Automated Detection of Refactorings in Evolving Components

Danny Dig, Can Comertoglu, Darko Marinov, and Ralph Johnson

Department of Computer Science
University of Illinois at Urbana-Champaign
201 N. Goodwin Ave.
Urbana, IL 61801, USA
`{dig,comertog,marinov,johnson}@cs.uiuc.edu`

Abstract. One of the costs of reusing software components is updating applications to use the new version of the components. Updating an application can be error-prone, tedious, and disruptive of the development process. Our previous study showed that more than 80% of the disruptive changes in five different components were caused by refactorings. If the refactorings that happened between two versions of a component could be automatically detected, a refactoring tool could replay them on applications. We present an algorithm that detects refactorings performed during component evolution. Our algorithm uses a combination of a fast syntactic analysis to detect refactoring candidates and a more expensive semantic analysis to refine the results. The experiments on components ranging from 17 KLOC to 352 KLOC show that our algorithm detects refactorings in real-world components with accuracy over 85%.

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 to quickly build a system, but reuse makes the system dependent on the components. Ideally, the interface of a component never changes. In practice, however, new versions of components often change their interfaces and require the developers to change the system to use the new versions of the components.

An important kind of change in object-oriented software is a refactoring. Refactorings [FBB$^+$99] are program transformations that change the structure of a program but not its behavior. Example refactorings include changing the names of classes and methods, moving methods and fields from one class to another, and splitting methods or classes. An automated tool, called *refactoring engine*, can apply the refactorings to change the source code of a component. However, a refactoring engine can change only the source code that it has access to. Component developers often do not have access to the source code of all the applications that reuse the components. Therefore, refactorings that component developers perform preserve the behavior of the component but not of the applications that use the component; in other words, although the change is a refactoring from the component developers' point of view, it is not a refactoring from the application developers' point of view.

One approach to automate the update of applications when their components change is to extend the refactoring engine to record refactorings on the component and then to replay them on the applications. Record-and-replay of refactorings was demonstrated in CatchUp [HD05] and JBuilder2005 [Bor] and recently incorporated in Eclipse 3.2 Milestone 4 [Ecl]. As component developers refactor their code, the refactoring engine creates a log of refactorings. The developers ship this log along with the new version of the component. An application developer can then upgrade the application to the new version by using the refactoring engine to play back the log of refactorings.

While replay of refactorings shows great promise, it relies on the existence of refactoring logs. However, logs are not available for the legacy versions of components. Also, logs will not be available for all future versions; some developers will not use refactoring engines with recording, and some developers will perform refactorings manually. To exploit the full potential of replay, it is therefore important to be able to automatically detect the refactorings used to create a new version of a component.

We propose a novel algorithm that detects a likely sequence of refactorings between two versions of a component. Previous algorithms [APM04, DDN00, GW05, GZ05, RD03] assumed closed-world development, where codebases are used only in-house and changes happen abruptly (e.g., one entity dies in a version and a new refactored entity starts from the next version). However, in the open-world development, components are reused outside the organization, therefore changes do not happen overnight but follow a long deprecate-replace-remove lifecycle. Obsolete entities will coexist with their newer counterparts until they are no longer supported. Also, multiple refactorings can happen to the same entity or related entities. This lifecycle makes it hard to accurately detect refactorings. Our algorithm works fine for both closed- and open-world paradigms.

We aim for our algorithm to help the developer infer a log of refactorings for replay. To be practical, the algorithm needs to detect refactorings with a high accuracy. On one hand, if the algorithm adds to a log a change that is not actually a refactoring (false positive), the developer needs to remove it from the log or the replay could potentially introduce bugs. On the other hand, if the algorithm does not add to a log an actual refactoring (false negative), the developer needs to manually find it and add it to the log. Previous algorithms [APM04,DDN00,GW05,GZ05,RD03] aimed at detection of refactorings for the purpose of program comprehension. Therefore, they can tolerate lower accuracy as long as they focus the developer's attention on the relevant parts of the software.

Our algorithm combines a fast syntactic analysis to detect refactoring candidates and a more expensive semantic analysis to refine the results. Our syntactic analysis is based on Shingles encoding [Bro97], a technique from Information Retrieval. Shingles are a fast technique to find similar fragments in text files; our algorithm applies shingles to source files. Most refactorings involve repartitioning of the source files, which results in similar fragments of source text between different versions of a component. Our semantic analysis is based on the *reference graphs* that represent references among source-level entities, e.g., calls among methods[1]. This analysis considers the semantic

---

[1] These *references* do not refer to pointers between objects but to references among the source-code entities in each version of the component.

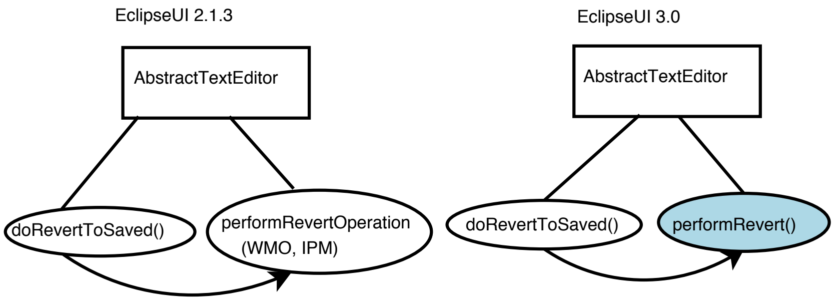


Fig. 1. An excerpt from Eclipse versions 2.1 and 3.0 showing two refactorings, rename method and changed method signature, applied to the same method. The squares represent classes, the ellipses methods, and arrows are method calls. The method that changes signature also changes name from performRevertOperation to performRevert.

relationship between candidate entities to determine whether they represent a refactoring.

We have implemented our algorithm as an Eclipse plugin, called RefactoringCrawler, that detects refactorings in Java components. The ideas in the algorithm can be applied to other programming languages. RefactoringCrawler currently detects seven types of refactorings, focusing on rename and move refactorings that we found to be the most commonly applied in several components [DJ05]. We have evaluated RefactoringCrawler on three components ranging in size from 17 KLOC to 352 KLOC. The results show that RefactoringCrawler scales to real-world components, and its accuracy in detecting refactorings is over 85%.

RefactoringCrawler and our evaluation results are available on the website [Ref].

2 Example

We next illustrate some refactorings that our algorithm detects between two versions of a component. We use an example from the EclipseUI component of the Eclipse development platform. We consider two versions of EclipseUI, from Eclipse versions 2.1.3 and 3.0. Each of these versions of EclipseUI has over 1,000 classes and 10,000 methods in the public API (of non-internal packages). Our algorithm first uses a fast syntactic analysis to find similar methods, classes, and packages between the two versions of the component. (Section 4 presents the details of our syntactic analysis.) For EclipseUI, our algorithm finds 231,453 pairs of methods with similar bodies, 487 pairs of similar classes, and 22 pairs of similar packages. (Section 8 presents more details of this case study.) These similar entities are candidates for refactorings. Our example focuses on two pairs of similar methods.

Figure 1 shows two pairs of similar methods from the two versions of the class `AbtstractTextEditor` from Eclipse 2.1 and 3.0. The syntactic analysis finds that the method `doRevertToSaved` in version 2.1 is similar to (although not identical with) the method `doRevertToSaved` in version 3.0, and the method `performRevertOperation` is similar to the method `performRevert`. Our algorithm then uses a semantic analysis to detect the refactorings that were performed on these pairs. As the result, our algorithm detects that the method `performRevertOperation` was renamed to `performOperation`, and its signature changed from having two arguments in the version 2.1 to no argument in the version 3.0. Our previous manual inspection [DJ05] of the Eclipse documentation and code indeed found that these two refactorings, renamed method and changed method signature, were performed.

Our semantic analysis applies a series of detection strategies that find whether candidate pairs of similar entities are indeed results of refactorings. The key information that the strategies consider is the *references* between the entities in each version. For methods, the references correspond to call edges. For our example methods, both `performRevertOperation` and `performRevert` have only one call in the entire EclipseUI: they are both called exactly once from `doRevertToSaved`. Our analysis represents this information with an edge, labeled with the number of calls, between these methods. We present how the two strategies for renamed methods and changed method signature proceed in our running example.

