# Task and Social Visualization in Software Development: Evaluation of a Prototype

**Jason B. Ellis, Shahtab Wahid,* Catalina Danis, Wendy A. Kellogg**

Social Computing Group
IBM T.J. Watson Research Center
P.O. Box 704, Yorktown Heights, NY 10598
{jasone | danis | wkellogg}@us.ibm.com

* Center for Human-Computer Interaction
Virginia Tech
Blacksburg, Virginia 24061
swahid@vt.edu

**ABSTRACT**

As open source development has evolved, differentiation of roles and increased sophistication of collaborative processes has occurred. Recently, we described coordination issues in software development and an interactive visualization tool called the Social Health Overview (SHO) developed to address them [12]. This paper presents an empirical evaluation of SHO intended to identify its strengths and weaknesses. Eleven informants in various open source roles were interviewed about their work practices. Eight of these participated in an evaluation comparing three change management tasks in SHO and Bugzilla. Results are discussed with respect to task strategy with each tool and participants' roles.

**Author Keywords**

Social visualization, task visualization, change tracking systems, information visualization, coordination of work, software development

**ACM Classification Keywords**

D.2.2 [**Software Engineering**]: Design Tools and Techniques – *evolutionary prototyping, user interfaces* H.5.3 [**Information Interfaces and Presentation**]: Group and Organization Interfaces – collaborative computing, computer-supported cooperative work.

**INTRODUCTION**

This paper is about a visualization called the Social Health Overview (SHO) that was designed to support open source software (OSS) development, in particular those aspects of development that comprise the reporting, assignment and resolution of change requests (CRs), including defects (bugs) and enhancements.


*CHI 2007,* April 28–May 3, 2007, San Jose, California USA.


The OSS development approach has captured the interest of many, including researchers and practitioners in software development, because of the widely publicized success of projects such as the Linux operating system and the Apache server [15,23]. Early reports, especially in the popular press, highlighted the ease with which successful software products appeared to simply *emerge* [15,23]. However, as studies of OSS projects appear in the literature, a more balanced view of OSS is emerging. Well-known issues in the commercial development world, such as the challenges of coordinating distributed work, also apply in OSS and new issues arise because of its participation characteristics.

One widely reported new problem, part of the motivation for this paper, stems from the large number of CRs that can be generated in OSS development projects. For example, Anvik and colleagues [1] reported that almost 3500 CRs were logged in the Eclipse open source project in one four-month period, or almost 30 per day. Significantly, their analysis showed that 36% of the reported CRs were later classified as inappropriate: they could not be replicated, they were duplicates, or they were marked 'out of scope' and therefore not addressed by the development team.

This high level of noise in change requests taxes an already overburdened group that does "triage:" vetting each CR and assigning it to someone for resolution [1,15]. Both the increase in volume and in noise result in part from the large number of participants in OSS projects, many of whom are involved only marginally and therefore may not be attuned to the realities of the project. For example, in their analysis of the Apache server OSS development community, Mockus and colleagues found that of 3,060 individuals who submitted a total of 3,975 CRs, a subset of 458 individuals submitted the 591 CRs that were addressed in subsequent code releases [15]. Thus, increased noise is a downside of the openness of OSS development.

The large volume of reported CRs and associated noise also has repercussions for the rest of the OSS team, especially for those who are responsible for "driving" or managing a project. As with collaboration generally, software development requires coordination for achieving goals [11,12,13,17,21], whether through formal or informal mechanisms [11,13,17]. While the literature varies on the

degree of self-organization in successful OSS efforts, there is general agreement that the larger the project, the more need for differentiation into roles and responsibilities.

The differentiation of the team into roles is an important mechanism for guiding an OSS project to completion [15,18,20], much as it has been in other work domains, whether with or without computer support (see [24] for an overview). For example, workflow systems for business processes and access-control systems often include defined roles [24]. Roles in OSS may be self-selected or assigned through a foundation or governing body associated with the project (e.g., the Mozilla Foundation). In their study of two OSS teams, Mockus and colleagues [15] found a variety of roles like "release manager" or "quality assurance lead." In addition, they showed that new roles often emerge during the development cycle. For example, in the Apache HTTP server project, an informal organization of people to guide the project ("the Apache Group") emerged [15, p317].

The perceived need for role specialization has resulted in the development of specialized tools targeted for particular roles. Confora and Cerulo [6] developed a tool for bug assignment to be used by project managers. Anvik and colleagues [1] took a machine learning approach to the same task. Roles and accompanying responsibilities are also evident in CR negotiation processes that determine where resources are allocated in development efforts [20].

Our prototype, the Social Health Overview (SHO), was intended primarily to support OSS drivers, who manage the change tracking process, rather than individual developers, who participate through fixing CRs, or high level managers who have overall responsibility for a project. We saw an opportunity to help drivers better identify the set of CRs on which their team's attention should be focused, particularly for sizeable software development efforts. Importantly, the tool draws on data already available in standard change tracking systems (in our case, the Mozilla Foundation's Bugzilla system). We focused our efforts on an interactive visualization tool because of the ability of visualizations to support discovery and drill-down in large datasets.

In the remainder of the paper, we first give background on change tracking systems, followed by a review of related work in the area of visualization and software visualization. We then describe the findings from user interviews aimed at understanding how informants approached their responsibilities in OSS development efforts. We then introduce the SHO visualization tool and describe the tasks study participants carried out. Finally, we discuss the task strategies used with each tool and the value of the SHO for people who play different roles in OSS development.

