.

Keller and Hölzle [13] present BCA, a tool that automatically rewrites the bytecodes of Java clients using information that a user supplies in delta files. BCA compiles the delta files and automatically inserts the bytecode patches into bytecodes of clients before loading the client classes. Although BCA can handle a large range of API changes, it requires that developers use the BCA compiler and class loader. For security and legal reasons, developers are reluctant to use solutions that require changing the bytecodes.

The Adapter/Wrapper [11] pattern adapts a class's interface to the interface that a client expects. The adapter/wrapper object delegates to the adaptee/wrapped object. This solution works fine when the client code can be changed to use the wrapper wherever it used the original class. However, the adapter pattern has problems when restoring the binary compatibility. Because the wrapper and the wrapped object need to have different names, their identity and type information is different. Clients could potentially end up with objects of the wrapper as well as wrapped objects, depending on whether the client instantiates the wrapper directly, or it gets an indirect instance from the library. Comparing the wrapper and the wrapped through `equals`, `==`, or `instanceof`, would not work.

Warth et al. [20] present *expanders*, wrappers automatically generated from annotations provided by the user. However, expanders are only available in code that explicitly imports them, thus it requires that client code knows a priori about the expanders. Being based on the Adapter pattern, expanders have the same limitations.

Savga and Rudolf [17] overcome the above limitation by ensuring that (i) the wrapper has the same type information as the one expected by the client and (ii) the library itself would always return only instances of the wrapper objects. To restore the old API of the library they execute *comebacks* which are program transformations that revert all the API refactorings in the library. This enables old clients to work, but not new clients. In contrast, our solution does not modify the library, thus new clients continue to work. Their solution, like ours, preserves the edits in the library. However, their solution generates wrappers for all public classes in the library, whereas our solution generates adapted-classes only for the classes that have changed through refactoring, thus reducing the delegation overhead. In addition, our solution handles the deletion of APIs.

Table 4: **Effectiveness and efficiency of ReBA using Eclipse plugins as case studies.**

Study	API-Breaking Changes	CompileErrors Before	CompileErrors After	RuntimeFailures Before	RuntimeFailures After	MemoryUsage[B] Before	MemoryUsage[B] After	Memory Overhead	RunningTime[s] Before	RunningTime[s] After	Running Overhead
#1	1 RenameClass	29	0	25	1	140003416	140004056	$457*10^{-6}$%	14.746	14.748	0.009%
#2	1 DeleteClass	6	0	18	1	140044056	140044072	$11*10^{-6}$%	14.67	14.40	0.09%
#3	4 DelClass, 4 DelMethod, 1 RenaMethod 2 ChangeMethSig, 1 MoveClass, 1 DelField	27	0	12	0	17589288	17589402	$648*10^{-6}$%	8.452	8.453	0.01%

We solve the object identity problem in a different way. Rather than creating a wrapper and a wrapped object, we combine the wrapper (the old interface) and the wrapped object (the new interface) into one single class, the adapted-class. In addition, the points-to analysis ensures that the client and the compatibility layer always instantiate the same class, namely the adapted-class, thus preserving object identity and type information.

There is large body of work in the area of bridging components using architectural connectors [1]. Architectural connectors can be used to adapt interface mismatches. Although architectural connectors can adapt other types of changes besides syntactic interface change (e.g., behavioral, protocol), this approach requires that library developers write formal specifications. From these specifications, tools can automatically synthesize adapters, for example as in [2, 22]. ReBA does not make any behavioral guarantees since it handles edits as well, although it ensures that the behavior of the client is not changed with respect to API refactorings in the library. However, ReBA is practical because it does not require that library developers write specifications.

Lastly, previous solutions (except [12,17]) demand that library developers write annotations/mappings/specifications that are used by the upgrading tools. Library developers are usually reluctant to write such annotations. ReBA harvests the refactoring information from the library's refactoring engine, while liberating the library developers from the burden of manually writing annotations.

