PorchLight: A Tag-Based Approach to Bug Triaging

Gerald Bortis and André van der Hoek
Department of Informatics
University of California, Irvine
Irvine, CA 92697-3440 U.S.A.
{gortis,andre}@ics.uci.edu

Abstract—Bug triaging is an important activity in any software development project. It involves developers 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 tools provide only minimal support for bug triaging and especially break down when developers must triage a large number of bug reports, since those reports can only be viewed one-by-one. This paper presents PorchLight, a novel tool that uses tags, attached to individual bug reports by queries expressed in a specialized bug query language, to organize bug reports into sets so developers can explore, work with, and ultimately assign bugs effectively in meaningful groups.

We describe the challenges in supporting bug triaging, the design decisions upon which PorchLight rests, and the technical aspects of the implementation. We conclude with an early evaluation that involved six professional developers who assessed PorchLight and its potential for their day-to-day triaging duties.

Index Terms—bug triaging, bug trackers, tags.

I. INTRODUCTION

Bug triaging is the process of determining, first, if issues reported in the 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 [1]. 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 [2]. It is as important, however, in commercial projects, especially because such projects can ill afford to ignore incoming issues.

Bug triaging is most frequently associated with dedicated meetings, scheduled regularly to prepare for the next phases of development – whether these phases are a coarse-grained series of major releases (release planning [3–5]) or a fine-grained set of sprints in a more Agile project [6]. However, bug triaging also takes place outside of such meetings. It is very 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 may have been had of whom the expert is to resolve an issue, or a certain developer might become overloaded and the work must be re-balanced among the available developers [7]. In the Eclipse project, for instance, about 44% of reported bugs were reassigned at least once [8]. Bug triaging, then, is an on-going activity [9].

The de-facto tools used today in support of bug triaging are bug trackers, with some of the more popular tools being JIRA [10], Bugzilla [11], FogBugz [12], and Trac [13]. These tools are typically positioned as supporting the entire life cycle of a bug report, from developers and users first reporting a problem encountered (or suggesting a possible enhancement 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. As a result, bug trackers serve many uses beyond being the repository for all open bug reports, including as the developer’s to-do list, a user-facing portal, and a key communication hub for large open source projects [14].

Naturally, bug triaging is one of the activities supported by bug trackers, as the assignment (and possible reassignment at a later time) of a bug to a certain developer and certain milestone is part of the life cycle of a bug [15]. 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 [16,17], concerns have been openly voiced. As an example, a former contributor to the Mozilla project posted on his blog [2] an explanation for no longer contributing to the project, stating that “Triaging is broken. Period.” He continues by explaining that the bug tracker they use, 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 author 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.”

The author is also not alone in these thoughts. Other examples of concerned developers can readily be found (e.g., [18,19]), with at least one academic study backing up the need for a role-oriented bug tracker interface that “emphasizes certain aspects of the tracker’s data while abstracting others away,” including during the process of searching and triaging [14,20].
This paper introduces a novel software development tool, PorchLight, which we designed explicitly to support bug triaging. PorchLight recognizes that a key obstacle to effective triaging with existing 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, however, have a need to deal with bug reports in meaningful groups (e.g., bugs that do not include a screenshot, bugs that were reassigned three times or more, bugs related to component X and filed in the past two days) [9], which, except for a handful of pre-defined filters, are simply impossible to achieve with current bug trackers. Especially when the number of open bugs is large, it therefore becomes very difficult to triage effectively.

PorchLight ameliorates this problem by letting developers explore, work with, and ultimately assign bugs in groups. It is based on a novel underlying bug query language through which its users can flexibly tag bug reports to create bug sets of their choosing, with a predefined list of bug sets available by default. These bug sets are explicitly represented in the PorchLight user interface and, by turning them on or off, act as filters on the full set of bugs. As a result, developers can quickly group and then examine related bugs in all sorts of ways, giving them access to the bug repository based on attributes, time, patterns, and even relationships among bug reports. Together with a small 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 thus enables developers much greater control over how they choose to triage a large repository of bugs1 effectively.

We evaluated PorchLight with six professional developers who frequently triage bugs for a major healthcare open source software package. We first presented PorchLight, then asked them to work through a limited triaging session with a copy of the real bug database, and concluded with a short interview that focused on comparing their experience with PorchLight to their current practice of using JIRA. 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, the evaluation also uncovered some areas where PorchLight can be enhanced.

