LINKSTER: Enabling Efficient Manual Inspection and Annotation of Mined Data

Christian Bird[†], Adrian Bachmann[‡], Foyzur Rahman[†], Abraham Bernstein[‡]

[†]Computer Science Department, University of California, Davis, USA
[‡]Department of Informatics, University of Zurich, Switzerland
{bachmann,bernstein}@ifi.uzh.ch
{cabird,mfrahman,ptdevanbu}@ucdavis.edu

ABSTRACT

While many uses of mined software engineering data are automatic in nature, some techniques and studies either require, or can be improved, by manual methods. Unfortunately, manually inspecting, analyzing, and annotating mined data can be difficult and tedious, especially when information from multiple sources must be integrated. Oddly, while there are numerous tools and frameworks for automatically mining and analyzing data, there is a dearth of tools which facilitate manual methods. To fill this void, we have developed LINKSTER, a tool which integrates data from bug databases, source code repositories, and mailing list archives to allow manual inspection and annotation. LINKSTER has already been used successfully by an OSS project lead to obtain data for one empirical study.

1. INTRODUCTION

Over the past 10 years, information mined from software archives has become an increasingly used source of data for both tools and empirical studies in software engineering research. The majority of such research relies on automatic methods of mining and analyzing such data. Oftentimes, as in the case of Sliwerski *et al.* [1] and Bird *et al.* [2], heuristics are used because sound and precise techniques do not exist. These heuristics are based on codifying the steps used in a manual process, and in many cases these methods have been shown to be quite effective. However there are cases where manual inspection and annotation may be required, for example:

- To *refine* the results of automatic, heuristics-based approaches.
- To *evaluate* how well a heuristic-based, or predictive tool, performs.
- To understand a phenomenon by examining multiple data sources in an exploratory way.
- To understand very noisy data.

In their 2006 CSCW distinguished paper, Cataldo *et al.* [3] examined the relationship between coordination patterns and the time to resolution for Modifiation Requests (MRs). They observed that communication patterns had a significant effect on resolution time. In this study, developer communication (in the form of IRC logs) had to be manually associated with relevant modification requests (MRs) because they were rarely mentioned explicitly in discussion. The authors were able to associate MRs via contextual information and key phrases like "John's issue" or "the memory problem." They estimate that the process took over 2000 man-hours divided among three co-workers to complete.

The difficulty with manual inspection is that it can be cumbersome to identify, integrate, view, and annotate different forms of data from multiple sources. Consider the steps required in our lab to record notes about the changes that introduced a severe bug into a piece of software:

1. Execute a SQL query on the database which contains bug tracking and source code repository data to identify the severe bug and the filenames and revisions associated with the fix.
2. Query repository system for meta-data associated with commit such as author, date, and log message.
3. Check out the content of the file before and after the bug fixing commit to examine change context.
4. For each line that was modified in the bug *fixing* commit, use `git blame` to determine which commits introduced the offending line.
5. Check out the contents of the file before and after each of the bug introducing commits, to examine the context of these changes.
6. For each bug introducing commit, issue a SQL query to extract meta-data such as author and log message.
7. Issue an insert SQL statement to record observations from manual inspection of the data.

In many steps, information in one step (e.g. a revision and a filename) must be recorded for use in the command of a future step. These steps also assume that one is familiar with the data schema, repository locations, and syntax of the commands used. Whether for impartiality or required expertise, the researchers (often the only people who have such knowledge) may not be annotating the data themselves. In these cases, efficiency is critical because researchers want to maximize the amount of data obtained, tedium may drive away potential contributors, and there may be some form of compensation provided (e.g. an hourly wage for undergraduates). We claim that when the process by which manual data is inspected and annotated becomes efficient enough, the types and uses of such data qualitatively change.

In an effort to take advantage of the benefits of manually inspected and annotated data and overcome the problems associated with the process of obtaining it, we have developed LINKSTER. LINKSTER is written in python and PyQt, and runs on Linux and OS X.

