CatchUp!
Capturing and Replaying Refactorings to Support API Evolution

Johannes Henkel*
henkel@cs.colorado.edu

Amer Diwan*
diwan@cs.colorado.edu

ABSTRACT

Library developers who have to evolve a library to accommodate changing requirements often face a dilemma: Either they implement a clean, efficient solution but risk breaking client code, or they maintain compatibility with client code, but pay with increased design complexity and thus higher maintenance costs over time.

We address this dilemma by presenting a lightweight approach for evolving application programming interfaces (APIs), which does not depend on version control or configuration management systems. Instead, we capture API refactoring actions as a developer evolves an API. Users of the API can then replay the refactorings to bring their client software components up to date.

We present CatchUp!, an implementation of our approach that captures and replays refactoring actions within an integrated development environment semi-automatically. Our experiments suggest that our approach could be valuable in practice.

Categories and Subject Descriptors

D.2 [**Software Engineering**]: Coding Tools and Techniques

General Terms

Design

Keywords

Software evolution, refactoring, application programming interfaces

*This work is supported by NSF grants CCR-0085792, CCR-0133457, CCR-0086255. Any opinions, findings and conclusions or recommendations expressed in this material are the authors' and do not necessarily reflect those of the sponsors.

ICSE'05, May 15–21, 2005, St. Louis, Missouri, USA.

Table 1: The # jar files roughly indicates the number of components.

	# jar files	# classes
Apache Tomcat 5.0.27	55	4,070
Eclipse 3.0-M7	147	21,753
JBoss 4.0.0RC1	205	20,643

1. INTRODUCTION

Developing and maintaining software remains a difficult problem. A promising technique is to keep design and implementation clean by restructuring them whenever complexity increases. The idea of such restructurings is to improve internal properties of the software, such as a particular kind of extensibility, without changing what the software does. Many development environments provide semi-automatic support for *refactoring* [8, 7, 2], which is the object-oriented variant of software restructuring. For example, modern integrated development environments *(IDEs)* such as Eclipse[1] support the "encapsulate field" refactoring, which replaces read- and write-accesses to a field with calls to automatically generated `get`- and `set`-methods. Refactoring support within IDEs has made it less cumbersome and expensive to improve code quality.

However, refactoring and other techniques cannot unleash their full potential when reusable software components *(libraries)* are being developed and used. If a library developer wants to refactor his code, he has to limit himself to changing the internal implementation or to expanding the application programming interface (API), but he cannot remove or change existing parts of the API without breaking client code. Breaking client code comes at a high cost, which is why library developers often have to refrain from making changes that could reduce the complexity of the API or improve efficiency. As large software projects are increasingly built from reusable components, this inflexibility becomes more significant. Fowler acknowledges this cost in his book; he advocates not to publish the interfaces of reusable components unless absolutely necessary [2, pages 64, 65]. Table 1 shows how many components some popular open source projects contain; most of these components have published interfaces, and evolving them is a daunting problem.

We present a new approach to address this problem: Our idea is to record how the library developer changes the API. As the library developer invokes refactoring actions within an IDE, we record them. Then, we play back the

[1]Eclipse is available at `http://www.eclipse.org/`.

	classes	methods	fields
Java 1.4.2 runtime library	52	365	76
Apache Tomcat 5.0.27	89	339	71
Eclipse 3.0 plugins	177	729	306
JBoss 4.0.0RC1	131	587	99

Table 2: Number of deprecated classes, methods and fields for Java programs.

recorded refactoring actions to update client code automatically. Therefore, with our approach, recording and replaying many API changes comes at a low cost for both the library developer and the developers of client applications. Our approach is light-weight: It does not require a centralized infrastructure such as version control or configuration management systems, and it integrates well with modern development environments.

While not all possible changes to an API can be captured as refactorings supported by current IDEs, our experiments indicate that even with the current IDE support for refactorings, our approach is worthwhile and cost-effective. We also suggest new refactorings which would be particularly helpful for our scenario. Based on our experiments, we believe that a realization of our approach in large-scale industrial projects could lead to more flexibility in developing reusable software components, in particular more flexibility in dealing with changing requirements. For example, when we studied the current use of deprecation, we found that there is a need for better tools to support API evolution: Java methods become *deprecated* but are rarely eliminated due to compatibility requirements and the high cost of updating client code. With our approach, developers can eliminate most of them with little cost.

Section 2 motivates and describes our approach using an example. Section 3 describes how two users, a *library developer* and a *client developer* experience our prototype CatchUp!, which implements the concepts described in this paper. Section 4 describes how we implemented CatchUp! within the Eclipse development environment. Section 5 summarizes the progress of our implementations and outlines future work, Section 6 discusses related work, and Section 7 concludes.

2. MOTIVATION AND OVERVIEW

> The need for deprecation comes about because as a class evolves, its API changes. Methods are renamed for consistency. New and better methods are added. Attributes change. But making such changes introduces a problem: You need to keep the old API around until people make the transition to the new one, but you don't want developers to continue programming to the old API. The ability to mark a class or method as "deprecated" solves the problem...
>
> *"The Java Tutorial", Sun Microsystems, 1995-2004*

Table 2 shows that large Java programs contain many deprecated classes, methods, and fields. The quote from the Java tutorial above promises that deprecation can be used to evolve an API; however, we found that at least for the Java runtime library, deprecated entities which are part of the published interfaces are almost never removed, which means that the number of deprecated entities keeps going up. In some instances, Sun developers even resurrected methods by removing the deprecation tags in later versions, apparently giving up on transitioning to a cleaner API.

refactoring to apply	**# deprecated methods**
rename method	85
delete method and replace with Java expression	64
delete method and replace with Java statements	27
reason for not refactoring	
no obvious replacement	75
design changes too radical	31
erroneously public or not well defined originally	18
total	300

Table 3: Deprecation in the Java core libraries (Sun JDK 1.4.2) and how to eliminate it.

