

Recommending Method Invocation Context Changes

Beat Fluri, Jonas Zuberbühler, and Harald C. Gall
s.e.a.l. – software evolution and architecture lab
Department of Informatics, University of Zurich, Switzerland
{fluri,zubi,gall}@ifi.uzh.ch

ABSTRACT

Our investigations of bug fixes in Eclipse showed that a significant amount of bugs were fixed by moving invocations of certain methods into the then or else-part of if-statements with similar conditions. Based on this finding, we leverage such context changes applied in the past to support developers while adding invocations of the same method. In this paper we present CHANGECOMMANDER, an Eclipse plugin that implements our approach to recommend insertions of particular if-statements before calling a method. CHANGECOMMANDER presents context change suggestions by highlighting affected method invocations in the source code and provides automated code adaptation support.

Categories and Subject Descriptors: D.2.7 [Software Engineering]: Distribution, Maintenance, and Enhancement

Keywords: Software evolution, source code change types, change patterns, context changes, change recommendations

1. INTRODUCTION

Brook's *essential complexities* in software engineering inhibit the development of bug-free software [2]. Bugs are continuously fixed, but because fixing bugs induces software changes, each fix has a substantial chance to introduce a new bug. Finding and fixing bugs before the software is delivered to customers is of great value because fixing bugs after delivery is ten times as expensive as fixing it before [5]. As a result, approaches and techniques that find bugs automatically as soon as possible and suggest solutions to fix bugs are crucial.

In the area of software evolution analysis, approaches exist that find bugs by mining error patterns in source code. For instance, Livshits and Zimmermann applied mining to software repositories to find errors in method usage patterns [13]. Kim *et al.* used their *BugMem* approach to find general error patterns by investigating changes that fixed bugs [12]. The major advantage of such approaches is that they are vertical. That means, they find project-specific er-

ror patterns. Horizontal approaches, such as *FindBugs* [10], find error patterns across all projects but are limited to predefined patterns.

Although *BugMem* can find a variety of error patterns, it fails at identifying *context changes* of method invocations that fixed a bug. We define a context change of method invocation as moving an existing method invocation into the then or the else-part of an if-statement. As our change distilling algorithm [8] can extract such context changes, we can extract corresponding bug fixes and overcome this limitation.

We observed that in Eclipse a significant number of bugs (about 15 percent) are fixed by context changes of method invocations (see Section 3). For instance, we found the following context change:

Old version:	New version:
...	...
<code>visited.add(outputFolder);</code>	<code>if (!visited.contains(</code>
...	<code>outputFolder)) {</code>
	<code>visited.add(outputFolder);</code>
	...

between the Revisions 1.16 and 1.17 of the method `org.eclipse.jdt.internal.core.builder.BatchImageBuilder.cleanOutputFolders()`.

These changes also happen even if they did not fix bugs, *i.e.*, they are also applied during preventive, perfective, or adaptive maintenance. In addition, we can extract patterns among these context changes, *i.e.*, invocations of the same method are moved into the then or else-part of if-statements that have similar conditions.

In this paper, we present a recommender approach that leverages context change data to suggest corresponding modifications when a developer adds a method invocation. We propose CHANGECOMMANDER, an Eclipse plugin that enables the automated application of the context change recommendations. It is integrated into the build process of Eclipse and provides visual feedback in the source code. By integrating CHANGECOMMANDER into the development process, we aim at reducing future bugs.

The remainder of this paper is structured as follows: In Section 2 we introduce EVOLIZER as well as CHANGEDIS-TILLER and illustrate how source code changes from a version control repository are obtained. We motivate our recommendation approach in Section 3 and describe the corresponding data preparation in Section 4. In Section 5 we present our recommendation approach realized as CHANGECOMMANDER. We sketch ongoing and future work in Sec-

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.

RSSE '08, November 10, Atlanta, Georgia, USA
Copyright 2008 ACM 978-1-60558-228-3 ...\$5.00.

tion 6, review related work in Section 7, and conclude the paper with Section 8.

2. EVOLIZER AND CHANGEDISTILLER

To recommend context changes on method invocations we leverage data provided by our EVOLIZER and CHANGEDISTILLER [8]. We briefly describe them in this section.

