Locating Matching Method Calls by Mining Revision History Data

Benjamin Livshits
Computer Science Department
Stanford University
Stanford, USA
livshits@cs.stanford.edu

Thomas Zimmermann
Computer Science Department
Saarland University
Saarbrücken, Germany
zimmerth@cs.uni − sb.de

Abstract

Developing an appropriate fix for a software bug often requires a detailed examination of the code as well as generation of appropriate test cases. However, certain categories of bugs are usually easy to fix. In this paper we focus on bugs that can be corrected with a *one-line code change*. As it turns out, one-line source code changes very often represent bug fixes. Moreover, a significant fraction of previously known bug categories can be addressed with one-line fixes. Careless use of file manipulation routines, failing to call `free` to deallocate a data structure, failing to use `strncpy` instead of `strcpy` for safer string manipulation, and using tainted character arrays as the format argument of `fprintf` calls are all well-known types of bugs that can typically be corrected with a one-line change of the program source.

This paper proposes an analysis of software revision histories to find highly correlated pairs of method calls that naturally form application-specific useful coding patterns. Potential patterns discovered through revision history mining are passed to a runtime analysis tool that looks for pattern violations. We focus our pattern discovery efforts on *matching method pairs*. Matching pairs such as ⟨`fopen`, `fclose`⟩, ⟨`malloc`, `free`⟩, as well as ⟨`lock`, `unlock`⟩-function calls require exact matching: failing to call the second function in the pair or calling one of the two functions twice in a row is an error. We use *common bug fixes* as a heuristic that allows us to focus on patterns that caused bugs in the past. The user is presented with a choice of patterns to validate at runtime. Dynamically obtained information about which patterns were violated and which ones held at runtime is presented to the user. This combination of revision history mining and dynamic analysis techniques proves effective for both discovering new application-specific patterns *and* for finding errors when applied to very large programs with many man-years of development and debugging effort behind them.

To validate our approach, we analyzed Eclipse, a widely-used, mature Java application consisting of more than 2,900,000 lines of code. By mining revision histories, we have discovered a total of 32 previously unknown highly application-specific matching method pairs. Out of these, 10 were dynamically confirmed as valid patterns and a total of 107 previously unknown bugs were found as a result of pattern violations.

The first author was supported in part by the National Science Foundation under Grant No. 0326227. The second author was supported by the Graduiertenkolleg "Leistungsgarantien für Rechnersysteme".

1 Introduction

Much attention has lately been given to addressing application-specific software bugs such as errors in operating system drivers [1, 4], security errors [5, 8], and errors in reliability-critical embedded software in domains such as avionics [2, 3]. These represent critical errors in widely used software and tend to get fixed relatively quickly when found. A variety of static and dynamic analysis tools have been developed to address these high-profile bugs. However, many other errors are specific to individual applications or sets of APIs.

Repeated violations of these application-specific coding rules, referred to as *error patterns*, are responsible for a multitude of errors. Error patterns tend to be re-introduced into the code over and over by multiple developers working on the project and are a common source of software defects. While each pattern may be only responsible for a few bugs, taken together, the detrimental effect of these error patterns is quite serious and they can hardly be ignored in the long term.

In this paper we propose an automatic way to extract likely error patterns by mining software revision histories. Moreover, to ensure that all the errors we find are relatively easy to confirm and fix, we limit our experiments to errors that can be corrected with a one-line change. When reporting a bug to the development team, having a bug that is easy to fix usually improves the chances that it will actually be corrected. It is worth noticing that many well-known error patterns such as memory leaks, double-`free`'s, mismatched locks, operations on operating system resources, buffer overruns, and format string errors can often be addressed with a one-line fix. Our approach uses revision history information to infer likely error patterns. We then experimentally evaluate the error patterns we extract by checking for their violations dynamically.

We have performed our experiments on Eclipse, a very large, widely-used open-source Java application with many man-years of software development behind it. By mining CVS revision histories of Eclipse, we have identified 32 high-probability patterns in Eclipse APIs, all of which were previously unknown to us. While the user of our system has a choice as to which patterns to check at runtime, in our experiments we limited ourselves to patterns that were frequently encountered in our revision history data. Out of these, 10 were dynamically confirmed as valid patterns and 107 bugs were found as a result of pattern violations.

