Customized Awareness: Recommending Relevant External Change Events

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 practice, this assumption can be false, as the developer has little practical control over changes to the environments in which their code will be deployed. To proactively respond to such situations, developers must constantly monitor a flood of information involving changes to the deployment environments; unfortunately, the vast majority of this information is irrelevant to the individual developer, and its sheer volume makes it likely that infrequent change events of relevance are overlooked. As a result, errors may arise at deployment time that the developer does not immediately detect.

This paper presents a recommendation approach for filtering the flood of change events on deployment dependencies to those that are most likely to cause problems for the individual developer. The approach is evaluated for its ability to drastically filter irrelevant details, while being unlikely to filter important ones. The relevance of the results is assessed on the basis of deployment problems that would have historically occurred within a set of industrial systems.

Categories and Subject Descriptors

D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement; K.6.3 [**Management of Computing and Information Systems**]: Software Management—*software maintenance*; H.1.2 [**Models and Principles**]: User/ Machine Systems—*human factors*

General Terms

Human Factors, Reliability, Design, Experimentation

Keywords

Change events, deployment environment, external dependencies, information overload, developer-specific awareness, customized awareness, recommendation system, YooHoo.

1. INTRODUCTION

A basic 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 noticed by everyone else. Unfortunately, this assumption does not always hold in practice [3, 9, 8].

As systems are rarely written entirely from scratch, building instead atop existing frameworks, libraries, and other systems, changes to this existing code have the potential to impact the developer. For example, one approach to support the extension of systems is through a plug-in infrastructure. The plug-in developer must cede some control over the external environment in which the plug-in can be installed: while the developer created the plug-in in a development environment that used version n of a framework, a user may try to deploy the plug-in in an environment that uses 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, and responds to, the problems.

As a concrete example, the Eclipse Metrics plug-in is dependent on code from 17 external projects; while its developers made only 643 file revisions to their own code between 2004 and 2008, the systems that Metrics depends on made 135,972 file revisions, an overwhelming volume of changes to keep abreast of, especially considering the majority of these changes will not impact the relatively small functionality from each external project on which Metrics depends.

Current plug-in infrastructures do not suffice to eliminate such issues; at best, the developer can prohibit deployment if a too-new version of a framework would be used, but developers are understandably reluctant to prohibit deployment before they know that a problem actually exists—it would immediately and often unnecessarily limit who could deploy their plug-in. Although the plug-in developer 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 such situations is that some external change may occur that could break the developer's code when it is deployed, but the developer will not immediately be aware of it [9]. The time delay between the change being made and the problem being detected exacerbates the effects of the problem [12], as users' opinions of the quality of the product or the time available for repairing the issues can be severely reduced. Furthermore, such a delay 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 [3]. It has been shown that facilitating communication between the right people can decrease the time required to resolve technical problems [5].

One way to avoid these situations is for the developer to monitor all the changes happening to externally depended-upon projects and to act upon these immediately. Ultimately, keeping track of all these projects and their changes, while being focused on the repair and extension of the developer's own product, is an onerous task that is easy to perform poorly [8]. The majority of these external changes simply are not relevant to the developer's own code; the few 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 present the YooHoo system. YooHoo analyzes each change to a depended-upon project for its potential impact on the developer's code, and thereby creates custom change event streams that recommend change events as relevant to a specific developer.

We evaluated YooHoo's ability to reduce the level of noise while not suppressing impactful events, by examining the historical changes to a set of industrial systems. For each system, we obtained an old version along with its deployment environment, analyzed its external dependencies, and used YooHoo to see which change events on these dependencies would have been announced to the system's developers. To quantify the confusion matrix, we analyzed whether these events would have caused problems in deployment and whether they were ever actually acted upon by the system's developer. We found that the cumulative true positive rate was 93% and that the cumulative false positive rate was nearly 0%. In addition, we examine the prospects for reducing the potential false positives and false negatives in the rare circumstances where they could conceivably occur.

Section 2 describes two problematic scenarios where customized awareness of external changes could be beneficial. Related work is provided in Section 3. Our approach is described in Section 4, while our evaluation is detailed in Section 5. Remaining issues are discussed in Section 6.

2. MOTIVATION

We illustrate two problematic scenarios involving keeping appraised of potential deployment problems due to changes in externally depended-upon projects.

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 Elliott, a developer on the sub-team that maintains the core component of a large application. Elliott needs to make a change to the `calculate(..)` method's pre-conditions in the core component; he 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 Elliott's changes into his development branch overnight, and his `CalculateAction` class breaks; Elliott did not find Lorenzo's dependency because his code had not yet been integrated into his repository. Now, Lorenzo must diagnose the problem, file a bug with the core team, and fix Elliott'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 Elliott will create in response to Lorenzo's bug report. While this bug may have been avoided if Lorenzo had noticed Elliott'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 that 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 herself current on changes within Eclipse, the Apache projects, and the SourceForge project, the sheer volume of changes overwhelmed her ability to track them.

2.3 Drinking from the fire hose

In both of our motivational scenarios, environmental changes, beyond the control of the developer, introduced errors into his or her system at deployment time. If the developer kept appraised of these external changes, the code could have been adapted to work with the modified environment; however, keeping appraised can be very expensive. Figure 1 gives a sample of the arcane detail that further obscures the actual significance of an individual commit message; Table 1 demonstrates the volume of commit messages for a 1-year period for several projects. Reading each message, interpreting its key content, and analyzing its potential impact on one's code would be an enormous chore.