EVOLIZER basically stems from the idea of having a Release History Database (RHDB) [6] that integrates information originating from various repositories, such as CVS and Bugzilla, into a single database. In particular, EVOLIZER is a set of Eclipse plugins and comparable with *Kenyon* [1] or *eROSE* [19].

When importing a version control repository, EVOLIZER parses the log output of the repository, stores all information provided by the log output, *i.e.*, file name, revision number, author, commit message, commit date, etc., and stores this information along with the complete file revision content in our EVOLIZER RHDB. EVOLIZER provides an API to access the RHDB.

CHANGEDISTILLER is an implementation of our *change distilling* algorithm (presented in full detail in [8]) and is also integrated into the Eclipse IDE as a plugin. The aim of CHANGEDISTILLER is to extract fine-grained source code changes applied on subsequent revisions of Java classes which are fetched from the RHDB.

The change distilling algorithm is a tree differencing algorithm customized to be applicable on pairs of abstract syntax trees (AST). For that, the algorithm first finds a matching set between the nodes of the two ASTs. Finding a match between two AST nodes is based on string similarity measures for leaves and tree similarity measures for subtrees.

Second, the algorithm generates an edit script, *i.e.*, a set of atomic changes, that transforms one AST into the other. An atomic change is one of the basic tree edit operations *insert*, *delete*, *move*, or *update* applied to an AST node. After generating the edit script, each operation is assigned to a *change type* according to our *taxonomy of source code changes* [7]. For instance, the tree edit operation for the change type *statement parent change* is the move operation of a statement. That means, a statement is moved to a particular position in the method body. We have defined over 40 different change types on body and declaration parts of attributes, classes, and methods. The most fine-grained level of the taxonomy is the statement level.

Leveraging the information provided by ASTs permits us to get exact information about a source code change. In addition to the information that a particular source code entity has changed, tree edit operations also provide information about the location of the change. For instance, we can tell that the method invocation `foo.bar()` was moved from the then-part to the else-part of the if-statement that has the condition `foo == null`.

3. MOTIVATION

We performed a study of changes and bug fixes for the Eclipse software system and made several bug fix observations:

OBSERVATION 3.1 *25 percent of bugs are fixed with fewer than four source code changes.*

In Eclipse, over 50 percent of bugs only affect one Java file; 25 percent are even fixed with fewer than four source code changes (instances of change types). For instance, they were fixed with two *statement inserts* and one *control structure condition expression change*. This observation is supported by the finding that small changes, *i.e.*, single line changes, often represent bug fixes [13,15]. While extracting the source code changes that fixed bugs and analyzing their change types we made the next observation.

OBSERVATION 3.2 *Context changes are responsible for fixing a significant amount of bugs.*

Our change distilling algorithm extracts over 120¹ different change types. Context changes are responsible for fixing 15 percent of all bug fixes. In particular, the fix is moving a single method invocation inside the then or the else-part of a newly added if-statement. By looking at the detailed change type information we found commonalities between the changes and the involved method invocations. The following observation provides the basis for our recommendation approach.

OBSERVATION 3.3 *The contexts of invocations of a method tend to be changed similarly.*

The contexts of a set of `List.add(..)` invocations change similarly if these invocations are moved into the then or the else-part of an if-statement that has a similar condition. Such a condition can be `!<qualifier>.contains(<argument>)` or `<argument> instanceof <Type>`.

Patterns of context changes that were applied in the past allow us to recommend context adaptations for new and existing method invocations during development. The basic idea is to check for each invocation of a method whether patterns of context changes to invocations of the same method exist and whether a context change is appropriate. Assuming, a significant number of context changes exist that move invocations of `List.add(..)` into if-statements with the condition `!<qualifier>.contains(<argument>)`. If a new `List.add(..)` invocation is inserted into the source code, recommending a corresponding context change is appropriate. Next, we describe the collection and aggregation of context changes from the history of a software system to generate such recommendations.

4. PREPARING RECOMMENDATIONS

We use EVOLIZER and CHANGEDISTILLER to extract the method invocation context changes from the history of a software system and store them in the EVOLIZER RHDB. The changes are then post-processed: (1) to classify the changes into bug fixes and normal changes, (2) to aggregate changes of invocations of the same method, and (3) to extract patterns among them. Next, we briefly describe the three post-processing steps.

