Social Visualization in Software Development

Jason Ellis
IBM Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598
USA
jasone@us.ibm.com

Catalina Danis
IBM Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598
USA
danis@us.ibm.com

Christine Halverson
IBM Almaden Research Center
650 Harry Road
San Jose, CA 95120
USA
krys@us.ibm.com

Wendy Kellogg
IBM Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598
USA
wkellogg@us.ibm.com

Abstract

Most software development tools focus on supporting the primary technical work – writing code, managing requirements, filing bugs, etc. Yet with large teams, managing the social aspects of a project can be as complex as managing code. Here, we discuss the iterative design of a visualization that helps developers better understand the social aspects of their work.

Keywords

Social computing; visualization; prototyping; software development

ACM Classification Keywords

H.5.3 Group and Organization Interfaces

Introduction

Many developers in large distributed teams feel overwhelmed. Among other things, they see many change requests (CRs) over the course of weeks – CRs that are in good shape and CRs where the fix is unclear, CRs that need to be examined to determine their priority, CRs that are looking for someone to work on them, CRs with proposed fixes that need to be evaluated. One common concern with developers is that they have forgotten about something important –

they've lost track of change requests that will come back and cause them significant grief in the future.

We saw an opportunity to address these concerns with visualizations that give overviews of data contained but not easily discerned in change tracking systems. Visualizations, which provide quickly graspable views of often complex data, might be able to help developers better understand their world. This idea met with significant interest from developers and managers on large distributed teams at Lotus and Rational. Those who worked on smaller teams saw less need for such a tool. Small teams frequently depend on personal communication to keep abreast of important issues. Thus, our target audience became large distributed software development teams.

Change Tracking Systems

Change tracking systems (sometimes called bug or issue tracking systems) help developers manage and prioritize their work. This is a highly social activity in the sense that decisions regarding the management of work are made through social interaction such as debate, consensus, or managerial edict. It is an activity that involves people in a variety of well-defined roles from managers to developers to testers to requirements writers, and so on.

For the discussion here, consider change tracking systems to be repositories containing textual descriptions of changes to software (defects, features, etc.). These descriptions are called change requests (CRs). Each CR features a detailed discussion surrounded by additional attributes like who is assigned to work on it, associated code, the current state of the change, and much more. As such, each CR can be seen as a kind of anchored discussion [2].

Change tracking systems are customizable and, thus, support a variety of approaches to software development. One increasingly common practice in distributed software development projects involves vetting a proposed fix prior to incorporating it into the software. A developer devises a piece of code to address the change request, then attaches it to that change request so that it can be vetted and approved by project managers. When the change is approved, it is checked into the source code repository and the associated CR is noted. As such, change tracking systems are one of the central organizing mechanisms in many software development projects. Popular change tracking systems include Mozilla Bugzilla, Rational ClearQuest, and IBM CMVC.

Method

Our design is based on five types of data: (1) E-mail-based interviews with seven developers in multi-person development projects at IBM Research and Apple, (2) eleven face-to-face interviews with developers working on large projects in Apple, Rational, Lotus, ibm.com, IBM Research, and Mozilla, (3) analysis of functionality in existing change tracking systems such as CMVC, Bugzilla, and ClearQuest, (4) in-depth analysis of specific change requests in Mozilla's Bugzilla database, and (5) numerous design documents produced through the process of iterative design with stakeholders.

Initial Fieldwork

Based on analyses of the interview data, we compiled the main issues programmers wanted help with. Examples include: (1) a project overview that gives a feeling for where new change requests are appearing, who owns them, and where work is being concentrated, (2) a better understanding of how long CRs have been around as compared with their severity, (3) more clarity on the workload of each developer past, present, and future, and (4) a way to discover problematic patterns, particularly situations where a CR is repeatedly resolved and reopened or assigned and reassigned. The remainder of this paper will focus on this fourth problem.

