Automated Detection of API Refactorings in Libraries

Kunal Taneja
Department of Computer Science
North Carolina State University
ktaneja@ncsu.edu

Danny Dig
Department of Computer Science
University of Illinois
dig@cs.uiuc.edu

Tao Xie
Computer Science Department
North Carolina State University
xie@csc.ncsu.edu

ABSTRACT

Software developers often do not build software from scratch but reuse software libraries. In theory, the APIs of a library should be stable, but in practice they do change and thus require changes in software that reuses the library. Our previous study of five reusable components shows that more than 80% of these API changes are caused by refactorings. If these refactorings could be automatically detected, they could be used to automatically upgrade applications.

In this paper, we present a technique and its supporting tool, `RefacLib`, to automatically detect refactorings in libraries. `RefacLib` uses syntactic analysis in the first phase to quickly detect refactoring candidates across two versions of a library. In the second phase, `RefacLib` uses various heuristics to refine the results. We used `RefacLib` to detect refactorings in five open source libraries and frameworks. The experiments show that `RefacLib` can process realistic code bases and detects refactorings with practical accuracy.

Categories and Subject Descriptors: D.2.13 [Software Engineering]: Reusable Software—*Reusable libraries*;

General Terms: Management, Design.

Keywords: Refactoring, Code reuse, Libraries, Software maintainance

1. INTRODUCTION

Refactoring is a disciplined technique for improving the internal structure of a program while preserving its observable behavior. The problem with refactorings is that they can change an Application Programming Interface (API) and require software that uses the old API to be updated to use the new API. Conventionally, such updating is done manually, which is error-prone, tedious, and disruptive to the development process. Thus, such updating makes maintaining software expensive. This problem is exacerbated when refactorings change the APIs of reusable software components (e.g., libraries and frameworks): our previous study [5] of five popular components shows that refactorings cause more than 80% of API changes that were not backwards-compatible.

In this paper, we present a novel technique to automatically infer refactorings that happened between two versions of a library. One of the key challenges is the size of real-life libraries (hundreds of KLOC). For example, for log4j, a medium-size library, if a tool looks at all pairs of methods across two versions, it would need to analyze 3.7 million pairs of methods. To reduce the search space, previous approaches [1, 3, 6, 11, 12] assume that older program entities in one version are removed and replaced with refactored entities in the subsequent version. Thus, all these approaches start by analyzing only pairs of program elements that disappear from the old version and program elements that appear in the newer version. While this assumption is true for software that is built and used in-house and does not need to be backwards-compatible, reusable software libraries follow a long *deprecate-replace-remove* cycle.

`RefactoringCrawler`, our previous tool [4], addresses these shortcomings. However, experiments on real components revealed new shortcomings. To achieve high accuracy levels (over 85%) and scalability (hundreds of KLOC), `RefactoringCrawler` combines a fast syntactic analysis with a precise semantic analysis. The syntactic analysis quickly identifies a set of program elements suspected of refactoring. For these elements, the semantic analysis builds *reference graphs*. In case of methods, these reference graphs contain all calls to the methods under analysis. If two methods are called from the same places across the two versions, have similar method bodies, and yet have different names, `RefactoringCrawler` infers that the methods are a rename of each other. However, it is not always possible to discriminate methods based on the similarity of their method calls due to the fundamental difference between the nature of frameworks (targeted by `RefactoringCrawler`) and libraries. The API methods that a framework provides are called from within the framework. In contrast, libraries offer API methods that are called only from outside by the application that reuses them. Moreover, some APIs might not be referenced internally even for frameworks. `RefactoringCrawler` cannot deal with cases of lack of API references.

To overcome the lack of internal references in the case of libraries, we developed a new suite of analyses and a new tool, `RefacLib`. `RefacLib` inherits all the good properties of `RefactoringCrawler` while it improves on `RefactoringCrawler`'s weaknesses. `RefacLib` uses the same fast and innovative syntactic analysis based on Shingles-encoding [2], a technique used in Information Retrieval to detect similarity in large bodies of text. The syntactic analysis produces pairs of program elements (across the two versions) that have similar bodies (e.g., methods with similar bodies). These pairs are fed into a suite of heuristic-based analyses that do not depend on references, and thus are able to analyze libraries. We evaluated `RefacLib` on three frameworks and two libraries. While the accuracy detection is comparable with that of `RefactoringCrawler` in the case of frameworks, `RefacLib`

ASE'07, November 5–9, 2007, Atlanta, Georgia, USA.

finds many more refactorings in the case of libraries. Based on the empirical findings, in future work, we plan to pair the two tools so that the weaknesses of one are made irrelevant by the strengths of the other.

