Toward Automatic Upgrade of Component-Based Applications

Danny Dig
University of Illinois at Urbana-Champaign
201 N Goodwin Ave,Urbana, IL, U.S.
dig@uiuc.edu

Advisor: Ralph Johnson
Dept of Computer Science
University of Illinois at Urbana-Champaign
johnson@cs.uiuc.edu

1. PROBLEM AND OVERVIEW

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. This can be error-prone, tedious, and disruptive of the development process.

An important kind of change to object-oriented software is a refactoring [10]. Refactorings 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, the refactoring engine operates within a *closed-world* paradigm: it 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; 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. There is a mismatch between the *closed-world* paradigm in which refactoring engines operate and the *open-world* paradigm where components are developed at one site and reused all over the world.

Our 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 recently demonstrated in CatchUp [14] and JBuilder2005 and is planned to be a standard part of Eclipse 3.2. 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 existing 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 of refactorings, it is therefore important to be able to automatically detect the refactorings used to create a new version of a component.

This paper addresses the following key questions on upgrades: How much of the component evolution can be expressed in terms of refactorings? Can refactorings lead to safe, automatic upgrading of component-based applications? Can refactorings be gathered automatically or with minimal involvement from component developers? Is the accuracy of the detection good enough to be used in practical applications? To answer these questions, we conducted two projects: (i) first project to characterize changes in real-world components and (ii) second (ongoing) project to build a tool-set that automates upgrades.

We studied the API changes of four mature, widely used frameworks and one library. We discovered that the changes that break existing applications (from hereon called *breaking changes*) are not random, but they tend to fall into particular categories. Over 80% of these changes could be considered *refactorings* if looked at from the point of view of the whole system. If the refactorings that happened between two versions of a component could be automatically detected, an *enhanced* refactoring engine could replay them on applications.

We propose a novel algorithm that detects refactorings performed between two versions of a component. Our algorithm works in the realistic open-world paradigm where components are reused outside the developing organization. Interface changes do not happen overnight but follow a long *deprecate-replace-remove* lifecycle. Obsolete entities (marked as deprecate) 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 style of development introduces enough challenges that existing algorithms for detection of refactorings cannot accurately detect the refactorings.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ACM Student Research Competition, Grand Finals, 2006
Copyright 2006 Copyright belongs to the author .

We have implemented our algorithm in an Eclipse [1] 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 the most common rename and move refactorings [9]. 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%.

2. RELATED WORK

Software evolution has long been a topic of study [16]. Others [5] have focused on *why* software changes; we want to discover *how* it changes. Our goal is to *reduce the burden of reuse* on maintenance. To our knowledge, no quantitative and qualitative study has been published about the kind of API changes that occur in components. Bansiya [3] and Mattson [17] used metrics to assess the stability of frameworks. Their metrics can only detect the effect of changes in the framework and not the exact type of change (e.g., they observed that method argument types have been changed between subsequent versions whereas we observe whether they changed because of adding/removing of parameters or because of changing the argument types).

The original work on refactoring was motivated by framework evolution. Opdyke [18] looked at the Choices operating system and the kind of refactorings that occurred as it evolved. Graver [13] studied an object-oriented compiler framework as it went through three iterations. Tokuda and Batory [21] describe the evolution of two frameworks, focusing on how large architectural changes can be accomplished by a sequence of refactorings. However, none of these studies determined the fraction of changes that are refactorings. How much of the component evolution can be expressed in terms of refactorings? The only way to tell is to look at changes in a component over time and categorize them.

Tool support for upgrading applications has been a long time interest. Several authors [6, 15, 19] discuss different annotations within the component's source code that can be used by tools to upgrade applications. However, writing such annotations is cumbersome. Balaban et al.[2] aim to automatically replace obsolete library classes with their newer counterparts. Component developers have to provide mappings between legacy classes and their replacements. A more appealing approach would be if tools could generate this information. Extending the refactoring engines to record the refactorings seems the most natural approach. However, since a lot of people do not capture the refactorings or logs of refactorings are not available for legacy systems, the next best solution is to infer the refactorings.