The strategy that detects renamed methods discards the pair of `doRevertToSaved` methods since they have the same name. This strategy, however, investigates further whether `performRevert` is a renaming of `performRevertOperation`. The strategy (lazily) finds the calls to these two methods and realizes that they are called (the same number of times) from the corresponding `doRevertToSaved` methods in both versions. Therefore, methods `performRevertOperation` and `performRevert` (i) are both in class `AbtstractTextEditor`, (ii) have similar method bodies, (iii) have similar incoming call edges, but (iv) differ in the name. The strategy thus concludes that `performRevert` is a renaming of `performRevertOperation`.

The strategy that detects changed method signatures also considers all pairs of similar methods. This strategy discards the pair of `doRevertToSaved` methods since they have the same signature. This strategy, however, investigates further `performRevertOperation` and `performRevert` methods, because they represent the same method but renamed. It is important to point out here that strategies *share detected refactorings*: although `performRevertOperation` and `performRevert` seemingly have different names, the RenameMethod strategy has already found that these two methods correspond. The ChangedMethodSignature strategy then finds that `performRevertOperation` and `performOperation` (i) have similar method bodies, (ii) "same" name, (iii) similar call edges, but (iv) different signatures. The strategy thus correctly concludes that a changed method signature refactoring was applied to `performOperation`.

3 Algorithm Overview

This section presents a high-level overview of our algorithm for detection of refactorings. Figure 2 shows the pseudo-code of the algorithm. The input are two versions of a component, and the output is a log of refactorings applied on `c1` to produce `c2`. The algorithm consists of two analyses: a fast *syntactic analysis* that finds candidates for refactorings and a precise *semantic analysis* that finds the actual refactorings.

```
Refactorings detectRefactorings(Component c1, c2) {
  // syntactic analysis
  Graph g1 = parseLightweight(c1);
  Graph g2 = parseLightweight(c2);
  Shingles s1 = annotateGraphNodesWithShingles(g1);
  Shingles s2 = annotateGraphNodesWithShingles(g2);
  Pairs pairs = findSimilarEntities(s1, s2);
  // semantic analysis
  Refactorings rlog = emptyRefactorings();
  foreach (DetectionStrategy strategy) {
    do {
     Refactorings rlog’ = rlog.copy();
     foreach (Pair<e1, e2> from pairs relevant to strategy)
        if (strategy.isLikelyRefactoring(e1, e2, rlog))
          rlog.add(<e1, e2>, strategy);
    } while (!rlog’.equals(rlog)); // fixed point
  }
  return rlog;
}
```

Fig. 2. Pseudo-code of the conceptual algorithm for detection of refactorings.

Our syntactic analysis starts by parsing the source files of the two versions of the component into the *lightweight* ASTs, where the parsing stops at the declaration of the methods and fields in classes. For each component, the parsing produces a graph (more precisely, a tree to which analysis later adds more edges). Each node of the graphs represents a source-level entity, namely a package, a class, a method, or a field. Each node stores a fully qualified name for the entity, and each method node also stores the fully qualified names of method arguments to distinguish overloaded methods. Nodes are arranged hierarchically in the tree, based on their fully qualified names: the node $p.n$ is a child of the node p.

The heart of our syntactic analysis is the use of the *Shingles encoding* to find similar pairs of entities (methods, classes, and packages) in the two versions of the component. Shingles are “fingerprints” for strings with the following property: if a string changes slightly, then its shingles also change slightly. Therefore, shingles enable detection of strings with similar fragments much more robustly than the traditional string matching techniques that are not immune to small perturbations like renamings or small edits. Section 4 presents the computation of shingles in detail.

The result of our syntactic analysis is a set of pairs of entities that have similar shingles encodings in the two versions of the component. Each pair consists of an entity from the first version and an entity of the same kind from the second version; there are separate pairs for methods, classes, and packages. These pairs of similar entities are candidates for refactorings.

Our semantic analysis detects from the candidate pairs those where the second entity is a likely refactoring of the first entity. The analysis applies seven strategies for detecting specific refactorings, such as RenameMethod or ChangeMethodSignature discussed in section 2. Section 5 presents the strategies in detail. The analysis applies each strategy until it finds all possible refactorings of its type. Each strategy considers all pairs of entities $\langle e_1, e_2 \rangle$ of the appropriate type, e.g., RenameMethod considers only pairs of methods. For each pair, the strategy computes how likely is that e_1 was refactored into e_2; if the likelihood is above a user-specified threshold, the strategy adds the pair to the log of refactorings that the subsequent strategies can use during further analysis. Note that each strategy takes into account already detected refactorings; sharing detected refactorings among strategies is a key for accurate detection of refactorings when multiple types of refactorings applied to the same entity (e.g., a method was renamed and has a different signature) or related entities (e.g., a method was renamed and also its class was renamed). Our analysis cannot recover the list of refactorings in the order they were performed, but it finds *one path* that leads to the same result.

4 Syntactic Analysis

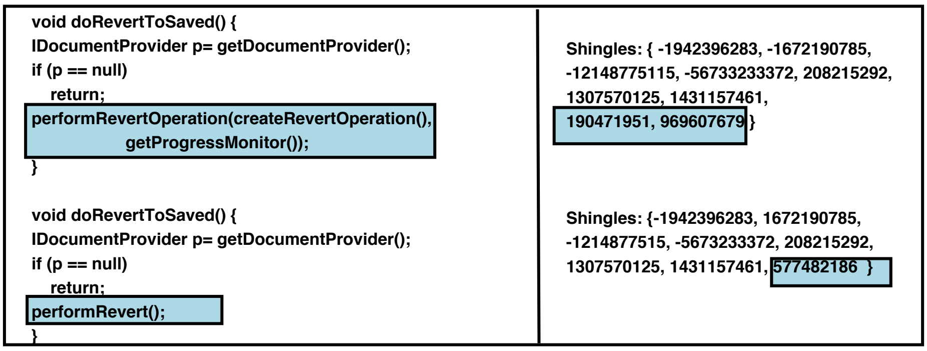
To identify possible candidates for refactorings, our algorithm first determines pairs of *similar* methods, classes, and packages. Our algorithm uses the Shingles encoding [Bro97] to compute a fingerprint for each method and determines two methods to be similar if and only if they have similar fingerprints. Unlike the traditional hashing functions that map even the smallest change in the input to a completely different hash value, the Shingles algorithm maps small changes in the input to small changes in the fingerprint encoding.

4.1 Computing Shingles for Methods

The Shingles algorithm takes as input a sequence of tokens and computes a multiset of integers called shingles. The tokens represent the method body or the Javadoc comments for the method (as interface methods and abstract methods have no body). The tokens do not include method name and signature because refactorings affect these parts. The algorithm takes two parameters, the length of the sliding window, W, and the maximum size of the resulting multiset, S. Given a sequence of tokens, the algorithm uses the sliding window to find all subsequences of length W, computes the shingle for each subsequence, and selects the S minimum shingles for the resulting multiset. Instead of selecting S shingles which have minimum values, the algorithm could use any other heuristic that deterministically selects S values from a larger set. Our implementation uses the Rabin's hash function [Rab81] to compute the shingles.

If the method is short and has fewer than S shingles, then the multiset contains all shingles. This is the case with many setters and getters and some constructors and other initializers. The parameter S acts as the upper bound for the space needed to represent shingles: a larger value of S makes calculations more expensive, and a smaller value makes it harder to distinguish strings. Our implementation sets the number of shingles proportional to the length of the method body/comments.