This paper makes the following main contributions:

- **Design**. We have designed a set of heuristic-based analyses to detect refactorings despite the noise introduced by the backwards-compatible evolution of software libraries.
- **Implementation**. We have implemented an efficient tool, `RefacLib`, to detect refactorings with practical accuracy in realistic software components.
- **Evaluation.** We have used `RefacLib` to find several refactorings in five real-world components. We compared `RefacLib` with the previous state-of-the-art tool to detect refactorings and found out that our tool is comparable in most cases and better in others.

2. APPROACH

Our approach consists of three phases. The *syntactic analysis* phase takes as input two versions of the component, v1 and v2, and produces pairs of syntactically-matching entities. The *classification* phase classifies these pairs as candidates for various kinds of refactorings based on some syntactic checks. We currently support seven refactorings: `ChangeMethodSignature` (CMS), `RenameClass`, `PushDownMethod` (PDM), `RenameP ackage` (RP), `RenameMethod` (RM), `PullUpMethod` (PUM), and `MoveMethod` (MM) (these were among the most frequently performed refactorings found in a previous study [5]). Finally, in the *heuristic-based analysis* phase, the algorithm computes a composite score for each pair based on various heuristics. We define a set of heuristics for each type of refactoring. For each candidate pair, the algorithm assigns a composite score that is a weighted sum of scores for all heuristics defined for that refactoring type. `RefacLib` reports as refactorings only the pairs with a score above a threshold.

2.1 Overview

Our syntactic analysis phase reuses the one developed in our previous tool `RefactoringCrawler` [4]. The syntactic analysis phase returns a set of pairs of entities that are similar textually. These pairs of similar entities are classified by the *classification* phase as candidates of one of the seven refactoring types that we support. The classification is done using some syntactic checks [4].

These pairs of similar entities are suspected candidates of refactorings, but may contain many pairs that are not actual refactorings and need to be filtered out. This filtering process is where `RefacLib` differs dramatically from our previous tool, `RefactoringCrawler`. For candidates of each refactoring type, our new heuristic-based analysis (our previous approach [4] used semantic analysis instead) selects pairs that are real refactorings. In particular, the analysis gathers facts from the source code and Javadoc comments, and computes similarity measures to assign an overall score that reflects the likelihood of a candidate to be a refactoring. The facts and similarity measures vary for different types of refactorings. Section 2.2 describes in detail the various types of heuristics that we developed.

The process of classification and heuristic-based analysis iterates visiting the set of all pairs, and taking into account already detected refactorings. The process continues until a fixed point is reached. This process ensures that `RefacLib` detects pairs of entities that underwent multiple refactorings.

Another key ingredient to detect coupled refactorings (cases when multiple refactorings happened to related entities) is a predefined order in which `RefacLib` searches for different types of refactorings. `RefacLib` first searches for renamings in packages, then in classes, and finally in methods (a top-down approach). When searching for moved elements, `RefacLib` uses a bottom-up approach: it first searches for moved methods, then moved classes, etc. The intuition is that some types of refactorings can be detected without knowledge about other types of refactorings.

2.2 Heuristic-Based Analysis

The heuristic-based analysis applies different heuristics for different refactoring types. The list of heuristics that we developed for each type of refactorings is shown in Columns 2 and 6 of Table 1. `RefacLib` ranks the candidate pairs based on various heuristics. The overall score of a pair is a weighted sum of all the scores assigned to each heuristic used for the pair.

We next describe the various heuristics that we developed in ranking the candidate pairs. We use a subset of these heuristics to detect refactorings of a particular type. For each entity pair $< e1, e2 >$ passed by the syntactic analysis phase, we use the following heuristics.

Name Similarity (NS). We use the NS heuristic when the simple names of entities *e1* and *e2* are different. Software developers rename entities to better convey their purpose or to correct a spelling mistake. In some cases, the name of a method is changed to one of its synonyms that better reflects its purpose.