4.1 Bug Fixes and Normal Changes

We first parse the commit message of each revision to find indications for a bug fix. We use the strategy developed by Śliwerski *et al.* [17]. Briefly, the strategy calculates two levels of confidence. First, at the syntactic level, the commit

¹When distinguishing all possible statement kinds. For instance, the change type *statement update* can be further split into *method invocation statement update*, *assignment statement update*, etc.

message is split into tokens and matched against regular expressions, such as bug numbers or bug related keywords. Second, at the semantic level, the link established from the syntactic analysis is validated. Factors such as the resolution, *e.g.*, `FIXED`, or the similarity between the commit message and the bug description are taken into account. When a link between the revision and a bug report could be established, all changes that correspond to the revision are marked as a bug fix.

4.2 Aggregation of Changes

After collecting context changes of all method invocations of the history of a software system, we aggregate the changes of invocations of the same method. For that, we resolve the method signature of each method invocation. Our extraction and collection approach is revision-based, meaning it does not load the source model of the complete project for each processing step. Because of that, we deal with incomplete source models and, therefore, we use `ZBINDER` to resolve the method signatures. `ZBINDER` uses information provided by the Java Development Tools (JDT) from Eclipse and a set of heuristics to reconstruct missing type information [14]. Dagenais and Robillard also dealt with incomplete source code models for recommending adaptive framework changes and used a similar approach [4].

4.3 Extraction of Change Patterns

A pattern among context changes appears when similarities between the single changes can be extracted. For instance, when a certain method invocation is often moved into a then-part of an if-statement that has the condition `!<qualifier>.contains(<argument>)`, the corresponding context changes form a pattern.

We distinguish between context changes in which (1) the qualifier appears in the if-condition, (2) the arguments of the method invocation appear in the if-condition, or (3) both, qualifier and arguments, appear in the if-condition. If the if-condition is composed out of more than one expression by `AND (&&)` or `OR (||)`, we first split the if-condition into single expressions.

5. CHANGECOMMANDER

Based on the observations described in Section 3 we are developing a recommender approach that suggests method invocation context changes during development. We describe the approach based on our `CHANGECOMMANDER` (see Figure 1). Currently, our approach is implemented as an Eclipse plugin. The approach is not limited to Eclipse. It can be integrated into other IDEs that provide AST leveraging functionalities and corresponding support to interact with a developer.

`CHANGECOMMANDER` makes use of various functionalities provided by the Eclipse platform and leverages the data we stored in the `EVOLIZER RHDB`. In the remainder of this section we describe the integration of `CHANGECOMMANDER` into Eclipse. We also present the preparation of the recommendations, how `CHANGECOMMANDER` gives feedback to developers, and how it provides automated adaptation support.

5.1 Build-Process Integration

The activation of the processing logic of `CHANGECOMMANDER` to generate recommendations is realized in two

ways: (1) By the implementation of an incremental project builder and (2) by observing and processing source code changes during development. The builder is added to the end of the existing chain of builders for a project and is invoked automatically after a modified source file has been saved. It then processes the whole file at once. The developer can additionally choose whether to receive instant feedback on recommendations for newly added or modified method invocations similar to Java warnings that are generated automatically on-the-fly. This allows for extracting change suggestions incrementally on modified source files and providing corresponding user feedback instantly (see Section 5.3). It also allows for triggering a complete build on all project resources.

5.2 Recommendation Filtering

The set of context changes of invocations of a particular method has to be filtered before recommendations are made to the developer. We propose a two-step filtering approach. First, the recommendations are ranked according to their relevance and the top ranked are selected whose number are configured by the developer. Second, recommendations that are not applicable to the method invocation at hand are filtered out. In addition, we plan to enable developer-specific selection of methods for which recommendations are provided.

5.2.1 Relevance ranking

Context changes that fixed bugs are more relevant than normal changes because they reflect that a missing if-statement induced a bug. On the other hand, context changes that did not fix bugs but are frequently applied are also relevant. That means, we need a relevance ranking that takes the frequency of a certain context change and its bug fix percentage into account.

We apply the following ranking scheme: First, we normalize the frequency of a certain context change with the frequency of the maximum occurring context change of the same method. This term is then weighted with the bug fix percentage of the context change which results in a score for the context change. A similar technique is used by Kim and Ernst to prioritize warnings that are generated by bug-finding tools [11]. Finally, the scores are ranked in descending order so that the highest score gets the highest rank.

