UNIVERSITY OF CALIFORNIA,
IRVINE

PorchLight: A Tag-based Approach to Bug Triaging

DISSERTATION

submitted in partial satisfaction of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

in Software Engineering

by

Gerald Terry Bortis

Dissertation Committee:
Professor André van der Hoek, Chair
Professor Cristina V. Lopes
Associate Professor James A. Jones

2016

DEDICATION

To my wife, Rebecca, and daughter, Adelina.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF LISTINGS

ACKNOWLEDGMENTS

First and foremost, thank you my advisor, André van der Hoek, for his mentorship, guidance, and support throughout this long process, and his unfaltering encouragement at the times when it was most needed. Thank you also for your patience over the years.

Thanks to my committee members, Cristina Lopes and James Jones, for their thoughtful questions and comments while selecting and refining my topic for this dissertation.

Thank you to Jon Teichrow for supporting both my Ph.D. and my professional career. My research would not have been possible if not for all that I had learned during my time at Mirth.

Thanks to Wayne Huang for his help with the programming for this dissertation, and thanks to Jacob Brauer, David Schramm, Nick Rupley, and Brent Moen for their valuable feedback.

Thanks to the members of the SDCL, for their friendship, feedback, and for being a sounding board. It was with this group that I had stimulating conversations that would inspire me to expand my research and challenge my assumptions. In particular, thank you to Alex Baker and Nicolas Mangano.

Thanks to Ben Mehling and Dr. Bob Murry for their regular checkins on my progress, and their urging to see this work through completion.

Thanks to my parents and brothers for being a wonderful family, for their support during this process, and the encouragement to continue my education through graduate school in the first place.

Most of all, my deepest gratitude goes towards my wife Rebecca, whose support and encouragement have been unwavering.

CURRICULUM VITAE

Gerald Terry Bortis

EDUCATION

Doctor of Philosophy in Software Engineering **2016**
University of California, Irvine *Irvine, California*

Master of Science in Information and Computer Science **2007**
University of California, Irvine *Irvine, California*

Bachelor of Science in Information and Computer Science **2005**
University of California, Irvine *Irvine, California*

PROFESSIONAL EXPERIENCE

Vice President, Platform **2014–Present**
NextGen Healthcare *Costa Mesa, California*

Chief Informatics Officer **2011–2014**
Mirth *Irvine, California*

Sr. Software Engineer **2007–2011**
Mirth (*formerly WebReach, Inc.*) *Irvine, California*

Software Engineer **2005–2007**
WebReach, Inc. *Newport Beach, California*

Technical Writer **2005**
WebReach, Inc. *Newport Beach, California*

LARC Tutorial Assistant Program Tutor **2004-2005**
University of California, Irvine *Irvine, California*

REFEREED CONFERENCE PUBLICATIONS

PorchLight: a Tag-based Approach to Bug Triaging **May 2013**
International Conference on Software Engineering

TeamBugs: A Collaborative Bug Tracking Tool **May 2012**
International Workshop on Cooperative and Human Aspects of Software Engineering

DesignMinders: A Design Knowledge Collaboration Approach **Nov 2009**
International Workshop on Knowledge Collaboration in Software Development

DesignMinders: Preserving and Sharing Informal Software Design Knowledge **Sep 2009**
Workshop on Knowledge Reuse

Software Pre-Patterns as Architectural Knowledge **May 2008**
International Workshop on Sharing and Reusing Architectural Knowledge

ABSTRACT OF THE DISSERTATION

PorchLight: A Tag-based Approach to Bug Triaging

By

Gerald Terry Bortis

Doctor of Philosophy in Software Engineering

University of California, Irvine, 2016

Professor André van der Hoek, Chair

Bug triaging is an important activity in any software development project. It involves triagers working through the set of unassigned bugs, determining for each of the bugs whether it represents a new issue that should receive attention, and, if so, assigning it to a developer and a milestone. Current bug tracking tools provide only minimal support for bug triaging, and especially break down when users must triage a large number of bug reports, since those reports can only be viewed one-by-one.

This dissertation seeks to further our understanding of bug triaging, particularly the conjecture that allowing triagers to work and organize bugs in sets better matches what they want to do when they triage. Our vehicle for exploring this conjecture is PorchLight, a prototype triaging environment that uses tags, assigned to individual bug reports by queries expressed in a specialized bug tagging language, to organize bug reports into meaningful sets so triagers can more easily explore, organize, and ultimately work with bugs in sets. We describe the current challenges in supporting bug triaging, the design decisions upon which PorchLight rests, and the technical aspects of the implementation.

We conducted two studies using PorchLight. The first was a preliminary user study to test the concept, assess the usability of the tool, and to determine if triagers would be able to employ the basic features. The second was a participant observation study with four participants over a set of six sessions in which we engaged them in discussions about their approach to triaging, and performed triaging using bugs from projects with which they were familiar.

Our study led to several key findings. We found that triaging can be a highly context-dependent activity that varies between individuals and projects. Through our discussions with the participants, we also uncovered latent workflows that were realized through the exploration of bugs in sets. Finally, the ability to directly work with bugs in sets rather than individually seems to have enabled triagers to both think about, and work with, bugs more naturally, particularly when there are a large number of bugs that need to be triaged.

Chapter 1

Introduction

Managing and working with bug reports and feature requests is an important part of any software project. Whether an open source project, a commercially developed product, or even an application developed internally within an organization, any software project that has an active user base will elicit additional changes that need to be tracked.

There are many aspects, both social and technical, to how these bug reports and feature requests (referred to collectively as just *bugs* from here on) are managed. From the perspective of the end user reporting the bug or requesting the feature, the primary need is for a portal to facilitate documenting the details of the bug, submitting it to the appropriate place, and subsequently keeping track of the bug to know if it has been triaged, resolved, or if it requires additional information. From the perspective of a project manager, knowing how many bugs and feature requests have been submitted for a project provides input for planning a release and generally measuring the status of the project as well as the workload of the development team. Finally, from the developer's perspective, a list of bugs can be a to-do list for the day [27], and a bug report can provide crucial information needed to track down and resolve a problem. Connecting each of these perspectives, bug reports often serve as a key communication hub, especially in large open source projects [27].

Each of these perspectives is addressed in some way by bug tracker tools. Indeed, bug trackers are typically positioned as supporting the entire life cycle of a bug report, from developers and end users first reporting a problem encountered or suggesting a possible feature they might like to see, to keeping track of the stage at which the bug is in its resolution cycle, to eventually closing the bug and providing the developer and organization, should they be interested, with a variety of statistics and reports.

While end users are able to report a bug, and, once assigned, developers can obtain the information needed to act on the bug, the process of assigning the bug to the right developer for the right milestone has been largely ignored in the implementation of bug trackers. This process is known as *bug triaging*, which is defined more formally as the process of determining, first, if issues reported through the project's bug tracker describe meaningful new problems or enhancements, and second, if they do, assigning them to the appropriate developers and target release milestones for further handling [24].

Bug triaging has a particularly visible role in open source communities, where it is not uncommon to find projects with hundreds if not thousands of open bug reports [31]. In these projects, volunteers are often assigned the task of being the at front-line of the bug triaging process: go through long lists of newly opened and re-opened bugs to sort, tag, and triage them for the developers downstream. Individuals distributed around the world participate in this process by triaging bugs on their own time and contributing to the development of the project. For example, the Mozilla project maintains a regular schedule for volunteer triagers to assist the project [15]:

> "Bugs can stay unnoticed in Bugzilla for a long period of time - they're either not moved in the right component or are missing vital information to get them in developers' hands. Our aim for the day is to manage incoming bugs in the

> Untriaged component. The Unconfirmed Bugs Triage Day is held on a weekly basis. This event is open to anyone wanting to help, contribute or just hang out with us on the #testday channel."

The GNOME project employs a similar approach to triaging with Bug Days [8]:

> "A Bug day is a day when we get together on IRC, find bugs and clean our bug database Bugzilla so that developers can get more work done by focusing on bugs that matter instead of wasting their time on duplicates, unconfirmable bugs and problems that they've already resolved. What we basically do is called 'triaging', i.e. analysing and processing fault reports in Bugzilla."

This process is as important, however, in commercially developed products, especially since such projects can ill afford to ignore incoming bugs reported by customers. In this setting, triaging is more commonly associated with an actual meeting: developers, project managers, and other stakeholders gathering in a conference room, loading the bug tracker on a large display, and working through the latest bugs and features. Some bugs are worked through quickly when there is agreement between the participants, others involve more discussion about when it should be worked on and who should work on it.

At a smaller scale, applications developed for internal organizational use also receive feedback from the end users it was designed for. A simple tool to assist a specific department or make some business process easier will inevitably receive requests for additional features or bug reports, and the author(s) of the tool will need to triage the bugs accordingly.

Triaging can happen in a group setting, or by individuals working alone. In some organizations, bug triaging occurs during meetings regularly scheduled to prepare for the next phases of development—whether these phases are a coarse-grained series of major releases (release planning [45, 38, 41]) or a fine-grained set of sprints in an Agile project [30]. Triagers working in a distributed setting can also participate in the process, either by sharing a screen during a triaging session or by asynchronously annotating bug reports for others to follow-up with.

Triaging can also be performed by individual triagers or developers in their own workspace. In open source projects, where community participants are often distributed globally, triagers share information but perform the actual activity of working through the list of new bugs on their own. If they require additional information from the reporter, or feedback from another triager, they can annotate or comment on the bug and move on to the next one. Similarly, projects developed internally within an organization can have limited resources, so individual developers will triage bugs and feature requests as part of the regular maintenance of the product.

However, bug triaging also takes place outside of such meetings or structured triaging activities of individuals. Triaging can occur anytime the status of a bug needs to be reassessed. It is especially common for bug reports to be reassigned, because an initial assignment may rely on an inaccurate assessment of the root cause of a bug, a wrong impression of whom the expert is to resolve an issue, or developer overload. As a result, the work must be redistributed among the available developers [32]. In the Eclipse project, for instance, about 44% of reported bugs were reassigned at least once [34]. Bug triaging, then, is an on-going activity [29]. In fact, the *Triage Best Practices* guide for the Chromium open source project recommends to triagers that they "triage at regular times, at least once a week." [20]

It is also important to note that the process of triaging goes beyond just assigning a bug to a developer or milestone. It can be a complex activity that requires knowledge of the development process as well as the norms and rules of the project's community. The *Triaging Bugs* guide for the Chromium project [21] described the following guideline for cleaning up old bugs:

> "If the bug has been waiting for information for over a month, and there has

> already been at least one reminder, add a comment saying that the bug is being closed due to lack of response and that a new bug should be filed with the requested information if it's still reproducible, then close the bug as *WontFix*. Otherwise, add the *Action-FeedbackNeeded* label if it's not already there."

This guideline is an example of how heuristics are applied when triaging a bug. The status of the bug ("waiting for information"), the timing of the report ("for over a month"), the level of activity ("at least one reminder"), and the recommended action ("close the bug") are key pieces of information that a triager uses to work through a large list of bugs, both old and new.

The GNOME Project *Steps of Triaging* guide [10] recommends the following:

> "We should wait at least **6 weeks** before closing a report INCOMPLETE that has unanswered questions or missing stack traces. Note: If the report has many duplicates (say >15), give it even more time. The more duplicates a report has, the longer we should wait, because the problem affects more people."

This suggestion is another example of how various factors are considered when triaging a bug, including the absence of information ("missing stack traces") and the relationship of the bug to other bugs that have been reported ("the more duplicates a report has").

Bug triaging is one of the activities inherently supported by bug trackers, since assigning and reassigning a bug to a certain developer and certain milestone is part of the life cycle of a bug [26]. Unfortunately, evidence is emerging that this support is rather limited. Particularly in the open source community, where triaging bug reports is a highly visible process and a responsibility that is distributed across the community [16, 21], concerns have been openly voiced. As an example, a former contributor to the Mozilla project posted on his blog [31] an explanation for no longer contributing to the project, stating that "Triaging is broken.

Period." He continues by explaining that the bug tracker used for the project, Bugzilla, is ill suited for handling the large influx of bug reports from the community as part of a new release process:

> "Right now, there is no real way to triage except 'Here is a list of 1700 bugs. Start at the top and work your way down.' We need a way to mark bugs that need triage ... But we also need to remember, BMO [Bugzilla@Mozilla] is being used for things it never was created for."

The former contributor calls for a new type of tool to support triaging:

> "This is why I have envisioned a separate BMO product of [Unconfirmed] bugs. That way, we can separate end-user bug submission from the development process, at least in the beginning stage. We can add flags to bugs, that while useless for developers, are incredibly useful for Triagers."

He is not alone in his frustration. Other examples of concerned developers can readily be found, with several academic studies backing up the need for a role-oriented bug tracker interface. For example, Bertram et al. suggest that "role-oriented interfaces that emphasize certain aspects of the tracker's data while abstracting away others may provide a better fit for the multitude of stakeholders" [27]. Just et al. also suggest that, in order to make it easier to find bugs, bug trackers should "provide a powerful, yet simple and easy-to-use feature to search bug reports" [36].

This dissertation focuses on bug triaging, and specifically on improving our understanding of the needs of triagers during this important activity. Our conjecture, based on the literature, industrial experience, and observations such as those made by the open source triager above, is that allowing triagers to work and organize bugs in sets better matches what they want to do when they triage. We are not the first to observe this. Members of the triaging community have used approaches like saving and sharing filters and search results, communicating guidelines and norms through wiki pages, and using tags to annotate bug reports for triaging purposes. While these approaches have had some success, they are ultimately auxiliary ways of addressing the problems that triagers face in working with a large collection of bug reports. Fundamentally, the triager must still work through the list of bugs one-by-one with limited context for making decisions, regardless of their overall process. Additionally, as they make assignments, they are lacking feedback to let them know the impact of their recent and prior actions. Finally, they must still address the problem of what to do with bugs that cannot be triaged and should be set aside or grouped, without losing track of them when working through hundreds or thousands of other bugs.

This work is based on the insight that a key obstacle to effective triaging with current bug trackers is that they force a developer to work with bug reports one-by-one, with significant overhead to move from one bug to a next. Triagers have a need to deal with bug reports in *meaningful groups* (e.g., bugs that do not include a screen shot, bugs that were reassigned three times or more, bugs related to component X and filed in the past two days) [29], which, except for a handful of predefined filters, is simply impossible to achieve with current bug trackers. It is especially difficult to effectively triage when the number of open bugs is large. Our investigation attempts to ameliorate this problem by letting triagers explore, work with, and ultimately assign bugs *in groups.*

We leverage *tags* as the way to organize bug reports into meaningful groups that will assist in working with a large list of bugs. We employ an existing definition of tags as “a freely chosen keyword or term that is associated with or assigned to a piece of information in order to support the process of finding these resources” [50]. We define a *tag set* as a set of bugs that are labeled with a particular tag. Tag sets provide a convenient way to categorize, search, and filter a list of bugs. Tags are especially useful in this context since the meaning of a tag can be defined in an ad hoc manner, which allows for the creation of ad hoc tag sets that can be used to initially sort through a large list of new bugs.

A particularly important aspect of our approach is that it enables triagers to build their own tag sets. It is impossible to predict all of the needs of triagers in terms of in which exact subsets of bugs they may be interested at a given moment in time. To support triagers in creating their own tag sets, we have designed a specialized bug tagging language, BTL, that allows for the creation of new tag sets, as well as the addition of new functions to help triagers specify new rules for organizing bugs.

We implemented our approach by constructing a software tool called PorchLight. Tag sets are explicitly represented in the PorchLight user interface and, by turning them on or off, act as filters on the full collection of bugs. The interface also includes a set of default tag sets that represent subsets of bugs commonly reviewed during triage of the bug database. As a result, developers can quickly group and then examine related bugs in many ways, giving them access to the bug database based on attributes, time ranges, and patterns among bug reports. Together with a number of additional features that provide at-a-glance inspection of the historical actions associated with a bug report, drag-and-drop assignment of bugs, and reflection on the emerging workload of different developers and milestones, PorchLight enables developers much greater control over how they choose to triage a large repository of bugs.

Returning to the main focus of this dissertation—understanding the needs of triagers—PorchLight is the vehicle through which we explore both that question and our conjecture that allowing triagers to work with and organize bugs in sets better matches what they want to do when they triage. By implementing PorchLight and using it in a series of bug triaging sessions with actual triagers, we both learn about the feasibility of working with sets of bugs and what the impact is on triaging work.

We conducted a two-part study. The first was an initial exploratory assessment of how professionals reacted to our novel approach and tool with six professional developers who frequently triage bugs for a major open source healthcare software package. We presented PorchLight, then asked them to work through a limited set of triaging tasks with actual bugs, and concluded with a short interview that focused on comparing their experience with PorchLight to their current practice of using the current bug tracker. Reactions were positive, with all of the participants either agreeing or strongly agreeing that the PorchLight interface provided functionality that improved upon their triaging experience. Naturally, we also uncovered some areas where PorchLight could be enhanced.

Based on the feedback we received in our preliminary assessment, we made several improvements, and then conducted a second broader participant observation study over the course of six sessions with four participants and observed both PorchLight and tag sets in use for triaging bugs.

1.1 Thesis Structure

This thesis is organized into eight chapters. The remaining chapters are structured as follows:

Chapter 2 - Background. This chapter briefly reviews current approaches to bug triaging as well as the literature on bug triaging and the problems that emerge when trying to use existing tools to perform bug triaging at a large scale.

Chapter 3 - Requirements. In order to frame the context of our approach to supporting bug triaging, this chapter takes a broad look at the different ways in which existing bug trackers are used in software development and outlines key requirements for supporting bug triaging.

Chapter 4 - PorchLight. This chapter describes PorchLight, our approach to bug triaging. Particularly, we describe the use of tag sets, combined with advanced filters, and an extensible tagging language.

Chapter 5 - Implementation. This chapter describes the implementation details of the different components that together provide the bug triaging functionality of PorchLight.

Chapter 6 - Analysis and Findings. This chapter describes our two-part multi-method study of PorchLight using professionals, as well as the multi-part analysis of the observations that lead to our findings.

Chapter 7 - Conclusions. This chapter presents our concluding remarks and summarizes the contributions of this dissertation.

Chapter 8 - Future Work. This chapter identifies areas for future work based on the results of our research.

Chapter 2

Background and Motivation

In this chapter, we review the role bug trackers play in the software development process, examine current approaches to bug triaging, and discuss the current state of bug triaging research.

2.1 Bug Trackers

Bug trackers serve an obvious need in software development. They "help software teams manage issue reporting, assignment, tracking, resolution, and archiving via a reliable, shared to-do list that is used by numerous stakeholders throughout the lifecycle of the software." [27]. Commercial bug trackers, like Atlassian JIRA [13] and FogBugz [2], and open source bug trackers like Bugzilla [1] and Trac [3], have emerged as invaluable tools for many projects. While a bug tracker is fundamentally a centralized database for tracking bugs and features, it serves a much broader purpose. For example, one qualitative study of the use of bug tracking systems by small, collocated software development teams revealed that the bug tracker played multiple roles, ranging from a to-do list of tasks for developers, to a communication and coordination hub for the entire team [27].

A particularly important role is that of a knowledge repository. Because over time “the issue tracker builds up a staggering amount of information” [27], it becomes a key source of knowledge about a project for the team. Various stakeholders contribute knowledge to the bug tracker by creating bug reports, commenting on issues, and annotating them with metadata. Bug trackers are often also linked to other sources of information, like the source code repository, to provide additional insights into how a particular bug was resolved. Over time as information is collected in the bug tracker, and as it is associated with other pieces of information about the project, it becomes a record of the rules and norms of the team or community that can provide valuable insights regarding how bugs should be triaged.

The bug tracker also acts as a communication and coordination hub for the project. Because the bug tracker is an easily accessible place to tie together all the various threads of information involved in software development, it is not surprising that a significant body of communication surrounding the data is stored [27]. Triaging discussions about a particular bug, requests for clarification on the bug, and design decisions made to resolve a bug are all frequently stored in the bug tracker. Some bug trackers even support adding comments to a bug report by replying to an email, which makes it easier for developers to quickly respond and archive their communication regarding a bug.

Given this, and other roles, it is no surprise that bug trackers have become popular tools indeed. Web pages for popular projects like Eclipse and Chromium include a prominent link to the bug tracker, and new users are directed to the bug tracker as a way to become familiar with what problems currently exist in a project, and what the future direction of the project is.

2.1.1 Information Captured

Bug trackers capture the data needed to act on a bug report, including:

- a summary and detailed description of the bug or feature request;
- a relative measure of the severity and/or priority of the issue;
- the stakeholders involved (such as the reporter and the assignee);
- the version in which the defect was discovered and the version in which it was or will be resolved; and
- a unique identifier that makes it easy to refer to the report in communications.

Some bug trackers also support the following functionality by default:

- an attachment or screen shot;
- a time estimate for resolving the bug or implementing the feature;
- votes from community members to indicate popularity; and
- dependencies on other issues (i.e., bugs that must be resolved before this one).

While the information that can be represented varies between bug trackers, most bug trackers are configurable and can be modified to capture any type of information related to the bug report. For example, some bug trackers include a field for capturing the component that is affected by the bug (*UI*, *database*, etc.) or the environment in which the bug was detected (operating system, browser, etc.), that can be added depending on the project characteristics.

Even though most bug trackers are able to capture this information, not all bugs are created equal. One study has shown that, by including certain pieces of information with a bug report, reporters can drastically increase the likelihood of the bug being resolved [28]. For example, including the steps to reproduce a bug is by far the most important piece of information on a bug report. However, both runtime stack traces and screen shots have been rated as important as well by developers. An analysis of 150,000 sample bug reports concluded that bug reports that contain stack traces get fixed sooner, bug reports that are easier to read have shorter lifetimes, and including code samples in a bug report increases the chances of it getting fixed [28].

2.1.2 Bug Life Cycle

Besides the data that is captured in the bug report, the life cycle of a bug is an important aspect of the bug tracker. Since a key role of the bug tracker is to "track bugs, features, and inquiries as they progress from their initial creation state to a final closed state" [27], most bug trackers allow users to transition a bug from one state to another, view the current state, and even create custom workflows. Figure 2.1 illustrates the default workflow using the default bug statuses in Bugzilla.

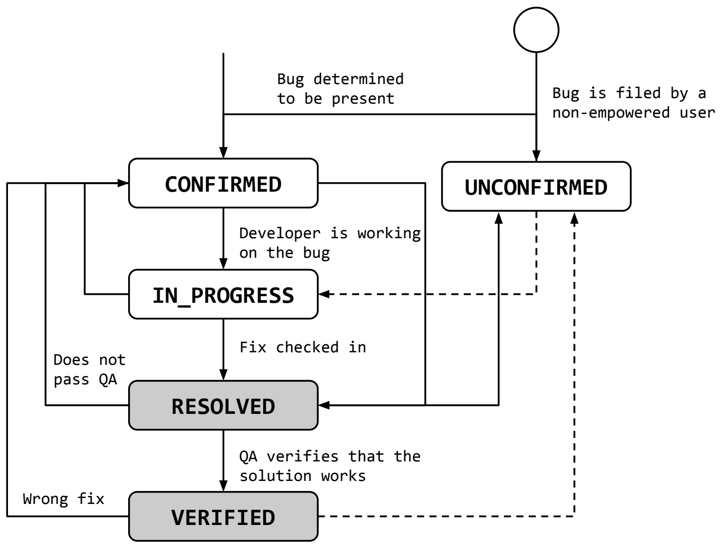

Bug determined to be present
Bug is filed by a non-empowered user
CONFIRMED
UNCONFIRMED
Developer is working on the bug
IN_PROGRESS
Fix checked in
Does not pass QA
RESOLVED
QA verifies that the solution works
Wrong fix
VERIFIED

Figure 2.1: Life cycle of a Bugzilla bug from [19].

The information needs of the developers working on resolving a bug typically differ based on the bug's state in the life cycle. In one analysis of 600 bug reports from the Eclipse and Mozilla open source projects, the authors of the study found that, after a bug has been opened, most questions from the assigned developer were requests for missing information or details for debugging in order to locate the source of the bug [29]. Later questions are mainly discussions of how to correct a bug once a cause has been found, as well as status inquiries to determine if a fix will be included in a certain version of build of the program [29].

In order to accommodate different software development workflows, most bug trackers allow the life cycles to be configured. For example, JIRA includes a graphical workflow designer [22] tool that allows administrators to define the states that bugs can be in, the possible transitions between the states, and the type of information that is collected in a bug report.

2.1.3 Examples of Bug Trackers

Because bug trackers play such an important role in the software development process, numerous tools have emerged, both commercial and open source, that provide a wide array of features. Bug trackers can be divided into two broad categories: stand-alone and add-on.

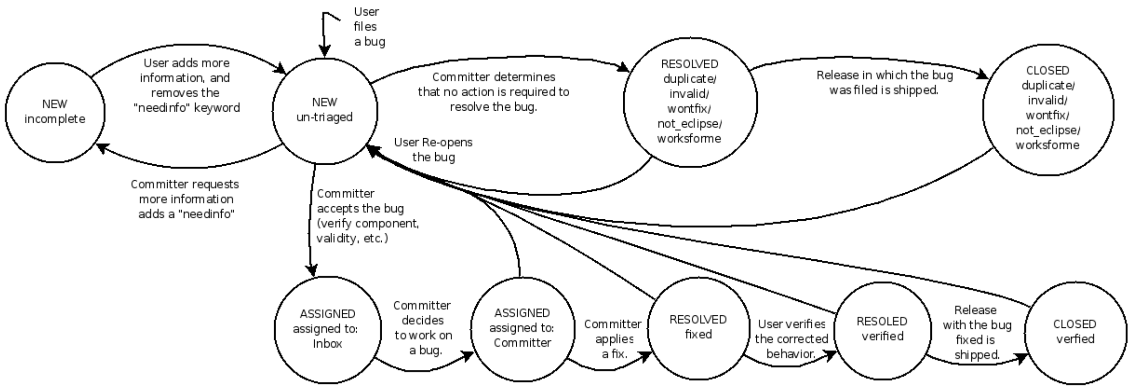

User files a bug
NEW incomplete
User adds more information, and removes the "needinfo" keyword
NEW un-triaged
Committer requests more information adds a "needinfo"
Committer determines that no action is required to resolve the bug.
RESOLVED duplicate/ invalid/ wontfix/ not_eclipse/ worksforme
Release in which the bug was filed is shipped.
CLOSED duplicate/ invalid/ wontfix/ not_eclipse/ worksforme
User Re-opens the bug
Committer accepts the bug (verify component, validity, etc.)
ASSIGNED assigned to: Inbox
Committer decides to work on a bug.
ASSIGNED assigned to: Committer
Committer applies a fix.
RESOLVED fixed
User verifies the corrected behavior.
RESOLED verified
Release with the bug fixed is shipped.
CLOSED verfied

Figure 2.2: Life cycle of an Eclipse bug from [6].

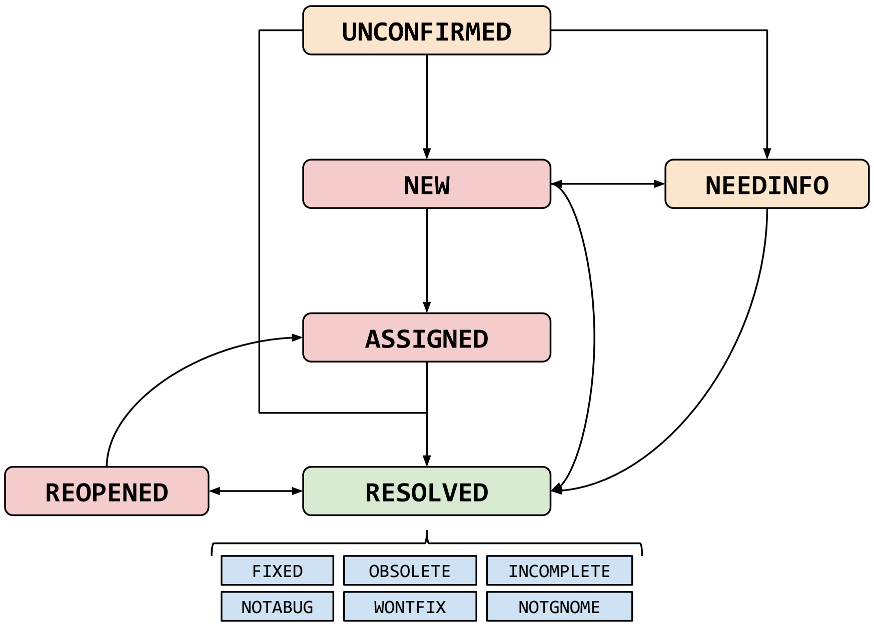


Figure 2.3: Life cycle of a GNOME Project bug.

Stand-alone bug trackers, like Bugzilla [1] or JIRA [13], are tools that are dedicated to bug tracking. While they integrate with other tools, like version control systems, their primary focus is on capturing bug information and managing the bug life cycle (see Figure 2.2 and Figure 2.3 for example bug life cycles). Many stand-alone bug trackers also support custom workflows, so the bug life cycle can be modeled after existing processes.

Add-on bug trackers are those that are built into tools that primarily provide some other functionality. These are most common on software project management or hosted version control services like GitHub [7], Trac [3], or Google Code [11]. While these bug trackers offer basic bug tracking functionality and are tightly integrated with other features such as the project wiki and source code, they often lack advanced functionality offered by stand-alone tools, such as configurable workflows.

2.2 Bug Triaging

Because most bug trackers lack any explicit support for triaging large numbers of bugs, the triaging community has developed several approaches, particularly filters and tags.

2.2.1 Search Filters

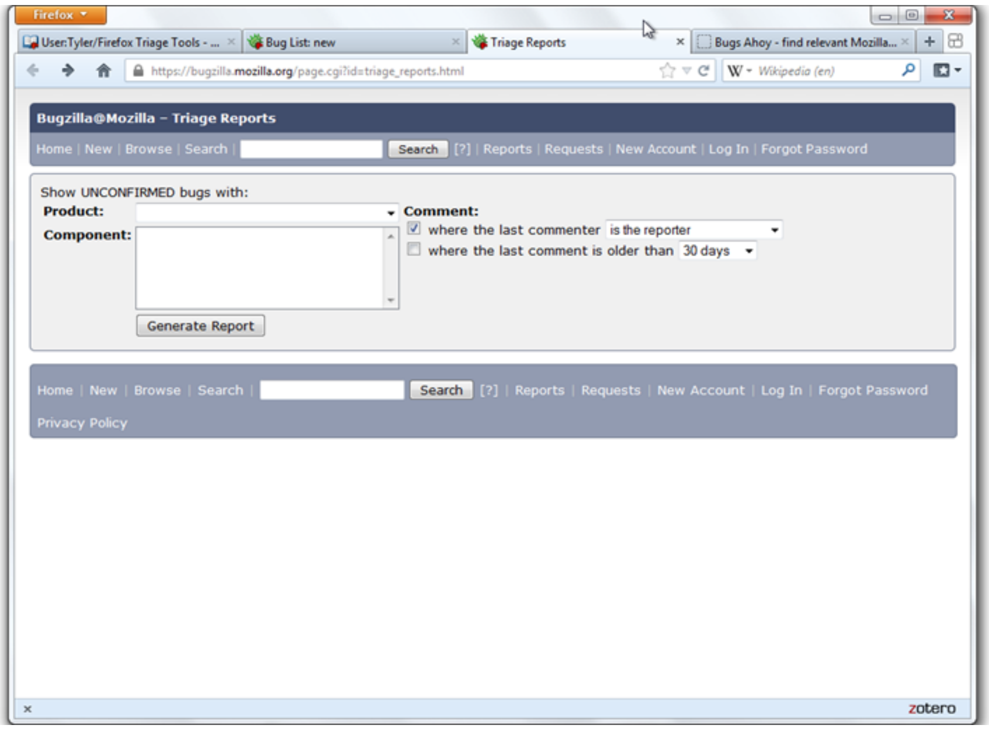
Some large software development projects provide triagers with persistent search filters that can assist in narrowing down the number of bugs that must be triaged. For example, the Mozilla Bugzilla site offers a variety of filters, including a triaging filter (Figure 2.4) that allows triagers to search for bugs from any project with filters for who the last commenter was and when the last comment was added. This allows triagers to quickly create a list of bugs that may require additional information from the reporter, or bugs that have been dormant for too long. Another example of a filter that is available on the Mozilla Bugzilla site is one that identifies bugs that have been filed in the last 24 hours. This provides an easy way for triagers to view recent activity and catch bugs which are relatively easy to triage before additional activity occurs on them and they are lost in the thousands of other bugs.

As another example, the *30 Second Triage Checklist* from the Chromium project wiki [20] recommends that all triagers:

> "Load up your base query (bookmark it): internals:network -feature:spdy,preload -label:spdy"

This query creates a list of bug reports that are related to network functionality but exclude features related to the new SPDY functionality. This bookmarked page is then the starting point for a triager.

The GNOME Project's *Find bugs to triage* guide [9] has a similar recommendation for


Firefox
User:Tyler/Firefox Triage Tools - ...
Bug List: new
Triage Reports
Bugs Ahoy - find relevant Mozilla...
https://bugzilla.mozilla.org/page.cgi?id=triage_reports.html
Wikipedia (en)
Home | New | Browse | Search | Search | [?] | Reports | Requests | New Account | Log In | Forgot Password
Privacy Policy
zotero

Bugzilla@Mozilla – Triage Reports

Show UNCONFIRMED bugs with:

Product:

Component:

Comment:

[✓] where the last commenter is the reporter

[ ] where the last comment is older than 30 days

Generate Report

Figure 2.4: Mozilla bug search filters.

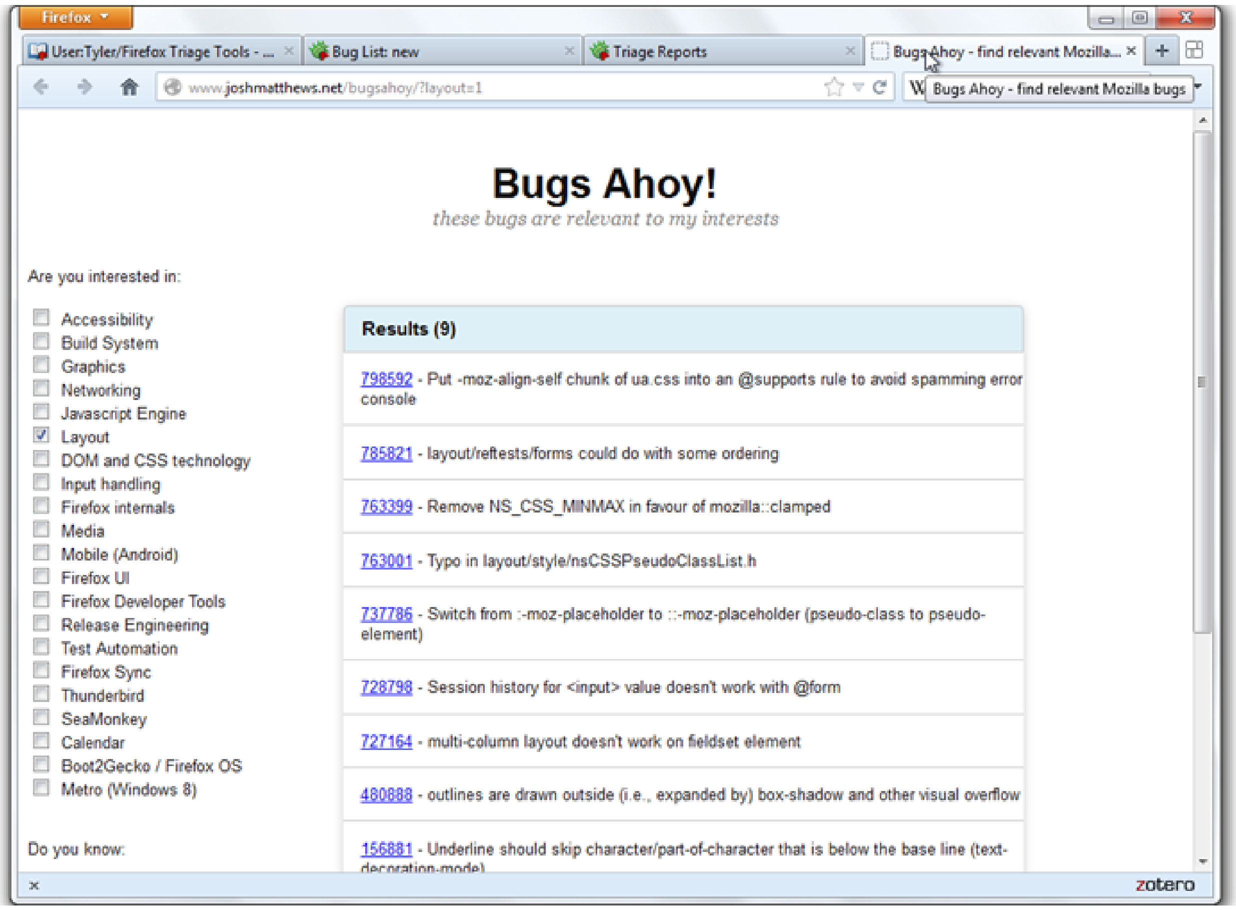
creating and persisting common filters:

> "If you're having a hard time finding bugs to triage in the above lists, you can also use Bugzilla's search/query page. There are some examples of using it below. Note that these searches can be saved for later use, which can come in handy."

Example filters for the project include:

- All the unconfirmed bugs changed in the last four days.
- All the new bugs changed in the last four days.
- Bugs with the words "crash" or "process" in the summary. These bugs tend to be ones that do not have enough information and that need to be marked as NEEDINFO.

Some community members have developed web applications that integrate with the bug tracker to help aspiring triagers identify bugs based on their assigned component. These


Firefox

User:Tyler/Firefox Triage Tools - ... | Bug List: new | Triage Reports | Bugs Ahoy - find relevant Mozilla...

www.joshmatthews.net/bugsahoy/?layout=1

Bugs Ahoy - find relevant Mozilla bugs

Bugs Ahoy!

these bugs are relevant to my interests

Are you interested in:

- ☐ Accessibility
- ☐ Build System
- ☐ Graphics
- ☐ Networking
- ☐ Javascript Engine
- ☑ Layout
- ☐ DOM and CSS technology
- ☐ Input handling
- ☐ Firefox internals
- ☐ Media
- ☐ Mobile (Android)
- ☐ Firefox UI
- ☐ Firefox Developer Tools
- ☐ Release Engineering
- ☐ Test Automation
- ☐ Firefox Sync
- ☐ Thunderbird
- ☐ SeaMonkey
- ☐ Calendar
- ☐ Boot2Gecko / Firefox OS
- ☐ Metro (Windows 8)

Do you know:

Results (9)

798592 - Put -moz-align-self chunk of ua.css into an @supports rule to avoid spamming error console

785821 - layout/reftests/forms could do with some ordering

763399 - Remove NS_CSS_MINMAX in favour of mozilla::clamped

763001 - Typo in layout/style/nsCSSPseudoClassList.h

737786 - Switch from :-moz-placeholder to ::-moz-placeholder (pseudo-class to pseudo-element)

728798 - Session history for <input> value doesn't work with @form

727164 - multi-column layout doesn't work on fieldset element

480888 - outlines are drawn outside (i.e., expanded by) box-shadow and other visual overflow

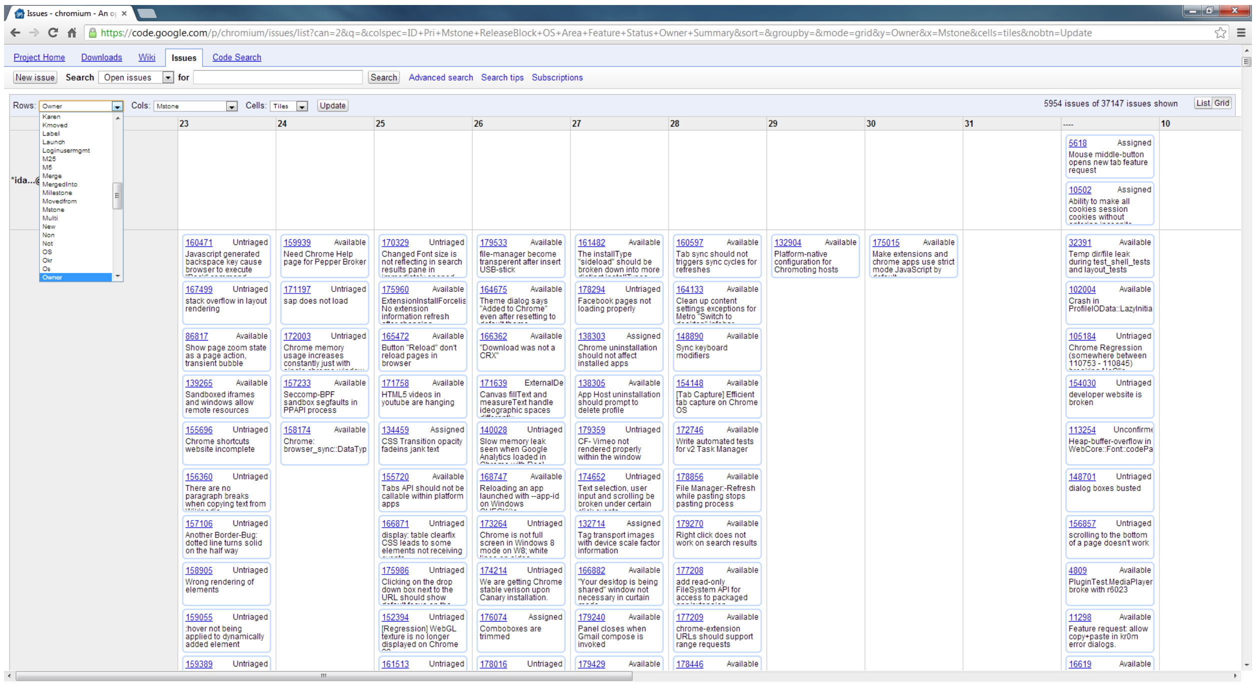
156881 - Underline should skip character/part-of-character that is below the base line (text-decoration-mode)

zotero

Figure 2.5: Bugs Ahoy! user interface.

handful of approaches are in some way a precursor to our work. They serve a dedicated triaging purpose and provide a lens on the vast amount of information that is available in the bug tracker. Rather that starting with a clean slate and having to form complex search queries, triagers (or even developers) can browse the page and discover bugs that are of interest. For example, Bugs Ahoy! [5], shown in Figure 2.5, is an open source project that displays a simple page for new triagers and developers to browse bug reports and feature requests for the Mozilla project. Bug report summaries are displayed in chronological order based on when they were last updated. Additional information, such as the exact date when the last update was and the developer to whom the bug may already be assigned, is available as a tool-tip when hovering over a bug.

