Automatic Detection of Refactorings for Libraries and Frameworks

Danny Dig, Can Comertoglu, Darko Marinov, Ralph Johnson
Department of Computer Science
University of Illinois at Urbana-Champaign
201 N. Goodwin Ave., Urbana, IL 61801, USA
{dig, comertoglu, marinov, johnson}@cs.uiuc.edu

Abstract

Current work on automatic detection of refactorings typically assumes a closed-world, i.e., codebases used only in-house. Changes in closed-world happen abruptly, and refactorings are easier to spot. Libraries and frameworks are developed to be reused outside the organization and therefore the changes do not happen overnight but follow a long deprecate-replace-remove cycle. Obsolete entities will coexist with their newer counterparts until they are no longer supported. This lifecycle introduces enough noise that current tools cannot accurately detect the refactorings. We propose a methodology and a tool that fits with an open-world paradigm. We use a Shingles algorithm to detect clones of the same entity and then refine the results with semantic analysis. Preliminary evaluation shows that our method scales for real-life libraries and can accurately detect when entities are renamed or moved.

1. Introduction

Part of maintaining a software system is updating it to use the latest version of its components. Developers like to reuse software components because it lets them build a system more quickly, but then the system depends on the components that they reused. Ideally, the interface to a component never changes. In practice, new versions of software components often change their interfaces and so require systems that use the components to be changed before the new versions can be used. Our goal is to reduce the burden of reuse on maintenance. This requires either reducing the amount of change or reducing the cost of adapting to change.

An important kind of change to object-oriented software is a refactoring. Refactorings [FBB+99] are program transformations that change the structure of a program but not its behavior. Refactorings include changing the names of classes and methods, moving methods and variables from one class to another, and splitting methods or classes. In [DJ05] we analyze quantitatively and qualitatively the API changes that break compatibility with older clients. We learned out that for three commonly used frameworks and one library, more than 80% of API changes were refactorings. Moving and renaming entities represent a large part of these refactorings.

Current work [APM04, DDN00, GZ05, RD03] detects refactorings automatically under a closed-world assumption. (codebases used only in-house). In this world changes happen abruptly and refactorings are easier to spot. Libraries and frameworks are developed to be reused outside the organization and therefore the changes don't happen overnight but follow a long deprecate-replace-remove cycle. Obsolete entities will coexist with their newer counterparts until they are no longer supported. This lifecycle introduces enough noise that current tools cannot accurately detect the refactorings.

Consider for instance a change such as renaming class `Category` to class `Logger` in log4j (`http://logging.apache.org/log4j/docs`). In order to maintain compatibility with old clients, class `Logger` (the new name) inherits from class `Category`. The constructor of `Category` became protected so that users cant create categories directly but invoke instead the creational method `getInstance()`. This method returns instances of the new class `Logger`. Any method in `Category` that returned on object of type `Category` became deprecated. Clients should replace all the references to `Category` with references to `Logger`. The two classes still coexist, but `Category` will be deleted eventually.

This paper describes a methodology and a tool to automatically detect the refactorings that took place in between two versions of a component. Our target are libraries and frameworks that strive to maintain backward compatibility. For this reason, our strategy is based on clone detection. Once we find the clones among multiple versions, we analyze the relationship between them.

The size of real-life libraries and frameworks usually

varies from a few KLOC to millions of LOC. To analyze the larger applications, we first employ some cheap techniques to detect potential candidates for restructuring. We use the Shingles encoding [Bro97] (a technique from Information Retrieval) to detect code that is similar between two versions. We further refine and classify the candidates using more expensive and powerful techniques like Call Graphs and AST queries. We built RefactoringCrawler, a research prototype to detect refactorings in Java programs. Although the implementation analyzes Java source files only, the ideas presented in this paper can be easily applied to other programming languages. RefactoringCrawler is a plugin for the Eclipse development environment that currently detects renamings of methods, classes and packages as well as method moves. Preliminary evaluation on code bases ranging from 62 KLOC to 100 KLOC shows that RefactoringCrawler scales for real-life applications.

2. Methodology

This section describes the overall process that we use to detect refactorings. Section 3 and section 4 give more details about the specifics of each technique.

