A Linguistic Analysis of How People Describe Software Problems

Andrew J. Ko, Brad A. Myers, and Duen Horng Chau
Human-Computer Interaction Institute
Carnegie Mellon University, Pittsburgh PA 15213
ajko@cs.cmu.edu, bam@cs.cmu.edu, dchau@andrew.cmu.edu
http://www.cs.cmu.edu/~marmalade

Abstract

There is little understanding of how people describe software problems, but a variety of tools solicit, manage, and analyze these descriptions in order to streamline software development. To inform the design of these tools and generate ideas for new ones, an study of nearly 200,000 bug report titles was performed. The titles of the reports generally described a software entity or behavior, its inadequacy, and an execution context, suggesting new designs for more structured report forms. About 95% of noun phrases referred to visible software entities, physical devices, or user actions, suggesting the feasibility of allowing users to select these entities in debuggers and other tools. Also, the structure of the titles exhibited sufficient regularity to parse with an accuracy of 89%, enabling a number of new automated analyses. These findings and others have many implications for tool design and software engineering.

1. Introduction

Many of the changes that occur in software maintenance are driven by the descriptions of software problems found in bug reports. These reports—which are also known as "problem reports," "modification requests," and a variety of other names—often translate directly into development tasks. For example, a report titled "info window sometimes has no scrollbars" indicates that the software may contain faulty code that should be repaired. Other descriptions might come from users or developers requesting new features, or identifying usability or performance issues.

Although bug reports have been used as a source of data for some tools [2, 3, 5, 7, 8], none of these projects have considered how people describe software problems. Therefore, we analyzed the titles of nearly 200,000 bug reports from five open source projects, discovering several useful trends. For example, most titles had the same basic content: an entity or behavior of the software (such as a user interface component or some computation), a description of its inadequacy, and the execution context in which it occurred. Also, nearly all of the noun phrases in the titles referred to visible entities, physical devices, or user actions, suggesting the feasibility of interaction techniques for selecting entities by direct manipulation. We also found that report titles can be accurately parsed, suggesting the feasibility of several new automated analyses.

These results have a number of implications for software engineering research. First, our results suggest designs for more structured problem report forms that better match people's phrasing of problems, while enabling tools to more easily reason about reports. The results also suggest new ways of specifying design requirements in a more natural way than current formal specification languages. Our results also find that the vocabulary problem frequently cited in HCI [6] is just as prevalent in software development, suggesting new requirements for search tools that require people to describe software behavior [15]. We also found several new types of questions that debuggers like the Whyline [9] should support, in order to allow developers to inquire about the full range of problems they discover.

Our results also suggest several ways that bug report titles can be explicitly analyzed in order to streamline software engineering workflow. Report titles can be parsed with an accuracy of 89% using a simple algorithm, enabling new types of automated analyses. Our results also suggest new strategies for separating problem reports from feature requests, grouping reports by the software quality attribute to which they correspond, and automatically assigning reports to developers [3, 12].

In this paper, we begin by describing related work and then our report corpus and the methods that we used for obtaining and preparing it for analysis. Then, we describe trends in the language, structure, and content of the descriptions, and we end with a discussion of the implications of these trends for tools, tool design, and software engineering.

2. Related Work

To our knowledge, no other work has specifically studied linguistic aspects of software problem descriptions. Twidale and Nichols [10] performed a qualitative study of usability bug reports, but their linguistic analyses were sparse and informal. Sandusky, Gasser, and Ripoche studied dependencies between bug reports, finding that they served an organizing role, but they did not study the content of the reports [14]. Several studies have used problem reports as data, for example, to better understand the process of software maintenance over time [5], to classify reports in order to assign them to developers [3], and to inform the design of new bug tracking systems [2]. Other studies have investigated similar version control data [7, 8], but they have mainly focused on the resulting changes to code and not the problematic behavior that motivated the changes.