PorchLight builds upon our previous work on TeamBugs, a prior version of PorchLight that we presented in a 3-page paper at the CHASE 2011 workshop [21]. Compared to TeamBugs, PorchLight provides the following contributions: tagging as a way to group bug reports, a bug query language through which new tag sets can be created, and a new evaluation.

The remainder of this paper is organized as follows. Section II presents related work in the area of bug triaging. Section III presents a set of requirements unique to triaging tools. Section IV details PorchLight. Section V presents our evaluation and in Section VI we conclude with an outlook to our future work.

---

1 Note that we use the terms bugs and bug reports interchangeably throughout the paper.
This approach can significantly reduce the number of reassignments while at the same time improving the accuracy of first-time automated assignments.

C. Field Studies

Bugs, and particularly bug histories, have been the subject of quite a few studies (e.g., \cite{9,14,15,28}). 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 \cite{1,29}.

Most influential on our work is a study in which its authors conducted interviews with industry and open source developers to understand coordination problems that arise when managing bug reports in large, distributed teams \cite{15,28}. Among many other insights, they found 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. The paper further recognizes particular 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 field study of bug tracker usage, this time in a smaller collocated team setting, reveals that the bug tracker serves as a boundary object and that participants in different roles leverage the bug report information in different ways and have different perspectives with different needs \cite{14}. These findings suggest that, for triaging, a new perspective is needed that meets the requirements of a triager.

III. REQUIREMENTS FOR BUG TRIAGING

While existing approaches such as duplicate detection and assignment automation clearly have made advances in easing the job of the developer when it comes to bug triaging, in their current state they still require a person to drive the process, as their success rate is insufficient to rely on automation only. The developer, 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.

Rather than attempting to provide more accurate duplicate detection algorithms, or automated assignment techniques with yet more precision, we took a step back and began to focus on this overall process of triaging – what is it that developers need to achieve, and how can they achieve it effectively? With this question in hand, we began to realize, much like the Mozilla developer we quoted in Section I, that, to provide effective tool support for triaging, it is necessary to part with the bug tracker and provide triaging support through its own, dedicated tool.

To guide the development of such a tool, we relied on the observations in the studies we described in Section II, our own professional experience, and the kinds of observations quoted in Section I and made in the open source community to identify a set of five key requirements to supporting bug triaging.

A. Bug Management

The number of bug reports that must 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 \cite{2}, with a multitude of new bugs being filed each day. To make the process of triaging these bug reports manageable, a triaging tool should provide developers with ways to quickly organize and sort the bug reports; that is, developers 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 bug reports.

It particularly has to be possible for the developer to control which subset they want to see. While predefined sets are useful, if only such sets are available, they necessarily limit the ability of the developer. Consider the following examples of advanced types of sets that a developer might want to explore:

1. Bugs which have no comments or replies from the community.
2. Bugs which have gone for 14 days with no reply or activity.
3. Bug reports that have been reassigned multiple times to different developers.
4. Bug reports filed within a week after the most recent release, that include a screenshot or stack trace, and come from users who have filed reliable bug reports in the past.

These, and other sets like it, should be easily obtainable by the developers. 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. An especially realistic scenario is set 4 above: the bug reports likely contain enough information to be triaged relatively quickly.

B. Ad-hoc Grouping

Situations arise when developers cannot immediately triage a bug until more information is obtained, or other bugs are first triaged \cite{9}. To support this, a bug triaging tool should provide support for ad-hoc grouping of bugs, i.e., the manual creation of sets of bugs that are seemingly unrelated but of importance for the developer to keep track of. This allows developers 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 screenshots [9]. It should also be possible to persist any ad-hoc groups across sessions, so they can be used a starting point for a future triaging session. New triagers, too, may want to use ad-hoc groups 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 also may be used as tentative plans. Rather than making definite assignments bug-by-bug, a triager can use an ad-hoc group to create a planned set of bug assignments for a given developer or milestone that they do not quite commit just yet. If one of the developers is already overloaded, for instance, it may still be desirable to assign him some bugs that he or she is most qualified to address. Doing so through an ad-hoc group allows the triager to safeguard against dumping too many new bugs on 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.