For instance, assuming that the context change t “move the method invocation into an if-statement that has the condition `!<argument>.equals(String.CONSTANT)`” was applied 10 times to the invocations of the method `List.add(java.lang.Object)`; 80% of these changes were bug fixes. Assuming further that the maximum occurrence of any context change found for `List.add(java.lang.Object)` is 20. Then, the context change t receives a relevance score $s = \frac{10}{20} \cdot 0.8 = 0.4$.

5.2.2 Filtering non-applicable recommendations

Not all generated recommendations are applicable to the method invocation under inspection. Whenever arguments of the inspected method invocation appear in the condition of the context change pattern, their types have to be checked. Assume a developer is adding the method invocation `list.add(o)`. `CHANGECOMMANDER` now finds a context change that calls the method inside the if-statement that has the condition

```
!<argument>.equals(a) && <argument>.isReady().
```

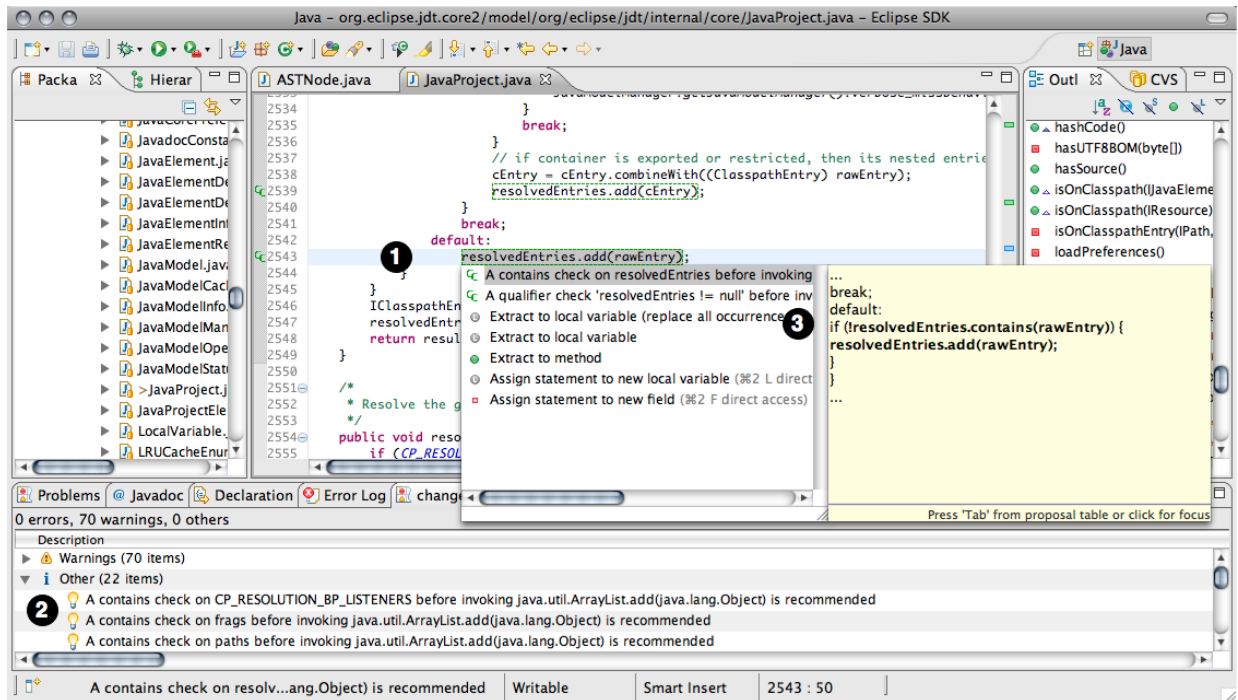


Figure 1: ChangeCommander in action

Before suggesting the context change `CHANGECOMMANDER` checks whether (1) `a` corresponds to the formal parameter type of `equals(...)` and (2) the type of `o` provides the method `isReady()`. If one of the two constraints is not valid `CHANGECOMMANDER` does not suggest the context change.

Whenever the method invocation is executed inside an if-statement which already performs the recommended condition check, the corresponding context change is not recommended.