Several studies have investigated linguistic aspects of other areas of software development to inform the design of new tools and languages. Pane et al. studied the language and structure of non-programmers' solutions to interactive and numerical programming problems, discovering a preference for certain language constructs, a tendency to misuse logical connectives such as *and* and *or*, and the common use of aggregate operations [11]. Begel and Graham studied features of programmers' speech about code in order to design new interaction techniques for creating code, finding several types of ambiguity [4]. In our work on the Whyline [9], we studied professional and non-programmers' questions about program failures (for example, "Why didn't Pac Man resize?"), but we only focused on high-level features of these questions, such as whether the question was negative (why didn't) or positive (why did), and whether programmers referred to multiple events and entities (such as "Why did Pac resize after eating the dot?"). There have also been more theoretical studies of linguistic aspects of software development in the area of semiotics [1].

3. Method

Our study focused on two research questions:

- *How varied are the nouns, verbs, adverbs and adjectives used to described software problems?*
- *What roles do these parts of speech play in identifying software problems?*

To answer these questions, we analyzed the *titles* of problem reports from several online bug tracking systems. These titles included statements like `Duplication of entries in package browser` and `crash if i try to clear cookies`. To address the first question, we determined the set of words of each particular part used in all titles. We compared this to the set of words of that part of speech in the electronic New Oxford American Dictionary (NOAD). For our second question, we manually classified the titles' words and phrases, generating descriptive categories. Unfortunately, we were unable to assess the reliability of our classifications with multiple raters.

Table 1. The five projects studied, the number of reports and unique reporters for each, and the date on which the project's reports were acquired.

project	# reports	# reporters	date acquired
Linux Kernel	5,916	3,296	Jan. 18, 2006
Apache	1,234	8,538	Jan. 18, 2006
Firefox	37,952	16,856	Jan. 17-18, 2006
OpenOffice	38,325	11,604	Jan. 18-19, 2006
Eclipse	90,424	9,175	Jan. 19, 2006
total	187,851	49,469	(44,406 unique)

We obtained our problem report data set from online bug tracking systems for several systems: a software development environment (*Eclipse*), a web browser (*Firefox*), a web server (*Apache*), an operating system kernel (*Linux*), and a suite of office applications (*OpenOffice*). These were obtained by downloading each project's bug database in comma-separated value format from the project's website. The reports for these projects come from diverse user and developer populations, the applications are implemented in various languages, and the projects all have unique communities and processes. The number of reports, the number of unique reporters, and the date of acquisition for each database are given in Table 1. The number of *unique* reporters was less than the sum of each project's number of reporters, suggesting that some reporters (11%) reported on multiple projects.

To prepare the five datasets for the analyses in this paper and also for future work, we determined common fields across all project databases, which included the title, and also the open date, priority, severity, assignee, reporter, status, resolution, product, component, and version. To aid our linguistic analyses, we then applied the Stanford probabilistic part-of-speech tagger [16] to each report title. For each word in each title, the tagger identified its part of speech (noun, verb, adjective, etc.), while accounting for words that have different meanings but identical spellings. The tagger that we used is reported to have 97% accuracy when used on a standard corpus of newspaper stories, meaning that about 3 of every 100 words are tagged incorrectly. We expected our accuracy to be lower, however, because the titles were very technical, often grammatically incorrect, and prone to misspellings. Our final data set is available for download at *http://www.cs.cmu.edu/~marmalade/reports.html.*

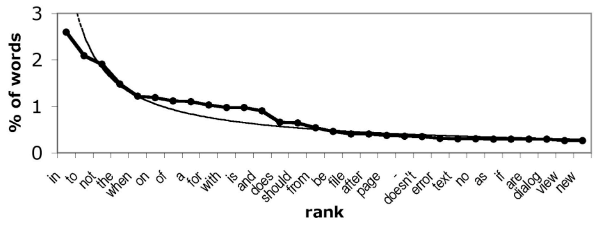

Figure 1. Frequencies of the 30 most common words.

4. Results

Overall, there were 123,417 unique words in the data set. Word frequencies generally followed Zipf's law, as seen in Figure 1.

Nouns and Noun Phrases

There were 82,181 distinct nouns in the dataset. Although the projects dictated what entities were described, we were able to identify categories that spanned all of the projects by using regular expressions to analyze the dataset's 82,181 distinct nouns. About 54% were proper nouns that represented code, file names, acronyms and version numbers. About 9% were quoted error messages and UI labels. About 7% were hyphenated, such as `file-format`, representing application-specific concepts or behaviors. About 3% were acronyms of data types, such as `PDF` or `URL`. Less than one percent were *abstract nouns* ending in `-ility`, `-ness`, and `-ance` that indicated some software quality; the most common of these are shown in Table 2.