Another example of tool created to provide view for triagers is Google Code's issue grid view



5618 Assigned — Mouse middle-button opens new tab feature request
10502 Assigned — Ability to make all cookies session cookies without

23
160471 Untriaged — Javascript generated backspace key cause browser to execute
167499 Untriaged — stack overflow in layout rendering
86817 Available — Show page zoom state as a page action, transient bubble
139265 Available — Sandboxed iframes and windows allow remote resources
155696 Untriaged — Chrome shortcuts website incomplete
156360 Untriaged — There are no paragraph breaks when copying text from
157106 Untriaged — Another Border-Bug: dotted line turns solid on the half way
158905 Untriaged — Wrong rendering of elements
159055 Untriaged — :hover not being applied to dynamically added element
159389 Untriaged

24
159939 Available — Need Chrome Help page for Pepper Broker
171197 Untriaged — sap does not load
172003 Untriaged — Chrome memory usage increases constantly just with
157233 Available — Seccomp-BPF sandbox segfaults in PPAPI process
158174 Available — Chrome: browser_sync::DataTyp

25
170329 Untriaged — Changed Font size is not reflecting in search results pane in
175960 Available — ExtensionInstallForcelis No extension information refresh
165472 Available — Button "Reload" don't reload pages in browser
171758 Available — HTML5 videos in youtube are hanging
134459 Assigned — CSS Transition opacity fadeins jank text
155720 Available — Tabs API should not be callable within platform apps
166871 Untriaged — display: table clearfix CSS leads to some elements not receiving
175986 Untriaged — Clicking on the drop down box next to the URL should show
152394 Untriaged — [Regression] WebGL texture is no longer displayed on Chrome
161513 Untriaged

26
179533 Available — file-manager become transperent after insert USB-stick
164675 Available — Theme dialog says "Added to Chrome" even after resetting to
166362 Available — "Download was not a CRX"
171639 ExternalDe — Canvas fillText and measureText handle ideographic spaces
140028 Untriaged — Slow memory leak seen when Google Analytics loaded in
168747 Available — Reloading an app launched with --app-id on Windows
173264 Untriaged — Chrome is not full screen in Windows 8 mode on W8; white
174214 Untriaged — We are getting Chrome stable verison upon Canary installation.
176074 Assigned — Comboboxes are trimmed
178016 Untriaged

27
161482 Available — The installType "sideload" should be broken down into more
178294 Untriaged — Facebook pages not loading properly
138303 Assigned — Chrome uninstallation should not affect installed apps
138305 Available — App Host uninstallation should prompt to delete profile
179359 Untriaged — CF- Vimeo not rendered properly within the window
174652 Untriaged — Text selection, user input and scrolling be broken under certain
132714 Assigned — Tag transport images with device scale factor information
166882 Available — "Your desktop is being shared" window not necessary in curtain
179240 Available — Panel closes when Gmail compose is invoked
179429 Available

28
160597 Available — Tab sync should not triggers sync cycles for refreshes
164133 Available — Clean up content settings exceptions for Metro "Switch to
148890 Available — Sync keyboard modifiers
154148 Available — [Tab Capture] Efficient tab capture on Chrome OS
172746 Available — Write automated tests for v2 Task Manager
178856 Available — File Manager:-Refresh while pasting stops pasting process
179270 Available — Right click does not work on search results
177208 Available — add read-only FileSystem API for access to packaged
177209 Available — chrome-extension URLs should support range requests
178446 Available

29
132904 Available — Platform-native configuration for Chromoting hosts

30
175015 Available — Make extensions and chrome apps use strict mode JavaScript by

----
32391 Available — Temp dir/file leak during test_shell_tests and layout_tests
102004 Available — Crash in ProfileIOData::LazyInitia
105184 Untriaged — Chrome Regression (somewhere between 110753 - 110845)
154030 Untriaged — developer website is broken
113254 Unconfirme — Heap-buffer-overflow in WebCore::Font::codePa
148701 Untriaged — dialog boxes busted
156857 Untriaged — scrolling to the bottom of a page doesn't work
4809 Available — PluginTest.MediaPlayer broke with r6023
11298 Available — Feature request: allow copy+paste in kr0m error dialogs.
16619 Available

Figure 2.6: Google Code issue grid view.

[11]. Google Code provides project hosting amenities for open source projects, including a bug tracker. One feature of the bug tracker, shown in Figure 2.6, is the ability to create grids with customizable rows and columns. In this example taken from the Chromium project, each row represents an individual developer while each column represent a project milestone. This feature gives a triager the ability to dissect and organize the bug list based on a number of dimensions.

2.2.2 Tags/Labels

Finally, tagging has become a prominent approach to manage triaging. Tagging provides a way for triagers to annotate bug reports without directly modifying the data provided by the reporter. In other words, tagging allows triagers to include metadata that can be used by other triagers and developers, or potentially even the triaging tools themselves. Many bug trackers natively support bug tagging natively, and the use of custom tags has expanded their use in the triaging workflow [16, 12].

Tagging, for instance, can be used to denote the state of a bug or trigger actions for other triagers. For example, the Mozilla triaging guidelines include the following:

> “When preparing to close a bug as INCOMPLETE, it is important to place a CLOSEME tag in the whiteboard of the bug. This way, other triagers know that the bug is waiting for a reply, and bugs with past dates can be closed using simple queries. The typical format of CLOSEME tags is [CLOSEME YYYY-MM-DD]. A CLOSEME should specify a date at least 3 weeks in advance, except in special cases. Once a bug is closed as INCOMPLETE, please keep the CLOSEME tag for tracking purposes.”

In this example, tagging is used to communicate the state of a bug report between triagers. A single tag encodes both the status of the bug, as well as the timing for when the bug should be triaged again if no further information becomes available. It is also important to note that triaging in this case is performed on a bug that has been resolved, and it thus is not an activity that only applies to newly created reports.

One study [50], in looking at why software developers tag work items (bugs) and the roles tags play in practice, found that the primary reason for tagging is for categorizing work items so that they can be more easily found later. For example, in interviews with developers one participant stated:

> “During the polish phase of [a release], we had two weeks to polish. It’s like, what polish things do we do? So we had tags like *polish* and *usability*, and I use that to kind of guide what work items we could work on.”

Tagging is also used to indicate the state of a bug in a workflow that is not necessarily captured by the bug tracker's default life cycle. When used this way, tagging becomes an indirect communication mechanism between triagers and developers to make others aware of particular incoming work items [50].

> "Based on our analysis, we discovered the existence of lifecycle related tags. These tags are used extensively only for a specific period of time as they are related to a milestone in the development process and usually have the name of this release in their name, e.g. *beta2candidate*"

From these examples, we find that the ad hoc nature of tagging makes it an effective practice for triagers and developers to quickly organize work items as they are reviewing them.

2.3 Problems with Current Approaches

While existing ad hoc approaches described in the previous section, such as filters and tagging, have aided in easing the job of the triager when it comes to bug triaging, there are several important shortcomings that still remain.

First, in their current state bug trackers still require a person to drive the process. The triagers, thus, still faces inspecting each bug report, making an assignment for it (either by confirming the suggestion or choosing one on their own), and moving on to the next bug – all through manually navigating the bug repository using the features available in the bug tracker. For most developers, this means using a web-based interface to first search through a vast amount of largely unorganized bug reports for a subset of bugs in which they are interested. They must then select a single bug and, after some inspection and thought, either decide to assign it to a developer and milestone, or skip the bug for now. They then return to the previous search results to select another bug (often in an arbitrary order) and start this process all over again, all without much, if any, feedback during the entire process.

Second, by lacking first-class support for bug triaging, triagers must rely on search filters and bookmarks to track triaging metadata. This approach relies on the filtering capabilities of the bug trackers and is hindered by the navigation functionality provided by the web browser. This makes it difficult for triagers to maintain a context during a triaging session. For example, if a triager views a preset filter for bug reports opened in the last 24 hours and starts from the top of the list, he or she must open a separate tab to review each bug report to either tag or triage it, and then return to the search results list. There is no context maintained with respect to which bugs have already been triaged or what the last action was. Furthermore, many relevant relationships amongst the bugs are not brought forth in the list interface. Bug reports that are related to a particular component, or have been reported by the same reporter, or fall within a certain time frame, are not reflected in the search results list. The list only provides a single dimension by which the large collection of bug reports can be categorized and worked through.

Third, bug trackers provide little, if any, feedback when triaging actions are performed. As triagers work through a list of bug reports provided by a search filter and triage individual bugs, there is no feedback to determine what effect the assignment may have on a particular developer's workload, or on the milestone road map, or even on other bugs that are related. This is especially important for triagers who are new to the project, the development team, and the norms of the community. Feedback in the form of suggestions for a triaging action, or a post-action report, are ways in which a triager can improve their performance.

Finally, bug trackers do not provide a history of bug assignments that can be used to learn from and guide future triaging decisions. For example, the *Triaging Bugs* guide for the Chromium [21] project recommends:

"The more you triage bugs, the more duplicates you will recognize immediately.

> If you suspect something might be a duplicate but aren't sure (such as with a feature request that seems likely to have been made before), search for as many of the variants of the key words in the bug as you can think of. Be sure to search "All issues", not the default of "Open issues" so that you can find *WontFixed* bugs and other duplicates."

The burden lies on the triager to maintain a working memory of the bugs that they have already seen, and what their action was for each particular bug. Bug trackers provide little to no feedback to decide, once a bug has been triaged, if there are other bugs that could be triaged in the same way, or what the effect of the assignment was on the list of bugs.

2.4 Related Work

We are not alone in realizing that bug triaging is an area that can benefit from improved and dedicated tool support. Prior work on the bug triaging process has fallen into three research areas: duplicate detection, assignment automation, and field studies of bug trackers and the triaging process. Work in the first two areas, duplicate detection and assignment automation, aims to reduce the burden on the developers who are involved in the triaging process. Work in the third category, field studies of actual bug triaging processes, aims to improve our understanding of how bug triaging takes place in practice, and informs the design of future bug triaging tools.

2.4.1 Duplicate detection

The goal of duplicate detection is based on the recognition that not every single bug report refers to a unique bug. Given that most software is used by many different users, when a new release introduces a bug, it is not uncommon for that bug to be reported multiple times. On one hand, this benefits the triagers, since they can readily recognize a serious issue because of the multitude of similar bug reports that are likely to be filed. Moreover, some reports may include additional detail (e.g., a stack trace, screen shot) that may help them in triaging. On the other hand, it is still up to the developer to process them all manually, one-by-one. Duplicate detection techniques address this issue by identifying similar bug reports and, in some cases, merging them into a single bug report.

For example, the summaries of the following bugs from the Firefox project were identified as duplicates [51]:

Bug-244372: "Document contains no data" message on continuation page of NY Times article

Bug-219232: random "The Document contains no data." Alerts

Since bug reports are written in natural language, the use of natural language processing over the text descriptions to identify duplicates is a popular technique [22,23]. Natural language processing, or NLP, looks at the frequency of information elements, or fields, in the bug report to determine the similarity of two bugs. Because this approach relies on the vocabulary used in describing the bug, it is applied to finding duplicates that describe the same failure, and not duplicates that describe different failures with the same underlying fault [46]. The processing stages of NLP are:

- **Tokenization:** Turning the characters in the bug description into a stream of tokens, which are typically words (without punctuation, capitals, etc.);
- **Stemming:** Identifying the ground form of each word (ex. *worked* and *working* are transformed into *work*);
- **Stop words removal:** Removing common words that do not carry any specific infor-

mation (ex. *the*, *that*, *when*, etc.);

- **Vector space representation:** Representing the words in a multi-dimensional vector space model to represent each word and its frequency; and
- **Similarity calculation:** Measuring the similarity between any two bug descriptions using the vector space model.

One approach to evaluating a duplicate detection technique is to treat it as an information retrieval system. That is, to determine how effective it is, one compares its results with bugs that have been actually determined to be duplicates by a human. The two measures used in evaluating information retrieval systems are *precision* and *recall* [46]. Precision is the fraction of retrieved bug reports that are relevant to the search, while recall is the fraction of the but reports that are relevant to the search that are successfully retrieved. One study using NLP over a set of bug reports from a mobile phone software developer found that the best results returned by a duplicate bug search were approximately 39% for a top list size of 10 (a top list is the number of possible candidates that are returned from a duplicate search) and 42% for a top list size of 15 [46].

This work led to refinement of the NLP approach by including runtime execution traces. In one study, Wang et al. used the bug repositories for the Firefox and Eclipse open source projects and generated runtime execution traces of error reports and feature requests [51]. Using several heuristics to combine the summary information with the traces, they were able to achieve an increase of up to 26 percentage points in recall rates over approaches using only NLP information.

The most recent work in duplicate detection using NLP by Somasundaram and Murphy incorporates clustering techniques to group duplicates [47]. Clustering is the unsupervised classification of multi-dimensional vectors into groups based on similarity [47]. The authors applied three clustering algorithms (k-means, normalized cut, and size regularized cut) to partition a list of bugs into clusters that describe the same defect. Though their experiments they found that LDA-KL produced recalls similar to those found with existing techniques, but with better consistency across all components for which bugs must be categorized.

2.4.2 Assignment automation

The goal of assignment automation is to lighten the load for a triager by recommending, for a given bug report, who might be appropriate to resolve it [24]. To date, two predominant types of approaches have emerged: machine learning techniques and statistical analysis techniques of bug tossing graphs.

Machine learning

The area of machine learning most applicable to bug triaging is text categorization, which involves the classification of a set of documents (bug reports) into a set of categories (assignments to a developer) [24]. A supervised machine learning algorithm takes as input a set of bugs with known assignments and generates a *classifier* which can then be used to suggest an assignment for an unassigned bug. The process of creating a classifier from a set of bugs is known as *training* the classifier. Features such as keywords and metadata are extracted from past bug reports and, together with data linking these bug reports to the developers who then fixed them, used to train a machine learning model [24]. As with duplicate detection, the performance of a particular ML approach is measured using recall and precision. In automated assignment, precision measures how often the approach makes an appropriate assignment recommendation for a bug. Recall measures how many of the developers who may be appropriate to resolve the bug are actually recommended.

In one study by Anvik et al., the authors applied a machine learning algorithm across three different open source projects: Firefox, Eclipse, and gcc. On the two larger projects, Firefox and Eclipse, the approach achieved precision rates of greater than 50%. However, for the gcc project the precision rate was only 6%. Recall rates across all of the projects were low, often hovering at a few percent, due to the way the authors calculated recall. One explanation for the disparity between the results is the characteristics of the project. Machine learning algorithms generally produce better results if more data is available from which the algorithm can learn. For some projects, like Apache Ant, there was insufficient data based on an examination of the number of reports assigned for the approach to be useful. For other projects, like gcc, one developer seemed to dominate the bug resolution activity, because of which the classification weight of the active developer skewed the automated assignments. These are examples of how project norms and history can come into play when triaging bugs, and are things that can be difficult to capture in fully-automated techniques.

This work led to several other studies that refine the pure machine learning approach. For instance, Somasundaram and Murphy applied Latent Dirichlet Allocation to categorize bug reports into components (e.g., "build", "UI") [47]. One limitation of an SVM (Support Vector Machines) approach is that it may not produce consistent results for each component for which recommendations might be needed if some component does not have as many reports appearing as other components [47]. To overcome this limitation, the authors of one experiment combine an existing machine learning approach with LDA, which is an unsupervised generative model that categorizes the words that appear in the bug reports into clusters. Looking across three different open source projects, the authors found that the combined approach of LDA with the Kullback-Leibler (KL) divergence "can produce recommendations with more consistency in recall values across all components of a system than previous approaches" [47].

Additional studies explore machine learning further. For example, Tamrawi et al. applied fuzzy set-based modeling to automate developer assignments and was able outperform other approaches both in terms of prediction accuracy and time efficiency [49]. For top-5 recommendation tests, the approach has an accuracy of 68% (i.e. in 68% of the cases, the developer that fixed the bug was in its top-5 recommended list), while other machine learning approaches reach the maximum accuracy of 53.02% in top-5 recommendations. Poshyvanyk et al. incorporated a combination of an information retrieval technique and processing of source code authorship information to recommend developers, resulting in "about 20% more accurate than an approach that uses machine learning on past bug reports" in one particular project [43].

Probabilistic analysis

The second type of approach relies on probabilistic analysis of the bug tossing graph. One study has shown that 37–44% of bugs have been tossed, or reassigned, at least once to another developer [34]. A bug tossing graph, like the one shown in Figure 2.7, captures the history of such reassignments from developer to developer, and is used as the source for a Markov chain model that aims to detect repeated patterns (e.g., any time a bug is assigned to developer A that pertains to component X, it is tossed to developer B; half the bugs assigned to developer C are reassigned to developer D) [34].

The authors of one study used this model to represent the bug tossing history of the Mozilla and Eclipse projects. They found that the tossing graphs could be useful in several ways, including identifying the developer structure of a project, reducing the tossing paths by predicting the proper developer, and improving bug triage by recommending developers for assignment using machine learning techniques. In their experiments, the authors were able to reduce tossing steps by up to 72% and improve the accuracy of bug assignment over traditional machine learning approaches by up to 23 percentage points [34].

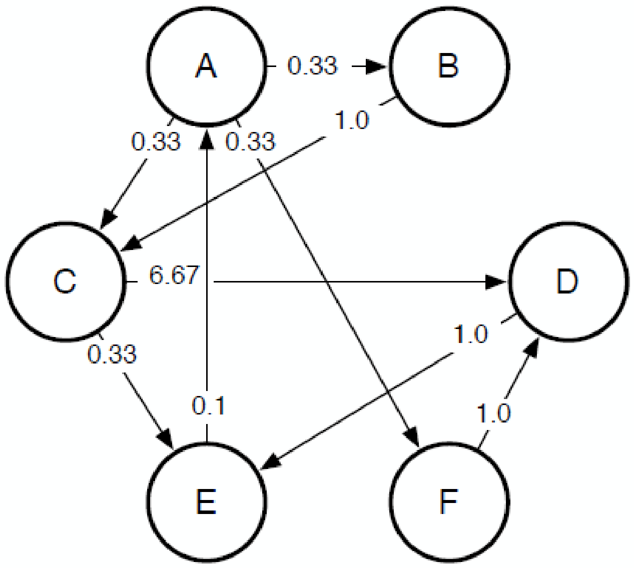


Figure 2.7: An example of a bug tossing graph showing the transition probabilities.

2.4.3 Field studies

Bugs, and particularly bug histories, have been the subject of quite a few field studies (e.g., [26, 27, 29, 33, 40]). The process of bug tracking, and more specifically bug triaging as relevant to our work, has received less attention, with only a handful of studies emerging to date [24, 25].

Most influential on our work is a study by Halverson et al. in which they conducted interviews with industry and open source developers to understand coordination problems that arise when managing bug reports in large, distributed teams [33]. The authors used a combination of interviews and analysis of the bug tracker data for the Mozilla project. They found that detecting patterns in a bug's history that could be used to triage using current bug trackers like Bugzilla can be difficult because "in most change tracking systems it is hard to assemble and see the relevant information," resulting in the assignment problems "going undetected for long periods of time to the detriment of the project" [33].

The authors further identify specific classes of bugs that require attention, such as *zombie* *bugs* (bugs that have been dormant for a relatively long period of time) and *hot potato bugs* (bugs that continue to be tossed from developer to developer without being resolved). Both observations point to a quite different way of exploring a bug repository, one in which bugs are not explored in isolation, but in relationship to each other—whether that relationship is one of functionality or of similarity in the type of bug or bug history.

Another study by Aranda et al. looks at the coordination patterns that emerge surrounding bug tracker user in a large commercial setting [26]. The authors conducted interviews and analyzed the history of bug reports and identified numerous patterns as a result. For example, it was observed that developers will "huddle" with other team members, or "summit" with people from different divisions to discuss a particular bug. This type of coordination complements triaging, which was also observed as team members discussed and decided whether an issue was worth addressing.

Similarly, the social aspects of the bug tracker were found to play an important role in how it is used. Attempts to automate bug assignments can be difficult due to the need for bug histories to "include the social, political, and otherwise tacit information that is also a part of the bread and butter of software development" [27]. This suggests that even for triaging, there are tacit rules and norms that cannot be detected using machine learning techniques.

Another field study of bug tracker usage by Bertram et al., this time in a smaller collocated team setting, reveals that the bug tracker serves as a boundary object and that participants in different roles make use of the bug report information in different, yet interrelated ways and have different perspectives with different needs [27]. For instance, developers and members of the QA team use the bug tracker as an organized list of their work to-be-completed so they could "just come in and sit down and start working" [27]. Meanwhile, one project manager used the bug tracker to maintain a customer-oriented view and as an "outboard brain to keep track of all the things that, from a customer perspective, are important" [27].

In a more recent study by Murphy-Hill et al., the authors conducted qualitative interviews and surveys with engineers working on a variety of products and observed six bug triage meetings to better understand the many factors that influence how bugs are resolved [40]. Their early findings reveal that developers take numerous factors into account when selecting which bugs will be fixed and how they will be fixed, specifically looking at the risk that new bugs would be introduced and the risk that spending significant time fixing one bug comes at the expense of fixing other bugs [40]. This suggests that beyond initial triage, deciding which bugs to fix can be a complicated process that incorporates information that is not only available in the bug tracker, but that resides with the development team and their understanding of the system architecture and the current state of the product.

This aligns with the insight by Halverson et al. that developers use a bug's relationship to other bugs in terms of their functional dependencies, in order to determine how they should be triaged and in which order they should be assigned and resolved. For example, in their study, numerous participants pointed out that "multiple related bugs could be a sign of a larger structural issue with the code, potentially requiring redesign" [33].

Chapter 3

Requirements

As part of our initial research, we explored existing approaches and tools to supporting bug tracking and bug triaging. In the open source community, for instance, we found several ad hoc approaches that have been built around existing bug trackers (as described in Section 2.2.1). We also looked at field studies which suggest that a much broader process exists, involving communication and coordination between different groups, through which bug tracking and triaging actually takes place (as described in Section 2.4.3). We also realized that, while there is much room for improvement in existing techniques, such as duplicate detection and assignment automation, any such improvement would still be within the confines of the bug tracker. As we have seen, however, working within the boundaries of the bug tracker has some fundamental limitations. Specifically, as further expanded in Section 2.3, triagers must still work through a list of bugs, one-by-one, with the limited aid of some filters and possibly tags, to triage.

Inspired by our findings, and our own experiences having witnessed and participated in triaging meetings in both a commercial and open source development setting, we decided to step back and look at triaging as a whole—what is it that triagers need to achieve, and how can they achieve it effectively?

To address these two questions, we developed a dedicated tool based on the concept of sets of bugs. To guide the development of the tool, we identified four elementary requirements for triaging; that is, basic needs that triagers have and that must be reflected in any triaging tool. Each of the following requirements first states the basic need for triaging, and then works out the implications for sets of bugs.

Initially these requirements were somewhat vague, based on discussions on blog posts, wiki articles from the triaging community, the studies that we have previously mentioned, and speaking informally with developers from a professional software development organization. As we explored our initial requirements, we began to construct several prototypes, and through these reflected more on the problem. The following requirements are the results of this iterative process, and represent the four most important requirements that we set out to meet in the research.

3.1 Requirement 1: Explore

The first requirement for a triaging tool is to provide triagers with the ability to browse new and unassigned bugs and to easily organize them into categories for further triaging.

The number of bug reports that should be triaged in a typical large project can be overwhelming. For example, in the spring of 2010, the Mozilla Firefox project had approximately 13,000 unconfirmed bugs that were unassigned to either a developer or a milestone [31], with a multitude of new bugs being filed each day. To make the process of triaging these bug reports manageable, a tool should provide triagers with ways to explore and organize the bugs; that is, they should be able to quickly get to a subset of the bugs that they find interesting, and change this subset at any moment in time when they want to take a look at a different slice of the overall set of bugs.

Situations arise in which triagers cannot immediately triage a bug until more information is obtained, or other bugs are triaged first [29]. Part of exploring a large collection of new bugs is being able to set select ones aside to come back to at a later time. To support this, a bug triaging tool should support ad hoc grouping of bugs, i.e., the manual creation of sets of bugs that are seemingly unrelated but of importance for the triager or developer to keep track of. This allows triagers to set aside bugs that may require follow-up from the reporter, need additional information from another developer, or can only be triaged after the workload across the developers is determined from triaging other bugs. Supporting ad hoc grouping is especially important in open source projects where many unconfirmed bug reports require follow-up from the reporter with more details, stack traces, or screen shots [29]. It should also be possible to persist sets of bugs across triagers, so they can be used as a starting point for a future triaging session. New triagers, too, may want to use sets as a way of putting bugs aside for review by more experienced developers, while still contributing to the process in assigning those bug reports that they can clearly handle.

Ad hoc groups could also be used for tentative planning. Rather than making definite assignments bug-by-bug, a triager could use an ad hoc group to create a planned set of bug assignments for a given developer or milestone that they do not want to quite commit just yet. If one of the developers is already overloaded, for instance, it may still be desirable to assign him or her some bugs that he or she is most qualified to address. Doing so through an ad hoc group allows the triager to safeguard against assigning too many new bugs to the developer. Particularly, if at the end of the triaging session only a few bugs exist in the ad hoc group, they can then be assigned as a whole, but if there clearly are too many, some can be reassigned first to other developers without having to look through all of the bugs that were already assigned before the triage session began.

3.2 Requirement 2: Search

The second requirement is to provide triagers with the ability to search for bugs using different criteria, as well as to group them based on these criteria.

While we could focus on creating a rich portfolio of search filters as the goal of our research, if one examines the comments from the triaging community and the types of searches that bug triagers actually want to express, it is clear that we need to go beyond simple search filters that can be applied in the bug tracker.

As stated under requirement 1 (Explore), triaging requires the ability to easily organize large sets of bugs, and then to be able to flexibly compose and work with these subsets. Triagers also need to be able to do this in a way that can be shared with other triagers or developers so that the rules and norms of the community can be developed and maintained. This suggests a need for not only a way to work with sets of bugs, but also a new approach to how these sets are defined. Specifically, we need an approach that allows triagers to compose sets of bugs that take into account the different fields, statuses, and heuristics that triagers apply to make triaging decisions based on the varying workflows and rules used by each community. In other words, triagers must be able to find bugs and organize them based on the criteria that are useful to them.

While predefined search filters are useful, they necessarily limit the ability of the developer since they cannot easily compose, modify, or share them with other developers. Consider the following examples of types of sets that a triager might want to explore while triaging:

- Bugs which have been reported in the last 24 hours.
- Bugs which have gone for 30 days with no comment from the reporter.
- Bugs which have been reassigned multiple times to different developers.

- Bugs which include a screen shot or stack trace as part of their description.
- Bugs which have been waiting for more information for over 30 days and already include one reminder from a triager.

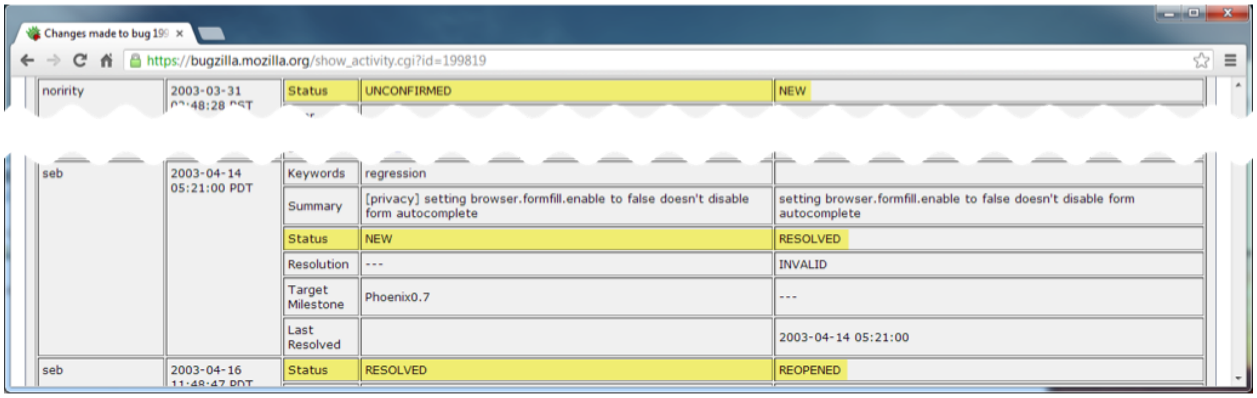
These, and other sets like it, should be easily obtainable by triagers. In fact, if they so wish, these sets should not just be viewable, but also assignable in their entirety to a particular developer or milestone.

3.3 Requirement 3: Inspect

The third requirement is to provide triagers with the ability to inspect individual bugs by reviewing comments, histories, and related bugs, to make informed triaging decisions.

Once a triager has identified the sets of bugs that need to be triaged, they must review each individual bug to take some action. The information that a triager should have available varies depending on the state of the bug that is being inspected. If a bug is early in the life cycle and has recently been reported, a triager will need to be able to view basic information about the bug, like the summary, the details, and the reporter. This information should be immediately available to the triager upon selecting a bug, especially since triagers may not have much time to dedicate to each individual bug. If the initial review of the bug information does not provide enough detail to make a decision, the triager should be able to easily inspect additional details and related data elements. For example, screenshots, attached files, or stack traces within the bug report can help inform the action that should be taken.

Later in the bug life cycle after it has been assigned however, a triager may require a different set of details to properly inspect the bug. For example, a triager should be able



noriity	2003-03-31 02:48:28 PST	Status	UNCONFIRMED	NEW
seb	2003-04-14 05:21:00 PDT	Keywords	regression	
		Summary	[privacy] setting browser.formfill.enable to false doesn't disable form autocomplete	setting browser.formfill.enable to false doesn't disable form autocomplete
		Status	NEW	RESOLVED
		Resolution	---	INVALID
		Target Milestone	Phoenix0.7	---
		Last Resolved		2003-04-14 05:21:00
seb	2003-04-16 11:48:47 PDT	Status	RESOLVED	REOPENED

Figure 3.1: An example Bugzilla history for a bug report showing the status changing from UNCONFIRMED to NEW to RESOLVED and to REOPENED.

to easily identify problematic patterns in the bugs history, such as frequent reassignments or inactivity, to take the appropriate action [26, 33]. For example, a bug that goes from high to no activity could indicate a “zombie bug” that may be at risk of falling through the cracks, and it is important for triagers to be aware of and potentially reassign the issue. For instance, Figure 3.1 shows the the history of a bug that has changed status several times.

Being aware of these status changes and their causes may well influence the actions of the triager, who, instead of simply assigning the bug, may add a comment requesting more information from the reporter, or contact the developer who was originally assigned the bug to learn what expertise is needed to resolve the bug.

Whether or not source code has been committed to the bug, comments from developers and the community, and any status changes or reassignments are examples of the kind of information that should be at hand when triggers are making assignment decisions. In the ideal case, the information should be summarized for at-a-glance examination and interpretation, but at the same time any details should be readily obtainable when they are needed to make an informed decision.

3.4 Requirement 4: Take Action

Finally, the fourth requirement is to allows triagers to easily take action by assigning a bug, adding a comment, assessing any feedback, and moving on to the next one.

It goes without saying that the process of taking an action on a particular bug, which usually means assigning a bug to a milestone or developer, must be lightweight and require minimal effort. When working with hundreds or even thousands of new bug reports, triagers must be able to efficiently work through the set of bugs and take action. Triaging actions generally fall into one of four categories, abbreviated as A.C.T.S.:

- **Assign**: Triagers assign a new bug to a developer or milestone, or re-assign a bug that has already been assigned.
- **Comment**: Triagers often comment on new bugs to either request additional information from the reporter, or to add context that can be used to assign the bug at a later time.
- **Tag**: Triagers leverage tags to add metadata to the bug report to both categorize bugs and also indicate state within a workflow.
- **Status**: Triagers move bugs between various states in the workflow as part of the triaging process.

Triagers should be able to easily perform these actions without disrupting the triaging process or altering the context created by the initial bug sets. This is not the case in existing bug trackers, since viewing a bug's details and performing an action usually require leaving the bug list and losing the search context. For example, the process of triaging a list of unverified bugs using Bugzilla is the following:

1. Load the Mozilla Bugzilla page at https://bugzilla.mozilla.org/

2. Click Search link
3. Select Advanced Search
4. Select a status of NEW (along with any additional filter criteria) and click Search
5. Sort the search results by a column, such as the bug's severity
6. Select an individual bug from the list and review its details
7. Select a developer from the Assigned To list, or set the Target Milestone, or both
8. Click the browser's Back button to return to the search results
9. Repeat step 6

It is crucial to avoid such context switching between searching, inspecting, and finally taking action.

Additionally, in existing bug trackers, after a decision has been made to assign a set of bugs to appropriate developers and milestones, all of that information moves to the background. The bug disappears from the list, but there is no tangible feedback for the triager. The triager has no easily accessible cumulative record of their assignment decisions, other than their personal memory. This is a problem, as triaging decisions are not just based on which developer is suitable to work on a bug, but also on the desire to appropriately manage the emerging workload of each developer and evenly distribute new feature requests across releases [26]. In commercial settings, it is particularly important for anyone triaging issues to not only appropriately label and assign bugs, but also to maintain a balance of bugs between developers and milestones, as there are limitations in the resources that are assigned. This information, available upon assignment, can provide valuable feedback during the few minutes allotted to each issue, and assure that one particular developer or milestone is not overloaded or underloaded. Once an assignment has been made, the effects of that action on the workload should be visible to provide awareness of the impact on the project.

A triager should also be able to assess their progress in handling the current set of bug reports. Particularly, they should be provided with up-to-date counts of how many bug reports are left to triage in each set that they are working with. For example, a triager may find that a large number of bug reports cannot be assigned since they require additional information from the reporter. Over time, the number of bugs in this group should decrease as the reporters comment and the bugs are assigned. Triagers should have a sense of whether bug counts in certain groups are increasing or decreasing in order to measure the effectiveness of the triaging process.

3.5 Summary

These four requirements suggest that, just as developers have an integrated development environment (IDE) for working with source code, triagers should have a dedicated workspace that integrates with the information stored in the bug tracker, but provides a view of the bug list that supports triaging. The same interface must support exploration and searching for bugs based on criteria, visualization of important bug details, and quick action with feedback. However, any approach should integrate with existing bug tracking systems. Migrating a large number of bug reports and mapping them to a new bug tracking system is a significant undertaking, especially in large and established projects. Since existing bug trackers work fairly well in some regards, it would be prudent to maintain the existing information and provide an interface on top of the bug tracker to facilitate triaging. Any new information that is generated as part of the triaging process should be kept in sync with the source bug tracking system. We summarize the requirements for a dedicated triaging tool in Table 4.3.

Explore	• Support organizing large collections into bug sets • Allow the creation of ad hoc bug sets • Allow the composition of bug sets
Search	• Allow triagers to search based on criteria • Support advanced criteria such as time windows and actions • Support the creation of tag sets from searches
Inspect	• Display information that is relevant to triaging at all stages of the bug life cycle • Allow triager to view additional details • Provide an at-a-glance view of a bug's event history
Take Action	• Support primary actions: assign, comment, tag, and change status • Support these actions on both individual bugs and on sets • Allow actions to be taken without unnecessary switching context • Provide reflective feedback after actions are taken

Table 3.1: Summary of requirements for a dedicated triaging workspace.

Chapter 4

PorchLight

Because bug trackers are ubiquitous and have established roles during software development, our approach has been to augment and integrate with existing bug trackers rather than try to replace them. We decided to design and implement a new user interface that *sits on top of* the bug tracker and is compatible with the data that has already been collected, but also introduces the key features that are needed to support and enhance the triaging process.

The main question in designing this new interface is: what metaphor would best satisfy our requirements for representing sets of bugs and provide an improved experience for triagers? Through numerous mock-ups and iterative designs, we explored a range of alternative metaphors, including a stacking metaphor (each bug report is represented by a note card that can be sorted into stacks to represent desired groupings, see Figure 4.1), a network metaphor (each bug is a node in a network that the triager could navigate and update by using and changing relationships between nodes to create clusters), and a grid metaphor (each bug report is represented by a tile in a grid that the triager can reshuffle to make assignments).

Ultimately, we chose to adopt tagging as a paradigm for working with bugs. Tagging has

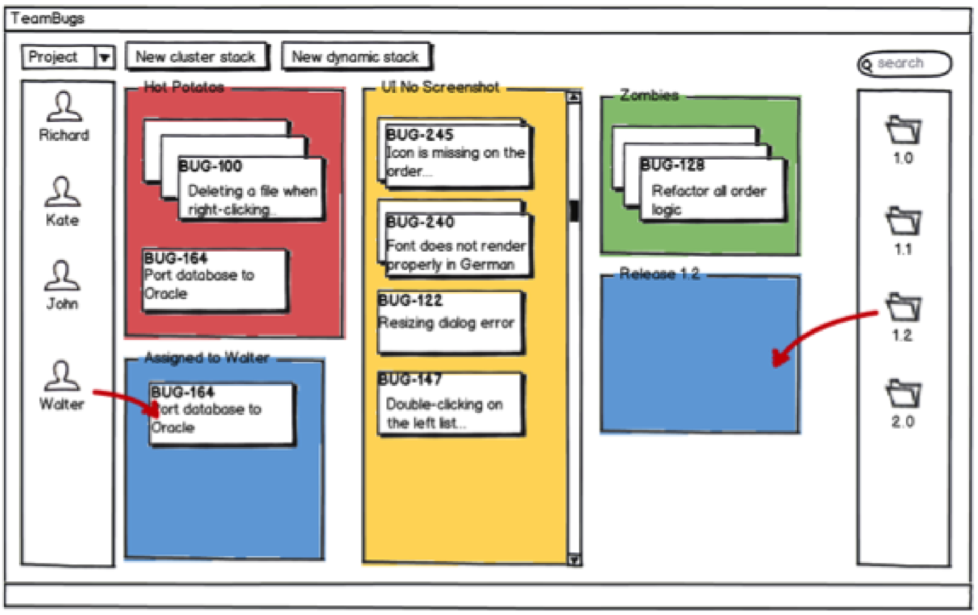


Figure 4.1: Early mock-up based on the card stacking metaphor.

emerged as a powerful technique that is applicable to many different situations (e.g., managing e-mail [37], collaborating through work items [50], annotating online media [23]). Many software projects have adopted tagging as a way to both organize and annotate bugs to indicate progress or state. Some communities even provide a collection of tags that should be used, and it is recommend that triagers organize bugs first by tagging them before triaging (The Chromium Projects [21] being the prime example). Tagging is also an easy way for a community to establish norms [50] and communicate them to new contributors who may wish to help with the triaging process. Finally, nearly every bug tracker already supports tagging of bugs in some way. For example, both JIRA and Bugzilla provide a field when reporting a bug for entering free-form tags that can subsequently be used to search for bugs or to create search filters.

Despite their potential, tags are not used consistently throughout most bug tracker interfaces as a means of organizing bugs or triaging. For example, both JIRA and Bugzilla provide search interfaces that emphasize searching based on descriptions, assigned versions, or other attributes which are often set *after* bugs have been triaged. JIRA additionally provides an

Agile view for organizing new feature requests into sprints, which emphasizes a different set of attributes, such as assigned story points and sprint version. While these fields can be used for triaging, they do not provide the ability to broadly categorize large numbers of bugs, as tagging can. As a result, tags are displayed as optional columns in tabular views, or as a field that can only be viewed after selecting an individual bug.

Our goal is to move tags to the forefront of the user interface as the primary means of categorizing large collections of bugs and making triaging decisions. That is not to say that tags are the only attributes that are important in a bug tracker. Rather, we view it as the most effective tool in tackling the problem of too many bugs during triaging. Tags provide triagers with an efficient way of taking a "first pass" at bugs and sorting them for further work. By developing a tool that is tailored for this process, we aim to improve how triagers can do their work.

The result of our design efforts is a novel bug triaging tool, PorchLight, that leverages tagging as the mechanism for organizing and working with bugs. In this chapter, we demonstrate our approach through PorchLight's basic features, and we highlight its novel features which unlock the power of tagging.

4.1 Design Decisions

To ensure that we adequately address the requirements for a triaging tool that we described in Chapter 3, we first made several key design decisions.

First, we wanted to make PorchLight a dedicated triaging tool. While there are numerous roles that use bug trackers during the software development process, and whose work we could improve with a dedicated tool, our goal was to focus on supporting the specific needs of a triager. We wanted to develop a tool that would support triaging in both in open source and commercial settings where triagers must work through large collections of bugs and make triaging decisions.

Second, we wanted PorchLight to integrate with existing bug trackers. Development teams and communities have invested much time (and in some cases, money) in their existing bug tracking tools. To encourage adoption, we wanted to build a new environment to supplement the existing bug trackers and that would use the same underlying bug report data. The rest of the team could continue using the bug tracker tool as is, while triagers could make use of this new environment.

Third, we wanted tag sets to be the primary mechanism in PorchLight by which triagers interact with large collections of bugs. Rather than just re-implementing the search filters and tabular view that triagers already use, we wanted to anchor the workspace in tag sets. We also wanted to use tagging as a way to create these tag sets since it is an approach that triagers are already familiar with.

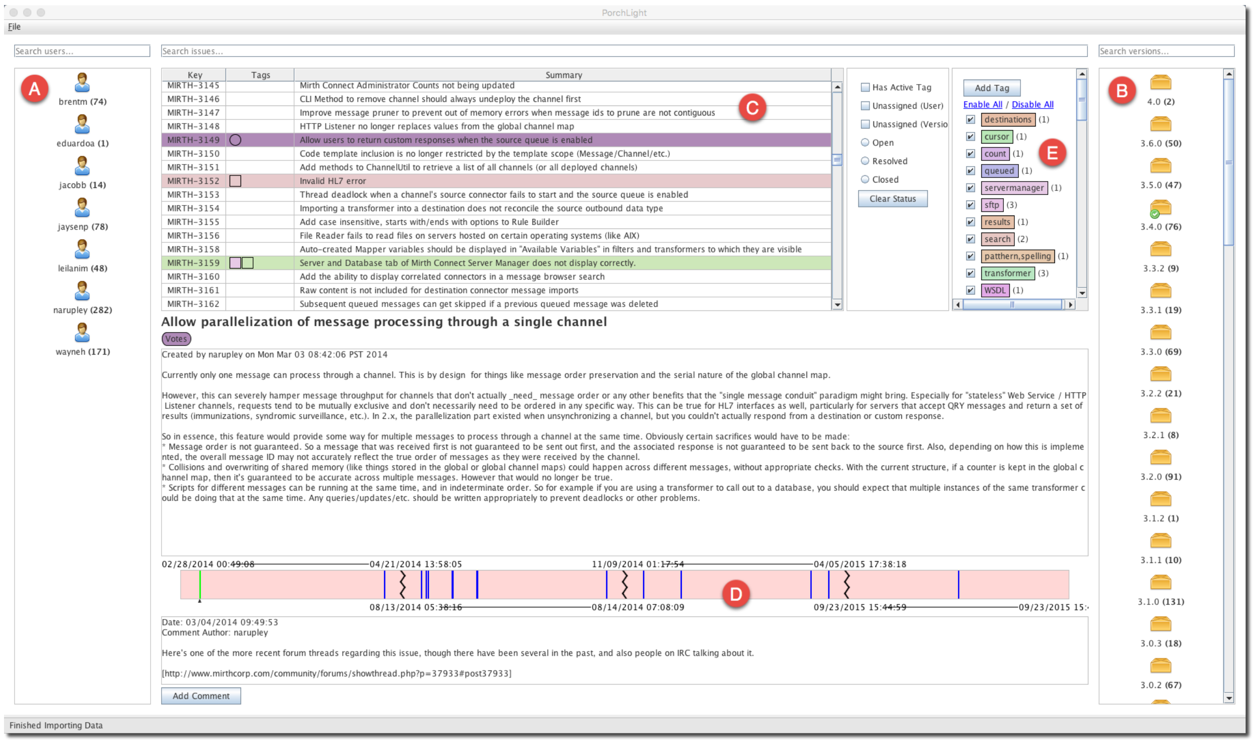
Finally, we wanted to avoid excessive context switching in PorchLight as much as possible. Because we have designed a new view on top of the bug tracker and were not bound by existing views and workflows, we had the opportunity to address one the major sources of frustration in working with large collections of bugs to triage. We wanted to make the process of exploring, searching, inspecting, and taking action possible through a single perspective that did not require the triager to navigate back and forth between different levels of detail.