A preliminary study suggests that in some open source projects, deprecation is more common than in the Java standard libraries, and library developers sometimes remove deprecated methods for major releases. As an example, the HttpClient class from the Jakarta HttpClient library[2] had 12 deprecated methods (out of 26 public methods) in version 2.0.2. In version 3.0-beta1, the library developers removed 10 deprecated methods, and deprecated 5 additional methods, which means that version 3.0-beta1 has 7 deprecated methods (out of 18 public methods). When library developers deprecate and remove as aggressively as in this case, client applications may choose to rely on the old version of the library for an extended period. The library developers now have to maintain two libraries, and applications that use many components can suffer from libraries with incompatible dependencies. For example, consider an application that needs to use libraries A and B, where A depends on HttpClient 3.X and B depends on HttpClient 2.X.

The use of deprecation points out serious problems with library development: once an API is published, developers are forced to maintain it. Whether or not developers choose to deprecate and later remove unwanted methods and types largely depends on the amount of client code that needs to be fixed in order to migrate to the new API. Our approach lowers this "deprecation/migration tax", since many deprecated entities can be removed by employing a refactoring that transforms client code automatically. In Table 3 we estimate how many of the deprecated documented methods and constructors shipping with the Sun Java Delevelopment Kit 1.4.2 could be refactored, and which kind of refactoring would be applicable. Many of the deprecated methods could be eliminated with the "rename method" refactoring (surprisingly many of these were deprecated due to typos in the method name). 64 methods could be eliminated by replacing each call with a Java expression, e.g. `enable()` $\mapsto$ `setEnabled(true)`, and 27 could be eliminated by replacing each call with a list of Java statements; they are thus slightly harder to refactor due to the stronger static analyses required. In total, we estimate that about 59% of the deprecated methods could be eliminated by refactorings.

However, deprecation in current use is just the tip of the iceberg, since it assumes the high costs of updating clients

[2] `http://jakarta.apache.org/commons/httpclient/`

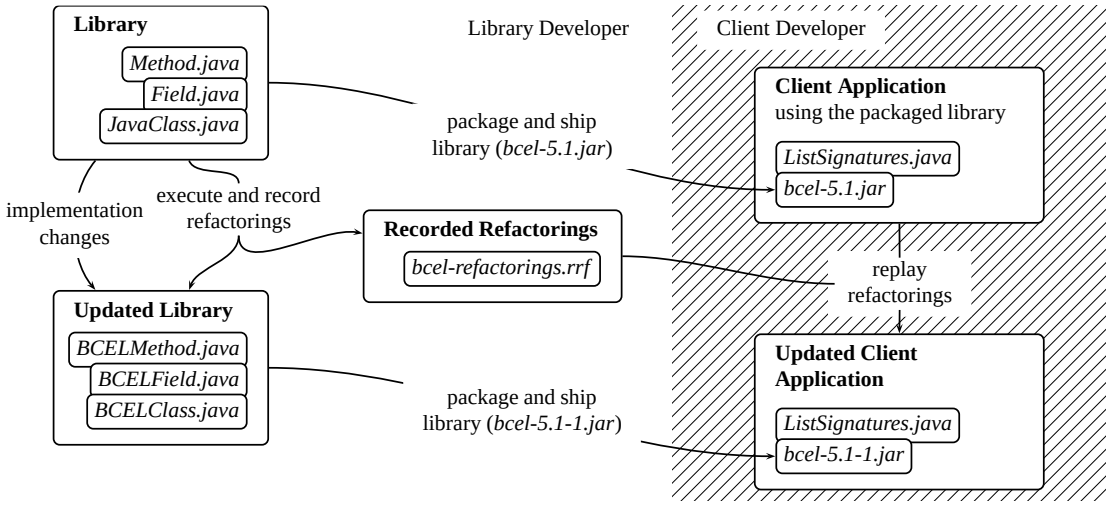

Figure 1: One step in the evolution of a library.

and maintaining old versions of libraries. We speculate that lower costs for updating client code may lead to greater flexibility to API design, ultimately making API evolution more practical.

Fig. 1 describes our approach for recording and replaying refactorings. We consider a scenario in which one party develops a library, which is then used by many other parties to build client applications. For clarity, we present only the activities carried out by one representative *library developer*, and the activities carried out by one representative *client developer*, who writes code which uses the library. The client application has to be updated to accommodate the evolving library. We record and replay refactorings to reduce the cost of these updates. The figure shows how the development process moves from one release of the library to the next release.

The upper left box in Fig. 1 depicts Version 5.1 of the *BCEL* library, which we use as our running example. *BCEL* (bytecode engineering library)[3] helps developers to write programs that manipulate Java bytecode. The box contains three representative classes which model Java bytecode entities that can be manipulated using the library: `Method`, `Field`, and `JavaClass`. As the library developer evolves the library from Version 5.1 to Version 5.1-1, he renames the three classes into `BCELMethod`, `BCELField`, and `BCELClass`. Adding `BCEL` prefixes to these classes makes writing programs that use both BCEL and the Java reflection API more convenient by avoiding name clashes, but names in programs are a matter of taste and our approach does not depend on any particular taste.

Library developer and client developer do not share a common repository for source code. Instead, the library developer packages up the library from time to time and delivers it to the client developer as a *JAR file*, which is a compressed archive containing the Java bytecode; this happens twice in Fig. 1. Our approach does not assume that the library is being shipped with source code—binary-only releases are sufficient. The relatively loose coupling between the library developer and the client developer is typical for open-source projects. In Fig. 1, the upper right box depicts `ListSignatures`. This is a small application which uses Version 5.1 of the BCEL library to print the signatures of fields and methods contained within a Java bytecode file.