Overall, there were about 4000 nouns consisting of only lower case letters. The eight most common were `file`, `page`, `error`, `text`, `dialog`, `view`, `menu`, and `editor`, which accounted for 5% of all the nouns used. The most frequent nouns of each project indicated its major kinds of entities. For example, the most common Eclipse nouns were `view`, `editor`, `dialog`, `file`, and `NPE` (null pointer exception), each occurring in about 3% of titles. The most common OpenOffice nouns were `document`, `file`, `text`, `page`, and `OOo` (the executable), each occurring in about 4% of titles.

To get an understanding of the kinds of noun *phrases* in the data, we sampled 100 titles and identified and categorized their 221 top-level noun phrases. We then sampled 1000 titles, and using the same categories, found less than 2% change in category proportions. The resulting categories are in Table 3. About 26% described some GUI component. The majority of these did not refer to the component itself, but to some functionality, described by its visual manifestation. For example, reporters referred to a search *dialog* to describe a problem with its search *algorithm*. About 23% were verbatim quotes of labels, filenames, commands, and error messages (but not necessarily within quotation marks). About 15% of noun phrases were application names, and about 8% described physical artifacts such as `mice`, `keyboards`, and `printer output`. About 7% described user actions, such as `clicking` or `changing color`, typically to indicate the event that resulted in some problematic system response. About 6% described some system-specific behavior, and another 6% described some visual attribute such as color, shape, order, or spacing. About 5% described some data type, typically to indicate the type of data on which some computation failed. The remaining 4% described some abstract concept such as `problem` or `issue`, or any of the software qualities from Table 2. These findings demonstrate that reporters described nearly all entities and behaviors by referring to some visual or physical entity or input event. This was even true for problems in Linux and Apache, where reporters described problems by relying on named entities, physical devices, and visible system behaviors such error messages, `boot` and `freeze`.

We expected variation in naming [6], and for unnamed entities, there was considerable. For example, commands to search in various applications were referred to in a variety of ways, including `find dialog`, `find`, `"Find..."`, `find command`, `finding`, `find/replace`, and `command to find`. We expected less variation in the named entities such as `NullPointerException`, but even they exhibited substantial variation. For example, we found nine variants of `NullPointerException`, and several, such as `NPE`, did not contain the word `null`.

Table 2. Qualities referred to most often and the relative proportions of each.

quality	% of qualities
performance	17
visibility	4
compatibility	6
usability	5
compliance	3
accessibility	4
badness	1
appearance	<1
(other)	60

Table 3. Categories of entities and behaviors to which reporters referred, and examples and relative proportions of each (based on 1000 titles).

entity or behavior	example	%
GUI component	**find toolbar** unexpectedly pops up	26
verbatim quote	**"firewire device not found"**	23
application name	**Apache - Tomcat - mod_Jk** 500 Internal Server Error	15
physical artifact	blue screen with **saa7134 tv tuner**	8
user action	**saving a webpage** as "Complete" messes with some tags	7
system behavior	Jasper performs **parallel compilations** of same JSP.	6
visual attribute	**Line backgrounds** and 'highlight current line'	6
data type	Cannot open **realvideo movies** in realplayer	5
abstract concept	**Performance:** slow object effects on solaris 9	4

Table 4. The nine most common verbs, the percent of titles in which each occurred, and examples of each.

word	%	example
is	13	Port specified in doc **is** incorrect
are		PNG icons **are** not transparent
be		BufferedLogs can't **be** disabled
does	6	Autocaption preview **does** not update
do		Tables **do** not import correctly from MS Word
using	2	Apache **using** 100% cpu
add	2	**add** ability to unzip into separate folders
fails	2	ScriptAlias **fails** with tilde in pattern
has	1	Export to PDF **has** no options

4.1 Verbs

There were 14,241 distinct verbs in the data set. Of these, there were about 1,000 distinct base forms of verbs that occurred more than twice throughout the titles. For comparison, there are 9,525 base forms of verbs in NOAD. The most common verbs and conjugations are shown in Table 4, accounting for 25% of all verbs used in the dataset.