3. RELATED WORK

Gross and Prinz [16] 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

```
Subject: [r9g-cvs] [hg] refactoring.java: #144961: fixing NPE
From: Jan P. <jp... (at) netbeans.org>
Message-id: hg.abbaf0f1ff63.1219918286.1797694213 (at)
        hg.netbeans.org
Date: 2008-08-28 12:11:26
changeset abbaf0f1ff63 in main
details: http://hg.netbeans.org/main?cmd=changes baf0f1ff63
description:
#144961: fixing NPE
diffs (12 lines):
diff -r eb7c38562250 -r abbaf0f1ff63
a/refactoring.java/src/org/netbeans/modules/refactoring/java/ui/
        UIUtilities.java
b/refactoring.java/src/org/netbeans/modules/refactoring/java/ui/
        UIUtilities.java
Aug 28 12:03:58 2008 +0200
@@ -92,7 +92,7 @@ public final class UIUtilities {
                headerRenderer = table.getTableHeader().
                getDefaultRenderer();
         }
             Component comp = headerRenderer.
                getTableCellRendererComponent(
-            null, column.getHeaderValue(), false, false, 0, 0);
+            table, column.getHeaderValue(), false, false, 0, 0);
             int width = comp.getPreferredSize().width;
```

Figure 1: A sample NetBeans commit message.

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. [5] demonstrate that developers complete tasks more quickly when they are able to coordinate with the right people; our approach is complementary 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., [19]). Much of this research has focused around software configuration management (SCM) systems as the key interaction point (e.g., [15, 1]). Various work has considered the problem of conflicting edits within team development situations (e.g., [23]); our problem is significantly different due to its lack of a single, consistent repository to analyze.

De Souza et al. [9] 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 [10, 6, 11]; we continue this trend in our approach.

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 [6]; we adapt the feed concept to a narrow focus on what relevant changes have occurred. Mylyn elides various elements in an IDE to help the developer focus on their current task [18]; the developer's current task is largely irrelevant in our context.

Project	Total	Daily
FreeBSD	37,843	151
KDE	128,755	515
Linux kernel	39,155	157
MySQL	19,366	77
NetBeans	88,293	353
Open Office	126,272	505

Table 1: Message traffic on the commit mailing lists for several prominent software systems between June 1, 2007 and May 31, 2008. Daily average is based on 5 workdays/week, 50 weeks/year.

Continuous integration (CI) tools can help a project team stay appraised of changes, builds, and tests; examples of these systems include Trac[1] and Cruise Control. Unfortunately, CI tools do not currently address version changes between organizations (e.g., external changes) or automatically configure new deployment environments. We view our approach as being complimentary to CI tools and could foresee integrating them to deliver project-specific (rather than developer-specific) notifications.

Fitzpatrick et al. [13] have examined the coordinated use of SCM systems, notification lists, and instant messaging in coordination. All these approaches suffer the same issues of information overload for external developers who depend on them [17, 20].

Palantír provides an online view of how developers are modifying their source before it is committed, enabling developers to predict future changes [24]. FishEye is one of many commercial products used for visualizing the overall activity within an SCM repository. The War Room Command Console [22] and FASTDash [2] provide real-time awareness systems to teams showing who is working on what code elements at any one time. Each of these approaches provides a global view that—while useful for various purposes—does not meet the needs of developer-specific support. Sarma et al. have also examined the needs of multiple teams working in concert on shared artifacts and extended Palantír to support it [26]; Sarma et al. have also performed notable work in evaluating awareness in SCM systems [27, 25]. However, their situation is quite different from our producer–consumer context.

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 [21]. Otherwise, we might detect changes in external application programming interfaces (APIs), and recommend changes to the developer's client code [28, 7]; however, for some projects, such recommendations will not be possible due to a lack of example transformations to mine. Furthermore, the correct course of action may not be as simple as replacing one method call with another.

There are a number of techniques for differencing source code versions to detect and classify changes—a necessary step if they are to be filtered on the basis of category; a notable, recent representative is the Change Distiller approach of Fluri et al. [14] that we apply in our solution. Canfora and Cerulo [4] demonstrate that fine-grained indexing of change repositories can help to track which changes were made to address which bugs; we apply a similar technique to track fine-grained changes.

4. APPROACH

To combat the volume of irrelevant change notifications we present YooHoo, a system for recommending external change events as impacting a specific developer.

[1]We do not provide URLs for tools that are easily located via a web search.

YooHoo has been designed to improve the key shortcomings of general change notification systems by providing the following three benefits. (1) YooHoo only provides developers with recommendations about changes that are *relevant* to them. Every change notification that the developer receives will be provided because a change was made to a resource that their code is dependent upon. Through this filtering mechanism, YooHoo eliminates the vast majority of change notifications. (2) YooHoo categorizes changes by their potential to impact the specific developer's source code. This categorization enables them to further reduce the message traffic that they must consider by enabling them to ignore changes that are unlikely to break their deployed code, if they so choose. (3) YooHoo integrates the change notifications from many isolated data sources, enabling potentially-relevant changes to be considered in a single location. Simply locating and navigating through each repository places a high burden on developers.