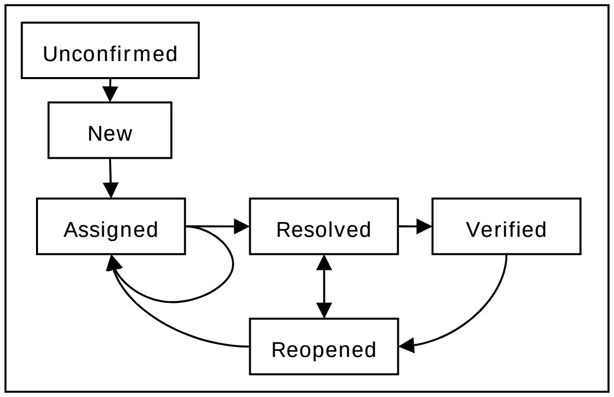


figure 1. Simplified Mozilla Bugzilla state transition diagram

One of our interviewees in Lotus referred to the assign/reassign pattern as the “ping-pong problem.” He describes the problem this way: “In product, ping pong is a serious issue – especially for large projects across multiple sites. It’s of particular concern when you’re dealing with multiple cultures and understandings of English – things get complex in terms of bug description and what has been done to resolve the bugs. It’s too complicated to get this information with current tools.” We provide more detail on this problem in the following section.

Problematic Patterns

A state machine loosely governs how CRs move through change tracking systems (see figure 1). Transitions between states are generally not automatic, instead requiring explicit human intervention to move a CR from one state to another. In Mozilla’s Bugzilla repository, the best-case path to resolution goes through five states, but many other paths are possible and some paths are indicative of problems [3]. For instance, interviewees told us that CRs that are repeatedly resolved and reopened or repeatedly reassigned are worth looking into more deeply. A resolve/reopen cycle could mean that there is disagreement about what it means to fix a problem – someone (perhaps several people) keep thinking the CR has been addressed and others feel that it has not. One way to think about this is as an argument. An assign/reassign cycle, on the other hand, means that the CR is not finding the right owner. Instead, each assignee looks at the CR and decides that they aren’t the right person to work on it. This could indicate a number of problems including a structural problem in the software or an organizational gap.

The history of these state changes along with data about the people who made them can help us uncover problematic patterns. However, getting this kind of information from existing change tracking systems is complex enough that interviewees reported it is seldom done, if ever. In Bugzilla, one must follow these steps:

1. Use the query interface to find a CR of interest

2. Navigate to the CR’s history page. The history is an (often) long date-ordered list of the modifications to the CR in question. Most of these are likely not to be related to the aforementioned problematic patterns

3. Cut the CR history down to the specific modifications needed to identify problematic patterns. For the most part, this means throwing out everything but the state changes

4. Read through the (often multi-paged) data and decide if a problem exists

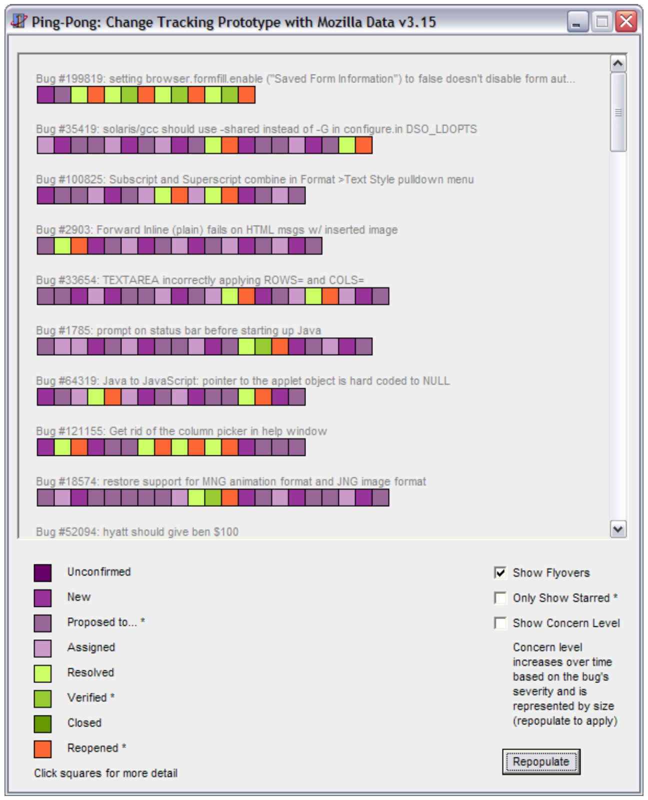
In CMVC, simply generating a CRs history is an expert-level task involving multiple hand-written queries. Thus, the design question becomes: Can we make finding these patterns less difficult?