1.1 Contributions

This paper makes the following contributions:

- We propose a *data mining strategy* that detects common matching method pairs patterns in large software systems by analyzing software revision histories. We also propose an effective ranking technique for the patterns we discover that uses one-line changes to favor previously fixed bugs.
- We propose a *dynamic analysis approach* for validating usage patterns and finding their violations. We currently utilize an off-line approach that simplifies the matching machinery. Error checkers are implemented as simple pattern-matchers on the resulting dynamic traces.
- We present an *experimental study* of our techniques as applied to finding errors in Eclipse, a large, mature Java application with many years of development behind it. We identified 32 patterns in Eclipse sources and out of these, 10 patterns were experimentally confirmed as valid. We found more that 107 bugs representing violations of these pattern with our dynamic analysis approach.

2 Revision History Mining

To motivate our approach to revision history mining, we start by presenting a series of observations.

Observation 2.1 *Given multiple software components that use the same set of APIs, it should be possible to find* common errors *specific to that API.*

In fact, much of the research done on bug detection so far can be thought of as focusing on specific classes of bugs pertaining to particular APIs: studies of operating-system bugs provide synthesized lists of API violations specific to operating system drivers resulting in rules such as "do not call the interrupt disabling function `cli()` twice in a row" [4].

Observation 2.2 *Method calls that are frequently added to source code simultaneously often represent a usage pattern.*

In order to locate common errors, we mine for frequent usage patterns in revision histories. Looking at incremental changes between revisions as opposed to full snapshots of the sources allows us to better focus our mining strategy. However, it is important to notice that not every pattern *mined* by considering revision histories is an actual *usage* pattern. Figure 1 lists sample method calls that were added to revisions of files `Foo.java`, `Bar.java`, `Baz.java`, and `Qux.java`. All these files contain a usage pattern that says that methods {`addListener`, `removeListener`} must be precisely matched. However, mining these revisions yields additional patterns like {`addListener`, `println`} and {`addListener`, `iterator`} that are definitely *not* usage patterns.

File	Revision	Added method calls
Foo.java	1.12	o1.addListener o1.removeListener
Bar.java	1.47	o2.addListener o2.removeListener System.out.println
Baz.java	1.23	o3.addListener o3.removeListener list.iterator iter.hasNext iter.next
Qux.java	1.41 1.42	o4.addListener o4.removeListener

Figure 1: Method calls added across different revisions.

```
SELECT l.Callee AS CalleeL, r.Callee AS CalleeR,
     COUNT(*) AS SupportCount
     INTO pairs FROM items l, items r
WHERE l.FileId = r.FileId
     AND l.RevisionId = r.RevisionId
     AND l.InitialCallSequence = r.InitialCallSequence
GROUP BY l.Callee, r.Callee;
```

Figure 2: Find correlated pairs ⟨`CalleeL`, `CalleeR`⟩ of methods sharing the initial call sequence, calls to which are added in the same revision of a file identified by `FileId` and `RevisionId`.

2.1 Mining Approach

In order to speed-up the mining process, we pre-process the revision history extracted from CVS and store this information in a general-purpose database; our techniques are further described in Zimmermann et al. [11]. This database contains method calls that have been inserted in each revision. To determine the calls inserted between two revisions r_1 and r_2, we build abstract syntax trees (ASTs) for both r_1 and r_2 and compute the set of all calls C_1 and C_2, respectively, by traversing the ASTs. $C_2 \setminus C_1$ is the set of calls inserted between r_1 and r_2.

After the revision history database is set-up, i.e., all calls that were added are recorded in the `items` table, mining is performed using SQL queries. The query in Figure 2 produces *support counts* for each method pair, which is the number of times the two methods were added to revision history together. We perform filtering based on support counts to only consider method pairs that have a sufficiently high support.