Verbs played two roles in describing problems. The most common was to indicate the *grammatical mood*, which is a linguistic concept that refers to the intent of a sentence. Based on a categorization of a sample of 100 titles, 69% had a declarative mood, describing an undesirable aspect of the software, as in `UI is non-responsive for a long time`. These could be labeled bugs. About 23% were noun phrases with no mood, as in `compile failure in drivers/scsi`. About 5% were imperative, requesting some feature, as in `Add more detailed descriptions for errors` and 3% were subjunctive, describing a desire, as in `Introduce parameter should work for locals`. Mood was a matter of choice: for example, we found four reports for the same problem, each using a different mood: `F1 help missing`, `Add F1 help`, `Should have F1 help`, and `F1 help`.

The other role that verbs played was to indicate some computational task, such as `add`, `open`, `build`, `update`, `find`, `use`, `set`, `select`, `show`, `remove`, `create`, `load`, `get`, `save`, `run`, `hang`, `install`, `click`, `try`, `change`, `display`, `appear`, `move`, `crash` and `freeze`. These 25 verbs and their conjugations alone accounted for 20% of verbs used in the dataset, meaning that these verbs and the 9 in Table 4 accounted for 45% of all verbs used. The remaining 55% were more specific. In some cases, these verbs were no more informative than their more general synonyms. For example, `mod_rewrite` **`disrupted`** `by URLs with newlines` could have used `fails` instead. Other specific verbs, however, concisely identified qualities of a behavior that would otherwise have been cumbersome to specify generally. For example, to rephrase `location bar` **`desynchronizes`** `when closing tabs` would have required a lengthy and inaccurate wording such as "does not occur at the same rate."

4.2 Adverbs

Only 28% of the titles contained an adverb. Among these, there were 1,751 distinct adverbs, but based on hand-inspection of these, 13% were misspellings and 60% were misclassified because the sentence contained some higher level punctuated structure. For comparison, there are 6,137 adverbs in NOAD.

The remaining 476 words that were actually adverbs served to characterize the inadequacy of some entity (by modifying an adjective) or behavior (by modifying a verb). The most common adverbs `not`, `up`, `only`, `too`, `correctly`, `properly`, and `always`, accounted for 58% of adverbs used and indicated some behavioral inadequacy, as in `method completion doesn't` **`always`** `trigger` or `Dialog` **`too`** `big`. The adverb `not` occurred in 15% of titles. Other adverbs helped to characterize the *kind of entity*, as in `unable to search for` **`deeply`** `overridden method`. These adverbs helped to reduce the scope of the entity being specified by attributing it some unique characteristic. The least common use of an adverb indicated some quality attribute, like those in Table 2. For example, adverbs such as `slowly` signified a performance issue, and `quickly` generally signified a usability issue.

4.3 Adjectives

About 48% of titles contained an adjective. Of the 17,034 distinct adjectives in the dataset, and the 5037 that consisted of lowercase letters, only 2,447 occurred more than once. There were 2,695 hyphenated, application-specific properties such as `non-ascii`. For comparison, there are 21,316 adjectives in NOAD.

Adjectives served two purposes. The first was to help identify an entity, by attributing a characteristic to a noun phrase, as in **`large`** `file Downloads size wrong`. The most common of these included `same`, `other`, `multiple`, `empty`, `first`, `different`, and `current`, which accounted for 6% of the adjectives used. Almost all uses of these common adjectives indicated some part of a data structure that the software was processing incorrectly. About 84% of the adjectives used were rarely used domain-specific characteristics, such as `multicolumn`, and misspellings.

The second purpose of an adjective was to identify the problem by characterizing an entity or behavior's inadequacy, as in **`improper`** `warning for varargs argument`. The most common type of inadequacy was `wrong`, for which there were at least twenty synonyms; these accounted for 8% of adjectives used. The other problems identified by adjectives, which accounted for about 2% of adjectives used, referred to quality attributes like those in Table 2. For example, the word `slow` indicated a performance problem.