Figure 3 shows the result of calculating the shingles for two method bodies with $W = 2$ and $S = 10$. The differences in the bodies and the shingle values are in grey boxes. Notice that the small changes in the tokens produce only small changes in the shingle representation, enabling the algorithm to find the similarities between methods.



```
void doRevertToSaved() {
IDocumentProvider p= getDocumentProvider();
if (p == null)
   return;
performRevertOperation(createRevertOperation(),
               getProgressMonitor());
}
```

```
Shingles: { -1942396283, -1672190785,
-12148775115, -56733233372, 208215292,
1307570125, 1431157461,
190471951, 969607679}
```

```
void doRevertToSaved() {
IDocumentProvider p= getDocumentProvider();
if (p == null)
   return;
performRevert();
}
```

```
Shingles: {-1942396283, 1672190785,
-1214877515, -5673233372, 208215292,
1307570125, 1431157461, 577482186 }
```

Fig. 3. Shingles encoding for two versions of `AbstractTextEditor.doRevertToSaved` between Eclipse 2.1 and 3.0. Notice that small changes (gray boxes) in the input strings produce small changes in the Shingles encoding.

4.2 Computing Shingles for Classes and Packages

The shingles for methods are used to compute shingles for classes and packages. The shingles for a class are the minimum S_{class} values of the union of the shingles of the methods in that class. Analogously, the shingles for a package are the minimum $S_{package}$ values of the union of the shingles of the classes in that package. This way, the algorithm efficiently computes shingles values and avoids recalculations.

4.3 Finding Candidates

Our analysis uses the shingles to find candidates for refactorings. Each candidate is a pair of similar entities from the two versions of the component. This analysis is an effective way of eliminating a large number of pairs of entities, so that the expensive operation of computing the reference graphs is only done for a small subset of all possible pairs. More specifically, let M_1 and M_2 be the multisets of shingles for two methods, classes, or packages. Our analysis computes similarity between these two multisets. Let $|M_1 \cap M_2|$ be the cardinality of the intersection of M_1 and M_2. To compare similarity for different pairs, the algorithm *normalizes* the similarity to be between 0 and 1. More precisely, the algorithm computes the similarity as the *average* of similarity from M_1 to M_2 and similarity from M_2 to M_1 to address the cases when M_1 is similar to M_2 but M_2 is not similar to M_1 :

$$\frac{\frac{|M_1 \cap M_2|}{|M_1|} + \frac{|M_2 \cap M_1|}{|M_2|}}{2}.$$

If this similarity value is above the user-specified threshold, the pair is deemed similar and passed to the semantic analysis.

5 Semantic Analysis

We present the semantic analysis that our algorithm uses to detect refactorings. Recall from Figure 2 that the algorithm applies each detection strategy until it reaches a fixed point and that all strategies share the same log of detected refactorings, `rlog`. This sharing is crucial for successful detection of refactorings when multiple types of refactorings happened to the same entity (e.g., a method was renamed and has a different signature) or related entities (e.g., a method was renamed and also its class was renamed). We first describe how the strategies use the shared log of refactorings. We then describe *references* that several strategies use to compute the likelihood of refactoring. We also define the multiplicity of references and the similarity that our algorithm computes between references. We finally presents details of each strategy. Due to the sharing of the log, our algorithm imposes an order on the types of refactorings it detects first. Specifically, the algorithm applies the strategies in the following order:

1. RenamePackage (RP)
2. RenameClass (RC)
3. RenameMethod (RM)
4. PullUpMethod (PUM)
5. PushDownMethod (PDM)
6. MoveMethod (MM)
7. ChangeMethodSignature (CMS)

5.1 Shared Log

The strategies compare whether an entity in one graph corresponds to an entity in another graph *with respect to the already detected refactorings*, in particular with renaming refactorings. Suppose that the refactorings log `rlog` already contains several renamings that map fully qualified names from version `c1` to version `c2`. These renamings map package names to package names, class names to class names, or method names to method names. We define a renaming function ρ that maps a fully qualified name `fqn` from `c1` with respect to the renamings in `rlog`:

$$\begin{aligned}\rho(\texttt{fqn}, \texttt{rlog}) &= \text{if } (\text{defined } \texttt{rlog}(\texttt{fqn})) \text{ then } \texttt{rlog}(\texttt{fqn}) \\ &\quad\ \text{else } \rho(\text{pre}(\texttt{fqn}), \texttt{rlog}) + \texttt{"."} + \text{suf}(\texttt{fqn}) \\ \rho(\texttt{""}, \texttt{rlog}) &= \texttt{""},\end{aligned}$$

where suf and pre are functions that take a fully qualified name and return its simple name (*suffix*) and the entire name but the simple name (*prefix*), respectively. The function ρ recursively checks whether a renaming of some part of the fully qualified name is already in `rlog`.

5.2 References

The strategies compute the likelihood of refactoring based on *references* among the source-code entities in each of the two versions of the component. In each graph that represents a version of the component, our algorithm (lazily) adds an edge from a node n' to a node n if the source entity represented by n' has a reference to a source entity represented by n. (The graph also contains the edges from the parse tree.) We define references for each kind of nodes/entities in the following way:

- There is a reference from a node/method m' to a node/method m iff m' calls m. Effectively, references between methods correspond to the edges in call graphs.
- There is a reference from a node n' to a node/class C iff:
  - n' is a method that has (i) an argument or return of type C, or (ii) an instantiation of class C, or (iii) a local variable of class C.
  - n' is a class that (i) has a field whose type is C or (ii) is a subclass of C.
- There is a reference from a node n' to a node/package p iff n' is a class that imports some class from the package p.

There can be several references from one entity to another. For example, one method can have several calls to another method or one class can have several fields whose type is another class. Our algorithm assigns to each edge a *multiplicity* that is the number of references. For example, if a method m' has two calls to a method m, then the edge from the node n' that represents m' to the node n that represents m has multiplicity two. Conceptually, we consider that there is an edge between any two nodes, potentially with multiplicity zero. We write $\mu(n', n)$ for the multiplicity from the node n' to the node n.

5.3 Similarity of References

Our algorithm uses a metric to determine the similarity of references to entities in the two versions of the component, with respect to a given log of refactorings. We write $n \in \mathtt{g}$ for a node n that belongs to a graph $\mathtt{g}$. Consider two nodes $n_1 \in \mathtt{g1}$ and $n_2 \in \mathtt{g2}$. We define the similarity of their incoming edges as follows. We first define the *directed similarity* between two nodes with respect to the refactorings. We then take the overall similarity between n_1 and n_2 as the average of directed similarities between n_1 and n_2 and between n_2 and n_1. The average of directed similarities helps to compute a fair grade when n_1 is similar to n_2 but n_2 is not similar to n_1.

We define the directed similarity between two nodes n and n' as the overlap of multiplicities of their *corresponding* incoming edges. More precisely, for each incoming edge from a node n_i to n, the directed similarity finds a node $n'_i = \rho(n_i, \mathtt{rlog})$ that corresponds to n_i (with respect to refactorings) and then computes the overlap of multiplicities between the edges from n_i to n and from n'_i to n'. The number of overlapping incoming edges is divided by the total number of incoming edges. The formula for directed similarity is:

$$\delta(n, n', \mathtt{rlog}) = \frac{\sum_{n_i} \min(\mu(n_i, n), \mu(\rho(n_i, \mathtt{rlog}), n'))}{\sum_{n_i} \mu(n_i, n)}$$

The overall similarity is the average of directed similarities:

$$\sigma(n_1, n_2, \mathtt{rlog}) = \frac{\delta(n_1, n_2, \mathtt{rlog}) + \delta(n_2, n_1, \mathtt{rlog}^{-1})}{2}$$

When computing the directed similarity between n_2 and n_1, the algorithm needs to take into account the inverse of renaming log, denoted by $\texttt{rlog}^{-1}$. Namely, starting from a node n_i in g_2, the analysis searches for a node $n_{i'}$ in g_1 such that the renaming of $n_{i'}$ (with respect to `rlog`) is n_i, or equivalently, $\rho(n_i, \texttt{rlog}^{-1}) = n_{i'}$.

We describe informally an equivalent definition of directed similarity based on the view of graphs with multiplicities as multigraphs that can have several edges between two same nodes. The set of edges between two nodes can be viewed as a multiset, and finding the overlap corresponds to finding the intersection of one multiset of edges with the other multiset of edges (for nodes corresponding with respect to the refactorings). In this view, similarity between edges in the graph is conceptually analogous to the similarity of multisets of shingles.

