Promoting Developer-Specific Awareness

Reid Holmes and Robert J. Walker
Laboratory for Software Modification Research
Department of Computer Science
University of Calgary
Calgary, Alberta, Canada
rtholmes,rwalker@cpsc.ucalgary.ca

ABSTRACT

Maintaining a developer's awareness of changes in the software on which she depends is challenging. Awareness is often impeded at two ends of the spectrum: a lack of information, when the changes only become apparent when a build breaks or bugs appear; or an excess of information, where the changes are announced but the majority of the changes are not relevant to the developer in her particular project and context. In the middle ground lies the possibility of support for developer-specific awareness (DSA), wherein information about the changes is filtered on the basis of the developer's own code and interests. This paper discusses how the DSA problem is manifested in software development and briefly examines the design space involved in providing DSA notifications. A particular point in the space is proposed for a target implementation, called the YooHoo awareness system, that will help developers in loose organizations to keep apprised of any code changes that are specifically relevant to the source code for which they are responsible.

Categories and Subject Descriptors

D.2.3 [**Coding Tools and Techniques**]: Object-oriented programming

General Terms

Human Factors

Keywords

Developer-specific awareness, distributed teams, YooHoo

1. INTRODUCTION

Software systems are created by large, often geographically distributed, teams or more informally organized groups of developers. Thus, it should not be surprising that communication breakdowns have often been cited as a major cause of delays for industrial software systems [6]. The interconnected nature of complex software systems means that changes made by a committer to their own code can often have implications for other developers whose code depends on that of the committer. For the other developers, remaining aware of important changes can be challenging. Awareness is often impeded at two ends of the spectrum: a lack of information, when the changes only become apparent when a build breaks or bugs appear; or an excess of information, where the changes are announced but the majority of the changes are not relevant to the developer in her particular project and context. The middle-ground is unpopulated: we lack automated support for *developer-specific awareness* (DSA).

In practice, developers often maintain awareness of one another using email and instant messaging but this requires dedicated effort from the developer [8] and is disconnected from their development activities. This disconnection of awareness from the source code is significant as developers often use the code as their primary information source [10]. Developers often only notice external changes when they induce failures; keeping apprised of any changes in source code they depend upon can allow them to proactively remedy their systems before they break. However, developers can also suffer from an overabundance of information: being notified of every update made by their own team and all of the teams developing any external code they depend upon can quickly become overwhelming [3].

Several systems have investigated providing different aspects of awareness to developers. The Jazz system provides the concept of a *feed* that lists many recent changes to the system and provides an overview of what other team members are working on [2]. Palantír provides an online view of how developers are modifying their source before it is committed, enabling developers to predict future changes [9]. FASTDash also provides a real-time awareness system to small teams showing who is working on what code elements at any one time [1]. Each of these approaches provides a global view that—while useful for various purposes—does not meet the needs of developer-specific support.

Instead, filtering such information to present only what is relevant in the context of a particular developer both reduces extraneous information that the developer need consider and increases the visibility and effectiveness of the remainder. To this end, several factors must be considered: (1) how to determine what has changed, and the nature of the change, without burdening the committer; (2) how to analyze the developer's context to determine what would likely matter to them; (3) how to allow the developer to specify their preferences regarding the kind of information they want to receive, and to allow this specification to change dynamically; and (4) how to present the filtered information to the developer, to balance visibility with obtrusiveness. This paper outlines the requirements to support DSA and an implementation to achieve them.

CHASE'08, May 13, 2008, Leipzig, Germany.

The remainder of the paper is structured as follows. Section 2 outlines a scenario that motivates the utility of developer-specific awareness tools. Section 3 describes the requirements of such tools while Section 4 proposes a particular design scheme (called YooHoo) to achieve them. Section 5 outlines the related work and how YooHoo will contribute to this body of research. Section 6 provides future directions for our approach.

This paper contributes the concept of developer-specific awareness and a proposal about how it could be provided.