## BACKGROUND

### Change Tracking Systems and Distributed Development

Early research on change tracking focused on management through formal methods, whether tools [e.g., 16] or processes (see [9,10] for reviews). In subsequent work, there was a gradual recognition of the role social factors play in addressing CRs. This broadened the research focus to include informal communication, which provides awareness of team members and their work [11,13,17,21], and supports negotiation for the resolution of submitted CRs [5,20].

Mozilla's Bugzilla system (Figure 4) is a cornerstone, not only of the Mozilla OS community, but of many other OSS development projects. It is a richly social place where bugs, features, evangelism requests, meeting planning, and more are discussed and debated, and work is tracked. Because many projects use Bugzilla for tracking a good deal more than just bugs, we refer to it here as a change tracking system, rather than as a bug tracking system. Although this broader use of bug tracking systems is well recognized by the research and development communities (e.g., [18]), this practice is also confirmed by our interviews results. As a further clarification, the "textbook term" for a record in a change tracking system is "change request" or "CR." However, in the vernacular, the term "bug" is more often used. We will use these terms interchangeably.

Our interest in the change tracking task began with a simple observation that accords with that of Anvik and colleagues [1]: A very large number of bugs are filed every day. How do developers manage this? Is it a problem? Are there special mechanisms in place that allow them to cope? When we asked these questions of the driver of a major Mozilla product he replied: "Yes, it's a very big problem! Whenever we get close to a milestone, I'm convinced that I've lost track of something critical that will come back to bite me the day after we ship."

## RELATED WORK

### The Value of Visualizations

Visualizations have been documented as playing important roles in thought and communication. This can in part be understood from the distributed cognition perspective [14], which argues that collaborative tasks draw on both internal and external representations. Working within this perspective, Zhang and Norman [24] discuss the design of an external representation for a hierarchical task which facilitates access to its component tasks. One value of well-designed representations is their ability to offload onto the perceptual system some aspects of cognitive processing [19,22]. Card, Mackinlay and Shneiderman [4] differentiate six aspects of the benefit to individuals: 1) an increase in memory and processing resources available to the user, 2) reduction of the search for information, 3) enhancement of the detection of patterns, 4) enablement of perceptual inference, 5) use of perceptual attention mechanisms for monitoring, and 6) the encoding of information in a malleable medium. By these means, visualizations support the discovery of information and free cognitive resources which can then be applied to problem solving. One overriding design consideration for visualization tools is the placement of the boundary between visualization and

interpretation [9]. A tool that emphasizes visualization builds in flexibility that can be exploited by users with different, but related task needs [9].

### Visualizations of Code

Visualizations have a long history of being applied to helping people understand complex codes. Much of the early work was focused on helping people understand the evolution of structural properties [3]. A tactic shared with this earlier work by the SHO visualization draws on the insight that data created for other goals or in other contexts can be re-purposed for new goals. For example, Ball and colleagues [3] used data generated for version control to assess the impact of local implementation decisions on the final software product.

The SeeSoft tool developed by Eick and colleagues [2,7] is a visualization of source code at the line level of detail. The system provides an overview of the current state and history of large source files. It depicts each line of code by a row of dots and associates with it color-coded statistics from version control systems, static analyzers, code profilers and project management tools. For example, it might show who programmed the line of code, when the line was created or when it was modified. This gives the programmer an overview of the code artifact as it is embedded in the social system of development. The SHO differs from the SeeSoft visualization in focusing on CRs rather than lines of source code, but it shares its focus on actors in the process.

Froehlich and Dourish's [9] Augur system, also based on lines of code, is designed to support activity monitoring in a distributed software development project. It enables programmers to explore the distribution of development activities by augmenting the code with techniques that place it in relationship with both the people involved and its structure. Once again, the Social Health Overview differs in its central focus on CRs rather than code artifacts, but shares the focus on individuals involved in the process.

At least one cautionary note has been raised with respect to the use of visualization by software development teams. Gutwin and colleagues [11] focus on the importance of simple text communication tools, like discussion databases, for supporting awareness of team activities, and caution that visualizations could conceivably "siphon off public interaction from communication channels that help the entire group stay aware." [11, p. 73].

## EMPIRICAL WORK OVERVIEW

Our empirical work has two parts. First, we conducted interviews to understand how our informants work with change requests in their project. Second, we conducted a study in which a subset of the interviewees performed tasks in both SHO and Bugzilla in order to evaluate the utility of the visualization compared to an existing tool. We discuss each piece of work in turn.

## INTERVIEWS

Our overall goal was to conduct a comparative evaluation of SHO and Bugzilla with drivers of large components or software projects. But to understand what our informants might do when they carried out our tasks in each system, we wanted to have a more detailed understanding of their job responsibilities, what was hard, what worried them, how they used Bugzilla themselves and across their team.

### Interview Method

We conducted semi-structured interviews with eleven informants who were familiar with Bugzilla and engaged in OS development as drivers, committers, or triagers. We found potential participants through contacts within and outside our company. Informants were actively involved in development projects under Rational, Eclipse, and Mozilla.

Interviews generally lasted about thirty minutes and were conducted by phone. They were digitally recorded and later loosely transcribed. Handwritten notes also were taken by a second researcher during each interview.

Interviews began with the informant's position, role, and team. Informants were then asked to explain how they (and their team) handled bugs on a daily basis. Topics included descriptions of the change tracking process, the number of incoming CRs, methods for prioritizing and keeping track of CRs, and how they (and their team) used Bugzilla. The final part focused on how informants monitor and detect problems within their projects: We asked what they worry about in their role, how they monitor project health, outstanding bugs, comments, state changes, and the like.