Design Evolution

Following our initial stakeholder interviews, we built prototypes (not all discussed in this paper) to address a variety of problems and did iteration with stakeholders in Lotus and Rational using those prototypes in the context of feedback interviews. The earlier prototypes we developed gave an overview of the system at a particular snapshot in time. This is useful for getting an understanding of the current state of things – where the action is today or what current severity levels are. However, looking at a snapshot also misses some important issues. For instance, if one looks at a particular CR today, it might be assigned to someone knowledgeable and be in a state that seems appropriate for work to progress. However, a look at the history of the CR may show that it has, in fact, been assigned to numerous people over the past six months. Or, that it has been resolved repeatedly, only to be reopened. Or both. We respond to these issues in the prototypes presented here.

Problematic Patterns Prototype 1

The starting point for our thinking about this prototype was the Task Proxy [1], a visualization technique that gives an overview of two-state (not finished/finished) tasks in social context. The challenge we are confronting in this paper is the meaningful display of CRs, a significantly more complex task representation.


Ping-Pong: Change Tracking Prototype with Mozilla Data v3.15

Bug #199819: setting browser.formfill.enable ("Saved Form Information") to false doesn't disable form aut...

Bug #35419: solaris/gcc should use -shared instead of -G in configure.in DSO_LDOPTS

Bug #100825: Subscript and Superscript combine in Format >Text Style pulldown menu

Bug #2903: Forward Inline (plain) fails on HTML msgs w/ inserted image

Bug #33654: TEXTAREA incorrectly applying ROWS= and COLS=

Bug #1785: prompt on status bar before starting up Java

Bug #64319: Java to JavaScript: pointer to the applet object is hard coded to NULL

Bug #121155: Get rid of the column picker in help window

Bug #18574: restore support for MNG animation format and JNG image format

Bug #52094: hyatt should give ben $100

■	Unconfirmed
■	New
■	Proposed to... *
■	Assigned
■	Resolved
■	Verified *
■	Closed
■	Reopened *

Click squares for more detail

[✓] Show Flyovers
[ ] Only Show Starred *
[ ] Show Concern Level

Concern level increases over time based on the bug's severity and is represented by size (repopulate to apply)

figure 2. Problem patterns prototype 1 with Mozilla data

The goals for this prototype were to (1) display only portions of the CR history relevant to the patterns we were trying to highlight and (2) display the entire history as compactly as possible to facilitate quick pattern recognition by users. To this end, we built a visualization in Borland Delphi that lays out the state changes for a number of CRs side-by-side. Showing the state changes in this way provides for a compact display of each CR's history state-wise. Each state change is represented by a box and colored based on the kind of state change it is.

Following a round of feedback from stakeholders, we refined the prototype by adjusting the state colors to make it easier to detect patterns (see figure 2). We used purples for the progression from unconfirmed to assigned (where code is written) and greens for the path from resolved to verified (where code is tested and verified). The reopened state indicates a problem in and of itself, so we made it stand out using a brighter orange color. This iteration of the prototype was the first to run against real data: CRs from Mozilla's Bugzilla database.

This prototype served as an initial proof of concept and allowed us to share our ideas with stakeholders. Feedback was largely positive, with the majority of stakeholders asking when it would be available for use against their change tracking data.

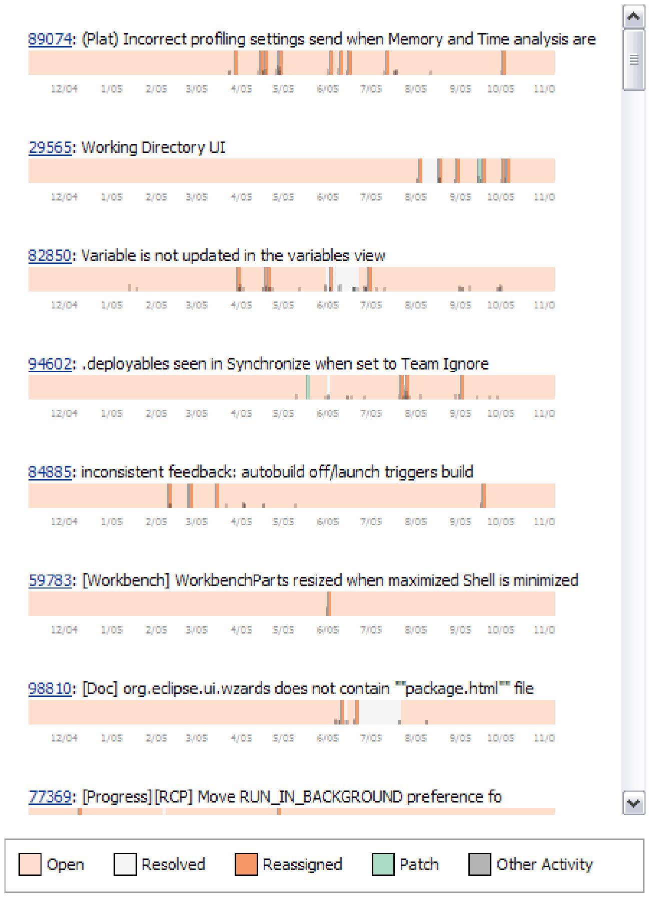
A significant new issue arose in these discussions as well. While it is nice to be able to see a CR's entire state history in one compact display, where the display gains in compactness it loses in specificity. For instance, the prototype shows *that* reassign or resolve/reopen cycles happened, but the only way to find out *when* they happened is by mousing clicking on each block. This is a problem because a CR may look like it is exhibiting a problematic pattern but when we take a more detailed look at its history, we find that the problems happened years ago. At the same time, CRs that do have recent problematic patterns become more difficult to see because they are indistinguishable from those that do not. Our second prototype was designed to address this issue.