C. Instant Feedback

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 triager has no easily accessible cumulative record of their decisions, other than their personal memory. This is a problem, as triaging decisions are not just based on which developer is the suitable person 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 [15]. Having this information available instantly as bug reports are assigned can serve as valuable feedback during the few minutes allotted to each issue, and assure that one particular developer or milestone is not overloaded. 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.

Similarly, a triager should be able to gauge 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.

D. Rich Bug Histories

Another important factor when assigning bugs is the history of activities related to the issue [15]. The fact that source code has been committed to the bug, comments from developers and users, and any status changes or reassignments are examples of the kind of information that should be at hand when developers are making triaging decisions. For instance, a bug that has been repeatedly reassigned to a later milestone may indicate that the bug report does not contain sufficient information, or that it is difficult to identify a developer with sufficient expertise [7]. Such information may well influence the actions of the triager, who, instead of simply assigning the bug, may send a request for more information to the reporter, or contact the developer who was originally assigned the bug to learn what expertise is needed to address the bug.

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.

E. Single Context

It goes without saying that the process of assigning a bug to a milestone or developer must be lightweight and require little effort. When dealing with hundreds or thousands of new bug reports, developers should be able to efficiently work through the bugs and make assignments. Particularly, assigning a bug should not disrupt triaging or alter the context created by the developer. Existing bug tracker interfaces require developers to first search for a list of bugs to triage. Viewing more details about a bug, creating a new filter, or simply putting a bug aside for triaging at a later time, all require leaving the list and losing the context. It is crucial to avoid such context switching. This means that the same interface must support bug management and ad-hoc grouping, visualization of bug histories, feedback, and assignment of bugs to developers and milestones.

IV. PORCHLIGHT

To explore these requirements, we designed and developed PorchLight, a novel bug triaging tool. In this section, we detail the key design decisions underlying PorchLight, show how its basic features already change the process of triaging, highlight the advanced features through which its full power is unlocked, and provide a brief look at its implementation details.

A. Key Design Decisions

We explored a range of alternative metaphors in designing PorchLight, including a stacking metaphor (each bug report is represented by a notecard and notecards are sorted into stacks to represent desired groupings), a network metaphor (each bug report is a node in a network that the triager can navigate and configure), and a grid metaphor (each bug report is represented by a tile in a grid that the developer can resharuffle at will).

Ultimately, we chose to adopt a tagging metaphor. Tagging has emerged over recent years as a powerful technique that is applicable in many different situations (e.g., managing e-mail [30], collaborating through work items [31], annotating online media [32]). Moreover, every bug tracker internally already relies on a set of predefined tags and in some projects it is now recommended to first organize bugs using tags before triaging them (Chromium being the prime example [16]). Tagging, too, is an easy way for a community to establish norms [31] and communicate them to new contributors who may wish to help in the triaging process. We thus choose to expand on this emerging practice: instead of hiding or restricting the tags available, PorchLight puts them front and center to the experience. This leads to the following four design decisions underlying PorchLight:

1. Leverage tagging to organize bug reports into tag sets. It is not sufficient to just rely on the presence of tags to say that bug triaging is now better supported. Rather, the tags should be leveraged to provide higher level functionality to a triager. In PorchLight, every unique tag automatically
provides an index into the full set of bug reports to create a tag set. These tag sets become one of the primary ways of exploring, working with, and assigning bugs.

2. Provide a set of default tag sets, together with filters, for common tasks. Triagers should not have to construct their view of the bug reports from the ground up each time they triage. PorchLight therefore includes default tag sets that represent subsets of bugs commonly needed during triage of a bug database. Further, it combines the default tag sets with filters that operate over the currently selected tag sets to reduce the number of bugs being viewed.

3. Provide a bug tagging language to support attaching tags to individual bug reports. It is impossible to predict all of the needs of triagers in terms of what exact subset of bugs they may be interested in at a given moment in time. As a consequence, it is necessary to allow a triager to construct their own tag sets. PorchLight provides a specialized bug tagging language, called BTL, that allows for the creation of new tag sets.