5.3 Feedback to Developers

Our goal is to provide instant feedback on suitable recommendations by means of using visual representations familiar to developers and supported by the majority of IDEs. We extend the source code annotations of the Eclipse Java editor by highlighting method invocations for which context changes are found. In the Java editor the corresponding line is marked and the invocation itself is emphasized (see (1) in Figure 1). This approach integrates seamlessly with the existing views provided by Eclipse since annotations added by `CHANGECOMMANDER` are additionally listed in the problems view (see (2) in Figure 1). To apply a recommended context change, we present a list of quick fixes to the developer (see (3) in Figure 1). The order of the list corresponds to the relevance ranking. The modifications to the AST needed for applying a context change are performed instantly upon the selection of a quick fix.

6. ONGOING AND FUTURE WORK

The build-process integration and the feedback for the developers are implemented. Currently, we are testing and refining our two-step filtering approach. The implementation of the automated execution of the recommended context changes via quick fixes is not yet finished.

Future work. As `CHANGEDISTILLER` is able to extract many more different change types than context changes, we are investigating change pattern among other change types. For instance, method invocations themselves also tend to be updated similarly: In the history of Eclipse, we observed that string constants used as arguments of particular methods are similarly changed. First, they were replaced by static fields. Then, they were replaced by the `Message`-construct used throughout Eclipse. That means each string constant is configured in a `message.properties` file which is instantiated during start-up and made available through a `Message` class.

Plan for evaluation. We plan a short and a long term evaluation for our recommendation approach with `CHANGECOMMANDER`. In the short-term, we apply `CHANGECOMMANDER` on a set of open-source software systems written in Java that have an issue tracking system: (1) We first select a release and collect method invocation context changes up to this release and prepare the data for the recommendations. (2) We apply `CHANGECOMMANDER` to the selected release and collect the set of recommendations. For each recommendation we check whether the corresponding context change was also applied in later versions. The more recommendations are correct the more accurate `CHANGECOMMANDER` is.

In the long-term we intend to perform a thorough user study with students and our industrial partners. The study with students will be a controlled experiment. We split the students into two groups and let both perform special tasks. One group will use `CHANGECOMMANDER`, the other will not. We aim at showing that the group with recommendation support will have fewer bugs left open than the other group.

Currently we are negotiating with one of our industrial partner to include `CHANGECOMMANDER` in their develop-

ment process. We intend to let their developers use CHANGE-COMMANDER for a certain time period and then gather their experiences to enhance the tool in performance, usability, and accuracy.

7. RELATED WORK

We describe previous efforts in the area of recommendation systems that either build on recent source code information or on historical data of a software system.

Holmes *et al.* implemented the example recommendation tool *Strathcona* [9]. It uses the structure of source code under development to find relevant examples in a repository. The found examples then assist a developer on how to use or extend an API. In contrast, we recommend context changes that were applied in the past to prevent future bugs.

Ying *et al.* and Zimmermann *et al.* developed approaches that guide programmers along related changes by telling them “developers who changed these functions also changed ...” [18, 19]. The Hipikat tool of Čubranić *et al.* uses project history information to provide recommendations for a modification task [3]. These approaches are complementary to our approach: They focus on system wide programming task containing more than one possible file. We, instead, focus on the recommendation of fine-grained, single changes in method bodies.

The *Hatari* tool [16] rates the risk of changing a method according to the frequency of method changes that caused a bug. Such changes are also called fix-inducing changes, meaning that a bug was reported after a particular change was made. The risk of changing a method is then visually highlighted in the Java editor. *Hatari* is related to CHANGE-COMMANDER: It uses similarly constructs in Eclipse and it also uses bug fix changes to generate the feedback. CHANGE-COMMANDER does also highlight a certain risk. It shows the developer which method invocations are likely to be risky not to change and, in contrast to *Hatari*, suggests corresponding automated adaptations.

Based on how a framework adapts to its own changes Dagenais and Robillard developed a recommendation system that suggest replacements for framework elements accessed by client programs [4]. The change information we provide can complement these suggestions. For instance, we can additionally recommend necessary condition checks before calling a recommended method.

8. CONCLUSIONS

We presented our recommender approach that leverages context change data to suggest corresponding modifications when a developer adds a method invocation. The context changes are collected from the history of a software system and aggregated to extract patterns. Based on these patterns, we prepare suitable context change recommendations for method invocations.