4.4 Conjunctions and Prepositions

About 69% of the titles in our dataset used at least one conjunction or preposition. There were several outliers: 871 titles had five or more, and one had fifteen:

```
after a few months of use, downloading or saving
of files or pictures from web pages results in
slowing of mozilla to the point of computer
lockup for up to 30 seconds or more.
```

In this section, we focus on conjunctions and prepositions that occurred in more than 1% of titles in our corpus, which represented 93% of those used throughout all titles in the dataset.

Table 5. Properties attributed to entities and behaviors, the words used for each type, the percent of titles containing each word (based on a sample of 100 titles), and an example of each in use.

property	word	%	example
group	and	7	breakpoints in archives **and** files
	or	2	xconfig **or** gconfig not working
trait	with	8	doesnt save files **with** long names
	without	1	CVS timeout on project **without** tags
	that	1	Word file **that** is borked in OO.o.
place	from	2	segfault on search **from** toolbar
	under	1	Border isn't removed **under** tab
	at	1	font changes **at** end of line
owner	of	9	improve explanation **of** 'proxy'
purpose	for	8	Beep **for** typos
source	from	2	panic when copying file **from** cdrom
type	as	2	inserting charts **as** links
amount	by	2	last modified dates off **by** an hour
scale	at	1	Pages not tiled **at** low zoom level
action	by	1	breakpoint **by** double-click off by 1

Table 6. Information used to contextualize a problem, the words used for each type, the percent of titles containing each word (based on a sample of 100 titles), and an example of each in use.

context	word	%	example
entity	in	20	Typo **in** the error message
	on	10	error saving file **on** samba share.
	of	9	no PDF export **of** graphics
	by	2	Path explosed **by** multiple ' in url
	under	1	junit tag **under** 1.5 gives errors
	between	1	Rendering problems **between** rows
event	when	10	image shrinks **when** protecting sheet
	after	3	Firefox freezes **after** reading PDF
	if	2	poweroff fails **if** "lapic" forced on
	during	1	Toolbar displays **during** slideshow
	at	1	xerces error **at** startup
	on	1	initrd refuses to build **on** raid0
	and	1	Manager accesses drive **and** freezes
	while	1	freezes **while** opening this document
contrast	but	1	Exporter error **but** saves correctly
	than	1	i need more **than** 32000 rows

The conjunctions and prepositions in the dataset served two roles. The first was to attribute one of the ten properties shown in Table 5 to an entity or behavior. For example, the words `and` and `or` served to group entities, and the words `with`, `without`, and `that` helped to indicate some trait (or missing trait) of an entity. These properties generally helped indicate the type of data types on which some computation failed. The words `from`, `under`, and `at` indicated some visual location, helping to specify a particular part of a user interface. The word `of` helped to select some smaller part of a larger entity, as in `explanation` **`of`** `'proxy'`. The word `for` described the purpose of some entity, and was generally used to clarify large classes of entities, as in `Beep` **`for`** `typos`. The other five types of information were less common, so we do not discuss them in detail.

The second role of conjunctions and prepositions was to describe the context of a problem. There were generally three types of context, which we list in Table 6. The most common type of context was some *entity*. The words `in`, `on`, `of`, `by`, `under`, and `between` all helped to indicate the entity that exhibited some problem. For example, `Typo` **`in`** `the error message` indicates that the `error message` entity that contained the `Typo` problem. The other common problem context was some *event*. The words `when`, `after`, `if`, `during`, `at`, `on`, `and` and `while` all indicated some situation or action which seemed related to the problem. For example, `Firefox freezes` **`after`** `reading PDF` indicates that the `reading PDF` action led to `Firefox freezes`. The last type of context was *contrast*, which was indicated with the words `but` and `than`. These words presented some property or behavior of an entity, and then indicated some way in which it should differ. For example, `Exporter error` **`but`** `saves correctly` indicates that the `Exporter` gives some error message that is inconsistent with its actual behavior.

Conjunctions and prepositions were central to determining the structure of a report title. For example, consider the structure of `kernel panic` **`when`** `copying big file` **`from`** `cdrom`. The `when` clearly defines the structure of the sentence as a `kernel panic` that occurs during the `copying big file from cdrom` event. The `from` indicates that the `cdrom` is the source of the `big file`. Of course, there is some ambiguity in the precedence of the various words because usage can vary. For example, in the title above, `when` has the highest precedence, but in the title `Custom styles in CSS` **`when`** `embedded in multiple DIVs don't track`, the `when` has lower precedence than the verb `don't`, because it is used like the word `that` to indicate some characteristic of the `CSS`. In general, however, the usage of the these words seemed fairly consistent, so tools that attempt to do basic parsing based on conjunctions and prepositions may be accurate most of the time.