Figure 1 shows that while the library developer refactors his code, the integrated development environment (IDE) collects a trace (`bcel-refactorings.rrf`), which the library developer ships to the client developer, along with the new release of the library (`bcel-5.1-1.jar`). The library developer invokes refactorings supported by the IDE explicitly through a menu within the IDE; some refactorings require further information, which is collected in dialogs. We record all this information in our trace file (`bcel-refactorings.rrf`), since it is needed to later replay the refactorings. Relying on the IDE to collect the refactoring trace has the advantage that the information we record is correct, unambiguous, and complete. Beyond changing the API by invoking supported refactorings, the library developer is free to change the implementation of the library with or without using refactorings supported by the IDE, as long as these changes do not impact the API.

The client developer can update his code (`ListSignatures.java`) to use the new version of the BCEL library by *replaying* the recorded refactorings shipped with the library (`bcel-refactorings.rrf`). Thus, in this way the client developer is able to migrate his code to the new version of BCEL without manually updating his code. Note that since the client code is updated automatically, the library writer does not have to maintain deprecated methods. An updated version of the client application is shown in the lower right of Fig. 1.

In principle, any change to a software program that preserves behavior can be understood as a refactoring. To provide as much benefit to their users as possible, modern IDEs focus on the most commonly used, low-level refactorings (e.g. Move Method), which are described in Martin Fowler's book [2]. As Joshua Kerievsky's book demonstrates [5], IDE support for the low-level refactorings is very useful even for programmers who want to incorporate larger refactorings, such as introducing a design pattern. While our approach is not limited to low-level refactorings, this paper focuses on the basic refactorings that are supported by current development environments.

[3]BCEL is available at `http://jakarta.apache.org/bcel/`.

3. USER EXPERIENCE

This section describes how two users, a library developer and a client developer, experience CatchUp!, our prototype implementation of the idea described in Section 2. Our prototype is implemented as a plugin for the Eclipse IDE, which is an open-source Java IDE. For illustration, we use our running example introduced in Section 2, Fig. 1.

3.1 Evolving the BCEL Library

Let's assume the library developer has just shipped BCEL version 5.1, as shown in Figure 1. Before he continues with evolving the library, he downloads and installs CatchUp!, our Eclipse plugin. To activate the CatchUp! plugin, he invokes the "start refactoring session" menu entry within the Eclipse IDE. This initializes CatchUp! with an empty recorded refactoring trace and also opens a view in the IDE which continuously displays the trace. As discussed in Section 2, the library developer renames some of the classes within BCEL (`Method` ↦ `BCELMethod`, `Field` ↦ `BCELField`, `JavaClass` ↦ `BCELClass`) by invoking the "RenameType" refactoring within Eclipse. The only difference to a regular Eclipse installation is that refactorings are now recorded and added to the refactoring trace.

Fig. 2 shows a screenshot of the recorded refactorings view in Eclipse, which displays the three "RenameType" refactoring events carried out by the library developer. For each of these refactoring events, CatchUp! has recorded the type to be renamed and the new name. Additionally, the "recorded refactorings" view allows the library developer to annotate a refactoring event with a comment. By annotating the refactoring events with comments, the library developer can explain why he is making particular changes, for example "Renaming to avoid name clashes". The comments are displayed when the client developer replays the refactorings.

After renaming the classes, the library developer uses a refactoring to introduce a factory. BCEL contains a class `Type` and several subclasses to model types such as primitive types, array types, and object types. `ObjectType` is a subclass modeling types introduced by classes, e.g. `java.util.LinkedList`. The constructor of `ObjectType` is used by other classes within BCEL itself, but it is also used by many client applications. It looks as follows:

```
1 public ObjectType(String class_name) {
2   super(Constants.T_REFERENCE, "L" +
3         class_name.replace('.', '/') + ";");
4   this.class_name=class_name.replace('/','.');
5 }
```

The library developer sees an opportunity to improve the performance of BCEL itself and other applications using the `ObjectType` class; he wants to introduce a factory, which allows him to cache instances of `ObjectType`. Therefore, he applies Eclipse's "Introduce Factory" refactoring to the constructor. A dialog asks for the name of the new factory method, in which class it should be created, and whether or not the visibility of the constructor should be `protected`. Eclipse transforms the above snippet into:

```
1 protected ObjectType(String class_name) {
2   super(Constants.T_REFERENCE, "L" +
3         class_name.replace('.', '/') + ";");
4   this.class_name=class_name.replace('/', '.');
5 }
6
7 public static getObjectType(String class_name){
8   return new ObjectType(class_name);
9 }
```

Additionally, Eclipse replaces all `new ObjectType(...)` invocations with calls to `getObjectType(...)`. The library developer can now reimplement the body of `getObjectType` so that it caches `ObjectType` instances. CatchUp! records the refactoring, which means that client applications will also enjoy the benefit of this optimization without having to be patched manually.

```
1  public class ClassParser {
2    private static final int BUFSIZE = 8192;
3    ...
4    public ClassParser(InputStream file,
5                       String file_name) {
6      this.file_name = file_name;
7      ...
8      this.file = new DataInputStream(
9          new BufferedInputStream(file, BUFSIZE));
10   }
11   ...
12 }
```

Figure 3: `ClassParser` class from the BCEL library (simplified).

Next, the library developer tackles another problem: The `ClassParser` class (Fig. 3) uses a hard-coded buffer size for loading bytecode files. The library developer would like to give the control over the buffer size to the client application. One way to accommodate this need without sacrificing client compatibility is to add a parameter `int bufferSize` to all constructors of `ClassParser` and to then create a convenience constructor for each constructor. The convenience constructor does not have the additional parameter but just delegates using the default buffer size. Sometimes, this option is adequate; however, each time such a change is made, the library developer has to double the number of constructors. Furthermore, the signature of a convenience constructor might conflict with another constructor already in the class. Our approach provides a more flexible solution: The library developer developer can introduce a new parameter and use the default buffer size as a default parameter, which will be added at all existing call sites. This works as follows.

