Automatically Generating Refactorings to Support API Evolution

Jeff H. Perkins
Massachusetts Institute of Technology
Computer Science and Artificial Intelligence Laboratory
32 Vassar Street, Cambridge MA USA
jhp@csail.mit.edu

Abstract

When library APIs change, client code should change in response, in order to avoid erroneous behavior, compilation failures, or warnings. Previous research has introduced techniques for generating such client refactorings. This paper improves on the previous work by proposing a novel, lightweight technique that takes advantage of information that programmers can insert in the code rather than forcing them to use a different tool to re-express it. The key idea is to replace calls to deprecated methods by their bodies, where those bodies consist of the appropriate replacement code. This approach has several benefits. It requires no change in library development practice, since programmers already adjust method bodies and/or write example code, and there are no new tools or languages to learn. It does not require distribution of new artifacts, and a tool to apply it can be lightweight. We evaluated the applicability of our approach on a number of libraries and found it to to be applicable in more than 75% of the cases.

1. Introduction

In software, change is a constant. Between 60% and 90% of software development consists of "maintenance", or modifying existing software [12, 1, 2]. Ideally, changes to one part of a system can be isolated behind interfaces; but often, a redesign in one part of the code — particularly in widely-used libraries — forces changes elsewhere, in a cascading effect. This is not necessarily undesirable, as such changes can improve overall code organization and quality.

This paper discusses changes in libraries: how to adjust client code to such changes without inconveniencing either library writers or clients, and by doing so encouraging desirable changes in libraries. These library improvements might otherwise have been avoided because of the inconvenience they would have presented to clients. This paper makes two primary contributions. The first contribution is a technique that permits refactorings to be automatically generated for the most common types of library API changes, without any extra work on the part of the library writer. The second contribution is a methodology that requires minimal work on the part of the library writer (and no special tools or changes in development practice) but which enables additional changes and permits testing before the change is performed.

PASTE '05 Lisbon, Portugal

Libraries are changed for a variety of reasons. In some cases, the change results from errors in a previous implementation. The errors might be corrected, or they might be left in place to preserve backward compatibility, with a corrected version of the code added in parallel to the flawed one; the result is to change the library interface. In other cases, the change is a result of discovering a better way to perform a task; in this case, the old version is likely to be retained, though its use is discouraged.

When a library's interface is changed, client code that used the old interface should be updated to use the new interface instead. Typically, the library maintainer suggests specific changes to client code that makes the clients conform to the library's new conventions or intended use. These suggestions are traditionally communicated in plain text documentation. It is crucial to have a human-understandable explanation of the change and how a client should accommodate it, but the plain text format has the disadvantages that it is easy to ignore the recommendations and difficult to apply them.

If a client program is not updated as suggested by the library writer, a variety of negative consequences may result.

- The client code may behave incorrectly, if the change is not backwards compatible. This situation is undesirable, but it occurs in practice [4].
- The client code may behave incorrectly, if the original behavior was incorrect but was retained for backward compatibility, with a new more correct method added.
- The client code may fail to run and to compile, if old elements of the interface were removed.
- Compilation may issue a warning that the library uses deprecated methods. This indicates that a failure of the second type is already present, and/or a failure of the third type is imminent. Typically, library writers use this approach in advance of behavioral changes to the code.

Java supports the latter option with its `@Deprecated` annotation (and with the `@deprecated` Javadoc tag), and Eiffel supports it with its `obsolete` keyword. This is the usual mechanism for informing clients of a library change, and the compiler warnings encourage conscientious developers to update their code according to the library writer's desires.

It is desirable to automate the changes to client code that are required when library APIs change. They are tedious and error-prone to perform. Interface changes are relatively uncommon, but they affect large numbers of users. Client code maintainers may not deeply understand the library, the modifications to it, or the legacy code that uses it — and they should not have to, when tool support (as we propose in this paper) is available.

2. Goals

We propose the following three goals for a refactoring tool that updates client code in response to library API changes.

1. The library maintainer does not need to change his or her development practice. There are no new tools or languages to learn (much less to purchase or develop) and no new specifications to write.
2. There are no additional artifacts to distribute; the library writer packages exactly the code or binaries that would otherwise have been distributed.
3. The client code maintainer runs a simple, lightweight, automated tool that is capable of accommodating a variety of user interfaces, or none at all.