YooHoo has two main kinds of component: *generic change analysis (GCA) engines*, each of which delivers a stream of change events about a project without tailoring to the needs of an individual developer; and *developer-specific analysis (DSA) engines*, each of which selects a set of GCA engines based on the project dependencies of the developer, and filters and refines the generic change event stream based on an estimate of the change impact on the developer's code. As a result, the developer receives recommendations about changes to his code's deployment environment that are very likely to cause failure, plus purely informational notifications about API changes and other project-related information that are estimated to be of interest to him. GCA engines typically run alongside the SCM that they monitor, while DSA engines run on the client's machine (typically within an IDE). A central YooHoo GCA dispatcher connects DSA engines with the GCA engines they require. YooHoo presently only analyzes changes to Java source code.

4.1 Generic change analysis engines

Each GCA engine monitors a source control repository (adapters for CVS and Subversion are currently supported) for commit events and alterations to repository tags and branches. When a change to the repository is detected, it is analyzed and one or more change events are recorded in the engine's internal database. When a GCA engine is initialized it can be set to follow a particular development tree (e.g., the repository head or a specific branch) to reduce the number of false positive changes it detects (e.g., to avoid reporting changes on an experimental branch that will never be integrated into the main development tree). Each GCA engine provides a communication interface for clients to obtain change events based on their generic impact, timestamp, and structural name.

The GCA engine can also compare sets of files, instead of using a source control repository. For example, the changes between the standard libraries provided by Java 1.5 (J2SE 5.0) and Java 1.6 (Java SE 6) can be inferred by comparing the source of the two versions. The drawback of performing an analysis at the file-level is that all the changes are grouped into a single revision set and the additional metadata provided by the SCM repository is absent. By not requiring access to the SCM repository, the GCA can compute changes for situations where the SCM repository is unavailable (e.g., prior to Java SE 6 the Java standard library SCM repository was not publicly accessible). This mechanism also enables changes to be determined on a release-by-release basis by only analyzing significant copies of the repository, rather than incremental changes as they are made.

Our GCA engine prototype implementation utilizes the Change Distiller system [14] as a preliminary means to analyze the changes. By analyzing the abstract syntax trees (ASTs) associated with a file before and after a commit, Change Distiller can classify the change into one of 35 different categories. YooHoo collapses some of these categories (e.g., "condition change" and "statement change" are currently treated simply as "implementation change") and adds others (e.g., "entity deprecated") to announce events at a granularity that seems most relevant to detecting API changes that might impact *some* developer in some situation. The kinds of change events currently recorded are listed in Table 2; we currently make no attempt to detect compound events such as refactoring (e.g., a pull-up refactor would delete a method from one type, and add a new identical method further up the type hierarchy). We discuss the costs and benefits of such extensions in Section 6.1.

Potentially impactful events
Non-private type/method/field added
Non-private type/method/field deleted
Non-private field type changed
Non-private method result type changed
Type/method/field visibility increased
Type/method/field visibility decreased
Type/method/field deprecated
Type/method/field undeprecated
Type/method/field finalized
Type/method/field unfinalized
Supertype added
Supertype removed
Implementation changed
Potentially informative events
New branch
New tag
Javadoc modified

Table 2: Change events announced by GCA engines.

4.2 Developer-specific analysis engines

Each DSA engine consists of three subcomponents: *code ownership analysis*, *external dependency analysis*, and *change impact estimation.*

4.2.1 Code ownership analysis

DSA engines can automatically determine the files of relevance to an individual developer. Automatic analysis determines which files a developer has recently committed; a configurable horizon based on file age or number of intervening revisions can be used to ignore files that have not been altered by the developer for some time or for which someone else seems to have taken over the maintenance. The developer can also indicate individual files, packages, or projects that are always to be ignored or always to be watched, overriding the automated analysis.

4.2.2 External dependency analysis

Once the developer's set of watched resources has been identified, the DSA engine then determines the external dependencies that those resources have. DSA engines are currently implemented as plug-ins to the Eclipse IDE, allowing them to utilize project information about depended-upon versions of external projects. By statically analyzing the source code and project information, a DSA engine identifies all the fields accessed, methods called, types accessed, and subtype relationships within the resources of interest.

This set provides YooHoo with a *watch list* of external resources that the developer is dependent upon and is automatically updated on a semi-regular basis to ensure the correct dependencies are being monitored. Changes made to resources that are not in this set are filtered by YooHoo as irrelevant to the developer. YooHoo can then determine which external resources already have associated GCA engines, and points the developer's DSA engine to these. The developer is warned about external resources without associated GCA engines; by proving a location for the appropriate repository—along with the appropriate authentication credentials where needed—YooHoo automatically constructs and populates a new GCA engine for each.

4.2.3 Change impact estimation

DSA engines periodically retrieve change events for the developer's dependent resources based on their timestamp; generally since the engine's last access or from a specific date (if YooHoo is freshly deployed by the developer). Change events not related to the dependencies in the watch list are immediately ignored. All other events are further analyzed to categorize their severity of impact on the developer's code:

- *Impactful events.* An IMPACTFUL event is one that the DSA estimates is likely to break the specific developer's code in some way. These generally involve changes that alter the external appearance of an API.
- *Uncertain events.* These events cannot be ruled out as breaking the developer's code, though it is unlikely. UNCERTAIN events arise due to our imprecise analyses. In some cases a more precise analysis is readily available that a future version of YooHoo could apply (see Section 6.1). Other cases involve potential behaviour-breaking changes that have a remote chance of propagating through an API. Pragmatism suggests that internal implementation changes will very rarely propagate through the API unless the API itself changes, which is, after all, the purpose of an API. In the end, all UNCERTAIN events could be recategorized usefully; we have retained them at this point to study whether they are really problematic or not (see Section 5).
- *Information events.* INFORMATION events indicate a change that the developer might want to be aware of if they have the time to consider alternative implementations, or changes that involve the depended-upon entities that do not definitively have an impact (e.g., updated documentation).