First, the library developer changes the visibility of the `BUFSIZE` field to `public` by editing the source code. Then, he uses the "Rename Field" refactoring supported by Eclipse to change the name of `BUFSIZE` to `DEFAULT_BUFSIZE`. This generates a refactoring event which is shown in the recorded refactorings view. The library developer then applies the "Change Signature" refactoring to the constructor shown in Fig. 3. "Change Signature" is a powerful refactoring which allows the developer to add and delete parameters, and to modify the return type. In this case, the library developer adds the parameter `int bufferSize` and sets its default value to the Java expression `ClassParser.DEFAULT_BUFSIZE`. Eclipse will insert the default value of the added parameter at all call-sites. CatchUp! records the "Change Signature" event in the refactoring trace, including the default parameter value. After introducing the parameter, the library developer edits the code in the body of the constructor to replace the use of the `DEFAULT_BUFSIZE` constant (alias `BUFSIZE` in Fig. 3, l. 9) with a use of the parameter `bufSize`. Fig. 4 shows the

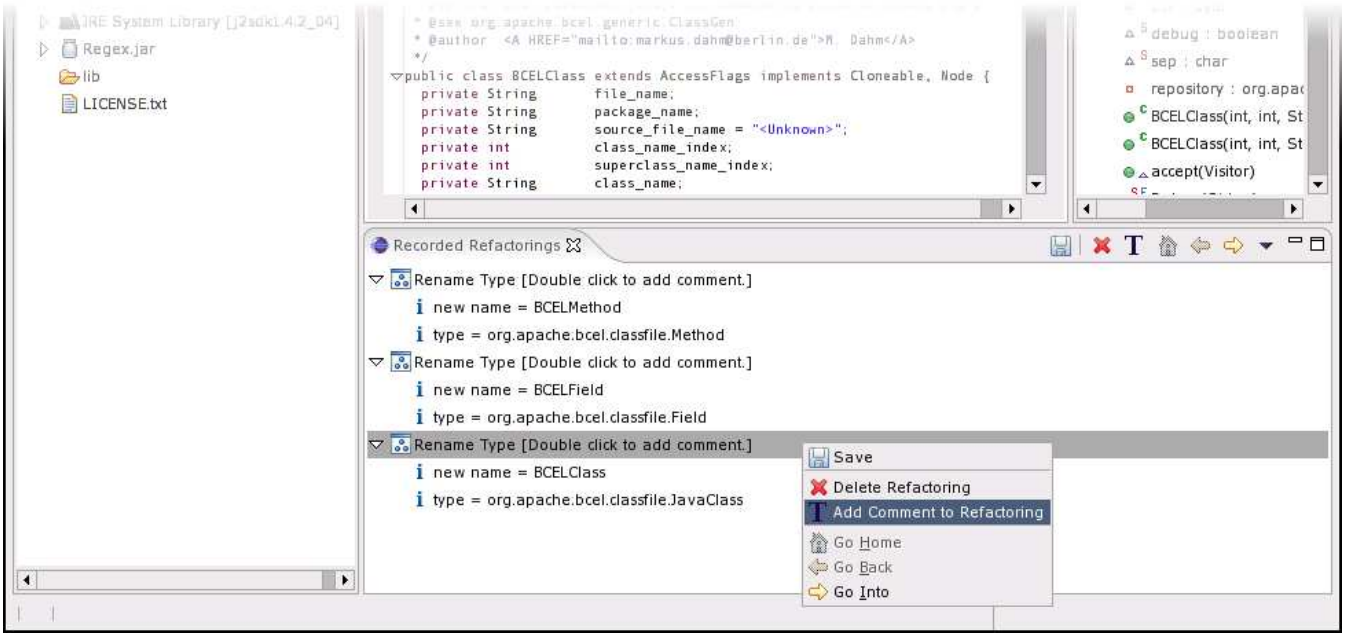

Figure 2: Eclipse view for recorded refactorings.

```
1  public class ClassParser {
2    public static final int DEFAULT_BUFSIZE = 8192;
3    ...
4    public ClassParser(InputStream file,
5                       String file_name,
6                       int bufSize) {
7      this.file_name = file_name;
8      ...
9      this.file = new DataInputStream(
10         new BufferedInputStream(file, bufSize));
11   }
12   ...
13 }
```

Figure 4: `ClassParser` after refactorings.

resulting version of the `ClassParser` class.

The recorded refactorings view also allows users to remove refactoring events from the trace (Fig. 2). Sometimes, the library developer knows for sure that a refactoring will not affect client code. Such refactoring events are not harmful, but they clutter the refactoring trace, which also serves as a changelog of the API (Section 3.3). The library developer therefore uses his judgment to remove unnecessary events from the trace. In our example, the "Rename Field" event discussed in the previous paragraph can be removed from the refactoring trace: Since it was `private` in the previous version of the library, and no reference from the client application could be introduced by the refactorings before the rename event, the library developer is certain that client code cannot reference `BUFSIZE` at the point when the rename event happens. Notice that this is true even though `BUFSIZE` is a `public` field when it gets renamed.

Now the library developer is ready to release a Version 5.1-1 of the BCEL library (Fig. 1). He saves the refactoring trace as a text file by clicking the "Save" button of the recorded refactoring view (Fig. 2), and uses Eclipse's "Export JAR" functionality to generate a new JAR file. He ships both the JAR file and the refactoring trace to the client developer.