5.4 Detection Strategies

We next precisely describe all detection strategies for refactorings. Each strategy checks appropriate pairs of entities and has access to the graphs `g1` and `g2` and the `rlog` of refactorings. (See the call to `isLikelyRefactoring` in Figure 2.) Figure 4 shows the seven strategies currently implemented in RefactoringCrawler. For each pair, the strategy first performs a fast syntactic check that determines whether the pair is relevant for the refactoring and then performs a semantic check that determines the likelihood of the refactoring. The semantic checks compare the similarity of references to the user-specified threshold value T.

RenamePackage (RP), RenameClass (RC), and RenameMethod (RM) strategies are similar. The first syntactic check requires the entity from `g2` not to be in `g1`; otherwise, the entity is not new. The second check requires the two entities to have the same name prefix, modulo the renamings in `rlog`; otherwise, the refactoring is a potential move but not a rename. The third check requires the two entities to have different simple names.

PullUpMethod (PUM) and PushDownMethod (PDM) are the opposite of each other. Figure 5 illustrates a PUM that pulls up the declaration of a method from a subclass into the superclass such that the method can be reused by other subclasses. Figure 6 illustrates a PDM that pushes down the declaration of a method from a superclass into a subclass that uses the method because the method is no longer reused by other subclasses. In general, the PUM and PDM can be between several classes related by inheritance, not just between the immediate subclass and superclass; therefore, PUM and PDM check that the original class is a *descendant* and an *ancestor*, respectively, of the target class. These inheritance checks are done on the graph `g2`.

MoveMethod (MM) has the second syntactic check that requires the parent classes of the two methods to be different. Without this check, MM would incorrectly classify all methods of a renamed class as moved methods. The second semantic check requires that the declaration classes of the methods not be related by inheritance; otherwise, the refactorings would be incorrectly classified as MM as opposed to a PUM/PDM. The third check requires that all references to the target class be removed in the second version and that all calls to methods from the initial class be replaced with sending a message to an instance of the initial class. We illustrate this check on the sample code in Figure 7. In the first version, method `C1.m1` calls a method `C1.xyz` of the same class `C1` and also calls a method `C2.m2`. After `m1` is moved to the class `C2`, `m1` can call any

Refactoring	Syntactic Checks	Semantic Checks
RP(p_1, p_2)	$p_2 \notin \texttt{g1}$ $\rho(\text{pre}(p_1), \texttt{rlog}) = \text{pre}(p_2)$ $\text{suf}(p_1) \neq \text{suf}(p_2)$	$\sigma(p_1, p_2, \texttt{rlog}) \geq \text{T}$
RC(C_1, C_2)	$C_2 \notin \texttt{g1}$ $\rho(\text{pre}(C_1), \texttt{rlog}) = \text{pre}(C_2)$ $\text{suf}(C_1) \neq \text{suf}(C_2)$	$\sigma(C_1, C_2, \texttt{rlog}) \geq \text{T}$
RM(m_1, m_2)	$m_2 \notin \texttt{g1}$ $\rho(\text{pre}(m_1), \texttt{rlog}) = \text{pre}(m_2)$ $\text{suf}(m_1) \neq \text{suf}(m_2)$	$\sigma(m_1, m_2, \texttt{rlog}) \geq \text{T}$
PUM(m_1, m_2)	$m_2 \notin \texttt{g1}$ $\rho(\text{pre}(m_1), \texttt{rlog}) \neq \text{pre}(m_2)$ $\text{suf}(m_1) = \text{suf}(m_2)$	$\sigma(m_1, m_2, \texttt{rlog}) \geq \text{T}$ $\rho(\text{pre}(m_1), \texttt{rlog})$ descendant-of $\text{pre}(m_2)$
PDM(m_1, m_2)	$m_2 \notin \texttt{g1}$ $\rho(\text{pre}(m_1), \texttt{rlog}) \neq \text{pre}(m_2)$ $\text{suf}(m_1) = \text{suf}(m_2)$	$\sigma(m_1, m_2, \texttt{rlog}) \geq \text{T}$ $\rho(\text{pre}(m_1), \texttt{rlog})$ ancestor-of $\text{pre}(m_2)$
MM(m_1, m_2)	$m_2 \notin \texttt{g1}$ $\rho(\text{pre}(m_1), \texttt{rlog}) \neq \text{pre}(m_2)$ $\text{suf}(m_1) = \text{suf}(m_2)$	$\sigma(m_1, m_2, \texttt{rlog}) \geq \text{T}$ $\neg\rho(\text{pre}(m_1), \texttt{rlog})$ anc.-or-desc. $\text{pre}(m_2)$ references-properly-updated
CMS(m_1, m_2)	$\rho(\text{fqn}(m_1), \texttt{rlog}) = \text{fqn}(m_2)$ $\text{signature}(m_1) \neq \text{signature}(m_2)$	$\sigma(m_1, m_2, \texttt{rlog}) \geq \text{T}$

Fig. 4. Syntactic and semantic checks performed by different detection strategies for refactorings: RP=RenamePackage, RC=RenameClass, RM=RenameMethod, PUM=PullUpMethod, PDM=PushDownMethod, MM=MoveMethod, and CMS=ChangeMethodSignature.

method in `C2` directly (e.g., `m2`), but any calls to methods residing in `C1` need to be executed through an instance of `C1`.

ChangeMethodSignature (CMS) looks for methods that have the same fully qualified name (modulo renamings) but different signatures. The signature of the method can change by gaining/loosing arguments, by changing the type of the arguments, by changing the order of the arguments, or by changing the return type.

6 Discussion of the Algorithm

The example from Section 2 illustrates some of the challenges in automatic detection of refactorings that happened in reusable components. We next explicitly discuss three main challenges and present how our algorithm addresses them.

The first challenge is the size of the code to be analyzed. An expensive semantic analysis—for example finding similar subgraphs in call graphs (more generally, in the entire reference graphs)—might detect refactorings but does not scale up to the size of real-world components with tens of thousands of entities, including methods, classes, and packages. A cheap syntactic analysis, in contrast, might find many similar entities but is fallible to renamings. Also, an analysis that would not take into account the semantics of entity relationships would produce a large number of false positives. Our algorithm uses a hybrid of syntactic and semantic analyses: a fast syntactic analysis

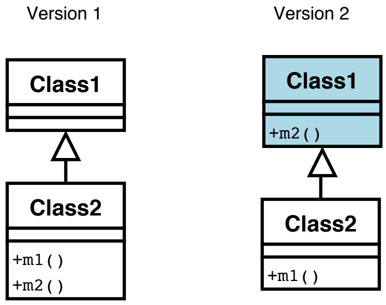


Fig. 5. PullUpMethod: method `m2` is pulled up from the subclass `C2` into the superclass `C1`.

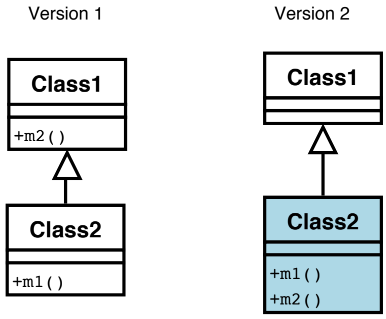


Fig. 6. PushDown: method `m2` is pushed down from the superclass `C1` into the subclass `C2`.

creates pairs of candidate entities that are suspected of refactoring, and a more precise semantic analysis on these candidates detects whether they are indeed refactorings.