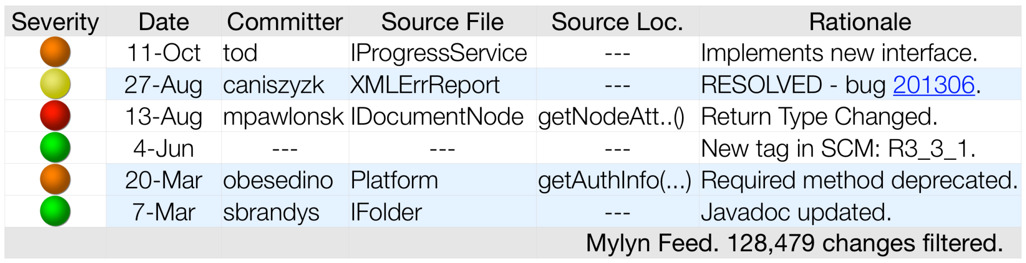
The resulting kinds of events and their severity are listed in Table 3, where "dependency" is defined as statically making reference to the entity in question. The current implementation does not attempt to be very precise in its impact analysis: for example, the removal of a supertype relationship is always recommended for attention (IMPACTFUL) even though it will not matter if the original supertype is not explicitly depended upon (like in a cast expression), if the methods inherited from it are to be found elsewhere in the remaining hierarchy, or if the methods inherited from it are not depended upon at all. We consider the costs and benefits of refining the classification analysis in Section 6.1.

IMPACTFUL **events**
Depended-on type/method/field deleted
Depended-on field type changed
Depended-on method result type changed
Depended-on type/field/method deprecated
Depended-on type/field/method visibility decreased
UNCERTAIN **events**
Depended-on type/field/method finalized
Type/method/field added to depended-upon type
Supertype removed from a depended-on type
Supertype added to a depended-on type
Implementation changed for depended-on method
INFORMATION **events**
Public type added to depended-on package
Depended-on type/method/field visibility increased
Depended-on type/method/field undeprecated
Depended-on type/method/field unfinalized
New tag in depended-on project source repository
New branch in depended-on project source repository

Table 3: Developer-specific change events as classified by default by a DSA engine.

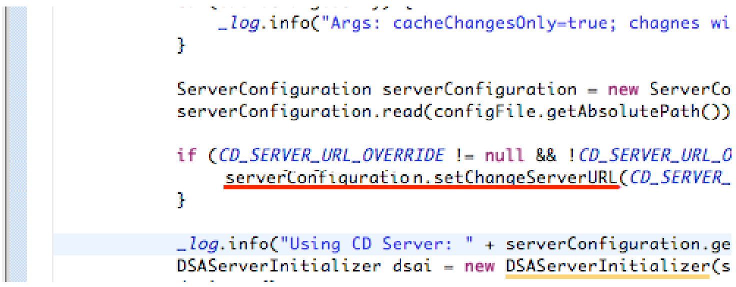
The developer can choose to downgrade or upgrade event classifications on a global or per-project basis. For example, he may be aware that parts of his code depend tightly on the current structure of the type hierarchy (such as which type implements a given method), so changes to the hierarchy will definitely have to be checked: he would likely make supertype changes IMPACTFUL rather than UNCERTAIN in such a situation.

Finally, recommendations for relevant change events need to be announced to the developer. Our current implementation is simple in this regards, presenting results in an Eclipse view for each externally depended-upon project (see Figure 2). As the goal of our prototype implementation is to evaluate whether such a classification and filtration approach would actually significantly reduce the flow of events without missing many important changes (i.e., the usefulness of the approach), this suffices. A more usable, real world deployment could involve in-line notifications (like Eclipse's "lightbulb" marginal annotations) and/or underlining annotations (see Figure 3 for a mock-up). As YooHoo knows both how a depended-upon element changed and how the developer used that element, change notifications can be very specific for the developer. Rather than a generic statement like "`ITableSelectionListener` has been deprecated", YooHoo can be more specific and report, "`ITableSelectionListener` from the `jface` project has been deprecated; your `IResultsView` interface is impacted." This kind of message is much more meaningful for the developer and can even point them in the right direction for resolving the problem.



Severity	Date	Committer	Source File	Source Loc.	Rationale
🟠	11-Oct	tod	IProgressService	---	Implements new interface.
🟡	27-Aug	caniszyzk	XMLErrReport	---	RESOLVED - bug 201306.
🔴	13-Aug	mpawlonsk	IDocumentNode	getNodeAtt..()	Return Type Changed.
🟢	4-Jun	---	---	---	New tag in SCM: R3_3_1.
🟠	20-Mar	obesedino	Platform	getAuthInfo(...)	Required method deprecated.
🟢	7-Mar	sbrandys	IFolder	---	Javadoc updated.
					Mylyn Feed. 128,479 changes filtered.