2.2 Pattern Ranking

Filtering the patterns based on their support count is not enough to eliminate unlikely patterns. To better target user efforts, a ranking of our results is provided. In addition to sorting patterns by their support count, a common ranking strategy in data mining is to look at the pattern's *confidence*. The confidence determines how strongly a particular pair of methods is correlated and is computed as

$$conf(\langle a,b\rangle) = \frac{support(\langle a,b\rangle)}{support(\langle a,a\rangle)}$$

Both support count and confidence are standard ranking approaches used in data mining; however, using problem-specific

```
SELECT DISTINCT Callee
INTO fixes
FROM (SELECT MIN(Callee)
        FROM items GROUP BY FileId, RevisionId
        HAVING COUNT(*) = 1) t;
```

Figure 3: Find "fixed" methods, calls to which were added in at least one *one-call* check-ins. A one-call check-in adds exactly one call, as indicated by `COUNT(*) = 1`.

```
SELECT p.CalleeL, p.calleeR, SupportCount
FROM pairs p, fixes l, fixes r
WHERE p.calleeL = l.Callee
   AND p.calleeR = r.Callee
```

Figure 4: Find pairs from the table `pairs`, where both methods had previously been fixed, i.e., are contained in the table `fixed`.

knowledge yields a significantly better ranking of results. We leverage the fact that in reality some patterns may be inserted incompletely, e.g., by mistake or to fix a previous error. In Figure 1 this occurs in file `Qux.java`, where `addListener` and `removeListener` were inserted independently in revisions 1.41 and 1.42. The observation below motivates a novel ranking strategy we use.

Observation 2.3 *Small changes to the repository such as one-line additions are often bug fixes.*

This observation is supported in part by anecdotal evidence and also by recent research into the nature of software changes. A recent study of the dynamic of small repository changes in large software systems performed by Purushothaman et al. sheds a new light on this subject [6]. Their paper points out that almost 50% of all repository changes were small, involving less than 10 lines of code. Moreover, among one-line changes, less than 4% were likely to cause a later error. Furthermore, only less than 2.5% of all one-line changes were *perfective* or adding functionality (rather than *corrective*) changes, which implies that most one-line check-ins are bug fixes.

We use this observation by marking method calls that are frequently added in one-line fixes as *corrective* and ranking patterns by the number of corrective calls they contain. The SQL query in Figure 3 creates table `fixes` with all corrective methods, calls to which were added as one-line check-ins. Finally, as shown in Figure 4, we find all method pairs where *both* methods are corrective. We favor these patterns by ranking them higher than other patterns.

3 Dynamic Analysis

Similarly to previous efforts that looked at detecting usage rules [9, 10], we use revision history mining to find common coding patterns. However, we combine revision history mining with a bug-detection approach. Moreover, our technique looks for pattern violations at runtime, as opposed to using a static analysis technique. This choice is justified by several considerations outlined below.

Scalability. Our original motivation was to be able to analyze Eclipse, which is one of the largest Java applications ever created. The code base of Eclipse contains of more than 2,900,000 lines of code and 31,500 classes. Most of the patterns we are interested in are spread across multiple methods and need a precise interprocedural approach to analyze. Given the substantial size of the application, precise whole-program flow-sensitive static analysis can be prohibitively expensive.

Validating discovered patterns. A benefit of using dynamic analysis is that we are able to "validate" the patterns we discover through CVS history mining as real usage patterns by observing how many times they occur at runtime. While dynamic analysis is unable to prove properties that hold in all executions, patterns that are matched many times with only a few violations represent likely patterns. With static analysis, validating patterns would not generally be possible unless flow-sensitive *must* information is available.

False positives. Runtime analysis does not suffer from false positives because all pattern violations detected with our system actually *do* occur. This significantly simplifies the process of error reporting and addresses the issue of false positives.

Automatic error repair. Finally, only dynamic analysis provides the opportunity to fix the problem on the fly without any user intervention. This is especially appropriate in the case of matching method pair when the second method call is missing. While we have not implemented automatic "pattern repair", we believe it to be a fruitful future research direction.

While we believe that dynamic analysis is more appropriate than static analysis for the problem at hand, dynamic analysis has a few well-known problems of its own. A serious shortcoming of dynamic analysis is its lack of coverage. In our dynamic experiments, we managed to find runtime use cases for some, but not all patterns discovered through revision history mining. Furthermore, the fact that a certain pattern appears to be a strong usage pattern based on dynamic data should be treated as a a suggestion, but not a proof.

We use an offline dynamic analysis approach that instruments all calls to methods that may be included in the method pairs of interest. We then post-process the resulting dynamic trace to find properly matched or mismatched method pairs.