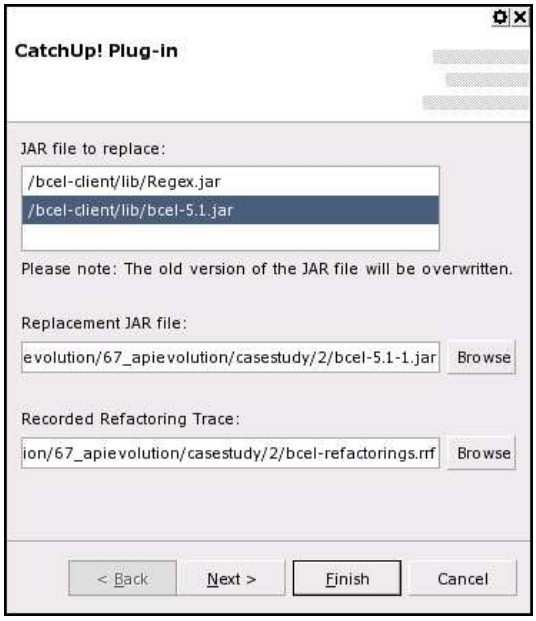

Figure 5: Eclipse wizard for replaying refactorings.

3.2 Replaying the refactorings

The client developer currently uses Version 5.1 of the BCEL library, but he would like to move to Version 5.1-1 (Figure 1). Within the Eclipse IDE, he invokes our plugin by selecting "Update Library" from the main menu. This brings up our replaying wizard (Fig. 5). At the top of the dialog, the replaying wizard lists the JAR files that the current project is using; the library developer selects which JAR file he wants to update, `bcel-5.1.jar`. Below, the library developer selects the replacement JAR file, `bcel-5.1-1.jar`,

and the refactoring trace `bcel-refactorings.rrf`. After having provided all necessary information, the library developer clicks the "Next" button.

This brings up a detailed preview of the changes shown in Fig. 6. This screen has three segments. At the top, the refactorings are listed, with the same view that we use for displaying recorded refactorings (Fig. 2). When the client developer clicks on one of these refactorings, the replay wizard displays annotations in the comment field (upper right), and also populates the lower two segments of the dialog. In the middle segment, the client developer can view a hierarchical change set that shows which resources will be changed, and in which way. In our example, the client developer is exploring how `ListSignatures.java` changes, which is the code of the client application (see Fig. 1). When the client developer clicks on the "update type reference" detail in the middle view, the affected source code is highlighted in the lower section of the screen. Note that the steps shown by this preview are incremental; they show how the original version of the client code is transformed step-wise into the final version. For example, the version shown in the lower left corner of Fig. 6 is the version just before applying the "Rename Type" refactoring to rename `Field` into `BCELField` (notice that `BCELMethod` has already been renamed in this version), and the version shown in the lower right corner is the version just after applying this refactoring. Since subsequent refactorings can destroy the results produced by previous refactorings, we feel that such a step-wise view is less confusing than an aggregate view of the changes, in particular when the client developer wants to exclude particular refactorings or changes from execution. We plan to add an aggregate view in a future version of our tool.

After reviewing and excluding particular changes, the client developer can commit the proposed changes or cancel the dialog.

3.3 Manually replaying refactorings

There are reasons that could prevent developers from using CatchUp! to replay the refactorings. For example, the client application might be written in a language extension that is incompatible with Eclipse, such as an embedded SQL dialect, or the developer is reluctant to use any tool that transforms his code automatically.

Even though this situation is not optimal, our refactoring trace file is in a simple, human-readable XML format, which can be used as a precise documentation for the API changes. To further improve the human readability of the trace, one can write a style sheet that formats the refactoring trace as a HTML document. Again, notice that the option to annotate the refactoring trace with comments while it is being recorded can add real value. As our tool matures, we also envision CatchUp! support for development environments other than Eclipse.

4. IMPLEMENTATION

This section describes how we implement our prototype CatchUp!, which incorporates the ideas described in Section 2. We implement CatchUp! as a plug-in for the Eclipse IDE; however, many of the problems and solutions we describe in this section should apply to other development environments as well. We chose Eclipse because it is open-source, which gave us great flexibility for our implementation.

4.1 Refactoring Support in Eclipse

Eclipse models refactorings as classes. The lifecycle of a refactoring object involves these steps: (i) Instantiating the appropriate refactoring class with parameters indicating which entities should be refactored; (ii) Checking whether the refactoring is applicable; (iii) Collecting all remaining information necessary to execute the refactoring; (iv) Computing a *change object* which models the changes to source code and filesystem; and (v) Committing the change.

As an example for step (i), to instantiate a `MoveStaticMembers` refactoring, one has to provide the *abstract syntax tree* (AST) nodes modeling the set of members that one wants to move. For example, suppose `methodF` and `methodG` are two AST nodes modeling two Java methods `f` and `g`. The following code creates a refactoring object `refObj`, which will help us to move `f` and `g` into a Java class `C`. [4]

```
MoveStaticMembersRefactoring refObj =          (step (i))
   new MoveStaticMembersRefactoring(
      new IMember[]{methodF,methodG});
```

The refactoring object then provides a method that can be used to test whether or not the refactoring is applicable (step (ii)); for example, `MoveStaticMembers` requires that the AST nodes provided as parameters are static members belonging to the same class. E.g.

```
if(refObj.checkInitialConditions().          (step (ii))
   hasFatalError())
      throw new Exception(
         ''Refactoring not applicable.'');
```

A wizard dialog tailored to the refactoring collects the the remaining information (step (iii)). Handler routines for the wizard dialog pass the user input to the refactoring object and use the validation services provided by the refactoring object. These can include program analyses which check for problems with the user-supplied input. A simple example is to detect name clashes: Moving a static method into a class that already has a static method with the same name and parameter types will be detected and reported to the user by the wizard dialog. Once the user inputs validate successfully, the refactoring object computes a change object (iv), which is a recipe for modifying the source code.

```
refObj.setDestinationType(''C'');            (step (iii))
if(refObj.checkFinalConditions().
   hasFatalError())
      throw new Exception(
         ''There is still a problem.'');
Change change = refObj.createChange();       (step (iv))
```

The change object consists of editing deltas which describe exactly how the source code needs to change. The wizard dialog presents a graphical view of the change to the user, and the user can commit the changes or cancel the refactoring dialog.

```
change.perform();                            (step (v))
```

4.2 Recording Refactorings

Our strategy is to extract enough information from the refactoring objects so that we can later recreate them to replay the refactoring (Section 4.3). Furthermore, we want to

[4]The example code provided in this section is greatly simplified.