2. SCENARIO

Complex systems often have many dependencies on external systems. Consider a developer on a team that has written an application that keeps track of store inventories. They have chosen to implement the application using the Rich Client Platform (RCP) application framework [1]. This means they have external dependencies on the RCP framework as well as the SWT widget framework. Additionally, to support full-text searches of their inventory, they use Apache Lucene. The Apache Derby database is used to track the inventory while JDOM is used for importing and exporting data from the system while HTTPClient enables importing entries from remote HTTP servers. JFreeChart is used to graph how the inventory has changed over time while gnupdf enables reports to be generated in PDF format. This system currently depends on no less than 8 external projects to properly function.

The team tests and deploys the system within the company. Two months later one of the developers updates the HTTPClient library to the latest version as it fixed several bugs relevant to some new code he was writing. Three days later one of his fellow developers is assigned a bug that states that her unit tests are failing: one of her methods is not catching an exception it should. This second developer is perplexed because she knows that she has not changed this code in a long time. Looking into the bug she sees that HTTPClient's `GetMethod.getResponseBodyAsString()` method is throwing an `IOException` that she is not handling properly. She knew that HTTPClient had been upgraded, from talking to one of her teammates, but did not know that this upgrade would affect her code as she thought the upgrade primarily fixed bugs. Unfortunately, the new version had changed the `getResponseBodyAsString()` method to throw an exception instead of returning null. After changing her code to handle the exception and re-running the test suite, she closes her bug.

In this case the developer was caught off guard; she did not know that the HTTPClient upgrade would affect her. Even looking through the extensive change list manually, she did not notice the change amongst the numerous other changes associated with a major update of the feature list. The difficulty in keeping apprised of these low-level changes is exacerbated by the fact that she must track the development of 7 other major projects in addition to HTTPClient; the burden of trying to keep abreast of these changes is excessive.

3. REQUIREMENTS

A developer-specific awareness system can only be effective if developers make use of it. For this to happen, it needs to provide an added value beyond simply listing the complete change history for a project as it happens. While there may be value in aggregating the change history streams when external dependencies from many projects must be considered, this also exacerbates the amount of information the developer must consider. DSA streams would be an effective mechanism to reduce the number of awareness events that the developer must consider; these would maximize the chance that an awareness notification is applicable to the developer.

DSA streams should be tailored to help developers keep abreast of what other developers are doing to code they are dependent upon; as such, these streams should involve analysis and presentation of the changes as more than just textually-represented events. By analyzing changes semantically, the stream can further increase the value of any notification to the developer by prioritizing the notifications it presents to them. For instance, a change that modifies the signature of a method that a developer depends upon (and would break his code after integration takes place) is of much higher priority than one that adjusts the white space or documentation for that same method.

There are three key tasks that should be automated to minimize the amount of effort required of the developer to configure and use a DSA system. (a) The system should infer what code the developer owns by analyzing all of the source code in the developer's project and its past change history. Ownership should be determined at as fine a granularity as possible (e.g., at the method- or field-level). (b) The system should determine what external structural dependencies the developer's code has using static analysis. An external dependency is defined as one on any class, interface, method, or field that the developer does not own. (c) Relevant changes for these dependencies should be retrieved by automatically locating and downloading any changes for a particular external dependency without the developer's intervention. In addition to this automated support the developer should also have the ability to set their own interest levels on a global and per-project basis.

Providing a flexible means for conveying awareness notifications to developers enables a DSA system to both employ several low-impact, non-intrusive visual cues to convey low-priority change information as well as selectively use highly-visible notification mechanisms for changes that may have a large impact on the developer's code.

4. THE YOOHOO DESIGN PROPOSAL

There are many possible designs that could be employed to satisfy the requirements given in Section 3. We describe one such design, which we call the YooHoo developer-specific awareness system. YooHoo will integrate with the developer's workflow while automatically providing developer-specific notifications in a flexible manner.