Figure 2: Developer-specific change awareness view.



```
        _log.info("Args: cacheChangesOnly=true; chagnes wi
    }

    ServerConfiguration serverConfiguration = new ServerCo
    serverConfiguration.read(configFile.getAbsolutePath())

    if (CD_SERVER_URL_OVERRIDE != null && !CD_SERVER_URL_O
        serverConfiguration.setChangeServerURL(CD_SERVER_
    }

    _log.info("Using CD Server: " + serverConfiguration.ge
    DSAServerInitializer dsai = new DSAServerInitializer(s
```

Figure 3: Mock-up of an annotation-based notification alternative.

4.3 Application to the motivational scenarios

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 "Elliott 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 IMPACTFUL changes made to code that he depends upon across any development branch in the repository. In this way he could contact Elliott 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. EVALUATION

Our evaluation addresses four main research questions:

RQ1: "How well is the change event stream compressed?"
RQ2: "Are the IMPACTFUL events actually impactful?"
RQ3: "Are any impactful events erroneously filtered?"
RQ4: "Would a simple approach to surfacing informational but non-impactful events again flood the developer?"

We performed a retroactive analysis of 5 deployed projects to answer the questions. These projects were selected arbitrarily from a set of Eclipse-based tools that we were familiar with that we considered to be somewhat mature. As Eclipse comprises a large number of plug-ins (more than 400 in version 3.5) and also heavily uses many other external projects (more than 75 in version 3.5) it seemed like an ideal platform to study. The 5 projects were:

Metrics: source code metric calculator.[2]
RSSOwl: cross-platform RSS reader.
ASM Plugin: bytecode instrumentation integration.
Colorer: multi-language syntax highlighting.
Checkstyle: coding standard enforcement tool.

Table 4 describes the number and kinds of external dependencies in each of the analyzed projects.

5.1 Methodology and results

In order to watch change events on these 5 projects and various others that we tested YooHoo on while developing it, we ultimately created GCA engines for 70 individual projects (including 47 Eclipse projects and 7 Apache projects). These repositories consisted of the entire length of history we could extract from each project's source repository; in the case of Eclipse this history was generally available from 2001 to present. This required analyzing more than 210 MLOC of source code. The GCA repositories ultimately contained information about 640,000 revisions to 65,000 resources by 370

[2]http://metrics.sf.net

System	External dependencies			
	Projects	Types	Methods	Fields
Metrics	17	229	736	347
RSSOwl	7	139	502	232
ASM	19	193	461	75
Colorer	13	102	242	58
Checkstyle	10	117	233	64

Table 4: Quantity of external dependencies.

Project	**External Changes**					**YooHoo Recommendations**			
(or dev.)	**Start**	**End**	**Rev. sets**	**Revs.**	**Events**	**Impactful**	**Uncertain**	**Info.**	**Irrelevant**
Metrics	06/2004	06/2008	29,770	135,972	619,175	12	1,725	396	617,042
sauerf						9	674	273	618,219
donv70						0	790	60	618,325
RSSOwl	06/2004	06/2008	6,137	42,377	98,941	7	2,627	416	95,891
bpasero						3	2,323	375	96,240
ijuma						0	1,068	166	97,707
ASM Plugin	05/2005	06/2008	6,502	119,912	580,810	5	1,347	307	579,151
Colorer	06/2004	06/2008	14,274	73,199	223,527	12	1,004	309	222,202
Checkstyle	06/2004	06/2008	17,183	84,591	382,664	6	604	266	381,788

Table 5: Scale of revision sets and # of external changes made during the evaluation period.

developers resulting in over 2,750,000 change events. While populating this initial repository took several days, keeping the repository consistent took less than one minute per indexed project per day (and this is fully parallelizable).

RQ1: Event stream compression.

For each of the 5 analyzed projects, we selected a start and end date for the analysis. We built each project and extracted its structural dependencies, as these existed on the start date. Building these projects correctly several years after the fact was laborious[3]; thus, we did not revise our analysis of the structural dependencies for the project for the length of the study (ultimately, this choice did not appear to cause problems with the results of the study). We generated the DSA change feeds for each project, as well as at least one specific developer on that project, on the end date. This list comprised all the impactful, information, and uncertain events for the developer or project, as well as a count of the irrelevant events. These results can be found in Table 5, clearly demonstrating the answer to research question RQ1: the cumulative compression of these change event streams is well over 99%.

RQ2: Correctness of impactful recommendations.

To analyze the accuracy of YooHoo's classifications, we examined the filtered event stream for a putative developer who "owned" the entire project to be analyzed. The small number of IMPACTFUL events were manually inspected to determine their nature and whether a corresponding change was ever made (by August 2009) to adapt to the change.

By building the version of the system from the start of the study period but with the version of the project's external dependencies at the end of the study period, we were able to use the code itself as an oracle: any compilation errors that arose should have been recommended as impactful by YooHoo. We compared the compilation errors against the YooHoo IMPACTFUL events; any error that was not reported as IMPACTFUL was considered a false negative. We also looked at the state of the code to confirm that each error listed as impactful by YooHoo was also really fixed by a developer; these were considered true positives, while if the developer did not make the change, or they were not really impactful, they were considered false positives. Some events are deliberately recommended for attention although the developer has the option to do nothing about them—in particular, deprecation events; we always count these as true positive events, regardless of whether an intervention ensued, although we did ensure that the deprecation event really happened and was applicable to the developer.