There are two main challenges to automatic detection of refactorings that happened in reusable components. The first is the size of the code to be analyzed. Expensive semantic analysis (like complete AST or Call Graph comparisons) can detect structural evolution but does not scale up to the size of real-life applications containing millions of nodes. Using only syntactic analysis might be cheaper, but it is very fallible to renamings of object entities.

We use instead a hybrid of syntactic and semantic analysis. We employ syntactic analysis on the bulk of the code to create a list of potential candidates. We only have to employ semantic analysis on these candidates.

The second challenge is the noise introduced because of preserving backward compatibility.

Our methodology is depicted in figure 1. In step 1, we parse the source files into a lightweight graph that is equivalent to the lightweight AST. We call it “lightweight” because the parsing stops at the declaration of the methods and fields in classes. Our tool then employs a cheap syntactic analysis based on the Shingle encodings of the method bodies (step 2). Shingles are fingerprints with the property that if a document changes by a small amount, its shingles also change by a small amount, therefore fragments that are similar can be detected. This technique is less susceptible to renamings than the traditional string matching techniques. The product of this step is pairs of entities that have similar Shingles encodings among two versions of the library, with separate pairs for methods, classes and packages. We prune those elements that have exactly the same fully qualified name in both versions.

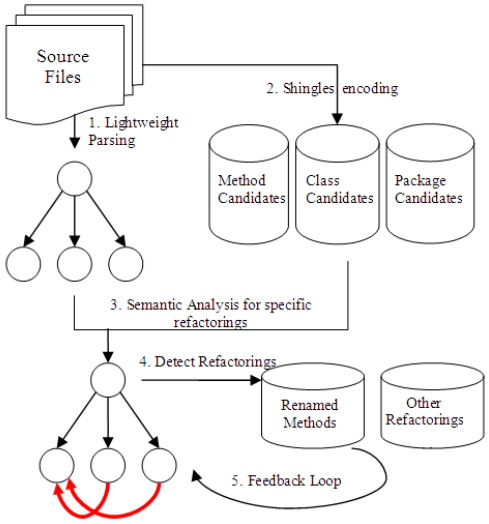

Figure 1. Methodology

Now (step 3) we do semantic analysis of only the remaining candidates. This set is considerably smaller than the the set of nodes in the lightweight graph. We embellish the original graph with the call graphs for the methods, classes and package candidates. Strategies for detecting specific refactorings are applied (step 4) over the resulting graph. Each detection strategy provides feedback (step 5) to the embellished graph that subsequent refactorings use during further analysis.

This feedback (step 5) is invaluable when there are multiple refactorings that happened to the same entity. (This step is not implemented in the current prototype of the tool.) Consider this real scenario from log4j library. Class `Category` was renamed to class `Logger` and `getRoot()` in class `Category` was renamed to `getRootLogger()` in class `Logger`. After detecting that `Category` is renamed to `Logger`, semantical information is added in the node of class `Logger`, denoting that its old name was `Category`. When we look for renamed methods we only compare methods in the same class. Now because `Logger` is equivalent (rename) to `Category`, the strategy for detecting renamed methods will correctly spot `getRootLogger()` as being a rename of `getRoot()`, even though the two methods are in different classes.

3. Shingles Encoding

To identify possible candidates for refactorings, RefactoringCrawler first determines sets of *similar* methods, classes, and packages. RefactoringCrawler uses the Shingles algorithm [Bro97] to compute a hash value for each method and determines two methods to be similar iff they have similar hash values. Unlike the traditional

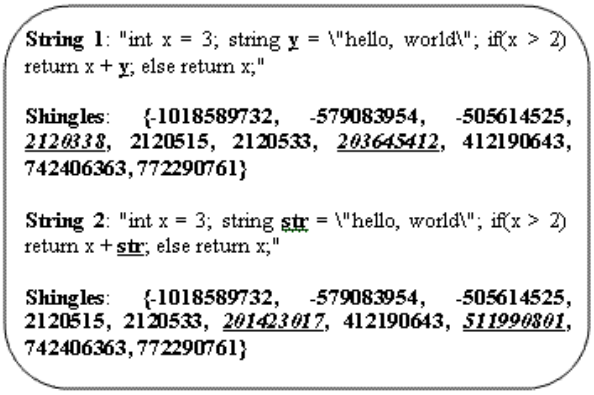

Figure 2. Shingles Encoding

hashing functions that map even the small change of the input to a completely different hash value, the Shingles algorithm maps small perturbations of the input to small changes in the hash values.

