TODO or To Bug: Exploring How Task Annotations Play a Role in the Work Practices of Software Developers

Storey, M.A.; Ryall, J.; Bull, I.; Myers, D.; Singer, Janice

National Research Council Canada
Conseil national de recherches Canada

Institute for Information Technology
Institut de technologie de l'information

NRC · CNRC

*TODO or To Bug: Exploring How Task Annotations Play a Role in the Work Practices of Software Developers **

Storey, M.A., Ryall, J., Bull, I., Myers, D., Singer, J.
May 2008

* published in the Proceedings of the International Conference on Software Engineering 2008 (ICSE 2008). Leipzig, Germany. May 10-18, 2008. NRC 50378.

Canada

TODO or To Bug: Exploring How Task Annotations Play a Role in the Work Practices of Software Developers

Margaret-Anne Storey[1] Jody Ryall[1] R. Ian Bull[1] Del Myers[1] Janice Singer[2]

University of Victoria[1]
Victoria, BC, Canada

National Research Council[2]
Ottawa, ON, Canada

ABSTRACT

Software development is a highly collaborative activity that requires teams of developers to continually manage and coordinate their programming tasks. In this paper, we describe an empirical study that explored how task annotations embedded within the source code play a role in how software developers manage personal and team tasks. We present findings gathered by combining results from a survey of professional software developers, an analysis of code from open source projects, and interviews with software developers. Our findings help us describe how task annotations can be used to support a variety of activities fundamental to articulation work within software development. We describe how task management is negotiated between the more formal issue tracking systems and the informal annotations that programmers write within their source code. We report that annotations have different meanings and are dependent on individual, team and community use. We also present a number of issues related to managing annotations, which may have negative implications for maintenance. We conclude with insights into how these findings could be used to improve tool support and software process.

Categories and Subject Descriptors

D.2.3 [**Software engineering**]: Coding tools and techniques.

General Terms

Documentation, Human Factors.

Keywords

Task annotations, work practices, source code comments.

1. INTRODUCTION

Software development is a highly collaborative activity that requires teams of developers to continually manage and coordinate their programming tasks. The management of tasks and subtasks is an important aspect of what Computer Supported Cooperative Work (CSCW) researchers call "Articulation Work" [20]. As Bannon and Schmidt note: "in 'real world' cooperative work settings… articulation work becomes extremely complex and demanding" [1]. Consequently, people develop techniques and protocols for reducing the overhead cost and complexity of articulation work.

ICSE'08, May 10–18, 2008, Leipzig, Germany.

Developers use a variety of tools to support collaborative task management in software projects. Popular tools include wikis, configuration management systems, bug tracking and issue tracking systems. Although much effort has been expended developing these tools both by industry and the research community, there is surprisingly little known about the work practices these tools support (notable exceptions include [2] and [6]).

In addition to tools that provide formal coordination mechanisms within teams, software developers use informal devices and develop processes to support their task management activities. In particular, they use annotations to manage their tasks. The prevalence of this activity has resulted in tool support within integrated development environments (IDEs) for navigating these customized annotations.

Through our research with software developers we have noted a gap between the more formal task management mechanisms supported by tools and the informal annotations that developers place in their source code. The long term goal of our research is to develop tool support that bridges this gap. However, we have come to realize that there is a lack of knowledge on how task annotations, embedded as comments in the program source code, play a role in the work practices of software developers. Without such an understanding, researchers designing tool support for this kind of articulation work will not be able to interpret and understand the effects a new tool may have on how programmers manage tasks.

To address this lack of knowledge we conducted an empirical study to explore the work practices of software developers surrounding task annotations. Specifically we considered task annotations embedded within source code comments. We did not expect to find a common theory that describes how programmers use task annotations, but instead were interested in the varied ways that this commenting feature has been appropriated by software developers to manage their personal work and coordinate with other developers.

We followed a multi-phase mixed methods approach in our research. In the first phase, we surveyed developers using the popular Eclipse IDE to find out whether they author task annotations, and if they do, what were the types and uses of these annotations. As the foundation for understanding how task annotations are used in industrial collaborative projects, we also extracted task annotations from ten open source projects. The key result from this first phase is that developers adopt very different tools and protocols both between and within teams for managing their tasks.

In the second phase, we narrowed our view and conducted contextual interviews with developers on three open source projects. We augmented the data collected from the interviews

with a deeper and targeted analysis of multiple versions of developer comments extracted from the projects' source code repositories. An analysis of this qualitative and quantitative data revealed several themes that have implications for tool design as well as insights into task management processes followed by developers. Before describing our empirical study and our findings, we review related work.