4.2 Overview

The PorchLight user interface is shown in Figure 4.2. Because PorchLight is implemented as a new interface to existing bug trackers, users can connect it to a source bug tracker by using the project import feature (Figure 4.3) which imports the project and all of the bugs from the bug tracker.

Once connected to a source bug tracker, any changes made in PorchLight, such as updating bug meta data or assigning a bug, are recorded until the user wishes to commit them. The ability to commit changes allows the triager to review changes made across numerous bugs before they become permanent in the bug tracker (Figure 4.4). Changes that have been committed to the source bug tracker are available for other users to view and modify. If there are any conflicts during the commit process (that is, the status or other details of a bug have been changed in the bug tracker), the conflicting bugs will be displayed and the user will be given the option to either preserve the bug state as it is in the bug tracker or continue by applying the changes.

If a source bug tracker hosts multiple projects, the currently active project is displayed in



Key	Tags	Summary
MIRTH-3145		Mirth Connect Administrator Counts not being updated
MIRTH-3146		CLI Method to remove channel should always undeploy the channel first
MIRTH-3147		Improve message pruner to prevent out of memory errors when message ids to prune are not contiguous
MIRTH-3148		HTTP Listener no longer replaces values from the global channel map
MIRTH-3149	○	Allow users to return custom responses when the source queue is enabled
MIRTH-3150		Code template inclusion is no longer restricted by the template scope (Message/Channel/etc.)
MIRTH-3151		Add methods to ChannelUtil to retrieve a list of all channels (or all deployed channels)
MIRTH-3152	□	Invalid HL7 error
MIRTH-3153		Thread deadlock when a channel's source connector fails to start and the source queue is enabled
MIRTH-3154		Importing a transformer into a destination does not reconcile the source outbound data type
MIRTH-3155		Add case insensitive, starts with/ends with options to Rule Builder
MIRTH-3156		File Reader fails to read files on servers hosted on certain operating systems (like AIX)
MIRTH-3158		Auto-created Mapper variables should be displayed in "Available Variables" in filters and transformers to which they are visible
MIRTH-3159	□□	Server and Database tab of Mirth Connect Server Manager does not display correctly.
MIRTH-3160		Add the ability to display correlated connectors in a message browser search
MIRTH-3161		Raw content is not included for destination connector message imports
MIRTH-3162		Subsequent queued messages can get skipped if a previous queued message was deleted

Allow parallelization of message processing through a single channel

Created by narupley on Mon Mar 03 08:42:06 PST 2014

Currently only one message can process through a channel. This is by design for things like message order preservation and the serial nature of the global channel map.

However, this can severely hamper message throughput for channels that don't actually _need_ message order or any other benefits that the "single message conduit" paradigm might bring. Especially for "stateless" Web Service / HTTP Listener channels, requests tend to be mutually exclusive and don't necessarily need to be ordered in any specific way. This can be true for HL7 interfaces as well, particularly for servers that accept QRY messages and return a set of results (immunizations, syndromic surveillance, etc.). In 2.x, the parallelization part existed when unsynchronizing a channel, but you couldn't actually respond from a destination or custom response.

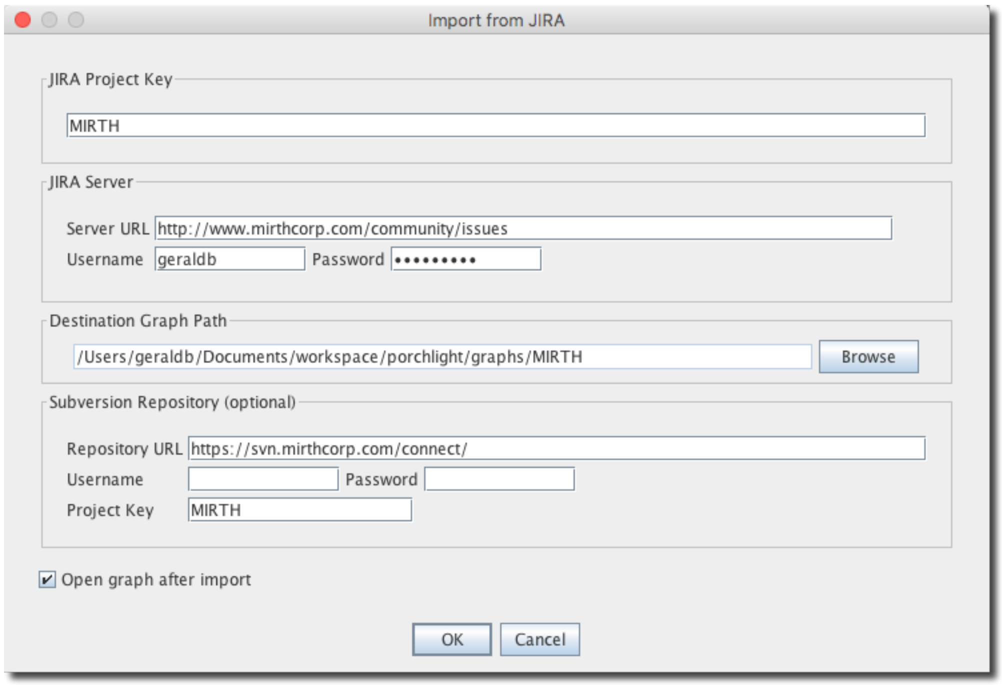
So in essence, this feature would provide some way for multiple messages to process through a channel at the same time. Obviously certain sacrifices would have to be made:
* Message order is not guaranteed. So a message that was received first is not guaranteed to be sent out first, and the associated response is not guaranteed to be sent back to the source first. Also, depending on how this is impleme nted, the overall message ID may not accurately reflect the true order of messages as they were received by the channel.
* Collisions and overwriting of shared memory (like things stored in the global or global channel maps) could happen across different messages, without appropriate checks. With the current structure, if a counter is kept in the global c hannel map, then it's guaranteed to be accurate across multiple messages. However that would no longer be true.
* Scripts for different messages can be running at the same time, and in indeterminate order. So for example if you are using a transformer to call out to a database, you should expect that multiple instances of the same transformer c ould be doing that at the same time. Any queries/updates/etc. should be written appropriately to prevent deadlocks or other problems.

Date: 03/04/2014 09:49:53
Comment Author: narupley

Here's one of the more recent forum threads regarding this issue, though there have been several in the past, and also people on IRC talking about it.

[http://www.mirthcorp.com/community/forums/showthread.php?p=37933#post37933]

Figure 4.2: The user interface of the PorchLight system, implemented as a Java desktop application. Clockwise from the top: (a) user list, (b) milestone list, (c) bug details, (d) timeline, (e) tag set list.


Import from JIRA

JIRA Project Key

MIRTH

JIRA Server

Server URL http://www.mirthcorp.com/community/issues

Username geraldb Password •••••••••

Destination Graph Path

/Users/geraldb/Documents/workspace/porchlight/graphs/MIRTH

Subversion Repository (optional)

Repository URL https://svn.mirthcorp.com/connect/

Username Password

Project Key MIRTH

[✓] Open graph after import

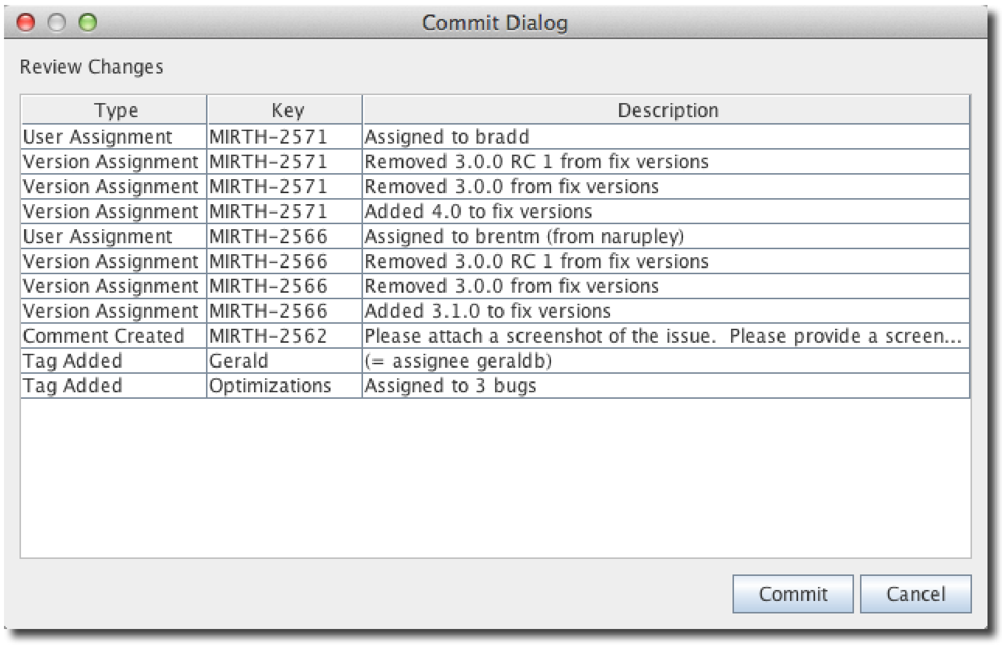
Figure 4.3: Importing project bugs from a source bug tracker.

the upper-right corner of the interface. Users can change the active project using the project selector dialog (Figure 4.5).

Selecting a project results in multiple lists being populated: a user list on the left of the interface, a milestone list on the right, and the list of bug reports in the center. The goal of the interface is to include all of the information that the triager needs to inspect a bug report and assign it to both a user and a milestone. Doing so in a single perspective prevents users from having to continuously switch between different views to see both bug details and bug lists.

4.2.1 User List

The user list (labeled "a" in Figure 4.2) displays all of the user accounts that are available in the bug tracker. Each user is visualized as an icon with a label containing the user's login name and current assigned bug count. In a typical open source project with many



Type	Key	Description
User Assignment	MIRTH-2571	Assigned to bradd
Version Assignment	MIRTH-2571	Removed 3.0.0 RC 1 from fix versions
Version Assignment	MIRTH-2571	Removed 3.0.0 from fix versions
Version Assignment	MIRTH-2571	Added 4.0 to fix versions
User Assignment	MIRTH-2566	Assigned to brentm (from narupley)
Version Assignment	MIRTH-2566	Removed 3.0.0 RC 1 from fix versions
Version Assignment	MIRTH-2566	Removed 3.0.0 from fix versions
Version Assignment	MIRTH-2566	Added 3.1.0 to fix versions
Comment Created	MIRTH-2562	Please attach a screenshot of the issue. Please provide a screen...
Tag Added	Gerald	(= assignee geraldb)
Tag Added	Optimizations	Assigned to 3 bugs

Figure 4.4: Committing changes made to to the bugs.

bug reporters and commenters, this list can be very long. While we have decided to include all users in this list to make it possible to assign to any participant in the community, we have taken several steps to make it more usable than just a large alphabetically sorted list of users. Because there are different types of users (i.e., not all users registered with the bug tracker are developers), we partition the user list into the following groups:

- Users that have been assigned bugs
- Users that have only commented on a bug
- Users that have only reported a bug

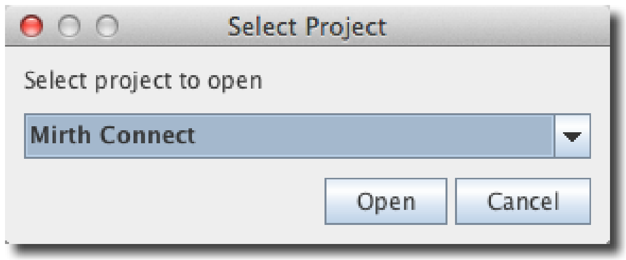


Figure 4.5: Selecting a project.

- All other users

Within these groups, the users are sorted by the total number of assigned bugs. If the number is equal, the group is sorted alphabetically based on the username.

In a project that has been active for some time or has quite a few bugs, the first group represents the makeup of the core contributing developers. These are the users to which a bug is most likely to be assigned for resolution. The second group represents users that have participated in the bug reporting process by commenting on a bug, but are not currently active in development. This group is likely to be triagers who request additional information from reporters through comments on bugs. The third group represents users who have contributed by creating a bug report, but not followed up with additional information. Finally, the remaining users—those who have created an account but have no activity—are displayed in the last group.

Even with the grouping and sorting method described above, the list of users can become quite long. To make it easier to quickly find a specific user in the list, we have included a search field that will automatically scroll the list to the user that most closely matches the search term entered (see Figure 4.6).

4.2.2 Milestone List

The milestone list ("b") displays all of the milestones, or release versions, for the project defined in the bug tracker. Each milestone is visualized as an icon with a label containing the milestone name (typically, but not always, a version number) and a count of the number of bugs that are currently assigned to the milestone. The milestone list is sorted using the Semantic Versioning, or SemVer, specifications [18]. This handles version numbers that follow the convention of X.Y.Z, where X denotes the major version, Y denotes the minor

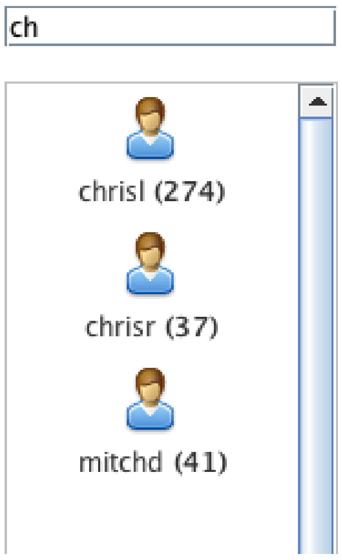


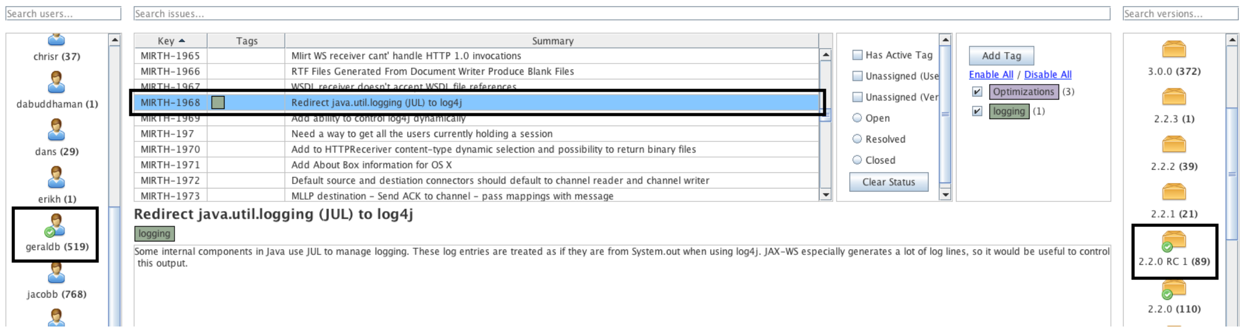
Figure 4.6: User list showing search results.

version, and Z denotes the patch version. This scheme also takes into the account pre-release qualifiers, such as *alpha* or *beta*.

The milestone list includes both released and unreleased milestones, as it is not an infrequent occurrence that a release contains bugs that were not resolved yet. In such cases, the triager still has to have access to the bug for further handling and rescheduling it to an upcoming release, and know where it was originally to be resolved. Planned releases are shown first in the list to prevent constant searching.

4.2.3 Bug List

The bug list ("c") is the primary focal point for triagers in PorchLight. Each row in the list displays the bug's unique identifier, colored squares representing the tags assigned to the bug, and the summary. The list is populated with all of the bug reports in the project, initially presented chronologically with newer bugs at the top and older bugs at the bottom. We have chosen these columns for the list because they convey the minimum amount of information that is needed to quickly identify a bug.



Key	Tags	Summary
MIRTH-1965		Mlirt WS receiver cant' handle HTTP 1.0 invocations
MIRTH-1966		RTF Files Generated From Document Writer Produce Blank Files
MIRTH-1967		WSDL receiver doesn't accept WSDL file references
MIRTH-1968	🟩	Redirect java.util.logging (JUL) to log4j
MIRTH-1969		Add ability to control log4j dynamically
MIRTH-197		Need a way to get all the users currently holding a session
MIRTH-1970		Add to HTTPReceriver content-type dynamic selection and possibility to return binary files
MIRTH-1971		Add About Box information for OS X
MIRTH-1972		Default source and destiation connectors should default to channel reader and channel writer
MIRTH-1973		MLLP destination - Send ACK to channel - pass mappings with message

Redirect java.util.logging (JUL) to log4j

logging

Some internal components in Java use JUL to manage logging. These log entries are treated as if they are from System.out when using log4j. JAX-WS especially generates a lot of log lines, so it would be useful to control this output.

Figure 4.7: Bug list with selected bug and associated user and milestone.

The triager navigates this list by moving up and down (either with the scroll bar or using arrow keys). If a selected bug has already been assigned, the developer and/or milestone to which it has been assigned are highlighted in the respective list. For instance, in Figure 4.7, MIRTH-1968 is selected and shown to be assigned to “geraldb” and milestone “2.2.0 RC1”. This makes it possible for a triager to quickly determine if a bug has already been assigned, and to which developer or milestone it has been assigned.

4.2.4 Timeline

Selecting a bug from the bug list displays additional information about the bug. This includes the summary or title of the bug, a more detailed description, and the timeline (Figure 4.8).

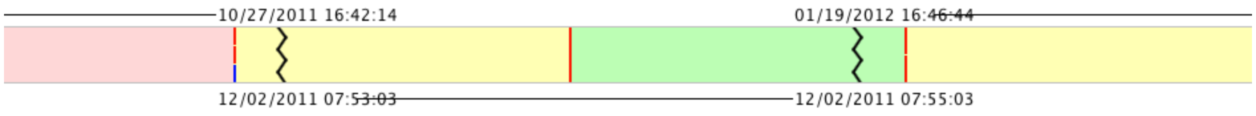


Figure 4.8: Timeline showing actions in the bug’s history.

The timeline displays all of the activity that has transpired relating to the bug report (Figure 4.9). Actions are plotted chronologically on the timeline oldest to newest from left to right. The starting date for the timeline is the date the bug was created, and the ending date is the date of the last action taken. Actions that are related to the bug appear as colored vertical markers ((1) in Figure 4.9). A cursor ((2) in Figure 4.9) under the timeline indicates the selected event, and can be used to traverse the history of the bug using the left and right arrow keys. Each action type is assigned a different color to make it easier to distinguish in the timeline. Table 4.1 shows the actions represented in the timeline with their associated color.

Action	**Color**
Comment added	Blue
Attachment added	Blue
Assignment (or reassignment) to user	Red
Assignment to milestone	Red
Status change to Closed	Red
Vote added	Red
Tag/label added	Purple
Source code committed	Orange

Table 4.1: Actions displayed on the activity timeline and their associated color.

It is not uncommon for multiple actions to occur in succession (for example, commenting on an bug and then resolving it). To make it easier to identify individual actions that occur within a small timeframe, multiple actions are represented as stacked markers, one per action ((3) in Figure 4.9). It is also common for there to be periods of inactivity between actions, which can cause areas of high activity to be concentrated in one area of the timeline. We have added several features to improve the visibility of individual actions and to make it easier to make sense of the history.

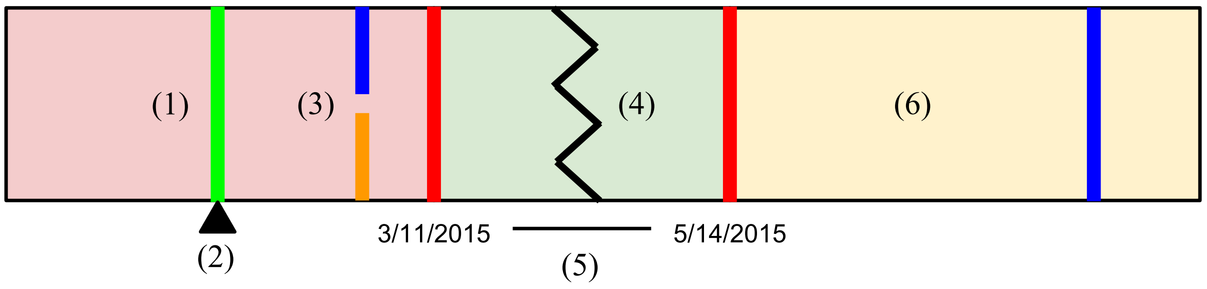
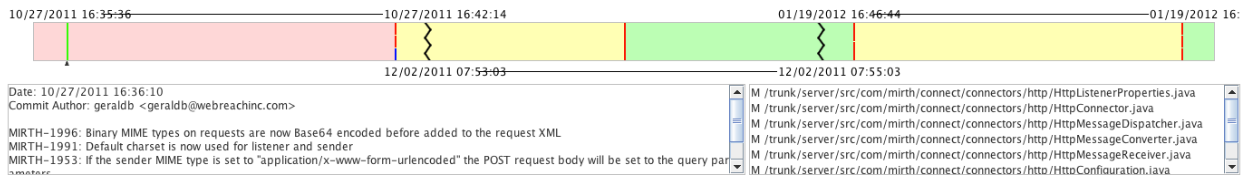


Figure 4.9: Different elements of the the timeline component.

Particularly, the timeline is further broken down into periods of activity. Long periods of



```
Date: 10/27/2011 16:36:10
Commit Author: geraldb <geraldb@webreachinc.com>

MIRTH-1996: Binary MIME types on requests are now Base64 encoded before added to the request XML
MIRTH-1991: Default charset is now used for listener and sender
MIRTH-1953: If the sender MIME type is set to "application/x-www-form-urlencoded" the POST request body will be set to the query par
```

```
M /trunk/server/src/com/mirth/connect/connectors/http/HttpListenerProperties.java
M /trunk/server/src/com/mirth/connect/connectors/http/HttpConnector.java
M /trunk/server/src/com/mirth/connect/connectors/http/HttpMessageDispatcher.java
M /trunk/server/src/com/mirth/connect/connectors/http/HttpMessageConverter.java
M /trunk/server/src/com/mirth/connect/connectors/http/HttpMessageReceiver.java
M /trunk/server/src/com/mirth/connect/connectors/http/HttpConfiguration.java
```

Figure 4.10: Timeline with event details.

inactivity are removed and the interruptions are noted by the zigzag ((4) in Figure 4.9). This allocates more space for the actions to be displayed, and decreases the amount of overlap between markers during those periods of activity. The timeline is broken down in up to four activity periods, with each of the three largest gaps between actions having to be longer than one week in order for the four periods to be shown. The start and end dates of each period of inactivity are indicated above or below the timeline, with a solid line connecting the two dates to indicate the time span ((5) in Figure 4.9). We chose this design so that the timeline can convey activity on a bug over any period of time.

We also fill the timeline background between markers with colors to indicate activity between those two actions ((6) in Figure 4.9). For example, the background between Open and Resolved markers is shaded red, the background between Resolved and Closed is shaded yellow, and the background between Closed and Reopened is shaded Green. We chose this design to make it easier to identify patterns in the bug's activity at-a-glance without having to view the details of each action.

Selecting a marker from the timeline changes, either by clicking on it or by iterating through the actions using the arrow keys, the marker's color to green and reveals additional details about the action under the timeline (see Figure 4.10). For example, if the action is a user commenting on the bug, the contents of the comment are displayed, along with the author and the date it was added. If source code was committed to the bug, the commit log message and a list of the source code files committed are displayed, along with the types of changes made to each of the files (added, modified, or deleted).

A timeline visualization was chosen to display each bug's history because it allows triagers to assess the level of activity related to a bug in an at-a-glance manner while also providing the ability to drill down and view additional details. Because triagers can iterate quickly through the bug list, it is important that the detailed action information be immediately available without requiring them to navigate to a separate view. A timeline also affords the ability to detect patterns in the activity for a bug. For example, a timeline with primarily assignment and reassignment actions, and few if any source code commits or comments, likely indicates that the bug is a "hot potato" that is being reassigned among developers. A timeline with a flurry of activity just before the current date can, on the other hand, indicate a bug that has become "hot" and is likely to be resolved, possibly leading up to a release.

As an additional example of how the timeline can be used for triaging, suppose that a triager selects a specific bug and views the timeline showing the bug's history. By traversing through each of the events, they notice that the bug had been closed once, but then re-opened with no other activity. This could indicate that there was some deficiency with the original fix and that the developer required more information to resolve the bug. This type of pattern might be a candidate for a tag set of bugs that require follow-up from the triager.

4.2.5 Quick Filters

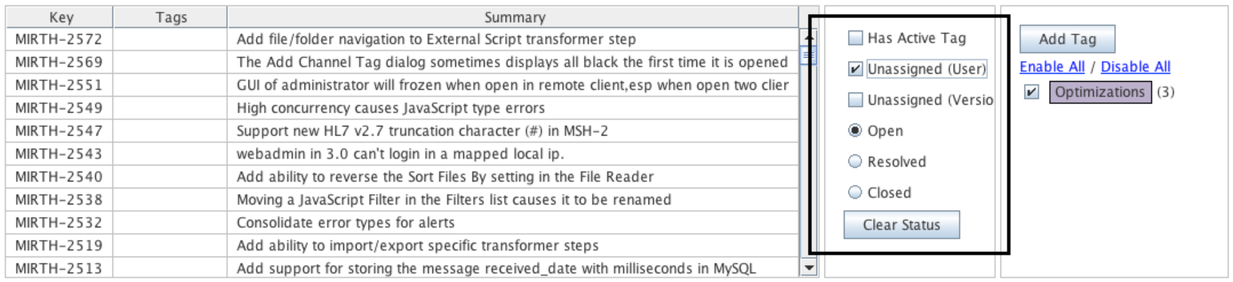
Before triaging, it is likely that a triager will need to immediately cull the list of all bugs to a subset that need to be reviewed. To make this easier, we have included a set of filters that filter the bug list based on a few key statuses. Triagers can enable a Quick Filter (see Figure 4.11) from the top of the interface at any time. The following quick filters are available:

- Unassigned (to a user)

- Unassigned (to a milestone)
- Open
- Resolved
- Closed
- Has Active Tag

Note that multiple Quick Filters can be active at the same time. For instance, selecting both *Unassigned* and *Open* will only display bugs that are open but are not yet assigned. In order to avoid conflicting filters being applied, the status options (Open, Resolved, and Closed) are radio buttons that only allow one to be selected at any time.

The *Has Active Tag* quick filter only displays bugs that have been assigned to one or more tag sets that are selected in the tag set list. This filter, in combination with the tag set functionality which is described later, allows triagers to easily categorize a large bug list. Triagers can use these quick filters to set up their environment before taking on the task of triaging new bugs. For example, a triager will likely enable the *Open* and *Unassigned* (both to user and milestone) filters to obtain a list of bugs that need to be assigned. From here, they can drill down into the details of each bug and perform an assignment action.



Key	Tags	Summary
MIRTH-2572		Add file/folder navigation to External Script transformer step
MIRTH-2569		The Add Channel Tag dialog sometimes displays all black the first time it is opened
MIRTH-2551		GUI of administrator will frozen when open in remote client,esp when open two clier
MIRTH-2549		High concurrency causes JavaScript type errors
MIRTH-2547		Support new HL7 v2.7 truncation character (#) in MSH-2
MIRTH-2543		webadmin in 3.0 can't login in a mapped local ip.
MIRTH-2540		Add ability to reverse the Sort Files By setting in the File Reader
MIRTH-2538		Moving a JavaScript Filter in the Filters list causes it to be renamed
MIRTH-2532		Consolidate error types for alerts
MIRTH-2519		Add ability to import/export specific transformer steps
MIRTH-2513		Add support for storing the message received_date with milliseconds in MySQL

[ ] Has Active Tag
[✓] Unassigned (User)
[ ] Unassigned (Versio
[✓ Open]
Resolved
Closed

[✓] Optimizations (3)

Figure 4.11: Quick Filters with several filters enabled.

4.2.6 Search

A search field in the upper-right corner of the interface allows triagers to perform a search of the bug reports in the bug list. If the search term entered into the field matches a bug identifier exactly, with or without the project prefix (ex. MIRTH-1968 or 1968), the exact bug will be displayed in the bug list. All other search terms will be partially matched against the bug summary or description and the results will be displayed in the bug list.

4.2.7 Assigning Bugs

Because of the layout of the PorchLight interface, it is possible to simultaneously inspect detailed information about a bug, view it within the context of the bug list, and assign it all within one perspective. Assignments can be easily performed through drag-and-drop: a bug, or set of bugs that have been selected via Ctrl-click, can be dragged from the bug list onto a user or milestone icon on the respective lists. The icon in the list is then highlighted to provide a visual indicator that the bug can be dropped onto the list. Releasing the mouse will drop the bug onto the user or milestone and the assignment will be made. The assigned bug count of either the user or milestone is then updated in real-time to reflect the action taken after a bug or set of bugs is dropped.

4.2.8 Quick Comment

The Quick Comment dialog shown in Figure 4.12 allows triagers to add a new comment to a selected bug report. The triager has the choice of either entering a new comment, or selecting from a list of snippets that will populate the comment text field. The snippets are defined in a separate configuration file that triagers can edit to include additional snippets. The status of the bug can also be optionally changed from this dialog. The following Quick

Comment snippets are included by default:

- "Please provide a screenshot of the defect described."
- "Please provide a stack trace of the runtime exception described."
- "There is not enough information to address this defect, please include additional steps to reproduce."

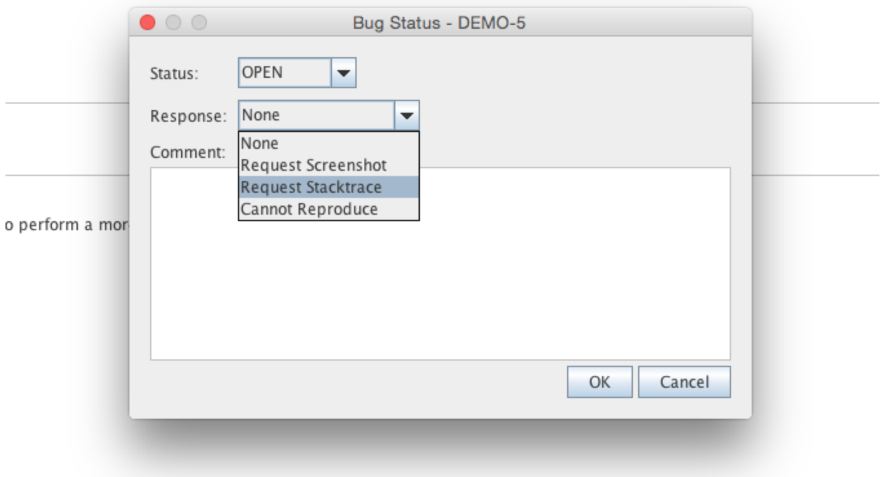
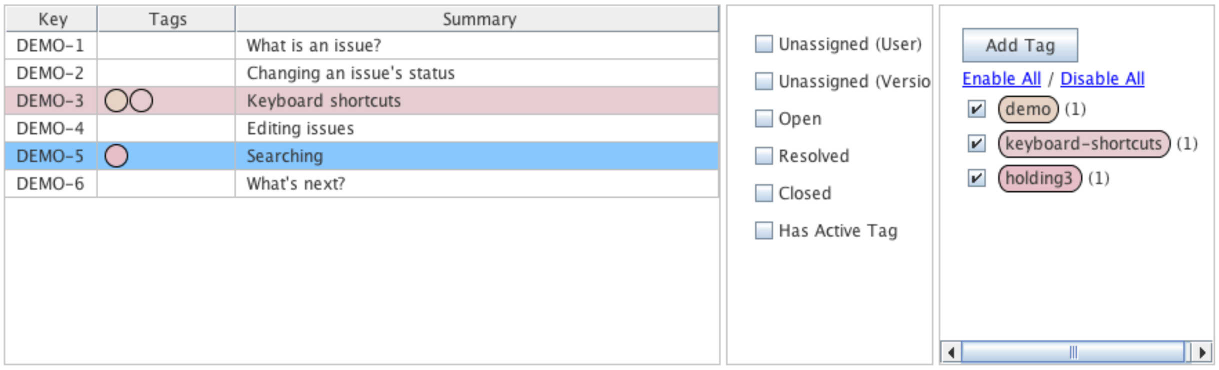


Figure 4.12: Quick Comment feature displaying commonly used comments.

4.2.9 Tag Sets

The basic features of PorchLight described thus far make it possible for a triager to assign groups of bugs to users and milestones (i.e., by using Ctrl-click to select multiple bugs and then dragging them to the desired user or milestone), but not necessarily very conveniently. Triagers still must manually select multiple bugs, and there is nothing they can do with the selection other than assigning it.

To better support triagers in exploring, working with, and assigning bugs in groups rather than individually, PorchLight employs the concept of a *tag set*, or a collection of bugs that is indexed by a tag, or unique keyword. Tag sets are managed in the tag set list where existing tag sets are listed with their names and associated color. Within this list, each tag set can be selected or deselected independently from the other tag sets. When a tag set is selected from the list, every bug that belongs to the tag set is identified in two ways. First, a new marker is added to the *Tags* column with the color that is assigned to the tag set. Additionally, the rows in the table are highlighted using the same tag set color. Deselecting a tag set removes the markers and row coloring from the bug list.



Key	Tags	Summary
DEMO-1		What is an issue?
DEMO-2		Changing an issue's status
DEMO-3	●●	Keyboard shortcuts
DEMO-4		Editing issues
DEMO-5	●	Searching
DEMO-6		What's next?

☐ Unassigned (User)
☐ Unassigned (Versio
☐ Open
☐ Resolved
☐ Closed
☐ Has Active Tag

☑ demo (1)
☑ keyboard-shortcuts (1)
☑ holding3 (1)

Figure 4.13: Tag set list and indicators in bug list.

Each tag set's color is also used to color a solid dot indicator in the *Tags* column of the bug list to denote that the bug belongs to the specific tag set (shown in Figure 4.13). If a bug belongs to multiple tag sets, as is often the case, then multiple dots are displayed in the cell. To make it easy to identify tag sets that a bug belongs to when scrolling through the bug list, we also chose to highlight the entire row in the table with the color of the tag set. In the case when a bug belongs to multiple tag sets, the row is highlighted using the color of the tag set containing the most number of bugs.

Colored markers were chosen to represent tag set membership since they are easy to detect when visually scanning through the bug list. We are also able to display more tags sets within the table cell as dots compared to displaying the entire tag name. Square markers indicate static tags that have been added to the bug from the source bug tracker. Round markers indicate dynamic tags, or tags that have been added to the bug through the creation of a new tag set.

To make tag sets more approachable, PorchLight includes several tag sets that represent criteria that are commonly used in triaging. These tag sets are included by default so that a triager does not need to create them each time they use PorchLight with a new project. The following predefined tag sets are included:

- Popular: bugs that have had more than three comments
- Missing Details: bugs that do not have a screen shot or stack trace attached
- Hot Potato: bugs that have been reassigned twice or more
- Zombie: bugs that have been open more than one month and have had no activity

It is important to note that all tag sets are always available to the triager from the tag set list and can be updated by recreating the tag set; that is, as new comments are made on bugs, stack traces submitted, or reassignments done (as just some examples), if one or more bugs now matches the criteria of a tag set to which it did not yet belong, this tag set or these tag sets will be updated to contain the bug (and it might be removed from others).

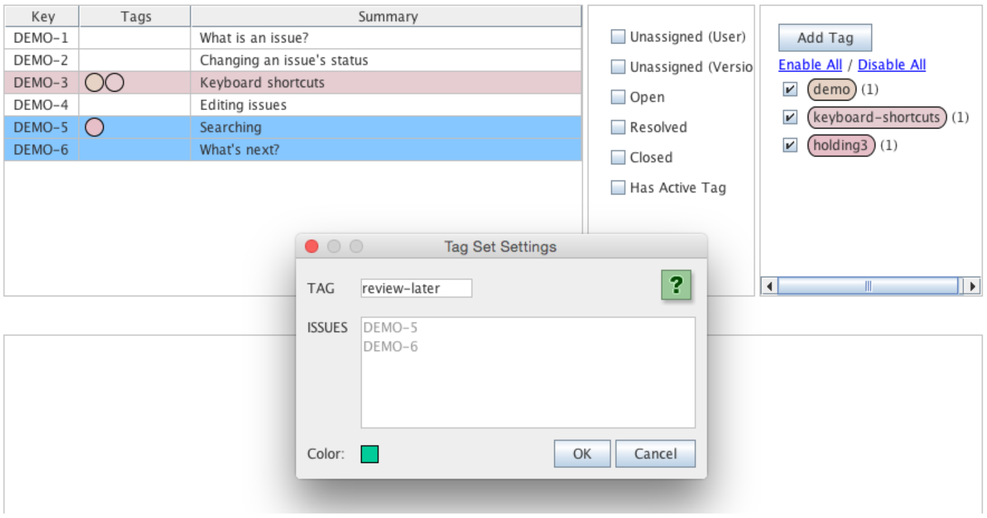
Working with Tag Sets

The true power of PorchLight lies in the ability to create new tag sets to help organize the collection of bugs. Because tag sets persist across triaging meetings, it is not only possible to continue where one left off, but also that the tag sets created by the triager can become a vehicle for sharing common rules and criteria for grouping bug reports (thereby establishing community norms). If a tag set has been created in error, it can easily be removed by right-clicking on a tag set in the list and selecting *Delete Tag*. This will remove the tag set and disassociate it from the bugs in the bug list. Tag can sets can be created using several techniques.

First, triagers can create tag sets to view view all the bugs assigned to a specific user or milestone. To do this, a triager can simple drag the user's icon from the user list on the left, or the milestone icon from the milestone list on the right, to the tag set area. This automatically creates a new tag set automatically that can be used henceforth. The new tag set is automatically assigned a random color selected from a pallet of pleasant pastels when it is created, though the user can manually select a color as well. The color is used as the background of the tag set label in the tag set list.

Triagers can also create tag sets from manually selected bugs from the bug list. The first requirement we presented in Chapter 3 motivates the need for such ad hoc tag sets: setting aside bugs that cannot be immediately triaged, grouping bugs that might seem unrelated but that the triager wants to keep together, or keeping track of tentative plans to be confirmed later. PorchLight supports such ad hoc tag sets by allowing triagers to select bugs from the bug list and dragging them (one-by-one or as a group after Ctrl-clicking them) onto a tag set in the tag set list. If the desired tag set does not yet exist, the triager can drop the selected bugs onto any empty area in the list. The tag set creation dialog is then displayed to allow the triager to specify the name and color of the new tag set.

These features are convenient for creating relatively simple tag sets. However, triagers may also want to create tag sets based on criteria that are too complex to describe by simply selecting bugs from the bug list. For example, they may want to identify bugs created within a period of time, or that have been reassigned a certain number of times. This can be accomplished using the tag set editor. This feature is particularly important, since triaging is a dynamic activity where the set of bugs of interest changes over time and cannot be predicted ahead of time, nor does the tag set always follow convention.



Key	Tags	Summary
DEMO-1		What is an issue?
DEMO-2		Changing an issue's status
DEMO-3	○○	Keyboard shortcuts
DEMO-4		Editing issues
DEMO-5	○	Searching
DEMO-6		What's next?

☐ Unassigned (User)
☐ Unassigned (Versio
☐ Open
☐ Resolved
☐ Closed
☐ Has Active Tag

Add Tag
Enable All / Disable All
☑ demo (1)
☑ keyboard-shortcuts (1)
☑ holding3 (1)

Tag Set Settings

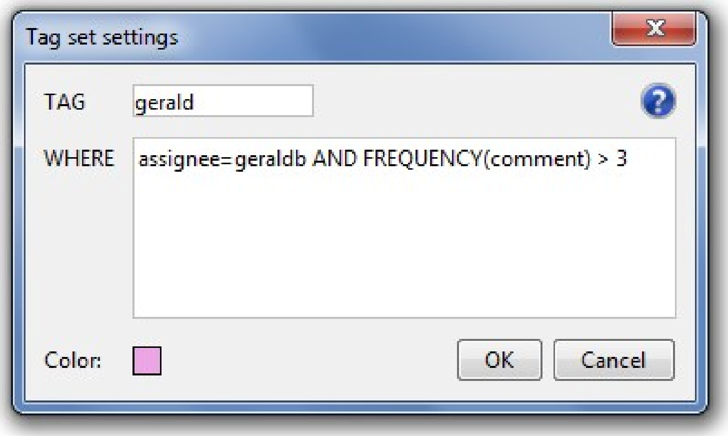
TAG review-later

ISSUES DEMO-5
DEMO-6

Color: 🟩

Figure 4.14: Creating a new ad hoc tag set.

To create these more complex tag sets, triagers can use a defining feature of PorchLight, a Bug Tagging Language, that allows triagers to create tag sets by expressing criteria for selecting bugs. Figure 4.15 shows how a new tag set can be created to tag all bugs assigned to user "geraldb" with at least four comments attached. Even after a tag set has been created with a given expression, it is possible to update the underlying criteria of a dynamic defined tag set. Simply double clicking brings up the dialog box with the current expression that can then be edited.



Tag set settings

TAG gerald

WHERE assignee=geraldb AND FREQUENCY(comment) > 3

Color: 🟪

Figure 4.15: Editing a BTL statement in PorchLight.

4.3 BTL: Bug Tagging Language

To support the creation of these complex tag sets, we implemented a *domain-specific set creation language* that allows triagers to:

1. Express basic tag sets using bug status and fields as criteria;
2. Leverage actions, specifically their presence and frequency;
3. Restrict tag sets to a specified window of time;
4. Express compound tag sets from a set of statements; and
5. Extend the language to support new fields, actions, and computations across each.

In BTL, bug tag sets are expressed through statements, which consist of clauses like `TAG` and `WHERE`, combined with predicates to specify attributes of the bug defined in the model. BTL statements follow the below syntax[1]:

```
TAG name
WHERE predicates
[SINCE LAST period]
```

As a simple example, a statement to find all bugs assigned to a user Gerald would be:

```
TAG "Gerald's Bugs" WHERE assignee = "Gerald"
```

The `TAG` clause is equivalent to the `SELECT` clause in SQL. It denotes the creation of a tag set with the specified name. The `WHERE` clause consists of a series of predicates, for example:

```
assignee = "Gerald"
```

[1]Throughout this document, BTL clauses and functions are shown in upper-case lettering while fields and statuses are shown in lower case. This is to make the expressions more readable; BTL is case insensitive except for quoted values.