5. Discussion

The regularities in the structure and content of the problem descriptions in our dataset have a variety of implications for software engineering tools. What follows are several potentially tractable design ideas that are motivated by our results.

5.1 Soliciting More Structured Titles

One implication of our results is that bug report forms could be redesigned to structure the information that reporters naturally include in report titles, making it easier for tools to analyze reports. Our results indicate that the content of these redesigned forms should consist of descriptions of (1) a software entity or an entity behavior, (2) a relevant quality attribute (3) the problem, (4) the execution context, and (5) whether the report is a bug or feature request. For example, the natural language title `toolbar tooltips take too long to appear when hovering` could have been reported in a more structured manner as follows:

(1) *entity/behavior:* `toolbar tooltips appearing`
(2) *quality:* `usability`
(3) *problem:* `slow`
(4) *context:* `when hovering`
(5) *bug or feature:* `bug`

Soliciting reports in this way essentially places the burden of parsing on the reporter, in order to simplify analyses of the report. Our results point to a number of ideas that could offset this burden. For example, the *quality* could be a pre-defined list of software quality attributes commonly used in software engineering practice. The *problem* could be a list of common adjectives that refine the type of quality specified. Furthermore, given the diversity of descriptions of entities and behaviors, the *entity or behavior* and *context* should just be free form text, to allow reporters to accurately describe the subject and context. These fields could then suggest similar phrases based on past reports as reporters type, to help both reporters and tools more easily identify duplicate reports.

5.2 Parsing Report Titles

An alternative to soliciting more structured reports from reporters is to instead parse the natural language titles, inferring their structure. The benefit of this approach is that it would generate a more detailed structure than that discussed in the previous section, while not imposing any burdens on reporters. The extra detail could make it easier to identify different types of context and individual noun phrases in report titles, among other things.

Our results show that many of the titles were not grammatical, so we could not use research on natural language "chunkers", which rely on proper grammar to parse. Instead, to test parsing we implemented a custom recursive scanning parser, where each scan of a title looks through a ranked list of parts of speech to split on, splits on the first kind found, and then recursively scans the resulting parts in a similar manner. Our results suggested a particular ranking. First the scanner checks for punctuated sentences, such as `Performance: slow object effects on solaris`. Next, it checks for verbs indicative of a declarative structure (such as `is` and `does`). Then, the scanner checks for conjunctions and prepositions, because they helped structure noun phrases. The last step splits on common verbs, with the resulting word sequences representing the noun phrases. To test the accuracy of our parser, randomly sampled 100 titles, and compared our parser's top-level structure against the top-level structure of our own hand-parsing of the titles. This showed the parser achieving 89% accuracy. This could easily be improved by increasing the accuracy of the part of speech tags (which had at least 3% error and likely more), which would be done by retraining the tagger on the report titles. There may also be a better ranking than the one used by our algorithm.

5.3 Identifying Problems and Requests

An important part of managing report databases is separating problem reports from requests. Our results suggest that one way to perform this classification automatically is to use the mood of the report title as an indicator. To investigate this, we analyzed the 38,325 OpenOffice reports, which included reporters' hand-classifications of reports as a "defect" or some types of request. We applied a standard decision tree algorithm to classify each report as a defect or not, using only features of the title that indicate the mood of a sentence, such as past tense and active verbs. This led to an average test accuracy of 79%, which is 3% higher than the baseline incidence of a defect in the dataset of 76%. Although this is only a marginal increase, the classifier could be further improved by improving the part of speech tag accuracy, as described above. Features based on the structure of the report titles, derived by the parsing described in the previous section, could lead to further gains. Of course, one issue with this approach is that what reporters described as bugs or requests may not correspond to the developers' perspective, because opinions are likely to differ on what the software's intended behavior is or should be. Mood may also be culturally specific, and so studies should investigate for what types of problems automate detection could be useful.