[3]Note that this difficulty is purely an artifact of our evaluation. YooHoo does not need to build projects to perform its function.

RQ3: Missing impactful notifications.

Negative events were so voluminous that manually determining if any of these events should have been impactful was impractical. We made three alternative pragmatic attempts to find false negatives. (a) We note that the majority of the event kinds that could cause new bugs would also cause compile-time issues, so compilation was again used to identify any problems that would have arisen from non-recommended events without corresponding changes. Implementation changes that did not affect the structural dependencies of the code base would not have resulted in problems from the compilation test procedure; errors arising from such changes cannot be perfectly detected, but we made two attempts to look for them, as follows. (b) We wished to run the test suites to ensure whether the analyzed projects would have deployed reliably, but we could not locate non-trivial test suites for these 5 projects. (c) Instead, we attempted to run the projects and exercise the functionality manually; this did not yield any additional false negatives.

The results from these analyses are shown in Table 6, wherein only IMPACTFUL events are considered positives (true or false). In response to research questions RQ2 and RQ3, we estimate that—on average for these 5 systems—actual positive events are classified as positive 93% of the time (the true positive rate, TPR) and actual negative events are misclassified as positive 0.000007% of the time (the false positive rate, FPR). This means that impactful change events are almost always recommended by YooHoo and that non-impactful changes are almost never falsely recommended by YooHoo. Looking at the detailed results (Section 5.2), we can see specific targets for improvement. In contrast to Table 6, a developer who is not keeping appraised of any changes would have a precision, recall, and true positive rate of 0, but would not have to proactively consider any change events until his code breaks; since YooHoo provides results with little noise or omissions, this does not seem to be a good alternative.

RQ4: Volume of information events.

Finally, we can see that, despite a potentially large number of change events being classified as INFORMATION, the de-

Project	TP	FP	TN	FN	TPR	FPR
Metrics	10	2	619,163	0	1.00	∼0
RSSOwl	0	7	98,934	0	—	∼0
ASM	2	3	580,804	1	0.67	∼0
Colorer	10	2	223,515	0	1.00	∼0
Checkstyle	5	0	382,658	1	0.83	0
cumulative	27	14	1,905,074	2	0.93	∼0

Table 6: Evaluation of the quality of the results provided by YooHoo for the evaluation period.

veloper would not have been hard-pressed to keep up with this trickle of information—in the worst case here, 416 INFORMATION events in 4 years translates to one message every other working day. On the other hand, one would need to be careful before simply re-categorizing the UNCERTAIN events as INFORMATION: with about 4,000 events combined in the worst case, this translates to roughly 4 messages every working day. While not a flood, splitting one's attention between multiple projects, falling behind on the notifications for a few days could easily turn into an ignored and thus useless backlog. Research question RQ4 is answered by saying that the INFORMATION events alone are few enough that they would not be overwhelming whether or not they are relevant, but that arbitrarily adding more and more events to this category could indeed be overwhelming. Further research into useful informational recommendations is warranted.

5.2 Detailed results

In this section we provide a qualitative overview of the customized change event streams generated during our evaluation for each project.

5.2.1 Metrics

Metrics received the largest number of impactful notifications in our study; at the same time, this project has been around the longest and the majority of its development was performed near or prior to the start of the study period. As such, a large number of its APIs were either deprecated (including 2 full classes) or removed from the repository outright. In particular, the developers had to migrate from `swt.custom.TableTree` to using a new widget that fixed a number of drawbacks in the old implementation. Many of the deprecation notifications (6) were not fixed as they were deprecated after the project went into maintenance-only mode. At the same time, the impactful event list could give a new developer a quick list of actions that should be taken to bring the plug-in up-to-date with the platform.

It is not surprising that `sauerf` received the majority of the impactful notifications; he was the primary developer and owned the majority of its dependencies. `donv70` owned a smaller portion of the code (especially at the start of the study, from which we computed the ownership for his event stream), and received correspondingly fewer events; ultimately none of his code needed to be updated to cope with any external changes.

5.2.2 RSSOwl

All 7 of RSSOwl's impactful events were false positives; in every case they represented the same condition: a method or field was deleted from a subclass and added to a superclass (that was also often newly added). Interestingly, the deletion event and the addition event were often separated by a surprising period of time (weeks or longer). These same false positives affected several of the other projects as they were on common classes (all the `SWT` widgets had their `dispose()` method moved into a supertype).

The developer-specific feeds again insulated individual developers from changes that only affected parts of their project they did not own; in larger projects we would expect this difference to be even more pronounced.

5.2.3 ASM Plugin

An interesting false negative appeared within the analysis of the ASM plug-in. A method, which previously declared a `throws` clause, removed this clause, causing a compilation error in the code. YooHoo had not considered exceptions and did not capture this change; nor does Change Distiller, which we build atop. We will be adapting the code to detect this in the future. The ASM developers also changed their code to adapt to the deletion of `SourceMapper.-fSourceRanges` and `TrimLayout.addTrim(Control, int)`.

5.2.4 Colorer

Colorer also ended up receiving several method and field deprecation events but during the course of study period the project ceased to be regularly updated. That said, as with Metrics the impactful list has 10 true positive changes with only 2 false positive changes that the developer could likely adapt to in a short period of time.