The WHERE clause supports the AND and OR Boolean operators between predicates. As an example, a statement to find all unresolved bugs assigned to either Gerald or Jacob can be specified as follows:

```
TAG "Gerald's or Jacob's Bugs" WHERE status = "Open" AND (assignee = "Gerald"
   OR assignee = "Jacob")
```

These examples demonstrate the basic querying capabilities of BTL using simple attributes. While BTL is similar in many ways to SQL, it is intended to be a more compact and specific language to allow triagers who may be non-technical to create tag sets. To fully realize the capabilities of our language and of PorchLight, we have also included features that can be used to create more meaningful tag sets that go beyond simple filters.

4.3.1 Functions

The predicate in the WHERE clause of a BTL statement can reference a bug's status (e.g., resolved), a field (e.g., summary), or an action (e.g., assignment). The distinction between these attributes becomes relevant when actions are used in conjunction with *functions*, which allow for computations to be performed on the specified status, field, or action. Functions enable more complex statements that can incorporate not only the content of a bug report, but also the history of actions on the bug. For example, a statement to find all bugs which have been assigned three or more times would be:

```
TAG "3-Toss Bugs" WHERE FREQUENCY(assignment) >= 3
```

In this case, the FREQUENCY function computes the frequency of the assignment action over the life of the bug. The result of this computation is a *derived attribute* (in this case, the number of assignments) that is then used in the evaluation of the WHERE clause. Table 4.2 summarizes the functions supported natively in the WHERE clause. While all functions must have a return value, the the value can be an integer, string, or Boolean value. Though it is not required, most functions have a parameter that is used in the computation. For example, the `CONTAINS` function takes in a keyword as a parameter that is uses to search the bug's summary and description. On the other hand, the `FREQUENCY` function takes in an action as a parameter that must be one of the actions that are part of the data model.

Function	**Description**
CONTAINS(*keyword*)	Returns true if the bug summary or description contain the specified keyword, false otherwise.
FREQUENCY(*action*)	Returns the integer frequency of the specified action.
HAS(*field*)	Returns true if the specified field is present in the bug report (e.g., stack trace or screenshot), false otherwise.

Table 4.2: Functions available natively in the Bug Tagging Language.

As another example, a statement to find all bugs which contain the keyword "printer" and have five or more comments would be:

```
TAG "Popular Printer Bugs" WHERE CONTAINS("printer") AND FREQUENCY(comment)
   >= 5
```

4.3.2 Time Windows

Another feature of BTL is the ability to specify time windows in combination with bug actions. For example, a statement to find all unresolved bugs that have been reassigned exactly once and that have been updated in the last two months would be:

```
TAG "Reassigned Last Month" WHERE status = "Open" AND FREQUENCY(assignment) =
   1 SINCE LAST "2 months"
```

The `SINCE LAST` clause restricts the results of the statement to bugs that have been last updated within the specified time window using a natural language expression of periods.

For example, “2 months”, “6 weeks”, and “1 years” are valid values for the clause. Only a single time window can be specified as part of the clause.

As another example, a statement to find all unresolved bugs that have received less than five comments in the last six months would be:

```
TAG "Less Than 5 Comments" WHERE status = "Open" AND FREQUENCY(comment) < 5
   SINCE LAST "6 months"
```

To fully appreciate the need for both functions and time windows in BTL, suppose that a triager must review thousands of new bug reports to take an initial pass at categorizing them. She may start the process by creating a new tag set to identify bugs which have been opened in the last few months, that affect the user interface, and that do not include a screenshot.

```
TAG "UI Bugs Without Enough Info" WHERE status = "Open" AND component = "UI"
   AND HAS(screenshot) = "false" SINCE LAST "2 months"
```

Without the additional clauses, this tag set would result in too many bugs to accurately triage. With the added clause, the triager can easily perform an action on the selected tag set, like request additional information.

4.3.3 Custom Fields and Actions

Since it is difficult to predict the tag sets a triager may want to create using BTL, or the types of data that will be available in a source bug tracker, we have designed BTL to be an extensible language. Custom fields and actions are extension mechanisms that allow triagers to define new fields or actions that can be used as predicates in statements or parameters in functions.

An example of a custom action is one that provides information about source code commits associated with a bug. A new action, called *Commit*, could be added to represent a source code commit that is linked to the bug. Since most bug trackers provide some way to link a commit to a particular bug, usually by providing the bug's identifier in the commit log message, the logic for this extension could scan the source repository commit logs and record the action for each bug. This new action would then enable the following statement that would create a tag set with all bugs that have been linked with at least two source code commits in the last two weeks:

```
TAG "Source Change Last Week" WHERE FREQUENCY(commit) >= 2 SINCE LAST "2
   weeks"
```

Custom fields and actions are created by implementing interfaces that are then loaded into the PorchLight runtime environment on startup. The code for these classes is executed during the import process, and as bugs are created in the bug graph and attributes are mapped, the custom fields and actions are generated. This process is discussed in more detail in Section 5.3.

4.3.4 Custom Functions

A custom function is a way for triagers to define new functions that can be applied in BTL statements to fields or actions. Custom functions are similar to *functions* in many SQL procedural languages.

As an example, suppose an custom function existed that would return a count of the number of dependents a bug has. Dependents are determined by the relationship in the bug tracker, but could be bugs that are sub-tasks of the parent bug, or bugs that have been linked as related to the bug. This function, called `DEPENDENTS`, could be invoked as part of the execution of a BTL statement and would count the number of dependent bugs. The implementation of the `DEPENDENTS` function would of course specify how a dependent bug is defined. This new function allows the following statement that creates a tag set with all bugs that have two or more dependents:

```
TAG "Two Plus Dependents" WHERE DEPENDENTS >= 2
```

Both custom fields and functions are features of BTL which make it not only a powerful language for creating tag sets, but also a platform on which to explore and build new approaches to bug triaging that can make the process more efficient. For instance, many of the recent approaches to assignment automation, such as the ones discussed in Chapter 2, can be implemented as a combination of BTL clauses and custom functions to create a tag set based on the recommended bugs. Consider a machine learning technique that extracts metadata from previously assigned bugs and can be used to predict an assignee [24]. Assuming the classifier has been trained on the bug graph to date, a process which could be done offline, a new custom function named RECOMMENDED could be implemented to determine if a specific user is the recommended assignee for a bug. This function would take in a developer as a parameter and, when invoked, would search the resolved bug history to return a new tag set consisting of bugs that *could* be assigned to the developer.

```
TAG "Recommended for Brad" WHERE RECOMMENDED("brad") = true
```

The advantage of this approach is that the assignments are not made automatically. The triager is kept in the loop and can review the recommendations and take action (by performing the actual assignment) in a way that is more efficient than reviewing each bug individually.

In summary, developers can expand the capabilities of PorchLight using custom fields, actions, and functions. Custom fields and actions allow it to be integrated with most bug trackers, since any fields and actions that are not part of the bug model can be added. Custom functions provide developers with the ability to implement new functions that can expand the expressiveness of BTL. This flexibility can be used to explore new approaches to bug triaging, and PorchLight provides the beginning of a platform on which the techniques can be built and tested.

4.4 Scenarios

Thus far, we have discussed the features in PorchLight that we designed to support the bug triaging process largely in isolation, one-by-one. To illustrate how all of these features work in concert, we now describe three hypothetical scenarios: (1) a group of developers participating in a release planning session, (2) an individual developer reviewing her assigned bugs, and (3) a volunteer triager working through new bugs.

4.4.1 Release Planning Session

Suppose that a group of three software engineers are gathered in a meeting room to plan the next release of the product they develop. The most recent version, 2.5, was released three months ago, and since then many new bug reports have been created. The developers have already assigned bugs to the next several releases (2.6, 2.7, and 2.8, to be precise), but as they near the 2.6 release date, they want to ensure that any bugs that have been reported against the previous release have been triaged and have a chance to be fixed.

Using PorchLight, they start by first looking at the open bugs that have been reported against the previous 2.5 release and that are unassigned. They enable the *Unassigned (User)* and *Unassigned(Version)* filters to ensure that they are only viewing unassigned bugs. They then create a new tag set using the *Status* and *Affects Version* field to find the relevant bugs:

```
TAG "Affects Last Release" WHERE status = "OPEN" AND affectsVersion = "2.5"
```

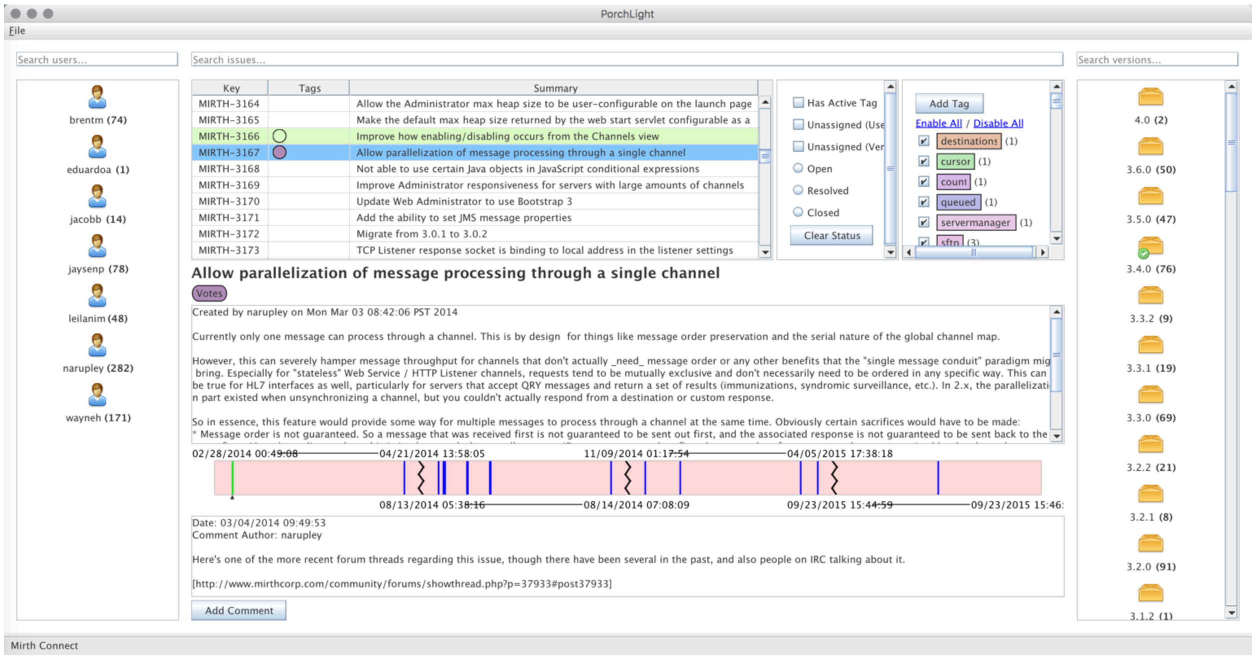
PorchLight executes the BTL statement and a new tag set labeled “Affects Last Release” appears in the tag set list. The matching bugs are highlighted in the bug list with the tag set indicator and assigned color. They enable the *Has Active Tag Set* filter to narrow the list down to just those that belong to the tag set.

They decide to expand this list by also including the bugs that may affect the security of the software. These bugs are particularly important since they may reveal a vulnerability, and the developers want to ensure that they have been reviewed and assigned in a timely manner. They know that security related bugs usually contains keywords like "permissions," "password," or "authentication," so they create a new tag set to find these bugs using the `CONTAINS` custom function:

```
TAG "Security" WHERE CONTAINS("permissions") = true OR CONTAINS("password") =
   true OR CONTAINS("authentication") = true
```

Once again, PorchLight executes the BTL statement and a new tag set labeled "Security" appears in the tag set list. Since the "Affects Last Release" tag set is still enabled in the tag set list, the updated bug list shows both bugs that have been reported against 2.6, or that contain any of the security-related keywords in the summary or description text. As they review the bugs, they notice one particular bug, shown in Figure 4.16, that has received numerous comments from users of the software in the last few weeks. A clear indication that this might be an important bug, they decide that they can fix it in the next release, and they assign it by dragging it onto 2.6 in the milestone list.

After an hour of triaging, they have made significant progress in assigning 37 bug reports. Most have been assigned to the next release, others have been assigned to future releases, and some were closed. Each of the developers returns to their desks and open their web-based bug tracker to view the new bugs that have been assigned to the 2.6 release, and proceed to work on resolving them.



Key	Tags	Summary
MIRTH-3164		Allow the Administrator max heap size to be user-configurable on the launch page
MIRTH-3165		Make the default max heap size returned by the web start servlet configurable as a
MIRTH-3166	○	Improve how enabling/disabling occurs from the Channels view
MIRTH-3167	●	Allow parallelization of message processing through a single channel
MIRTH-3168		Not able to use certain Java objects in JavaScript conditional expressions
MIRTH-3169		Improve Administrator responsiveness for servers with large amounts of channels
MIRTH-3170		Update Web Administrator to use Bootstrap 3
MIRTH-3171		Add the ability to set JMS message properties
MIRTH-3172		Migrate from 3.0.1 to 3.0.2
MIRTH-3173		TCP Listener response socket is binding to local address in the listener settings

Allow parallelization of message processing through a single channel

Created by narupley on Mon Mar 03 08:42:06 PST 2014

Currently only one message can process through a channel. This is by design for things like message order preservation and the serial nature of the global channel map.

However, this can severely hamper message throughput for channels that don't actually _need_ message order or any other benefits that the "single message conduit" paradigm mig
bring. Especially for "stateless" Web Service / HTTP Listener channels, requests tend to be mutually exclusive and don't necessarily need to be ordered in any specific way. This can
be true for HL7 interfaces as well, particularly for servers that accept QRY messages and return a set of results (immunizations, syndromic surveillance, etc.). In 2.x, the parallelizati
n part existed when unsynchronizing a channel, but you couldn't actually respond from a destination or custom response.

So in essence, this feature would provide some way for multiple messages to process through a channel at the same time. Obviously certain sacrifices would have to be made:
* Message order is not guaranteed. So a message that was received first is not guaranteed to be sent out first, and the associated response is not guaranteed to be sent back to the

Date: 03/04/2014 09:49:53
Comment Author: narupley

Here's one of the more recent forum threads regarding this issue, though there have been several in the past, and also people on IRC talking about it.

[http://www.mirthcorp.com/community/forums/showthread.php?p=37933#post37933]

Figure 4.16: Example of bug that has received numerous comments.

4.4.2 Individual Developer

Suppose that Nicole, a software engineer, is using PorchLight to review the bugs that have been assigned to her for the upcoming development sprint. She starts by enabling a tag set she created called "My Bugs" which identifies unresolved bugs that have been assigned to her:

```
TAG "My Bugs" WHERE STATUS = "Open" AND assignee = "nicole"
```

To help prioritize her work for the day, she decides to create an additional tag set to identify the bugs where her product manager, David, has left a comment. These are bugs that usually contain useful feedback and guidance that she can use when implementing a new feature. In order to create such a tag set, she first implements a new custom function called `COMMENTED` that returns true if the specific user has commented on a bug. She then creates the desired tag set using the newly implemented function:

```
TAG "Commented by David" WHERE COMMENTED(david) = true
```

PorchLight executes the BTL statement and a new tag set labeled “Commented by David” appears in the tag set list. She scans the bug list for any bugs that have belong to both of the enabled tag sets, and notices that three bugs have received comments from David. She selects one of the bugs that is both assigned to her, and has been commented on by her product manager. Based on this latest comment, she realizes that the enhancement being requested is in a module that she is not familiar with. After quick instant message conversation with Alan, the software engineer who usually works on the module, she selects the bug from the bug list and drags it over to his name in the user list, assigning it to him and removing it from her assigned bugs.

Having reviewed the bugs with with feedback, Nicole disables “Commented by David” tag set and creates a new tag set to view the bugs that have been assigned to the next release. She does this by dragging the milestone, in this case 2.6, from the milestone list onto the tag set list. A new tag set appears labeled “2.6” and the bugs belonging to the tag set appear in the bug list. Since she still has the “My Bugs” tag set enabled, she is able to quickly identify the bugs which have been assigned to her for the release.

After reviewing her assigned bugs, she realizes that that there are too many bugs to complete before the end of the sprint. She selects several of the bugs from the bugs list and drags them onto the tag set list, which pops up a dialog box where she enters “To Review” as the new ad hoc tag set name. She will review the bugs in this tag set with David after the stand up meeting to discuss if they can be reassigned to a later release.

4.4.3 Volunteer Triager

Suppose that Wayne, a volunteer triager for a popular open source project, finds some time to help by triaging the ever-growing list of new bug reports. Because he is one of the more one of the more experienced triagers on the project, Wayne makes extensive use of tag sets as a way to review the open bugs. He begins by creating a new tag set to tag the unassigned bugs that have been reported since his last triaging session:

```
TAG "Since Last Session" WHERE status = "OPEN" SINCE LAST "2 months 3 days"
```

PorchLight executes the BTL statement and a new tag set labeled "Since Last Session" appears in the tag set list. He does a cursory scan of the bug list and sees that there are over 50 new bug reports that have been created. To narrow this list down, he enables his tag set which contains bug reports that may require additional information, such as a screenshot of it is a bug affecting the UI component, or a stack trace if it is describing an error message. These tag sets make use of the *Component* field, and the `HAS` and `CONTAINS` functions.

```
TAG "Needs Screenshot" WHERE status = "OPEN" AND component = "UI" and
   HAS(screenshot) = false
```

```
TAG "Needs Stacktrace" WHERE status = "OPEN" AND CONTAINS("error") and
   HAS(stacktrace) = false
```

With these additional tag sets enabled, Wayne is able to identify the bugs in the bugs list which belong to two or more tag sets, that is, those which have been reported since his last triaging session and that are missing a screenshot (if related to the UI) or a stack trace (if describing an error message). He selects these bugs and uses the Quick Comment feature to add a comment to each requesting the additional details, and saving the development team the effort of reviewing them with insufficient information.

After triaging the bugs that require additional information, he looks to see if he can assign some bugs to developers working on the project. To help with this process, some of the developers have shared tag sets that make use of the previously described `RECOMMENDED` custom function to identify bugs that could potentially be assigned. This function takes into account previous assignments based on the description of the bug and determines if a bug is a "good fit" for a particular developer. Jacob, a frequent contributor to the project with expertise is in troubleshooting defects related to the database, has provided the triagers a tag set based on the following BTL statement:

```
TAG "Database Bugs for Jacob" WHERE status = "OPEN" AND component =
   "database" AND RECOMMENDED("jacob") = true
```

Wayne disabled all other tag sets, then enables “Database Bugs for Jacob” in the tag set list. He browses through the list of bugs that have been recommended for Jacob. Based on the description he decides that it is indeed a bug that Jacob should be able to fix, and he assigns it to him for the next release by dragging the bug to Jacob’s icon in the user list, and then the next release in the milestone list.

4.5 Discussion

To summarize our approach, we revisit our initial requirements for supporting bug triaging presented in Chapter 3, and describe how the design of PorchLight addresses each of them.

Requirement 1: Explore

The first requirement is to provide triagers with the ability to browse new and unassigned bugs and to easily organize them into categories for further triaging.

PorchLight addresses this requirement in several ways. First, it provides triagers with the entire contents of the bug tracker along with useful tools, like the user and milestone lists, to make sense of the task at hand: to review and triage a large number of bugs. Quick filters let triagers quickly cull the list down to bugs of interest, and the ability to scroll through the bug list and see which bugs have been assigned to which user and/or milestone makes it easy to get a sense of how much work needs to be done.

More significantly, tag sets provide a way for triagers to dynamically explore and partition the collection of bugs into meaningful groups that they can use to plan their work. Viewing the bug tracker not as an intimidating list of unassigned bugs, but as a collection of meaningful tag sets—like unassigned bugs related to the UI with no screenshots—gives triagers a way to tackle the flood of bugs in a project.

Requirement 2: Search

The second requirement is to provide triagers with the ability to search for bugs using different criteria (time, activity, missing elements, etc.), as well as to group them based on these criteria.

PorchLight addresses this requirement by leveraging the tagging capabilities of the Bug Tagging Language. BTL provides triagers with an extensible way of expressing the criteria needed to create tag sets that are meaningful to triaging. Language features such as time windows and functions allow for complex criteria to be used in finding the bugs that should be tagged. PorchLight also provides search capabilities within the bug list to make it easy for triagers to quickly find and navigate to a specific bug.

Requirement 3: Inspect

The third requirement is to provide triagers with the ability to review comments, histories, and related bugs to make informed triaging decisions.

PorchLight addresses this requirement by providing triagers with a detailed bug view which includes all of the captured information about the bug, along with a timeline which provides an at-a-glance visualization of the history of activity for a particular bug. The timeline also encapsulates numerous details, depending on the action type, that the triager can access to make triaging decisions, like files that have been committed as part of fixing a bug. Information about the bug is also inferred from the tag set(s) to which the bug belongs. Seeing that a bug belongs to multiple tag sets, and viewing the BTL expression which generated the tag sets, can provide useful insights when inspecting a bug.

Requirement 4: Take Action

Finally, the fourth requirement is to provide triagers with the ability to easily assign a bug, comment on it, assess any feedback, and move on to the next one.

PorchLight addresses this requirement by offering all of its functionality in a single interface so that triagers can take action and avoid the need for context switching (i.e., the back and forth syndrome of many bug trackers). Our design makes it feasible to assign a bug without leaving the current triaging context. Organizing, sorting, and filtering bugs, viewing of the details and history of individual bugs, and assigning bugs are all available at hand. The unique layout also makes it possible for triagers to drag-and-drop from the bug list to the user and milestone lists to take action.

Finally, PorchLight provides visual feedback when as assignment action is performed in the form of updated bug counts, which make it easier for triagers to asses the effect of that particular action on the project as a whole.

4.5.1 Conclusion

Combined, the features in PorchLight transform a *static* bug list to a *dynamic* workspace in which the bugs can be organized (through enabling and disabling tag sets, using the filters, and sorting the list), worked with (by selecting bugs of interest and viewing their details, assigning them to ad-hoc tag sets), and assigned (by dragging either individual bugs or entire tag sets onto developer and milestone icons). All the while, PorchLight gives clear, visible feedback on the actions that the triager performs, through the already described counts on the developer and milestone icons, the coloring of individual bugs in the bug list, and the

Explore	• Single view with user and milestone lists • Quick filters • Viewing status of a selected bug • Tag sets
Search	• BTL • Advanced features like time windows
Inspect	• Detailed bug view • Timeline
Take Action	• Single-pane interface • Drag-and-drop interaction • Visual feedback

Table 4.3: Features in PorchLight that address the requirements for a dedicated triaging tool.

markers that indicate to how many tag sets a bug belongs. Given that this information is available at-a-glance, a triager can flexibly move back and forth between the higher-level task of organizing and finding bugs and the lower-level task of inspecting and if so desired assigning bugs, without losing the overall context in which they are working.

Chapter 5

Implementation

In this chapter, we describe the implementation details behind PorchLight, including:

1. The data model that represents the information and history about a bug report and that is used for creating tag sets.
2. The language with querying and tagging capabilities based on the data model.
3. The extension of both the data model and tagging language using a plugin architecture.

We end the chapter by describing some of the challenges encountered during the design and implementation of PorchLight.

5.1 Data Model

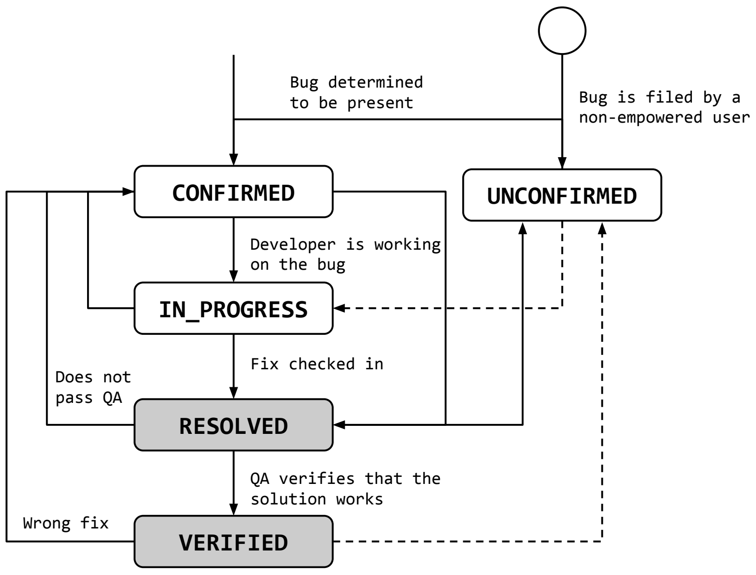
In order for PorchLight to be flexible enough to integrate with existing bug trackers, we had to first design a bug data model to standardize the data that is available. With a sufficiently rich data model, we can represent the status, fields, and history of a bug report to support building tag sets. Looking towards the more significant contributions of our tagging language, we must also be able to capture metadata about the bug report to form more meaningful tag sets.

5.1.1 Status

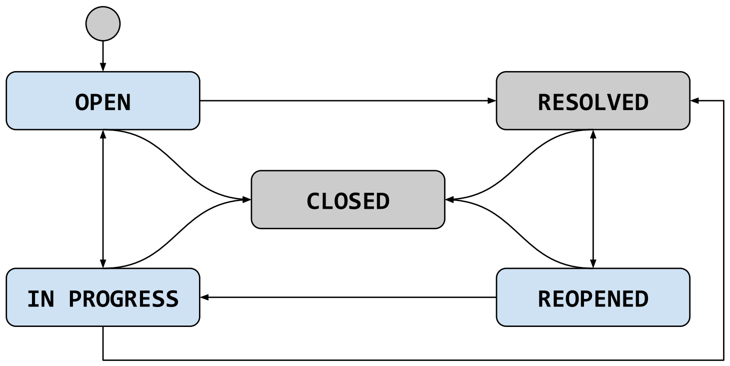
The starting point for our data model was the status, or the state of the bug report in the workflow. We began by comparing the default bug report life cycles from Buzgilla, JIRA, and Trac (Figure 5.1). These workflows represent the different statuses a bug report can have, from when it is first reported to when it is eventually resolved or closed, and the possible transitions between each status. Each model differed slightly in the names of the statuses and the transitions. For example, the default status for a newly created bug in Bugzilla is *Unconfirmed*, meaning that the bug has yet to be triaged and confirm as a bug, while the default state in both JIRA and Trac is *New*, and the transition to the *Accepted* status signifies that the bug has been triaged.

In our comparison of statuses, we observed that the two bug trackers that are developed under an open source model, Bugzilla and Trac, have a default starting bug status that signifies an *unconfirmed* bug, and only after it has been reviewed is it considered an actual bug. This is in contrast to JIRA, a commercially developed bug tracker, where the starting bug status is *open*, and the bug is considered ready to be assigned. This suggests that bug trackers used in open source projects typically see a larger volume of bug reports from a wider audience, and this preliminary status assists in the triaging process.

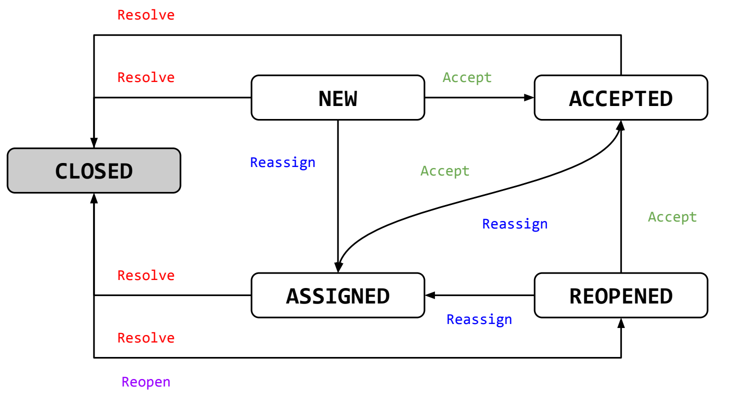
Because our data model needs to represent bugs from existing bug trackers, we chose statuses that would capture the most common workflow and that would allow a tool to import, or to ignore, statuses depending on what is available in the source bug tracker. The statuses selected for the data model are listed in the *Bug Data Model* column of Table 5.1.


Bug determined to be present
Bug is filed by a non-empowered user
CONFIRMED
UNCONFIRMED
Developer is working on the bug
IN_PROGRESS
Fix checked in
Does not pass QA
RESOLVED
QA verifies that the solution works
Wrong fix
VERIFIED

(a) Default Bugzilla bug life cycle.



(b) Default JIRA bug life cycle.



(c) Default Trac bug life cycle.

Figure 5.1: Default bug life cycles for popular bug trackers.

Bugzilla	**JIRA**	**Trac**	**Bug Data Model**
Unconfirmed		New	*New*
Confirmed	Open	Accepted	*Open*
In Progress	In Progress	Assigned	*In Progress*
Resolved	Resolved		*Resolved*
Verified	Closed	Closed	*Closed*
	Reopened	Reopened	*Reopened*

Table 5.1: Comparison of statuses from default workflows of Bugzilla, JIRA, and Trac, and the statuses selected for the Bug Data Model

5.1.2 Fields

Along with the status of a bug report, our data model must also represent the fields that capture information about the bug. Our goal was to select the fields that would allow us to capture the information necessary to demonstrate the feasibility and extensibility of our approach. We began by comparing the fields from the default software development workflows of several popular bug trackers: Bugzilla, JIRA, and Trac (see Table 5.2).

Bugzilla	**JIRA**	**Trac**
Summary	Summary	Summary
Component	Component/s	Component
Additional Comments	Description	Description
Assigned To	Assignee	Assigned to/Owner
Reporter	Reporter	Reporter
Priority	Priority	Priority
Version	Affects Version/s	Version
Target Milestone	Fix Version/s	Milestone
Resolution	Resolution	Resolution
CC list	Watchers	Cc
Platform and OS		
Attachments	Attachments	
Keywords	Labels	Keywords
	Type	Type
Severity		

Table 5.2: Comparison of fields from default workflows of Bugzilla, JIRA, and Trac.

Our approach was not to simply take the union or intersection of the fields in all of these bug trackers. Rather, we applied some basic rules to determine which would be selected for the data model. If a field was available in all of the bug trackers, like *Summary* and *Priority*, we selected it for the data model. Some of the fields represented the same information but had different names, like *Labels* and *Keywords*, in which case we chose the name used in JIRA since that is the bug tracker the author is most familiar with. In the case where a field was present in only one default workflow, like *Severity*, we made a determination to exclude it from the data model to the extent that it did not limit our ability to provide sufficient information for triaging.

Additional metadata that have been included in the data model are the *Date Created* and *Date Updated* fields. These fields capture a timestamp of when the bug was initially created, and when it was last modified, respectively. They are captured as part of the data model to be used when evaluating BTL expressions with time windows. For example, the `SINCE LAST` clause makes use of the *Date Updated* field to select bugs which fall within the specified time window. Finally, an *Identifier* field has been added so that each bug can be uniquely identified. The process of selecting fields from the default workflows, and including additional metadata fields, resulted in the fields for the data model listed in Table 5.3.

5.1.3 Actions

Thus far, our model captures the static information that is available in a bug report. In order to describe meaningful tag sets, we must also be able to capture the history of the bug as it is updated. There are different version control techniques we could employ for capturing these changes, such as persisting successive snapshots of the entire bug report, or representing deltas of the data model between changes [48]. For our approach, we have chosen to instead capture the changes and transitions themselves, since that is the most common outcome of bug triaging. We represent each transition as an action that can be derived from the history of modifications to the bug report.

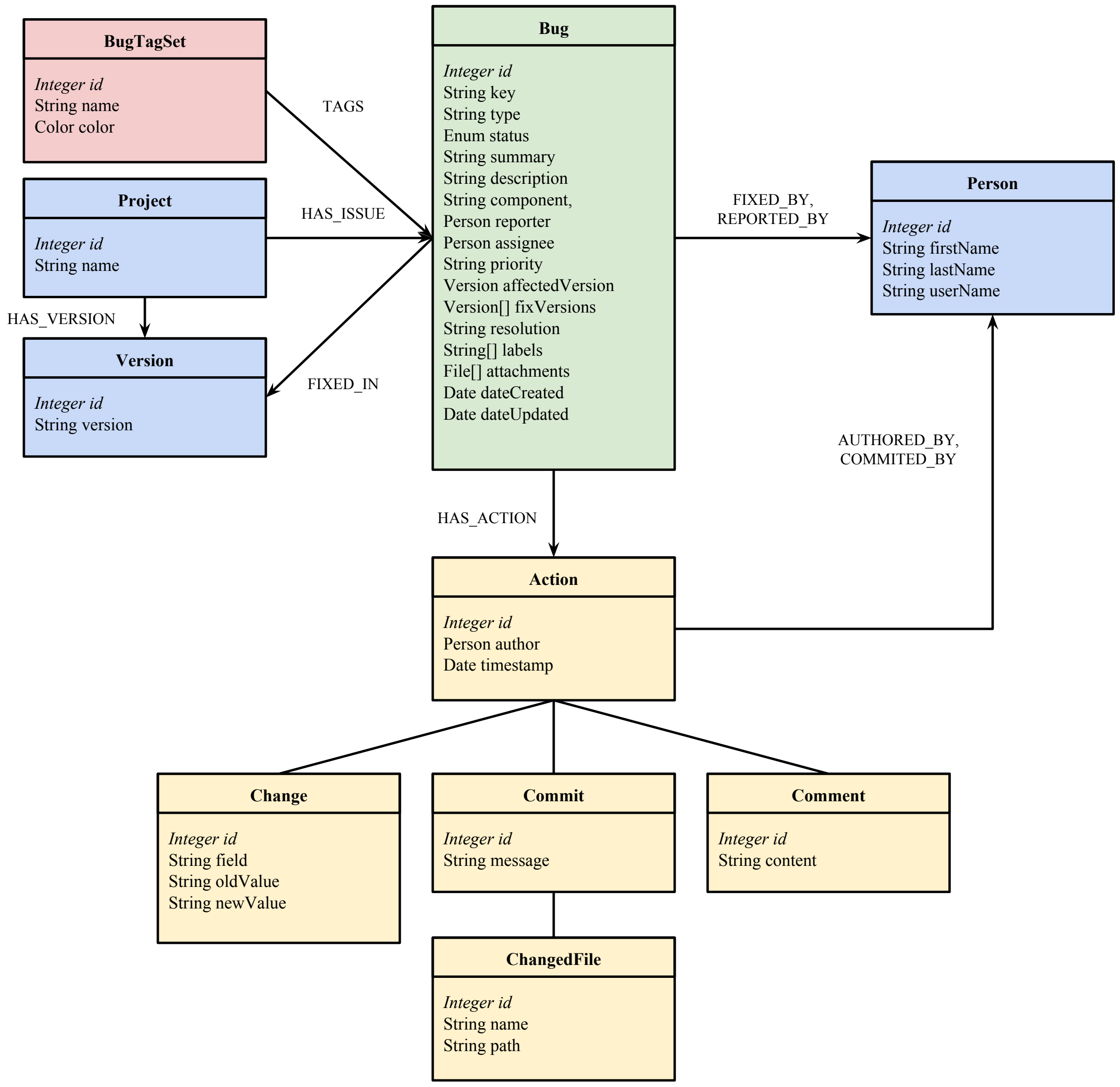


Figure 5.2: The PorchLight bug data model.

Field	Description
Identifier	The bug's unique identifier
Type	The type of bug reported (Problem, Feature, Improvement, etc.)
Summary	A brief text summary of the bug
Description	A more detailed text description of the bug
Component	The component or module which the bug affects
Reporter	The user that reported the bug
Assignee	The developer to which the bug has been assigned
Priority	The priority assigned by the developer
Affected Versions	The versions of the software that this bug effects
Fix Versions	The versions of the software in which the bug will be (or was) fixed
Resolution	The final resolution of the bug (Fixed, Can't Reproduce, etc.)
Comments	Additional information added by users and/or developers
Labels	Labels/tags assigned to the bug
Attachments	Attachments such as debug log files or screenshots.
Date Created	A timestamp of when the bug report was created
Date Updated	A timestamp of when the bug report was last modified

Table 5.3: Fields selected for the bug data model.

Since the transition between any two statuses, or the modification of any field, can be considered an action, the number of possible actions to capture is large. We have chosen to capture the actions (Table 5.4) that are most relevant to bug triaging, based on on existing research, in our model. These are actions that are performed on the bug and that can be used to make a triaging decision. For example, we know that frequent reassignment of a bug is a pattern that could indicate that there is not enough information to resolve it. Because that information is useful for triaging, we are capturing the assignment action as part of our model so that it can be used to create tag sets.

5.1.4 Metadata

Because users and milestones play an important role in the PorchLight user interface, these elements are captured in the data model as individual entities that are associated with

Action	**Description**
Developer Assignment	The bug has been assigned, or reassigned, to a developer
Fix Version Assignment	The bug has been assigned to a fix version
Comment	A comment has been added to the bug
Priority Change	The priority of the bug has been changed
Status Change	The status of the bug has been changed (e.g., from Open to Resolved)
Field Change	The value of a field has been changed (e.g., the description has been updated
Commit	A commit log message has been associated with the bug

Table 5.4: Actions selected for the bug data model.

bugs through relationships that can be be traversed. For example, the fact that a bug that is assigned to a user is captured by the `HAS_ISSUE` relationship between the person and the bug. Similarly, the fact that a bug that is assigned to a milestone is captured by the `FIXED_IN` relationship between the version and the bug. As the data model is populated with data from the bug tracker, these relationships are created between bugs, users, actions, and tag sets.

Several fields have been modeled as distinct entities in the data model and not as attributes of the bug. For instance, the *Project* to which the bug belongs, and the *Version* in which it has been or is assigned to be fixed, are represented by models and associated with the bug using the `HAS_ISSUE` relationship. Similarly, the bug reporter, assignee, and any comment or commit author is modeled by the *Person* model and associated using the `REPORTED_BY`, `FIXED_BY`, `AUTHORED_BY`, and `COMMITED_BY` relationships, respectively. These specific fields were normalized for performance reasons, since as, we discuss later in this chapter, being able to quickly traverse through a large collection of bugs and identify groups based on attributes is key to our approach.

Finally, bug tag sets themselves are represented using the *BugTagSet* model which has its own attributes, specifically a name and color. The `TAGS` relationship captures the membership of a specific bug to a bug tag set. This membership relationship is created during the BTL statement evaluation process.

The selection of statuses, fields, actions, and metadata lead us to the complete bug data model shown in Figure 5.2.

5.2 Internal Architecture

PorchLight is implemented as a fat-client application using a client-server architecture. Conceptually, the architecture can be grouped into two distinct components: one providing functionality for storing bug reports and evaluating BTL expressions, and one providing the user interface we have described in Chapter 4. Figure 5.3 provides an overview of the system architecture. In the remainder of this chapter, we detail each of the components.

5.2.1 Client

The PorchLight user interface was written using the Java Swing widget toolkit. Swing was chosen because of the numerous third-party libraries and frameworks available, including the open source MigLayout [4] layout manager which made controlling the position of each component on the screen simpler. Early in the design process, we evaluated alternative languages and frameworks, including Adobe Flex, JavaFX, and web-based frameworks, but found Swing's maturity as a platform and support for native interaction, such as drag-and-drop, to be superior.

The user interface makes extensive use of built-in Swing components like buttons, lists, and tables. We also extended numerous components to achieve some of the functionality needed in PorchLight. For example, we implemented a new `ListCellRenderer` for the user and

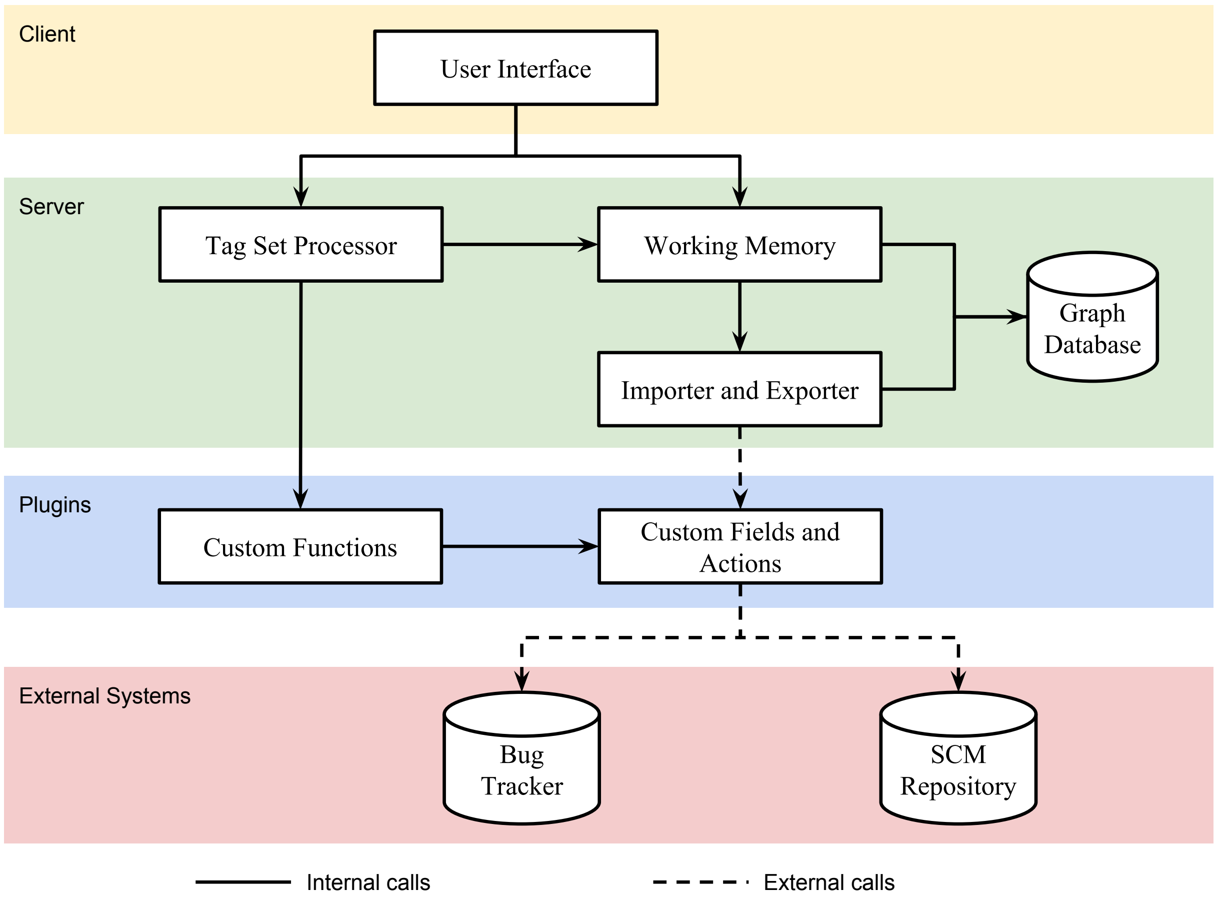


Figure 5.3: The PorchLight system architecture.

milestone lists to be able to display the icon, label, and bug count in a traditional list controller.

The client interfaces with the server for two functions: connecting to the working memory, and executing the tag set processor to create new tag sets. The client connects to the working memory using the `BugGraph` Java interface. In our case, the JIRA implementation of this interface was used to connect to the JIRA bug tracker. The interface provides methods for connecting to the working memory and retrieving tag sets a bugs. The client invokes the Tag Set Processor to create new tag sets in the working memory, a process which is described later in this section.