YooHoo notifications will not be instantly available to the developer; his development environment will poll for relevant changes at specified intervals. This interval can be set on a per-project basis: if a developer knows that a system he is dependent upon is frequently changing, YooHoo could check for updates every 5 minutes; if he does not need quick updates for another project, he can set it to poll on a daily basis. In the scenario of Section 2, the developer may want to be frequently notified of changes to her system that affect her, whereas changes to the 7 external libraries can happen less frequently.

4.1 Determining relevance

Identifying relevant dependencies. We will develop heuristics that will analyze the developer's system and past change history to infer what code the developer currently owns; the developer will also be able to add and remove code from this list manually. Developers can own code at a granularity as specific as as individual methods, fields, or classes; or as coarse as files, packages, and projects. Once the developer's code has been identified, YooHoo will perform lightweight static analysis on it to identify a list of

[1] http //wiki.eclipse.org/index.php/Rich_Client_Platform

external dependencies that his code relies upon. As with the ownership of the code, the developers will be able to de-select any of these external dependencies for which he does not want notifications (and add others in which he is interested).

Retrieving relevant changes. Using the list of external dependencies for which the developer is interested in notifications, YooHoo connects with the change repositories for these systems and retrieves information for these entities. This retrieval will be at the file-level as that is the level at which most software configuration management (SCM) repositories operate. YooHoo will not retrieve any change information for portions of the external systems that are not relevant to the developer. Further details about how this might be done are given in Section 4.3.

Assigning severity to a change. Once the file-level details have been retrieved from the remote change repositories, YooHoo will first determine if the change modifies any parts of the file that correspond to entities the developer is dependent upon. If the change is relevant, YooHoo will then further analyze the change to determine its nature: Was only white space changed? Did the method gain some more external dependencies of its own? Did the API for the method change? Such information will be used to summarize the nature of the change for the developer and to assign a severity indicator to it. The summarization will enable the developer to glance at a change notification and get a high-level summary of how the code was modified without having to look at the diff associated with the change.

4.2 Notification mechanism

YooHoo will provide a variety of notification mechanisms to give change notifications a visual prominence appropriate to their severity. Low-severity changes will not interrupt the developer's workflow, whereas high-severity changes will be made obtrusive as they may require some immediate action to resolve.

Assuming an extensible integrated development environment (IDE), YooHoo will annotate the source code in the developer's editor to provide passive notifications. By annotating both the text that corresponds to the dependency that has changed, as well as the gutter icons in the vertical ruler, we can provide feedback of both the severity and recency of the change using colour and translucency. Enhanced tool tips will give a complete overview of the change if the developer hovers over annotated code or its corresponding gutter icon.

If the developer is actively seeking recent related changes, we will provide a feed-style view that lists recent changes in reverse-chronological order, giving an indication of each change's severity, originating project, author, and name of the changed entity. Selecting any change loads the same page as the passive notification tool tip that shows all of the details of the change. The feed list will be sortable in terms of originating project, the name of the dependency that was changed, our analysis of the severity of the change, author, and when the changed occurred. The feed list will also have a status field that the developer can check off if they have dealt with the change. Developers will be able to filter the list (for instance removing low-priority updates for a specific project), and specifically delete any notifications they are not interested in. Another aspect of active notifications will be to promote high-severity change notifications into other views in the developer's IDE; this may include adding the change to the developer's current task list.

4.3 Technical details

A number of engineering problems must be overcome for YooHoo to work properly; these are important to resolve properly to ensure a seamless and automated experience for the developer. YooHoo will be deployed as a client–server system. Each SCM repository will have an associated YooHoo daemon that is notified whenever changes are committed. A master server will enumerate each of the project-level daemons so that other systems can access their change data. Individual daemons can choose not to list themselves with the master if they so choose (for privacy reasons); through the client preferences, these daemons can still be added by the developer.