In a recent study [4], an APACHE developer used LINKSTER to examine 677 commits in *one* day. He linked commits to their associated bugs, marked the type of each change, and included notes with commits to help us understand how the project worked, how well a linking heuristic worked, and what the effects of a perfect oracle are on a bug prediction method.

2. DESCRIPTION OF LINKSTER

LINKSTER efficiently displays, integrates, and allows inspection and annotation of information from three main sources of data:

source code repositories, developer mailing lists archives, and bug tracking databases. LINKSTER requires access to a source code repository for file content and a database which contains the raw mined repository, mailing list, and bug tracking information. All notes and annotations made by the user are also recorded in the database.

Commit Information

Figure 1-a contains a screenshot of the commit information window of LINKSTER. The top (**1**) contains a list of commits that satisfy some constraint, such as commits within a time window. Each line contains the revision identifier, date of the commit, author, and the first line of the commit message. Right clicking on a commit entry brings up a contextual menu that will allow the user to populate the *Message Information Window* with messages sent by the author of the commit within a time window of the commit.

When a commit entry in the list is selected, the meta-data is updated in the bottom half (**2**). The list of files modified in the commit (**3**) is also displayed. Double clicking a file brings up the *Blame & Diff Information* for the file, allowing the user to examine the exact changes that were made. For annotation purposes, the user may select the reason(s) for the commit by checking boxes (**4**) or drag and drop (or remove) a bug record from the *Bug Information Window* into the list of *bug IDs* (**5**), which is populated with the set of automatically identified links between the commit and bug records Finally, the user may enter free form notes for the commit (**6**).

Blame & Diff Information

Figure 1-b shows the blame & diff information window for the changes to a file in a particular commit. The left view (**8**) shows the content of the file prior to the change and the right view (**10**) displays the content of the file after. Lines that were removed in the commit are prefixed with "−" and are highlighted red, while those that are added have "+" and are green. In addition, each line is prefixed with revision identifier associated with the commit that introduced the line. Selecting a line causes all other lines introduced in the same commit to become highlighted in gray and updates the meta-data (**7**) with information from the line introducing commit. This can help the user determine why, when, and by whom, the line was added. When more information is desired, double clicking a line will bring up a new blame & diff window for the commit which introduced the line.

The views are synchronized such that scrolling in one view causes the other to change accordingly. The thumbnail view (**9**) graphically shows the differences for the entire file with red indicating removed lines and green, added lines. Clicking on a location in the thumbnail view will cause the pre and post views to jump to that location, making identifying and examining the changes easy for large files.

The *commit status list* (**11**) contains an entry for each commit that originates a line in the file views. This is for use in a study of bug introducing commits. When the user selects a line in either view, the matching commit number is highlighted in this list. Assuming that the file represents a bug fix, the user can annotate each prior commit as "guilty" of introducing the bug, an "innocent" bystander in the bug introduction, "unset", indicating that no one has examined it yet, or "unsure", indicating that the user has inspected it, but cannot confidently assign "guilty" or "innocent".

Bug Information

Figure 2-a contains the *Bug Information Window*. The top portion (**12**) is a list displaying bugs from the bug database the satisfy some criterion such as marked as closed or having a higher than normal severity level. Each entry contains the bug ID, date that the bug entry was created, and the one line summary of the bug. Hovering over an entry shows the bug severity in a tooltip. Any of these entries may be dragged to the *bug IDs* list (**5**) in the commit information window or to the *associated items* list (**20**) of the message information window to indicate that it is associated with a bug or message.

Selecting a bug entry will populate the bottom half of the window with detailed information. The left side (**13**) contains short attributes of the bug, while the right side (**14**) displays the full bug description followed by all of the comments in chronological order with author and date. Clicking on the *Bug Activity* tab (**15**) displays a list (not shown) of all changes to the bug record, such as assigning the bug to a developer or marking a bug as closed. Each entry indicates when the change was made, who made it, and the old and new values for the changed field. Finally, clicking on the *Fixing Files* tab (**16**) shows a list (not shown) of all of the file commits associated with the fix of the bug. This list is comprised of files automatically linked based on attributes of commits such as commit messages and also files manually linked with LINKSTER. Double clicking on a file in this list will bring up a blame & diff window for the commit.