### Interview Findings

Informants spanned a variety of roles, from quality assurance to developer to project manager. People in each role participated in different aspects of the software development process and reported interacting with Bugzilla and change requests in different ways. For example, some belonged to a development group and spent most of their time writing code and doing their own bug triaging. Others belonged to QA teams and focused most of their effort on handling incoming bugs and directing them to the correct team. Others were middle or high level managers focused on managing releases or overall guidance for the project.

We did find issues and concerns that were shared across roles, even if they affected the work somewhat differently: for example, difficulties that resulted from large numbers of bugs. We provide details in this section.

#### *Bug Volume*

Ten of eleven informants expressed concern about losing track of important bugs in Bugzilla due to bug volume. One team lead put it this way: *"I worry about things falling through the cracks. We're supporting an agile approach but you can't be agile if you're not responding to things. Sometimes people see the bug system as a black hole and that's not good."* Another developer simply said: *"I feel like I'm constantly drowning."*

Other informants were worried about what one person termed *"losing the battle of the inbox"* (where new bugs are placed). In most processes described to us, new bugs had to be vetted before being assigned. 'Losing the battle' meant more bugs were coming in than anyone had time to look at.

Informants also expressed dismay with what many called the "signal-to-noise ratio." Noise included trivial bugs, bugs with bad descriptions, obscure or hard to reproduce problems, and especially duplicates. In Mozilla and Eclipse, over 50% of bugs filed are duplicates of existing bugs, meaning they should never have been filed. The incidence is so high that several informants claimed "Most bugs are duplicates." Many pointed out that it takes resource to discover each of these duplicates and close them.

Some felt that the problem of bug overload is getting worse. One who serves in a QA role on the Mozilla project put it this way: *"It's known that Bugzilla's database is growing and growing and there are still more bugs […] it's getting harder to find the bugs."*

Finding that people across roles have difficulty dealing with large bug volumes and are anxious about the possibility of important things falling through the cracks is significant for the approach taken here. Creating a compact view of a large amount of information that can draw attention or be quickly filtered is one of the strengths of visualization.

### *Work Practices Reflected in Bugzilla Use*

Our interviews also revealed a variety of ways in which Bugzilla is used by development teams. The QA lead of one team told us: *"All roads lead to Bugzilla. It's our documentation system, our bug reporting tool, our planning tool, our internal infrastructure IT system, and more."*

The extent to which Bugzilla was used for communication and coordination varied from team to team and seemed correlated with team size and the degree to which they were collocated. A developer from a small collocated team said: *"Bugzilla is not now our main organizing principle. We more use meetings and milestones to communicate. Of course, with my team we are all in the same office, so we see each other face to face."* But a development lead for a much larger distributed team said, *"E-mail or talking never work. […] Not everyone uses their [company name] mail all the time. We just use Bugzilla for communication. […] We always try to put as much information in the [Bugzilla] discussion so that everyone can follow [it]."* Those in small groups or higher in management reported relying more on interpersonal communication. Larger teams often enforced use of Bugzilla, fearing that communication would get lost.

Thus, team work practices were clearly reflected in how Bugzilla was used. Knowing how and how much some teams use Bugzilla is important for designing an effective visualization: Not only does it suggest what information will be most useful, but the more a team uses it, the more useful tools derived from that data are likely to be.

### *Deciding What to Work on Next*

One thing informants had in common was having to prioritize work at one level or another. For those in quality assurance, prioritization emerged from the triage they performed – going through all the new bugs, making sure they were well formed, and vetting the severity rating. Completing these activities both enabled and prioritized the work of other team members.

For development leads, prioritization meant making sure the developers on their teams were working on the right things to ensure that milestones were met. One lead put it this way: *"First is show me all unassigned bugs that are in the NEW state. Show me the bugs that we've got targeted for the current milestone – making sure the right bugs are targeted and where people are working. Once a month, I look at the 'show me everything' query."*

For project managers, it meant making decisions about the higher level goals of the project and the bugs and features that should be part of that plan. A project lead explained: *"I don't do any triage on new bugs. I get involved when a decision needs to be made. [Decisions like] significant areas where we're going to do new work – guiding the project forward."*

While people in different roles contribute differently to deciding what to do next, the fact that people at all levels of the development hierarchy affect this process speaks to its significance. The SHO visualization aims to give users a variety of ways to reason about work priority. We turn now to a description of the system.

## THE SHO VISUALIZATION

The Social Health Overview provides an overview of all the open bugs in a Bugzilla installation. The name derives from *project* health tools, which are commonly used to evaluate the progress of software projects via development artifacts. In contrast, SHO mines the history of development artifacts to reveal social and historical patterns of interest underlying the development process.


**Figure 1. Flyover of a change request. Pointing to it results in a flyout tag that summarizes key information. Clicking on it brings up the underlying details in a Bugzilla window.**

In SHO, each bug is represented as a circle. The default settings show each circle colored by the pattern that is most prominent on that bug and sized based on "heat" (Figure 1). Heat is an aggregate measure that combines all the patterns

the bug exhibits. In theory, the bigger the bug, the more attention it should be paid. Mousing over a bug produces a tooltip that provides additional information. The patterns the visualization displays are:

- Assign/Reassign: The number of times a bug has been reassigned.
- Resolve/Reopen: The number of times a bug has been reopened.
- Blocking: The number of bugs that cannot begin work until this bug is fixed.
- Zombie: The number of days a bug has not been touched.
- Patch: The number of days a bug has had an unevaluated patch attached.
- Age & Severity: Combines the age of a bug and its severity (brand new major bugs may be ranked below new normal bugs by this metric).
- Popularity: Combines discussion, votes, number of people on the cc list, and duplicates.

The initial view (not shown here) shows bugs in their associated components, but this display can be adjusted in a number of ways (see Figure 2). First, users can zoom in and out arbitrarily to focus on portions of the bug space in which they are interested. Second, users can change the X-axis and Y-axis breakdowns to show bugs by component, age, or severity.