2. BACKGROUND

The focus of the empirical study reported in this paper is to explore how annotations embedded within source code are used for managing tasks. To our knowledge, with the exception of Ying's work [25] mentioned below, there is no published academic research on this topic. Instead, we review two strands of related research: 1) empirical work that investigates how comments are used and managed in software engineering, and 2) recent research on tool support for task management.

2.1 The Use of Comments in Source Code

Comments are a generic type of task annotation, where programmers insert documentation directly into source code. Researchers from several communities have studied comments and how to manage them.

Within the programming language community, the role of comments has been explored. With a few exceptions, such as Java's @javadoc construct, languages typically treat comments as white space. Authors have questioned this treatment, arguing for a more cogent use of comments in programming language design [7][9]. Almost twenty years ago, Kaebling [9] challenged programming language designers, "surely there is a better way to specify location-dependent information than scattering undirected one-dimensional strings throughout a file." This challenge has yet to be addressed, although some researchers have looked at tool support or ways to treat source code as a structured document (e.g., [11][16]).

In the software engineering community, several researchers have conducted experiments to understand how source code comments are used. Most of these studies looked at how well student subjects perform when programs are commented versus not-commented [22][23][24]. In general, students performed better when code was commented. Although in one study, students only performed better when low-level, as opposed to high-level, commenting was present [14]. In another experiment, Marin [12] found that when programmers were asked to add code to an existing program, they were more likely to comment newly inserted code if the previous code was heavily commented, suggesting that there are (perhaps implicit) social factors relating to commenting. In a survey of professional programmers [4], the authors found that comments were the second most used documentary artifact when understanding source code, with the code itself being the primary artifact.

A small amount of research within the mining software repositories (MSR) community has focused on comments in source code. Jiang and Hassan [8] studied the evolution of comments in PostgreSQL. They found that, with the exception of some fluctuation early in development cycles, the amount of comments remains constant. This finding was mirrored in a study by Fluri *et al.* [5]. In two of the three open source projects they looked at, commenting was relatively constant over time. However, there was variability in that one project increased the ratio of commented to non-commented lines of code over time. These authors also found great variability in the other two attributes they looked at: what is commented, and whether comments co-evolve with the source code, where the projects acted in almost contradictory ways. Tan *et al.* [21] recently showed that out-of-date comments may help reveal the locations of bugs.

Little research has been done to understand how task annotations embedded within source code play a role in software development. Ying *et al.* [25] is one exception. These authors conducted a preliminary study where they analyzed the task annotations from an Eclipse open source project. Based on their analysis, they developed a usage taxonomy for task comments. Our work expands upon their research, analyzing multiple projects and interviewing developers within open source projects to gain insights into actual developer processes.

2.2 Modern Tool Support for Task Management

Modern IDEs provide various approaches for managing developers' tasks. Tools such as Eclipse and Visual Studio support bookmarks and navigation of task annotations embedded in comments such as TODO, FIXME, and XXX. Eclipse also allows programmers to define new task annotations terms and view these in a separate Task View, facilitating navigation and browsing.

Issue tracking systems, such as Bugzilla (www.bugzilla.org) and Jira (www.atlassian.com), provide more structured support for task management. Task management activities and storage take place outside the source code, but contain links or references to the code. Task activities are visible to anyone on the project with access to the tracking system. When considering which issues programmers enter in these systems, it is not always clear what developers consider a bug or issue worthy of input, nor how bugs are contributed by internal project developers versus community members.

Eclipse is a popular platform for tool research and several task management plug-ins have been developed for it. The Mylyn (formerly Mylar) project combines a degree of interest model with task management facilities [10]. Mylyn also allows linking of tasks to issue tracking systems. Mylyn has received wide adoption, although its features do not suit all work practices. Even with Mylyn, traditional task annotations are inserted within the code, suggesting there is still a need for these types of annotations.

There has also been recognition of the importance of tool support for collaborative tasks. The Jazz project (www.jazz.net) has integrated its work item tracking system with the code repository. Built on top of Eclipse, Jazz also supports task annotations, but there is no integration of these annotations with work items.