The second challenge is the noise introduced by preserving backward compatibility in the components. Consider for example the following change in the Struts framework from version 1.1 to version 1.2.4: the method `perform` in the class `Controller` was renamed to `execute`, but `perform` still exists in the later version. However, `perform` is deprecated, all the internal references to it were replaced with references to `execute`, and the users are warned to use `execute` instead of `perform`. Since it is not feasible to perform an expensive analysis on all possible pairs of entities across two versions of a component, any detection algorithm has to consider only a subset of pairs. Some previous algorithms [APM04, DDN00, GZ05] consider only the outdated entities that die in one version and then search for refactored counterparts that are created in the next version. The assumption that entities change in this fashion indeed holds in the closed-world development (where the only users of components are the component developers) but does not hold in the open-world development where outdated entities coexist with their refactored counterparts. For example, the previous algorithms cannot detect that `perform` was renamed to `execute` since `perform` still exists in the subsequent version. Our algorithm detects that `perform` in the first version and `execute` in the

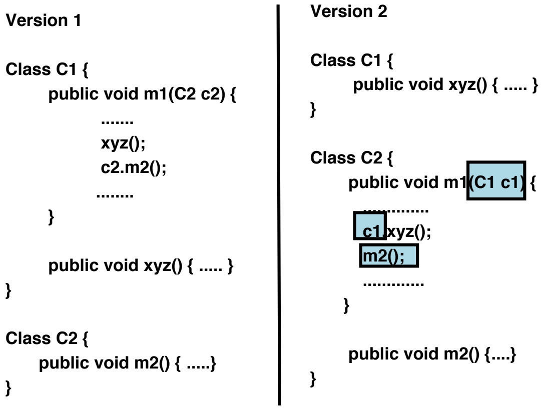


Fig. 7. Method `m1` moves from class `C1` in one version to class `C2` in the next version. The method body changes to reflect that the local methods (e.g., `m2`) are called directly, while methods from the previous class (e.g., `xyz`) are called indirectly through an instance of `C1`.

second version have the same shingles and their call sites are the same, and therefore our algorithm correctly classifies the change as a method rename.

The third challenge is multiple refactorings happening to the same entity or related entities. The example from Section 2, for instance, shows two refactorings, rename method and change method signature, applied to the same method. An example of refactorings happening to related entities is renaming a method along with renaming the method's class. Figure 8 illustrates this scenario. Across the two versions of a component, class `C1` was renamed to `C1REN`, and one of its methods, `m2`, was renamed to `m2REN`. During component evolution, regardless of whether the class or method rename was executed first, the end result is the same. In Figure 8, the upper part shows the case when the class rename was executed first, and the lower part shows the case when the method rename was executed first.

Our algorithm addresses the third challenge by imposing an order on the detection strategies and sharing the information about detected refactorings among the detection strategies. Any algorithm that detects refactorings conceptually reconstructs the log of refactorings and thus not only the start and the end state of a component but also the intermediate states. Our algorithm detects the two refactorings in Figure 8 by following the upper path. When detecting a class rename, the algorithm takes into account only the shingles for class methods and not the method names. Therefore, our algorithm detects class `C1REN` as a rename of class `C1` although one of its methods was renamed. This information is fed back into the loop; it conceptually reconstructs the state 2a, and the analysis continues. The subsequent analysis for the rename method checks whether the new-name method belongs to the same class as the old-name method; since the previous detection discovered that `C1` is equivalent modulo rename with `C1REN`, `m2REN` can be detected as a rename of `m2`.

The order in which an algorithm detects the two refactorings matters. We described how our algorithm detects a class rename followed by a method rename. Consider, in

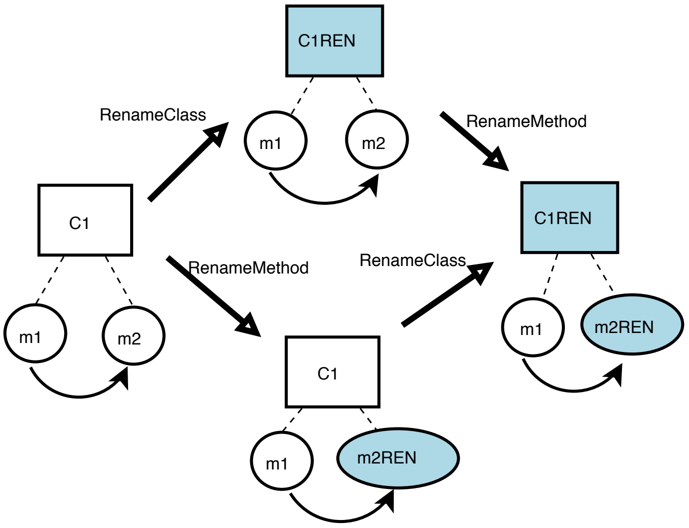


Fig. 8. Refactorings affect related entities class C1 and method m2. The class rename happens before the method rename in the upper path, the reverse happens in the bottom path. Both paths end up with the same result.

contrast, what would happen to an algorithm that attempts to follow the bottom path. When analyzing what happened between the methods `m2` and `m2REN`, the algorithm would need the intermediate state 2b (where `m2REN` belongs to `C1`) to detect that `m2` was renamed to `m2REN`. However, that state is not given, and in the end state `m2REN` belongs to `C1REN`, so the algorithm would mistakenly conclude that `m2REN` was moved to another class (`C1REN`). The subsequent analysis of what happened between classes `C1` and `C1REN` would presumably find that they are a rename and would then need to backtrack to correct the previously misqualified move method as a rename method. For this reason, our algorithm imposes an order on the detection strategies and runs detection of renamings top-down, from packages to classes to methods.

To achieve a high level of accuracy, our algorithm uses a fixed-point computation in addition to the ordering of detection strategies. The algorithm runs each strategy repeatedly until it finds no new refactorings. This loop is necessary because entities are intertwined with other entities, and a strategy cannot detect a refactoring in one entity until it detects a refactoring in the dependent entities. For instance, consider this example change that happened in the Struts framework between the versions 1.1 and 1.2.4: in the class `ActionController`, the method `perform` was renamed to `execute`. The implementation of `perform` in `ActionController` is a utility class that merely delegates to different subclasses of `Action` by sending them a `perform` message. For 11 of these `Action` classes, their callers consist mostly of the `ActionController.perform`. Therefore, unless a tool detects first that `perform` was renamed to `execute`, it cannot detect correctly the similarity of the incoming call edges for the other 11 methods. After the first run of the RenameMethod detection, our RefactoringCrawler tool misses the 11 other method renames. However, the feedback loop adds the information about the rename of `perform`, and the second run of the RenameMethod detection correctly finds another 11 renamed methods.

Even though we only analyze seven types of refactorings, conceptually similar combination of syntactic and semantic analysis can detect many other types of refactorings. A lot of the refactorings published by Fowler et al. [FBB$^+$99] can be detected in this way, including extract/inline method, extract/inline package, extract/inline class or interface, move class to different package, collapse class hierarchy into a single class, replace record with data class, replace anonymous with nested class, replace type conditional code with polymorphism, as well as some higher-level refactorings to design patterns [GHJV95] including create Factory methods, form Template Method, replace type code with State/Strategy.

The largest extension to the current algorithm is required by 'replace type conditional code with polymorphism'. This refactoring replaces a switch statement whose branches type-check the exact type of an object (e.g., using *instanceof* in Java) with a call to a polymorphic method that is dynamically dispatched at run time to the right class. All the code in each branch statement is moved to the class whose type was checked in that branch. To detect this refactoring, the syntactic analysis should not only detect similar methods, but also similar statements and expressions within method bodies. This requires that shingles are computed for individual statements and expressions, which is overhead to the current implementation, but offers a finer level of granularity. Upon detection of similar statements in a switch branch and in a class method, the semantic analysis needs to check whether the class has the same type as the one checked in the branch and whether the switch is replaced in the second version with a call to the polymorphic method.

7 Implementation

We have implemented our algorithm for detecting refactorings in RefactoringCrawler, a plugin for the Eclipse development environment. The user loads the two versions of the component to be compared as projects inside the Eclipse workspace and selects the two projects for which RefactoringCrawler detects refactorings. To experiment with the accuracy and performance of the analysis, the user can set the values for different parameters, such as the size of the sliding window for the Shingles encoding (Section 4); the number of shingles to represent the digital fingerprint of methods, classes and package; and the thresholds used in computing the similarity of shingles encoding or the reference graphs. RefactoringCrawler provides a set of default parameter values that should work fine for most Java components.