Existing algorithms for detection of refactorings [1, 7, 12, 11, 20] assume a 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). From our case studies we noticed that mature components follow a long deprecate-replace-remove lifecycle where refactored entities coexist with the newer counterparts for a number of major releases. This lifecycle introduces enough noise that existing algorithms cannot accurately detect the refactorings. Also, these algorithms 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.

We aim for our algorithm to detect refactorings for replay, with the *minimal* involvement from developers. Therefore, the algorithm needs to detect refactorings with a high *accuracy*: 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 introduce extraneous interface changes; 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.

3. THE ROLE OF REFACTORINGS IN API EVOLUTION

To learn what types of API changes cause problems for application developers, we looked at four well known frameworks and libraries from the open source realm, namely Eclipse [eclipse.org], Struts [struts.apache.org], JHotDraw [www.jhotdraw.org], and log4j [logging.apache.org/log4j]. To check whether the production environment affects the type of API changes, we chose one more proprietary framework, Mortgage, from a large banking corporation in the Midwest. These systems were developed by five different groups. We tried to be unbiased in the selection of the case studies, the main concern being that the systems have good documentation. All the case studies (see Table 1) are mature software, namely components that have been in production for more than three years. By now they have proved themselves to be useful and therefore acquired a large customer base. At this stage, API changes have the potential to break compatibility with many older applications.

For each component we chose for comparison two major releases that span large architectural changes. There are two benefits to choosing major releases as comparison points. First, it is likely that there will be lots of changes between the two versions. Second, it is likely that those changes will be documented thus providing some starting point for a detailed analysis of the API changes.

	Eclipse 3.0	Mortgage	Struts 1.2.4	log4j 1.3	JHD 5.0
Size[KLOC]	1,923	52	97	62	14
API Classes	2,579	174	435	349	134
BreakingChanges	51	11	136	38	58
ReleaseNotes	24	-	16	4	3

Table 1: Size of the studied components. The number of API classes include only those classes that are meant to be reused. Breaking changes are changes that are not backwards compatible. ReleaseNotes give the size (in pages) of the documents describing the API changes; these change logs were provided by the component developers.

The case study components are medium-size to large (Mortgage is 50 KLOC, Eclipse is roughly 2 million LOC). It is hard to discover the changes in a large system and many authors suggest that tools should be used to detect changes. For instance, Demeyer et al. [7] describe how they used metrics tools to discover refactorings. However, most API changes follow a long deprecate-replace-remove cycle to pre-

[1]Eclipse [www.eclipse.org] is the most used open-source development environment for Java

serve backward compatibility. This means that on obsolete API can coexist with the new API for a long time. This introduces enough noise that tools might mislead us about the exact kind of change that happened.

Besides, we wanted to do a quantitative and qualitative analysis of all breaking changes in these components. Since we were the first to attempt such a venture, we were not sure what would be all types of changes encountered. For these reasons, we chose instead to do a manual analysis of the API changes. Even for the larger components, this was feasible because we started from the change logs that describe the API changes for each release. For the proprietary Mortgage framework, for two days we interviewed the framework and application developers and then studied the source code.

Sometimes the changes described in the release notes would map one-to-one with changes documented in refactoring catalogs [10]. Other times, the release documents would be vague, reading for example "we removed duplication in class X". In those cases, we read and compared the two versions of the source code in order to discover the exact type of change.

We classified all the breaking API changes from the case studies into structural and behavioral changes (qualitative analysis), then we recorded how many times each type of change occurred (quantitative analysis). We published a detailed analysis of all these changes [9]. We learned that between two versions for each of the five systems we studied, respectively 84%, 81%, 90%, 97%, and 94% of the API breaking changes are refactorings. Most API changes occur as responsibility is shifted between classes (e.g., methods or fields moved around) and collaboration protocol changes (e.g., renaming or changing method signature). These results made us believe that refactoring plays an important role as mature components evolve and therefore, we propose that refactoring operations serve as documentation for component evolution.