As a common convention in Java, names representing methods start with a verb followed by some parts of speech (noun, verb, etc.) and written as starting with a lower case such as `getActionMappingName`. The names of classes are nouns starting with upper case such as `UserMenuAction`. These naming conventions are recommended by Sun [7]. To determine the name similarity of *e1* and *e2*, we consider only their simple names (suffix). We decompose the names of the two entities into subparts, and match the subparts. For example, if *e1* and *e2* are *performUpdates* and *executeNewUpdates*, respectively, they are decomposed into the sets of subparts {*perform, Updates*} (*s1*) and $\{execute, New, Updates\}$ (*s2*), respectively. Each of the subparts in *s1* is matched with all the subparts in *s2*. If a subpart in *s1* does not match with any of the subparts in *s2*, `RefacLib` automatically retrieves all the synonyms of the subpart using the Wordnet library for Java [8], and match them to the subparts in *s2*. If none of the synonyms matches with the subparts in *s2*, `RefacLib` compares the two subparts syntactically and gives the pair a score based on their syntactic similarity. The final score for this heuristic depends on the fraction of matching parts of the smaller of the two sets of subparts, and the differences between the sizes of the two sets.

Deprecated Entities (DE). Source code entities in an evolving software component follow the deprecated-replace-delete life cycle. Initially when an entity is refactored, it is marked deprecated. In a later version of the software component, the entity is replaced by the new refactored entity; however, the obsolete entity still exists but is deprecated. Given a pair of entities, $< e1, e2 >$, if entity *e1* exists in Version *v2* but is deprecated, it is likely that the entity is refactored. On the other hand, if entity *e2* is deprecated in any of *v1* or *v2*, the refactoring pair containing *v1* and *v2* cannot be a refactoring, since *e2* is an obsolete entity. If the entity *e1* exists in Version *v2* as well but is not deprecated, it is unlikely that the pair $< e1, e2 >$ is a refactoring. `RefacLib` assigns a score to a candidate pair based on the above observations.

Deprecated Containing Class (DC). We use the DC heuristic when entities *e1* and *e2* are methods. If the class containing *e1* is deprecated either in *v1* or *v2*, it is likely that the class will either be moved/deleted or the methods inside the class will be moved to some other class. Similarly if the class containing *e2* is deprecated in Version *v2*, the method *e1* is unlikely to be a refactored version of *e1* unless the classes of *e1* and *e2* are the same. `RefacLib` assign a score to the pair $< e1, e2 >$ based on the above observations.

Method Size (MS). `RefacLib` uses the MS heuristic while analyzing pairs containing methods. If the methods in the pairs are too small in size, it is likely that their bodies will be similar and they will pass the syntactic analysis phase. For example, the bodies of getter and setter methods may be quite similar (in many cases exactly the same). So there can be many such method pairs that pass the syntactic analysis phase, but are not real refactorings. However, such methods may have different Javadoc comments. `RefacLib` computes the shingles of their Javadoc comment body and assigns a score that reflects the similarity of shingles.

Signature Change Pattern (SCP). We use the SCP heuristic while detecting `ChangeMethodSignature` refactorings. Kim et al. [9] found that the three most common patterns of changing signatures are (in decreasing popularity): addition of one parameter, changing the type of a parameter to a more complex type, and deletion of a parameter. `RefacLib` ranks the candidates by a signature-pattern score based on the above observations. If *e1* and *e2* have a very different signature, it is highly unlikely that the pair $< e1, e2 >$ is a refactoring; as a result, `RefacLib` assigns it a low score.

Deprecated type of a Method Parameter (DMP). We use the DMP heuristic while detecting the `ChangeMethodSignature` refactoring. If the type of a parameter in method *e1* is deprecated in either Version *v1* or *v2*, the parameter is likely to be replaced by its refactored version.