4 Preliminary Experience

In this section we discuss our practical experience of applying our system to Eclipse. Figure 5 lists matching pairs of methods discovered with our mining technique.[1] A quick glance at the table reveals that many pairs follow a specific naming strategy such as `pre`–`post`, `add`–`remove`, and `begin`–`end`. These pairs could have been discovered by simply pattern matching on the method names. However, a large number of pairs have less than obvious names to look for, including ⟨`HLock`, `HUnlock`⟩, ⟨`progressStart`, `progressEnd`⟩, and ⟨`blockSignal`, `unblockSignal`⟩. Finally, some pairs are very difficult to recognize as matching method pairs and require a detailed study of the API to confirm, such as ⟨`stopMeasuring`, `commitMeasurements`⟩, ⟨`suspend`, `resume`⟩, etc. Many more potentially interesting matching pair patterns become available if we consider lower support counts; for the experiments we have only considered patterns with a support of five or more.

We also found some unexpected matching method pairs consisting of a constructor call followed by a method call that at first we thought ware caused by noise in the data. One such pair is ⟨`OpenEvent`, `fireOpen`⟩. This pattern indicates that all objects of type `OpenEvent` should be "consumed" by passing them into method `fireOpen`. Violations of this pattern may lead to resource and memory leaks, a serious problem in long-running Java program such as Eclipse, which may be open at a developer's desktop for days at a time [7].

[1]The methods in a pair are listed in the order they are supposed to be executed. For brevity, we only list unqualified method names.

METHOD PAIR $\langle a, b\rangle$		CONFIDENCE			SUPPORT
Method a	**Method** b	$conf_{ab} \times conf_{ba}$	$conf_{ab}$	$conf_{ba}$	$count$
CORRECTIVE RANKING					
`NewRgn`	`DisposeRgn`	0.75	0.92	0.22	49
`kEventControlActivate`	`kEventControlDeactivate`	0.68	0.83	0.83	5
`addDebugEventListener`	`removeDebugEventListener`	0.61	0.85	0.72	23
`beginTask`	`done`	0.59	0.74	0.81	493
`beginRule`	`endRule`	0.59	0.80	0.74	32
`suspend`	`resume`	0.58	0.83	0.71	5
`NewPtr`	`DisposePtr`	0.57	0.82	0.70	23
`addListener`	`removeListener`	0.56	0.68	0.83	90
`register`	`deregister`	0.53	0.69	0.78	40
`malloc`	`free`	0.46	0.68	0.68	28
`addElementChangedListener`	`removeElementChangedListener`	0.41	0.73	0.57	8
`addResourceChangeListener`	`removeResourceChangeListener`	0.41	0.90	0.46	26
`addPropertyChangeListener`	`removePropertyChangeListener`	0.39	0.54	0.73	140
`start`	`stop`	0.38	0.59	0.65	32
`addDocumentListener`	`removeDocumentListener`	0.35	0.64	0.56	29
`addSyncSetChangedListener`	`removeSyncSetChangedListener`	0.34	0.62	0.56	24
REGULAR RANKING					
`createPropertyList`	`reapPropertyList`	1.00	1.00	1.00	174
`preReplaceChild`	`postReplaceChild`	1.00	1.00	1.00	133
`preLazyInit`	`postLazyInit`	1.00	1.00	1.00	112
`preValueChange`	`postValueChange`	1.00	1.00	1.00	46
`addWidget`	`removeWidget`	1.00	1.00	1.00	35
`stopMeasuring`	`commitMeasurements`	1.00	1.00	1.00	15
`blockSignal`	`unblockSignal`	1.00	1.00	1.00	13
`HLock`	`HUnlock`	1.00	1.00	1.00	9
`addInputChangedListener`	`removeInputChangedListener`	1.00	1.00	1.00	9
`preAddChildEvent`	`postRemoveChildEvent`	1.00	1.00	1.00	8
`preRemoveChildEvent`	`postAddChildEvent`	1.00	1.00	1.00	8
`progressStart`	`progressEnd`	1.00	1.00	1.00	8
`CGContextSaveGState`	`CGContextRestoreGState`	1.00	1.00	1.00	7
`annotationAdded`	`annotationRemoved`	1.00	1.00	1.00	7
`OpenEvent`	`fireOpen`	1.00	1.00	1.00	7
`addInsert`	`addDelete`	1.00	1.00	1.00	7

Figure 5: Matching method pairs discovered through CVS history mining. The support count is $count$, the confidence for $\{a\} \Rightarrow \{b\}$ is $conf_{ab}$, for $\{b\} \Rightarrow \{a\}$ it is $conf_{ba}$. The pairs are ordered by $conf_{ab} \times conf_{ba}$.