9. CONCLUSIONS

Managing software evolution is complex, and no technique will be a silver bullet. In the long run, source code must evolve to be compatible with new versions of libraries. In the short run, however, it is valuable to allow new versions of libraries to replace their old versions without changing the clients that use them. Our method of binary compatibility, embodied in the ReBA system, allows a library to simultaneously support the old and the new APIs. This allows it to be used in a system with both updated and non-updated clients. The producer of the library will make an adapted version that supports both versions of the API, but which has a lot of redundancy. The consumer of the library then makes a version of the adapted-library that eliminates the redundancy by specializing it to the client that is using it. The result, as shown by our evaluation, is a system that is both effective and efficient.

Compared to other binary compatibility techniques, our solution is easy to apply and does not require modifying bytecodes. It should be considered by library producers who are worried about the cost of library evolution on their customers.

10. ACKNOWLEDGMENTS

We would like to thank John Brant, Darko Marinov, Roger Whitney, Steve Lauterburg, Tanya Crenshaw, Shan Lu, Emerson Murphy-Hill, students in CS527 class at UIUC, and anonymous reviewers for insightful comments on previous drafts of this paper. First author thanks the UIUC Graduate College for partially funding this work through a Dissertation Completion Fellowship.

11. REFERENCES

[1] R. Allen and D. Garlan. Formalizing architectural connection. In *ICSE '94*, pages 71–80.
[2] M. Autili, P. Inverardi, A. Navarra, and M. Tivoli. Synthesis: A tool for automatically assembling correct and distributed component-based systems. In *ICSE '07*, pages 784–787.
[3] I. Balaban, F. Tip, and R. Fuhrer. Refactoring support for class library migration. In *OOPSLA '05*, pages 265–279.
[4] K. Chow and D. Notkin. Semi-automatic update of applications in response to library changes. In *ICSM'96*, pages 359–368.
[5] S. Demeyer, F. V. Rysselberghe, T. Gîrba, J. Ratzinger, R. Marinescu, T. Mens, B. D. Bois, D. Janssens, S. Ducasse, M. Lanza, M. Rieger, H. Gall, and M. El-Ramly. The LAN-simulation: A Refactoring Teaching Example. In *IWPSE'05*, pages 123–134.
[6] D. Dig, C. Comertoglu, D. Marinov, and R. Johnson. Automatic detection of refactorings in evolving components. In *ECOOP'06*, pages 404–428.
[7] D. Dig and R. Johnson. How do APIs evolve? A story of refactoring. *JSME*, 18(2):87–103, 2006.
[8] Eclipse Foundation. http://eclipse.org.
[9] I. R. Forman, M. H. Conner, S. H. Danforth, and L. K. Raper. Release-to-release binary compatibility in som. In *OOPSLA '95*, pages 426–438.
[10] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts. *Refactoring: Improving the Design of Existing Code.* Adison-Wesley, 1999.
[11] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.*
[12] J. Henkel and A. Diwan. CatchUp!: Capturing and replaying refactorings to support API evolution. In *ICSE'05*, pages 274–283.
[13] R. Keller and U. Hölzle. Binary component adaptation. *ECOOP'98*, pages 307–329.
[14] T. Lindholm and F. Yellin. *The Java(TM) Virtual Machine Specification, Second Edition*, pages 102–103.
[15] J. M. Purtilo and J. M. Atlee. Module reuse by interface adaptation. *Softw. Pract. Exper.*, 21(6):539–556, 1991.
[16] S. Roock and A. Havenstein. Refactoring tags for automatic refactoring of framework dependent applications. In *XP'02*, pages 182–185.
[17] I. Savga and M. Rudolf. Refactoring-based support for binary compatibility in evolving frameworks. In *GPCE'07, To Appear.*
[18] Apache Foundation Struts Framework. http://struts.apache.org.
[19] WALA Static Analysis Library. http://wala.sourceforge.net/.
[20] A. Warth, M. Stanojević, and T. Millstein. Statically scoped object adaptation with expanders. In *OOPSLA '06*, pages 37–56.
[21] W. E. Weihl. Interprocedural data flow analysis in the presence of pointers, procedure variables, and label variables. In *POPL '80*, pages 83–94.
[22] D. M. Yellin and R. E. Strom. Protocol specifications and component adaptors. *ACM Trans. Program. Lang. Syst.*, 19(2):292–333, 1997.