Probably the most important advantage of documenting component evolution in terms of refactoring operations is that refactoring operations carry deep semantics of the change along with the syntax of the changes. These semantics are discussed in refactoring catalogs [10] and are incorporated into refactoring engines. Semantics become crucially important in OO languages where programmers make heavy use of class inheritance and dynamic dispatching of method calls at runtime. For instance, a simple refactoring like renaming a public method in a class involves not only the renaming of the method declaration, but also the updating of all its call sites, the renaming of all methods in the class hierarchy that are overridden by or that override the method under discussion, as well as updating all the call sites of these methods. Before applying such an operation, the refactoring engine checks whether the new name would result in a name collision or in a potential change in method dispatch. The thorough analysis ensures that the change preserves the existing behavior of the software, thus leading to safe updates. A small extension to the refactoring engine allows it to replay the component refactorings on applications.

4. AUTOMATIC DETECTION OF REFACTORINGS

To exploit the full potential of replay of refactorings, we designed an algorithm to efficiently and accurately detect component refactorings. Figure 1 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 log = emptyRefactorings();
  foreach (DetectionStrategy strategy) {
    do {
     Refactorings log' = log.copy();
     foreach (Pair<e1, e2> from pairs relevant to strategy)
        if (strategy.isLikelyRefactoring(e1, e2, log))
          log.add(<e1, e2>, strategy);
    } while (!log'.equals(log)); // fixed point
  }
  return log;
}
```

Figure 1: Pseudo-code of the conceptual algorithm for detection of refactorings.

4.1 Syntactic Analysis

Our syntactic analysis starts by parsing the source files of the two versions of the component into the *lightweight* Abstract Syntax Trees (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.

Most refactorings involve restructuring the source code entities into *similar* (but not identical) code fragments. The heart of our syntactic analysis is the use of Shingles encoding [4] (a technique from Information Retrieval) to find similar pairs of entities 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. Users can change similarity thresholds.

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.

4.2 Semantic Analysis

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, namely RenameMethod, RenameClass, RenamePackage, MoveMethod, ChangeMethodSignature, PullUpMethod and PushDownMethod (these were the most often performed refactorings that we noticed in the case studies [9]). 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.

We use two mechanisms to detect cases when a source-code entity passed through multiple refactorings. First, the algorithm runs each strategy repeatedly until it finds no new refactorings (it reaches a fixed point). This loop is a key for detection of cases when the *same* type of refactoring happened to related entities (e.g., a method was renamed and the methods that call it were renamed too). Second, each strategy takes into account already detected refactorings; sharing detected refactorings among strategies is a key for accurate detection of refactorings when *different* types of refactorings applied to the same entity (e.g., a method was renamed and its signature takes different arguments) or related entities (e.g., a method was renamed and also its class was renamed).

The heart of each refactoring strategy is the use of *reference graphs*. The strategies compute the likelihood of refactoring based on *references* among the source-code entities (e.g., calls among methods) in each of the two versions of the component.

In addition to reference graph similarity we also used other analyses fit for OO systems, class inheritance hierarchies and method overriding hierarchies. A detailed description of the analyses can be found in our paper [8].

5. RESULTS AND CONTRIBUTIONS

This section describes the impact of our findings that refactorings play an important role in the evolution of real world components. To fully exploit this trend in component evolution, we have implemented and evaluated our algorithm for detecting component refactorings. Once detected, these refactorings can be automatically and safely incorporated into applications by replaying.