The Shingles algorithm takes as input a sequence of tokens and uses sliding windows to compute the hash value that is a bag of integers called shingles. RefactoringCrawler applies it to a method by treating the method body as a sequence of tokens. The algorithm takes two parameters W, the length of the sliding window, and S, the maximum size of the resulting bag. Given a sequence of tokens, the algorithm computes bag of shingles by finding all subsequences of length W, computing the shingle for each subsequence, and selecting the S minimum shingles. We use Rabin's Hash Function [Rab81] to compute the shingles values. If there are less than S subsequences, then the bag consists of less than S integers. The parameter S acts as the upper bound for the space needed to represent shingles: a large value makes calculations expensive, and a small value makes it hard to distinguish strings.

Figure 2 shows the result of calculating the shingles for two method bodies where $W = 2$ and $S = 10$; these values seem to work well for Java programs. The differences in the bodies and the shingle values are underlined. Notice that the small changes in the tokens produce only small changes in the shingle representation, enabling RefactoringCrawler to find the similarities between methods.

RefactoringCrawler uses the shingles for methods to compute shingles for classes and packages. The shingles of a class are computed by taking the union of the shingles of the methods and selecting the minimum S_{class} values. Similarly, the shingles of a package are computed by taking the union of the shingles of the classes and selecting the minimum $S_{package}$ values. This way, RefactoringCrawler efficiently computates shingles values and avoids recomputations [RILD04].

RefactoringCrawler uses the shingles to find candidates for the specified refactorings. More specifically, let B_1 and B_2 be bags of shingles for a pair of methods, classes, or packages. RefactoringCrawler computes similarity as the average overlap of these two bags

$$\frac{\frac{|B_1 \cap B_2|}{|B_1|} + \frac{|B_2 \cap B_1|}{|B_2|}}{2}.$$

If this similarity value is above the user-specified threshold, the pair is considered as candidates for refactorings.

4. Detection Strategies

Each refactoring requires a different detection strategy. We have implemented four strategies in RefactoringCrawler to detect four types of refactorings. Each strategy is implemented by two predicates, `IsRelevant` and `IsAlike`. The strategy will first select a pair of elements (methods/classes/packages) that have similar shingles, and then apply `IsRelevant` and `IsAlike` to see if they represent a refactoring. `IsRelevant` is a cheap syntactic test that is not based on shingles but is based on the refactoring being detected. `IsAlike` indicates whether one element is the result of refactoring the other element. It tends to be expensive, so is only performed after less expensive tests have succeeded.

Rename Method: Predicate `IsRelevant` takes in a pair of methods that have similar shingles encoding. It responds whether both elements have different names and have the same parent class.

$IsRelevant(\eta_1, \eta_2) =$
$haveSameParentClass(\eta_1, \eta_2)$
$\wedge\, haveDifferentName(\eta_1, \eta_2)$

Predicate `IsAlike` lazily finds the call sites of each method and then responds whether or not the two methods have the same call graph (similarity is over the specified threshold).

Rename Class: `IsRelevant` takes in a pair of classes that have similar shingles encoding. It responds whether the classes have different names.

$IsRelevant(\eta_1, \eta_2) =$
$haveDifferentName(\eta_1, \eta_2)$

Predicate `IsAlike` finds all the methods of each class and all the places where instances of the class are created. It computes the similarity of classes based on the similarity of their methods and the similarity of places that instantiates the classes. It returns whether or not this similarity is larger than the user-specified threshold.

Rename Package: `IsRelevant` takes in a pair of packages that have similar shingles encoding. It responds whether the packages have different names.

$IsRelevant(\eta_1, \eta_2) =$
$haveDifferentName(\eta_1, \eta_2)$

Predicate `IsAlike` lazily finds the import statements referencing the packages. An "imports" edge from each

compilation unit that imports the package to the package declaration is added. The predicate responds whether or not two packages are imported by the same compilation units.

Move Method: `IsRelevant` takes in a pair of methods that have similar shingles encoding. It responds whether the methods have the same name but are in different classes.

$$IsRelevant(\eta_1, \eta_2) = haveSameName(\eta_1, \eta_2) \wedge haveDifferentParentClass(\eta_1, \eta_2)$$

Predicate `IsAlike` lazily finds the call sites of each method in the pair. The predicate distinguishes between methods whose declaration classes happen to be renamings of each other. The predicate responds whether or not two methods have the same call graph (similarity is over the specified threshold).