B. Basic Features

Figure 1 shows the PorchLight user interface. Upon starting PorchLight, users are presented with a project selector (top left, currently “Mirth Connect” is selected), that is populated with a list of projects recorded in the bug tracker to which PorchLight is connected (as we shall discuss in Section IV.E, PorchLight is implemented as a separate interface to an existing bug tracker, pulling all its data from this bug tracker and recording all of the triaging decisions in it as well). Selecting a project from the list leads to three other lists being populated: a developer list (1) on the left of the interface, a milestone list (2) on the right, and the list of bug reports in the middle (3).

The developer list contains all individuals who to date have contributed to the data in the bug tracker in any way, including opening, resolving, or commenting on a bug. In a typical open source project, this list can become very long, given that many different individuals may submit bug reports. The list therefore consists of two blocks, the first block listing developers and the second block listing people who only submit bugs or comment on them. Each block is sorted alphabetically (to make it easy to locate developers to be assigned), and the number of bug reports assigned is shown on the label for each developer (i.e., 704 bug reports have overall been assigned to jacobb).

The milestone list holds all of the milestones for the project, with the labels providing a count of the number of bug reports that are assigned to each milestone. Note that 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 addressed.

The bug list is populated with all of the bug reports in the project, presented linearly with newer bugs at the top and older bugs at the bottom. The triager navigates this list by moving up and down (either with the mouse or using the keyboard). When they do so and encounter a bug that has already been assigned, the developer and milestone to which it is assigned are marked. For instance, in the figure, MIRTH-1968 is selected and shown to be assigned to geraldb and milestone 2.2.0.

Selecting a bug from the bug list also displays details from the bug report (4, 5). This includes a summary of the bug, any tags that have been associated with the bug (see Section IV.C), a full description of the bug, and the event timeline. The event timeline displays, in an at-a-glance manner, the activity that has occurred on the bug report. The starting date for the timeline is the date the bug was created, and the ending date is the date of...
the last activity. Actions taken in relation to the bug, such as commenting, committing source code, or reassigning, appear as colored markers. Each event type is assigned a different color to make it easier to distinguish them in the timeline. Clicking a marker reveals the details of the activity (6), for instance, in the case of a commit, the list of files associated with that commit is shown, together with the types of changes made to each of the files (added, modified, or deleted). To make it easier to quickly iterate through the events on the timeline, developers can use the left and right keys to scroll backward and forward through the timeline.

To quickly filter the bug list to a subset based on the status of the bug report, developers can use the quick filters (7) at the top of the interface. Marking any of these filters reduces the list of bugs in the center of the view to those that match the status of the filter, i.e., bugs that are unassigned (to either a developer or milestone), have a specific status (open, resolved, closed), or belong to an active tag set (see Section IV.C).

Finally, a search box (8) allows the developers to search the bug reports textually, just like they currently can in an existing bug tracker. Any bug report in which some portion of the text or metadata matches the provided search term is included in the resulting bug list.

Actual assignments are performed through drag-and-drop: a bug, or set of bugs that has been Ctrl-clicked, is dragged onto a developer or milestone on the left or right of the interface. The respective count of the developer or milestone is then updated to reflect the action taken after a bug or set of bugs is dropped.

Together, these basic features already change the nature of triaging. They address requirements 3-5 as presented in Section III: developers receive visual feedback on the actions they take (particularly on bug assignment), a rich bug history is available for the triager to take in at-a-glance and explore in more detail if so needed, and, perhaps most importantly, PorchLight offers all of its functionality in a single interface that avoids the need for explicit context switching (i.e., the back and forth syndrome of existing 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.

C. Tag Sets

PorchLight’s basic features make it possible for a developer to assign groups of bugs to developers and milestones (i.e., by using Ctrl-click to select multiple bugs and then dragging them to the desired developer or milestone), but not necessarily very convenient. Triagers still must manually select multiple bugs and there is nothing they can do with the collective other than assigning it.

To better support developers in exploring, working with, and assigning groups of bugs, PorchLight employs the concept of tag sets. A tag set is an explicit entity that provides an index into the overall set of bug reports to identify some subset. Each tag set has a name and a color associated with it, and each can be selected or deselected independently from the other tag sets. Figure 1 shows the tag set part of the user interface (9), with at five tag sets available to the triager: 3.0, geraldb, Popular, Hot Potato, and Zombie. Enabling a tag set has two effects: (1) in the bug list, it colors every bug report belonging to it, and (2) to the left of the bug list, it puts a colored marker (making it possible to identify bugs that belong to multiple tag sets).