The client will poll for changes at a predetermined interval from the server; this interval will be configurable on a per-project basis (e.g., if a team is changing its code rapidly, a higher frequency of polling them would be possible). The analysis portion will be undertaken partially at the daemon and partially on the client. The client will request changes for specific dependencies from the daemon but the final filtering and analysis of these changes will be undertaken on the client.

Our initial client implementation will integrate with either the Eclipse or Jazz IDEs; daemons will be provided for CVS, Subversion, and the Jazz SCM repository.

4.4 Addressing the scenario

When built, the YooHoo DSA system would help the developer in the scenario from Section 2 in one specific way. After getting to work one morning, she could see from her awareness feed that a high-priority change had been made to a dependency that would affect her. YooHoo would state that `GetMethod.getResponseBodyAsString()` had been changed and that it now threw an `IOException`. She could also directly navigate to the locations in her code where this API was used, and fix the problem before her team even upgraded to the latest version of HTTPClient.

This scenario dealt with an external library being updated; a developer working on a large project comprised of several development teams in a rapid development phase would also have dependencies on 'external' code that is frequently changing. The YooHoo DSA system would also help keep these developers appraised of changes relevant to them, reducing the number problems they must resolve when the teams integrate their code.

5. RELATED WORK

Damian et al. studied some of communication challenges faced by a globally-distributed software team. Two of their three main observations were that awareness networks dynamically change and an overabundance of information can lead to broken builds [3]. Our design for YooHoo addresses these two observations. As the code a developer owns and the dependencies within that code changes the notifications the developer receives will automatically be adapted. To address information overload developers will only receive notifications that are relevant to the code they own.

After interviewing several software teams, Singer found that programmers rely heavily on the code as their primary source of information when they are trying to understand a system [10]. Gutwin et al. found that developers can effectively communicate using chat and mailing-lists if they committed to those tools [8]. These findings have strongly influenced how we have designed YooHoo: we rely only on source-code related information (rather than other communication artifacts such as bugs and documents) and our communication mechanism is designed to be lightweight; we do not require explicit work on the part of the committer or client developer to signal or gather notifications.

Several systems enable developers to see visual representations of various aspects of source code. SeeSoft provides a line-by-line view of source code attributes [5]. The Aspect Browser [7] and Code Thumbnails [4] projects provide graphical, abbreviated views

of source code that the developer can use to understand large swaths of source code at a time. These systems differ from YooHoo in that they are developer-driven exploratory tools; they do not actively notify the developer of interesting changes. Ariadne creates a version of the structural graph that focuses on the people associated with each edge; this system helps developers to identify any relevant developer for any part of the program's call graph [11].

Two existing awareness systems are of particular relevance to YooHoo. FASTDash provides a real-time awareness visualization of the all of the artifacts shared by a team [1]. This visualization updates in real-time, as developers open and close files. FASTDash is well-suited to large-screen displays in team common-areas, or on dedicated monitors. It enables any developer to figure out what every other team member is doing at-a-glance. In contrast, YooHoo creates customized awareness streams for every developer based on their current development context, their configurable preferences, and any code that they own. YooHoo is not suitable for gaining general awareness of other team members but instead helps the developer become aware of events that are specifically relevant to their work.

Palantír provides a push-model configuration management system that increases awareness of changes in the source code repository by broadcasting them to other users of the same repository [9]. Palantír analyzes every commit that is made to the repository to measure the severity of the change; this information is also pushed to other repository clients, in a lightweight manner. Palantír and YooHoo share many common traits: they both aim to inform developers of relevant changes in a lightweight fashion using a notification system that actively alerts developers to changes. Neither system requires the developer to do any extra work to ensure that the notifications are sent or that the correct developers receive them. The systems also have major differences: Palantír provides online updates of what other developers are working on as a global overview; it does not tailor its display to show a developer only those specific changes that are relevant to them.