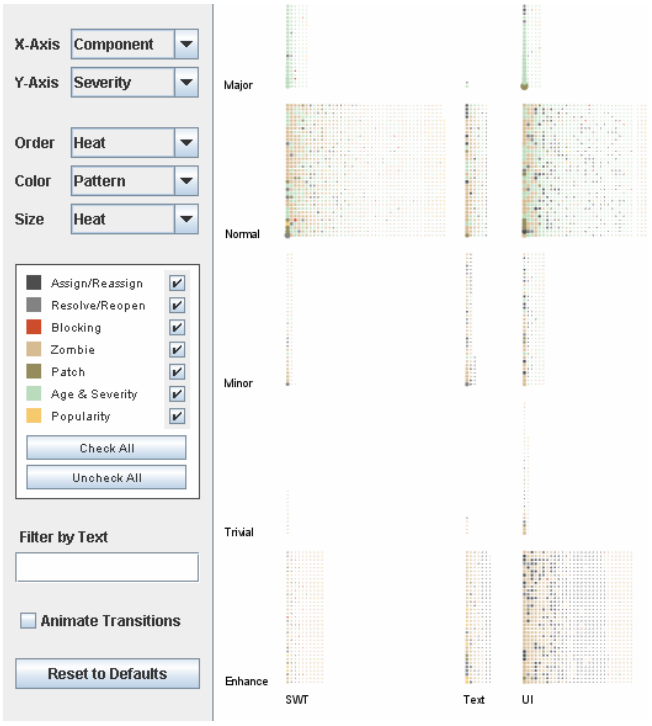


**Figure 2. Social Health Overview with X-axis set to Component and Y-axis set to Severity. Controls at left.**

For instance, Figure 2 shows the visualization with the X-axis set to Component and the Y-axis set to Severity. This allows users to compare across components and see that there are more bugs marked with severity Enhancement or Normal than Trivial, Minor, or Major in nearly all cases.

Bugs can also be ordered and colored by attributes like age, assignee, heat, and pattern. They can be sized by heat or made all the same size. Bugs with particular patterns can be hidden. For instance, if bugs that exhibit the Zombie pattern are not of interest, they can be excluded from the display.

Lastly, users can search and/or filter the display using text strings. Bugs that do not contain the string are dimmed. This is useful if one wants to highlight bugs owned by a particular person or group of people, for example.

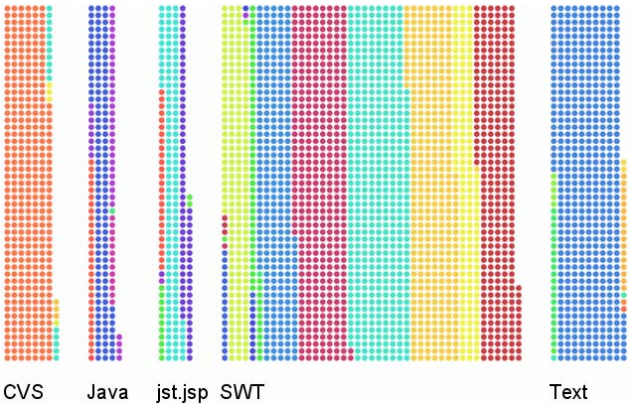


**Figure 3: Close-up of bugs colored and ordered by Assignee, with X-axis set to Component.**

Using several settings in conjunction with one another can produce useful effects. Figure 3 shows bugs ordered and colored by assignee, resulting in all bugs assigned to each

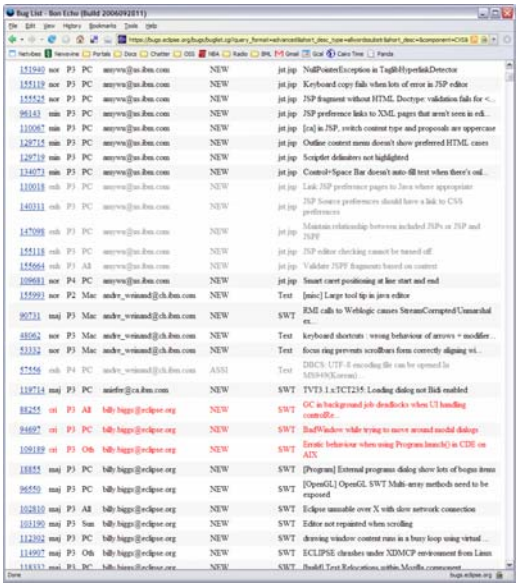

**Figure 4: Bugzilla showing results of the same query that produced Figure 3.**

developer being clumped together. The first page of results for the same query in Bugzilla is shown in Figure 4. The visualization allows users to see how bug ownership in each

component breaks down. This is useful when one needs to get an understanding of how much work a developer has relative to others. Showing this display with bugs sized by heat (rather than all the same size as shown here) introduces the additional consideration of urgency.

One also can break down the display vertically. Figure 5 shows the data in Figure 3 broken down vertically by Age. This allows users to see how assignments are changing over time, providing a way (among other things) to view and debug team processes. In addition, one can filter the display textually. For example entering "inbox" into the filter field (not shown) will dim all assignees except inboxes. In Figure 5, the inboxes that would be revealed in this way are outlined with bold lines.