5.2.2 Server

Importer

In order to integrate with the numerous bug trackers to import bug reports and associated metadata, we designed an `Importer` interface (see Listing 5.1) for abstracting the methods needed to retrieve data from the source bug tracker, and an `Exporter` interface for saving changes back to the bug tracker.

This abstraction layer was implemented as a set of Java classes that encapsulate connectivity to the source bug tracker, and perform the mapping of bug reports and associated metadata (users, comments, source code commits, etc.) from the bug tracker's native schema to our data model. Though the interfaces allow for implementations for the different bug trackers, for practical purposes we implemented one for JIRA. Our implementation of the Importer interface makes extensive use of JIRA's REST API [14] for retrieving bug reports, and handles retrieving all of the bug reports, mapping the fields to the data model, and generating the actions associated with each bug.

The Importer interface also includes methods for retrieving commit data from a source code repository. This is used to associate commit information, such as the commit log message and files, to a bug. For our implementation, we built an importer the Subversion version control system.

Exporter

Actions performed through the PorchLight user interface, such as changing the assignee for a bug or creating a new tag set, are performed immediately in the working memory copy of the bug report. Our `Exporter` interface (see Listing 5.2) was designed to abstract the

```java
public interface Importer {
  /* Import all data from specified host into specified data directory (where
     graph is stored) */
  public void importFromServer(File dataDir, String host, String username,
     String password, String projectKey);

  /* Import all projects in the source bug tracker */
  public void importProject();

  /* Import all issues for the specified project */
  public void importBugs(long projectId);

  /* Import all versions associated with the project */
  public void importMilestones(long projectId);

  /* Import all labels (tags) for the specified bug */
  public void importLabels(long issueId);

  /* Import all comments for the specified issue */
  public void importComments(long issueId);

  /* Import the specified user */
  public void importUser(String userName);

  /* Download the user's avatar */
  public void downloadUserAvatar(String userName);
}
```

Listing 5.1: The `Importer` Java interface.

```
public interface Exporter {
  /* Apply all changes made (captured as DiffItems) to the source bug tracker
     */
  public void exportChangesToServer(List<DiffItem> diffItems, String host,
     String username, String password, String projectKey);

  /* Create a new comment associated with a bug */
  public void updateComments(DiffItem diff);

  /* Apply a label to a bug as a result of a new tag set */
  public void updateLabels(DiffItem diff);

  /* Assign a bug to a person */
  public void updateAssignee(DiffItem diff);

  /* Assign a bug to a milestone */
  public void updateMilestone(DiffItem diff);
}
```

Listing 5.2: The `Exporter` Java interface.

methods needed to apply the changes made in PorchLight to the source bug tracker. Our implementation of the Exporter interface for JIRA makes use of the same REST API to commit changes that have been made in the local graph database back to the source bug tracker. As triagers make assignments and changes through the PorchLight user interface, the changes are captured using the `DiffItem` class. When the user wants to commit the changes, the Exporter class is invoked to connect to the source bug tracker and apply the changes, which fall into one of the following types: add a comment, set the milestone, set the assignee, change the status (e.g., from Open to Resolved), or add a new tag (as a result of being added to a tag set).

Because the changes made to the bugs are first applied locally, we are able to present the user with the option to review the changes made before committing them to the remote bug tracker. Similar to committing changes after modifying source code, this feature allows triagers to review their work performed during a triaging session and commit the changes so that they are publicly reflected. Any conflicts that occur during the synchronization process, such as bugs that have been deleted, are presented to the user (see Section 5.2).

Working Memory

Because bug trackers can have tens of thousands of bug reports, it is unfeasible from a performance perspective to connect to the bug tracker server and request a bug report each time a tag set is created or modified. To address this limitation, we have designed a *working memory* into which the bug report data is downloaded for use through the user interface. The working memory is an interface to a local copy of the bug report data that has been retrieved from the the source bug tracker.

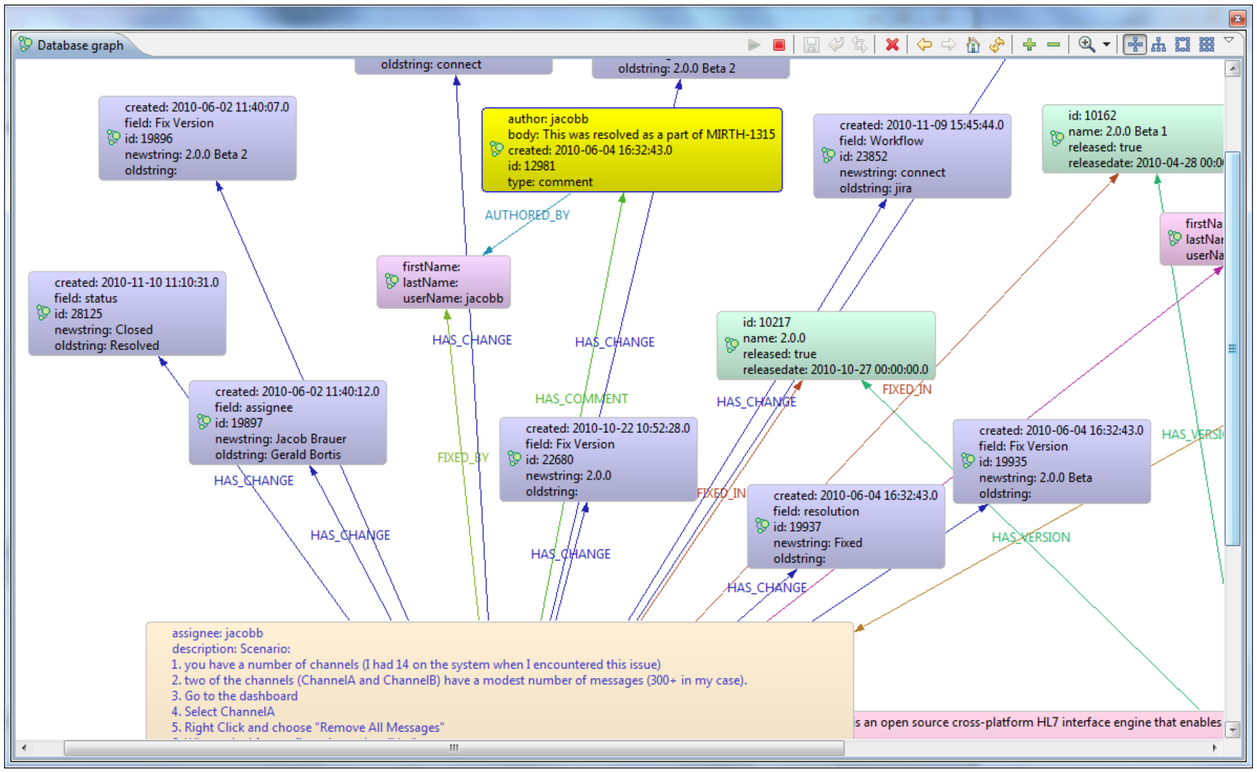
We have implemented the working memory for PorchLight using the neo4j [17] graph database engine. A *graph database* is an "online database management system with Create, Read, Update, and Delete (CRUD) methods that expose a graph data model" [44]. The primary distinction of a graph database compared to traditional relational databases is that "relationships are first-class citizens of the graph data model, unlike other database management systems, which require us to infer connections between entities" [44]. Specifically, neo4j implements the *property graph model*, which has the following characteristics [44]:

- It contains nodes and relationships
- Nodes contain properties (key-value pairs)
- Relationships are named and directed, and always have a start and end node
- Relationships can also contain properties

We chose to use a graph database for our implementation of the working memory in PorchLight so that we could capture the numerous relationships between the data elements that make up a bug report, and to make them readily available when creating tag sets.

The logic defined in the Importer maps bug reports and associated data, such as projects, versions, and people, into nodes and relationships in the graph. Each bug, user, comment, and source code commit is represented as a node in the graph. The attributes of each node are represented as name-value pairs. The relationships are then established between the nodes to allow for querying against the graph. The relationship types used to created edges between the nodes in the graph are listed in Figure 5.2.

This model allows us to quickly traverse the graph to perform searches like “find the bugs that Jacob has commented on.” Figure 5.4 shows a visual representation of the nodes (for example, the node representing the user `jacobb`) and relationship types (for example, the `HAS_CHANGE` relationship between a bug and an action) as created in the graph database.


Database graph

oldstring: connect

oldstring: 2.0.0 Beta 2

created: 2010-06-02 11:40:07.0
field: Fix Version
id: 19896
newstring: 2.0.0 Beta 2
oldstring:

author: jacobb
body: This was resolved as a part of MIRTH-1315
created: 2010-06-04 16:32:43.0
id: 12981
type: comment

created: 2010-11-09 15:45:44.0
field: Workflow
id: 23852
newstring: connect
oldstring: jira

id: 10162
name: 2.0.0 Beta 1
released: true
releasedate: 2010-04-28 00:0

firstNa
lastNa
userNa

created: 2010-11-10 11:10:31.0
field: status
id: 28125
newstring: Closed
oldstring: Resolved

firstName:
lastName:
userName: jacobb

id: 10217
name: 2.0.0
released: true
releasedate: 2010-10-27 00:00:00.0

created: 2010-06-02 11:40:12.0
field: assignee
id: 19897
newstring: Jacob Brauer
oldstring: Gerald Bortis

created: 2010-10-22 10:52:28.0
field: Fix Version
id: 22680
newstring: 2.0.0
oldstring:

created: 2010-06-04 16:32:43.0
field: Fix Version
id: 19935
newstring: 2.0.0 Beta
oldstring:

created: 2010-06-04 16:32:43.0
field: resolution
id: 19937
newstring: Fixed
oldstring:

assignee: jacobb
description: Scenario:
1. you have a number of channels (I had 14 on the system when I encountered this issue)
2. two of the channels (ChannelA and ChannelB) have a modest number of messages (300+ in my case).
3. Go to the dashboard
4. Select ChannelA
5. Right Click and choose "Remove All Messages"

s an open source cross-platform HL7 interface engine that enables

Figure 5.4: Visualization of graph representation of bugs in the neo4j database.

To improve performance during the initial loading of bug reports from the source bug tracker, we made extensive use of both the `BatchInserters` and `Index` Java classes in neo4j. `BatchInserters` allow for nodes and relationships to be added to graph in batches rather than individually, allowing for significantly faster initial loading of the bug reports. As each type of node is created in the graph, we also add the node's unique identifier to its associated index to ensure that queries were as fast as possible.

Tag Set Processor

The Tag Set Processor, or TSP, is the component that parses BTL statements and evaluates them against the working memory to create the specified tag sets. The TSP also provides the plugin mechanism for extending BTL with custom functions that can make use of custom fields. We first implemented a prototype of the TSP as a command-line program to validate our graph implementation as well as the BTL syntax. We then integrated our TSP implementation into the PorchLight client so that the BTL statements created through the user interface could be executed.

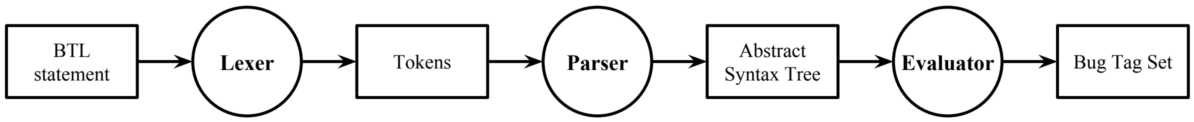


Figure 5.5: Tag Set Processor data flow.

One of the jobs of the TSP is the evaluation of BTL statements. The BTL statement is passed as input to the lexer which produces the tokens that represent the different parts of the statement. The lexer also returns errors if the BTL statement is incorrectly formed. The generated tokens are then passed as input to the parser which generates an *abstract syntax tree*, or AST, of the provided statement. The AST is then passed as input to the evaluator which traverses the AST and, at each branch of the tree, evaluates the part of the statement against the working memory to identify matching bugs. Once the bugs have been identified by traversing the graph model, a new tag set is created in the working memory and each bug within the tag set is tagged with the tag name.

The recognizer component of the TSP was implemented using ANTLR, a "parser generator you can use to implement language interpreters, compilers, and other translators" [42].

ANTLR is used to generate recognizers that apply grammatical structure to a stream of input symbols, which can be characters, tokens, or tree nodes [42]. We specified the grammar for BTL using the Extended Backus-Naur Form notation that ANTLR supports (see Appendix A for the complete BTL grammar). The following rule from the grammar demonstrates the highest-level definition of a BTL *statement*:

```
statement : 'TAG' (name=IDENT | name=STRING) 'WHERE' clause ';'? EOF -> ^(TAG
    $name clause);
```

5.3 Plugin Architecture

Developers can implement new plugins to expand the capabilities of PorchLight by augmenting the bug data model or the tagging language. There are three types of plugin points available: fields, actions, and functions.

Plugins are implemented as Java classes and must first be compiled and packaged as a JAR file before they can be used in PorchLight. The plugins that are in use by PorchLight are defined in the `plugins.json` file that resides in the application directory (example shown in Listing 5.3). To install a new plugin, a developer must add a new entry to the appropriate section of the file. The Java class file specified in the `plugins.json` file are then added to the Java virtual machine classpath when the application is started.

Fields and Actions

While we have defined our data model for capturing information from a variety of bug trackers, our use of a graph database to store the bugs report data means that we can add new properties dynamically. Developers can implement plugins that define the logic for importing new fields and creating new actions from a source bug tracker. The plugin

```
{
  "field-handlers": [{
    "jar": "custom-libs/time-estimate.jar",
    "class": "edu.uci.ics.sdcl.porchlight.plugins.field.EstimateFieldHandler",
    "enabled": true
  }],
  "functions": [{
    "jar": "custom-libs/commented.jar",
    "class": "edu.uci.ics.sdcl.porchlight.plugins.function.Commented",
    "enabled": true
  }]
}
```

Listing 5.3: An example of a `plugins.json` file.

architecture allows new attributes to be extracted and makes them available to the TSP so that they can be used in BTL statements. Custom fields and actions defined in the `plugins.json` file are invoked during the import process.

For example, suppose there is field in a bug tracker for keeping track of the time estimate provided by the assignee, specifying how long the work should take. While we do not import this field in our but model natively, it is relatively straightforward for a developer to implement a new plugin that can import this field as part of the bug report data into PorchLight. The `FieldHandler` interface is shown in Listing 5.4.

```
public interface FieldHandler {
  /* Map a field from the source bug tracker to the bug model */
  public void map(Map<String, Object> issueProperties, JSONObject bugJson)
      throws JSONException;
}
```

Listing 5.4: The `FieldHandler` Java interface.

During the import process, the Importer will iterate through the list of field handler plugins that are enabled and invoke the `map` method, which has access to the bug that is being imported. The method can be implemented to extract the field from any location within the bug's original schema. For example, the implementation of the custom field handler in

Listing 5.5 maps the `timeestimate` property from the bug tracker to a new `estimate` field.

```
public class EstimateFieldHandler implements FieldHandler {
    public void map(Map<String, Object> issueProperties, JSONObject bugJson)
        throws JSONException {
         issueProperties.put("estimate", bugJson.getString("timeestimate"));
    }
}
```

Listing 5.5: The `EstimateFieldHandler` plugin implementation.

Similarly, new actions can be added by implementing the `Action` interface.

```
public interface Action {
  /* Adds a new action indicating a change to a field to the bug in the graph
     defined by bugKey in the source bug tracker */
  public String addAction(String bugKey, String changedField, String
     oldValue, String newValue);
}
```

Listing 5.6: The `Action` Java interface.

Functions

Along with defining new fields and actions, developers can also implement plugins that provide new logic that can be invoked as a function during the evaluation of BTL statements. Functions are implemented using the `Function` interface, and the custom functions defined in the `plugins.json` file are invoked during the BTL statement evaluation process.

```
public interface Function {
  public String evaluate(BugGraph bugGraph, Bug bug, String param);
}
```

Listing 5.7: The `Function` Java interface.

```java
public class Commented implements Function {
    public String evaluate(BugGraph bugGraph, Bug bug, String param) {
        for (Comment comment : bugGraph.getCommentsForBug(bug.getKey())) {
            String[] authors = StringUtils.split(param);

            for (int i = 0; i < authors.length; i++) {
                if (StringUtils.equalsIgnoreCase(comment.getAuthor(),
                    authors[i])) {
                     return "true";
                }
            }
        }

        return "false";
    }
}
```

Listing 5.8: The COMMENTED function implementation.

For example, consider the relatively straightforward custom function, COMMENTED, implemented in Listing 5.8. The logic of the function is in the `evaluate` method, which has three parameters: the complete bug graph, the specific bug that is being evaluated by the function, and the parameter that was passed in to the custom function. In this example, the function will retrieve all of the comments for the specified bug and, for each, compare the author to the parameter, which in this case is a username. If there is a match, then it will return true, since the user has commented on that bug. Otherwise, the function will return false.

The TSP evaluates functions by first narrowing down the list of bugs to those that can be determined using the attributes that are captured in the bug graph, such as the assignee or status. Each bug in the resulting list is then evaluated by the functions invoked in the statement. After each function returns its result, the entire statement is evaluated.

5.4 Implementation Challenges

The main challenge in implementing PorchLight was ensuring that the architecture supported a responsive user interface, especially at a large scale with thousands of bug reports.

We wanted to both be able to present the user with a sufficient amount of information about a bug report, including historical actions, while allowing them to quickly navigate through the list, create tag sets, and to triage. We overcame this challenge by creating a local copy of the bug data in a graph database. This made both the import process sufficiently responsive, and allowed for complex relationships between nodes to be rapidly traversed during the evaluation of BTL statements.

Another challenge was in the design and implementation of the timeline. The current design is the result of multiple iterations, each of which addressed a new challenge in visualizing the richness of a bug's history in a way that can still be viewed at-a-glance. Features like the zigzag to eliminate unnecessary gaps between events and avoid overlapping markers, and filling the timeline background with colors based on the type of event, are the direct result of testing the initial visualizations with users.

Finally, developing a language that could be sufficiently expressive to create meaningful tag sets, that could be used through a user interface easily, and that could be extended to incorporate new functionality was an overarching challenge. Relying on standard techniques for developing new languages, and using tools like ANTLR, helped us to avoid common pitfalls. We also made extensive use of open source components, like neo4j, and their capabilities to focus our efforts on the problem of creating a tool that would support the outlined requirements rather than on the underlying technologies.

Chapter 6

Analysis and Findings

Now that we have presented PorchLight and its implementation, we return to the primary objective of this dissertation, which is to understand the needs of triagers and to assess our conjecture that working with tag sets better matches how they perform triaging. We performed a two-part study, using a different method for each part.

For the first part we conducted a *preliminary user study.* The goal of the study was to perform *concept testing* to assess how the functionality we had built into PorchLight would be perceived by the users. We wanted to conduct this study early in the development of the tool to identify any potentially major flaws in its design, specifically in areas like the user interface, that could prevent it from being used in a more comprehensive study. With the preliminary user study, we did not yet set out to assess the impact of the tool on the triaging process, or to identify how the tool might change the way triagers view the process altogether, even though we did learn some things in this regard.

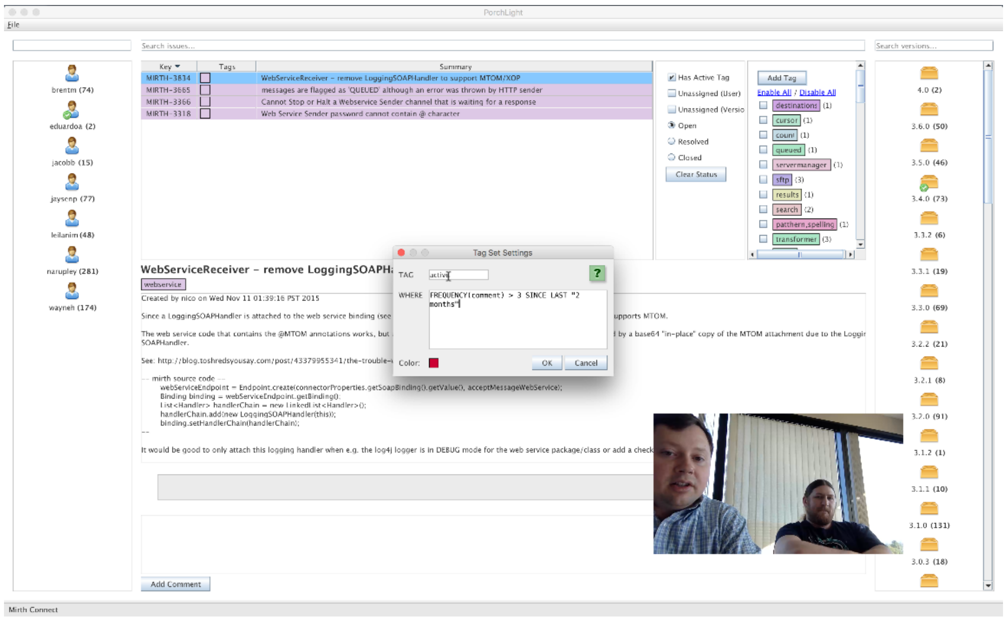
In the user study, we asked participants to perform a series of scripted tasks using bugs with which the participants had some prior familiarity. The tasks involved basic actions like performing assignments to users and milestones, and answering questions about a bug based on the information presented in the timeline. We concluded each session with a free-form interview to discuss what the participants did or did not like about the tool, and a ten question survey to assess its suitability for triaging.

For the second part, we employed a lightweight *participant observation study* method [35, 39], combining elements of a field study and a field deployment. For this part, we wanted to observe the impact our approach would have on how triagers approach their task. While we could not simply ask the participants to use the tool to do triaging *per se*, we wanted to situate the study in a real-world context. The goal of this part, then, is to assess the suitability of the tag set based approach, while at the same time learning *more* about triaging as we know it, now that they are provided with a more powerful way to do triaging.

To accomplish this, we recruited a group of software professionals from a software development organization. We provided each participant an initial training on the features of the tool, as well as an introduction to the concept of tag sets as an approach to triaging. We then engaged in an unstructured discussion about how the participants thought they could employ the approach. As questions were raised, we guided the participants to the functionality and ideas embodied in the tool, and collaborated with them on how they might achieve their desired result. We observed the participants and recorded their actions as they interacted with the tool and the bugs, while actively participating in the discussion.

We also made the decision to conduct two sessions with each set of participants. We used the first session to train them on the basic functionality of PorchLight, to introduce the concept of tag sets within the context of their project, and to begin the discussion through an initial exploration of their triaging needs and how PorchLight could help them, or not. We then used the interlude between the two sessions to create tag sets or plugins to better support some of the ideas that were brought up in the first session. Allowing a few days to pass between the sessions also gave the participants time to think about the approach and their needs. We used the new tag sets and features to seed the second session, and continued with the topic of how they could be used during triaging, which resulted in a more informative and rich discussion.

We note that, as a participant observation study, the researcher did assist the participants in working with the tool. We took on the roles of *facilitator*, by providing the technology, of *technical support* to help use the tool and build a positive relationship with the participants [35]. In some instances, the researcher acted as the *encourager* when there were features of the approach that he felt were appropriate for the situation.



Key	Tags	Summary
MIRTH-3834		WebServiceReceiver – remove LoggingSOAPHandler to support MTOM/XOP
MIRTH-3665		messages are flagged as 'QUEUED' although an error was thrown by HTTP sender
MIRTH-3366		Cannot Stop or Halt a Webservice Sender channel that is waiting for a response
MIRTH-3318		Web Service Sender password cannot contain @ character

TAG: active

WHERE: FREQUENCY(comment) > 3 SINCE LAST "2 months"

Color: 🟥

WebServiceReceiver – remove LoggingSOAPH

webservice

Created by nico on Wed Nov 11 01:39:16 PST 2015

Since a LoggingSOAPHandler is attached to the web service binding (see upports MTOM.

The web service code that contains the @MTOM annotations works, but by a base64 "in-place" copy of the MTOM attachment due to the Loggir SOAPHandler.

See: http://blog.toshredsyousay.com/post/43379955341/the-trouble-

```
-- mirth source code --
      webServiceEndpoint = Endpoint.create(connectorProperties.getSoapBinding().getValue(), acceptMessageWebService);
      Binding binding = webServiceEndpoint.getBinding();
      List<Handler> handlerChain = new LinkedList<Handler>();
      handlerChain.add(new LoggingSOAPHandler(this));
      binding.setHandlerChain(handlerChain);
--
```

It would be good to only attach this logging handler when e.g. the log4j logger is in DEBUG mode for the web service package/class or add a check

Figure 6.1: Researcher (left) and study participant (right) during participant observation session.

The engagement of the participants with the tool during both sessions varied. Some participants, especially during the first session, did not make use of all of the features of the tool, and in instances where they did have ideas for new tag sets, were hesitant to take control and create them personally. In those situations, we assisted by creating the tag sets alongside the users and presenting the results to them for feedback (see Figure 6.1). Once the participants were more familiar with the tool, and saw the results of their feedback from the first session realized in the second session, they became more comfortable and they took more control. In general, the primary use of the tool by the participants was with the basic features: browsing the bugs list, viewing comments, performing assignments, and so on. When it came to creating new tag sets, the researcher was more directly involved in writing the BTL based on their suggestions.

We chose the participant observation study method because it allowed us, as the researcher, to be part of the discussion, while at the same time allowing the participant to guide the discussion. Our primary goal in this part was to enable the participants to engage in exploring the tool and the concept of tag sets, and to discuss what the impact would be on their triaging activities. By being actively engaged with the participants, we mitigated the risk of them abandoning the tool out of frustration. The method also gave us the opportunity to have a dialog with them as they used the tool, giving us insights into the impact of the approach on their thought process.

6.1 Preliminary User Study

After several iterations of design and development of PorchLight, we conducted a preliminary user study to confirm the decisions we had made, specifically regarding the layout of the interface and the interactions that were available. We particularly wanted to ensure that these elements would not get in the triager's way of the participant observation study, which focuses much more broadly.

6.1.1 Setup and Procedure

For our preliminary user study, we recruited six professionals, who work at a nearby software development organization in the healthcare industry, and who perform triaging on a regular basis. With each participant, we conducted a 45–60 minute session. Before each session, we populated PorchLight with bug reports from three different active projects to which the participants regularly contribute to, either by reporting or resolving issues, or by assigning bugs to developers or milestones. The bug reports were imported directly from the organization's existing bug tracker.

We then provided each participant with a tutorial describing PorchLight. This included an overview of how to perform assignments using drag-and-drop between lists, how to view the activity timeline, how to use the quick search and filters, and how to add a comment or update the status of a bug. We also introduced the concept of tag sets, and provided a brief tutorial on how to create tag sets from sample BTL statements, or by using drag-and-drop from the user or milestone lists.

We then asked each of the participants to complete a series of triaging tasks using the bug reports from their project that we had identified ahead of time.

1. We first asked the participants to identify all bugs that had been assigned to a particular developer and version.
2. We then asked the participants to identify all bugs that had been commented on at least three times in the previous two months.
3. We then asked the participants to locate two open bugs, and assign one to a developer and another to a milestone, based on their knowledge of the bug's history and of the project.
4. Finally, we asked the participants to locate several bugs that we had identified as particularly interesting, and asked them to explain what could be determined based on the information available in the tool.

As a final task, we asked the participants if they could identify and, if possible, specify in BTL statements any tag sets that they thought could be useful when performing their own triaging.

After completing these tasks, we invited the participants to provide feedback regarding their past experiences with bug trackers, ways in which the process could be improved, as well as their impressions of PorchLight and its approach to bug triaging. We then asked them to complete a 10-question five-point (5—strongly agree, 1—strongly disagree) Likert scale questionnaire rating their experience using PorchLight and BTL for triaging tasks, as well as their openness to adopting PorchLight as their triaging tool.

6.1.2 Observations

Overall, the PorchLight interface was well received. In performing the tasks, virtually all of the features of PorchLight were used, including tag sets and BTL statements. In tasks that involved making assignments, the participants made use of the drag-and-drop assignment feature, and several commented on the ease with which assignment was possible without needing to modify search filters or lose one's context. All six participants agreed or strongly agreed that PorchLight made it easy to identify bugs that needed to be triaged (Question 1, average 4.3—see Table 6.1 for complete list), as well as to perform the actual triaging (Question 2, average 5.0). One participant commented:

> "[PorchLight] might make [our triaging meetings] a bit quicker because the interface is simpler. It's a lot easier to get around, you don't have to click too much, so I think it would be very helpful for pre-planning meetings and backlog reviews."

Most participants made extensive use of the activity timeline, noting the frequency with which some of the issues had transitioned from open to resolved, and the source code modifications that accompanied each status change. We received positive feedback on this ability to display multiple aspects of a bug report in a single view. One participant stated:

	Question	**Average**
1	I found it easy to identify the issues that needed to be triaged.	4.3
2	I found it easy to assign an issue to a developer and/or a release.	5.0
3	I found it easy to set issues aside for triage at a later time.	4.0
4	I found it easy to update the status of, or comment on, an issue.	4.5
5	I had sufficient feedback from the tool as I was making assignments.	4.2
6	I had sufficient information available about an issue to quickly make an assignment decision.	4.2
7	I had sufficient context of related issues to make an assignment decision.	3.8
8	I found tag sets to be useful in triaging issues.	4.5
9	PorchLight has some features or functionality that improves my issue triaging experience.	4.5
10	I would use PorchLight as my issue triaging tool.	4.3

Table 6.1: Preliminary user study questions and average responses.

> "I would honestly use [PorchLight]. With the timeline and the filters, it has the information I need. I would just have it open all the time."

Participants also commented that tag sets could could make their triaging more efficient, since their existing bug tracker forces them to continuously switch between creating and managing filters, viewing bug report details, and making an assignment. With tag sets, they were able to identify the bugs that needed to be triaged using BTL, merge the tag sets, and make the assignment in the same view. Several participants attempted to write their own BTL statements to generate new tag sets. One was unable to write the statement since the attribute he wanted to use, the time estimate for the bug, was not available (and was not included in the bug data model).

Others had difficulty recalling the syntax for BTL, and resorted to explaining what they intended to include in the statement (unsurprising, since it is unrealistic to expect them to master the full syntax of BTL in the relatively short time frame available). All six participants, however, agreed or strongly agreed that tag sets were useful for triaging (Question 8, average 4.5). One participant commented:

> "If you know how to use the tag sets, it's really powerful, because it lets you do anything. And you can also use the [BTL statements] ... it has a steep learning curve, but other than that it gives you every option possible."

We were encouraged by the feedback received during the open ended discussion, as several participants put forth examples of tag sets they would use were they available, including:

- Bugs that have been reported by customers within the last few weeks (as opposed to by the open source community).
- Bugs that have been recently commented on by a project manager or boss.
- Bugs that have been in the sprint backlog for longer than a few months.
- Bugs that have no time estimate set on them.

Furthermore, all of the participants either agreed or strongly agreed that PorchLight had functionality that would improve their triaging experience compared to their current bug tracker (Question 9, average 4.5), and they would use PorchLight as their bug triaging tool if it were made available (Question 10, average 4.3). One participant commented:

> "[PorchLight] could definitely replace [our bug tracker], because you can just sort it by backlog. We can even try using it for one of our sprints and see how it goes, because it has all the information, you don't have to go to a different page and it doesn't screw up by hitting the back button."

While this preliminarily user study was not comprehensive, it did confirm that we were headed in the right direction with the functionality that we were building into PorchLight.

The feedback we received pointed to the ability to view the information necessary for triage in one interface, the ability to quickly browse through a list of bugs, and the ability to quick perform an assignment as key features of the tool. Encouraged by these findings, we continued down the path of making improvements to these functions, and also incorporating more advanced tag set capabilities.

6.2 Participant Observation Study

While the preliminary user study provided us with positive feedback and confirmed some of the design decisions we made, we wanted to take a step back and look at our original conjecture and study the behaviors that the triagers exhibited when working with bugs in sets. Specifically, we wanted to understand how tag sets could help triagers work with sets of bugs, and what the impact of a dedicated triaging environment and tagging language built around tag sets would be.

For this study, we asked the participants to think of PorchLight as a *research prototype* designed to express certain ideas, and to consider the *concept* of tag sets and the ability to create and work with tag sets rather than the specific features of the tool or the tagging language. This subtle but important modification to the study prompt made a significant difference to the type of feedback that was received.

The participant observation study included four participants, all employees of the same software development organization involved in the preliminary user study. Two of the individuals had participated in both the preliminary user study and the participant observation. The organization consists of multiple development teams working on different products. The organization uses an Agile development methodology that involves sprint planning meetings every 2-3 weeks.

6.2.1 Setup and Procedure

In preparation for each of the user study sessions, we configured PorchLight to work against the bug tracker for the projects, in this case JIRA. We imported several thousand of the most recent bug reports into the tool and verified the collected data (bug reports, comments, milestones, users) were present and accurate. The specific projects that were imported depended on the participants the projects they were most closely involved with at the time.

Project	Age	Num. Issues	Num. Issues Imported
Project A	10 yrs	3,700	3,000
Project B	5 yrs	3,500	3,000
Project C	9 yrs	8,100	3,000

Table 6.2: Summary of projects imported into PorchLight for the study.

During each session, we recorded the participants activity using a screen capture tool. This included a recording of the actions performed while using the tool, as well as the audio of the dialog between the participants and the researcher, along with a video of the person currently "driving" the tool. We have included screen captures in the narrative from the sessions that contained information that is publicly available, but not of the other sessions which contain information that is proprietary in nature.

For this user study we conducted two sessions with each set of participants. Each session was approximately 1 hour long, and the two sessions were conducted several days apart. We chose to conduct two sessions for several reasons. First, we wanted to expose the participants to the tool on multiple occasions and to give them a chance to become familiar with the concepts and the functionality before diving in to the bug reports. The goal was to take into account any learning curve associated with tag sets. Second, we wanted the opportunity to analyze observations on how tag sets were used from the first session in order to prepare the environment during the second session. We conducted the first pair of sessions with two participants together as a team since they shared triaging responsibilities for their project, and we conducted the other two pairs of sessions with the participants individually.

Similar to the preliminary user study, we began the first session by providing each participant with a primer on PorchLight (see Appendix B for the complete study prompt). The purpose of this part of the user study was to familiarize the participants with the features, including the layout of the user interface and the assignment actions. Participants were provided with an outline of the tutorial, and a BTL reference sheet, during the entire session.

After the tutorial, we directed the participants to explore the bugs that had been preloaded from their project. This was a free-form exercise, and the only prompt was to browse through the bugs, optionally making use of the tag set functionality. The participants were encouraged to "think aloud" and describe any observations. Based on these comments, we asked the participants questions about what their typical triaging approach is, and if applicable, provided instructions on how to use the tool to answer a specific question.

After each initial session, we reviewed the recordings and identified the instances where the participants attempted to create a tag set to express a group of bugs that were of interest to them. If they were unable to create this tag set, we noted the reason and addressed it in one of two ways. If the request was for a particular tag set that could not be easily created during the session, we created this tag set so that it would be readily available during the second session. If the request was for a particular field, action, or function that was not available in BTL, we implemented it as a new plug-in that could then be used.

After creating new tag sets or modifying BTL based on the feedback from the first session, we conducted a second session. The prompt for this session was to explore the same bug reports drawn from their project, but they had several tag sets available that they could use as a starting point. The participants were encouraged to review the bug reports that were identified by these tag sets, and to triage them if appropriate.

Each session was highly *interactive*, and we paused at various points during the discussion to recap observations that we had made about their specific triaging workflow or model, or comments that the participants had made. Often this would prompt the participant to elaborate on how they would perform some action using their current bug tracker, and predict how they might do it differently if they were using PorchLight or tag sets. If we felt that a specific feature or tag set would address some need that the participant posed during the session, we provided suggestions and guidance on how to use the tool to address the need.

6.2.2 Summary of Participant Observation Sessions

In this section we provide a narrative overview of each of the sessions. For each of the participants, we present the concepts that were discussed and the actions that were performed wile using PorchLight, as well as the changes that were made between sessions. More specifically, we highlight the needs presented by the participants and the tag sets that were created in response during each session.

Development Co-Leads

The first participants are both senior-level software engineers that shared the role of co-lead of a commercially developed open source product. They oversee a team of three other software engineers, and share the responsibility of planning releases for the product and guiding the technical direction. One of their responsibilities as co-leads for the product is to perform regular triage of new bugs that are opened in the project's bug tracker. All bugs for the project are tracked in JIRA, and there have been over 3,700 change requests created since the team began using the bug tracker in 2006 (of which we have loaded the 3,000 most recent into PorchLight).

We note that the sessions were conducted with both of the participants together as a team. This was done intentionally, since both were intimately familiar with the bugs in the bug tracker, and could complement each other in the context and knowledge they brought to bear during the sessions. There were several points during the sessions where one participant confirmed, or added to, the suggestions and ideas proposed by the other, which led to refinements in many of our observations.

Session 1

The first session lasted 1 hour and 4 minutes. The overview and tutorial portion lasted 21 minutes. The remainder of the time was a discussion on tag sets and exploration of the bug reports. We asked the participants to act as if they were conducting a "hypothetical triaging session" and to explore the bug reports.

During the tutorial part of the session, we created a new tag set to demonstrate the capabilities of BTL. In the example tag set we made use of the `FREQUENCY` function to identify bugs that have received more than three comments and that have been updated in the last two months. The resulting tag set contained 179 bug reports from the sample that was imported.

```
TAG "ACTIVE" WHERE FREQUENCY(comment) > 3 SINCE LAST "2 months"
```

Pushing Bugs Out

They began by wanting to look at all of the bug reports currently assigned to the next release to decide the ones that need to be "pushed out." This involved reviewing the ones that were not likely to be resolved due to time constraints. To find the bugs assigned to 3.4.0, we demonstrated the ability to drag the milestone from the milestone list onto the tag set list, which created a new tag set using the following BTL expression:

```
TAG "3.4.0" WHERE fixVersion = "3.4.0"
```

The tag set contained 73 bugs. The participants reviewed each one to make sure that it was correctly assigned to the 3.4.0 release. Since bugs related to the theme of the release are typically assigned to one developer, they found the ability to multi-select bugs from the bug list and assign them to a user an efficient way of performing the triaging action. One of the participants began reviewing the bugs in the bug list, while referring to the the timeline to both assess the level of activity on a bug and to review the comments.

Themes and Priorities

Both of the participants commented that they plan releases and triage bugs based on themes. They typically identify a handful of major features that will be part of the release, and everything else can be moved out to a subsequent release during the triaging sessions. For the 3.4.0 release, the major theme is improvements to the Web Services Connector component and any related changes. The participants identified this set of bugs as a potential starting point for triaging. To identify these bugs, we suggested using the CONTAINS function which searches the summary and description text for matching keywords.

```
TAG "WS" WHERE STATUS = "Open" AND CONTAINS("web service") = true OR
   CONTAINS("soap") = true OR CONTAINS("http") = true
```

We attempted to create this tag set during the session, but the BTL evaluation failed and the tag set did not contain any bugs. We noticed errors in the console output and made a note to address them before the second session.

One of the participants also commented that the concerns themselves are prioritized as part of triaging. There are some concerns that are specific to a release, while others apply to all releases and are more fundamental. For example, bugs related to security defects that may describe some vulnerability are the highest priority bugs that should be assigned and resolved in a timely manner, and having a tag set that identifies these, based on keyword or another property, would be a starting point for planning releases.

Because these bugs may not always contain easily identifying keywords that indicate a security related defect, one of the participants commented that he would initially review new defects and manually tag them. To demonstrate this functionality, we scanned the bug list and identified several bugs that could be related to security and created a new ad hoc tag set by transferring them onto the tag set list. This new static set could then be used to prioritize bugs.

The next level of priority would be bugs that indicate a problem with the foundational components of the software. A bug that potentially prevents the processing of messages, or the software from launching, would be one that should also be triaged and assigned. The last set of bugs are the trivial ones: defects related to the user interface or ones that do not prevent the software from being used. These defects are more likely to be distributed across releases and pushed out during a triaging session.

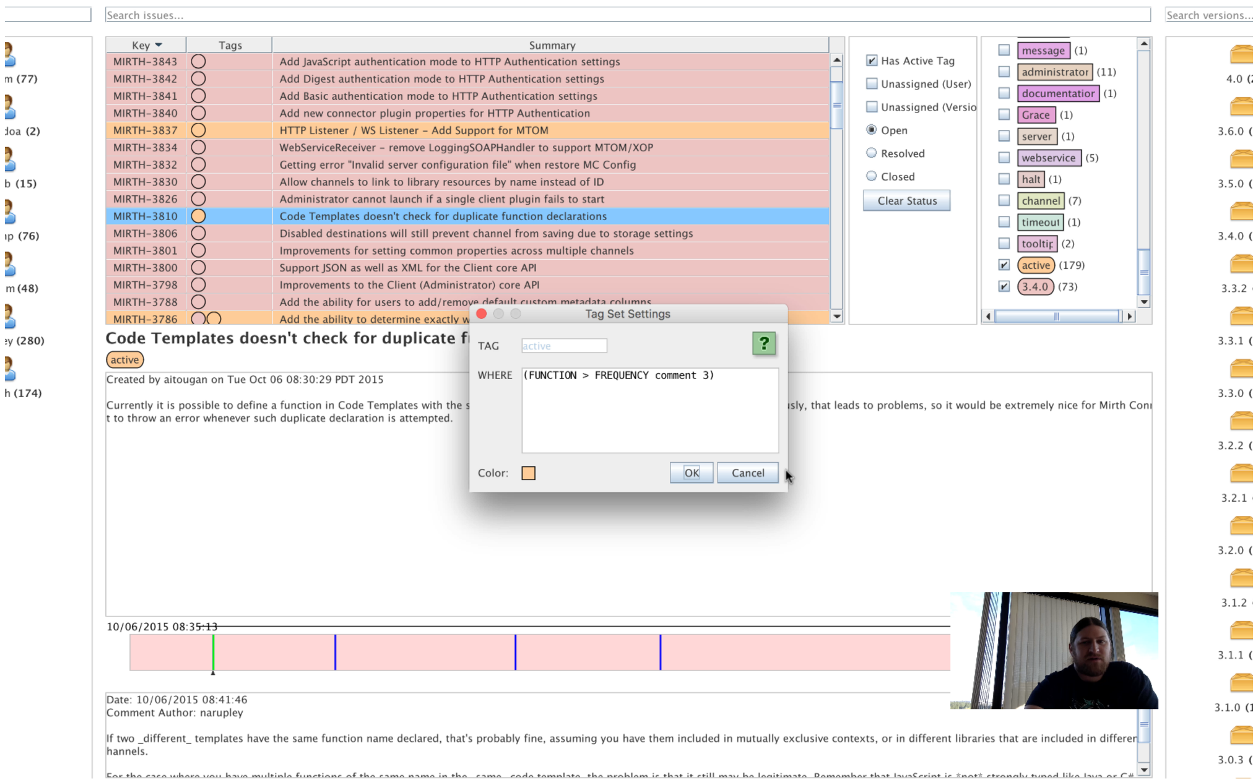
Votes

In situations where there are too many bugs identified in the tag set to assign to a single milestone, one of the participants requested the ability to create an additional tag set based on the number of votes associated with each bug. Since their project is open source and many bug reports come from the community, votes are a way for users to provide input on what should be prioritized for each release. The participant requested the ability to identify the bugs that have received eight or more votes. Since votes was not an action that was implemented in the data model, we were not able to create a tag set based on votes at this time.