Problematic Patterns Prototype 2

Based on the above feedback, we chose to relax the requirement of showing every state a CR has passed through in favor of showing near-term history. We built a new Java prototype that shows only the past year (see figure 3). Each pixel going across the display is a day, dark orange bars in the display are reassigns, and green bars are when patches were provided (these are code – proposed fixes for the CR). If the background is orange, the CR is open and if it is white, the CR is closed (resolved). Lastly, dark lines show any of the myriad other types of operations that are done to CRs (comments, people added to cc list, priority changed, and so on). A higher bar means more of these operations were done on the given day.

This approach gives stakeholders a better idea of time with respect to the CR. Where the previous prototype shows only that one event happened after the next, this prototype allows users to see how temporally close or distant these events are. For instance, users can see that there were numerous resolve/reopen cycles for a particular CR in the middle of the year but none recently. Or, they might see that the CR and was repeatedly reassigned over the past few months but a patch was just recently supplied which means the CR might be nearly resolved. Lastly, this design allows stakeholders to see when a CR goes from high activity to no activity, something that was not visible in the previous prototypes (some call these CRs "zombies").

This prototype shows similar problematic patterns when run against both Mozilla and Eclipse data. Thus, this approach has utility across at least two open source projects. While we have not run the prototype against corporate data, our partners in Rational and Lotus felt it would have utility for them as well.



89074: (Plat) Incorrect profiling settings send when Memory and Time analysis are

29565: Working Directory UI

82850: Variable is not updated in the variables view

94602: .deployables seen in Synchronize when set to Team Ignore

84885: inconsistent feedback: autobuild off/launch triggers build

59783: [Workbench] WorkbenchParts resized when maximized Shell is minimized

98810: [Doc] org.eclipse.ui.wzards does not contain ""package.html"" file

77369: [Progress][RCP] Move RUN_IN_BACKGROUND preference fo

figure 3. Problem patterns prototype 2 with Eclipse data

Conclusion

Current change tracking systems are pregnant with social data that can help developers better understand and manage their work world. However, this data is often hidden. In this paper, we have presented one way to surface such data: visualizations of social patterns in change requests. Making this data available for use by developers has the potential to help them better manage the social aspects of software development, alongside more traditional practices.

As this work moves forward, we will continue doing iterative design with stakeholders. We will explore providing drill-down to allow users to investigate events on the timeline in more detail. For instance, we hope to provide the ability to see exactly which activities compose the “other activity” bars as well as provide more information on the events that are already called out. We would like to make people more prominent by showing who initiated each activity. For instance, a CR might be shown to have been reassigned ten times in the current visualization. Making who the CR passed between clearer could allow users to see that assignment was cycling between just two people, which tells us something different if it were passing among a larger group with few repeated names. We also plan to use this visualization as a tool to uncover new classes of problematic patterns that we can highlight alongside the patterns we have shown here.

References

[1] Erickson, T., Huang, W., Danis, C. and Kellogg, W. (2004). A Social Proxy for Distributed Tasks. CHI 2004.

[2] Guzdial, M. (1997). Information Ecology of Collaborations in Educational Settings. CSCL 97.

[3] Mozilla Bugzilla Lifecycle http://www.bugzilla.org/docs/2.18/html/lifecycle.html.