6. FUTURE WORK

This paper has outlined a conceptual framework for developer-specific awareness notifications the problem they attempts to resolve. We have also discussed YooHoo, our proposed DSA solution. We have written the structural-analysis portion of the tool that can determine the dependencies for which the developer should receive notifications. We will next build the agent that collects changes and prepares them for distribution. Once this is done we will build the client tool that alerts the developer of any relevant changes; this will be in the form of both an IDE integration and a more traditional feed-based approach. Finally, we will investigate means for analyzing the changes that have been made, to provide the developer with some indication of level-of-impact with respect to their systems.

Evaluation of YooHoo will proceed in two phases: first we will conduct an empirical evaluation using existing development history to determine how effectively we can deliver developer-specific awareness notifications for a modelled developer owning a particular module of an existing system; secondly, we will provide real developers with YooHoo and perform a longitudinal study to see if they find it useful and usable.

7. CONCLUSION

As developers work on larger software systems, they work in increasingly disconnected organizations and their code becomes increasingly dependent on external systems. These two factors combine to make their code more susceptible to external source code changes that could impact their code, while remaining oblivious to the issue. In this paper we have proposed the YooHoo developer-specific awareness system that tracks changes as they are committed and provides developers with personalized notification streams alerting them to changes that may directly impact their code. These alerts can help the developer adapt their code as needed, or to initiate a conversation with the committing developer, as necessary. YooHoo aims to help developers be more connected both with their teammates' activities and with external changes that could break their code; by preempting the manifestation of these changes as bugs, the developer can spend more time working on their system without having to fear being surprised by changes in the code upon which they depend.

8. REFERENCES

[1] Jacob T. Biehl, Mary Czerwinski, Greg Smith, and George G. Robertson. FASTDash: A visual dashboard for fostering awareness in software teams. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 1313–1322, 2007.

[2] Li-Te Cheng, Susanne Hupfer, Steven Ross, and John Patterson. Jazzing up Eclipse with collaborative tools. In *Proceedings of the Eclipse Technology Exchange*, pages 45–49, 2003.

[3] Daniela Damian, Luis Izquierdo, Janice Singer, and Irwin Kwan. Awareness in the wild: Why communication breakdowns occur. In *Proceedings of the International Conference on Global Software Engineering*, pages 81–90, 2007.

[4] Robert DeLine, Mary Czerwinski, Brian Meyers, Gina Venolia, Steven Drucker, and George Robertson. Code thumbnails: Using spatial memory to navigate source code. In *Proceedings of the Visual Languages and Human-Centric Computing*, pages 11–18, 2006.

[5] Stephen G. Eick, Joseph L. Steffen, and Jr. Eric E. Sumner. SeeSoft: A tool for visualizing line-oriented software statistics. *IEEE Transactions on Software Engineering*, 18(11):957–968, 1992.

[6] Jr. Fred P. Brooks. The mythical man–month. In *Proceedings of the International Conference on Reliable Software*, page 193, 1975.

[7] William G. Griswold, Jimmy J. Yuan, and Yoshikiyo Kato. Exploiting the map metaphor in a tool for software evolution. In *Proceedings of the International Conference on Software Engineering*, pages 265–274, 2001.

[8] Carl Gutwin, Reagan Penner, and Kevin Schneider. Group awareness in distributed software development. In *Proceedings of the ACM Conference on Computer-Supported Cooperative Work*, pages 72–81, 2004.

[9] Anita Sarma, Zahra Noroozi, and André van der Hoek. Palantír: Raising awareness among configuration management workspaces. In *Proceedings of the International Conference on Software Engineering*, pages 444–454, 2003.

[10] Janice Singer. Practices of software maintenance. In *Proceedings of the International Conference on Software Maintenance*, pages 139–145, 1998.

[11] Erik Trainer, Stephen Quirk, Cleidson de Souza, and David Redmiles. Bridging the gap between technical and social dependencies with Ariadne. In *Proceedings of the Eclipse Technology Exchange*, pages 26–30, 2005.