Class Size Reduction (CSR). We use the CSR heuristic for detecting `MoveMethod`, `PullUpMethod`, and `PushDownMethod` refactorings. The most common intent of these refactorings is to reduce the responsibilities of a large class. `RefacLib` considers the size of the class (*C1*) containing *e1* in Versions *v1* and *v2* (If Class *C1* exists in Version *v2*). When the size of the class *C1* is reduced from *v1* to *v2*, it is more likely that *e1* is moved to *e2* in Version *v2*. Similarly if size of the class containing *e2* increases from Version *v1* to *v2* it is more likely that the candidate is a real refactoring. Additionally, if the size of class *C1* is too small, it is unlikely that a method will be moved from it. The `PullUpMethod` refactoring can also be used to remove duplicated code from a software system. For example, a method can be pulled up from various classes to a common super class. `RefacLib` uses this fact to check if the method *e1* is also removed (or deprecated) from some other classes that are a subclass of *C2* containing *e2*.

For each refactoring type, we use a subset of the preceding heuristics. Columns 2 and 6 in Table 1 show the set of heuristics that we use for each refactoring type. The heuristic-based analysis computes, for each refactoring candidate pair, a score based on all the heuristics applicable for that refactoring. `RefacLib` assigns the pair a composite score that is a weighted sum of all the scores of individual heuristics. `RefacLib` assigns different weights for different refactorings since there can be some heuristics that are more important than others. The weights that we use for each heuristic are shown in Column 3 and 7 of Table 1. Once we have the scores of all candidates, `RefacLib` reports the pairs with a score above a threshold as detected refactorings.

3. EVALUATION

This section evaluates the accuracy of `RefacLib` in comparison with that of `RefactoringCrawler` [4]. For comparison we use the same three frameworks on which `RefactoringCrawler` was evaluated; we add two new case studies of libraries, and evaluate both tools on all five case studies. We next describe the objectives,

Refac	Heuristics	Wts		Refac	Heuristics	Wts
CMS(m_1, m_2)	DE	0.2		RM(m_1, m_2)	NS	0.3
	DC	0.2			DE	0.3
	SCP	0.35			MS	0.1
	DMP	0.25			DC	0.3
RC(C_1, C_2)	NS	0.5		PUM(m_1, m_2)	DE	0.3
	DE	0.5			DC	0.3
					MS	0.1
					CSR	0.3
PDM(m_1, m_2)	DE	0.3		MM(m_1, m_2)	DE	0.3
	DC	0.3			DC	0.3
	MS	0.1			MS	0.1
	CSR	0.3			CSR	0.3
RP(p_1, p_2)	NS	1.0				

Table 1: Heuristics used for different refactorings along with their weights.

subjects, and the process of evaluation, and finally present and discuss the results.

Objectives. We investigate the following questions:

- Is our tool more accurate than existing tools for detecting refactorings in libraries?
- Is our tool comparable for detecting refactorings in frameworks?
- Does our tool scale to real-world software components?

To answer the first two questions, we compare the accuracy of `RefacLib` with that of `RefactoringCrawler` using *precision* and *recall*. To answer the third question, we sample our subjects from real-world software components. The size of the subjects varies from 30K to 352K lines of code.

Our initial goal was to evaluate the accuracy of `RefacLib` in comparison with that of previously developed tools [1, 3, 6, 11, 12]. However, none of these is available for public download. Moreover, some of them work for different programming languages, while those evaluated for Java did not make public the exact refactorings being found. Thus, we could compare only `RefacLib` and `RefactoringCrawler`.

Experimental Setup. We evaluated the accuracy of `RefacLib` on five real-world, open-source software components (two libraries, *Log4j* and *Lucene*, and three frameworks, (*EclipseUI*, *Struts*, and *JHotDraw*). For measuring the accuracy of refactoring detection, one needs to find out the false positives and false negatives. False positives are easily found by inspecting the source code of the program elements returned by `RefacLib`. However, to find out the false negatives (refactorings that `RefacLib` did not find), one needs to know a-priori the refactorings that happened in those components. The process of manually finding the real refactorings is laborious and requires weeks of careful, manual inspection. Fortunately, we reuse the fruits of our labor from a previous study [5], which combined analysis of release documents, manual code inspection, and interviews with the component developers. Therefore, we had a solid base of manually found refactorings to compare against the ones found by `RefacLib`.

Measurements. We measure the accuracy of `RefacLib` using two standard metrics from the Information Retrieval field: *Recall* and *Precision*. It is hard to achieve 100% precision and recall. For practical purposes, the recall value is more important than the precision, since the false positives (if not too many) can be removed by manual inspection. However, it is almost impossible to find out the false negatives by inspecting thousands of methods inside the whole software component. In addition, we use another metric called *F-Measure* (first introduced by Rijsbergen [10]). *F-Measure* combines recall and precision into a single efficiency measure. In particular, *F-Measure* is a harmonic mean of recall and precision.