Consider the inbox CRs shown in Figure 5 (i.e., those boxed by bold lines). In the SWT component (left), light green (boxed) indicates the SWT inbox whereas all other colors are developers. The display shows that within 6 months, all SWT bugs have been moved out of the inbox and assigned (or resolved). In the Text component (middle), blue (boxed) represents the Text inbox. There we see roughly the same number of bugs in each age group; few are assigned. This could mean that this team works out of the inbox (never making assignments); or it could indicate trouble. Lastly, the UI component (right) shows from the present to 29 months ago, most developers own roughly the same number of bugs. By 35 months, we see a large orange (boxed) assignee, which turns out to be that team's inbox. This could mean that the team changed its process over time. Three years ago they left many bugs in the inbox, but since then bugs appear to be promptly assigned.

A key aspect of the SHO visualization is that the context a user brings will play a significant role in how they interpret what they see. For instance, the development lead of the UI team may interpret the very old bugs in the UI inbox quite differently than an 'outsider' from, say, the SWT team.

## STUDY

We turn now to the study of change management tasks that eight of our informants carried out using the Social Health Overview visualization and Bugzilla. Our primary purpose was to identify strengths and weaknesses of the SHO visualization with users experienced in OS project work.

### Study Method

As mentioned, participants came from Eclipse, Mozilla, and Rational projects. The visualization data came from the Eclipse project. All task data with which participants worked were unfamiliar; however some from the Eclipse project commented on familiar data they saw in passing or in post-study explorations. The choice of tasks was based on our previous [12] and current interviews with open source informants, as well as the goals for the visualization.

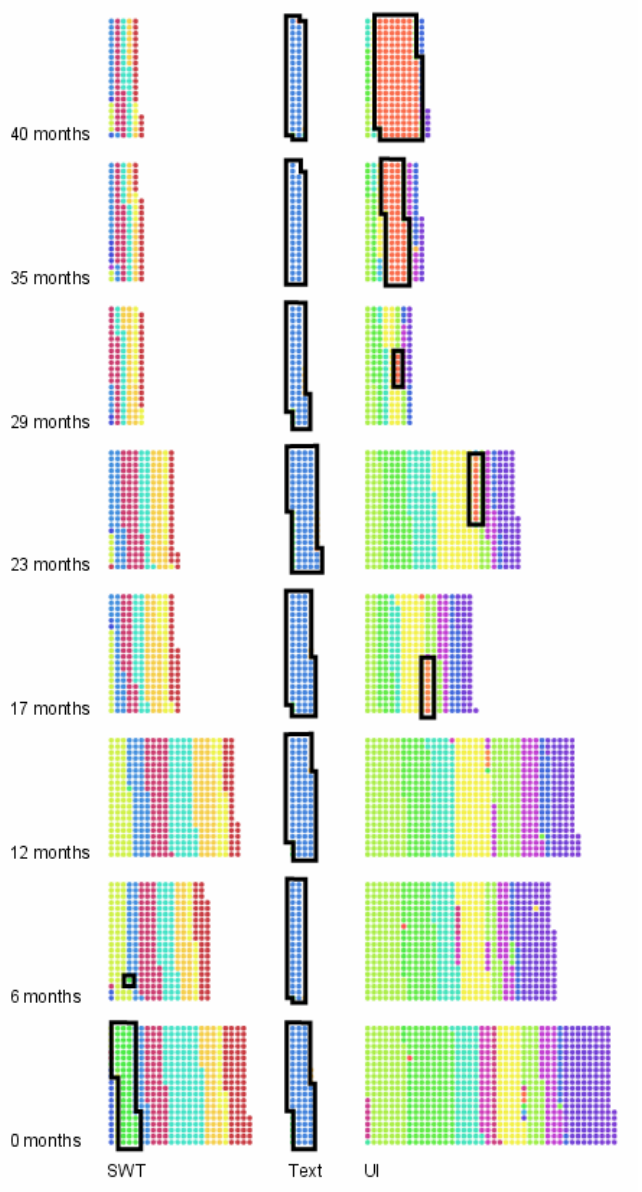


**Figure 5: Bugs colored and ordered by assignee; X-axis set to Component, and Y-axis set to Age. Outlined bugs are those assigned to the inbox of each component.**

We chose tasks that we hoped would be familiar and relevant to driver concerns. One task focused on discerning historical patterns in change tracking data that earlier work suggested were potentially important but hard to detect in systems like Bugzilla [12]. A second task highlighted issues associated with individual team members, such as workload or scope of responsibilities, that might have implications for team management. A third served as a more open-ended and realistic prioritization task that permitted participants to analyze several views or queries of their choosing, and allowed us to observe a rich decision making process. While studying decision-making in the laboratory with unfamiliar data raises well-known issues of ecological validity, we wanted to use the tasks primarily as a focusing strategy for our participants: we were as interested, if not

more interested, in what they took into account in coming to a conclusion as the conclusion itself.

Participants were asked to carry out the same three tasks using Bugzilla and SHO. The first task, "Assign/Reassign," was to find a bug of greater than normal severity with a repeating pattern of assigns and reassigns, a potential indication of a bug ownership problem. The second task "Developer in Trouble," was to find a developer in the Eclipse SWT component whose work the participant thought might be at risk of failure as evidenced by the number and state of the developer's bugs. Participants were encouraged to use whatever definition of trouble made sense from their perspective. The final task, "Next 3 Bugs," directed participants to the Eclipse Compare component and asked them to identify the next three bugs that ought to be worked on.

The study took about an hour to complete and consisted of completing the three tasks using the Bugzilla and SHO systems. The order of systems was counterbalanced across participants. Participants were given a maximum of five minutes to complete each task. Before beginning work with the SHO system, participants completed a tutorial orienting them to the visualization. In addition to explaining terms and how the controls worked (Figure 2), participants worked through three brief examples of use. They were also furnished with a one-page "cheat sheet" that contained the definitions and some tips on useful things to try with the visualization. The tutorial took about twenty minutes to complete. Since all participants were experienced or expert users of Bugzilla, no instruction on Bugzilla was provided. Participants were asked to "think aloud" as they worked through the tasks. They were encouraged to ask clarifying questions as necessary.

