Developer-Specific Awareness of External Changes

Reid Holmes
Department of Computer Science & Engineering
University of Washington
Seattle, WA, USA
rtholmes@cs.washington.edu

Robert J. Walker
Department of Computer Science
University of Calgary
Calgary, AB, Canada
walker@ucalgary.ca

Abstract

It is often assumed that developers' view of their system and its environment is always consistent with everyone else's. In certain situations that arise in practice, this assumption is false, and current development practice does not adequately address the resulting shortcomings. This paper examines when the assumption does not hold, and the implications for developers. A method for helping developers understand and cope with these situations is described and an evaluation of this method is proposed.

1. Introduction

A base assumption of many development approaches is that every member of an organization (or group of organizations) always has a consistent view of their system, modulo any changes each developer is currently work on. Because of this model it is assumed that any change a developer makes to their source code will be immediately detectable by everyone else. Unfortunately, this assumption does not always hold in practice.

For example, one approach to support the extension of systems is through a plug-in infrastructure. The plug-in author must cede some control over the external environment in which the plug-in can be installed: while the developer may have created the plug-in for version n of a framework, a user may try to install the plug-in in version $n + k$. If the developer is not aware of the changes between the two framework versions that will affect their plug-in, the plug-in may fail and it could take a long time before the developer hears of the problem. Although the plug-in author may not be a member of the team that maintains the framework, their plug-in can still be affected by the actions of an external framework developer.

The main problem in these situations is that some external change may occur that could break the developer's code, but the developer will not immediately be aware of it. The time delay between the change being made and the problem being detected exacerbates the effects of the problem, as users' opinions of the quality of the product or the time available for repairing the issues can be severely reduced. Furthermore, this delay also impedes the developer's ability to inform the author of the initial change of the detrimental impact of the change, and to have them respond with a reasonable compromise. It has been shown that facilitating communication between the right people can decrease the time required to resolve technical problems [1].

One way to avoid such situations is for the developer to monitor all the changes happening to the depended-upon environment and to act upon them immediately. Ultimately, keeping track of all the relevant changes to the depended-upon environment, while being focused on extending and repairing the functionality of the developer's own product, is an onerous task that is easy to perform poorly. The majority of these changes simply are not relevant to the developer's own code, and the few changes that are relevant can easily be lost in the noise.

To overcome the burden of managing a high volume of mostly-irrelevant changes, and the effort required to monitor changes across many different projects, we outline the YooHoo system. YooHoo analyzes each change to a depended-upon project for its potential impact on the developer's code. YooHoo thus creates custom change logs that automatically identify changes relevant to a specific developer and allows them to filter these changes in terms of importance to their system.

Section 2 describes two problematic scenarios where developer-specific awareness could be beneficial. Related work is provided in Section 3. Our approach is described in Section 4, while our proposed evaluation is considered in Section 5.

2. Problematic Scenarios

We illustrate two instances in which keeping appraised of relevant source code changes is onerous and error-prone.

2.1. Large development teams

Large development teams are often split into many sub-teams, each of which works on an isolated branch of the main source code repository. Integration engineers integrate these branches with the head of the repository at regular intervals and reverse-integrate the head back down into each of the sub-team branches. For some large teams, their branch may be integrated with the head of the repository only twice a month; in these situations, it could take as long as a month for a change from one team to be distributed to every other developer on all the sub-teams (two weeks up and two weeks back down into all other branches).

Consider the situation of Zoe, a developer on the sub-team that maintains the core component of a large application. Zoe needs to make a change to the `calculate(..)` method's pre-conditions in the core component; she searches the repository and finding no conflicts, makes the change. One month later Lorenzo, a developer on the UI component sub-team, finds that his code has broken. Although he did not change anything, the integration engineers reverse-integrated Zoe's changes into his development branch overnight, and his `CalculateAction` class breaks; Zoe did not find Lorenzo's dependency because his code had not yet been integrated into her repository. Now, Lorenzo must diagnose the problem, file a bug with the core team, and fix Zoe's code temporarily, until the official changes are distributed through another integration cycle.