3. Technique: method inlining

We present a technique that automatically refactors client programs to use a new version of a library based on the established practice of marking methods as deprecated or obsolete. The idea is a simple one: replace calls to a deprecated method by the method's body. Most deprecated methods are intended to be replaced by new code with a similar purpose and equivalent or superior functionality. Library maintainers write, in the documentation, a description and often an example of substituting calls to the deprecated method by calls to the replacements. It is straightforward, and often desirable, for the library maintainer to replace the body of the deprecated method by exactly that new code, so that the deprecated method delegates to its replacement. When that coding practice is followed, then inlining the deprecated method body is exactly the desired refactoring for the client. This allows the library maintainer to define in code precisely the best replacement for the original call.

This transformation does not require access to source code for the library; it can be performed on a library distributed as bytecode (.class) files. A tool can easily read the deprecated method's bytecode body and convert it to Java code. Since the recommended replacement code is usually just one or a few method calls, reverse-engineering is easy (and Java decompilation tools are extremely effective in any event [8]). The result is likely to be comprehensible, with few or no variables to name.

Because a tool based on this technique can be simple and automated, it is easy for the client maintainer to run, and it can be integrated with a user interface or integrated development environment for displaying the changes or querying the user about them.

3.1 Class rename

Another possible library API change is to rename a class or move a class to a different package. In this case, the library developer can indicate this in the deprecated class by modifying it to extend the new class. For example consider the class `OldClass`:

```
class OldClass {
  public void m1() { ... }
  public viod m2() { ... }
  ...
}
```

In the new release, `OldClass` is replaced by `NewClass`. The new release would contain the following class definitions:

```
class NewClass {
  public void m1() { ... }
  public viod m2() { ... }
  ...
}
@Deprecated class OldClass extends NewClass {
}
```

If all of the methods are the same, the above is complete. If some methods have changed as well, they can be included in the deprecated class and delegate to their replacement as described previously.

3.2 Limitations of the technique

The method inlining approach is simple, but novel and effective, and it satisfies the three goals of section 2. However, it has important limitations. (Section 6 compares it to related work.)

The technique is applicable only to classes, methods and static final fields that are explicitly marked as deprecated; it cannot handle all possible modifications to the API of a library. However, changes to an API that result in deprecated methods with straightforward replacements are common and previous work [7] found them to be significant.

More importantly, the technique is applicable only to methods whose body can consist of calls to the replacement(s) for the method. Whereas this is a reasonable practice that makes the code smaller, easier to read, and easier to reason about, it is not universal.

The body might not call a replacement method because there is none; for example, Java's `Thread.suspend()` method is inherently unsafe, and there is no safe method that satisfies its contract. Automatic refactoring of client code in such cases is far beyond the state of the art, because code that uses such methods requires deep re-design.

A more important reason that it may be undesirable for the body to call the replacement method is that the replacement behaves differently in certain circumstances than the deprecated method. In this case, the library maintainer has indicated, by leaving the method body alone, that the recommended change is *not* a refactoring: it does not preserve behavior, or at least the library maintainer is not positive that it does.

4. Non-behavior-preserving changes

A library maintainer who is particularly concerned with backward compatibility (such as the Java JDK maintainers) may leave deprecated methods as is, even when a clearly superior replacement exists whose behavior differs only little from the deprecated version.

In this case, we propose a slight change in the programming methodology. The library maintainer changes the body of the method to select at run time between two versions: the backward compatible version and the new version.

```
@Deprecated
int old_method(Object x) {
  if (complete_backwards_compatibility) {
    // old code
  } else {
    return new_method(x);
  }
}
```

The backwards compatible version provides as much backwards compatibility as possible (given other changes to the class). The new version is the recommended replacement for the deprecated method. Essentially it is a concrete implementation of the plain text suggestion for updating client code. The refactoring tool would recognize this idiom and use the new version when inlining.

Equally importantly, library clients would have a way of testing whether the refactoring would change their behavior. The client programmers could run tests (or perform production runs) with the backwards compatibility flag turned off, then make a reasoned decision about whether to accept the refactoring. By contrast, today, clients are forced to refactor first and test later. If testing indicates that the change is undesirable, the refactoring must be undone.

Run-time overhead of the checks is negligible, and should not be a concern in deprecated methods that are not intended to be used frequently.

A refactoring tool could even permit removal of methods, something that is effectively impossible today because of desire for backward compatibility. Removed methods could be left in the code but marked with a `@Removed` annotation that is an even stronger version of `@Deprecated`. These methods could be removed from the Javadoc (or moved to the end), and the compiler could refuse to compile code that used them. But since they still appear in the `.class` file, compiled programs continue to work properly, and code that uses them can still be refactored easily by a lightweight tool.

It is also easy to imagine an annotation that indicates that uses of a particular Java method should not be refactored, even though the method is deprecated.