Six of eight participants were in geographical locations remote from our lab. They were furnished with a copy of the tutorial, experimental tasks, and the visualization prior to the start of the session. They were asked not to open the documents prior to the session. During the session we used a desktop sharing application to see what the participant was doing, along with an audio conference call. Conversations were digitally recorded and one researcher took notes on all the steps taken in SHO and Bugzilla toward completing each task. Participants' actions with the systems were not automatically recorded.

## RESULTS AND DISCUSSION

Overall, participants were more successful in completing tasks in SHO than in Bugzilla. For Task 1, "Assign-Reassign," no one was able to find such a bug using Bugzilla within the 5 minute time limit, whereas all participants were able to find such a bug using SHO. Several participants pointed out that finding historical patterns such as assign-reassign isn't really supported in Bugzilla. For Task 2, "Developer in Trouble," 6 of 8 participants identified a developer who might be in trouble using Bugzilla and 8 of 8 using SHO. For Task 3, "Next 3 bugs," again 6 of 8 participants specified the next three bugs they thought should be worked on in a particular component using Bugzilla, and 8 of 8 participants were able to do so within the time limit using SHO.

While participants fared better within the time limit using SHO than Bugzilla, this finding should not be over-emphasized, for at least two reasons. First, we already knew that it was difficult for developers to find historical patterns in Bugzilla; making that easier was one goal of SHO. Second, and more importantly, the goal of SHO is to aid developers "in the wild." For that reason, coming up with an answer to the task within 5 minutes is not nearly as interesting or significant as the kind of reasoning in which participants engaged as they searched for an answer.

We did not solicit comments on the relevance of the study tasks. However, four of eight participants volunteered that they found them representative. One said: *"Each of these tasks that you're going through are all tasks that we've struggled with."* But another participant commented to the contrary, saying: *"The first two tasks are not typically things I would do."* We believe this difference of opinion reflects the diversity of approaches to change management in open source projects as well as the different roles and responsibilities participants had in their projects.

### Comparative Analysis of Tasks in SHO and Bugzilla

While each participant went about completing the tasks in different ways, some broad similarities existed in the way participants approached each problem in each tool. Here we provide caricatures of the approaches observed to provide a feeling for how the tools were used differently.

#### *Task Strategies*

Bugzilla Task 1 ("Assign/Reassign"): Participants used the advanced query screen to generate a list of bugs in a particular component. Looking at the results, they then wondered about how they might get historical information about assignments. They would select a particular bug and show its details and look for a way to get historical information. Once they found the historical view, they would look at the history. Here or before, the participant would note that finding a bug with the assign/reassign pattern in Bugzilla would require doing all these steps for each and every bug (until one was found).

SHO Task 1 ("Assign/Reassign"): Participants set either the X or Y axis to severity and turned off all patterns except for assign/reassign. They then looked for the bug with the most heat (biggest circle) and selected that one as their answer.

Bugzilla Task 2 ("Developer in Trouble"): Participants used the Bugzilla advanced query page to list open bugs in the SWT component. They then sorted the result set by severity and assignee. Looking at this tabular list of bugs, they attempted to guess how many of each severity of bug each assignee had by eyeballing or counting the lines. The list of

bugs for one person often covered more than one screen and there were typically numerous assignees to review.

SHO Task 2 ("Developer in Trouble"): Participants decided what patterns were most important to them in terms of indicating "trouble" and unchecked the remaining ones. They then ordered the bugs by assignee and looked at the SWT component for the color (developer) that seemed to appear most in the SWT component.

Bugzilla Task 3 ("Next 3 Bugs"): Participants used the Bugzilla advanced query page to list open bugs in the Compare component that were either high priority, high severity, or both. They then reasoned about which of these to focus on by diving into them and reading the details of particular bug reports.

SHO Task 3 ("Next 3 Bugs"): This task exhibited a great deal of divergence in approach, largely because participants had different ideas about how to decide what should be worked on next. In general, though, participants first checked the patterns they felt would impact their decision. (The pattern(s) chosen varied widely.) They then set one of the axes to severity, the other to component, and set colors to show heat or pattern. They then used a combination of heat, pattern, and severity to reach a conclusion.

#### *Task Strategy Comparison*

A key difference in the way participants worked to complete these tasks is that most of the decision making using Bugzilla was based on the severity of bugs. When using SHO, users took into account severity as well as additional patterns, creating a wider basis for decisions. Of course, it should be noted that additional patterns could be incorporated into Bugzilla.

However, another advantage the visualization offers is the ability to see all bugs represented on-screen at once. In Task 2, for instance, the Bugzilla condition ended with most people scrolling through a tabular list trying to remember the severity of bugs and reason about the number of bugs owned by particular people. The visualization is designed to allow users to see all the bugs at once while retaining the ability to make quantity and quality judgments about them. One participant put it this way: *"I've never been able to see all the CVS bugs in a single view before this."* This kind of overview can aid in pattern recognition and analysis.

It should be noted that Bugzilla had important advantages over the SHO prototype. In particular, its advanced query screen allows significantly more control over the sorts of queries that can be executed against the bug database. In the hands of an expert, Bugzilla can be a powerful tool. In fact, one participant suggested: *"It might be good to have the query functionality from Bugzilla in the visualization so that it can become a day to day triaging tool."*

### Perspective on SHO by Roles

Our participants fell into four major role categories: quality assurance (2 people), quality assurance lead (1), lead developer (4), and project manager (1). In this section, we discuss some of the differences in approach or reaction to SHO that we observed based on role.