5.1 Insights into the evolution of APIs

Table 2 provides a summary of the study of API evolution. There are several implications of our findings [9]. First, they confirm that refactoring plays an important role in the evolution of components. Second, they offer a ranking of refactorings based on how often they were used in five systems. Refactoring vendors should prioritize to support the most frequently used refactorings. Third, they suggest that component producers should document the changes in each product release in terms of refactorings. Because refactorings carry rich semantics (besides the syntax of changes) they can serve as explicit documentation for both manual and automated upgrades. Fourth, migration tools should focus on support to integrate into applications those refactorings performed in the component. Our compelling results and the interaction with the developers of Eclipse led to the extension of Eclipse's refactoring engine with support for record and playback.

Component	# Breaking Changes	% Refactorings
Eclipse	51	84%
Eclipse*	99	87%
Mortgage	11	81%
Struts	136	90%
Struts*	77	100%
Log4J	38	97%
JHotDraw	58	94%

Table 2: Most of the API breaking changes are refactorings. Eclipse* and Struts* denote recommended changes, that is changes that will become enforced in next major release.

5.2 RefactoringCrawler: implementation and evaluation

We have implemented our algorithm for detecting refactorings in RefactoringCrawler, a plugin for the Eclipse development environment. RefactoringCrawler provides an efficient implementation of the algorithm shown in Figure 1. RefactoringCrawler lazily runs the expensive computation (such as finding references) and caches the intermediate results.

To measure the accuracy of our tool when detecting refactorings, we use two standard metrics from the Information Retrieval field. *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:
$RECALL = \frac{GoodResults}{GoodResults+FalseNegatives}$

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:
$PRECISION = \frac{GoodResults}{GoodResults+FalsePositives}$

Ideally, the recall and precision should be 100%. If that was the case, the refactorings found could be fed into a tool that replays the refactorings to automatically upgrade component-based applications. However, due to the intricacies introduced in order to maintain backwards compatibility of open-world components, it is hard to have 100% precision and recall. Eventually, a human expert needs to validate the results. However, the high accuracy of RefactoringCrawler reduces the human effort to a minimum.

Table 3 shows the accuracy detection of RefactoringCrawler for three real world components, ranging in size from 17 KLOC (JHotDraw) to 352 KLOC (EclipseUI is the UI subcomponent of Eclipse). In general, it is easier to spot the false positives (refactorings erroneously reported by RefactoringCrawler) by comparing the refactoring against the source code than it is to detect the false negatives (refactorings that RefactoringCrawler missed). The extensive manual analysis that we did in the first study allowed us to build a repository of refactorings that happened between the two versions. We compared these manually found refactorings against the refactorings that were found by RefactoringCrawler to determine the false negatives.

Since RefactoringCrawler combines both syntactic and semantic analysis, it can process a realistic size of software with practical precision. RefactoringCrawler ran on a Fu-

jitsu laptop with a 1.73GHz Pentium 4M CPU and 1.25GB of RAM. It took 4 min and 55 sec for detecting the refactorings in Struts, 37 sec for JHotDraw, and 16 min 38 sec for EclipseUI.

	Precision	*Recall*
EclipseUI 2.1.3 - 3.0	90%	86%
Struts 1.2.1 - 1.2.4	100%	86%
JHotDraw 5.2 - 5.3	100%	100%

Table 3: RefactoringCrawler detects refactorings with practical accuracy

Manual analysis revealed the reason why RefactoringCrawler missed a few refactorings. In Struts, for instance, methods `computeParameters` and `pageURL` are moved from class `RequestUtils` to `TagUtils`. There are quite a few calls to these methods from a test class. It appears that the test code was not refactored, and therefore it still calls the old methods (that are deprecated). This results in quite different call sites between the old and the refactored methods.

RefactoringCrawler and the evaluation results are available at `http://netfiles.uiuc.edu/dig/RefactoringCrawler` .