The time lag in this scenario makes it difficult to assess the original failure and increases the likelihood that another developer may take a dependency on the official but incorrect version of the `calculate(..)` method, rather than the version that Zoe will create in response to Lorenzo's bug report. While this bug may have been avoided if Lorenzo had noticed Zoe's change to `calculate(..)` when it happened, keeping abreast of changes on a single sub-team is difficult enough, without considering other development branches.

2.2. Plug-in infrastructures

Consider the situation of Stefania, an Eclipse plug-in developer. Her plug-in uses functionality from five different Eclipse plug-ins, two Apache projects, and one system she found on SourceForge. When Laura installed Stefania's plug-in into her environment it failed to behave as she expected. After several frustrating rounds of emails, Stefania realizes that Laura has new versions of three of the eight external dependencies her plug-in uses. Keeping up on the changes of these eight projects is overwhelming so Stefania never realized her code was affected by any of the changes these projects had made; after debugging her plug-in with the new versions of the dependencies she is able to resolve Laura's problem.

In this scenario, Laura has changed the external environment for Stefania's plug-in in a way that the plug-in was not designed to accommodate. While Stefania ideally would have kept current of the changes within Eclipse, the Apache projects, and the SourceForge project, the sheer volume of changes overwhelmed her ability to track them.

3. Related Work

Gross and Prinz [8] describe the need to present awareness information in the context of the user's current work activities, but do not go beyond a general model; our focus on the specific problem of maintaining awareness of environmental changes allows for a solution that can eliminate details that are irrelevant to an individual developer. Cataldo et al. [1] demonstrate that developers complete tasks more quickly when they are able to coordinate with the right people; our approach is complimentary to theirs, focusing on inferring relationships between developers from the source code, rather than explicitly-provided links extracted from issue tracking systems.

The need for coordination support in software development has long been recognized (e.g., see [9]). Much of this research has focused around software configuration management (SCM) systems as the key interaction point [7]. Various work has considered the problem of conflicting edits within team development situations (such as that of Perry et al. [11], de Souza et al. [4], and Sarma et al. [12]); our problem is significantly different due to its lack of a single, consistent repository to analyze.

De Souza et al. [3] identified problems in crossing the boundary from project-private information to project-public information, that lead to rough transitions even within teams, despite the application of tools that support it. Coordination and change awareness have moved beyond the artifacts contained within SCM systems, by also drawing upon the process-related metadata contained therein [5, 2]; we continue this trend in our approach.

Ideally, we would analyze the precise semantic effects of the changes made in external projects; unfortunately, this is not possible in the general case due to formal undecidability [10]. Otherwise, we might detect changes in external application programming interfaces (APIs), and automatically update the developer's client code (Xing and Stroulia [13] provide a recent attempt at this); however, in some cases the developer may judge that accommodating such API changes is not in the best interests of the project. This kind of decision cannot be made automatically.

4. Developer specific awareness

In both of our scenarios, environmental changes, beyond the control of the developer, introduced errors into their system. If the developer kept appraised of these external changes

Project	Total	Daily avg.
FreeBSD	37,843	103
KDE	128,755	352
Linux kernel	39,155	107
MySQL	19,366	53
NetBeans	88,293	241
Open Office	126,272	345

Table 1. Message traffic on the commit mailing lists for several prominent software systems between June 1, 2007 and May 31, 2008.

they could have adapted their code to work with the modified environment; however, keeping appraised can be very expensive. Table 1 demonstrates the volume of check-in messages for a 1-year period for several projects. Clearly, reading each message and analyzing its potential impact on their code would be an enormous chore.

4.1. The YooHoo approach

To combat the volume and quality of check-in notifications we present the YooHoo developer-specific awareness system. YooHoo runs as a client/server system; the server operates as a daemon along with a source control system (such as CVS or Subversion) and analyzes changes as they happen; the client is integrated into the developer's IDE. YooHoo has three main facets that help to ensure developers are notified about only those code changes that are relevant to them.