Other tools such as TagSEA [19] and ConcernMapper [15], while not explicitly designed to manage tasks, can aid in the management and navigation of information structures. ConcernMapper allows the developer to link software artifacts to a concern, for future navigation, without changing the comments themselves. TagSEA allows developers to tag related code by adding keywords within comments (similar to, but less structured than task annotations). Our research with TagSEA indicated that early adopters are using it to support task management [18]. It is this observation that prompted the research reported here.

To further develop these tools, we need to better understand how developers use comments. The research on commenting to date suggests that code comments influence comprehension and that there are potentially cultural aspects to how comments are added. It may also be possible to learn about the software development process by examining archival data from a project's memory stored in a repository. We designed a study to explore these findings, with specific consideration for how comment annotations impact task management. The design of our study is described next.

3. RESEARCH DESIGN

We address our research goal of revealing how software developers use task annotations by investigating the following research questions:

1. Which annotations do developers create to support their programming tasks?
2. Why do developers create these annotations?
3. How do these annotations support developers' informal and formal work practices?
4. Are task annotations kept up-to-date or are they forgotten as the code evolves?
5. What processes do the developers use for managing annotations?

Our research followed a mixed methods design [3] with two distinct phases, summarized in Table 1. The first phase was predominantly an exploratory phase with collection of quantitative data. We conducted a survey with professional developers that use the Eclipse IDE. The survey was designed to provide insights on questions 1-3. During this first phase we also extracted task annotations embedded within code comments from ten open source projects within the Eclipse and Apache domains. The high level analysis of this data provided insights on questions 1 and 4.

In the second phase, we moved from an exploratory mode of research to an explanatory one, involving the analysis of qualitative and some quantitative data. We conducted interviews with developers from three open source development projects to obtain a more detailed understanding of why developers create such annotations and how they are used in their personal and team work practices. The interviews provided detailed narratives that contribute to answering research questions 1-5. The interview data was augmented by extracting and analyzing multiple versions of archival data from the projects, providing further information on questions 1 and 4.

Table 1. A mixed methods research design

Research Questions	1	2	3	4	5
Phase 1 (exploratory):					
Survey	X	X	X		
Analysis of annotations from ten projects	X			X	
Phase 2 (explanatory):					
Interviews	X	X	X	X	X
Multiple version comment analysis from three projects	X			X	

4. DATA GATHERING AND RESULTS

In this section we describe the data collected and results from the two phases of our research methodology. Sections 4.1 and 4.2 describe Phase 1, and Sections 4.3 and 4.4 describe Phase 2. We used the popular Eclipse IDE for all data collection because it has built-in support for task annotations. Eclipse task annotations are associated with a specific set of keyword tags (TODO, XXX, and FIXME) that can be customized by the developers. For clearer presentation of results and findings, we refer to Eclipse task tag annotation as "TODOs" in the next two sections.

4.1 Survey

We prepared an online survey that asked software developers how they comment their code, recruiting participants from a high traffic Eclipse community blogging site (planeteclipse.org). We collected 81 responses between April 4 and May 13, 2007. The goal of the survey was to provide insights on research questions 1-3. The survey asked 15 multiple choice and open-ended questions.

The results confirmed that many developers make use of task annotations, but in varying ways. The survey asked questions on keyword usage within a team, the use of bookmarks, and how developers navigate to annotations. A selection of results is presented below (see tagsea.sourceforge.net/research.html for further details).

Survey respondents reported working on proprietary (37%) and open source projects (14%), with 49% of respondents saying they worked on both. Our analysis found no noticeable difference in results, based either on this measure or the size of the team. Table 2 shows whether teams agree on a common set of keywords (note that only 65 respondents worked on a team).

Table 2. When collaborating on a team has your team agreed to use the same keywords?

	# of respondents (N=65)
I use my own keywords	11 (17%)
I use a mixture of my own keywords and my team's keywords	14 (22%)
My team has an informal agreement to use the same keywords	32 (49%)
My team has a formal agreement (or coding practice) to use same keywords	8 (12%)

A recurring issue that developers faced was whether to store their annotations in the code or privately in their workspace. Eclipse provides Bookmarks as an alternative mechanism for saving code locations. Bookmarks are not stored in the code, but instead reside in the user's workspace. In the survey, we were interested in finding out if developers use this feature. The answers to this question are shown in Table 3.

Table 3. Do you use Eclipse bookmarks in your source code?

	# of respondents (N=80)
Never	43 (54%)
Rarely	24 (30%)
Sometimes	12 (15%)
Often	1 (1%)

We were also curious about the vocabulary terms the Eclipse developers tended to use for task annotations. The results from this question are shown in Table 4.