5.4 Identifying Quality Attributes

A common software engineering practice is to create a ranked list of quality attributes for a project in order to prioritize development and maintenance efforts. Therefore, a tool that could automatically classify reports by the quality attribute to which they correspond would be extremely helpful. The results of our analyses provide many insights for the design of such a tool. First, although only a fraction of the report titles explicitly referred to quality attributes, about 62% of the titles contained at least one adverb or adjective that may have referred to a software quality, such as `slow` or `slowly`. In addition to suggesting the feasibility of such a classification tool, our results also suggest several features that might be helpful in performing the classification. Many adjectives had specific usage. For example, `slow` always indicated a performance problem, and the more common adjectives, such as `same`, `different`, `first` and `empty` typically indicated some failure on a particular configuration of data. Titles with references to conventional user interface components, such as buttons and toolbars typically identified some usability problem or user interface defect. Future studies could perform an investigation into these common usages in order to identify suitable features for classification.

5.5 Assigning Reports to Developers

Another difficulty in managing reports is how to determine the most qualified developer for each report. There have been attempts to automatically match developers with reports, based on the full report text and to whom reports have been assigned in the past [3]. Although these approaches show promise, their accuracy ranged from 6-64%. One possible reason for this is that the "appropriateness" of a report for any given developer depends on what functionality the report regards and what functionality the developer has expertise in. Perhaps the full text reports used by these systems contained too many words that were irrelevant to the functionality being discussed in the report, distracting the classifier from detecting the functionality. Our results suggest that noun phrases that may be less noisy than the full report text, because they tended to correspond to system functionality.

5.6 Identifying Duplicate or Similar Reports

Problem reporters are typically responsible for finding similar or duplicate reports to associate with a new report, which can be very time consuming [10]. Our results suggest that tools could be designed to extract and cluster noun phrases and execution contexts from existing reports' titles, and present reports in the cluster most similar to the reporter's current description of the problem. Then, rather than having to do a raw text search for related reports, similar reports could be presented to the reporter based on the title they supplied. Furthermore, the reporter would also be able to get a sense for the types of phrases are already being used to describe various entities and behaviors, indirectly improving the clustering algorithms.

5.7 Asking Questions about Problems

The Whyline [9] has been shown to reduce debugging time by allowing programmers to ask questions about program output. In generalizing the Whyline to more professional languages and more types of software, however, it is important to have a detailed understanding of the full range of questions that people ask about software behaviors.

Our results point to many new requirements. For example, temporal context was often specified in our dataset, using words such as `when`, `during`, and `after`, in order to indicate the situation in which a problem occurred. Therefore, the Whyline should offer techniques for selecting a segment of an execution history to specify the *time* or *event* after which some problem occurred, allowing the tool to generate more specific answers.

The Whyline currently only allows questions about the most recent execution of an output statement, but many of the behaviors described in our dataset represented computations or system actions that executed over time or executed repeatedly. Therefore, new techniques must be designed to allow questions about multiple executions of an output statement, or even patterns of output, in order to help developers refer to higher-level behaviors in their questions.

The Whyline is also limited to questions about a single entity. However, based on our investigation of conjunctions and prepositions, there are a number of common ways that entities were identified relative to other entities, such as by group, location, source, and action. Future tools should allow questions about multiple objects, allowing users to specify one of the various types of relationships listed in Table 5. For example, a user should be able to ask, "Why didn't data from this object appear after this click event?"

Our results regarding the various quality attributes that users referred to also suggest the need for a whole new class of debugging tools for investigating different software qualities. Current debugging tools only focus on helping developers deal with correctness and performance issues. Future work should consider what kinds of tools would help developers debug qualities such as consistency, robustness, and visibility.

5.8 Are Hand-Written Reports Necessary?

One way to think about problem reports is as a concise, but imprecise representation of a set of program executions in which a particular entity had an inappropriate characteristic or behavior. Whoever reports the problem must translate these executions into a description, and whoever handles the description, such as a tester or software developer, must translate the description back into the set of executions that originally inspired the description. If reporters had tools that allowed them to capture these executions, they would only have to supply a description of the *expected* behavior, annotating the captured data with the problematic entities and context (through direct manipulation as described before). If such tools were integrated into the software itself, even end users would be able to generate precise reports with little effort. Given our evidence that nearly all entities were identified via something on-screen, one could imagine tools that allow users to point to an object to identify the entity. Testers and software developers might then be able to use these reports to automatically determine the relevant source code, like the concept of a *concern* [13].