These mechanisms could reduce the reluctance of developers to remove methods, and our technique in general could encourage users to make code-improving refactorings, as has been previously noted [7].

5. Applicability

We examined the deprecated methods and fields in the Java 1.5 java.awt (AWT) package and the Apache Byte Code Engineering Library (BCEL) version 5.1 and categorized them as to the type of change. We found the following types of changes.

- **Rename method.** The method has been renamed but is otherwise unchanged. This includes cases where the method has been moved to a superclass.
- **Method arguments changed** The method is replaced by a new method with different arguments, but the new arguments can be determined from the original call. For example `enable()` is replaced by `setEnable(true)`
- **Method semantics changed.** The method is replaced by a similar method with different semantics. This includes bug fixes, throwing new exceptions, etc.
- **Rename static final field.** The field has been moved to a different class. The deprecated field is initialized to the new field value. For example: `public static final int CURSOR = Cursor.CURSOR`.
- **Replace constructor with factory.** A constructor has been replaced with a factory method or a static final constant.
- **Redesign required.** There is no simple replacement for the call. For example in the AWT package, the `Event` class was replaced by a number of more specific `Event` classes (`AwtEvent`, `MouseEvent`, etc). Users of methods that accept `Event` parameters require a more complex change.

Figure 1 shows the number of methods or fields in each of the categories and whether or not our technique can handle the type of change.

In the "Rename method" and "Method arguments changed" cases, the deprecated method can delegate directly to its replacement and simple inlining of that body will refactor the client code correctly. In these cases the replacement code will be a single function call. In the "Method semantics changed" case, the idiom from Section 4 would need to be used. In this case the code to be inlined may be more complex (for example, it may handle new exceptions). In the "Rename static final field" case, the initializer for the deprecated variable can be substituted directly for any use of that variable.

	AWT	BCEL	Inline works
Rename method	73	0	Yes
Method arguments changed	9	0	Yes
Method semantics changed	1	4	Yes
Rename static final field	13	0	Yes
Replace constructor with factory	0	1	No
Redesign required	26	0	No
Total deprecated methods/fields	122	5	

Figure 1: Categories of deprecated methods and fields in the java.awt package from the Java standard library and the Byte Code Engineering Library. The AWT and BCEL columns represent the number of methods or fields that fall into each category. The 'Inline works' column represents whether or not the inlining technique will work on that category

The technique does not work in the "Replace constructor with factory" case because a constructor cannot return a different object. There is a standard client refactoring (replace `new X()` with `X.factory()`), but there is no straightforward way to express it in Java. It also does not work in the "Redesign required" case because the replacement code must be client specific.

It is important to note that our technique would not work on the libraries as they are. The libraries would have to be changed as indicated above. Interestingly, in AWT, the developers call the deprecated method from the new method rather than the other way around. If these changes were made, however, our technique could handle 79% of the changes to AWT and 80% of the changes to BCEL.

6. Related work

Chow and Notkin [4] propose a methodology for changing library clients in response to library changes: "a library maintainer annotates changed functions with rules that are used to generate tools that will update the applications that use the updated libraries." The rules appear in `.h` files, so no separate files are required, and are written by hand in the language of Sorcerer [11], permitting arbitrary refactorings. The tool supports 8 different refactorings. This technique supports the second goal of Section 2 (no additional artifacts), but not the first or the third.

Chow and Notkin [4] note previous industrial projects with more limited goals (for instance, they are not user-extensible): Borland's ObjectWindows Library converter and Microsoft's migrate tool.

Recently, there has been a resurgence of interest in client conversion via refactoring, a technique for performing behavior-preserving code transformations [6, 10, 5, 9]. Borland demonstrated a "team refactoring" tool at the 2004 JavaOne conference [3]; it permits any developer to replay a refactoring that another developer had stored in a configuration management system. A team at Lund University is attempting to implement similar functionality in Eclipse (`http://www.lucas.lth.se/cm/cmeclipse.shtml`). Henkel and Diwan [7] propose a similar approach, which differs in that it stores the refactorings not in a version control system but in an XML file that can be edited by hand.

The refactoring-based approach has several disadvantages.

1. The changes are limited to those supported by the refactoring tool, rather than arbitrary ones that can be expressed by writing code.
2. The approach introduces a new language or file format to express the changes, and a separate file (whether XML or part

of a version control system) must be shipped to clients. Compared to including the the suggested change in the source file, these choices make it both more difficult for library developers to produce/review their changes and for users to evaluate the changes.
3. The approach forces library developers to use a special tool to record refactorings, rather than using their usual development environment to edit code, a process with which they are already familiar and facile. We speculate that this technique requires developers to do their work twice: once in the usual way, then a second time to be recorded. Duplication of work is both annoying and error-prone, especially when performed in an unfamiliar environment.
4. The result of making the changes can only be tested *after* performing the refactoring, making it difficult to determine whether the change is appropriate and unnecessarily difficult to back out of it if necessary.