Overlaying Tag Sets

We also explored the ability to have multiple tag sets active in the bug list. We did this by enabling both the *3.4.0* tag set and the *ACTIVE* tag set. The bug list then contained bugs that were assigned to the 3.4.0 release or that were considered active using our tag set (have more than three comments and have been updated in the last two months).

One of the participants suggested that these two tag sets could serve as a starting point for a triaging session, and on top of this they would want to create additional *one-off* tag sets based on search criteria, similar to the *WS* tag set created earlier in the session. The ability to enable and disable the tag sets would allow him to easily identify bugs that are members of multiple tag sets, and thus match the criteria specified by each of the tag sets.



Key	Tags	Summary
MIRTH-3843	○	Add JavaScript authentication mode to HTTP Authentication settings
MIRTH-3842	○	Add Digest authentication mode to HTTP Authentication settings
MIRTH-3841	○	Add Basic authentication mode to HTTP Authentication settings
MIRTH-3840	○	Add new connector plugin properties for HTTP Authentication
MIRTH-3837	○	HTTP Listener / WS Listener – Add Support for MTOM
MIRTH-3834	○	WebServiceReceiver – remove LoggingSOAPHandler to support MTOM/XOP
MIRTH-3832	○	Getting error "Invalid server configuration file" when restore MC Config
MIRTH-3830	○	Allow channels to link to library resources by name instead of ID
MIRTH-3826	○	Administrator cannot launch if a single client plugin fails to start
MIRTH-3810	●	Code Templates doesn't check for duplicate function declarations
MIRTH-3806	○	Disabled destinations will still prevent channel from saving due to storage settings
MIRTH-3801	○	Improvements for setting common properties across multiple channels
MIRTH-3800	○	Support JSON as well as XML for the Client core API
MIRTH-3798	○	Improvements to the Client (Administrator) core API
MIRTH-3788	○	Add the ability for users to add/remove default custom metadata columns
MIRTH-3786	○○	Add the ability to determine exactly w

[✓] Has Active Tag
[ ] Unassigned (User)
[ ] Unassigned (Versio
[● Open] ○ Resolved ○ Closed
Clear Status

[ ]	message	(1)
[ ]	administrator	(11)
[ ]	documentatior	(1)
[ ]	Grace	(1)
[ ]	server	(1)
[ ]	webservice	(5)
[ ]	halt	(1)
[ ]	channel	(7)
[ ]	timeout	(1)
[ ]	tooltip	(2)
[✓]	active	(179)
[✓]	3.4.0	(73)

Code Templates doesn't check for duplicate f

active

Created by aitougan on Tue Oct 06 08:30:29 PDT 2015

Currently it is possible to define a function in Code Templates with the s … usly, that leads to problems, so it would be extremely nice for Mirth Conn
t to throw an error whenever such duplicate declaration is attempted.

Tag Set Settings

TAG: active

WHERE:
```
(FUNCTION > FREQUENCY comment 3)
```

Color: 🟧

OK   Cancel

10/06/2015 08:35:13

Date: 10/06/2015 08:41:46
Comment Author: narupley

If two _different_ templates have the same function name declared, that's probably fine, assuming you have them included in mutually exclusive contexts, or in different libraries that are included in differen
hannels.

Figure 6.2: Study participant verifying tag set syntax.

Channels and Patches

We also discussed the ability to create tag sets based on the presence of specific items in the bug, like screenshots or stack traces. One of the participants did not think those two would be particularly useful, but suggested that being able to identify bugs that had *specific* attachments would be helpful. For example, a *channel* is an artifact that is created using the editing environment developed as part of the project. If an XML file defining the channel is attached to the bug, it means that there is additional information that could be used to debug the defect, increasing the likelihood that it can be resolved. Source code patches were also identified as important attachments since they indicate that the reporter has not only opened a bug report but has included a fix. A tag set with bugs that contain channels or patches can be used as another layer that would indicate bugs that can be assigned to an approaching release.

Who Commented?

We also discussed additional attributes that may be useful in creating tag sets. Earlier in the session we created a tag set to identify bugs that had received more than three comments. We provided the participants with additional examples of fields and functions available in BTL, which prompted one of them to remark that identifying bugs based on *who had commented on them* would be useful. Users from the open source community will often leave comments on bugs with additional information, or request that a particular bug be prioritized for a future release. Because the software is also available under a commercial license, customers will also leave comments on bugs requesting additional information or changes.

One participant requested the ability to create a tag set that contained bugs that were open and that had been commented on by a specific individual, either from the open source community or that was using the commercially licensed version of the software. They would use this set of bugs to identify ones that may need additional attention during the triaging process. We were unable to create this tag set since BTL at the time did not have the ability to tag bugs based on the author of a comment on a bug.

Bug Inbox

One of the participants commented that one attribute that would be useful for triaging is when the bug was last reviewed by one of the triagers.

> "One thing I've been playing around with is, not just necessarily time it was last modified or created, but when is the last time I looked at this or reviewed it? Maybe I didn't make any changes, but I want to know 'did I look at this issue?' because maybe I looked at this issue last week and I forgot. It was fine, I didn't need to make any changes and it was exactly where it needed to be. But, if there was a way, maybe just another field or something, like *last reviewed date*. Then it turns into an inbox type of thing: here's all the issues that I haven't look at at all, or ones that I haven't looked at in 6 months, and that determines its age."

Similarly, while reviewing the bug list generated from overlaying multiple tag sets, the other participant commented that it would be useful to have the ability to *mark a bug as viewed*. This would allow the triager to note that it has been reviewed, even if it has not been modified or assigned. Additionally, the timestamp for when it was last reviewed could be used in future triaging sessions to identify the ones that have not been reviewed in some time so they can be triaged.

We also asked the participants to describe what an ideal "starting tag set" would look like to prepare a "bug inbox" for a triaging session. Combining the tag sets discussed earlier in the session, the participants agreed that a tag set that identified open bugs assigned to the 3.4.0 release that contain one of the theme keywords and had more than five votes from the community would be a recurring list they would want to review for release planning. The desired tag set is described below:

```
TAG "STARTING" WHERE STATUS = "Open" AND CONTAINS("web service") = true OR
   CONTAINS("soap") = true OR CONTAINS("rest") = true AND FREQUENCY(votes) >
   5
```

Changes Made

Based on the feedback provided during the first session, we made several changes to the tool in preparation for the second session. First, we implemented a new function called COMMENTED which allows users to tag bugs which have comments authored by a specified user. The implementation of this plugin is listed in Listing 5.8 as an example of a custom function.

Second, we implemented a new field handler to track the number of votes a bug has. This field was available in JIRA as an endpoint in the REST API, and we extended the bug data model using the plugin architecture and added a new field called *votes* that would be populated during the import process. This field could then be used in conjunction with the FREQUENCY function to tag bugs that had received a certain number of comments.

Lastly, we addressed a defect in the CONTAINS function that had prevented us from creating the tag set that identified bugs based on a set of keywords. The error was caused when the function was evaluated against bugs that had an empty description, which was the case for several in the bug tracker. The fix for this defect involved simple checking for an empty string and not performing the regular expression search in that case.

Session 2

The second session was conducted with the same two participants five days after the first session and lasted 53 minutes. We began this session with a quick review of what we had discussed in the first session, and the new features that had been added to address some of the feedback.

Community Votes

To demonstrated the changes that were made from the previous session, we created a new tag set using the *Votes* field to identify bugs, in the same project that was imported before, that had received more than one vote:

```
TAG "VOTES" WHERE FREQUENCY(votes) > 1
```

The resulting tag set contained 36 bugs. Upon review, one of the participants felt that this tag set was a useful starting point for triaging, stating, "I think this is a small enough list to begin [triaging] with." In addition to this tag set, the participant wanted to refine the criteria and tag the bugs which had received more than four votes. Using the built-in autocomplete functionality in the BTL editor (see Figure 6.3), the participant proceeded to create a new tag set using the following BTL expression:

```
TAG "VOTES2" WHERE FREQUENCY(votes) > 4
```

The resulting tag set contained only 8 bugs. With both tag sets active, the list contained bugs that had received more than one vote, or that had received more than four votes. This was indicated by the two markers in the tag set column of the bug list. The participant that had created the tag set observed that since one of the tag sets is a *subset* of the other, it would be useful to be able to sort the bug list in descending order based on the number of tag sets the bug belongs to. This would put the bugs with more than four votes at the top of the list and allow him to review and triage them quickly:

> "I would see these [bugs] as higher priority, or just something to look at first. If I'm doing to decide to put something into a version, I'm more likely to put these issues here rather than something else that a couple of people have voted for but not as many as these."

At multiple points during the session, the participants pointed out that the assignment indicators (green check marks) in the user and milestone list, meant to highlight the currently assigned developer or milestone, were not prominent enough and were difficult to see when quickly scrolling through the bug list. They requested that the indicators be more prominent, or that the user and milestone items in the list be highlighted to indicate assignment.

Using `CONTAINS`

One of the participants then created a new tag set using the fixed CONTAINS function to identify bugs in one of the recent releases that were related to the Web Service Connector.

```
TAG "31WS" WHERE fixVersion = "3.1.0" AND CONTAINS("web service") = true OR
   CONTAINS("soap") = true OR CONTAINS("rest") = true
```

This tag set contained 17 bugs. He then enabled the existing *VOTES2* tag set (which tagged bugs that had more than 4 votes) and found that none of the bugs in the bug list were tagged with *both* tags. This indicated that, while these bugs had matched the theme of the release, the community did not view them as particularly important (based on votes as rough indicator).

Tag Sets for Specific Users

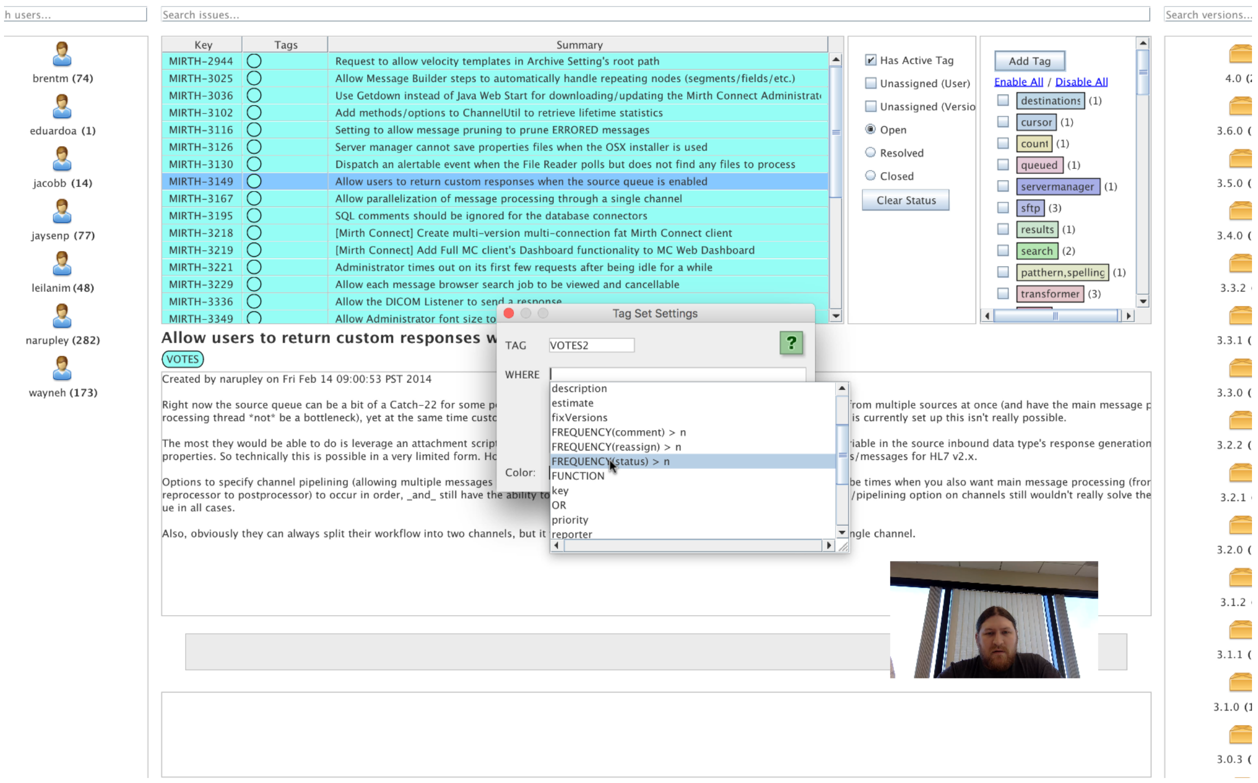
We then proceeded to create a new tag set that demonstrated the new COMMENTED function to identify bugs which had been commented on by particular users in the community. Prior to this session, the participants had provided a list of 15 usernames of users in the community that were relevant to their triaging. We created the following tag set:

```
TAG "COMMENTED" WHERE COMMENTED(user1) = true OR COMMENTED(user2) = true ...
   OR COMMENTED(user15) = true
```

The resulting tag set contained 23 bugs. Another request that came up during the session was the ability to identify bugs that were reported by a specific user. We demonstrated this ability by creating a new tag set using the following BTL statement:

```
TAG "USER" WHERE reporter = "user"
```

The tag set contained 21 bugs. Enabling both of these tag sets presented a bug list which contained bugs that had been commented on by a select list from the community, or had been reported by a specific user. In general, the participants found the ability to create tag sets based on the users involved, whether commenters or reporters, to be useful.



Key	Tags	Summary
MIRTH-2944	○	Request to allow velocity templates in Archive Setting's root path
MIRTH-3025	○	Allow Message Builder steps to automatically handle repeating nodes (segments/fields/etc.)
MIRTH-3036	○	Use Getdown instead of Java Web Start for downloading/updating the Mirth Connect Administrat
MIRTH-3102	○	Add methods/options to ChannelUtil to retrieve lifetime statistics
MIRTH-3116	○	Setting to allow message pruning to prune ERRORED messages
MIRTH-3126	○	Server manager cannot save properties files when the OSX installer is used
MIRTH-3130	○	Dispatch an alertable event when the File Reader polls but does not find any files to process
MIRTH-3149	○	Allow users to return custom responses when the source queue is enabled
MIRTH-3167	○	Allow parallelization of message processing through a single channel
MIRTH-3195	○	SQL comments should be ignored for the database connectors
MIRTH-3218	○	[Mirth Connect] Create multi-version multi-connection fat Mirth Connect client
MIRTH-3219	○	[Mirth Connect] Add Full MC client's Dashboard functionality to MC Web Dashboard
MIRTH-3221	○	Administrator times out on its first few requests after being idle for a while
MIRTH-3229	○	Allow each message browser search job to be viewed and cancellable
MIRTH-3336	○	Allow the DICOM Listener to send a response
MIRTH-3349	○	Allow Administrator font size to

Allow users to return custom responses w

VOTES

TAG VOTES2

Created by narupley on Fri Feb 14 09:00:53 PST 2014

Right now the source queue can be a bit of a Catch-22 for some p rom multiple sources at once (and have the main message p
rocessing thread *not* be a bottleneck), yet at the same time custo is currently set up this isn't really possible.

The most they would be able to do is leverage an attachment script iable in the source inbound data type's response generation
properties. So technically this is possible in a very limited form. Ho s/messages for HL7 v2.x.

Options to specify channel pipelining (allowing multiple messages be times when you also want main message processing (fror
reprocessor to postprocessor) to occur in order, _and_ still have the ability to /pipelining option on channels still wouldn't really solve the
ue in all cases.

Also, obviously they can always split their workflow into two channels, but it ngle channel.

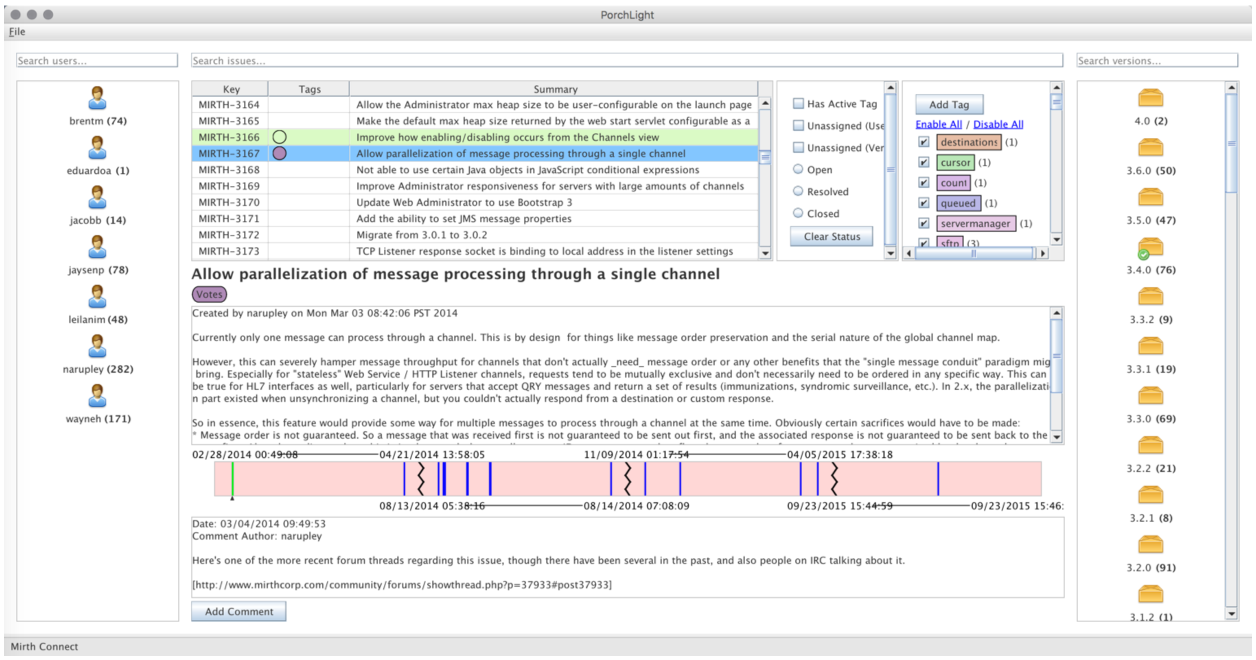
Figure 6.3: Study participant creating new tag set.

Timeline and Importance

While reviewing the bugs that were in the *COMMENTED* tag set, one of the participants began to review the individual comments for bugs in the list using the timeline. The discussion around comments and activity led to the notion that the number of comments correlates to the importance of the issue, or the demand for an issue. As we looked at one bug in particular (MIRTH-3167 shown in Figure 6.4), one participant observed that, "this is why he [one of the developers] is working on this one right now."

We then asked him to also enable the *VOTES* tag set, and we saw that the particular bug was tagged with both *COMMENTED* and *VOTES*. The number of comments and the number of votes present in the bug tracker seemed to be related to the importance of the issue and the amount of development effort that was currently going into it.

Layering Intersecting Tag Sets



Key	Tags	Summary
MIRTH-3164		Allow the Administrator max heap size to be user-configurable on the launch page
MIRTH-3165		Make the default max heap size returned by the web start servlet configurable as a
MIRTH-3166	○	Improve how enabling/disabling occurs from the Channels view
MIRTH-3167	●	Allow parallelization of message processing through a single channel
MIRTH-3168		Not able to use certain Java objects in JavaScript conditional expressions
MIRTH-3169		Improve Administrator responsiveness for servers with large amounts of channels
MIRTH-3170		Update Web Administrator to use Bootstrap 3
MIRTH-3171		Add the ability to set JMS message properties
MIRTH-3172		Migrate from 3.0.1 to 3.0.2
MIRTH-3173		TCP Listener response socket is binding to local address in the listener settings

Allow parallelization of message processing through a single channel

Created by narupley on Mon Mar 03 08:42:06 PST 2014

Currently only one message can process through a channel. This is by design for things like message order preservation and the serial nature of the global channel map.

However, this can severely hamper message throughput for channels that don't actually _need_ message order or any other benefits that the "single message conduit" paradigm mig
 bring. Especially for "stateless" Web Service / HTTP Listener channels, requests tend to be mutually exclusive and don't necessarily need to be ordered in any specific way. This can be true for HL7 interfaces as well, particularly for servers that accept QRY messages and return a set of results (immunizations, syndromic surveillance, etc.). In 2.x, the parallelizati
n part existed when unsynchronizing a channel, but you couldn't actually respond from a destination or custom response.

So in essence, this feature would provide some way for multiple messages to process through a channel at the same time. Obviously certain sacrifices would have to be made:
* Message order is not guaranteed. So a message that was received first is not guaranteed to be sent out first, and the associated response is not guaranteed to be sent back to the

Date: 03/04/2014 09:49:53
Comment Author: narupley

Here's one of the more recent forum threads regarding this issue, though there have been several in the past, and also people on IRC talking about it.

[http://www.mirthcorp.com/community/forums/showthread.php?p=37933#post37933]

Figure 6.4: Bug details and activity timeline for MIRTH-3167.

One participant asked if it was possible to sequentially enable tag sets so that bugs that belonged to multiple tag sets could be revealed. In an effort to facilitate this, we demonstrated two features in the tool. First, we pointed out the tag set indicators that appear in the bug list when tag sets are enabled. If multiple tag sets are enabled, and the *Has Active Tag* filter is enabled, then the bugs in the list are those that belong to *any* of those tag sets.

Second, we demonstrated the ability to take a tag set label from the tag set list and drag-and-drop it onto another label. This prompted the participant to enter a new name for a tag set which combined the clauses from the two tag sets using the *and* operator, creating a new tag set which was the intersection of the two. While both of these achieved the desired outcome, the ability to more easily enable and disable individual tag sets and dynamically switch between viewing all of the bugs and ones that were only in certain tag combinations of tag sets was desired.

Release Planning

We entered the next part of the discussion by asking the question, if a base list is the starting point for triaging, what criteria do you use to create the tag set? This led to a discussion about the team's approach to release planning and the role that tag sets can play. Specifically, tag sets could be used at various checkpoints during the process, such as at the beginning of a development sprint or right before it ends, to ensure that bugs have been properly assigned. Once the triaging has been performed, the development team can continue using the existing bug tracker to view details about the individual bugs. This supports our assertion that triagers would be well served by having a *dedicated* triaging environment that meets their specific needs.

Product Manager

The second participant is a software product manager that oversees product direction for two commercial products. On one product is solely responsible for managing and triaging bug reports opened in the JIRA project. Part of this includes triaging new bugs as they are opened. To date, this product has over 3,500 change requests that have been created since the team began using the bug tracker in 2011 (of which, again, we loaded the most recent 3,000 into PorchLight). He also oversees a new product that is under development and has not yet been released. While he does not perform triage on bugs for this project yet, he is responsible for creating and managing the feature requests for the project, and distributing them across future milestones.

Session 1

The first session lasted 48 minutes. The overview and tutorial portion lasted 11 minutes. The tutorial was significantly shorter since the participant was already somewhat familiar with the features since he had also been a participant in the preliminary use study. We began the exploratory part of the session with the following tag set:

```
TAG "ACTIVE" WHERE FREQUENCY(comment) > 3
```

The tag set contained 700 bugs. While reviewing the bugs in this tag set, the participant suggested creating a new one that tagged bugs that were related to ZIP codes since he knew that there were numerous tickets describing a feature request related to that topic. Additionally, he wanted to identify those bugs in that set that had received more than a few comments, and that had a screenshot attached. We created the following tag set:

```
TAG "ZIP" WHERE CONTAINS("zip") = true AND HAS(screenshot) = true AND
   FREQUENCY(comment) > 2
```

The tag set contained 0 bugs. We then suggested removing the comment and screenshot criteria from the BTL statement, and we created a simpler tag set:

```
TAG "ZIP" WHERE CONTAINS("zip") = true
```

The tag set contained 9 bugs. We then enabled the *Has Active Tag* quick filter and enabled the *COMMENTS* tag set. This led to 705 bugs being displayed in the bug list, with 4 belonging to both tag sets. We then used the resulting bug list as a starting point for further exploration. Browsing through the bug list, we identified a bug that belonged to both tag sets, and had an additional static tag with a customer's name.

One use for this tag set is to identify the key bug for a specific concern. For example, of the 9 bugs in the *ZIP* tag set, one or two may have the biggest impact and if resolved, could also address the remaining bugs. The tag set that was created provides the triager with a much shorter list of bugs to review in order to easily identify such key bugs and ensure that it is addressed before any development time is spent on the other bugs.

What Is an Active Bug?

The participant remarked that the number of comments was one of the factors that he would use to determine the level of activity for a bug. Other factors include any updates (changes to any field in the bug), particularly ones that have happened more recently. He also suggested that the number of watchers, or users that have requested to be notified of any changes made to a bug, would be a useful criteria for creating tag sets. He also suggested that being able to identify bugs that are *being watched by a particular user* would also be useful.

Preparing for Release

Tag sets could also be used to prepare for a release by identifying bugs that are not ready to be included in the release notes. The desired tag set would be one that tags bugs that do not yet have the *Resolution Summary* field populated. Another tag set would identify the bugs that have not been assigned to a QA resource yet for testing (which is different from the assignee field). The participant wanted to use tag sets as a way to keep track of specific groups of bugs during various points during the development and release process. We were, however, unable to create either of these tag sets since neither the resolution summary field nor the assigned QA field were part of the default bug data model.

Tags as Temporary State

The participant also commented on the need to review bugs and set them aside for review and verification, either personally or by assigning it later to another member of the team. We demonstrated the ability to create an ad hoc tag set named *TO BE VERIFIED* using arbitrary bugs from the bugs list, as well as the ability to add bugs to an ad hoc tag set, which was well received. As in previous sessions, the ability to multi-select bugs from the bug list and assign them to a user in bulk was seen as a valuable complement of having a *TO BE VERIFIED* tag set.

The participant also commented that this tag set could be used as a way to hand off bug reports to another member of the project to verify the defect. He would perform the first level of triage himself, but then have another member use the same tag set as a work list of bugs that need to be verified. This comment introduces the notion of sharing tag sets as a way to communicate the state of a bug without changing any of the assignee fields. This would likely happen in situations where triaging is a collaborative activity, or it is done in tiers and the role of the lower tier is to prepare tag sets for the next tier. It also echoes earlier feedback for a need to be able to create an inbox bug list.

Challenging Bugs

One of the ideas that came up during this session is that of *challenging* bugs, or ones that can not be quickly triaged and assigned but require careful consideration. These bugs need to be identified and set aside for review with a group rather than individually.

> "Let's find the really interesting things. There are going to be tickets that it's not clear should this be in the product or not. It's it not clear how they should be resolved. But people do have opinions, and there's interest in this area, that's why there are so many comments, watchers, and activity. There are a lot of things to work with here, so this is a great thing to review, this would be a source of input."

In this case, the participant used comments and aforementioned watcher count as proxies to represent challenging bugs. A tag set to identify these might look like the following:

```
TAG "CHALLENGING" WHERE FREQUENCY(comment) > 2 AND FREQUENCY(watchers) > 2
```

This list would then serve as a starting point for a triaging session. From the bugs in this list certain themes may emerge in the form of specific features or needs that need to be prioritized for a future release. The key is that tag sets can be used to narrow down this initial list and make the themes more apparent by providing context about the bug.

Cleaning Up Bugs

Another use case for tag sets that was discussed during the session is the need to manage bug reports that do not have enough information. As a first level triager for the project, the participant viewed his role as making sure that these bugs do not make it past him into a development sprint since his teammates would waste significant time trying to gather needed information. Instead, he would like to identify these bugs as part of his regular triaging and either remove them, or request additional information. He proposed a tag set called *CLEANUP* that would contain bugs that are missing a screenshot, have no comments, and have no labels:

```
TAG "CLEANUP" WHERE HAS(screenshot) = false AND FREQUENCY(comments) = 0 and
   FREQUENCY(labels) = 0
```

We were unable to create this tag set during the session since the `FREQUENCY` function did not apply to the *labels* field at the time. Had bugs been tagged with this tag set, the next step would have been to review them and either remove them, or request additional information from the reporter using the Quick Comment feature.

Static Tags

PorchLight proved to be useful in managing static tags along with tag sets. The participant noted that the ability to enable and disable tag indicators on the bug list was useful, especially when tags were used internally to the project to indicate specific customers of the product. He could then enable specific customer tags and identify bugs which had been requested by multiple customers. He stated:

> "As a product manager, if I fix those I make three customers happy, versus just making one customer happy. I'm doing three times the work for the same number of hours. [My goal] is to maximize satisfaction across the board."

The large number of tags used in the project led the participant to add the number of tags a bug has to his list of criteria for a bug that may be in his *CHALLENGING* tag set, since multiple tags can indicate multiple customers who have similar but slightly different needs.

Changes Made

Based on the feedback provided during the first session, we made several changes to the tool in preparation for the second session. First, we implemented a new function called WATCHING which allows users to tag bugs which are being watched by a specified user. Listing 6.1 shows the implementation of this plugin.

```
public class Watching implements Function {
    public String evaluate(BugGraph bugGraph, Bug bug, String param) {
        for (User user : bugGraph.getWatchersForBug(bug.getKey())) {
            if (StringUtils.equalsIgnoreCase(user.getUserName(), param)) {
                return "true";
            }
        }

        return "false";
    }
}
```

Listing 6.1: The WATCHING function implementation.

Second, we implemented a new field to track the number of watchers a bug has. This field was available in bug report definition in JIRA as the *watchCount* attribute. We extended the bug data model using the plugin architecture and added a new field called *watchers* that would be populated during the import process. This field could then be used in conjunction with the FREQUENCY function to tag bugs that had have a certain number of watchers.

Session 2

The second session was conducted with the same participant eight days after the first session and lasted 58 minutes. We began this session with a quick review of what we had discussed in the first session, and the new features that had been added to address some of the feedback.

We created a new tag set to identify bugs that have a certain number of watchers:

```
TAG "EYEBALLS" WHERE FREQUENCY(watchers) > 3
```

The tag set contained 334 bugs. The participant performed a cursory review of this list and made some observations about a particular bug based on the timeline, specifically the pattern of comments. We then created an additional tag set that takes comments into account:

```
TAG "EYEBALLS2" WHERE FREQUENCY(watchers) > 3 AND FREQUENCY(comments) > 3
```

This tag set contained 152 bugs. Using this refined tag set, the participant began a deeper exploration of the bugs. As a product manager, he considers one of his tasks to be not only a first level triager of bugs before they are assigned to the development team, but also he must regularly "explore the backlog" of bugs to understand the number and types of requests that have come in. He began by looking at the first bug in the list and reviewing the timeline and scanning the comments. Based on the bugs in the list, we discussed the distinction between watchers and comments. Watchers represent external interest in a particular bug, while a significant number of comments can represent "clarifications, questions, thoughts, potential solutions, or requests of urgency." The combination of these two factors in a tag set allows the triager to select bugs for assignment from a "more urgent pool" of bugs.

As in previous sessions, the notion of themes for release was introduced by the participant. He stated that releases are populated based on different themes, sometimes focused on customer requests, other times more focused on architectural changes to the software. To help identify candidate bugs for the former, a tag set based on watchers, comments, and tags could be used. To help with the later, the *component* field in the bug data model becomes useful:

```
TAG "ARCHITECTURE" WHERE component = "LDAP" OR component = "API"
```

A similar tag set could be used as a starting point to identify bugs that would be candidates for a release focusing on architecture changes.

Specific Watchers

Based on feedback from the first session, we also created a tag set to identify the bugs that a specific stakeholder was a watcher on:

```
TAG "STAKEHOLDER1" WHERE WATCHING(stakeholder1) = true
```

This tag set contained four bugs. The participant began reviewing each of the bug reports and though out loud. For the first one, he noted that there were a number of comments on the bug, and that it looked like an architectural change. His triaging decision would be to assign it to the technical lead for the team to provide additional feedback. On the second bug on the list he noticed that there were numerous comments on the issue and noted that "this one should be labeled major" and that "we should probably figure it out," but took no immediate triaging action. On the third bug he noticed that it was labeled with several customer names and remarked that the particular bug was not a defect but rather a question that came up during a support case, and could be addressed by providing documentation to the reporter rather than assigning it to a milestone. On the fourth and final issue on the list, he noted that the bug had a short description with no activity (based on the timeline), so he would want to "investigate to see if it's worth looking at or not."

In this instance, the specific stakeholder used in the tag set oversees a different project which depends on the participant's own project. Bugs being watched by the stakeholder are ones that are typically "changes that are needed to support [the project]," and so this tag set becomes a proxy for another project's list of important bugs. These bugs may also have urgency since the dependent project is one that is in production use, and therefore newly reported defects may need to be resolved quickly. The participant found this tag set to be a useful starting point for triaging, remarking: "I think starting with him is a better starting point than other random [users]."

To identify bugs important to another stakeholder on the project, the participant also created another tag set to find bugs where he was a watcher:

```
TAG "STAKEHOLDER2" WHERE WATCHING(stakeholder2) = true
```

This tag set contained zero bugs. To verify if there were any bugs that he commented on instead rather than watch, the participant created an additional tag set:

```
TAG "STAKEHOLDER2" WHERE COMMENTED(stakeholder2) = true
```

This tag set contained three bugs, which the participant proceeded to review. He noted that several of the bugs that were in the list also belonged to the *EYEBALLS* tag set, based on the presence of the tag set indicator under the summary. He used the left and right arrows keys to iterate through the comments on the issue and found the comments from the stakeholder to identify what his level of interest in the bug is. We observed that, as a triager, the participant used the ability to easily explore the bugs in the list to identify patterns on how specific users interacted with bugs: the types of comments they left (informative versus questions), if they watched issues versus commenting, and so on. We think that this aligns with an exploratory phase that a triager would be in when first starting to explore a larger set of bugs belonging to a project.

There is also a certain awareness of specific watchers, since they are notified via email of any changes made to the bug. That means that if as a triager, you made a chance that is not viewed as the right decision, you may receive negative feedback. Having a tag set for a specific list of watchers can be useful in providing this awareness during triaging to inform—if not impact—the decisions made.

The discussion then transitioned into how tag sets could be used to create the different pools of bugs that could be drawn from at various point for release planning. In his model, he described the states in which a bug can be in, ranging from highly unorganized and missing key information, to groomed and ready for assignment into a development sprint. While we did not work through this entire lifecycle and create the associated tag sets during the session, we did note that the participant began to discuss the triaging process as a workflow where bugs were transitioned from one state to another, and that these pools of bugs were more useful to consider rather than individual bugs.

In the end, the product manager made the least direct use of the functionality in PorchLight and performed the fewest triaging actions. This is likely due to his role on the team, which involves more early-stage triaging—filtering issues and adding information before they are assigned to users or milestones by the development manager for the project. Even so, the feedback provided confirmed that tag sets can be a useful model during the triaging process, particularly for transitioning bugs between various internal states that may not be formally represented in the bug tracker.

Development Manager

The third participant is a software development manager who oversees development of a large commercial project. The project currently has over 8,100 change requests that have been reported since the team began using the bug tracker in 2007 (of which we loaded the most recent 3,000 into PorchLight). One of his responsibilities is the triage and tracking of new bugs that are reported, and making sure that they are assigned to milestones and developers. He coordinates with other members of the team, including the assigned product manager and QA resource, to verify and manage the large number of bugs and updates to bugs.

Session 1

The first session lasted 58 minutes. The overview and tutorial portion lasted 7 minutes. As with the product manager, the tutorial was significantly shorter since the participant was already somewhat familiar with the features because he had also participant in the preliminary use study.

Comment to Verify

After completing the tutorial, the participant remarked that the Quick Comment feature in particular was useful, and that instead of right away assigning bugs to developers and milestones, foresaw a model in which he would use an additional comment requesting feedback from a teammate. Rather than assigning the bug to the user, he would prefer to leave a comment and mention the specific user so that they receive a notification and review it. He stated, “I won’t usually assign an issue to [QA] or [the Product Manager], but I’ll ask them to give me feedback to figure out how to assign it.” Only when the bug has been confirmed will he assign the bug to a milestone and to a developer. This theme of validating bugs by addressing comments to specific team members was discussed towards the end of the session as well.

Email as a Bug Queue

This led us into our first topic, which was prompted by our question: do you maintain a list of bugs that you have asked your teammates to review before triaging them? His response was that he uses his email inbox as a work queue to keep track of bugs that are out for review:

> “The way that I usually track them, which works for me, is every update to any issue I get an email notification for. An so I’ll go through that queue of a hundred plus emails a day as they come through, with updates, comments, whatever, because I have it setup in JIRA so that I’m tracked on everything. I’ll go through each of those and see every update, I’ll look at it and see if it pertains to me, if it’s something that I care about.”

His email inbox becomes a launching point into a workflow that involves others on the team responding to requests for verification or information, and his being notified of those changes as a trigger to take the next triaging action. Sometimes the responses will be that the bug is not actually a defect and can be closed, other times the defect is verified by QA and he assigns the bug to the appropriate developer. He stated:

> "I'd say 95% of the time I will just make sure that I have zero in my queue, which is essentially just a filter in my email, by the end of day. So every morning I'll try and go through them, and then every afternoon I'll try and go through them, because I usually have over a hundred updates a day. In my email it's a work queue, because I can see which ones I've read and which ones I haven't read."

We then transitioned to how he could potentially create this work queue using tag sets. We used the following tag set as an example:

```
TAG "COMMENTS" WHERE FREQUENCY(comment) > 3
```

This tag set contained 288 bugs. The notion of having dynamic tags that are automatically applied based on the BTL expression rather than manually having to tag them was well received, as he stated:

> "Frequency of comments I think is a good indicator to tell if an issue is important to people. Even though there are other metrics in JIRA that should be used for that, like votes and watchers."

Since we now had the votes field available in BTL, we created a second tag set to find bugs with votes:

```
TAG "VOTES" WHERE FREQUENCY(votes) > 2
```

This tag set contained zero bugs, which confirmed that votes were not used in this project to indicate demand from the users of the bug tracker. We created a tag set for watchers based on the participant's suggested threshold:

```
TAG "WATCHERS" WHERE FREQUENCY(watcher) > 5
```

This tag set contained 115 bugs. The participant commented that this tag set is helping him to "discover what other people think is important." While browsing through the bug list he noticed a particular bug that stood out based on it's description and the activity in the timeline:

> "This is interesting. This issue that I'm looking at right now, I know for a fact that this has been opened, closed, reopened, probably a dozen times. It's had a lot of problems in development (it's a performance optimization) and so I would expect this one to show up because it's not just an important issue, but it's had a lot of issues, so it's probably one that is good to keep an eye on."

We then enabled the first tag set that we created showing the bugs that had received more than three comments. Like in previous sessions, the participant noted that the presence of multiple tag set indicators in the bug list represented the intersection of the enabled tag sets, and he attempted to sort the bug list using that column header (this functionality was not available). This was important because he wanted to see the bugs that matched by criteria. He commented that the these two tag sets would not help him identify new bugs to triage, but instead it helps to identify bugs that he knows have had "either complications or problems or were heavily requested."

He then enabled the open Quick Filter and narrowed the bug list down to the ones that were in the *COMMENTS* and *WATCHERS* list and that were still open. We then discussed if this list would be a good starting point for triaging, and he suggested an alternative tag set that identified bugs that were:

- Assigned to him
- Unassigned
- Do not have a fix version (unassigned to a milestone)

He created the first tag set by transferring his username from the user list into the tag set list. This created a tag set containing 109 bugs. He then enabled the *Unassigned (Version)* Quick Filter to achieve the second part of the tag set. We continued the session by exploring the bug list:

> "Then I could take the intersection of those tag sets, say okay here's everything that's not assigned to a fix version but has a lot of activity or comments or watchers."

Recent Activity

Recency also was identified as an important triaging factor, since "being able to figure out how recent the activity is important because [a bug] might have dozens of comments from years ago, but we ended up resolving it a different way and went back to the issue to close it out." We suggested creating a new tag set using the SINCE LAST clause of the BTL expression. The participant indicated that he would create multiple tag sets: one showing bugs updated in the last 30-60 days, and one showing the ones updated in the last 6 months. He also suggested that it would be useful to sort the bug list by the recency. We created a new tag set using the SINCE LAST clause to demonstrate how this could be accomplished:

```
TAG "ACTIVITY" WHERE FREQUENCY(comment) > 3 SINCE LAST "30 days"
```

We noted that this tag set would identify bugs that had received at least three comments and has been updated in the last 30 days. This tag set contained 288 bugs, which he enabled along with the previously created *Watchers* tag set. He then proceeded to review the bugs in the list, scanning the descriptions and rapidly traversing through the comments using the timeline feature. He there after enabled the tag set containing issues that were assigned to him that we created earlier, and began reviewing the bug list for bugs which belonged to all three tag sets.

Stack Traces and Error Messages

We then transitioned to looking at specific fields in the bug and if they were useful for triaging by creating a tag set to identify bugs which contained a stack trace:

```
TAG "STACK" WHERE HAS(stacktrace) = true
```

This resulted in an error in the console output and the tag set was not created. We noted that there was a defect in this feature and mentioned we would attempt to address it in the next session. We nevertheless discussed how a stack trace could help to identify bugs for triage:

> “I *would* use stack trace. Attachment maybe not, people post screenshots for a lot of reasons, people post other things as attachments for other reasons. A stack trace is something that I feel should never be seen. So, if someone reported an issue and was able to provide a stack trace, to me it’s already a legitimate bug... it’s something that someone on my team needs to look at because the stack trace should never have appeared to the user.”

The presence of a stack trace in a bug automatically escalates the bug to him as something that needs to be triaged. We continued to discuss other content in the bug that might identify problems that the user should not see, including specific error messages. Based on his comments, we suggested potentially creating a tag set that would identify bugs that either had a stack trace, or contained in their description the phrase ‘Internal Server Error,’ that he could the use for regular triaging of those bugs. We then discussed the idea of identifying bugs that are missing information and how they would be addressed.

Being Mentioned

Referring back to the ability to create tag sets based on if a person has commented on a bug, the participant proposed a tag set that he thought would be useful for his team to use, which would identify all of the issues that he had commented on in the preceding 30 days. This would ensure that they see his comments, which may contain concerns about a proposed implementation or a complement for having addressed a problem in a creative way. The participant requested that the `SINCE LAST` clause have the option to apply to *specific fields* specified in the clause, rather than only checking the date the bug was last updated.

We then asked the participant if he found the ability to create tag sets based on either the number of people watching, or a specific set of watchers. He responded, "I would want to know if I was *mentioned*." In JIRA, a user has the ability to add a comment referring to a specific user, which then automatically notifies the user of the comment and makes them a watcher on the bug. The participant wanted to write a tag set that would identify the bugs which contained a comment in which he had been mentioned. We were unable to create this tag set because the existing `CONTAINS` function only searches the description and summary fields for text. To continue the discussion, we instead implemented a tag set for the bugs *which he was watching.*

```
TAG "PARTICIPANT" WHERE WATCHING(participant) = true
```

This tag set contained 176 bugs. The participant began reviewing each of the bugs in the bug list and thinking out loud the reasons he thought he was added as a watcher on the bugs. Some of the bugs he had created, others he had commented on, and others he had been added as a watcher by someone else on the team. We suggested that he enable another tag set which contained bugs that he had been assigned to be able to distinguish those in the bug list. He enabled that tag set in the tag set list, and continued reviewing the list.