RefactoringCrawler provides an efficient implementation of the algorithm shown in Figure 2. The syntactic analysis starts by parsing the source files of the two versions of the component and creates a graph representation mirroring the *lightweight* ASTs. We call it lightweight because the parsing stops at the declaration of the methods and fields in classes. RefactoringCrawler then annotates each method and field node with shingles values corresponding to the source code behind each node (e.g. method body or field initializers). From the leaves' shingles values, RefactoringCrawler annotates (bottom-up) with shingles values all the nodes corresponding to classes and packages. Since each node contains the fully qualified name of the source code entity, it is easy to navigate back and forth between the actual source code and the graph representation.

During the semantic analysis, RefactoringCrawler uses Eclipse's search engine to find the references among source code entities. The search engine operates on the source code, not on the graph. The search engine does a type analysis to identify the class of a reference when two methods in unrelated classes have the same name. Finding the references is an expensive computation, so RefactoringCrawler lazily runs this and caches the intermediate results by adding edges between the graph nodes that refer each other.

RefactoringCrawler performs the analysis and returns back the results inside an Eclipse view. RefactoringCrawler presents only the refactorings that happened to the public API level of the component since only these can affect the component users. RefactoringCrawler groups the results in categories corresponding to each refactoring strategy. Double clicking on any leaf Java element opens an editor having selected the declaration of that particular Java element. RefactoringCrawler also allows the user to export the results into an XML format compatible with the format that CatchUp [HD05] uses to load a log of refactorings. A similar XML format is used for the Eclipse 3.2 Milestone 4. Additionally, the XML format allows the developer to further analyze and edit the log, removing false positives or adding missed refactorings.

The reader can see screenshots and is encouraged to download the tool from the website [Ref].

8 Evaluation

We evaluate RefactoringCrawler on three real-world components. To measure the accuracy of RefactoringCrawler, we need to know the refactorings that were applied in the components. Therefore, we chose the components from our previous study [DJ05] that analyzed the API changes in software evolution and found refactorings to be responsible for more than 80% of the changes. The previous study considered components with good release notes describing the API changes. Starting from the release notes, we manually discovered the refactorings applied in these components. These manually discovered refactorings helped us to measure the accuracy of the refactoring logs that RefactoringCrawler reports. In general, it is easier to detect the false positives (refactorings that RefactoringCrawler erroneously reports) by comparing the reported refactorings against the source code than it is to detect the false negatives (refactorings that RefactoringCrawler misses). To determine false negatives, we compare the manually found refactorings against the refactorings reported by RefactoringCrawler. Additionally, RefactoringCrawler found a few refactorings that were not documented in the release notes. Our previous study and the evaluation of RefactoringCrawler allowed us to build a repository of refactorings that happened between the two versions of the three components. The case study along with the tool and the detected refactorings can be found online [Ref].

For each component, we need to choose two versions. The previous study [DJ05] chose two major releases that span large architectural changes because such releases are likely to have lots of changes and to have the changes documented. We use the same versions to evaluate RefactoringCrawler. Note, however, that these versions can present hard cases for RefactoringCrawler because they are far apart and can have large changes. RefactoringCrawler still achieves practical accuracy for these versions. We believe that RefactoringCrawler could achieve even higher accuracy on closer versions with less changes.

8.1 Case Study Components

Table 1 shows the size of the case study components. ReleaseNotes give the size (in pages) of the documents that the component developers provided to describe the API changes. We next describe the components and the versions that we analyze [DJ05].

	Size KLOC	Packages	Classes	Methods	ReleaseNotes [Pages]
Eclipse.UI 2.1.3	222	105	1151	10285	-
Eclipse.UI 3.0	352	192	1735	15894	8
Struts 1.1	114	88	460	5916	-
Struts 1.2.4	97	78	469	6044	16
JHotDraw 5.2	17	19	160	1458	-
JHotDraw 5.3	27	19	195	2038	3

Table 1. Size of the studied components.

Eclipse Platform [eclipse.org] provides many APIs and many different smaller frameworks. The key framework in Eclipse is a plug-in based framework that can be used to develop and integrate software tools. This framework is often used to develop Integrated Development Environments (IDEs). We focus on the UI subcomponent (Eclipse.UI) that contains 13 plug-ins.

We chose two major releases of Eclipse, 2.1 (March 2003) and 3.0 (June 2004). Eclipse 3.0 came with some major themes that affected the APIs. The *responsiveness* theme ensured that more operations run in the background without blocking the user. New APIs allow long-running operations like builds and searches to be performed in the background while the user continues to work. Another major theme in 3.0 is *rich-client platforms*. Eclipse was designed as a universal IDE. However many components of Eclipse are not particularly specific to IDEs and can be reused in other rich-client applications (e.g., plug-ins, help system, update manager, window-based GUIs). This architectural theme involved factoring out IDE-specific elements. APIs heavily affected by this change are those that made use of the filesystem resources. For instance `IWorkbenchPage` is an interface used to open an editor for a file input. All methods that were resource specific (those that dealt with opening editors over files) were removed from the interface. A client who opens an editor for a file should convert it first to a generic editor input. Now the interface can be used by both non-IDE clients (e.g., an electronic mail client that edits the message body) as well as IDE clients.

Struts [struts.apache.org] is an open source framework for building Java web applications. The framework is a variation of the Model-View-Controller (MVC) design paradigm. Struts provides its own Controller component and integrates with other technologies to provide the Model and the View. For the Model, Struts can interact with standard data access technologies, like JDBC and EJB, and many third-party packages. For the View, Struts works with many presentation systems.

We chose two major releases of Struts, 1.1 (June 2003) and 1.2.4 (September 2004). All the API changes reveal consolidation work that was done in between the two releases. The developers eliminated duplicated code and removed unmaintained or buggy code.

JHotDraw [jhotdraw.org] is a two-dimensional graphics framework for structured drawing editors. In contrast to the Swing graphics library, JHotDraw defines a basic skeleton for a GUI-based editor with tools in a tool palette, different views, user-defined graphical figures, and support for saving, loading, and printing drawings. The framework has been used to create many different editors.

We chose two major releases of JHotDraw, 5.2 (February 2001) and 5.3 (January 2002). The purpose of 5.3 release was to clean up the APIs and fix bugs.

8.2 Measuring the Recall and Precision

To measure the accuracy of RefactoringCrawler, we use precision and recall, two standard metrics from the Information Retrieval field. *Precision* is the ratio of the number of relevant refactorings found by the tool to the total number of irrelevant and relevant refactorings found by the tool. It is expressed as the percentage:

$$PRECISION = \mathit{GoodResults}/(\mathit{GoodResults} + FalsePositives)$$

Recall is the ratio of the number of relevant refactorings found by the tool (good results) to the total number of actual refactorings in the component. It is expressed as the percentage:

$$RECALL = \mathit{GoodResults}/(\mathit{GoodResults} + FalseNegatives)$$

Ideally, precision and recall should be 100%. If that was the case, the reported refactorings could be fed directly into a tool that replays them to automatically upgrade component-based applications. However, due to the challenges mentioned in Section 6, it is hard to have 100% precision and recall.

Table 2 shows how many instances of each refactoring were found for the three components. These results use the default values for the parameters in RefactoringCrawler [Ref]. For each refactoring type, we show in a triple how many good results RefactoringCrawler found, how many false positives RefactoringCrawler found, and how many false negatives (according to the release notes [DJ05]) RefactoringCrawler found. For each component, we compute precision and recall that take into account the refactorings of all kinds.

We further analyzed why RefactoringCrawler missed a few refactorings. In Struts, for instance, method `RequestUtils.computeParameters` is moved to

	RM	*RC*	*RP*	*MM*	*PUM*	*PDM*	*CMS*	*Precision*	*Recall*
EclipseUI 2.1.3 - 3.0	2,1,0	0,0,0	0,0,0	8,2,4	11,0,0	0,0,0	6,0,0	90%	86%
Struts 1.2.1 - 1.2.4	20,0,1	1,0,1	0,0,0	20,0,7	1,0,0	0,0,0	24,0,1	100%	86%
JHotDraw 5.2 - 5.3	5,0,0	0,0,0	0,0,0	0,0,0	0,0,0	0,0,0	19,0,0	100%	100%

Table 2. Triples of (GoodResults, FalsePositives, FalseNegatives) for RenameMethod(RM), RenameClass(RC), RenamePackage(RP), MoveMethod(MM), PullUpMethod(PUM), PushDownMethod(PDM), ChangeMethodSignature(CMS)

`TagUtils.computeParameters`, and method `RequestUtils.pageURL` is moved to `TagUtils.pageURL`. There are numerous calls to these methods from a test class. However, it appears that the test code was not refactored, and therefore it still calls the old method (that is deprecated), which results in quite different call sites for the old and the refactored method.