Message Information

Information from messages sent on the developer mailing lists for a project is displayed in he window depicted in Figure 2-b. As with the others, the top portion (**17**) lists messages that satisfy some constraint, in this case, those sent by Joe Orton. When a user chooses to see the messages from a mailing list participant, we use aliasing information to identify messages sent by that person via all of their known email addresses.

Selecting a message updates the information shown below. Relevant information from message headers, such as sender, date, and subject, are displayed (**18**) as well as the body of the email message (**19**). Bugs from the bug tracking database and commits from the source code repository that are associated with the message are listed in the *Associated Items* list (**20**). The user may drag bug and commit entries to this list to indicate that the message references them. This information is backed directly by a database so that it can be filled from automatic methods and the data can be used by other tools or for later analysis easily.

3. CONCLUSION

The variety and sophistication of the questions investigators seek to study in empirical software engineering have grown dramatically; increasingly, manual annotation and labeling of data is becoming necessary. LINKSTER is an interactive, integrated, browsing and querying tool that facilitates the exploration and annotation of large volumes of semi-structured softare engineering data.

4. REFERENCES

[1] J. Śliwerski, T. Zimmermann, and A. Zeller, "When do changes induce fixes?" in *Proc. of the international workshop on Mining software repositories*, 2005.

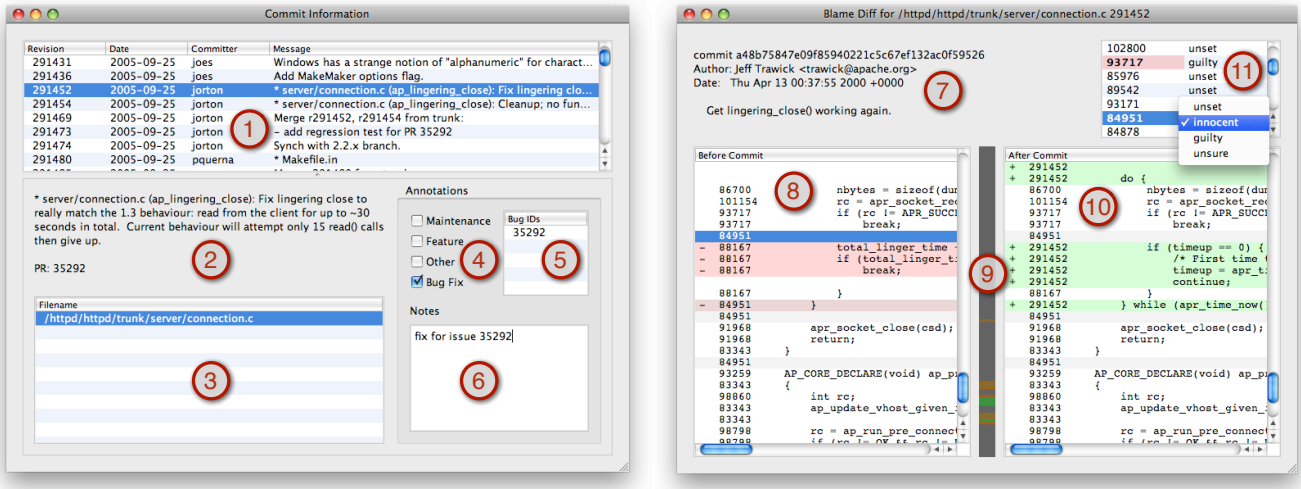
[2] C. Bird, A. Gourley, and P. Devanbu, "Detecting Patch Submission and Acceptance in OSS Projects," in *Proc. of the International Workshop on Mining Software Repositories*, 2007.