Code ownership analysis. One of the main drawbacks of change notification schemes is that each developer receives the same notifications, regardless of which parts of the system they may be interested in. To combat this, YooHoo first analyzes the developer's activity within their source code repositories and through historical ownership analysis determines which source code files they have changed in the past. These files represent resources that could be affected by an external change.

Structural relevance. Once the developer's set of resources has been identified, YooHoo then determines which external dependencies those resources have. By statically analyzing the source code, YooHoo can identify all the fields referenced, methods called, and subclass relationships made by the resources in which the developer is interested. This set provides YooHoo with a list of external resources that the developer may want to watch. Changes made to resources that are not in this set are filtered by YooHoo to allow the developer to focus their limited time on only a subset of changes.

Change impact analysis. All changes are not equivalent in their potential to impact a developer. Modifying a comment is of significantly less interest than changing the method signature of a method upon which the developer's code depends. YooHoo analyzes each change using the Change Distiller system [6] to gain insight into the nature of each change. By analyzing the abstract syntax tree (AST) associated with the change, Change Distiller classifies the changes into 35 different categories. YooHoo considers each of these categories (and some custom categories not provided by the Change Distiller analysis) and compares them to how the developer's code uses the structural element that changed. Using this information, YooHoo is able to categorize a change into HIGH, MEDIUM, or INFO priorities for the developer. HIGH priority changes will definitely break the developer's code; MEDIUM priority changes are ones that the analysis can neither rule out as irrelevant nor determine to be definitely impactful. INFO priority changes involve the developer's code but without an obvious impact.

YooHoo has been designed to improve the key shortcomings of general change notification systems by providing the following two benefits. (1) YooHoo only provides developers with notifications about changes that are *relevant* to them. Every change notification that the developer receives will be provided because the change happened to a resource that their code is dependent upon in some way. Through this filtering mechanism, we expect YooHoo to eliminate the vast majority of change notifications. (2) YooHoo categorizes changes by their potential to impact the specific developer. This categorization enables them to further reduce the message traffic that they must consider by enabling them to select only HIGH priority changes, if they so choose.

YooHoo should enable developers to keep appraised of all the relevant changes happening on multiple branches of a source code repository, or to follow the development of a large framework, without being overwhelmed by irrelevant changes. YooHoo also enables the rationale for the change's relevance to be viewed by the developer; e.g., in the first scenario, YooHoo could state to Lorenzo that "Zoe changed the `calculate(...)` method signature: this will cause an error in your `CalculateAction` class." In the first scenario, Lorenzo's YooHoo notification stream would alert him to any HIGH priority changes made to code that he depends upon across any development branch in the repository. In this way he could contact Zoe immediately about how he would be impacted, or to immediately respond by modifying his `CalculateAction` class. For the second scenario, using YooHoo Stefania could monitor all the changes in her dependent projects without becoming overwhelmed; rather than having to sift through thousands of changes, YooHoo would promote at most tens of changes for her to react to. In both of these situations the developer is able to act proactively to changes in external resources, rather than reacting to breakage when it happens.

5. Proposed evaluation

To evaluate our solution, we intend to address three research questions: (1) "Does YooHoo reduce the volume of notifications to a level that is manageable?"; (2) "Are the notifications that YooHoo presents as HIGH priority, actually impactful on the developer's code?"; and (3) "Are the notifications that YooHoo filters out really not impactful on the developer's code?" Paraphrasing, Question 2 asks about false positives, and Question 3 asks about false negatives.

We propose to analyze historical changes over a period of one year for multiple developers from several different projects that continue to be developed or maintained. For this time frame we will generate the list of YooHoo notifications for that developer and compare this count against the total number of changes they may have had to consider otherwise. This will give us a sense of the amount of reduction that YooHoo is able to provide developers, addressing Question 1.