Table 4. Which of the following Eclipse task tags do you use? (Select all that apply)

	# of respondents (N=79)
TODO	77 (98%)
FIXME	34 (43%)
XXX	12 (15%)
OTHER	11 (14%)
HACK	6 (8%)

We noted from examining source code that many developers add details to the TODOs they created, such as their name, date and bug ID. We asked about this activity in the survey and the results are shown in Table 5.

Table 5. Do you add any additional details to your comments? If so, what details do you add? (Select all that apply.)

	# of respondents (N=70)
Reference to another class, method, plug-in, or module	45 (64%)
My name or initials	36 (51%)
Bug id	31 (44%)
URL	21 (30%)
Date	13 (19%)
None of the above (I do not add additional details)	9 (13%)
Memorable keywords	7 (10%)

4.2 Task Annotation Extraction

As a preliminary step in our exploration of how (or even whether) task annotations are used, we extracted task annotations from ten open source Eclipse projects. The versions of the code analyzed were extracted from the CVS repository in July 2007 and updated in September 2007. We recognize that the presence or absence of tasks could be highly dependent on the time in the development cycle of each project, and limit the use of this data to help us understand if TODOs are used and how prevalent they are in selected projects. Some of the results from this extraction are presented in Table 6.

Table 6. Task tag usage in Java files of selected projects

	lines of code	todo	fixme	xxx	revisit	total
JDT.UI	693,683	417	15	83	0	**515**
Xerces-J	246,122	30	1	25	455	**511**
MYLYN	200,325	440	3	43	0	**486**
SWT	521,603	265	48	0	0	**313**
BIRT.CHART	324,826	167	2	0	0	**169**
PDE.UI	193,945	89	3	1	0	**93**
EQUINOX	73,403	75	4	2	0	**81**
EMF	670,933	23	0	0	4	**27**
JFACE	93,835	11	0	6	1	**18**
UI.FORMS	23,996	4	0	0	0	**4**

Corresponding with the results from our survey, we saw that the TODO term made up a clear majority of the task annotations found in the code. The other keywords that are configured by default in Eclipse (FIXME and XXX) are used significantly less. Custom tags, such as REVISIT, INTRO, and CONTEXTLAUNCHING were found in some of the projects. Initials were also commonly used to mark locations in the code. This data is interesting, as it shows that these kinds of annotations appear in a range of software projects. In displaying this data, we realize that there is a possible effect caused by the size of the team, the stage in the project's lifetime, and other processes.

Another factor that may influence the presence of tasks in the source code is auto-generation facilities in an IDE. We analyzed the task annotations to determine how many of these auto-generated TODOs get committed to the source control system. In seven of the ten projects, these annotations were partially or completely removed. Auto-generated annotations exclusively use the TODO keyword, which may also be a reason for their prevalence over other keywords.

Table 7. Percent of task annotations that are auto-generated

Project	Auto-generated
JDT.UI	78 of 515 (15%)
Xerces-J	1 of 511 (<1%)
MYLYN	46 of 486 (10%)
SWT	0 of 313 (0%)
BIRT.CHART	92 of 169 (54%)
PDE.UI	1 of 93 (1%)
EQUINOX	0 of 81 (0%)
EMF	0 of 27 (0%)
JFACE	0 of 18 (0%)
UI.FORMS	0 of 4 (0%)

4.3 Developer Interviews

We interviewed four software developers from three Eclipse projects. To preserve the anonymity of the developers interviewed, we refer to these projects using fictitious names: MiddleWare, Backend, and UserInterface. These projects were selected using convenience sampling [3], as they were open source Eclipse projects and access was available to their developers. Although the developers contributed to open source projects, they did so as part of their professional work positions. All three projects have been developed in the open for at least four years and each has had at least one million downloads.

The interviewer was somewhat familiar with the three projects and had a list of the TODOs present in the source code of each of the projects, which were referred to during the interviews. Two members of the MiddleWare project were interviewed together. Interviews lasted from 30 minutes to one hour. Audio recordings of these interviews were transcribed to facilitate later analysis.

Our analysis approach involved three investigators individually coding each of the interviews, and then grouping and regrouping codes to reveal themes. The investigators then agreed on a set of themes that encompassed their shared observations. The themes were checked with the interviewer to verify that the transcription and interpretation we arrived at resonated with the interviewer.