[3] M. Cataldo, P. Wagstrom, J. Herbsleb, and K. Carley, "Identification of coordination requirements: implications for the Design of collaboration and awareness tools," *Proc. of the 20th conference on Computer supported cooperative work*, 2006.

[4] A. Bachmann, C. Bird, F. Rahman, P. Devanbu, and A. Bernstein, "The Missing Links: Bugs and Bug-fix Commits," in *Proc. of the 16th Symposium on Foundations of Software Engineering*, 2010.

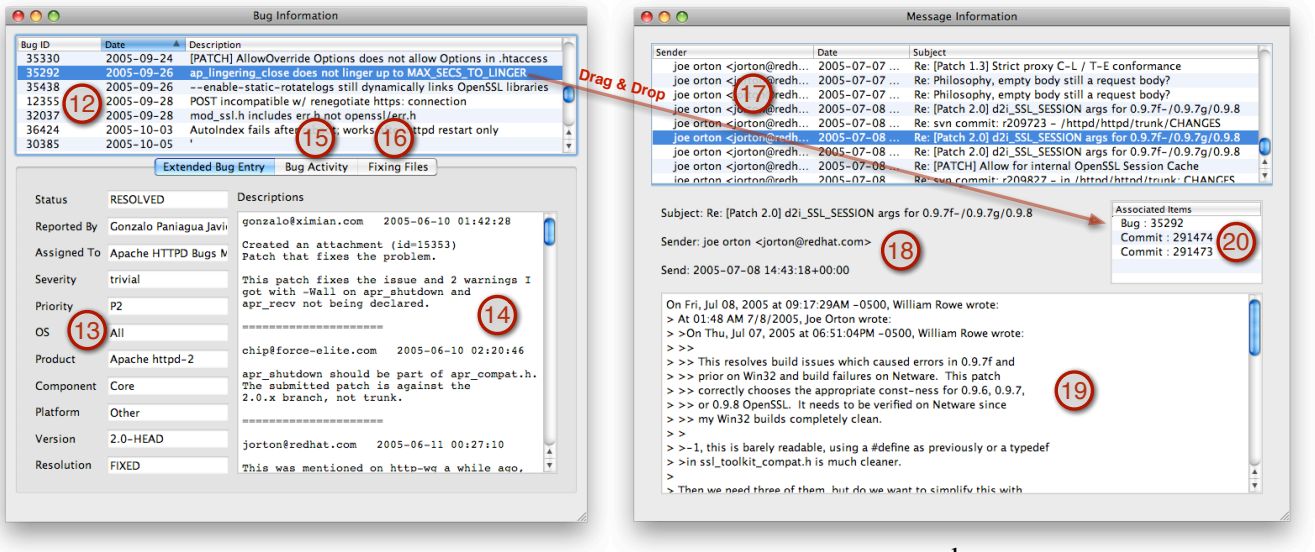
Appendix B. SETUP AND EXECUTION

Appendix A. SCREEN SHOTS



a b

Figure 1: The commit information window and the blame diff window



a b

Figure 2: The bug information window and the email information window

The setup for the demonstration will be relatively straightforward. Although the tools allows for a distributed setup (*e.g.* the database can reside on a system different from the source code repository), I will have everything required on my laptop. Thus, all that I require is access to a projector so that others can view my screen as I point out the different features and uses of LINKSTER. I expect that setup will take all of a few minutes maximum.

During the execution of my demonstration, I will take a few minutes to describe the data sources and then describe each of the GUI elements of LINKSTER, indicating its purpose and use. Next, I will describe some of the current studies that we are using LINKSTER for and go through mock exercises that actual users have performed. One example would be looking at the context of a change in the source code, indicating the type of change (e.g. bug fix, feature addition, backport), and adding a free form annotation. Another would be manually linking a closed bug to the set of commits that close the bug and also to an email message discussing the bug. Lastly, I would demonstrate creating an association between a change in the repository and the code review performed via the mailing list. I would expect that the entire demonstration would take between 12 and 15 minutes, but could be pared down to 10 minutes if available time is an issue.