Validating Before Triaging

We then discussed his role in the triaging process, and he stated that on the large project that he oversees he is not able to triage issues by himself, and that he required input in the process from his teammates: "If it was [the smaller project], I could probably make 100% of the decisions on my own. With [the larger project], with all the different pieces, and especially with customer input, it's not usually true. We have more bugs that may or may not be actual bugs, so there is validation needed."

We asked the participant to elaborate on the type of information he looks at to help make triaging and validation decisions, especially on the larger project where there are numerous bugs being created regularly. He responded stating:

> "Sometimes the person that made it. Sometimes, whether or not it has information like a stack trace. If it has information it's not a garbage bug, so that's important." However, he stated that rather than approaching from the angle of cleaning up bugs that have reported, he stated that "rather than getting rid of garbage bugs, I would prioritize bugs that have valuable information, like a stack trace."

We asked if the reporter field or other labels are useful for triaging. He responded "yes to both" and described a tag set that identified issues that had been created by a specific developer on the team that worked exclusively on bugs relating to a specific customer. As he described this tag set, we implemented it using the following BTL statement:

```
TAG "DEVELOPER" WHERE reporter = "developer"
```

This tag set contained 104 bugs. He also identified labels used in the bug tracker to denote bugs reported by, or related to, specific customers. He enabled those tag sets in the tag set list and began reviewing the list. While reviewing the bug list, he used the Quick Filters to ensure that he was only viewing bugs that were currently open. After reviewing a few of the bugs, he confirmed that they were indeed the bugs that were relevant to a specific customer and that they needed to be closely watched and triaged if not already assigned.

After reviewing the list, he also identified another developer that he could create a similar tag set for that would identify bugs that were likely related to performance or scalability bugs.

The participant then mentioned that he used the component field often to associate bugs with different parts of the software. He commented that he would want the ability to easily assigned multiple bugs to a component using the multi-select functionality that he used earlier:

> "If something comes in, and I'm triaging, and it is supposed to be [component] related, I need to make sure the component is on it. I don't do that enough right now because it's kind of a pain, but if I could bulk do that, that would be nice."

Finally, the participant referred back to two ideas that we had discussed earlier. First, his desire to be able to create a work queue using tag sets that would allow him to identify the important bugs that needed to be reviewed and triaged on a regular basis. Second, he suggested the possibility of using some of the tag sets that were created in the session, in conjunction with the `SINCE LAST` clause available in BTL, to limit the work queue to a specific time window, like the last 24 hours, or the last time that he had reviewed the bug queue.

Changes Made

Based on the feedback provided during the first session, we implemented a new function called `MENTIONED` which allows users to tag bugs which in which the specified user has been mentioned in the comments. Our implementation of this function, shown in Listing 6.2, is specific to JIRA in that it uses a particular format to determine if the text is a mention, which in the bug tracker is presented as a link to the user's profile.

During the first session we encountered a defect in the BTL evaluation logic that resulted in

```
public class Mentioned implements Function {
    public String evaluate(BugGraph bugGraph, Bug bug, String param) {
        Pattern pattern = Pattern.compile("\\[\\~" + param + "\\]",
           Pattern.CASE_INSENSITIVE);

        if (StringUtils.isNotEmpty(bug.getDescription()) &&
           pattern.matcher(bug.getDescription()).find()) {
            return "true";
        }

        for (Comment comment : bugGraph.getCommentsForBug(bug.getKey())) {
            if (pattern.matcher(comment.getBody()).find()) {
                return "true";
            }
        }

        return "false";
    }
}
```

Listing 6.2: The MENTIONED function implementation.

a NullPointerException being thrown when invoking the HAS(stacktrace) function. Upon inspecting the code we saw the same defect the affected the use of CONTAINS: not taking into account that some bugs had a blank description. We implemented a check for this condition and were able to create a new tag set using the function.

Session 2

The second session was conducted with the same participant three days after the first session and lasted 54 minutes. We began this session with a quick review of what we had discussed in the first session, and we presented the new features that had been added to address some of the feedback.

Before the session, we created two new tag sets based on the suggestions from the previous session. One was a a tag set to identify bugs that contained a stack trace (which failed the previous session):

```
TAG "STACKTRACE" WHERE HAS(stacktrace) = true
```

This tag set contained 58 bugs. The other was a tag set that contained bugs in which the participant was mentioned in a comment on the bug.

```
TAG "MENTIONED" WHERE MENTIONED(participant) = true
```

This tag set contained 73 bugs. With both tag sets enabled, the participant took control of the tool and began browsing through the bugs list. His first step was to again use the Quick Filter to make sure that he was only looking at the open bugs, excluding ones that had already been resolved or closed. While bugs belonging to both tag sets were in the bug list, he wanted to first look at the ones in the *STACKTRACE* tag set, so he used the tag set indicators and the row color to identify the ones belonging to that tag set.

He began to review the bugs and immediately identified one that he had not seen before, stating: "[PorchLight] would help because this is an open issue that I apparently should close." Based on the comments on the issue, and his knowledge of the history of the bug, he knew that it was not longer a problem and it had already been resolved in a prior release. As he continued down the list, he identified another pair of bugs that he did not know were still open. He stated, "They have not been triaged, and they should have been. They should have been closed, they should not still be open, so it's just cluttering our JIRA." He then proceeded to use the Quick Comment feature to change the status of the bug to close, and added a comment "No longer an issue." He repeated the action for the second bug.

Before moving on the remaining bugs, he commented that he did not see much overlap between the two tag sets that were enabled (*MENTIONED* and *SCREENSHOT*, so he disabled the *MENTIONED* tag set from the tag set list to only view the ones that had contained a screenshot. This action reduced the bug list down to only 13 bugs. As he continued down the list, he identified another issue that should have been closed, and again used the Quick Comment feature to close the bug and comment that it was no longer an issue. When we asked how he was making the decision to close the bug, he recalled information that he knew again about the history of the bug and how it had been addressed in some other way in a prior release. He also referenced the age of the bug, referring to the timestamp in the timeline, and commented that the bug was too old to be relevant any longer.

As he reviewed the bugs, the participant continuously referred to the assignment indicators in the user and milestone lists to determine if a bug had been assigned, and who it had been assigned to. He observed that many of the bugs that had been identified in the *SCREENSHOT* tag set were also assigned to a product manager that was no longer on the team. He stated:

> "There are probably ones that need to be looked at, but never were caught. This is shortly before [the product manager] left, so these would be important to check out and see. I wouldn't have been able to find these, I don't think."

When we asked the participant about how he was making decisions about a bug in the list, he would frequently use the timeline to traverse through the comments and to tell a narrative about the history of the bug. This included questions that he had asked others on the team, and decisions that had been made through the conversation. He came across another bug that he thought was not valid, so he used the Quick Comment feature to leave a comment asking "What's the use case for this test? Why are you linking directly into an action page?"

For the next bug in the list the participant decided that it should be assigned to the next development sprint, so he used the drag-and-drop feature to drag it from the bug list and drop it onto the milestone. He then also assigned it to a developer on his team. He used the drag-and-drop assignment and Quick Comment feature several more times as he reviewed the remaining bugs in the list.

Ad Hoc Tag Sets

When discussing his triaging decision for one of the bugs, the participant commented that the bug was of a class of bugs that he was content to "just let sit there" because he knew that they would be resolved indirectly. He stated, "I could start closing them, but I haven't been." We suggested he use the ad hoc tag set feature to create a new tag set for holding these issues, and he created it by dragging the single bug from the list. The next bug he reviewed he identified as also belonging to a an ad hoc tag set, so he proceeded to drag-and-drop it into the ad hoc tag set in the tag set list.

The participant then used the multi-select feature in the bug list to select three bugs and assign them to a development sprint, and to himself, so that he would know to come back to it and assign the individual bugs to specific developers. He also requested an option to disable milestones that had been released in the milestone list, which at the time displayed both released and unreleased versions. He then disabled the *STACKTRACE* tag set in the tag set list and enabled the *MENTIONED* tag set, which contained bugs that he was "mentioned in." Upon seeing the list, he stated that:

> "These I feel like they're helpful, but to be more useful I'd almost need a work queue so that I can work through them. Usually if someone mentions me that means I have to respond to them. So, it would be nice if I could say okay I've responded and get it out of my work queue."

Working the Queue

He described how he would make a triaging decisions about each bug in his work queue: "Assuming it's a work queue, I would want to look at the very latest comment on each of these, because that's probably where someone is saying 'look at this'. And so I'd look at the latest comment and mark it as read, or do something with it."

We asked how he keeps track of the bugs that he needs to assign to development sprints. He described his workflow for managing these bugs: "First I pick everything that goes into a sprint, and keep it [assigned] to myself if I'm not sure. I'll review them and some of them I'll move out, some of them I'll add more information, and some I'll assign to people." When we pointed out that he could potentially use ad hoc tag sets to track the bugs that needed review before being assigned to a sprint, he responded "that would be nicer because it wouldn't mess up our metrics as much," referring to the bug distribution across milestones.

He realized that there was another situation in which ad hoc tag sets could be useful. He described a set of bugs related to a specific type of enhancement request that he frequently triages in the same way. He normally identifies each bug individually, and only mentally knows that it belong to that group. He points out that he could create an ad hoc tag set for those bugs, and then refer to that tag set periodically while triaging to keep tabs on their status. He commented, "Creating a tag set for those would be really helpful. I don't do that now because I have to go to each one and tag it and I just don't do that." This avoid having to "hand-pick the 7 or 8 issues" each time he reviews the bugs assigned to a milestone, since it is normally "really hard to find them again."

The participant also created a new tag set to identify bugs that had been assigned to him for the next milestone. He then, with our guidance, created a tag set using the following BTL statement:

```
TAG "SPRINT20" WHERE assignee = 'participant' AND fixVersion = "2.5 S20"
```

This resulted in an error, and the tag set contained zero bugs. We later confirmed that this was a bug in PorchLight, exposed by the space in the milestone name. Recognizing that there was a different way to create the same tag set, he used the drag-and-drop functionality to drag the *2.5 S20* milestone from the milestone list onto the tag set list, which created the tag set containing bugs assigned to that development sprint. To find the milestone, he used the search function in the milestone list which narrowed down the number of items. He then began to review the bug list, referencing the user list to identify the ones that had been assigned to him.

He identified several bugs in the list that belonged to one of the themes that he described earlier, and he proceeded to drag-and-drop them onto the tag set list, creating a new ad hoc tag set called *Improvements*. He did this for four of the bugs in the list. We pointed out that he could enable only the *Improvements* tag set to view those bugs, and multi-select them to perform a triaging action. He replied that he found that feature useful since he could assign that group of bugs to the next development sprint if needed, adding, "because I do that all the time."

Forgotten Improvements

We then suggested that he create a tag set containing bugs that were still assigned to the product manager that had left the project, and which he as the project lead had inherited. He did this by dragging her user name from the user list onto the tag set list, which created a tag set using the following BTL statement:

```
TAG "PM" WHERE assignee = 'productmanager'
```

This tag set contained 283 bugs. Commenting on the large number of bugs in the list, the participant joked that "This list represents the list of stuff I don't want go through." He then proceeded to enable the *Unassigned (version)* Quick Filter to identify the bugs that the product manager had not yet assigned to a version. This did decrease the size of the bug list, but the participant observed that it was still "a little overwhelming." When we asked how he would approach a large list of bugs he replied that he would "normally narrow it down by version," meaning he would look at the custom milestones (like *Enhancements*) that had been created to track unassigned bugs. To find this milestone in the list, he again used the search feature to narrow down the list. Once he found it, he dragged it onto the tag set list which resulted in a new tag set based on the following BTL statement:

```
TAG "PM's Enhancements" WHERE fixVersion = 'PM's Enhancements'
```

This tag set contained 151 bugs. Like the participants in the previous sessions, the participant remarked that it would be useful to be able to view the *intersection* of multiple tag sets, even if it could be accomplished by enabling multiple tag sets and sorting the bug list based on the tag set indicator column. This would allow him to use the *PM* tag set as a base list, and then enable the tag sets that represented the types of bugs in which he was interested.

Using the bug list, which now contained bugs in both the *PM* and *PM's Enhancements* tag sets, he proceeded to review each of the bugs and think out loud about what triaging action he should take. His objective with this list was to essentially perform cleanup of bugs that he knew had not been reviewed by anyone in some time. He identified several bugs during this review that he then assigned to himself for the next development sprint, again using the drag-and-drop functionality. As with previous reviews of the bug list, he also made heavy use of the timeline to review comments, and of the Quick Comment feature to add a comment.

At one point he came across a bug that he thought had a duplicate, and he wanted to see if he could find it somehow. We suggested using the Search feature at the top of the bug list which would search the contents of the bug in the bug list in real time. He began entering keywords into the search box, which narrowed down the bug list to matching bugs. He was not able to find the bug that he had thought of, however, since it likely was not in the bug list with the tag sets and Quick Filters that he had enabled. He then identified a bug that described a defect which resulted in a `NullPointerException` being thrown. The participant commented that he would need the full stack trace from the reporter to be able to triage it, and we reminded him of the pre-defined responses available in the Quick Comment feature. He then used the Quick Comment feature to then add a comment to the bug requesting a stack trace.

Validating Old Bugs

He also remarked on the fact that the age of a bug is a factor when triaging, stating "A bunch of these may still be valid, and this point I don't know. A lot of these issues have probably been reported again, as a new ticket, and we've probably solved it six months after it was created. If these are nine months old like this one we've probably resolved it a different way or as a part of a different issue." He frequently referred to the date when the bug was created, along with the amount of activity on the timeline, to determine if it he considered it a valid issue, adding "every issue I go to, I look how old it is first." He then proposed a tag set to help identify these types of bugs that needed validation, using the following criteria:

> "I'd probably look at anything that's open, that is reported as a bug, that is not assigned to me or an actual fix version... that would be to whittle this down from 300 to hopefully like 20 or 30."

Finding Duplicates

We ended the session with an open discussion on identifying duplicate bugs, and how the participant approaches the problem. While he had no specific technique for identifying duplicates during triaging other than his own memory of the bugs that he had seen, he commented:

> "It would be nice when triaging an issue to see a list of suggested possibly related issues. And then from there you could look at the related open issues and possibly close one as a duplicate, or link them."

6.3 Discussion

In this section, we analyze the narrative from the sessions presented in Section 6.2 and distill the main themes and lessons learned about triaging with PorchLight. We have structured our analysis in three parts. First, we summarize the features used and tag sets that were created during the participatory observation sessions, and discuss the types of tag sets. Then, we analyze the same sessions at a higher level and look at how the participants talked about and used tag sets, and the role tag sets could play in triaging, to identify the themes that were present across the sessions. Finally, we present an analysis of the observations from both studies across six assessment criteria.

6.3.1 Feature and Tag Set Usage

As a first step in our analysis of the data collected from each session, we to simply look at how the basic features, and especially tag sets, were used throughout. Table 6.3 summarizes the features that were used during the sessions.

Feature	**Dev Co-Leads**	**Prod Manager**	**Dev Manager**
User assignment			✓
Milestone assignment	✓		✓
Multi-select	✓		✓
Timeline	✓	✓	✓
Viewing comments	✓	✓	✓
Quick Comment			✓
Quick Filters	✓	✓	✓
Search	✓		✓
Tag sets	✓	✓	✓
Ad hoc tag sets	✓		✓

Table 6.3: Summary of PorchLight feature usage by participant.

We see that, while all of the notable features of PorchLight were used throughout the sessions, they were not used by all of the participants. For example, the product manager did not use the assignment features, though he did use the timeline and tag sets to explore bugs. As another example, the development co-leads did not make use of the Quick Comment feature while triaging, but the development manager did. In the analysis below, we review each of the features, highlight its key role, and the feedback we received. For this, we include observations from the preliminary user study, which exposed users to the entire range of functionality available in PorchLight, and facilitated more direct feedback on those aspects.

Basic triaging and features (layout, assignment, and multi-select). We received positive feedback from the participants on overall user interface, particularly the layout and the drag-and-drop user and milestone assignment functionality, which they found useful for triaging. One of the participants stated that, "it's useful to have the list and users and the versions right apparent to you," referring to the user and milestones lists. He also stated:

> "We typically have our list of JIRA issues and then we open them up one at a time as tabs in the browser. Then we go through each one and assign them one after the other. That's typically how we do it, but *it's nice to have all this information at once so that we can quickly flip between issues without having to go to different tabs in a browser.*"

We also observed heavy of the ability to multi-select bugs from the bugs list and either assign them, or create new ad hoc tag sets from them. The participants found the interaction to be intuitive and preferable to their current approach, which involves a multi-step process of identifying the bugs and then performing a bulk action with them.

Timeline, viewing comments, and Quick Comment. We also received positive feedback on the timeline, since it provided an at-a-glance overview of activity on the bugs, and allowed the participants to easily review the comments and other actions that had been performed. One of the participants noted the number of comments on a particular bug and remarked on the usefulness of the timeline, stating: "Activity [in the timeline] conveys a general sense of the density of people commenting." Another participant also commented on the usefulness of the timeline, stating:

> “I think the timeline is useful because you can see if the comments are all concentrated in one period time two years ago, versus an issue where its spread out evenly and there are recent comments, and so that would tell me that it’s probably more important, [if] the activity has been spread out across the last year or so.”

We also observed heavy use of the Quick Comment feature during one of the sessions. The participant used the feature to both change the status of a bug (from Open to Closed), and also to add a comment requesting additional information or a stack trace for the error described in the bug.

Quick Filters and search. We also saw use of the Quick Filters and search features throughout the sessions. In many cases, the participants used the Quick Filters to narrow down a bug list created from selecting tag sets, rather than additionally specifying a clause for the status of the bug in the BTL statement. This allowed them to easily switch, for instance, between the open and closed bugs in the bug list. The search features were also used to find bugs based on specific keywords or based on their unique identifier. The filter attached to the milestone list was used on several occasions to narrow the list of milestones when looking for a specific one to assign to. One participant commented during the preliminary user study that:

> “I would honestly use [PorchLight]. *With the timeline and the filters, it has the information I need.* I would just have it open all the time.”

Tag sets. Beyond the basic feature usage, we wanted to analyze both the number and type of tag sets that were created. In total, there were 32 tag sets that were created, either directly by the participants or indirectly via the researcher with input from the participants, during the 6 sessions. Below is a summary of the tag sets that were created during each of the sessions.

```
TAG "ACTIVE" WHERE FREQUENCY(comment) > 3 SINCE LAST "2 months"
TAG "3.4.0" WHERE fixVersion = "3.4.0"
TAG "WS" WHERE STATUS = "Open" AND CONTAINS("web service") = true OR
   CONTAINS("soap") = true OR CONTAINS("http") = true
TAG "STARTING" WHERE STATUS = "Open" AND CONTAINS("web service") = true OR
   CONTAINS("soap") = true OR CONTAINS("rest") = true AND FREQUENCY(votes) >
   5
```

Listing 6.3: Tag sets created during first session with development co-leads.

```
TAG "VOTES" WHERE FREQUENCY(votes) > 1
TAG "VOTES2" WHERE FREQUENCY(votes) > 4
TAG "31WS" WHERE fixVersion = "3.1.0" AND CONTAINS("web service") = true OR
   CONTAINS("soap") = true OR CONTAINS("rest") = true
TAG "COMMENTED" WHERE COMMENTED(user1) = true OR COMMENTED(user2) = true ...
   OR COMMENTED(user15) = true
TAG "USER" WHERE reporter = "user"
```

Listing 6.4: Tag sets created during second session with development co-leads.

```
TAG "ACTIVE" WHERE FREQUENCY(comment) > 3
TAG "ZIP" WHERE CONTAINS("zip") = true AND HAS(screenshot) = true AND
   FREQUENCY(comment) > 2
TAG "ZIP" WHERE CONTAINS("zip") = true
TAG "CHALLENGING" WHERE FREQUENCY(comment) > 2 AND FREQUENCY(watchers) > 2
TAG "CLEANUP" WHERE HAS(screenshot) = false AND FREQUENCY(comments) = 0 and
   FREQUENCY(labels) = 0
```

Listing 6.5: Tag sets created during first session with product manager.

```
TAG "EYEBALLS" WHERE FREQUENCY(watchers) > 3
TAG "EYEBALLS2" WHERE FREQUENCY(watchers) > 3 AND FREQUENCY(comments) > 3
TAG "ARCHITECTURE" WHERE component = "LDAP" OR component = "API"
TAG "STAKEHOLDER1" WHERE WATCHING(stakeholder1) = true
TAG "STAKEHOLDER2" WHERE WATCHING(stakeholder2) = true
TAG "STAKEHOLDER2" WHERE COMMENTED(stakeholder2) = true
```

Listing 6.6: Tag sets created during second session with product manager.

```
TAG "COMMENTS" WHERE FREQUENCY(comment) > 3
TAG "VOTES" WHERE FREQUENCY(votes) > 2
TAG "WATCHERS" WHERE FREQUENCY(watchers) > 5
TAG "ACTIVITY" WHERE FREQUENCY(comment) > 3 SINCE LAST "30 days"
TAG "STACK" WHERE HAS(stacktrace) = true
TAG "PARTICIPANT" WHERE WATCHING(participant) = true
TAG "DEVELOPER" WHERE reporter = "developer"
```

Listing 6.7: Tag sets created during first session with development manager.

```
TAG "STACKTRACE" WHERE HAS(stacktrace) = true
TAG "MENTIONED" WHERE MENTIONED(participant) = true
TAG "SPRINT20" WHERE assignee = "participant" AND fixVersion = "2.5 S20"
TAG "PM" WHERE assignee = "productmanager"
TAG "PM’s Enhancements" WHERE fixVersion = "PM’s Enhancements"
```

Listing 6.8: Tag sets created during second session with development manager.

We first look at the frequency with which the features of BTL were used in the creation of tag sets in Table 6.4. Note that, if a feature was used more than once in a single statement, we only count it once.

Feature	**Type**	**Instances**
FREQUENCY	Function	14
CONTAINS	Function	5
HAS	Function	4
watchers	Field (custom)	4
votes	Field (custom)	4
fixVersion	Field	4
WATCHING	Function (custom)	3
COMMENTED	Function (custom)	2
SINCE LAST	Time Window	2
reporter	Field	2
assignee	Field	2

Table 6.4: Summary of BTL feature usage.

From this analysis we can make several observations. First, the most frequently used feature of BTL was the `FREQUENCY` function. This function was used in conjunction with fields like *watchers*, *votes*, and *comments*. Based on the feedback we received during the session, this observation correlates with the notion that tag sets allow triagers to offload complex search criteria that they can then use to explore the bugs. The tag sets become a proxy for specific concerns that are brought to bear when triaging. For example, the participants used actions like comments and votes, in conjunction with the `FREQUENCY` function, as proxies for demand. Or, they used just comments as a proxy for activity, and watchers as a proxy for importance.

Second, we note that, while a number of fields and functions were used multiple times, very few of the fields available in the entire bug data model were used to create tag sets. For instance, the *summary*, *type*, *priority*, and *resolution* fields that are part of the bug model were not used directly. We think this is partly due to the fact that some of the functions that are provided by PorchLight encapsulate multiple of these fields in their implementation. For example, the CONTAINS function searches both the *summary* and *description* fields, so there may just be no need to be more specific. However, we also think that the small number of fields that were used suggests that triagers were focused more on identifying trends and patterns in the bugs, rather than focusing on individual ones and identifying them by their fields.

Tag set types. Taking this analysis one step further, we grouped the tag sets that were created into three categories: simple, advanced, and custom. Simple tag sets are ones that use basic fields or actions that are already available in the bug tracker. Advanced tag sets use functions or time windows to identify bugs. Custom tag sets are ones that require the addition of new functionality to BTL using the plugin framework. Table 6.5 summaries the frequency of the tag set types across all sessions.

Type	**Frequency**
Simple	7
Advanced	13
Custom	12

Table 6.5: Frequency of tag sets created by type.

We can make some initial observations about these results. Both advanced and custom tag sets were used more than basic ones. This may be due to the fact that the participants felt that they could already identify bugs based on simple criteria using their bug tracker, and wanted to explore the more interesting ones using the advanced functionality. It could also be that the standard Quick Filters for identifying the open and unassigned bugs mitigated the need to include these fields when creating the tag sets. This also correlates with the fact that FREQUENCY and HAS were two of the most frequently used functions, which suggests that the triagers are identifying bugs based on their metadata rather than the data contained explicitly in the fields.

Another observation we can make from these numbers is that tag sets are more useful when created using the more advanced functionality that is not readily available when triaging with traditional triaging tools, such as the frequency of actions or time windows. An example is the `HAS` function, which could be used in conjunction with specific fields. One participant commented on the usefulness of the `HAS(stacktrace)` function when he stated:

> "I think the stack trace [tag set] was really useful. I could actually use that list of changes I made, I wouldn't have really had another good way to find those that I know of."

Plugins to support new tag sets. For the custom plugins that we developed, we also counted the number of lines that were needed to implement the extensions in PorchLight and BTL once the desired functionality was specified by the participant. Table 6.6 summarizes each of the custom plugins that were developed, with Listing 5.8 and Listing 6.1 showing the source code for the *Commented* and *Watching* implementation, respectively.

Plugin Name	**Type**	**Description**	**Lines of Code**
Commented	Function	Returns true if the specified user has commented on a bug.	31 (Shown in Listing 5.8)
Votes	Field	Return the number of votes attributed to a bug.	14
Watching	Function	Returns true if the specified user is watching a bug.	26 (Shown in Listing 6.1)
Watchers	Field	Returns the number of watchers subscribed to a bug.	52
Mentioned	Function	Returns true if the specified user has been mentioned in the comments for a bug.	34 (Shown in Listing 6.2)

Table 6.6: Summary of custom plugins developed.

In the situations where a new plugin was needed to create the desired tag set, the number of lines of code needed to implement the functionality was relatively low. The one exception was the Watchers field, which required more significant changes to the PorchLight implementation beyond what the plugin mechanism afforded. This was due to the fact that the desired field from the bug tracker, in this case *watchCount*, was stored in a schema that was only available through another request to the REST API. The plugin mechanism that we had designed assumed that the data would be available in the bug schema that is provided by the initial request for a bug report.

Even with the limitations in the implementation, we note that we were able to incorporate the new field into the bug data model and make it available for use in a relatively short period of time. The fact that we were able to implement five new capabilities in our approach to address the needs of triagers using different approaches in the span of a few weeks speaks to the design decisions we had made during our implementation. Looking at Table 6.6, we can state that the changes that we had to make, even outside of the scope of the plugin architecture, were minimal. We did not have to start from scratch, or change the architecture significant to incorporate a new function into the tagging language. In one instance, we were able to incorporate a new field and make it available for use with only 14 lines of code.

Summary

While we did observe commonalities in how tag sets were used across the sessions, we did not see a large number of tag sets created directly by the participants. This could be in part due to a learning curve associated with using BTL to express ideas for tag sets. Or, it may be because the fields and functions that we chose for our initial data model only partially addressed the desire for more metadata-based tag sets. It is important to note, however, that the tag sets that were created are persisted and can be used across triaging sessions. This means that once the initial work of creating the tag set, or implementing the custom plugin, is complete, the value of the tag set can be realized many times. Furthermore, the tag sets created by others can be shared, enabling other triagers to benefits from the thought that went into their creation.

6.3.2 Thinking In Tag Sets

Throughout the sessions, we observed that tag sets became a useful model for thinking about how groups of bugs could be composed to explore the entire collection. Even in instances where the functionality was not available in PorchLight or in BTL, the participants had no trouble proposing potential tag sets using new attributes and functions that they thought would be useful during their triaging. Furthermore, the participants built upon this idea, and began modeling their own latent workflows for triaging to better understand how tag sets could help them improve their process.

Tag sets as proxies for concerns. We observed that tag sets provided a way for participants to offload complex search criteria that they then used to explore and understand the myriad bugs. Tag sets became a shorthand, or *proxy*, for specific concerns. In this context, we define a *concern* as a fact about a bug that is important when performing triaging. These concerns can be ones held by the individual participant, or shared by a group or team. For example, the participants used actions like comments and votes as proxies for demand, or the number of comments as a proxy for activity, or watchers and specific commenters as proxies for importance. One participant remarked that a bug with a large number of comments "means it's usually a heavily debated area, so there's some importance to it" and that he has to "look at it more closely" when triaging. Another participant commented:

> "Let's find the really *interesting* things. There are going to be tickets that it's not clear should this be in the product or not. It's not clear how they should be resolved. But people do have opinions, and there's interest in this area, that's why there are so many comments, watchers, and activity. There are a lot of things to work with here, so this is a great thing to review, this would be a source of input."

During one session, the participant commented that, because a bug had been tagged with a specific tag, to him it represented multiple concerns, when he stated:

> "I know more about the issue because it says that. I know that it's a [customer] issue, which means it came from specific [internal stakeholders], and it's probably related to [code] changes that they want, which we're not working on right now, so that would impact my decision."

A single tag became shorthand for that complex assessment process, and it could be used by the participant through the triaging process.

In many instances, these concerns took on nicknames that were used throughout the discussion. Rather than being concerned about which individual bugs were being assigned to specific users, the triagers began referring to higher level concepts like *security bugs* or *messy bugs*. Vague terms that, when defined as a tag set, take on a specific meaning. In another session, the participant listed criteria he would use to identify what he considered *embarrassing* bugs, that is, those that users should never see, when he stated:

> "The other thing I look for besides stack trace for the web application is I often see someone say like 'Internal Server Error 500.' And so when they paste server an 'Internal Server Error 500,' that's also something I automatically say 'this should never appear in the application,' I don't care the reason, therefore it's an issue that needs to be resolved. I would create a tag set for internal server errors... that's the most embarrassing thing."

This tag set would then become a proxy for the embarrassing bugs, and could be used throughout the triaging process to ensure that they are assigned and properly resolved in a reasonable time frame. One participant used a tag set based on the assigned developer and associated labels to identify bugs that were important for a specific customer. He stated:

> “Neat. Yeah, so these are basically ones that matter for [customer], so yeah, these would be important ones if they’re still open. Yeah, this is useful. I didn’t really have a way of doing this before.”

Front-loading decision making. The idea that tag sets become proxies for concerns, represented by specific criteria, allowed triagers to front-load their triaging workflow. This meant that they could perform the work of identifying the interesting or relevant bugs first, and then work with the bugs as sets to triage. Once the bugs have been tagged as part of a tag set, they can be assigned or modified together, and each one no longer needs to be considered individually. The advantage is that this approach to triaging, though it can take some time, will only need to be done once during the planning process, saving time later on. If the bug is in the tag set, the triager has already made some decision about the quality of the bug, and can assume that attribute when assigning it.

In one example, the participant used a tag set to represent a theme for a release. They identified the bugs that contained the word “ZIP code.” Once the bugs that matched this theme were identified, either by executing a BTL statement, or by manually creating ad hoc tag sets, the resulting tag set could be considered shorthand for that theme. This allowed the triager to work with more abstract tag sets, as one participant demonstrated by selecting all of the bugs contained in a tag set and assigning them a future milestone.

Layering tag sets. Another theme that emerged from our observations was that tag sets could serve as *layers* of information that could be enabled or disabled to reveal interesting properties about the bugs in each layer. Because tag sets can represent concerns the triager has, overlaying the tag sets in the bug list view is a way of overlaying the different concerns as they explore the bugs, and they can use the layers to identify bugs that are the most important for them. We designed the bug list in PorchLight such that, if multiple tag sets are enabled in the tag set list, and the *Has Active Tag* filter is enabled, the bug list will display the bugs that belong to the *union* of all of the tag sets. This design was intentional to allow triagers to bring multiple tag sets into a single view. However, because the ability to sort the bug list based on the number of associated tag sets was not available, the participants were unable to easily view the bugs which belonged to the most number of tag sets, which would be the ones associated with the most number of concerns.

One of the participants noted that a useful variation would be a mode that allows him to enable specific tag sets and view their *intersection* in the tag set list. In one of the sessions, he wanted the ability to create a "base list" of bugs using a tag set, and then build on top of that by enabling additional tag sets which represent different criteria (such as *COMMENTED* or *VOTES*). As new tag sets are enabled, the bug list would contain all of the bugs from the original tag set, in addition to the intersection of the original tag set and the newly enabled tag set. Both participants in one of the sessions agreed when one suggested that: "Once I have [the base list], I want to overlay additional information that doesn't change the list but just provides me more information at a glance." This functionality would eliminate the need to sort the bug list and rely on the tag set indicators to determine which bugs belong to all of the enabled tag sets.

These observations suggest that the participants began to adopt the idea of tag sets as a construct that would allow them to simplify the triaging process. They wanted to use tag sets to create a triaging list starting with a base set, and then build upon it using tag sets of increasing importance to identify ones that should be assigned. Each tag set provides a different lens on the original set, which can inform the triaging decision that is made.

Creating a work queue with tag sets. One recurring theme across all sessions was the need for an inbox-style view of the bugs that needed to be triaged. The participants wanted the ability to review the details of a bug, and then mark it as having been viewed, even if no assignment action was made. This would allow users to track their progress while triaging. The key question here is: how can such an inbox bug list be modeled and populated? This is where tag sets are useful. Without tag sets, the inbox would be the entire list of bugs, or an arbitrary selection, like just the most recent ones. With tag sets, however, it becomes possible to define criteria that identify relevant bugs, and populate the work queue with those bugs. One of the participants shares this belief, stating:

> "Basically, the tag sets would allow me build, I think, a work queue. Or, *the criteria to include things that I want in a work queue.*"

What emerged from our observations is a two step model of triaging. The first step is creating tag sets. This can be done either using BTL or manually using ad hoc tag sets. Each tag set is associated with some concern and has an implicit priority. The purpose of this step is to identify the bug and add metadata, in the form of tags, without the pressure of having to make a resource planning decision. This process of identification and refinement can happen over multiple iterations, as bugs are tossed between developers in the project, or between the triagers and the reporter, to provide additional information.

The second step is to then triage, or "work the list," and to perform the actual assignment of bugs to developers and milestones. This step can be done more easily once the inbox bug list has been identified since the triager is assured that these bugs have been vetted and contain the relevant information. In other words, there's nothing missing that would prevent them from making an informed triaging decision.

In this model, tag sets are not necessarily used while triaging, but rather *in preparation for* triaging. The time spent up front to create these tag sets and to identify the relevant bugs, whether that means important ones or ones that need cleanup, is time well spent if the end result is a refined work list that can be more easily reviewed and triaged. The tag sets provide crucial context to the process of building the work list, which offloads having to maintain the complex mental model from the triager.

Release planning with tag sets. Another observation that we made is that tag sets can also play a role in release planning. During one of the sessions in which we discussed creating layers of tag sets, one of the participants noted that there were three distinct modes during the development process during which triaging was done.

The first is triaging issues throughout the development process. Community members, QA, and other developers create new bug reports that are added to the bug tracker. As development leads, they receive email notifications about the new bug reports and, if they have time, will often make some triaging decision at that moment. The triaging decisions made during this process are typically simple and involve either requesting additional information from the reporter, or sometimes even assigning it to a future release, typically using the milestone as a placeholder so that it can be triaged at a later time. This real-time triaging is possible in situations where the volume of new bug reports is manageable, and the triagers can make a decision to "throw it into a version" without impacting the development process.

The second is when approaching the end of development for a version. The development leads meet specifically to review the bugs that are still open and decided if they can be resolved within the time remaining, or if they should be pushed out to future releases. Depending on the length of the release cycle, this checkpoint triaging session can happen up to 2 months prior to the release. The best set for this triaging session would contain the bugs assigned to the current release, and additional layers of tag sets would be applied to determine which of the remaining ones should be kept in the release. For example, the number of votes or comments from the community could be a factor in deciding if a bug should be pushed out, or kept in the release.

The third is when planning for the next major release. As new bug reports come in and are initially triaged, they may find themselves distributed among multiple future releases. At regular points during the development process the team leads meet and review issues, based on themes or other criteria discussed earlier, and assign bugs to the next set of releases.

The base list for this triaging session would contain bugs that have been added to the next several releases. For example, if the last release was for version 3.3.0, the following tag sets may be used to identify bugs that will be assigned to 3.4.0:

- Bugs that are currently assigned to 3.4.0
- Bugs that are currently assigned to 3.5.0, 3.6.0, or 3.7.0
- Bugs that are related to a pre-determined theme for the 3.4.0 release
- Bugs that not assigned to a release and that describe a security related defect
- Bugs that have received a specific number of comments or votes from the community
- Bugs that have been commented on by specific users in the community
- Bugs that have been reported by specific users

Different combinations of these tag sets would allow the triagers to successively identify bugs of importance to make the most informed triaging decision.

The participants also commented that the tag sets could potentially be used during these checkpoints to create the base lists and enable additional tag sets to identify bugs. In this model, milestones are not used as definitive assignments but rather as temporary buckets, and there is a periodic review at the beginning and end of the development lifecycle to assess if the triaging decision made as a real-time triaging decision, or during the early planning phases, still makes sense.

Modeling workflow transitions using tag sets. We also noted that participants discussed not only how tag sets could be used to groups bugs, but how bugs transitioned between the different groups. In one of the sessions, the participant presented his mental model of how tag sets could be used to create the different "pools of bugs" that could be drawn from at various points for release planning. Using a *body of water* analogy, the participant outlined his process for triaging defects. The *ocean* represented bug reports before they were created in the bug tracker. In some situations, bugs begin as support tickets from customers, in others they originate from conference calls or internal meetings during which a request for an enhancement is made.

After the bugs are created in the bug tracker, they "wash ashore" from the ocean onto the *beach*. The beach represents volume: a large number of bugs are created while development is ongoing and they are collecting on the shore. Bugs that have been "washed off" by the triager, in this case the product manager, and have enough information to potentially be assigned to a milestone, are moved into the *pool*. Once in the pool, bugs move into the *deep end* after they have been prioritized based on various factors, like internal or external demand. Finally, from the deep end the bugs move into the *hot tub*, which represents bugs that have been assigned to developer and a development sprint, and are actively being worked on. During each transition, the bugs are refined in preparation for further triage, which can involve other members of the team, as the participant stated:

> "I would present all of the hot tub bugs to [the lead product manager], and we'll talk through it. There's a triage level there, too, but she doesn't care about the three levels before that."

While this model of a triaging workflow by itself is interesting, our observation is that tag sets can be used to facilitate the *transition* between these different states. For example, the *CLEANUP* tag set that was created during the session could be used to identify the bugs that "wash up on the beach," that is, bugs lacking basic information, like a long enough description, a screenshot, or steps to reproduce the problem.

```
TAG "CLEANUP" WHERE HAS(screenshot) = false AND FREQUENCY(comments) = 0 and
   FREQUENCY(labels) = 0
```

Once identified as part of this tag set, a bug can be refined with additional information so that it can be transitioned into the pool, or it can “thrown back” onto the beach for a multitude of reasons. Another participant commented on these types of bugs when he stated:

> “It would be nice to create a tag set for ‘don’t care right now, we’ll look at this later’ [bugs]. Like the next major version, without actually putting it in the version for now.”

Once a bug has made it into the pool, additional information is needed to transition it into the deep end. Labels, fix versions, watchers, and comments can be factors that influence if a bug works its way into the deep end where it is a candidate for an upcoming development sprint. While bugs should not typically “jump right into the hot tub,” there are factors like the priority of the ticket, or if it concerns a security related defect, that can move it there immediately.

Tag sets as meta-states for collaboration. Another observation we made is that tag sets can be used to facilitate collaboration between different members of the team. Some ad hoc tag sets created during the session, like *TO BE VERIFIED*, identified bugs that needed to be seen by someone other that the triager. Rather than assigning it to that person, the participant began to think about the tag sets as a way to identify these bugs, and then allow another person to use that bug list as their inbox.

Additionally, tag sets can be used to create a *highly adaptable layer of metadata* that is not restricted by the rules of the bug workflow, and thus can be adapted to the needs of different projects. For example, both the product manager and the development manager used tag sets to create lists of bugs that needed further review by a team mate. Rather than setting that person as the assignee, which would throw off the project workload, tagging is used to define a new *meta state* and tag sets can be used to manage these meta states. For example, the development manager proposed a tag set that he would recommend his development team to use to keep track of bugs that he has commented on:

> "If I see [a bug] that comes through and catches my eye and concerns me, I comment on it. So, I would I think a filter that [my development team] could use would be to look at anything that I've commented on in the last 30 days. It really has to pique my interest for some reason for me to comment on an issue now."

Triaging is a context-dependent activity. A final observation that we made is that triaging is a highly context-dependent activity. Looking strictly at the tag sets that were created during each session, we can see that no two sessions make use of the same collection of tag sets, and each session makes use of different features of both PorchLight and the tagging language. We see that the participants not only adopted tag sets as a way of understanding and working with collections of bugs, but they adapted the approach to their specific context and use cases.

Where we might have expected to see a standard set of tags reflecting a defined group of concerns emerge from the sessions, shared by all of the triagers, our observations show that the types of tag sets created reflected many dimensions. The approach to triaging varied based on the individual, on their role in the project, on the type of project (open source versus commercial). This uniqueness led to the variety of tag sets that were created. For example, the participants working on an open source project created tag sets that used the number of votes from the community as a criteria. On the other hand, participants working on commercial product used criteria like the number of watchers and comments rather than votes.

Summary

From our discussion, it should be clear that our participants all transitioned from thinking about individual bugs to thinking about them in sets. We observed numerous possible uses of tag sets, from proxies for concerns, to an approach to creating a bug inbox, to metadata that could be used to model workflows. Each of the participants discussed these applications of tag sets in a natural way, suggesting that they had adopted the approach of thinking about bugs in tag sets, if not entirely the tool for creating them. Given that, we believe this analysis begins to show the possible usefulness of PorchLight in presenting bugs in a way that triagers can begin to explore and manage them using tag sets.

We also note that the themes we have observed could not have emerged if our observation had focused on the individual bugs being triaged. The conversations around proxies for importance and transitions between "pools of bugs" were only possible as the participants were able to reify their mental models of their triaging process using tag sets.

6.3.3 Making Triaging Better

In the previous analysis, we identified ways in which triagers began to think about bugs in terms of sets. A subject of future work clearly is to determine if conclusive evidence can be obtained on the impact of tag sets on the triaging process. In this final analysis, we present some observations in this regard, and look at what it means to work with bugs in sets rather than individually in terms of what might eventually be more quantifiable criteria.

We chose a set of six criteria to initially explore. Some of the criteria are more objective. For example, does improving the triaging process make it *faster*, meaning triagers spend less time performing the assignment action? Does it decrease the *volume* of bugs that need to be triaged in the first place by grouping them? Or, does it improve the *accuracy* of the assignments to the developers and milestones, so that there are fewer reassignments made later on?

Other criteria are more subjective, but nonetheless indicate an improvement to the triaging process. Does improving the triaging process result in assignments to milestones being more evenly distributed, producing a more *balanced* development roadmap? Does it cause triaging to find the *relevant* bugs that can have a bigger impact during triage? Does it result in fewer bugs "slipping through the cracks," giving triagers *confidence* that they are catching more of them before they are lost?