Results. Table 2 shows the results obtained by `RefactoringCrawler`(denoted as RC) and `RefacLib`(denoted as RL) while detecting refactorings of the five chosen components. We observe that, for both library subjects, `RefacLib` performs generally better than `RefactoringCrawler` in terms of recall. For the frameworks, the recall of `RefacLib` is comparable to the recall of `Refac`

		RM	*RC*	*RP*	*MM*	*PUM*	*PDM*	*CMS*	*Precision*	*Recall*	*F-Measure*
Log4j 1.2.1 - 1.3alpha6	RC	0,0,5	0,0,13	0,0,0	14,0,0	1,0,8	0,0,0	30,0,3	100%	60.81%	75.63%
	RL	5,1,0	11,0,2	0,0,0	3,0,11	7,0,2	0,0,0	33,0,0	98.33%	79.73%	87.79%
Lucene 1.4.2 - 1.9	RC	0,0,11	0,0,1	0,0,0	2,0,33	0,0,0	0,0,0	32,0,10	100%	38.2%	55.28%
	RL	10,0,1	0,0,1	0,0,0	33,0,2	0,0,0	0,0,0	33,7,9	91.57%	85.39%	88.37%
EclipseUI 2.1.3 - 3.0	RC	2,1,0	0,0,0	0,0,0	8,2,4	11,0,0	0,0,0	6,0,0	90%	86%	87.95%
	RL	2,10,0	0,0,0	0,0,0	7,0,5	7,7,4	0,0,0	6,0,0	56.41%	73.33%	63.77%
Struts 1.2.1 - 1.2.4	RC	20,0,1	1,0,1	0,0,0	20,0,7	1,0,0	0,0,0	24,0,1	100%	86%	92.47%
	RL	20,1,1	1,0,1	0,0,0	25,4,2	1,1,0	0,0,0	23,0,2	92.1%	93.33%	92.71%
JHotDraw 5.2 - 5.3	RC	5,0,0	0,0,0	0,0,0	0,0,0	0,0,0	0,0,0	19,0,0	100%	100%	100%
	RL	5,1,0	0,0,0	0,0,0	0,1,0	0,0,0	0,0,0	19,0,0	92.31%	100%	96%

Table 2: Triplets of (GoodResults, FalsePositives, FalseNegatives) found by *RefacLib* (RL) and *RefactoringCrawler* (RC).

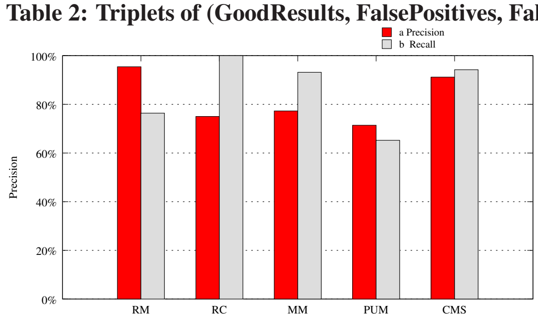

Figure 1: Recall and Precision value for *RefacLib* for detecting individual refactorings, for the five subjects we chose

`toringCrawler`. In terms of the precision value, `RefactoringCrawler` performs better for all the five subjects, but the precision of `RefacLib` is above 90% for four of the five components. Figure 1 shows the precision and recall of `RefacLib` for five refactoring types. We do not list the `PushDownMethod` and `RenamePackage` refactorings in the figure as no refactoring instance of these two types were detected in the five subjects. From the figure, we observe that the recall for `RenameMethod` and `ChangeMethodSignature` refactorings is above 90%, while the recall for the other three refactorings is between 70-80%. The precision of `RefacLib` is above 90% for `RenameClass`, `MoveMethod` and `ChangeMethodSignature`.

In summary, the evaluation answers the questions that we listed in Section 3. First, `RefacLib` performs better than `RefactoringCrawler` when detecting refactorings in libraries. Second, accuracy of `RefacLib` is comparable to that of `Refac toringCrawler` while detecting refactorings for the three framework subjects. Finally, the subjects that we chose were all real-world open-source components with up to 352K lines of code, and thus `RefacLib` is scalable to real-world software components.