8.3 Performance

The results in Table 2 were obtained when RefactoringCrawler ran on a Fujitsu laptop with a 1.73GHz Pentium 4M CPU and 1.25GB of RAM. It took 16 min 38 sec for detecting the refactorings in EclipseUI, 4 min and 55 sec for Struts, and 37 sec for JHotDraw. Figure 9 shows how the running time for JHotDraw varies with the change of the method similarity threshold values used in the syntactic analysis. For low threshold values, the number of candidate pairs passed to the semantic analysis is large, resulting in longer analysis time. For high threshold values, fewer candidate pairs pass into the semantic analysis, resulting in lower running times. For JHotDraw, a .1 method similarity threshold passes 1842 method candidates to the RenameMethod's semantic analysis, a .5 threshold value passes 88 candidates, while a .9 threshold passes only 4 candidates.

The more important question, however, is how precision and recall vary with the change of the similarity threshold values. Very low threshold values produce a larger number of candidates to be analyzed, which results in a larger number of false positives, but increases the chance that all the relevant refactorings are found among the results. Very high threshold values imply that only those candidates that have almost perfect body resemblance are taken into account, which reduces the number of false positives but can miss some refactorings. We have found that threshold values between 0.5 and 0.7 result in practical precision and recall.

8.4 Strengths and Limitations

We next discuss the strengths and the limitations of our approach to detecting refactorings. We also propose new extensions to overcome the limitations.

Strengths

- **High precision and recall.** Our evaluation on the three components shows that both precision and recall of RefactoringCrawler are over 85%. Since RefactoringCrawler

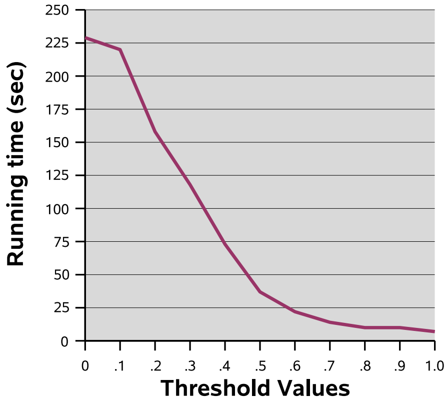


Fig. 9. Running time for JHotDraw decreases exponentially with higher threshold values used in the syntactic analysis.

combines both syntactic and semantic analysis, it can process a realistic size of software with practical accuracy. Compared to other approaches [APM04,DDN00, GW05, GZ05, RD03] that use only syntactic analysis and produce large number of false positives, our tool requires little human intervention to validate the refactorings. RefactoringCrawler can significantly reduce the burden necessary to find refactoring logs that a replay tool uses to automatically upgrade component-based applications.

- **Robust.** Our tool is able to detect refactorings in the presence of noise introduced because of maintaining backwards compatibility, the noise of multiple refactorings, and the noise of renamings. Renamings create huge problems for other approaches but do not impede our tool. Since our tool identifies code entities (methods, classes, packages) based on their body resemblance and not on their names, our tool can successfully track the same entity across different versions, even when its name changes. For previous approaches, a rename is equivalent with an entity disappearing and a brand new entity appearing in the subsequent version. Another problem for previous approaches is the application of multiple refactorings to the same entity. Our tool takes this into account by sharing the log of refactorings between the detection strategies and repeating each strategy until it reaches a fixed point. Lastly, our tool detects refactorings in an open-world development where, due to backwards compatibility, obsolete entities coexist with their refactored counterparts until the former are removed. We can detect refactorings in such an environment because most of refactorings involve repartitioning the source code. This results in parts of the code from a release being spread in different places in the next release. Our algorithm starts by detecting the similarities between two versions.
- **Scalable.** Running expensive semantic analysis (like identifying similar subgraphs in the entire reference graph) on large codebases comprising of tens of thousands of nodes (methods, classes, packages) is very expensive. To avoid this, we run first an inexpensive syntactic analysis that reduces the whole input domain to a relatively

small number of candidates to be analyzed semantically. It took RefactoringCrawler 16 min 38 sec to analyze for the org.eclipse.ui subcomponent (352 KLOC) of the Eclipse Platform.

Limitations

- **Poor support for interfaces and fields.** Since our approach tracks the identity of methods, classes, and packages based on their textual bodies and not on their names, it does not fit for those entities that lack a body. Both class fields and interface methods do not contain any body other than their declaration name. After the syntactic analysis, only entities that have a body resemblance are passed to the semantic analysis. Therefore, refactorings that happened to fields or interface methods cannot be detected. This was the case in org.eclipse.ui where between versions 2.1.3 and 3.0 many static fields were moved to other classes and many interface methods were moved to abstract classes. To counteract the lack of textual bodies for fields or interface methods, we treated their associated javadoc comments as their text bodies. This seems to work for some cases, but not all.
- **Requires experimentation.** As with any approach based on heuristics, coming up with the right values for the detection algorithms might take a few trials. Selecting threshold values too high reduces the false positives toward zero but can miss some refactorings as only those candidates that have perfect resemblance are selected. Selecting too low threshold values produces a large number of false positives but increases the chances that all relevant refactorings are found among the results. The default threshold values for RefactoringCrawler are between 0.5 and 0.7 (for various similarity parameters) [Ref]. When default values do not produce adequate results, users could start from high threshold values and reduce them until the number of false positive becomes too large.

9 Related Work

We provide an overview of related work on refactoring, automated detection of refactorings, and the use of Shingles encoding.

9.1 Refactoring

Programmers have been cleaning up their code for decades, but the term *refactoring* was coined much later [OJ90]. Opdyke [Opd92] wrote the first catalog of refactorings, while Roberts and Brant [RBJ97,Rob99] were the first to implement a refactoring engine. The refactoring field gained much popularity with the catalog of refactorings written by Fowler et al. [FBB^{+}99]. Soon after this, IDEs began to incorporate refactoring engines. Tokuda and Batory [TB01] describe how large architectural changes in two frameworks can be achieved as a sequence of small refactorings. They estimate that automated refactorings are 10 times quicker to perform than manual ones.

More recent research on refactoring focuses on the analyses for automating powerful refactorings. Tip et al. [TKB03] use type constraints to support an analysis for refactorings that introduce type generalization. Donovan et al. [DKTE04] use a pointer analysis and a set-constraint-based analysis to support refactorings that replace the instantiation of raw classes with generic classes. Dincklage and Diwan [vDD04] use various heuristics to convert from non-generic classes to generic classes. Balaban et al. [BTF05] propose refactorings that automatically replace obsolete library classes with their newer counterparts. Component developers have to provide mappings between legacy classes and their replacements, and an analysis based on type constraints determines where the replacement can be done. Thomas [Tho05] points out that refactorings in the components result into integration problems and advocates the need for languages that would allow developers to specify refactorings to create customizable refactorings.

9.2 Detection of refactorings

Researchers have already developed 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. 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 al. [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 some refactorings like renaming, move method, split, and merge for procedural code. Whereas we start from shingles analysis, they employ origin analysis along with a more expensive analysis on call graphs 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. Gorg and Weisgerber [GW05] analyze subsequent versions of a component in configuration management repositories to detect refactorings.

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 the changes to the API. Our work complements existing work because we must look also for lower level refactorings that affect the signatures of methods. We also address the limitations of existing work with respect to renamings and noise introduced by multiple refactorings on the same entity or the noise introduced by the deprecate-replace-remove cycle in the open-world components.