Criteria	**Question**
Efficiency	Does the approach reduce the time needed to perform triaging?
Volume	Does the approach reduce the number of bugs that should be triaged?
Accuracy	Does the approach improve the accuracy of the triaging assignment?
Balance	Does the approach results in a more balanced distribution of bugs across milestones?
Relevancy	Does the approach result in more relevant bugs being found during triaging?
Confidence	Does the approach results in fewer bugs being lost or forgotten?

Table 6.7: Criteria for analysis.

Other criteria exist as well. One could imagine assessing improvement by measuring the number of related bugs were triaged together, or by the average time between when a bug is created and it is eventually assigned. In this analysis however, we focus on the six we introduced above, and summarize in Table 6.7. Figure 6.5 plots the relative subjectivity and objectivity of the selected criteria. Again, we perform our our analysis knowing that conclusive evidence is not possible through our study. Rather, we seek to find supporting patterns in the conversations and observations to guide future detailed studies in where to focus.

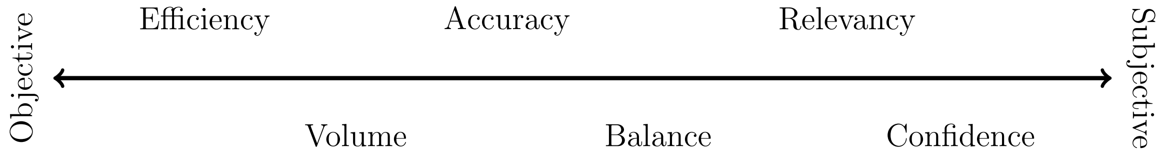


Figure 6.5: Spectrum of analysis criteria.

If we look at the narrative from the participant observation sessions, we can see early supporting evidence for improvement across all six of the criteria. We observed use of the tool, and noted comments made and thoughts shared by the participants, that seemed to indicate the PorchLight has indeed improved the triaging process across multiple dimensions. In this analysis, we discuss each of the criteria and the supporting evidence. We group several of the criteria in instances where the evidence we identified supported both.

Efficiency and Volume. We can look to the comments made by the participants, particularly on the usefulness of the layout and the assignment functionality, as evidence that the features implemented in PorchLight do indeed provide notable improvements to the efficiency of triaging. For example, by not having to constantly switch between tabs with individual bugs, or quickly navigating through the comments for a bug, or by being able to drag-and-drop a bug immediately on a user or milestone, we believe that triagers will be more efficient when triaging with PorchLight. From our observations, and from the comments by the participants, we can *tentatively* confirm that it might be faster and more efficient. During one of the sessions, one of the participants commented on the use of ad hoc tag sets to organize bugs prior to triaging, stating:

> "That's cool because I don't have to stress about how we're going to verify it. *I can just put it in the tag set and deal with it later.*"

We can look at how participants used tag sets as proxies for concerns to support an improvement in the volume of bugs during triaging. Tag sets provided an abstraction mechanism that was used to front-load the decision making process. Initially, the number of bugs that the triager reviewed may have been larger, since their use of tag sets helped to identify bugs that would have previously not been reviewed. However, once the bugs were grouped into tag sets based on concerns, the overall volume seems to be less because the participants are working with *meaningful groups of bugs* rather than individual bugs during the assignment process. One participant supported this finding when he stated:

> "Instead of worrying about *a thousand bugs*, I get to worry about a couple of *hundred tag sets*. And so I've reduced my whole workload down to just the tag sets. I'm making the problem smaller."

We also observed extensive use of the multi-select feature to assign bugs from the bug list as a group to users or milestones. When combined with tag sets, this relatively simple feature had an impact on both the efficiency and volume of triaging. Tag sets allowed the triagers to identify and organize bugs up front during the process, and the multi-select capability allowed them to quickly perform an assignment action on all or many of the identified bugs. One participant, after using the feature, commented:

> "Yes, [this is useful] because *I do it all the time.*"

Accuracy and Balance. We also observed that tag sets probably improve the process of release planning, which is a specific type of triaging that occurs at various points during development. We saw several examples of how tag sets helped triagers achieve a more balanced distribution of bugs across milestones by identifying bugs that could be redistributed to future milestones. For instance, the participants working on the open source project presented an approach to release planning that involved reviewing the bugs assigned to the current release at multiple points during the development sprint, and they created tag sets to help identify ones that could be "pushed out" to future milestones. Other participants saw the ability to use ad hoc tag sets to create temporary milestones as a way to prevent future milestones from being overloaded.

We believe that tag sets also helped triagers to perform more accurate assignments to milestones. By allowing triagers to create layers of tag sets, PorchLight enabled a more precise approach to identifying bugs belonging to themes for a release. Rather than having to identify the bugs belonging to a theme one-by-one as they come across them, tag sets allow the triager to identify related bugs based on criteria and assign them to the appropriate milestone. Additionally, tag sets helped one participant to identify bugs that should have been triaged but were not, allowing him to more accurately assign them to the appropriate milestone. Another participant observed that tag sets could be used to identify bugs that belong to a theme for release planning when he stated:

> "There are groups [of bugs] based on urgency of customers asking for it, or groups based on themes, like architecture [bugs]. And so, the theme of this release is customer facing, *let's look at that tag set and see what's in that pool and see what we should fix.*"

Similarly, another participant commented on his approach to identifying bugs for release planning when he stated:

> "I would probably create a couple of tag sets and layer them on top of each other."

One aspect of improving accuracy that we do not have enough evidence for in our observations is the accuracy of assignments to developers. PorchLight prominently displays the list of users along with the number of bugs that have been assigned to them. It also makes important information about the bug, including the comments made by stakeholders, readily available. While we assume that access to these features would support the triager in making better and more accurate assignments to developers, there were not enough assignments of bugs to developers or feedback on that aspect to suggest an improvement.

Relevancy. We observed that tag sets became proxies for concerns throughout the triaging process. The concerns brought to bear on the process by the triagers are the criteria which determine relevant bugs. By providing a mechanism which can embody these concerns, tag sets allow triagers to attach relevancy to a list of bugs. Making it easier to identify the bugs that have attracted attention, or have significant activity, increases the relevance of the triaging actions being performed. Participants made reference to "interesting" or "embarrassing" bugs that were more relevant and could be more easily identified and triaged by using tag sets. For example, one participant commented on the use of tag sets to identify bugs that had a large number of watchers and comments:

> "If it has multiple watchers, it increases the amount of people that care about it. It means it should probably be looked it ... if six people care about this, *then there's got to be something there, let's figure out what that thing is...* If there are a lot of watcher and a lot of comments, I think of the ticket that's a really hard problem to solve, and is causing customer issues."

Confidence. Finally, we observed that tag sets seem to increase the triager's confidence in the triaging process by providing a way to *more thoroughly* explore and organize the bugs. Because tag sets, once specified, can be persisted and reused through the development process, it provides triagers a way to quickly review and assess groups of bugs to prevent them from slipping through the cracks. One participant stated that:

> "I think I really did find some that kind of slipped through the cracks. Keeping in mind, this is already a managed project, I mean, *if I had been using* PorchLight *the whole time it would have made it easier in the first place.* I already thought I had gone through everything, so those [that I just triaged] slipped through the cracks."

This is particularly important in projects that have a large number of bugs that need to be reviewed and triaged regularly. Even with a development process in place, meetings dedicated to reviewing the backlog of bugs that need to be assigned, there is still the potential that some can be forgotten. This can be made worse when users of the bug tracker use different mechanisms to temporarily categorize bugs. One participant commented on the usefulness of tag sets in identifying bugs that have been dispersed across multiple impermanent versions, components, and assignees in the the project:

> "Yeah, definitely. What I usually use for that is *version*, which is probably why we have so many weird versions. We have *Holding*, we have *Backlog*, we have *Alice's Enhancements*, *Alice's Backlog.* [Laughs]. She created her own because she wanted her own little buckets to play with. So, yes, [tag sets] would help."

Summary

From these observations, together with the features we discussed in Section 6.1 and the kinds of tag sets we presented in Section 6.2, we have some early verbal indication that PorchLight seems to have a positive impact on the quality of triaging along all six of the criteria we selected for our analysis.

The basic features of PorchLight, like the activity timeline and bug multi-select, provided triagers with access to the information they need to identify bugs that need triage, and once identified, the ability to quickly assign them. Tag sets contributed to making the process more efficient by allowing triagers to front-load their workflow. By spending more time early on to organize the bugs, they felt they were able to more **efficiently** perform assignments. This also impacted the **volume** of bugs that needed to be triaged, though indirectly. With tag sets, participants were able to transition from having to make successive small triaging decisions about individual bugs, to making larger decisions about how to triage sets of bugs.

Once the participants became comfortable with the tool, they began to think about bugs in terms of tag sets, and they began to explore ways in which their triaging process could be modeled and improved using this concept. The study participants were triagers with extensive experience and intimate knowledge of the projects and their bugs, yet with a few tag sets specified using BTL statements, they were able to quickly identify bugs that would otherwise have been lost. This resulted in an increase in their **confidence** that the bugs that needed to be identified and triaged would be. On several occasions we observed the participants' surprise as they came across bugs that they believed should have already been triaged, and their reassurance that our approach would have helped them identify those bugs sooner.

Our approach also had an impact on identifying **relevant** bugs during the triaging process. Tag sets provided a way for triagers to express and proxy specific concerns using criteria. Some of the concerns could be expressed more simply, like looking for specific keywords in the description, or for labels with customer names. Other concerns were more abstract and made use of the more advanced features in our approach. For instance, looking at the level of activity on a bug to make sure the most active ones receive attention, or assessing the importance of a bug based on the number of watchers, or even based on *who* the specific watchers are. These concerns are crucial to the triaging process, and our approach provides a way for triagers to express them, and then monitor the bugs that are most relevant.

We also identified likely improvements to the process with regards to the other assessment criteria, though we do not believe it was as significant. Tag sets assisted triagers in performing more **accurate** assignments to milestones by provided them with a way to create ad hoc groups that could be used as temporary placeholders that they felt were important. Assignments to specific milestones then were more accurate by definition, since triagers had confidence that the bugs that would need to eventually be assigned could be found later on using tag sets.

This also prevents triagers from overloading and cluttering actual milestones that are shared by the entire project, and avoids skewing the workload distribution across milestones, thus leading to a better **balance**, or distribution of bugs across milestones. Tag sets and multi-select also provided triagers with a way to, at various points during the development process, identify bugs within milestones and then easily "push out" or redistribute the bugs across future milestones as necessary.

We did not, however, observe a similar improvement to the accuracy of assignments when it comes to the developer. This may be due to the fact that, from our observations, the assignment to the developer and milestones did not happen simultaneously. Rather, the milestone must be determined first and only when the development sprint for the milestone approaches is the developer assigned. We, thus, saw fewer developer assignments than milestone assignments.

Stepping back, our approach provided a model that triagers used to become aware of and effectuate their *underlying process* which, until now, they have been unable to adequately express. Tag sets provided a way for triagers to accomplish things that it appears they have long wanted to do, but could not, or could only do in an inefficient manner using their existing techniques. We see this observation as an important outcome, and the most significant improvement to the triaging process that was enabled by our approach.

6.4 Weaknesses

There were several weaknesses in our implementation of PorchLight and how it allowed the participants to work with sets of bugs. We describe these weaknesses in two areas: weaknesses with our concept and approach, and weaknesses with our technical implementation.

6.4.1 Conceptual

A weakness that we observed during the participant observation study was in how we chose to present the relationship between the tag sets and the bugs. One recurring request was for the ability to view the intersection of tag sets in order to progressively reveal the bugs which were the most relevant based on the criteria defined in each tag set. Our use of tag set indicators in the bug list, and the ability to selectively enable tag sets, was a small step in this direction, but we think the use and comments by the participants revealed a weakness in our conceptual approach to visualizing tag sets. More than just overlaying the tag sets, the participants wanted a way to *see* the overlap. They wanted the criteria for each tag set to be visible, and as more tag sets were enabled, they wanted to be able to visually identify the concerns being expressed by the tag sets, and see the increasing relevancy of the bugs. A complementary view (though not a replacement, since we believe the bug list has other strengths) of the tag sets and bugs which emphasizes the overlapping nature of the tag sets may address this weakness.

We observe that our approach failed to support some tag sets that participants requested, and that the bug data model and tagging language we developed as part of our approach was not sufficiently complete when we started. This was evident in the number of iterations the researcher had to undergo when creating BTL statement during the sessions, or in the various plugins that were implemented to extend the language. Even though the changes needed to address many of these limitations were small in scope, requiring only a few dozen lines of new code in some cases, it does suggest that there may be other weaknesses in the tagging language that have yet to be uncovered.

We also observe that with our tagging language it was difficult to express some of the more *relative* criteria that were used by the participants. For example, the desire to identify bugs that were “unassigned to me” or “unassigned to the next milestone” was present in the observations. The notion of “me” and “next milestone” are not easily expressed without explicitly identifying those parameters in the BTL statement. One significant improvement would be to automatically extract this metadata from the bug tracker, or other sources, and make it available in the tagging language. Because of our approach to implementing the tagging language, such an improvement would be straightforward.

Another weakness that we observed was in the high-level feature set we ultimately chose to implement for PorchLight. While the features we implemented were sufficient for the purposes of the study, there were numerous that we did not implement that would have addressed several of the requests from the participants. Specifically, we could have explored the concept of a more feature rich environment for triagers that would have allowed the creation of temporary work queues based on tag sets. While possible, the triagers would need to do much of this work themselves, which is not ideal. Another concept we could have explored was incorporating duplicate detection techniques into BTL to allow triagers to create bugs sets which identified similar bugs automatically. While we did not implement these features, we feel like they would need to be an important part of a production version of PorchLight.

6.4.2 Technical

One weakness of our technical implementation of PorchLight was in the integration between the server and the source bug tracker. For example, while the BTL evaluation process described in Section 5.2.2 allowed for bugs to be tagged as part of the evaluation of a BTL statement, we did not implement a mechanism for automatically keeping the tag sets up to date if new bugs were added that met the specified criteria. For example, if a tag set includes criteria for identifying bugs that have been assigned to a specific developer, and the assignee is changed, the tag will not automatically be removed and, therefore, the tag set will no longer accurately reflect that state of the bug tracker. This functionality was omitted from the prototype since we did not think it was necessary for assessing our approach. More specifically, we were able to receive sufficient feedback on the use of tag sets without this ability. However, it was brought up in all of the sessions as the participants asked how the tag sets were synchronized with the source bug tracker.

Another weakness of our implementation that emerged was the inability to sort the bug list based on the tag set membership column. In hindsight, this should have been a basic feature, but it was not implemented due to technical limitations of the Java library used to render the table cell. As discussed earlier in this chapter, multiple participants requested the ability to sort the bug list based on the number of tag sets to be able to visualize the intersection between multiple tag sets.

Another weakness was in the limited number of actions that were displayed in the timeline. Numerous additional activity types could have been shown in the timeline, such as votes or field changes. However, we only implemented a subset of activities against the bug tracker, specifically comments being added and certain status changes. While we feel this was sufficient to convey the purpose of the timeline, and it still saw heavy use during the participant observation study, we could have explored the information that can be conveyed through the timeline more thoroughly if there were more actions displayed.

A final and more minor weakness that emerged was in the way we chose to indicate assignment to a user or milestone in the respective lists. While browsing through the bug list, the assignee and milestone for a selected bug were indicated by a green checkmark on the associated icon in the list. While this did provide an indication of assignment, a consistent point of feedback was that the indicator was not clear enough and should either be larger, or the icon of the user or milestone itself should be changed to indicate assignment. To make the assignment to a milestone more clear and to reduce visual clutter, we also should have provided the ability to toggle the milestones that appear in the list to just those that have not yet been released.

6.5 Threats to Validity

Several threats to both the internal and external validity of our analysis exist.

6.5.1 Internal Validity

The participants in both the preliminary user study and the participant observation study included professional colleagues of the researcher at the time of the study, which took place on site at the place of employment. The participants may have been *favorably biased* toward the tool because of this relationship and setting. Though this threat exists, we do not believe a strong favorable bias exists since, along with positive feedback, we also received numerous requests for improvements, both to the PorchLight user interface and the underlying tagging language. Furthermore, the fact that one of the participants began using the tool to perform actual triaging suggests that he found it useful for his actual work and was not just providing positive feedback due to a bias.

The researcher's high level of involvement during the participatory observation may have introduced an *experimenter bias*, since influences from the discussion and suggestions made during the sessions may have led to certain results. Particularly, the openness with which the researcher shared and discussed observations with the participants, and made suggestions for tag sets during the participant observation study, may have caused the participants to come to certain conclusions about the usefulness of the approach. The researcher was careful in maintaining objectivity and not leading the discussion, but this threat remains.

6.5.2 External Validity

While a participant observation study allowed us to situate our approach in a real-world setting with professionals reviewing actual bugs from active projects they were intimately familiar with, the participants were not observed performing actual triaging in an actual triaging meeting with their team, as the setting was more discussion oriented in its setup. It is possible that the triaging tasks performed do not adequately represent the challenges participants would normally encounter while triaging. Under these limitations, this is still not a conclusive study. Rather it is an exploratory study of the use of the tool that has let to a number of rich observations. It is precisely the depth of these observations, however, that we believe is evidence of this threat being minimal. Participants clearly projected their triaging roles during the sessions and actively engaged in meaningful discussion about bugs, tag sets, and their approach to triaging.

Because the study was conducted in a single organization with a relatively small number of participants, the generalizability across situations and people may be questionable. Our findings may be localized to the single organization in which the study was conducted, or it may be specific to the individuals that participated in the study. While we tried to mitigate this threat by including participants with different roles working on different projects within the organization, the workflows and models revealed during the sessions may not apply to triagers elsewhere. On the other hand, the observations being made echo those of the triagers we quoted in the beginning of tis dissertation, and the fact that different triaging approaches were readily supported, implies that PorchLight is more broadly applicable because of the versatility of tag sets.

Chapter 7

Conclusions

In the introduction to this dissertation we presented a volunteer triager who, out of frustration, resigned his role on a large open source project. He posted a message on his personal blog, stating:

> "Right now, there is no real way to triage except 'Here is a list of 1700 bugs. Start at the top and work your way down.' We need a way to mark bugs that need triage ... But we also need to remember, BMO [Bugzilla@Mozilla] is being used for things it never was created for."

The former contributor appealed to the triaging community:

> "We also need more coordination as a community so we can touch all these bugs in an effective and professional manner. Having better tools for the Triage community to find bugs that haven't been replied to within a certain time period. Having better communication between developers and triagers, and between triagers themselves."

If we look at the cause of his frustration, it was the lack of a tool that matches how they want to work. As a research community, however, we know precious little about triagers and their work. The primary purpose of this dissertation, then, is to develop more of an understanding of triaging, specifically through the creation of PorchLight, a new triaging tool based on the conjecture that working with tag sets better matches how triagers do their work.

We began this dissertation by examining more deeply the source of the triager's frustration. We informally surveyed the current state of triaging support in popular bug trackers and in the triaging community at large, and identified the problems that exist with the approaches. In particular, we examined how search filters and tags are used as makeshift solutions to address the problem of making large sets of bugs more manageable.

We then presented current research related to bug tracking. We investigated the most active areas of research, which included duplicate detection, automated assignment, and field studies. We learned that there are a variety of techniques that can be deployed to improve the accuracy of assignments, or to reduce the number of bugs that must be triaged by identifying duplications. From previous field studies we confirmed that triaging is an important activity that occurs in both open source and commercial software development, and that there are many roles that the bug tracker takes on in the development lifecycle. Based on our survey of the current state of the art, we proposed the need for a *dedicated triaging environment* based on sets of bugs in order to better match the needs of the triager. We outlined a set of requirements for how working with sets of bugs should be reflected in a tool along the dimensions of exploring, searching, inspecting, and taking action.

We implemented these ideas in PorchLight, particularly through the use of tag sets and a specialized tagging language that, when used through the triaging interface, would be useful in creating meaningful sets of bugs. We also described the capabilities of the environment which would make the triaging actions, like commenting and performing assignments, more efficient. We also described the implementation details behind our approach, noting the design and technical decisions we made along the way. This included details of our plugin architecture which allows others to extend our approach.

Finally, we performed a multi-part analysis consisting of two studies and an in-depth analysis of, on one hand, the feasibility and usability of PorchLight, and on the other hand, the impact of using tag sets as the primary mechanism for organizing and working with large sets of bugs. We summarized the findings from the preliminary user study which we conducted to validate our general direction. We also provided a narrative overview of a participant observation study conducted with four participants across six sessions. We then presented an analysis of the observations from both studies across six assessment criteria, and highlighted key areas in which our approach had an impact on improving the triaging process.

Overall, through the studies we conducted, we were able to confirm our conjecture that working with tag sets better matches how triagers do their work. We now also know more about the process of triaging than we knew at the beginning of this dissertation, and we uncovered numerous findings about triaging and the needs of triagers when they work. Moreover, we know that it is possible to build a dedicated triaging environment, based on the concept of tag sets, to meet these needs. Below, we first focus the discussion on the feasibility of a dedicated triaging environment, and then return to the overarching purpose of this dissertation, which was the idea that allowing triagers to work with and organize bugs in sets better matches what they want to do when they triage.

In terms of feasibility, we first note that we could design and implement the environment, and that at one level, the basic features of the tool, including tag sets and BTL, were used. The participants created a variety of tag sets in their exploration of the concept, and prompted the addition of new features through plugins. At another level, we see that the participants began thinking about the bugs differently, and began modeling their latent triaging process using tag sets. At the highest level, we see through the feedback from the participants that both tag sets and PorchLight appeared to have an impact on improving the quality of the triaging experience.

We particularly learned that tag sets are a useful model for *thinking about bugs*. More specifically, they are useful for *breaking down* a large collection of bugs into smaller sets that are more meaningful to the triager. They can be used as proxies for specific concerns, or as a way to temporarily represent the state of a bug in a triaging workflow. Additionally, tagging itself has proven to be a useful technique by which to associate the name of a set with the bugs contained in it. Because most bug trackers support tags or labels, the tag sets created through PorchLight can be reflected in the source bug tracker, tying the concepts that were explored in our study back to a larger process that potentially involves not only triagers, but the entire development team.

While triaging is a common activity across software development projects, we learned that the way in which it is performed can be unique to each project and individual. This is partially due to the fact that so much of the decision making process is dependent on the context in which the decision is made. There is an entire set of experience and historical knowledge that is brought to bear on the decision making process that cannot yet be reflected in any tool or bug tracker. For example, knowledge about the history of a bug as related to a particular developer, or the theme of a milestone based on an understood agreement within a small development team, are intangible factors that, as one participant in the study mentioned, places the act of triaging somewhere "between an art form and a science."

We learned that, while we initially set out to better support the work that triagers do, we ended up with a realization that we still know less than we had anticipated about how triaging is fundamentally done. While at the outset we were interested in the mechanics of how often assignments are performed, or how often bugs are tossed between assignees, we soon discovered that triaging is not a well defined process. Even within the same organization, individuals approach triaging differently, based on their goals and the information they have available to them. Beyond the context are the habits and practices that the triagers have adopted over years of working with the bug tracker and assigning bugs. We caught a glimpse of this through the models that were discussed, but there is certainly more that should be explored in that area.

Tag sets were also a useful tool for revealing, modeling, and enabling the *latent workflows* used by each individual triager. Tag sets provided the triagers a structure, both for describing their triaging process, and for the types of bugs that are important to identify and review during this process. During the course of the participant observation sessions, participants began to reflect on their process, identifying the possible limitations in their approach, and the gaps in the workflow where bugs could be lost. This is a significant finding, since PorchLight allows triagers to express and, to some extent, apply the context that is important during the triaging process. The variety of tag sets that we observed in our participant observation study support the observation of bug triaging being idiosyncratic in nature.

Finally, we conclude with the observation that, at the end of the day, some of the participants began using PorchLight to perform *actual* triaging during the study sessions. We should not overlook the importance of this, especially given the context of this action. The participants were being asked to consider the concept of tag sets, and to assess the usefulness of a tool using a set of bugs they were familiar with. Without our prompting, some found both the concept and the tool useful enough to take the initiative and close the gap between a research evaluation and performing actual triaging for their work. We believe that this simple observation is the most direct confirmation of the impact of our work.

Chapter 8

Future Work

In this section, we describe potential future directions for this work, as broken down into three categories: conceptual, further studies, and technical improvements.

8.1 Conceptual

Automating contextual tag set creation. We believe there is strong potential for future work to build on the concepts presented in this research that incorporates different aspects of improving triaging into our approach. For example, the ongoing research in machine learning and automation techniques could be incorporated into both PorchLight and the bug tagging language. We can envision a version of our approach where, as bugs are created, they are automatically assigned to a tag set based not only on criteria defined by the triager, but new criteria that are discovered by the tool based on an analysis of the entire set of bugs and based on learning how triagers handle different kinds of bugs. Similarly, state of the art duplicate bug detection techniques could be incorporated into the environment so that tag sets containing potential duplicates are automatically created and presented to the triager.

As a starting step in this direction, one could simply automate the process of categorizing bugs into tag sets based on their age and when they were last viewed by the triager.

Tailoring the approach to individual triagers. Now that we know that triaging is a context dependent activity that can be idiosyncratic in nature, we believe that both the tool and the tagging language can be improved to better adapt to and support the different mental models and workflows used by individuals and teams. One potential for work in this direction would be to allow PorchLight to automatically detect the workflow and model employed by a triager over some period of time, and then based on the patterns detected, create or suggest tag sets that could assist in the triaging process. Rather than providing triagers we a pre-defined set of "one size fits all" tag sets, this would allow the tool to provide targeted and highly specialized tag sets that would address the triager's specific needs. This has the potential to have a significant impact on how triagers work on a daily basis. As this may be difficult to achieve, a basic first step might be to package PorchLight with a set of predefined workflows that a triager or triaging team may choose to adopt.

8.2 Further Studies

Field deployment in a professional setting. Due to the inherent weaknesses of using a participant observation study, one direction for future work would be to deploy a more robust version of PorchLight in a professional setting. Observing the tool *in actual use* for an extended period of time would be an important extension of our research. Based on the feedback we received during the participant observation study, PorchLight could be used in a limited capacity to perform triaging during specific periods in the development lifecycle. It would not replace the bug tracker used, but rather supplement it as a dedicated triaging environment. An actual field deployment would ideally yield rich insights into how tag sets can impact the triager's workflow *over a period of time*, which would demonstrate the long-term feasibility of our approach.

Field deployment in an open source community. Because this research was partially motivated by the frustrations of triagers in the open source community, another direction of future work would be to deploy PorchLight for use within a large open source project. Deploying PorchLight to such a larger community would require enhancements that would lower the threshold to use and adoption. Additionally, usage tracking functionality could be incorporated to report the use of the tool, and how tag sets are employed. While this type of study would be more difficult to conduct, it would provide the most comprehensive view of how tag sets and the tool are being used to perform actual triaging in an authentic setting.

8.3 Technical

Support for additional bug trackers. A first need is to simply expand the number of bug trackers supported by PorchLight. While the architecture and data model were designed to support any bug tracker, implementing the importers and exporters to work against another popular bug tracker like Bugzilla would help to reveal any possible additional limitations in our bug data model, or in the capabilities of BTL. Additionally, it would expand the potential user base for the tool, and lower the barrier for performing additional studies.

Enhanced plugin capabilities. We discovered situations where the data model could not be easily modified using the plugin mechanism we had designed. For instance, the ability to view source code commit events in the timeline was not utilized since the projects used the Git version control system, while our approach had only implemented a Subversion adapter. Similarly, our ability to extend the data model was limited when the source of the information resided somewhere other than in the bug schema provided by the source bug tracker. Expanding the capabilities of the plugin architecture to allow for more complex extensions to the data model and to the functions would move the tool towards becoming a platform for future work to expand on. For example, we can envision custom functions that look to sources beyond the bug tracker to perform computations that can be used in the evaluation of BTL statements.

Platform for evaluating triaging techniques. Finally, with some enhancements to the plugin capabilities of PorchLight and BTL, one viable direction for future work would be to transform the tool into a platform for exploring, implementing, and evaluating different techniques for improving triaging. More specifically, PorchLight could become the reference environment in which techniques could be assessed. For example, the ongoing work related to assignment automation could be incorporated into BTL. Multiple tag sets could be created using different techniques, and participants in a study could use these tag sets to perform triaging and provide feedback on the accuracy or usefulness. The basic functionality in the user interface could be expanded to include ways to provide real-time recommendations based on the output of these techniques. Similarly, the research that is being done on duplicate detection could be explored through PorchLight as tag sets identifying similar bugs.

Bibliography

[1] Bugzilla. `http://www.bugzilla.org/`, 2010.

[2] FogBugz from Fog Creek Software. `http://www.fogcreek.com/fogbugz/`, 2010.

[3] The Trac Project. `http://trac.edgewall.org/`, 2010.

[4] MigLayout - Java Layout Manager for Swing, SWT and JavaFX 2! `http://miglayout.com`, 2013.

[5] Bugs Ahoy! `http://www.joshmatthews.net/bugsahoy/`, 2015.

[6] Eclipsepedia - CDT/Bugs. `https://wiki.eclipse.org/CDT/Bugs`, 2015.

[7] GitHub - Build software better, together. `https://github.com`, 2015.

[8] GNOME Wiki! - Bug Days. `https://wiki.gnome.org/Bugsquad/BugDays`, 2015.

[9] GNOME Wiki! - Finding bugs to triage. `https://wiki.gnome.org/Bugsquad/TriageGuide/FindingBugs`, 2015.

[10] GNOME Wiki! - Triage Guide. `https://wiki.gnome.org/Bugsquad/TriageGuide#Steps_of_Triaging`, 2015.

[11] Google Code. `https://code.google.com`, 2015.

[12] How we organize github issues: A simple styleguide for tagging. `http://bit.ly/1QaKLiB`, 2015.

[13] JIRA - Issue & Project Tracking Software. `https://www.atlassian.com/software/jira`, 2015.

[14] JIRA REST API Reference. `https://docs.atlassian.com/jira/REST/latest/`, 2015.

[15] Mozilla Developer Network - Bug Triage Day. `https://developer.mozilla.org/en-US/docs/Mozilla/QA/Bug_Triage_Day`, 2015.

[16] Mozilla Developer Network - Triaging crash bugs. `https://developer.mozilla.org/en-US/docs/Triaging_crash_bugs`, 2015.

[17] Neo4j, the World's Leading Graph Database. `http://neo4j.com`, 2015.

[18] Semantic Versioning 2.0. `http://semver.org`, 2015.

[19] The Bugzilla Guide - Life Cycle of a Bug. `https://www.bugzilla.org/docs/4.4/en/html/lifecycle.html`, 2015.

[20] The Chromium Projects - Triage Best Practices. `http://www.chromium.org/for-testers/bug-reporting-guidelines/triage-best-practices`, 2015.

[21] The Chromium Projects - Triaging Bugs. `http://www.chromium.org/getting-involved/bug-triage`, 2015.

[22] Configuring Workflow: Workflow designer. `https://confluence.atlassian.com/jira/configuring-workflow-185729632.html`, 2016.

[23] M. Ames and M. Naaman. Why we tag: Motivations for annotation in mobile and online media. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '07, pages 971–980, New York, NY, USA, 2007. ACM.

[24] J. Anvik, L. Hiew, and G. C. Murphy. Who should fix this bug? In *Proceedings of the 28th International Conference on Software Engineering*, ICSE '06, pages 361–370, New York, NY, USA, 2006. ACM.

[25] J. Anvik and G. C. Murphy. Reducing the effort of bug report triage: Recommenders for development-oriented decisions. *ACM Trans. Softw. Eng. Methodol.*, 20(3):10:1–10:35, Aug. 2011.

[26] J. Aranda and G. Venolia. The secret life of bugs: Going past the errors and omissions in software repositories. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 298–308, Washington, DC, USA, 2009. IEEE Computer Society.

[27] D. Bertram, A. Voida, S. Greenberg, and R. Walker. Communication, collaboration, and bugs: The social nature of issue tracking in small, collocated teams. In *Proceedings of the 2010 ACM Conference on Computer Supported Cooperative Work*, CSCW '10, pages 291–300, New York, NY, USA, 2010. ACM.

[28] N. Bettenburg, S. Just, A. Schröter, C. Weiss, R. Premraj, and T. Zimmermann. What makes a good bug report? In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, SIGSOFT '08/FSE-16, pages 308–318, New York, NY, USA, 2008. ACM.

[29] S. Breu, R. Premraj, J. Sillito, and T. Zimmermann. Information needs in bug reports: Improving cooperation between developers and users. In *Proceedings of the 2010 ACM Conference on Computer Supported Cooperative Work*, CSCW '10, pages 301–310, New York, NY, USA, 2010. ACM.

[30] M. Cohn. *Succeeding with Agile: Software Development Using Scrum.* Addison-Wesley Professional, 1st edition, 2009.

[31] T. Downer. Some Clarification and Musings. `http://tylerdowner.wordpress.com/2011/08/27/some-clarification-and-musings/`, 2013.

[32] P. J. Guo, T. Zimmermann, N. Nagappan, and B. Murphy. "not my bug!" and other reasons for software bug report reassignments. In *Proceedings of the ACM 2011 Conference on Computer Supported Cooperative Work*, CSCW '11, pages 395–404, New York, NY, USA, 2011. ACM.

[33] C. A. Halverson, J. B. Ellis, C. Danis, and W. A. Kellogg. Designing task visualizations to support the coordination of work in software development. In *Proceedings of the 2006 20th Anniversary Conference on Computer Supported Cooperative Work*, CSCW '06, pages 39–48, New York, NY, USA, 2006. ACM.

[34] G. Jeong, S. Kim, and T. Zimmermann. Improving bug triage with bug tossing graphs. In *Proceedings of the the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ESEC/FSE '09, pages 111–120, New York, NY, USA, 2009. ACM.

[35] R. Johnson, Y. Rogers, J. van der Linden, and N. Bianchi-Berthouze. Being in the thick of in-the-wild studies: The challenges and insights of researcher participation. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '12, pages 1135–1144, New York, NY, USA, 2012. ACM.

[36] S. Just, R. Premraj, and T. Zimmermann. Towards the next generation of bug tracking systems. In *Proceedings of the 2008 IEEE Symposium on Visual Languages and Human-Centric Computing*, VLHCC '08, pages 82–85, Washington, DC, USA, 2008. IEEE Computer Society.

[37] Y. Koren, E. Liberty, Y. Maarek, and R. Sandler. Automatically tagging email by leveraging other users' folders. In *Proceedings of the 17th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '11, pages 913–921, New York, NY, USA, 2011. ACM.

[38] M. Li, M. Huang, F. Shu, and J. Li. A risk-driven method for extreme programming release planning. In *Proceedings of the 28th International Conference on Software Engineering*, ICSE '06, pages 423–430, New York, NY, USA, 2006. ACM.

[39] J. E. Moore and S. E. Yager. Understanding and applying participant observation in information systems research. In *Proceedings of the 49th SIGMIS Annual Conference on Computer Personnel Research*, SIGMIS-CPR '11, pages 126–130, New York, NY, USA, 2011. ACM.

[40] E. Murphy-Hill, T. Zimmermann, C. Bird, and N. Nagappan. The design space of bug fixes and how developers navigate it. *Software Engineering, IEEE Transactions on*, 41(1):65–81, Jan 2015.

[41] A. Ngo-The, G. Ruhe, and W. Shen. Release planning under fuzzy effort constraints. In *Proceedings of the Third IEEE International Conference on Cognitive Informatics*, ICCI '04, pages 168–175, Washington, DC, USA, 2004. IEEE Computer Society.

[42] T. Parr. *The Definitive ANTLR Reference: Building Domain-Specific Languages.* Pragmatic Bookshelf, 2007.

[43] D. Poshyvanyk, H. Dang, K. Hossen, H. Kagdi, M. Gethers, and M. Linares-Vasquez. Triaging incoming change requests: Bug or commit history, or code authorship? In *Proceedings of the 2012 IEEE International Conference on Software Maintenance (ICSM)*, ICSM '12, pages 451–460, Washington, DC, USA, 2012. IEEE Computer Society.

[44] I. Robinson, J. Webber, and E. Eifrem. *Graph Databases.* O'Reilly Media, Inc., 2013.

[45] G. Ruhe and M. O. Saliu. The art and science of software release planning. *IEEE Softw.*, 22(6):47–53, Nov. 2005.

[46] P. Runeson, M. Alexandersson, and O. Nyholm. Detection of duplicate defect reports using natural language processing. In *Proceedings of the 29th International Conference on Software Engineering*, ICSE '07, pages 499–510, Washington, DC, USA, 2007. IEEE Computer Society.

[47] K. Somasundaram and G. C. Murphy. Automatic categorization of bug reports using latent dirichlet allocation. In *Proceedings of the 5th India Software Engineering Conference*, ISEC '12, pages 125–130, New York, NY, USA, 2012. ACM.

[48] W. Swierstra and A. Löh. The semantics of version control. In *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, Onward! 2014, pages 43–54, New York, NY, USA, 2014. ACM.

[49] A. Tamrawi, T. T. Nguyen, J. Al-Kofahi, and T. N. Nguyen. Fuzzy set-based automatic bug triaging (nier track). In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 884–887, New York, NY, USA, 2011. ACM.

[50] C. Treude and M.-A. Storey. How tagging helps bridge the gap between social and technical aspects in software development. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 12–22, Washington, DC, USA, 2009. IEEE Computer Society.

[51] X. Wang, L. Zhang, T. Xie, J. Anvik, and J. Sun. An approach to detecting duplicate bug reports using natural language and execution information. In *Proceedings of the 30th International Conference on Software Engineering*, ICSE '08, pages 461–470, New York, NY, USA, 2008. ACM.

Appendix A

BTL Grammar

```
grammar BTL;

options {
  language     = Java;
  output       = AST;
  ASTLabelType = CommonTree;
}

tokens {
  TAG;
  FUNCTION;
  WINDOW;
  SHOW;
  DELETE;
}

@header {
package edu.uci.ics.sdcl.porchlight.antlr;
}

```

```
@lexer::header {
package edu.uci.ics.sdcl.porchlight.antlr;
}

statement
  : 'TAG' (name=IDENT | name=STRING) 'WHERE' clause ';'? EOF -> ^(TAG $name
      clause)
  | 'SHOW' (name=IDENT | name=STRING) ';'? EOF -> ^(SHOW $name)
  | 'DELETE' (name=IDENT | name=STRING) ';'? EOF -> ^(DELETE $name)
  ;

clause
  : (predicate | function | 'SINCE LAST'! window) (BOOLEAN^ (predicate |
      function | 'SINCE LAST'! window))*;

predicate
  : name=IDENT rel=RELATION (value=IDENT | value=STRING) -> ^($rel $name
      $value);

function
  : fn=IDENT '(' (param=IDENT | param=INTEGER) ')' rel=RELATION (value=IDENT
      | value=INTEGER) -> ^(FUNCTION $rel $fn $param $value);

window
  : value=INTEGER (type='DAYS' | type='WEEKS' | type='MONTHS' | type='YEARS')
      -> ^(WINDOW $value $type);

RELATION : '=' | '>' | '<' | '>=' | '<=';

BOOLEAN : 'AND' | 'OR';

STRING : '\"' ( options { greedy=false; } : . )* '\"' {
    setText(getText().substring(1, getText().length() -1)); };
```

```

IDENT : ('a'..'z' | 'A'..'Z' | '-' | '.') ('a'..'z' | 'A'..'Z' | '0'..'9' |
   '-' | '.')*;

INTEGER : '0'..'9'+;

WS : (' ' | '\t'| '\r'| '\n'| '\f')+ { $channel = HIDDEN; };
```

Appendix B

PorchLight User Study

B.1 Introduction

You are being asked to evaluate a new approach to bug triaging using a tool called PorchLight. Bug triaging is the process of reviewing a bug (defect or enhancement request) and making some decision about it, whether it's to assign it to a user or milestone, comment on it and request more information, or close it.

PorchLight is based on a concept called **tag sets**. Tag sets let you group and name bugs using a wide array of attributes. For example, a tag set could reference all unresolved bugs that have had at least 3 comments in the last 2 months. To help you create tag sets, a new language is available to express the attributes of the bugs you'd like to group. This language is called **BTL** (**B**ug **T**agging **L**anguage) and is available to you in PorchLight. For example, this is the BTL statement to create the tag set described above:

```
TAG "Active Bugs" WHERE FREQUENCY(comment) > 3 SINCE LAST "2 Months"
```

You can use this tag set to explore all of your bugs that have significant comment activity and may need attention. This allows you to narrow down your list of bugs into a more manageable set and explore them in more detail.

Note that PorchLight is not intended to be a replacement for your issue tracker. Its features are geared towards exploring the concept of tag sets and their potential value to triaging. As a research prototype, it's not as fully featured as JIRA or other bug trackers. So, bear with us.

The purpose of this evaluation is to introduce you to tag sets and how they can be created, managed, and used for triaging through the tool. This evaluation will be conducted in two parts, and each session will be video and audio recorded for research purposes. No personal information beyond your name and job role will be used, and this evaluation will not be used to assess your work performance.

B.2 User Study: Part I

The first part of this evaluation will familiarize you with the features in PorchLight. To make it easier, the tool has been loaded with bug reports from a project with which you are already familiar.

B.2.1 Tutorial

1. Basics
    (a) User list
    (b) Milestone list
    (c) Bug list
2. Bug detail view

    (a) Summary
    (b) Timeline

3. Quick filters and search
4. Actions
    (a) Assign to a user and milestone
    (b) Add to static tag set
    (c) Add comment or update status
5. Tag sets
    (a) Static tags
    (b) Dynamic tags
    (c) BTL reference sheet
6. Review and commit dialog

Once you are familiar with the functionality in the tool, you will have the chance to explore the bug reports available from the issue tracker. You are free to just browse through individual bugs, or use the pre-populated tag sets. A BTL Cheat Sheet has been provided if you are inclined to create a new tag set. During this part of the evaluation, we will prompt you with questions about the types of bugs you look for when you typically triage or work with bugs. We ask that you "think aloud" during this process.

If at any point during this session you have an idea for a tag set that you'd like to create to help you explore the bugs, feel free to ask for assistance and we can discuss your goal and help you create the tag set using the features in PorchLight.

B.3 User Study: Part II

In the second part of this evaluation we will explore tag sets and BTL in more depth by looking at the types of tag sets you find interesting and potentially useful for triaging. To help you get started, tag sets that were mentioned in Part I have been pre-populated for you.

B.4 Bug Tagging Language (BTL) Reference

Statement

```
TAG name % Separate field in tag set dialog
WHERE predicates  % Clauses, operators, and functions
[SINCE period] % Optional
```

Predicates

```
assignee = "jim"
status = "OPEN"
```

Operators

```
assignee = "jim" AND status = "OPEN"
(assignee = "jim" OR assignee = "sally") AND status = "OPEN"
```

Functions

Has more than 3 comments:

```
FREQUENCY(comment) > 2
```

Has been reassigned more than 3 times:

```
FREQUENCY(assignment) > 3
```

Has a Java stacktrace in the description:

```
HAS(stacktrace) = true
```

Has a screenshot attached:

```
HAS(screenshot) = true
```

Jim has authored a comment on the issue:

```
COMMENTED(jim) = true
```

Time Windows

```
SINCE LAST "2 months"

SINCE LAST "2 months 7 days"
```