4. RELATED WORK

Demeyer et al. [3] use change metrics to detect refactorings that serve as markers for the reverse engineer, whereas we combine heuristics with other syntactic analyses. Their precision rates are in the range of 20% while they do not report the recall rates. Rysselberghe and Demeyer [11] use a clone finding tool (Duploc) to detect methods that were moved across the classes; our Shingles-based syntactic analysis is a different clone finding technique.

Godfrey and Zou [6] implemented a tool to detect refactorings in procedural code. They employ the origin analysis along with a more expensive analysis on call graphs to detect and classify these changes. Kim et al. [9] propose an algorithm based on heuristics to detect rename-method refactorings between two versions of software. They use eight similarity factors to detect these refactorings. Like our approach, they also take the weighted mean of the eight factors.

Weissgerber and Diehl [12] propose an algorithm to detect refactorings by using the information from code repositories and later use code-clone detection to refine the results. Xing and Stroulia [13] detect refactorings at the design level from UML diagrams using the structural changes between the two versions of the diagrams.

All previous approaches assume that obsolete entities disappear in one version and new entities appear in another version. While this assumption might be true for software built and reused in-house, open-source libraries follow a deprecate-replace-remove life cycle: obsolete entities co-exist with their refactored counterparts. In addition, `RefacLib` works even when multiple refactorings change the same program entities.

5. CONCLUSIONS

Software systems are often built by reusing software libraries, whose APIs may undergo changes over time; a high percentage of these API changes are caused by refactorings [5]. When such changes occur, the software systems also need to be upgraded, requiring substantial maintenance efforts. To reduce the efforts, we have developed a tool, `RefacLib`, to automatically detect refactorings in library APIs; these detected refactorings can be used to automatically upgrade the software systems. `RefacLib` uses syntactic analysis to quickly detect refactoring candidates across two versions of a library and uses various heuristics to refine the results. `RefacLib` improves over `RefactoringCrawler`, our previous tool [4], by providing effective support for library APIs whose internal references are lacking. We used `RefacLib` to detect refactorings in five open source libraries and frameworks. The experiments show that `RefacLib` can process realistic code bases and detects refactorings with practical accuracy.

6. REFERENCES

[1] G. Antoniol, M. D. Penta, and E. Merlo. An automatic approach to identify class evolution discontinuities. In *Proc. IWPSE'04*, pages 31–40.

[2] A. Broder. On resemblance and containment of documents. *in Proc. of SEQUENCES*, 1997.

[3] S. Demeyer, S. Ducasse, and O. Nierstrasz. Finding refactorings via change metrics. In *Proc. OOPSLA'00*, pages 166–177.

[4] D. Dig, C. Comertoglu, D. Marinov, and R. Johnson. Automatic detection of refactorings in evolving components. In *Proc. ECOOP'06*, pages 404–428.

[5] D. Dig and R. Johnson. How do APIs evolve? a story of refactoring. *J. Softw. Maint. Evol.*, 18(2):83–107, 2006.

[6] M. W. Godfrey and L. Zou. Using origin analysis to detect merging and splitting of source code entities. *TSE*, 31(2):166–181, 2005.

[7] Code Conventions for the Java Programming Language. http://java.sun.com/docs/codeconv/html/CodeConvTOC.doc.html, Apr. 1999.

[8] Java WordNet Library. http://sourceforge.net/projects/jwordnet.

[9] S. Kim, K. Pan, and J. E. James Whitehead. When functions change their names: Automatic detection of origin relationships. In *Proc. WCRE'05*, pages 143–152.

[10] C. J. V. Rijsbergen. *Information Retrieval*. Butterworth-Heinemann, Newton, MA, USA, 1979.

[11] F. V. Rysselberghe and S. Demeyer. Reconstruction of successful software evolution using clone detection. In *Proc. IWPSE'03*, pages 126–130.

[12] P. Weissgerber and S. Diehl. Identifying refactorings from source-code changes. In *Proc. ASE'06*, pages 231–240.

[13] Z. Xing and E. Stroulia. Refactoring detection based on umldiff change-facts queries. In *Proc. WCRE'06*, 0:263–274.