4.1 Experimental Setup

In this section we describe our revision history mining and dynamic analysis setup. When we performed the pre-processing on Eclipse, it took about four days to fetch all revisions over the Internet because the complete revision data is about 6GB in size and the CVS protocol is not well-suited for retrieving large volumes of history. Computing inserted methods by analyzing the ASTs and storing this information in a database took about a day.

Figure 6 summarizes our dynamic results. Because the incremental cost of checking for additional patterns at runtime is generally low, when reviewing the patterns for inclusion in our dynamic experiments, we were fairly liberal in our selection. We usually either looked at the method names involved in the pattern or briefly examined a few usage cases. We believe our setup to be quite realistic, as we cannot expect the user to spend hours poring over the patterns. Overall, 50% of all patterns we chose turned out to be either usage or error patterns; had we been more selective, a higher percentage of patterns would have been confirmed dynamically. To obtain dynamic results, we ran each application for a few minutes, which typically resulted in a few hundred or thousand dynamic events being generated.

After we obtained the dynamic results, the issue of how to count errors arose. A single pattern violation at runtime involves one or more objects. We can obtain a *dynamic count* by counting how many different objects participated in a particular pattern violation during program execution. The dynamic error count is highly dependent on how we use the program at runtime and can be easily influenced by, for example, rebuilding a project in Eclipse multiple times. However, dynamic counts are not representative of the work a developer has to do to fix an error, as many dynamic violations can be caused by the same error in the code. To provide a better metric on the number of errors found in the application code, we also compute a *static count*. This is done by mapping each method participating in a pattern to a static call site and counting the number of unique call site combinations that are seen at runtime. Static counts are obtained for both validated and violated dynamic patterns.

Dynamic and static counts are shown in parts 2 and 3 of the table, respectively. The rightmost section of the table shows a classification of the patterns. We use information about how many times each pattern is validated and how many times it is violated to classify the patterns. Let v be the number of validated instances of a pattern and e be the number of its violations. We define an error threshold $\alpha = min(v/10, 100)$. Based on the validation and violation counts v and e, patterns can be loosely classified into the following categories:

- **Likely usage patterns**: patterns with a sufficiently high support that are mostly validated with relatively few errors ($e < \alpha \wedge v > 5$).
- **Likely error patterns**: patterns that have a significant number of validated cases as well as a large number of violations ($\alpha \leq e \leq 2v$).
- **Unlikely patterns**: patterns that do not have many vali-

METHOD PAIR $\langle a, b \rangle$		DYNAMIC EVENTS		STATIC EVENTS		PATTERN TYPE		
Method a	**Method b**	**Validated**	**Errors**	**Validated**	**Errors**	**Usage**	**Error**	**Unlikely**
CORRECTIVE RANKING								
`addDebugEventListener`	`removeDebugEventListener`	4	1	4	1			✓
`beginTask`	`done`	334	642	42	21		✓	
`beginRule`	`endRule`	7	0	4	0	✓		
`addListener`	`removeListener`	118	106	35	26		✓	
`register`	`deregister`	1,279	313	6	7		✓	
`addResourceChangeListener`	`removeResourceChangeListener`	25	4	19	4			✓
`addPropertyChangeListener`	`removePropertyChangeListener`	1,789	478	55	25		✓	
`start`	`stop`	40	36	9	12		✓	
`addDocumentListener`	`removeDocumentListener`	39	1	14	1	✓		
Result subtotals for the corrective ranking scheme:		**3,635**	**1,581**	**188**	**97**	**2**	**5**	**2**
REGULAR RANKING								
`preReplaceChild`	`postReplaceChild`	40	0	26	0	✓		
`preValueChange`	`postValueChange`	63	2	11	2	✓		
`addWidget`	`removeWidget`	1,264	16	5	2	✓		
`preRemoveChildEvent`	`postAddChildEvent`	0	172	0	3			✓
`annotationAdded`	`annotationRemoved`	0	8	0	2			✓
`OpenEvent`	`fireOpen`	0	3	0	1			✓
Result subtotals for the regular ranking scheme:		**1,367**	**201**	**42**	**10**	**3**	**0**	**3**
OVERALL TOTALS:		**5,002**	**1,782**	**230**	**107**	**5**	**5**	**5**

Figure 6: Result summary for the validated usage and error patterns in Eclipse.

dated cases or cause too many errors to be usage patterns ($e > 2v \vee v \leq 5$).