The refactoring-based approach satisfies the third goal of Section 2 in part, but neither of the first two goals.

Our work is inspired by that of Henkel and Diwan, and it aims to solve the same problem, but without its limitations. Henkel and Diwan's tool fully supports 3 different library refactorings: renaming a class, moving a class to a different package, and changing a method signature. Each of these is supported by our approach as well. Thus, a tool built on our (theoretically less general) technique would be able to perform as many refactorings as theirs, in addition to other benefits of our approach.

7. Conclusion

We have proposed a technique for refactoring client code in response to API changes in libraries. While inspired by previous work addressing the same problem, our technique is simpler and avoids the limitations of earlier approaches. The key idea is to refactor client code via method inlining in order to eliminate calls to deprecated or obsolete methods. The technique satisfies three key goals. First, no change in development practice (including new tools, languages, or specifications) is required. Second, there are no additional artifacts to distribute or understand. Third, and partly as a result of the first two goals, the system is simple and lightweight to implement and to understand. Despite its simplicity, it is capable of performing all the refactorings of some previous work, and the most important refactorings of other previous work.

We have additionally proposed a programming methodology that retains both old and new versions of a method in the codebase. This methodology has two key benefits. First, it permits testing of compatibility with new versions of the code before refactoring, via easy selection of a version of the code on a class-by-class basis. Second, it enables the library programmer to specify (via code) the suggested replacement for methods for which there is no completely back compatible replacement, and enables a user to apply them after testing or analysis has increased confidence in their safety.

Our proposals are novel but very simple, and that simplicity is the reason they are attractive. Using a straightforward and easy-to-understand code-based approach avoids the complexities of other tools and languages. It is familiar to software engineers and can be manipulated using arbitrary programming tools and environments. Our approach is applicable to most or all of the important situations that arise when refactoring library clients.

More generally, this research is part of a broader program that advocates re-use of existing artifacts when possible. The most important such artifacts are code and tests. Programmers invest great effort in constructing these artifacts, and they are the final authority about programmer intention and program behavior. As a result, programs and tests contain a wealth of implicit information, and we advocate mining that information for a variety of tasks, from verification to debugging. This research demonstrates that refactoring in response to library API changes can equally take advantage of existing code artifacts rather than relying on new ones that require additional effort to construct and are potentially inconsistent with the code.

Acknowledgments

Danny Dig and Adam Kiezun provided very useful comments and feedback on the paper. Comments from the anonymous referees also helped us to improve the presentation of this paper.

8. References

[1] R. Balzer, J. T. E. Cheatham, and C. Green. Software technology in the 1990's: Using a new paradigm. *Computer*, 16(11):39–45, Nov. 1983.
[2] B. W. Boehm. Industrial software metrics top 10 list. *IEEE Software*, 4(5):84–85, Sept. 1987.
[3] Borland. Making development a team sport. demo at JavaOne, June 29, 2004.
[4] K. Chow and D. Notkin. Semi-automatic update of applications in response to library changes. In *ICSM*, pages 259–368, Nov. 1996.
[5] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 2000.
[6] W. G. Griswold. Program restructuring to aid software maintenance. Technical Report 91-08-04, U. Wash. Dept. of Comp. Sci. & Eng., Seattle, WA, USA, Aug. 1991. PhD dissertation.
[7] J. Henkel and A. Diwan. Catchup! capturing and replaying refactorings to support API evolution. In *ICSE*, May 2005.
[8] A. Kalinovsky. *Covert Java: Techniques for Decompiling, Patching and Reverse Engineering*. Sams, Indianapolis, Indiana, 2004.
[9] T. Mens and T. Tourwé. A survey of software refactoring. *IEEE TSE*, 30(2):126–139, Feb. 2004.
[10] W. F. Opdyke. Refactoring: A program restructuring aid in designing object-oriented applications frameworks. Technical Report 1759, University of Illinois at Urbana-Champaign, Dept. of Computer Science, 1992. PhD dissertation.
[11] T. J. Parr. An overview of SORCERER: A simple tree-parser generator. In *Compiler Construction '94*, Apr. 1994.
[12] L. H. Putnam. A general empirical solution to the macro software sizing and estimating problem. *IEEE TSE*, 4(4):345–361, July 1978.