5.2.5 Checkstyle

As one might expect from a project designed to enforce good coding style, the Checkstyle project is very conservative with the APIs they depended upon. As such, few of the APIs they used changed, although they resolved each of the 6 deprecations they encountered. One note of interest for Checkstyle is that a change in `apache.commons.lang.StringUtil` was not detected because we failed to associate the `apache.-commons.lang` project with a GCA engine. While YooHoo detects situations when an appropriate GCA engine cannot be located, the developer must manually fill-in the details to get one populated for the missing project. Since such oversights on the part of the developer might indeed happen in practice, we record this as a false negative.

6. DISCUSSION

In this section, we discuss a number of remaining issues about the approach and ideas for future work.

6.1 Analytic limitations and extensions

YooHoo cannot detect all changes that will definitely impact the developer's code, and only those changes, as this is undecidable in general [21]. Instead we take the approach of ordering and filtering change events in terms of which ones are most likely to matter to the developer.

YooHoo does not consider the inheritance hierarchy in many of its analyses. For example, if a developer's code depends on `Preferentially` but its subtype `ConcretePref-Page` is changed, the developer will not be notified. This decision was made because the developer, by registering a dependency on the interface, implicitly claims to not be dependent on the concrete class; any changes in these situations should be hidden by the interface. Of course, interfaces do not completely shield internal changes from propagating outwards, so there are situations where internal changes would be of importance [9]. Nevertheless, were we to notify the developer of this relatively rare occurrence, we would likely need to also notify her of a plethora of unimportant changes. Thus, this was a pragmatic design decision.

Instead, there are a number of opportunities for more precise analyses to help categorize events. For example, we encountered false positives a few times in our evaluation when a method was moved via a pull-up refactoring: YooHoo reported method deletions involving depended-upon methods, despite the fact that the methods were now inherited by the original type. Some of our UNCERTAIN events are categorized as such because there can be unusual cases where such changes could cause compile- or load-time failures; making a class final, for example, is only a problem if your code had been extending that type—this should be simple to detect. The implementation changes that YooHoo frequently detects can be much more difficult (even impossible) to decide whether they will be impactful or not, but due to developers' desire to avoid API changes, they should not be impactful without a significant accompanying change in the documentation; more heavyweight impact analyses could be brought to bear here, but it is not clear that this would be worth the cost. There are two significant drawbacks to adding such analyses: the occasional difficulty in correctly implementing and validating them, and the potential performance cost required to run them. The approach, after all, is intended solely for providing recommendations: keeping it lightweight matters for practicality.

6.2 Results and threats to validity

The results of our retroactive analysis confirm our hypothesis that the majority of change events are trivial or irrelevant to a particular developer; thus, these can be elided from a change event stream customized for them.

The external validity of our experiment is threatened by the fact that we used 5 specific systems and a particular 4-year period for the analysis; the results may not apply to other systems or other periods. Each of these systems is of non-trivial scale and activity. These systems come from different domains and developers, reducing the likelihood that there is something non-representative about them; they were also in differing modes of development (maintenance versus active extension). These systems are used by many other projects, making them reasonable targets.

The construct validity of our experiment is threatened by the fact that we performed a post-hoc analysis of change repository data to determine what would or would not have been of interest to the developer. The inference needed in this analysis could have led to false results; however, the steps we took to determine whether supposed IMPACTFUL events actually would have been, and whether any non-IMPACTFUL events would have caused a failure at deployment time demonstrates that the results are a reasonable estimate of the actual true positive rate and false positive rate for this analysis.

6.3 Performance

Analyzing a changed pair of files takes an average of 500 ms on a commodity desktop machine. While this results in a significant one-time cost for large repositories (e.g., `org.eclipse.jdt.ui` contains 72,128 revisions, requiring 10 hours for initial database population), the costs of maintaining a repository once it is built is relatively low. For example, YooHoo would take only 6 seconds of analysis per day to keep the `org.eclipse.jdt.ui` project current. GCA engines can be spread arbitrarily across machines; the primary point of contention is the GCA dispatcher, but this just maintains a lookup table mapping types to URLs of GCA engines and can easily scale to tens of millions of types.

6.4 YooHoo for large teams

In many ways, the large-team scenario (Section 2.1) is similar to the plug-in developer scenario. Even in large teams different sub-teams have different schedules, priorities, and goals. In these cases, YooHoo can create GSA engines along with each branch; even though the same pieces of code will be monitored repeatedly across many branches, the nature of developer-centric code ownership makes it likely that very few branches will actually record changes to any specific resource. The results from these per-branch GSA engines can then be aggregated to provide a complete view of the state of the code, to anyone on the team, regardless of what branch they are working on. This can enable developers to more quickly find out about changes, and to contact their originating author about them, before they are integrated into the main repository branch, potentially preventing new problems from arising when the code migrated upstream.

6.5 Maintaining GCA engines

From the user's perspective, YooHoo can transparently handle changes to the underlying kind of repository (e.g., migration from CVS to Subversion) and changes to the location of those repositories (e.g., migrating from SourceForge to Google Code). The GCA dispatcher abstracts the details of these changes away from the DSA engines. In the back end, whichever developer configured the GCA engine for a system that moved from one repository type or location to another must reconfigure the GCA engine (a one line parameter change). The GCA engine automatically updates to reflect the new configuration and notifies the GCA dispatcher of the changes. If a legacy repository is maintained but commits are applied to a new repository location, a silent failure would occur. To combat this, the GCA engine can announce a "cessation of activity" event after some timeout period has elapsed. Manual investigation of the situation could then ensue along with appropriate reconfiguration of the associated GCA engine.