6. Limitations

There are several limitations of our data and results. We were unable to include any reports from closed source projects, which may have stricter standards than the projects that we studied. We do not know whether the reports in our data were created by developers or users, so we cannot describe the population of people who created our dataset. It is also possible that the reports did not include “in the moment” problems that developers encountered while developing, and such problems may be described in different ways. We do believe that some of these probably were included, given evidence that many reports are written as reminders or because a problem turns out not to be the developer’s responsibility to fix.

7. Conclusion

This study is just the beginning of a larger effort to better understand how software problems are described, and how tools might help everyone involved in software engineering to better manage and utilize bug reports. In our future work, we hope to investigate many of the analyses and tool ideas presented in this paper, and analyze higher level issues in bug reporting processes, such as the difference between intended, expected, and actual behavior.

8. Acknowledgements

We would like to thank the contributors to the open source projects that we studied, and also the reviewers for their thorough and insightful reviews.

This work was supported by the National Science Foundation, under NSF grant IIS-0329090, by the EUSES consortium under NSF grant ITR CCR-0324770, and by an NDSEG fellowship.

9. References

[1] P. B. Anderson, B. Holmqvist, and J. F. Jensen, *The Computer as Medium*. Cambridge: The Cambridge University Press, 1993.

[2] G. Antoniol, H. Gall, M. D. Penta, and M. Pinzger, Mozilla: Closing the Circle, Technical University of Vienna, Vienna, Austria TUV-1841-2004-05, 2004.

[3] J. Anvik, L. Hiew, and G. Murphy, Who Should Fix this Bug?, *ICSE* 2006, Shanghai, China, 361-368.

[4] A. Begel and S. L. Graham, Spoken Programs, *VL/HCC* 2005, Dallas, Texas, 99-106.

[5] M. Fischer, M. Pinzger, and H. Gall, Analyzing and Relating Bug Report Data for Feature Tracking, *WCRE* 2003, 90-99.

[6] G. W. Furnas, T. K. Landauer, L. M. Gomez, and S. T. Dumais, The Vocabulary Problem in Human-System Communication, *CACM*, 30, 964-971, 1987.

[7] H. Gall, M. Jazayeri, and J. Krajewski, CVS Release History Data for Detecting Logical Couplings, *IWPSE* 2003, 13.

[8] D. M. German, An Empirical Study of Fine-Grained Software Modifications, *ICSM* 2004.

[9] A. J. Ko and B. A. Myers, Designing the Whyline: A Debugging Interface for Asking Questions about Program Behavior, *CHI* 2004, Vienna, Austria, 151-158.

[10] D. M. Nichols and M. B. Twidale, Usability Discussions in Open Source Development, *HICSS* 2005, 198-207.

[11] J. F. Pane, C. A. Ratanamahatana, and B. A. Myers, Studying the Language and Structure in Non-Programmers' Solutions to Programming Problems, *IJHCS*, 54, 2, 237-264, 2001.

[12] A. Podgurski, D. Leon, P. Francis, W. Masri, M. Minch, J. Sun, and B. Wang, Automated Support for Classifying Software Failure Reports, *ICSE* 2003, Portland, Oregon, USA, 465-475.

[13] M. P. Robillard, Representing Concerns in Source Code, *Department of Computer Science*, *University of British Columbia*, Vancouver, Canada, November 2003.

[14] J. Sandusky, L. Gasser, and G. Ripoche, Bug Report Networks: Varieties, Strategies, and Impacts in an OSS Development Community, *MSR* 2004, Edinburgh, Scotland.

[15] J. Stylos, Designing a Programming Terminology Aid, *VL/HCC* 2005, Dallas, Texas, 347-348.

[16] K. Toutanova, D. Klein, C. Manning, and Y. Singer, Feature-Rich Part-of-Speech Tagging with a Cyclic Dependency Network, *HLT-NAACL* 2003, 252-259.