If so desired, the bug list can be sorted by clicking the “Has Active Tag” filter (7) and then clicking on the “Tags” column header. The list will be sorted by number of tags; that is, bugs belonging to multiple tag sets will appear at the top and bugs belonging to none at the bottom of the list.

The four different kinds of tag sets supported by PorchLight are: (1) predefined tag sets, (2) developer and milestone tag sets, (3) ad-hoc tag sets, (4) user-defined tag sets.

Predefined tag sets. PorchLight includes predefined tag sets that provide groupings that are commonly used in triaging. The creation of such groupings should not be necessary for a triager to specify each time they work with PorchLight. This includes tag sets for bugs that:

- *Popular*: bugs that have had more than three comments.
- *Missing details*: bugs that do not have a screenshot 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.

These tag sets are always available to the triager.

Developer and milestone tag sets. Sometimes, it is useful for a triager to see all the bug reports associated with a developer or milestone. In PorchLight, this is a very simple action to perform: drag the developer icon from the list of developers on the left, or the milestone icon from the list of milestones on the right, to the tag set area. This creates a new tag set automatically that can henceforth be used.

Ad-hoc tag sets. The second requirement we discussed in Section III motivates the need for 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 any bugs from the bug list and dragging them (one-by-one or as a group after Ctrl-clicking them) onto one of the listed tag sets. If a new tag set must be created first, they can drag to an empty area in the list, which brings up the tag set settings dialogue (see Figure 2), in which the WHERE field is pre-populated.

User-defined tag sets. The same tag set settings dialogue can be used to create user-defined tag sets: tag sets where the set of bugs that is included is determined by a bug tagging language (BTL, see Section IV.D) statement. 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 a-priori nor does it always follow convention. Triagers must be able to quickly get to such a set, and BTL provides this ability. BTL statements act like SQL queries, and dynamically select all of the bugs that match the criteria provided. In Figure 2, for instance, a new tag set is being created that represents all bugs assigned to geraldb with at least 4 comments attached.
Combined, these four types of tag sets transform the central bug list of PorchLight from a list to working area 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 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. Combined, then, the basic features described in Section IV.B and the tag sets described here cover all of the requirements we laid out in Section III.

We briefly note two additional functionalities. First, at any moment in time it is possible to update the BTL statement of a tag set. Simply double clicking brings up the dialogue box with the current statement that can then be edited. Second, all of the tags and tag sets persist across triaging meetings. This means it is always possible to continue where one left off, but also that the tags and tag sets can become a vehicle for sharing common rules and criteria for grouping bug reports (thereby establishing community norms).

### D. BTL

BTL is a SQL-like declarative language that allows for the specification of tag sets based on attributes, both explicit and implicit, that are available in bug reports. Explicit attributes are those captured as part of the bug report, like the description or the affected release. Implicit attributes are those that are derived by looking at a bug’s history, such as how often it has been reassigned. Table 1 shows several examples, taken from Section III.

The BTL grammar specifies three primary commands: TAG, SHOW, and DELETE. The TAG command is used to tag a set of bugs using a series of clauses. A clause can be a simple predicate (e.g., assignee="geraldb"), a function (e.g., FREQUENCY(comment) > 3), or a time window expression (e.g., SINCE LAST “2 weeks”). Supported time window values for narrowing the scope of the clauses include free-form expressions such as “7 days” or “5 months”. The evaluation of a TAG command results in a tag set being created, and the associated bugs being stored in a working memory (Section IV.E). The SHOW command is used to retrieve a tag set from the working memory so it becomes available to the triager in the user interface. The DELETE command is used to remove a tag set, and thus dissociate all bugs with that specific tag.

The language is extensible, since functions are not specified explicitly in the grammar, but in the implementation. It is thus possible to add new functions through plug-ins. For instance, we plan to add support for functions that detect duplicates and functions that obtain relevant information from the configuration management system to allow statements such as:

- TAG “Bugs For Gerald” WHERE COMMITTED("geraldb")=true;
- TAG “Dupes of Bug 75” WHERE STATUS=open AND ISDUPE("75") = true

### E. Implementation Details

Since PorchLight is not a standalone bug tracker, but rather a new interface focused on triaging, we implemented it so that it integrates with and complements existing bug trackers. This allows developers to continue using their preferred bug tracker for the roles for which it is best suited, while triagers have a specialized interface to support the triaging process.