About a half of all method pair patterns that we selected from the filtered mined results were confirmed as likely patterns, out of those 5 were usage patterns and 5 were error patterns.

Overall, corrective ranking was significantly more effective than regular ranking schemes that are based on the product of confidence values. The top half of the table that addresses patterns obtained with corrective ranking contains 16 matching method pairs; so does the second half that deals with the patterns obtained with regular ranking. Looking at the subtotals for each ranking scheme reveals 188 static validating instances with corrective ranking vs only 42 for regular ranking; 97 static error instances are found vs only 10 for regular ranking. Finally, 7 patterns found with corrective ranking were dynamically confirmed as either error or usage patterns vs 3 for regular ranking. This confirms our belief that corrective ranking is more effective.

5 Conclusions

In this paper we presented an approach for discovering matching method pair patterns in large software systems and finding their violations at runtime. Our framework uses information obtained by mining software revision repositories in order to find good patterns to check. User input may be used to further restrict the number of checked patterns. Checking of patterns occurs during program execution, with the help of dynamic instrumentation.

We experimentally evaluated our system on Eclipse, a very large Java application totalling more than 2,900,000 lines of code shows that our approach is highly effective at finding a variety of previously unknown patterns. Overall, we discovered a total of 32 matching method pairs in our benchmarks. Out of these, 5 turned out to be dynamically confirmed usage patterns and 5 were frequently misused error patterns responsible for many of the bugs. Our ranking approach that favors corrective ranking overperformed the traditional data mining ranking strategies at identifying good patterns. In our experiments, 1,782 dynamic pattern violations were responsible for a total of 107 dynamically confirmed errors in the source code.

References

[1] T. Ball, B. Cook, V. Levin, and S. K. Rajamani. SLAM and static driver verifier: Technology transfer of formal methods inside Microsoft. Technical Report MSR-TR-2004-08, Microsoft, 2004.

[2] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. A static analyzer for large safety-critical software. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation (PLDI'03)*, pages 196–207, June 7–14 2003.

[3] G. Brat and A. Venet. Precise and scalable static program analysis of NASA flight software. In *Proceedings of the 2005 IEEE Aerospace Conference*, 2005.

[4] D. Engler, B. Chelf, A. Chou, and S. Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *Proceedings of the Fourth Symposium on Operating Systems Design and Implentation*, pages 1–16, 2000.

[5] Y.-W. Huang, F. Yu, C. Hang, C.-H. Tsai, D.-T. Lee, and S.-Y. Kuo. Securing Web application code by static analysis and runtime protection. In *Proceedings of the 13th conference on World Wide Web*, pages 40–52, May 2004.

[6] R. Purushothaman and D. E. Perry. Towards understanding the rhetoric of small changes. In *Proc. International Workshop on Mining Software Repositories (MSR 2004)*, pages 90–94, May 2004.

[7] B. A. Tate. *Bitter Java*. Manning Publications Co., 2002.

[8] D. Wagner, J. Foster, E. Brewer, and A. Aiken. A first step towards automated detection of buffer overrun vulnerabilities. In *Proceedings of Network and Distributed Systems Security Symposium*, pages 3–17, San Diego, California, Feb. 2000.

[9] C. Williams and J. K. Hollingsworth. Bug driven bug finders. In *Proceedings of the International Workshop on Mining Software Repositories*, pages 70–74, May 2004.

[10] C. Williams and J. K. Hollingsworth. Recovering system specific rules from software repositories. In *Proceedings of the International Workshop on Mining Software Repositories*, pages 7–11, May 2005.

[11] T. Zimmermann and P. Weißgerber. Preprocessing CVS data for fine-grained analysis. In *Proc. International Workshop on Mining Software Repositories (MSR 2004)*, pages 2–6, May 2004.