6. CONCLUSIONS

In order to understand the component changes that cause problems for application developers, we looked at one proprietary and three open source frameworks and one library, and studied what changed between two major releases. Then we analyzed those changes in detail and found out that in the five case studies, between 81% and 97% of the API breaking changes are structural, behavior-preserving transformations (refactorings). If these refactorings could be automatically detected, they could be automatically and safely introduced in applications.

For the purpose of detecting refactorings, 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. Once detected, refactorings can lead to safe migration of applications.

The availability of powerful migration tools will change things for the component designers as well. Without fear that they break the clients, the designers will be bolder in the kind of changes they can make to their designs. Given this new found freedom, designers won't have to carry bad design decisions made in the past. They will purge the design to be easier to understand and reuse.

7. REFERENCES

[1] G. Antoniol, M. D. Penta, and E. Merlo. An automatic approach to identify class evolution discontinuities. In *IWPSE'04: Proceedings of the 7th International Workshop on Principles of Software Evolution*, pages 31–40, 2004.

[2] I. Balaban, F. Tip, and R. Fuhrer. Refactoring support for class library migration. In *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object oriented programming systems languages and applications*, pages 265–279, New York, NY, USA, 2005. ACM Press.

[3] J. Bansiya. *Object-Oriented Application Frameworks: Problems and Perspectives*, chapter Evaluating application framework architecture structural and functional stability. 1999.

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

[5] N. Chapin, J. E. Hale, K. M. Khan, J. F. Ramil, and W.-G. Tan. Types of software evolution and software maintenance. *Journal of Software Maintenance*, 13(1):3–30, 2001.

[6] K. Chow and D. Notkin. Semi-automatic update of applications in response to library changes. In *ICSM'96: Proceedings of International Conference on Software Maintenance*, pages 359–368. IEEE Computer Society, 1996.

[7] S. Demeyer, S. Ducasse, and O. Nierstrasz. Finding refactorings via change metrics. In *OOPSLA*, pages 166–177, 2000.

[8] D. Dig, C. Comertoglu, D. Marinov, and R. Johnson. Automatic detection of refactorings in evolving components. *To appear in ECOOP'06: Proceedings of European Conference on OO Programming*, 2006.

[9] D. Dig and R. Johnson. The role of refactorings in api evolution. In *ICSM '05: Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM'05)*, pages 389–398, Washington, DC, USA, 2005. IEEE Computer Society.

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

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

[12] C. Gorg and P. 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.

[13] J. O. Graver. The evolution of an object-orietented compiler framework. *Softw., Pract. Exper.*, 22(7):519–535, 1992.

[14] J. Henkel and A. Diwan. Catchup!: capturing and replaying refactorings to support api evolution. In *ICSE'05: Proceedings of International Conference on Software Engineering*, pages 274–283, 2005.

[15] R. Keller and U. Hölzle. Binary component adaptation. *Lecture Notes in Computer Science*, 1445:307–329, 1998.

[16] B. P. Lientz and E. B. Swanson. *Software maintenance management: a study of the maintenance of computer application software in 487 data processing organizations.* Adison-Wesley, 1980.

[17] M. Mattson and J. Bosch. Three evaluation methods for object-oriented frameworks evolution - application, assessment and comparison. Technical Report 1999:20, Department of Computer Science, University of Karlskrona/Ronneby, Sweden, 1999.

[18] B. Opdyke and R. Johnson. An aid in designing application frameworks and evolving object-oriented systems. In *Proceedings of 1990 Symposium on Object-Oriented Programming Emphasizing Practical Applications (SOOPPA '90)*, 1990.

[19] S. Roock and A. Havenstein. Refactoring tags for automatic refactoring of framework dependent applications. In *XP'02: Proceedings of Extreme Programming Conference*, pages 182–185, 2002.

[20] F. V. Rysselberghe and S. 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.

[21] L. Tokuda and D. Batory. Evolving object-oriented designs with refactorings. *Automated Software Engg.*, 8(1):89–120, January 2001.