<table>
<thead>
<tr>
<th>Selection</th>
<th>BTL statement</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bugs which have no comments or replies from the community</td>
<td>TAG “No Comments From Community” WHERE STATUS=open AND FREQUENCY(comment)=0</td>
</tr>
<tr>
<td>Bugs which have gone for 14 days with no reply or activity</td>
<td>TAG “No Activity” WHERE FREQUENCY(comment)=0 AND SINCE LAST “14 DAYS” AND STATUS=open</td>
</tr>
<tr>
<td>Bug reports that have been reassigned multiple times to different developers</td>
<td>TAG “Reassigned Multiple Times” WHERE FREQUENCY(assign)=3 AND STATUS=open</td>
</tr>
<tr>
<td>Bug reports filed within a week after the most recent release, that include a screenshot and stack trace, and that stem from users who have filed reliable bug reports in the past</td>
<td>TAG “New Bugs” WHERE STATUS=open AND HAS(screenshot)=true OR HAS(stacktrace)=true AND reporter IN reliable-reporters AND SINCE AFTER “5-15-2012”</td>
</tr>
</tbody>
</table>
PorchLight consists of two components. The first component is the user interface we have described in the previous section (shown in the top right of Figure 3, which presents the architecture). The second component, labeled *Tag Set Processor*, serves as the back end. It specifically imports bug reports and their metadata from a bug tracker, evaluates BTL statements, maintains a working memory of bugs and tag sets, and provides a plug-in mechanism for extending BTL.

The importer is used to import and map bug reports and associated metadata (users, comments, source code commits, etc.) from different bug trackers into TSP’s internal data store. An importer interface is provided to implement importers for new bug trackers and configuration management systems. For our purposes, we built a JIRA and Subversion importer.

The evaluation process begins when a BTL statement is provided by the triager. The statement is then parsed and the resulting query is evaluated by matching it against the bug reports as well and certain derived attributes (such as comment and assignment counts). Once the bugs have been identified, a new tag set is created in the working memory and the bugs are associated with the tag.

As mentioned, the set of functions that BTL supports can be extended. Since the BTL grammar allows for any function to be invoked as part of a clause, these functions can be developed as Java classes that implement a specific interface. These plug-ins are loaded on initialization of the TSP.

V. Evaluation

To evaluate PorchLight, we recruited six professional software developers and project managers who work at a nearby software development company and perform triaging on a regular basis. With each participant, we conducted a 45-60 minute evaluation session. Before each session, we populated PorchLight with bug reports from three different ongoing projects that 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 company’s existing bug tracker.

We then provided each participant with a tutorial describing PorchLight, including an overview of performing assignment using drag-and-drop between lists, the activity timeline, quick search and filters, and adding comments or updating the status of a bug report. We also introduced the concept of tag sets, and provided a brief tutorial on how to create tag sets from sample BTL expressions, or by using drag-and-drop from developer or milestone lists.

We then asked each of the participants to complete a series of bug triaging tasks using the bug reports from their project.

- We first asked the participants to identify all bugs that had been assigned to particular developer and version.
- We then asked participants to identify all bugs that had been commented on at least three times in the previous two months.
- When then asked 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 the project.
- Finally, we asked 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 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 for triaging tasks, as well as their openness to adopting PorchLight as their triaging tool.

A. Results

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 (Q1, avg. 4.3), as well as to perform the actual triaging (Q2, avg. 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 developer commented, “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 helped them save significant time with triaging, 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 custom 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 (an oversight on our part, since we forgot to include it in the bug metadata we imported). Others had difficulty recalling the syntax 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 (Q8, avg. 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.”

During the open ended discussion, several participants provided examples of tag sets they would use, 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.

Regarding potential adoption, all six of the participants either agreed or strongly agreed that PorchLight had functionality that would improve their triaging experience compared to their current bug tracker (Q9, avg. 4.5), and they would use PorchLight as their bug triaging tool if it were made available (Q10, avg. 4.5). 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.”

B. Suggestions for Improvement

During the open discussion following the evaluation tasks, we received numerous suggestions for how PorchLight could be improved. Some of the suggestions were minor user interface changes, like changing the developer and milestone lists to use decorators on the icons to indicate the assignee rather than showing the items as selected, since it led them to believe that the items could be clicked on. Another common suggestion was to provide more information in the timeline event details, including the author of a comment and the date and time when the event occurred.