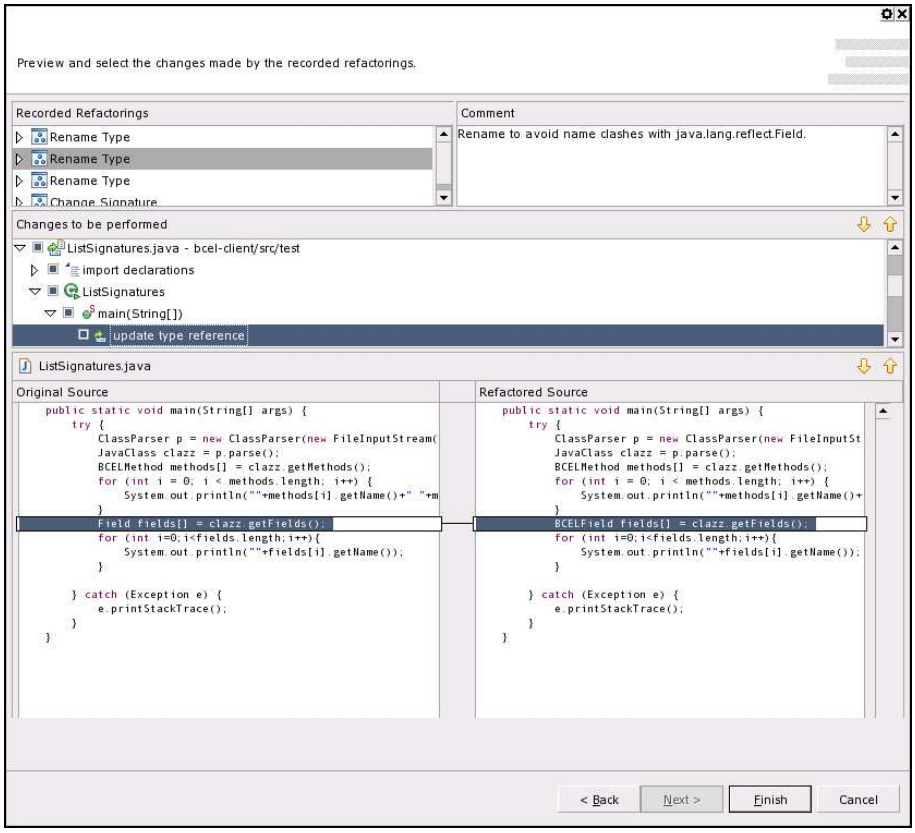

Figure 6: Eclipse preview for replaying refactorings.

only record refactoring events that the user commits. Unfortunately, Eclipse provides no hooks for observing refactoring activities. This means that we had to add instrumentation to one of the plugins provided with Eclipse. The constraints for where to insert the instrumentation are: (i) Since committing refactorings by executing the edits modeled within the change object causes destructive updates and thereby invalidates some of the information collected by the refactoring object, we need to extract the information from the refactoring object before the changes become effective. (ii) We have to wait until all information has been accumulated within the refactoring object.

It would seem sufficient to insert instrumentation triggers whenever a user clicks the "Finish" button of the refactoring dialog. Unfortunately, there are many implementations of this dialog (one for each refactoring); we want to make it easy to add support for new refactorings. Capturing an event whenever a refactoring dialog has been closed does not work either, since Eclipse creates a new thread which executes the changes in the background to increase the perceived responsiveness of the development environment. This means that if we try to extract the information from the refactoring object right after the refactoring dialog has been closed, Eclipse's internal representation may or may not be valid, depending on when the thread executing the changes is scheduled by the Java Virtual Machine.

To avoid these problems, we inserted code that passes the refactoring object to our own plugin in two places: (a) just after a change object has been created (after step (iv) in Section 4.1), and (b) just after the refactoring has been committed (after step (v) in Section 4.1). After a change object has been created (a), we collect all necessary information from the refactoring object and store it in a corresponding trace object. We maintain a mapping from refactoring objects to trace objects. If the user commits the refactoring (b), we add the corresponding trace object to our refactoring trace. The mapping is implemented using a `WeakHashMap` to prevent memory leaks. We store the refactoring trace in a simple, human readable XML format.

4.3 Replaying Refactorings

To replay refactorings, we recreate Eclipse's refactoring objects according to the specifications given in the refactoring trace, and apply the changes they compute to the client application. More precisely, this works as follows.

We start in the upper left of Fig. 7 with the client application (`ListSignatures.java`) and the library packaged as a JAR file (`bcel-5.1.jar`). To instantiate the refactoring classes provided by Eclipse, we need to provide references to the source code of the entities to be refactored. For example, to rename `Method` to `BCELMethod`, we need to create an instance of the "Rename Type" refactoring, which requires a reference to the AST (abstract syntax tree) node modeling `Method`.