We introduced CHANGE-COMMANDER, the implementation of our recommendation approach as an Eclipse plugin. The tool is seamlessly integrated into the build-process of Eclipse to instantly generate context change recommendations and give visual feedback by means of highlighting affected method invocations in the code. In addition, CHANGE-COMMANDER provides automated adaptation support through quick fixes. By integrating our recommendation approach into the development process, we aim at reducing future bugs.

Acknowledgments. The authors would like to thank Martin Pinzger and the reviewers for their insightful suggestions that greatly helped to improve the paper.

9. REFERENCES

- [1] J. Bevan, J. E. James Whitehead, S. Kim, and M. W. Godfrey. Facilitating software evolution research with Kenyon. In *Proc. European Software Eng. Conf. and ACM SIGSOFT Symposium Foundations of Software Eng.*, pages 177–186, Sep 2005.
- [2] F. P. Brooks. *The Mythical Man-Month*. Addison Wesley Longman, Inc., anniversary edition, 1995.
- [3] D. Cubranic, G. C. Murphy, J. Singer, and K. S. Booth. Hipikat: A project memory for software development. *IEEE Trans. Software Eng.*, 31(6):446–465, Jun 2005.
- [4] B. Dagenais and M. P. Robillard. Recommending adaptive changes for framework evolution. In *Proc. Int’l Conf. Software Eng.*, pages 481–490, May 2008.
- [5] A. M. Davis. *201 Principles of Software Development*. McGraw-Hill, 1995.
- [6] M. Fischer, M. Pinzger, and H. Gall. Populating a release history database from version control and bug tracking systems. In *Proc. Int’l Conf. Software Maintenance*, pages 23–32, Sep 2003.
- [7] B. Fluri and H. C. Gall. Classifying change types for qualifying change couplings. In *Proc. Int’l Conf. Program Comprehension*, pages 35–45, Jun 2006.
- [8] B. Fluri, M. Würsch, M. Pinzger, and H. C. Gall. Change distilling: Tree differencing for fine-grained source code change extraction. *IEEE Trans. Software Eng.*, 33(11):725–743, Nov 2007.
- [9] R. Holmes, R. J. Walker, and G. C. Murphy. Approximate structural context matching: An approach to recommend relevant examples. *IEEE Trans. Software Eng.*, 32(12):952–970, Dec 2006.
- [10] D. Hovemeyer and W. Pugh. Finding bugs is easy. In *Proceedings of the 19th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 92–106. ACM, October 2004.
- [11] S. Kim and M. D. Ernst. Which warnings should I fix first? In *Proc. European Software Eng. Conf. and ACM SIGSOFT Symposium Foundations of Software Eng.*, pages 45–54, Sep 2007.
- [12] S. Kim, K. Pan, and J. E. James Whitehead. Memories of bug fixes. In *Proc. ACM SIGSOFT Symposium Foundations of Software Eng.*, pages 35–45, Nov 2006.
- [13] B. Livshits and T. Zimmermann. Dynamine: Finding common error patterns by mining software revision histories. In *Proc. European Software Eng. Conf. and ACM SIGSOFT Symposium Foundations of Software Eng.*, pages 296–305, Sep 2005.
- [14] M. Pinzger, E. Giger, and H. C. Gall. Handling unresolved method bindings in Eclipse. Technical report, University of Zurich, 2007.
- [15] R. Purushothaman and D. E. Perry. Toward understanding the rhetoric of small source code changes. *IEEE Trans. Software Eng.*, 31(6):511–526, Jun 2005.
- [16] J. Śliwerski, T. Zimmermann, and A. Zeller. Hatari: Raising risk awareness. In *Proc. European Software Eng. Conf. and ACM SIGSOFT Symposium Foundations of Software Eng.*, pages 107–110, Sep 2005.
- [17] J. Śliwerski, T. Zimmermann, and A. Zeller. When do changes induce fixes? In *Proc. Int’l Workshop Mining Software Repositories*, pages 24–28, May 2005.
- [18] A. T. Ying, G. C. Murphy, R. Ng, and M. C. Chu-Carroll. Predicting source code changes by mining change history. *IEEE Trans. Software Eng.*, 30(9):574–586, Sep 2004.
- [19] T. Zimmermann, P. Weissgerber, S. Diehl, and A. Zeller. Mining version histories to guide software changes. *IEEE Trans. Software Eng.*, 31(6):429–445, Jun 2005.