We also received feedback on more conceptual issues. For example, several participants felt that the developer and milestone list could be used as a filtering mechanism without having to first create a tag set. Also, some suggested that the bug counts listed for each developer and milestone should be broken down in a more meaningful way, for instance by showing the number of open and closed bugs, or the number of new bugs assigned in the current triaging session. There were also several comments regarding the activity timeline. Particularly, some bug reports involved many events within a short window of time, making it difficult to identify and select individual events to view their details. Numerous suggestions were proposed, including showing only the time periods when the activity occurred and skipping long periods of inactivity, or providing a way to zoom in to areas within the timeline.

Finally, several participants commented during the evaluation that not knowing the full extent of BTL’s capabilities resulted in a learning curve, and that a guide to the attributes and functions that can be used in a BTL statement should be made available directly in the user interface. We recognize this concern, but at the same time find it a validation of our approach since it demonstrates that the participants were actively looking to use BTL to create tag sets to filter the bug list, but did not have sufficient information to do so easily. We can address this in a subsequent version by including rich documentation and active code completion within the BTL statement editor.

C. Threats to Validity

Comparative setting and bug tracker not representative. All six participants work at the same company and use the same bug tracker for triaging (JIRA). Their collective responses might therefore be biased toward a single current practice and tool, since other projects could well have different practices or use different tools. However, there are striking similarities between the comments made by the participants during our evaluation regarding issues with their current bug tracker and those quoted in Section I. Specifically, the inability to quickly identify bugs that need to be triaged, and then to perform the assignment without losing one’s context, was emphasized by both parties, even though they work in very different contexts with different bug tracking tools.

Limited participant sample size. We acknowledge that six participants are insufficient to draw strong conclusions as to the benefits of PorchLight. However, as a preliminary study, the level of enthusiasm displayed by the participants is indicative of the potential of PorchLight in addressing pressing triaging problems, and determining this potential was the initial goal for this evaluation. It is also worth noting that the author of the blog post we quoted in Section I, when contacted to request permission for the quote and explained our goals, became interested and has asked us to provide him with our paper and findings. Clearly, people recognize that PorchLight seems to fill an important gap in the current tool spectrum.

Representativeness of the tasks. We realize that asking the participants to perform a handful of triaging tasks is not the same as asking them to actually perform triaging “live” using PorchLight. This risk is mitigated to an extent, because we pre-loaded the tool with bug reports from the project’s bug tracker and asked the participants to perform tasks that represent what they would normally do in a triaging meeting (finding certain kinds of bugs, assigning them, etc).
VI. CONCLUSION

Bug triaging is a crucial activity in any software project, but has received little attention to date, particularly when it comes to dedicated tool support. This paper contributes PorchLight, a novel bug triaging tool that offers a unique, tag-based interface. Shaped by a set of five requirements, PorchLight offers triagers the ability to explore, work with, and assign bugs in groups that are identified through queries expressed in BTL, our novel bug tagging language. With PorchLight, triagers no longer need to manually click through pages and pages of bug reports, manage filters and bookmarks that are uncoordinated, or experience the frustration of losing the context of a triaging session.

To assess PorchLight, we performed an evaluation with six professional software developers who frequently act as triagers. We are encouraged by the results and the feedback, particularly with the ready adoption of the concepts underneath PorchLight in conducting the triaging tasks, as well as the enthusiasm that the participants expressed and their desire to explore the use of PorchLight in future, actual triaging meetings.

We are keen to continue this work. First and foremost, we will take the feedback from the participants in refining the user interface of PorchLight, particularly in providing triagers with more refined feedback in their actions and in assisting them in specifying BTL statements. Second, it is clear that the next step is to deploy PorchLight and examine it in action in actual triaging meetings. Finally, we see several opportunities for broadening the functionality of PorchLight, including integrating duplicate detection and automated assignment techniques by letting their results serve as advice to triagers.

ACKNOWLEDGMENTS

This work was sponsored in part by NSF grant IIS-1111446. We thank Wayne Huang for his help in implementing the PorchLight user interface, and also Mirth Corporation for their assistance with the evaluation.

REFERENCES