CatchUp! generates Java source code stubs from the bytecode for all classes contained in the library (Fig. 7, first transition). The source code stubs only need to provide the declarations for the classes, methods, and fields that are part of the API (e.g., no private methods); this is sufficient to create Eclipse's refactoring objects. Since method bodies are not important, we generate trivial bodies such as `return null;`. We implemented our stub-generator using

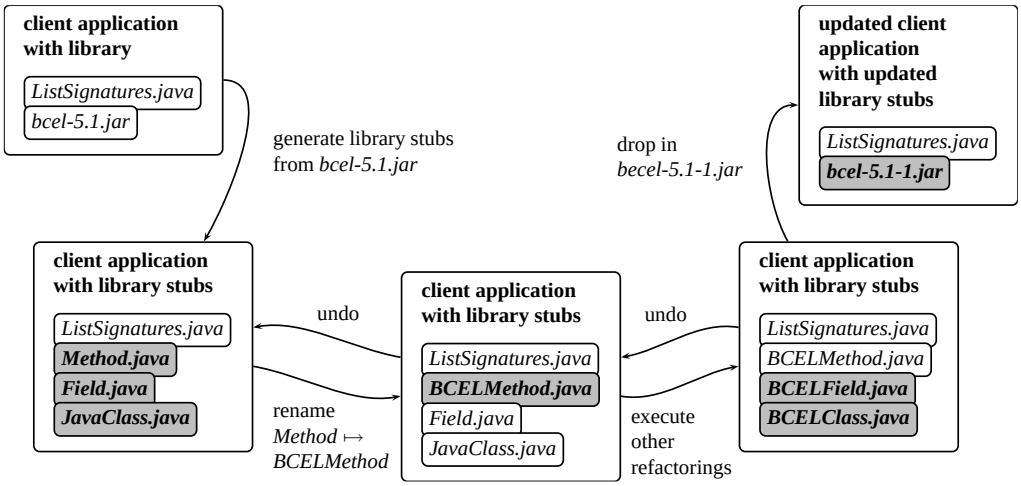

Figure 7: Implementation of replaying refactorings.

the bytecode engineering library BCEL[5]. Generating stubs for the BCEL library (370 bytecode files) takes about 20 seconds on a Pentium M laptop (1.3 Ghz). While this is reasonable for a prototype, a production version of our tool could be optimized by pruning many of the stubs. In particular, only the following stubs are needed: (i) stubs which are referenced by the client application, (ii) stubs which are referenced by a refactoring, (iii) the transitive closure of (i) and (ii), i.e., stubs which are referenced by (i), (ii), or (iii). Note that many methods and fields, even public ones, can be pruned as well.

After generating the stubs, we remove the JAR file containing the original library from the classpath of the client application. At this point, we can use the refactoring objects provided by Eclipse to execute the changes. In Fig. 7, we first apply the refactoring that renames `Method` to `BCELMethod` (second transition). Similarly, we apply all remaining refactorings: Each refactoring object computes a change object, which we apply to move to the next version of the code. Moving from the lower left to the lower right in Fig. 7 corresponds to moving from the unpatched version of the client application to the updated version. This is almost everything we need for visualizing the refactoring steps in the preview dialog (Fig. 6): As the user clicks to the next refactoring in Fig. 6, we move one step to the right in Fig. 7. Fortunately, executing a change object results in an *undo object* being computed, since change objects implement the command pattern as described in the "gang of four" book [3]. Therefore, when we move right in Fig. 7, we store the undo objects. This allows us to move back and forth between refactorings, providing the "preview" experience to the user.

Once the user clicks the "Finish" button of the refactoring wizard (Fig. 6), we execute all outstanding refactorings to arrive in the lower right of Fig. 7. Then, we remove the stubs and drop the new version of the library (`bcel-5.1-1.jar`) into the classpath of the client application.

[5]Yes, the very same library that we also use as a running example.

5. IMPLEMENTATION STATUS AND FUTURE WORK

Our future work will focus on improving the usefulness of CatchUp! as a software development tool.

First, we currently only support a subset of the refactorings supported by Eclipse. Table 4 gives a description of the current implementation status. The names of the refactorings in the table correspond to refactoring classes provided by Eclipse; between some refactorings, there is an overlap in functionality. The most flexible refactoring is the change method signature refactoring, which we support. Adding support for all refactorings is tedious but not difficult. It involves adding code for capturing and replaying, which are both around 30 lines of code for most refactorings, and sometimes even less. Recently, the Eclipse project has added support for participant-based refactorings[6], which allows participant objects to observe *some* refactorings; participants can then update update references to source code in text-files and other non-Java code files such as Enterprise Java Beans. With more refactorings supporting participants, the code for recording refactorings could be further simplified.

A second improvement that would make it more convenient to use the tool would be to embed the refactoring trace within the JAR files. This means that only one download will be necessary to update a library. When embedding the refactoring trace within a JAR file, we will store all refactoring events since the beginning of the project within the trace. This will allow us to update to the new library from any previous version, by only executing the refactorings that are new. This is significantly more convenient than our current tool, which can only move from one library version to the next.

We hope to improve the XML format for storing the refactoring trace. Currently, the names of the refactorings in the trace are Eclipse-specific; we have engineered the format to make it easy to implement within Eclipse. A more general format will allow for easier collaboration between development environments.

[6]`http://dev.eclipse.org/viewcvs/index.cgi/~checkout~/jdt-ui-home/r3_0/proposals/refactoring/participants.html`

Table 4: Refactorings currently supported by our prototype implementation. For more details and updated information, please visit `http://www-plan.cs.colorado.edu/henkel/catchup/`.