7. CONCLUSION

Software systems are typically heterogenous: they make use of software from different projects and frameworks, not all of which have the same development schedule or are a part of the same organization. Developers do not always have control over when their external dependencies change; typically, if one of these changes causes a problem, their response is both 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 difficulty to keep appraised of relevant changes.

This paper has described the YooHoo relevant change recommendation system. YooHoo provides aggregate streams of recommended changes, tailored to a developer or team, that are customized for their external dependencies. We have demonstrated that these streams reduce the number of change notifications a developer need consider by over 99% while still maintaining a very high true positive rate (over 93%). We have explored the feasibility of also surfacing certain non-impactful events purely for sake of developer-specific awareness, and found that the low volume of such events makes this a promising avenue for further study.

By providing customized change recommendations, developers can be proactive about ensuring that their system stays current with their external dependencies while remaining productive in other tasks.

8. ACKNOWLEDGMENTS

We wish to thank Rylan Cottrell and the anonymous reviewers for their insightful comments. This work was supported by the Natural Sciences and Engineering Research Council through a Postdoctoral Fellowship and a Discovery Grant.

9. REFERENCES

[1] D. Bertram, A. Voida, S. Greenberg, and R. Walker. Communication, collaboration, and bugs: The social nature of issue tracking in software engineering. In *Proc. ACM Conf. Comput. Support. Coop. Work*, 2010. In press.

[2] J. T. Biehl, M. Czerwinski, G. Smith, and G. G. Robertson. FASTDash: A visual dashboard for fostering awareness in software teams. In *Proc. SIGCHI Conf. Hum. Fact. Comput. Sys.*, pp. 1313–1322, 2007.

[3] F. P. Brooks, Jr. *The Mythical Man–Month*. Addison-Wesley, Anniversary edition, 1995.

[4] G. Canfora and L. Cerulo. Fine grained indexing of software repositories to support impact analysis. In *Proc. Int'l Wkshp. Mining Softw. Repos.*, pp. 105–111, 2006.

[5] 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*, pp. 353–362, 2006.

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

[7] B. Dagenais and M. P. Robillard. Recommending adaptive changes for framework evolution. In *Proc. Int'l Conf. Softw. Eng.*, pp. 599–602, 2008.

[8] D. Damian, L. Izquierdo, J. Singer, and I. Kwan. Awareness in the wild: Why communication breakdowns occur. In *Proc. Int'l Conf. Glob. Softw. Eng.*, pp. 81–90, 2007.

[9] 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 SIGGROUP Int'l Conf. Support. Group Work*, pp. 105–114, 2003.

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

[11] J. Estublier and S. Garcia. Process model and awareness in SCM. In *Proc. Int'l Wkshp. Softw. Config. Mgmt.*, pp. 69–84, 2005.

[12] M. E. Fagan. Design and code inspections to reduce errors in program development. *IBM Systems Journal*, 38(2–3):258–287, 1999.

[13] G. Fitzpatrick, P. Marshall, and A. Phillips. CVS integration with notification and chat: Lightweight software team collaboration. In *Proc. ACM Conf. Comp. Supported Coop. Work*, pp. 49–58, 2006.

[14] 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.

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

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

[17] C. Gutwin, R. Penner, and K. Schneider. Group awareness in distributed software development. In *Proc. ACM Conf. Comp. Supported Coop. Work*, pp. 72–81, 2004.

[18] M. Kersten and G. C. Murphy. Using task context to improve programmer productivity. In *Proc. ACM SIGSOFT Int'l Symp. Foundations Softw. Eng.*, pp. 1–11, 2006.

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

[20] T. D. LaToza, G. Venolia, and R. DeLine. Maintaining mental models: A study of developer work habits. In *Proc. Int'l Conf. Softw. Eng.*, pp. 492–501, 2006.

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

[22] C. O'Reilly, D. Bustard, and P. Morrow. The War Room Command Console: Shared visualizations for inclusive team coordination. In *Proc. ACM Symp. Softw. Vis.*, pp. 57–65, 2005.

[23] 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.

[24] A. Sarma, Z. Noroozi, and A. van der Hoek. Palantír: Raising awareness among configuration management workspaces. In *Proc. Int'l Conf. Softw. Eng.*, pp. 444–454, 2003.

[25] A. Sarma, D. Redmiles, and A. van der Hoek. Empirical evidence of the benefits of workspace awareness in software configuration management. In *Proc. ACM SIGSOFT Int'l Symp. Foundations Softw. Eng.*, pp. 113–123, 2008.

[26] A. Sarma and A. van der Hoek. Towards awareness in the large. In *Proc. IEEE Int'l Conf. Global Softw. Eng.*, pp. 127–131, 2006.

[27] A. Sarma, A. van der Hoek, and D. F. Redmiles. A comprehensive evaluation of workspace awareness in software configuration management systems. In *Proc. IEEE Symp. Vis. Lang. Hum. Centr. Comput.*, pp. 23–26, 2007.

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