#### *Lead Developer*

The lead developers who participated were all working on some aspect of the Eclipse or related projects, so they were quite familiar overall with the data in the visualization and in using Bugzilla to look at those data (though again, none were familiar with the data used for the study tasks). Because of this, after the study tasks had been completed, each of them was interested in looking at the component(s) they owned. The overview of all components that the visualization defaults to was not as interesting to them. (Focusing on a particular component can be accomplished in the visualization by zooming or text filtering but a more direct approach would allow them to select just a component of interest from a list.)

When viewing their own component data, two leads saw potential problems with bugs in the visualization that they were unaware of and made notes to themselves to follow up later with their teams. One issue was a high number of unevaluated patches, and the other a bug that was thought to be resolved but instead was being repeatedly reassigned.

One lead said he would use the tool to look at the data but would not necessarily trust it. He spent some time looking for a bug he knew about to see how the visualization would rate it but could not find it because he could not remember its name. He concluded that he would certainly use the tool to look at heat but it would take some time for him to trust that metric.

When the third development lead reached the end of the tasks, he asked if he could use the tool at the end of each milestone. He said it would be most useful for *"keeping an eye on whether things are piling up."* He also noted that *"it would be good to look at whose heat is high if they had to get a few people to go work on another project temporarily."* He concluded: *"The only thing that would hold me back from using this tool is getting an up-to-date data snapshot."*

The fourth lead did not comment as much during the tasks, though his approach was similar to his cohorts. At the end of his participation, he indicated an interest in the patterns: *"I really like the idea of tracking patterns. Maybe I could come up with some new ones that would be useful, but I'm not sure exactly what they would be."*

#### *Project Manager*

Considering that lead developers only wanted to see the components for which they have responsibility, one might assume that a view of all bugs across a number of components would be most useful to someone at a higher level of management. However, the one such manager who

participated in the study said he received most of his information about the state of the bug world from the lead developers reporting to him. He preferred to use their intimate expertise with the bugs in their components rather than get involved in the details himself. When asked to select the next 3 bugs for the team to work on, he put it this way: *"I literally have a mail in my inbox that tells me what the next three bugs are that this team needs to do."*

This same manager, however, did find patterns in the visualization of which he was unaware. During his work on the tasks, he achieved settings that displayed bug ownership over time (Figure 5). He used this display to reason about the process being used by particular components: *"This picture is quite clear. Comparing across teams, it's clear that some teams find having bugs in the inbox abhorrent while others don't."* One team he is very familiar with drew his attention. After some inspection, he came to the conclusion that the team was following a different process than he had expected; leaving many bugs in the inbox over time. He asked to stop the study so that he could make a detailed note to himself, which he could use to follow up with the team later. He put it this way: *"Interesting. I have no explanation for this. It could represent a change in workflow over time. I find this sufficiently disturbing that I will go talk with the team about this."*

This comment raises the issue of management surveillance enabled by the overview of bug states, and the possible consequences for individuals or teams. Indeed, we heard more than one story from informants of teams that stopped categorizing bugs by priority because management "took it too seriously." Often 'too seriously' meant that the team would receive pointed communications from management about unaddressed high priority bugs, which in reality the team had considered and decided not to fix, but had neglected to reflect in the change tracking system. While making things visible always raises privacy issues, our view is that privacy is not inherently good or bad, but depends on the context. For example, a tool such as SHO may raise awareness of work patterns, but like other workplace technologies (e.g., presence awareness systems), accurate inferences require context. Organizations with poor management or team dynamics might misuse such a tool, but for others it will stimulate early conversation about potential problems, and perhaps more successful outcomes.

### *Quality Assurance*

The two QA participants worked through the tasks similarly to the development leads, however one indicated that the tasks were not representative for them: *"These tasks aren't really things I do. These tasks are for someone who is managing developers. I'm just a triager."* The other seemed to have a bit of a different role, saying his manager sometimes asks him to find out things like "*what the most active component is."* Both provided less feedback than those in other roles suggesting that SHO is less relevant to their responsibilities.

### *Quality Assurance Lead*

Of all the participants, the QA lead spent the most time exploring the visualization and trying new things. When he adjusted the Y-axis to show severity he made an interesting point about the need to be responsive to volunteers in OS development: *"So, you can see unevaluated patches broken down by severity. I love this! This is huge because we have volunteers who attach code and no one looks at it and they think their contribution isn't valued."*

He had similar reactions to many of the other features of the visualization, but left us with a compliment and a caveat: *"If I were a new manager, this tool would be really good to get a starting view of what's going on. If I were a developer, it would give me a good idea of how load-balanced our team was. But knowing if the visualization was actually a legitimate representation of what I was after, I'd have to know how they used Bugzilla better. I believe there's great promise here; and if I were on our Bugzilla with this, I'd be able to do a lot more a lot faster."*

So, clearly he was enthused about the tool but, like some of the development leads, felt the proof really comes from evaluating the tool with data with which you are familiar – seeing what you know about the data reflected accurately.

### *Roles Comparison*

Although one must be cautious about generalizing from a small number of participants, looking across roles, we find that those who would likely benefit the most from the Social Health Overview are those with roles in middle management. These are people who are not quite in the trenches doing all the detail work but are also not in oversight project management roles. This means the best candidates for benefiting from SHO are the quality assurance lead and lead developers. The project manager appears to be at too high a level to benefit from the visualization while those working QA are perhaps focused at too low a level to gain much from a broad overview.