Refactoring name	**Description**	**Support**
Rename Type	Rename a class or an interface	Full support
Moving Java Elements	Move a Java type from a package to another package or into a class	Full support
Move static member	Move type, method, or field from one type to another	Untested
Change Method Signature	Change method name, return type, visibility Add, remove, rename parameters Set default value for added paramters Add / remove exception types	Full support
Rename non-virtual method	Rename a non-virtual method	Recording only
Rename virtual method	Rename a virtual method	Recording only
Change Type	Generalize the type of a field, a paramter, or a return type	Recording only
Rename Field	Rename a field and optionally rename getters and setters	Replay untested
Use super-type where possible	Replace all occurrences of a type with a particular super-type	Recording only
Introduce Factory	Add a factory method Optionally make the constructor private Update clients to use factory method	Recording only

We have seen some cases in which we would like to add refactorings to Eclipse. For example, we have come across situations in which we would like to replace all calls to method `f()` with calls to method `g()` (before deleting `f`). This is especially useful to clean redundant methods out of libraries which are using the Java deprecation language feature. Currently, a way to achieve this using Eclipse's refactorings is to rewrite `f(){...}` into `f(){g();}` and to then use the "inline method" refactoring to inline `f` at all call sites, thereby eliminating `f` and replacing all calls to `f` with calls to `g`. This is is rather involved and unintuitive, and it also exposes the implementation of the library and therefore only works if the source code of this method ships to the client developer. It would be nice to have a "Remove Method" refactoring instead, which allows users to replace all calls with a Java expression.

More generally, we believe that there are refactorings that are particularly useful for refactoring APIs; as we gain more experience from our prototype, we want to identify and support these refactorings particularly well. We believe that API refactorings should also take advantage of formal specifications that describe the "protocols" used by client applications. For example, one way to evolve a class might be to add a method `f` which has to always be called before another method `g` can be called; this could be expressed as a refactoring which also captures the temporal constraint `f` *before* `g`, and updates client code accordingly.

6. RELATED WORK

Refactoring is a well studied area. Opdyke introduced the term in his PhD dissertation [8]. A recent survey by Mens and Tourwe covers refactoring research [7]. A book by Fowler explains many refactorings in great detail and motivates their usefulness in practice [2].

Chow and Notkin present a system for semi-automatically updating applications in response to library changes [1]. Our work shares the same intention, which is to reduce the cost of changing client applications. However, our use of integrated development environments to record refactorings alleviates the library programmer from writing a library change specification by hand. Also, while replaying change specifications, the programmer of the client application can study in detail the effect of each refactoring step on the client application, along with annotations added by the library developer. Furthermore, by reusing the implementations of refactorings within an IDE, our implementation delegates the tricky work of transforming and validating abstract syntax trees to well-tested, industrial implementations.

Much work that has been done in database schema evolution and persistent type evolution (e.g., [6]) is complementary to our work. These threads of research analyze changes in source code, in particular type declarations, to infer in which way the types or schemas evolve. Even though automated inference of change operations or, in our case, detection of refactorings could be used as a basis for our system, we leverage the rich information available in modern development environments instead, which is available anyhow and very accurate.

Keller and Hölzle [4] present a method for patching software components for binary compatibility. Their idea would be a useful addition for our system: Instead of replaying the refactorings to source code, we would generate specifications for the binary component adaptation system and modify bytecode when it is loaded into the JVM. While this feature would be useful for dealing with legacy software components, most refactorings are unsafe due to Java language features such as reflection and native methods. In practice, this is not problematic if the source code is reviewed and the application is retested as advocated by Fowler [2], but it would become an issue if we were to refactor binaries.

Concurrently to us, Borland, a commercial IDE vendor, presented a set of new products at the JavaOne conference, which allow teams of developers to share refactorings.[7] A similar approach, based on Eclipse, is being implemented at Lund University[8]. However, both solutions are centralized and integrated within a configuration management server. Therefore, they only scale to projects which share a common infrastructure, such as a central repository. In contrast, our solution is much more light-weight and also accommodates loosely coupled projects which distribute their changes by

[7]June 29th, 2004, http://java.sun.com/javaone/borland2.html
[8]http://www.lucas.lth.se/cm/cmeclipse.shtml

posting new JAR files on websites, a technique typical for open-source projects.

7. CONCLUSION

We described a novel approach to deal with changing requirements for software components. Our idea is to extend development environments to record and replay refactoring events. This allows library developers more flexibility in evolving APIs, and client developers more convenience by automatically porting their application to a new version of a library.

We implemented our idea as an Eclipse plugin called CatchUp!. We described how users experience CatchUp!, and how it is implemented. While we have some anecdotal evidence that the tool is usable, we hope to collect more case studies to guide the future development of CatchUp!. We will make CatchUp! available for download in the near future[9] and are interested in your feedback.

8. ACKNOWLEDGEMENTS

We thank Christoph Reichenbach, Martin Hirzel, Marco Gruteser, and the anonymous referees for their insightful and valuable feedback on this paper.

9. REFERENCES

[1] K. Chow and D. Notkin. Semi-automatic update of applications in response to library changes. In *ICSM*, pages 359–. IEEE Computer Society, 1996.

[2] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999. ISBN 0-201-48567-2.

[3] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patters*. Addison-Wesley, 1995.

[4] R. Keller and U. Hölzle. Binary component adaptation. In *ECOOP'98 - Object-Oriented Programming: 12th European Conference*, Brussels, Belgium, July 1998.

[5] J. Kerievsky. *Refactoring to Patterns*. Addison-Wesley, 2004.

[6] B. S. Lerner. A model for compound type changes encountered in schema evolution. *ACM Trans. Database Syst.*, 25(1):83–127, 2000.

[7] T. Mens and T. Tourwe. A survey of software refactoring. *IEEE Transactions on Software Engineering*, 30(2):126–139, Feb. 2004.

[8] W. F. Opdyke. *Refactoring Object-Oriented Frameworks*. PhD thesis, University of Illinois at Urbana-Champaign, 1992.

[9] http://www-plan.cs.colorado.edu/henkel/catchup/