In addition, given every HIGH priority change, we will analyze the developer's subsequent changes to see if they responded to the change in some way or if a bug remains unrepaired, thus addressing Question 2. Because of the potential time lag between the change happening and the developer's reaction, the start of the one year period used to analyze the relevance of changes will need to be at least two years before the present.

Unfortunately, identifying false negatives (Question 3) for this type of system is more problematic. Firstly, formal undecidability is a fundamental limitation to our ability to detect relevant, impactful changes, and likewise would hamper any post hoc analysis of the fidelity of the results. Secondly, going through the code—even with the help of the developer who worked on it—to identify every change that was made in response to an external change would be very expensive. However, we intend to randomly sample a very small subset of the filtered-out changes, to determine whether the developer reacted to them, for example, by adding a dependence on newly added APIs. If the rate of false negatives were much greater than zero, a problem would exist.

6. Conclusion

Developers do not always control when their external dependencies change. Typically, their response to these changes is reactive and late: a bug report is filed and they must scramble to resolve the problem. While responding to changes in this manner is inefficient and increases the difficult in fixing bugs, developers take this route because of the amount of work required to keep appraised of relevant changes. In this paper we outline the YooHoo system that provides developer-specific notification streams, enabling a developer to only keep track of those changes that are relevant to them. We have proposed an evaluation of YooHoo and are specifically interested in feedback on the suitability of this evaluation for our approach.

References

[1] M. Cataldo, P. Wagstrom, J. D. Herbsleb, and K. M. Carley. Identification of coordination requirements: Implications for the design of collaboration and awareness tools. In *Proc. ACM Conf. Comp.-Supported Coop. Work*, pages 353–362, 2006.

[2] L.-T. Cheng, S. Hupfer, S. Ross, and J. Patterson. Jazzing up eclipse with collaborative tools. In *Proc. Eclipse Technol. eXchange*, pages 45–49, 2003.

[3] C. R. B. de Souza, D. Redmiles, and P. Dourish. "breaking the code": Moving between private and public work in collaborative software development. In *Proc. ACM SIGROUP Int'l Conf. Support. Group Work*, pages 105–114, 2003.

[4] C. R. B. de Souza, D. Redmiles, G. Mark, J. Penix, and M. Sierhuis. Management of interdependencies in collaborative software development. In *Proc. Int'l Symp. Empir. Softw. Eng.*, pages 294–303, 2003.

[5] D. Draheim and L. Pekacki. Process-centric analytical processing of version control data. In *Proc. Int'l Wkshp. Princip. Softw. Evol.*, pages 131–136, 2003.

[6] B. Fluri, M. Wuersch, M. Pinzger, and H. Gall. Change distilling: Tree differencing for fine-grained source code change extraction. *IEEE Trans. Softw. Eng.*, 33(11):725–743, 2007.

[7] R. E. Grinter. Supporting articulation work using software configuration management systems. *Comp. Supported Coop. Work*, 5(4):447–465, 1996.

[8] T. Gross and W. Prinz. Awareness in context: A light-weight approach. In *Proc. Europ. Conf. Comp. Supported Coop. Work*, pages 295–314, 2003.

[9] R. E. Kraut and L. A. Streeter. Coordination in software development. *Commun. ACM*, 38(3):69–81, 1995.

[10] M. Moriconi and T. C. Winkler. Approximate reasoning about the semantic effects of program changes. *IEEE Trans. Softw. Eng.*, 16(9):980–992, 1990.

[11] D. E. Perry, H. P. Siy, and L. G. Votta. Parallel changes in large-scale software development: An observational case study. *ACM Trans. Softw. Eng. Method.*, 10(3):308–337, 2001.

[12] A. Sarma, G. Bortis, and A. van der Hoek. Towards supporting awareness of indirect conflicts across software configuration management workspaces. In *Proc. IEEE/ACM Int'l Conf. Autom. Softw. Eng.*, pages 94–103, 2007.

[13] Z. Xing and E. Stroulia. API-evolution support with Diff-CatchUp. *IEEE Trans. Softw. Eng.*, 33(12):818–836, 2007.