Other Refactorings: The techniques presented here have the potential to detect other refactorings: extract method, pull-up/push-down method, replace method call, split/merge class, change method signature. All these refactorings involve restructuring bodies of code into chunks that are similar across two versions. Since our technique can detect code that is similar across two versions, these refactorings (and more) are all in our realm.

5. Implementation: RefactoringCrawler, an Eclipse plug-in

We implemented these detection strategies in a plugin for the Eclipse development environment. We assume that the two versions of the components to be compared are loaded as projects inside the Eclipse workspace. The user has to select the two projects for which RefactoringCrawler detects refactorings. The user can control various parameters like the size of the sliding window for the shingles encoding (see Section 3), the number of shingles encodings to represent the digital signature of an method, class or package and the threshold to be used by the ISALIKE predicate (see figure 3). Except for the threshold, MoveMethod refactoring uses the same parameters like RenameMethod to detect potential method candidates.

RefactoringCrawler performs the analysis and returns back the results inside an Eclipse view (see figure 4). The results are grouped in categories corresponding to each refactoring strategy that was selected in the settings window. Double clicking on any leaf Java element opens an editor having selected the declaration of that particular Java element.

We evaluated the performance of RefactoringCrawler on a real case study library. We chose log4j, a popular open-source library for logging the execution of Java code. We chose for comparison version 1.2 (May 2002) and version 1.3alpha6 (January 2005). The library passed through an expansionary phase and it grew from 30KLOC to 62KLOC. We picked randomly a method that existed in both version, namely Hierarchy.exists(). In version 1.3 we renamed this method to Hierarchy.existsRenamed() using the refactoring engine. It took RefactoringCrawler 50 seconds (on a Pentium 4 at 1.33GHz with 1GB of RAM and the settings shown in figure 3) to find out the renamed methods. The Shingles comparison produced 81 pairs of candidate methods. Most of these are further prunned by semantic analysis. The results are shown in figure 4. Among other renamings, RefactoringCrawler found the rename method that we performed and it announced that the refactored nodes are both in Hierarchy class and in LoggerRepository (which is the interface that Hierarchy extends). One rename that shows the noise introduced by backward compatibility, happened in class MDC. Log4J designers intended to rename method `put()` to `put0()` but because of backward compatibility reasons, they still keep it in version 1.3. The `put()` method merely delegates to `put0()`. RefactoringCrawler correctly spots this as a rename. With a threshold of 0.6, RefactoringCrawler found two false positives, namely methods `getAddress()` and `parseAddress()` in class `SMTPAppender`. When examining the methods, they have very similar bodies and have the same call graph. When we repeated the experiment with threshold raised to 0.7, RefactoringCrawler discards them.

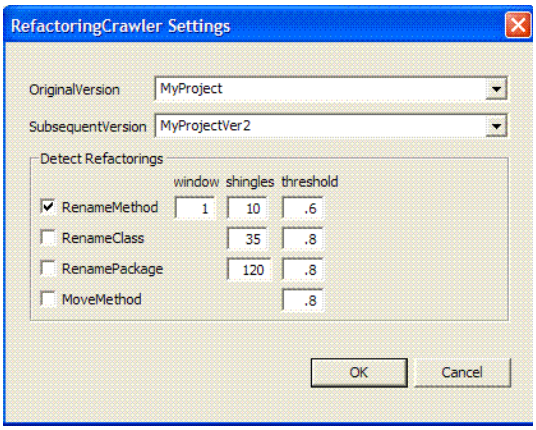

Figure 3. Settings Window

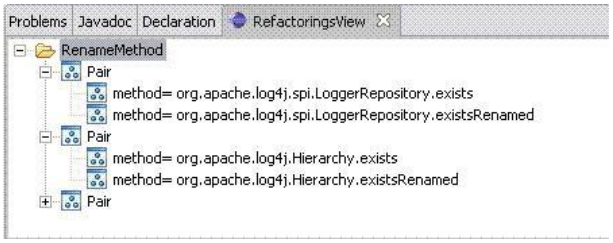

Figure 4. Refactoring View

Manual analysis of the two versions of the library reveals that there are about 8 other method renames. This method renamings happened in classes that were renamed too and our preliminary implementation does not detect them. This happens because step 5 in the methodology (section 2, see figure 1) is not implemented in the tool yet.