9.3 Shingles encoding

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. Dynamic web pages cannot be cached, but performance can be improved by caching fragments of web pages. They used Shingles encoding to detect fragments of web pages that did not change. Manber [Man93] and Kulkarni et al. [KDLT04] employ shingles-based algorithms to detect redundancy in the file system. They propose more efficient storage after eliminating redundancy. Li et al. [LLMZ04] use shingles to detect clones of text in the source code of operating systems. They further analyze the clones to detect bugs due to negligent copy and paste.

10 Conclusions

Syntactic analyses are too unreliable, and semantic analyses are too slow. Combining syntactic and semantic analyses can give good results. By combining Shingles encoding with traditional semantic analyses, and by iterating the analyses until a fixed point was discovered, we could detect over 85% of the refactorings while producing less than 10% false positives.

The algorithm would work on any two versions of a system. It does not assume that the later version was produced by any particular tool. If a new version is produced by a refactoring tool that records the refactorings that are made, then the log of refactorings will be 100% accurate. Nevertheless, there may not be the discipline or the opportunity to use a refactoring tool, and it is good to know that refactorings can be detected nearly as accurately without it.

There are several applications of automated detection of refactorings. First, a log of refactorings helps in the automated migration of component-based applications. As our previous study [DJ05] shows, more than 80% of the API changes that break compatibility with existing applications are refactorings. A tool like Eclipse can replay the log of refactorings. The replay is done at the application site where both the component and the application reside in the same workspace. In this case, the refactoring engine finds and correctly updates all the references to the refactored entities, thus migrating the application to the new API of the component.

Second, a log of refactorings can improve how current configuration management systems deal with renaming. A tool like CVS looses all the change history for a source file whose main class gets renamed, since this appears as if the old source file was removed and a source file with a new name was added. A log of refactorings can help the configuration management system to correlate the old files/folders with the new files/folders when the main class or package was renamed.

Third, a log of refactoring can help a developer understand how an object-oriented system has evolved from one version to another. For example, an explicit list of renamings tells how the semantics of the refactored entity changed, while a list of moved methods tells how the class responsibilities shifted.

The tool and the evaluation results are available online [Ref].

11 Acknowledgments

We would like to thank Zheng Shao and Jiawei Han who suggested the use of shingles for detecting similar methods. Adam Kiezun, Russ Ruffer, Filip Van Rysselberghe, Danny Soroker, anonymous reviewers, and members of the SAG group at UIUC provided valuable feedback on the drafts of this paper. This work is partially funded through an Eclipse Innovation Grant for which we are very grateful to IBM.

References

[APM04] Giuliano Antoniol, Massimiliano Di Penta, and Ettore Merlo. An automatic approach to identify class evolution discontinuities. In *IWPSE'04: Proceedings of International Workshop on Principles of Software Evolution*, pages 31–40, 2004.

[Bor] What's new in Borland Jbuilder 2005. http://www.borland.com/resources/en/pdf/white_papers/jb2005_whats_new.pdf.

[Bro97] Andrei Broder. On the resemblance and containment of documents. In *SEQUENCES '97: Proceedings of Compression and Complexity of Sequences*, pages 21–29, 1997.

[BTF05] Ittai Balaban, Frank Tip, and Robert Fuhrer. Refactoring support for class library migration. In *OOPSLA '05: Proceedings of Object-oriented programming, systems, languages, and applications*, pages 265–279, New York, NY, USA, 2005. ACM Press.

[DDN00] Serge Demeyer, Stéphane Ducasse, and Oscar Nierstrasz. Finding refactorings via change metrics. In *OOPSLA'00: Proceedings of Object oriented programming, systems, languages, and applications*, pages 166–177, 2000.

[DJ05] Danny Dig and Ralph Johnson. The role of refactorings in api evolution. In *ICSM'05: Proceedings of International Conference on Software Maintenance*, pages 389–398, Washington, DC, USA, 2005. IEEE Computer Society.

[DKTE04] Alan Donovan, Adam Kiezun, Matthew S. Tschantz, and Michael D. Ernst. Converting Java programs to use generic libraries. In *OOPSLA '04: Proceedings of Object-oriented programming, systems, languages, and applications*, volume 39, pages 15–34, New York, NY, USA, October 2004. ACM Press.

[Ecl] Eclipse Foundation. http://eclipse.org.

[FBB$^+$99] Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. *Refactoring: Improving the Design of Existing Code*. Adison-Wesley, 1999.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[GW05] Carsten Gorg and Peter Weisgerber. Detecting and visualizing refactorings from software archives. In *IWPC'05: Proceedings of the 13th International Workshop on Program Comprehension*, pages 205–214, Washington, DC, USA, 2005. IEEE Computer Society.

[GZ05] Michael W. Godfrey and Lijie Zou. Using origin analysis to detect merging and splitting of source code entities. *IEEE Transactions on Software Engineering*, 31(2):166–181, 2005.

[HD05] Johannes Henkel and Amer Diwan. CatchUp!: Capturing and replaying refactorings to support API evolution. In *ICSE'05: Proceedings of International Conference on Software Engineering*, pages 274–283, 2005.

[KDLT04] Purushottam Kulkarni, Fred Douglis, Jason D. LaVoie, and John M. Tracey. Redundancy elimination within large collections of files. In *USENIX Annual Technical Conference, General Track*, pages 59–72, 2004.

[LLMZ04] Zhenmin Li, Shan Lu, Suvda Myagmar, and Yuanyuan Zhou. CP-Miner: A tool for finding copy-paste and related bugs in operating system code. In *OSDI'04: Proceedings of the Sixth Symposium on Operating System Design and Implementation*, pages 289–302, 2004.

[Man93] Udi Manber. Finding similar files in a large file system. Technical Report 93-33, University of Arizona, 1993.

[OJ90] Bill Opdyke and Ralph Johnson. Refactoring: An aid in designing application frameworks and evolving object-oriented systems. In *SOOPPA'90: Proceedings of Symposium on Object-Oriented Programming Emphasizing Practical Applications*, 1990.

[Opd92] Bill Opdyke. *Refactoring Object-Oriented Frameworks*. PhD thesis, University of Illinois at Urbana-Champaign, 1992.

[Rab81] Michael O. Rabin. Fingerprinting by random polynomials. Technical Report 15-81, Harvard University, 1981.

[RBJ97] Don Roberts, John Brant, and Ralph E. Johnson. A refactoring tool for Smalltalk. *TAPOS*, 3(4):253–263, 1997.

[RD03] Filip Van Rysselberghe and Serge Demeyer. Reconstruction of successful software evolution using clone detection. In *IWPSE'03: Proceedings of 6th International Workshop on Principles of Software Evolution*, pages 126–130, 2003.

[Ref] RefactoringCrawler's web page:. https://netfiles.uiuc.edu/dig/RefactoringCrawler .

[RILD04] Lakshmish Ramaswamy, Arun Iyengar, Ling Liu, and Fred Douglis. Automatic detection of fragments in dynamically generated web pages. In *WWW '04: Proceedings of the 13th international conference on World Wide Web*, pages 443–454, New York, NY, USA, 2004. ACM Press.

[Rob99] Don Roberts. *Practical Analysis for Refactoring*. PhD thesis, University of Illinois at Urbana-Champaign, 1999.

[TB01] Lance Tokuda and Don Batory. Evolving object-oriented designs with refactorings. *Automated Software Engineering*, 8(1):89–120, January 2001.

[Tho05] Dave Thomas. Refactoring as meta programming? *Journal of Object Technology*, 4(1):7–11, January-February 2005.

[TKB03] Frank Tip, Adam Kiezun, and Dirk Bauemer. Refactoring for generalization using type constraints. In *OOPSLA '03: Proceedings of Object-oriented programing, systems, languages, and applications*, volume 38, pages 13–26, New York, NY, USA, November 2003. ACM Press.

[vDD04] Daniel von Dincklage and Amer Diwan. Converting Java classes to use generics. In *OOPSLA '04: Proceedings of Object-oriented programming, systems, languages, and applications*, pages 1–14. ACM Press, 2004.