## CONCLUSION

One of the chief values of a visualization tool derives from its flexibility: the degree to which it can be manipulated by users to support their particular information needs, shaped by the project context and individual roles. A visualization that can be manipulated along meaningful dimensions by users will have a greater capacity to be appropriated to support and coordinate work. In this respect, we agree with Froehlich and Dourish's [9] assertion that tools that support "distributed" coordination strategies can be particularly effective. They note: "a distributed strategy attempts to support separate coordination between individuals without requiring a common perspective or shared understanding across the team" and that such a tool "aids coordination not by bringing everyone into alignment with a common perspective, but by providing developers with an enhanced understanding of the work of others and of the group, allowing them to make appropriate decisions about their own activity [9, p.6]. SHO supports distributed

coordination, but in the context of a common underlying data source (i.e., the Bugzilla change tracking system). Thus, rather than "siphoning off" interaction from shared channels, such as the discussions taking place in Bugzilla, the visualization may act as an extension. We suspect there will always be people and roles in distributed software development that will want or need an overview of activity such as SHO provides. As one participant noted: "Most people don't live in Bugzilla like I do, they're writing code! So they don't see the whole flow of activity; I'm one of the few people who does that […] and this tool could make my life a lot easier." Taking that comment to mean 'this *type* of tool,' the current work, along with other research exploring integrated visualizations of development artifacts and activities, suggests that this is indeed a promising approach.

**ACKNOWLEDGEMENTS**

We are grateful to the busy informants who took the time to participate and to the AC and CHI reviewers for excellent comments.

**REREFERENCES**

1. Anvik, J., Hiew, L., and Murphy, G.C. Who should fix this bug? *Proc. ICSE 2006*, ACM Press, (2006), 361-370.
2. Ball, T. and Eick, S.G. Software visualization in the large. *IEEE Computer*, 29, 4, (1996), 33-43.
3. Ball, T., Kim, J.-M., Porter, A.A., and Siy, H.P. If your version control system could talk. *ICSE 1997 Workshop on Process Modeling and Empirical Studies of Software Engineering*, Boston , MA, 1997.
4. Card, S.K., Mackinlay, J.D., and Shneiderman, B. (Eds.) (1999). *Readings in information visualization: Using vision to think.* San Francisco, CA: Morgan Kaufman.
5. Carstensen, P. H., Sorensen, C., and Tuikka, T., Let's talk about bugs! *Scandinavian Journal of Information Systems*, 7,1 (1995), 33-54.
6. Confora, G. and Cerulo, L. Supporting change request assignment in Open Source development. *Proc. SAC 2006*, ACM Press (2006), 1767-1772.
7. Eick, S., Steffen, J., and Summer, E. SeeSoft: A tool for visualizing line-oriented software statistics. *IEEE Transactions on Software Engineering*, 18,11, (1992), 957-968.
8. Ellis, J.B., Danis, C., Halverson, C., and Kellogg, W.A. Social visualization in software development. *Ext. Abstracts, CHI 2006*, ACM Press (2006), 742-747.
9. Froehlich, J. and Dourish, P. Unifying artifacts and activities in a visual tool for distributed software development teams. *Proc. ICSE 2004*, ACM Press, (2004), 387-396.
10. Grinter, R. Using a configuration management tool to coordinate software development. *Proc. COOCS 1995*, ACM Press (1995), 168-176.
11. Gutwin, C., Penner, R., and Schneider, K. Group awareness in distributed software development. *Proc. CSCW 2004*, ACM Press (2004), 72-81.
12. Halverson, C., Ellis, J.B., Danis, C., and Kellogg, W.A. Designing task visualizations to support the coordination of work in software development. *Proc. CSCW 2006*, ACM Press (2006), 39-48.
13. Herbsleb, J.D. and Grinter, R.E. Splitting the organization and integrating the code: Conway's law revisited. *Proc. ICSE 1999*, ACM Press, (1999), 85-95.
14. Hollan, J., Hutchins, E., and Kirsch, D. Distributed cognition: Toward a new foundation for human-computer interaction research. *ACM Transactions on Computer-Human Interaction*, 7,2, June 2000, 174-196.
15. Mockus, A., Fielding, R.T., and Herbsleb, J.D. Two case studies of Open Source software development: Apache and Mozilla. *ACM Transactions on Software Engineering and Methodology*, 11,3, (2002), 309-346.
16. Knudsen, D.B., Barofsky, A. and Satz, L.R. A modification request control system. *Proc. ICSE 1976*, IEEE Computer Society (1976), 187-192.
17. Kraut, R.E. and Streeter, L. Coordination in software development. *Communications of the ACM*, 38,3, (1995), 69-81.
18. Reis, C.R. and de Mattos Forte, R.P. An overview of the software engineering process and tools in the Mozilla project. Proc. The Open Source Software Development Workshop, (2002), 155-175.
19. Robertson, G.C., Card, S.K., and Mackinlay, J.D. Information visualization using 3D interactive animation. *Communications of the ACM*, 36,4 (1993), 57-71.
20. Sandusky, R.J. and Gasser, L. Negotiation and the coordination of information and activity in distributed software problem management. *Proc. Group 2005*, ACM Press, (2005), 187-196.
21. de Souza, C.R.B., Redmiles, D., and Dourish, P. "Breaking the code:" Moving between private and public work in collaborative software development. *Proc. Group 2003*, ACM Press, (2003), 105-114.
22. Tversky, B. Spatial schemas in depictions. In M. Gattis (ed.) *Spatial schemas and abstract thought*. Cambridge: MIT Press, (2001), 79-111.
23. Weber, S. *The success of open source*, MA: Harvard University Press (2004).
24. Zhu, H. and Zhou, M.C. Role-Based Collaboration, Workshop, CSCW 2006, ,Banff, Canada.