6. Related Work

There exists some tool support for detecting and classifying structural evolution, mostly spawned from the reengineering community. Detection of class splitting and merging was the main target of the current tools described in [APM04, DDN00, GZ05, RD03]. Demeyer et al. [DDN00] use a set of object-oriented change metrics and heuristics to detect refactorings that will serve as markers for the reverse engineer. Antonio et. all [APM04] use a technique inspired from the Information Retrieval to detect discontinuities in classes (e.g. a class was replaced with another one, a class was split into two, or two classes merge into one). Based on Vector Space cosine similarity, they compare the class identifiers found in two subsequent releases. Therefore, if a class say Resolver was present in version n, but disappears in version n+1 and a new class SimpleResolver appears in version n+1, they conclude that a class replacement happened. Godfrey and Zou [GZ05] are the closest to the way how we envision detecting structural changes. They implemented a tool that can detect for procedural code some refactorings like renaming, move method, split and merge. Where as we start from shingles encodings analysis, they employ origin analysis along with more expensive analysis like call hierarchy to detect and classify these changes. Rysselberghe and Demeyer [RD03] use a clone finding tool (Duploc) to detect methods that were moved across the classes.

Clone detection based on Shingles encoding is a research interest in other fields like internet content management and file storage. Ramaswamy et al. [RILD04] worked on automatic detection of duplicated fragments in dynamically generated web pages. Since dynamic web pages cannot be cached, at least fragments can be cached thus improving web server's performance. For detecting such fragments, they use shingles encoding. Udi [Mam93] and Kulkarni [KDLT04] employ Shingles-based algorithms to detect redundancy in the file system. They propose more efficient storage after eliminating redundancy.

Existing work on automatic detection of refactorings addresses some of the needs of reverse engineers who must understand at a high level how and why components evolved. For this reason, most of the current work focuses on detecting merging and splitting of classes. However, in order to automatically migrate component-based applications we need to know at the API level what are the changes. Our work complements existing work because we must look also for lower level refactorings that affect the signatures of methods. We also address some of the limitations of existing work with respect to renamings and noise introduced because of preserving backward compatibility.

7. Conclusions

This paper presents a methodology and a tool to detect refactorings that happened between two versions of a reusable component. A trace of refactorings helps in the automatic migration of component-based applications. Based on Shingles encoding, our tool quickly detects a list of potential candidates. More powerful techniques based on semantic analysis are employed to further refine the list of initial candidates. Preliminary evaluation of RefactoringCrawler shows that the methodolgy seems to scale for the size of real-life applications.

References

[APM04] G. Antoniol, M. Di Penta, and E. Merlo. An automatic approach to identify class evolution discontinuities. *in Proceedings of the 7th International Workshop on Principles of Software Evolution*, 2004.

[Bro97] A. Broder. On resemblance and containment of documents. *in Proceedings of SEQUENCES*, 1997.

[DDN00] S. Demeyer, S. Ducasse, and O. Nierstrasz. Finding refactorings via change metrics. *in Proceedings of OOPSLA*, 2000.

[DJ05] D. Dig and R. Johnson. The role of refactoring in the api evolution. *submitted to ICSM*, 2005.

[FBB+99] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts. *Refactoring: Improving the Design of Existing Code*. Adison-Wesley, 1999.

[GZ05] M. Godfrey and L. Zou. Using origin analysis to detect merging and splitting of source code entities. *IEEE Transactions on Software Engineering*, 2005.

[KDLT04] P. Kulkarni, F. Douglis, J. LaVoie, and J. Tracey. Redundancy elimination within large collections of files. *USENIX*, 2004.

[Mam93] Udi Mamber. Finding similar files in a large file system. *Tech Report 93-33, University of Arizona*, 1993.

[Rab81] M. O. Rabin. Fingerprinting by random polynomials. *Technical report, Center for Research in Computing Technology*, 1981.

[RD03] F. Van Rysselberghe and S. Demeyer. Reconstruction of successful software evolution using clone detection. *Proceedings of the International Workshop on Principles of Software Evolution*, 2003.

[RILD04] L. Ramaswany, A. Iyengar, L. Liu, and F. Douglis. Automatic detection of fragments in dynamically generated web pages. *WWW*, 2004.