In the following subsections, we present a brief overview of each project and the key work practices of the developers related to their use of task annotations. Additional pertinent insights gathered from the interviews are described in Section 5.

4.3.1 Middleware Interview

The middleware project provides an open source framework and API. Clients of the framework subclass existing features to adapt and customize the framework to their needs. There are very few internal packages that are not accessible by the user community. Because of this, the code is often examined by its users. The developers of this project make heavy use of Bugzilla to track feature requests, bugs and other tasks. The developers have a strict policy that each code change must be directly linked to a Bugzilla entry and each commit comment must follow a specified syntax. Source control is managed through CVS and all changes are reviewed using the Eclipse compare tools. While the project is split into several components and each developer is charged with maintaining a specific part of the code, all three developers are well versed in the entire project.

The two developers interviewed (referred to as Middleware Dev1 and Middleware Dev2) rarely make use of TODOs or other task annotations, but discussed adding initials or using bookmarks as a way of indicating tasks to be completed. On occasion, a developer who was not interviewed by us, included their initials as a way of "signing" a comment. Bug numbers were not put in the code because developers saw this as "cluttering the code".

4.3.2 Backend Interview

The Backend project is also an open source API; however, sub-classing and extending existing functionality is not as common as in the Middleware project. Users can simply interact with the published interface to make use of the features provided by the toolkit. This project also uses an issue tracking system, but it is not a requirement that all code changes be linked to an issue number. There is no formal syntax used in the commit comments and most comments are high level descriptions of the changes. No new features are currently being added to the project and code changes are limited to bug fixes and maintenance.

One developer on this project was interviewed (Backend Dev1) and he indicated that the developers on this project made use of the keyword REVISIT to indicate that something may need to be addressed in the future. Several reasons were given for marking with this keyword including: 1) a suboptimal solution had been implemented, 2) an incomplete solution had been implemented, and 3) no solution had been implemented and it is uncertain if a solution is needed. The developer indicated that the REVISIT keyword is useful in the short term, but if the code is not reviewed promptly it will remain indefinitely. While the usefulness of a REVISIT tag is relatively short lived, the developer pointed out that use of this keyword can be helpful in the future when trying to understand why a piece of code does not work as expected.

4.3.3 UserInterface Interview

The UserInterface project provides a set of tools for developers. Unless users are actively contributing new features to the code, there are very few reasons to examine the source. There are seven committers, three of whom contribute on a daily basis. The project uses CVS for source control and Bugzilla for issue tracking. There is a policy that each code check-in must contribute to a Bugzilla entry, however, the developers admit to "loosely" grouping check-ins. That is, a commit may include a fix for more than one bug.

The developer we interviewed (UserInterface Dev1) from this project makes heavy use of task annotations for his daily work. The developer added his initials and other metadata to the TODOs. The metadata helped him filter TODOs in the Eclipse Task View. In this project, TODOs are used both for subtasks - things that he must complete for a particular issue - and future tasks - issues that may require work in the future. The developer felt that placing a TODO in the code made the task less of a priority than opening a bug, but still provided some context if the problem was to be addressed in the future. TODOs that are used for current tasks are reviewed regularly, although no formal review process exists. While many of the TODOs in the code are now irrelevant, the developer stated that he has more respect for others who indicate incomplete solutions through a TODO over those who simply commit an incomplete solution without reason.

4.4 Comment Analysis

We extracted versions of the code at weekly intervals, and grouped the extractions into monthly increments, over a three year period to determine changes in the task annotations over time. To normalize these results, we also determined the number of lines of code (including code, comments and white space) in each of these projects at each time interval.

For both the Middleware and UserInterface projects, where new features were still being added, the ratio of task annotations to lines of code increased over time. Whereas, with the Backend project, this ratio declined, albeit very slowly. An interesting spike in task usage occurred in the UserInterface project in August 2006, with the arrival of a new developer (UserInterface Dev1). Previously the number of task annotations had been relatively stable. From our examination of this small set of projects, we can see that a single developer can have a large impact on the number of tasks present in the code, and that developers can have significantly different processes involving tasks.

We also analyzed all three projects looking at how long annotations remained in the code. Figure 1 shows for the UserInterface and Middleware projects, that the annotations that were removed tended to be removed soon after being added. For example, less than 15% of all annotations that we studied in the UserInterface project had a lifespan longer than one month. Given that the Backend project was not adding new features, insufficient lifespan information was available during this time frame to see a trend.

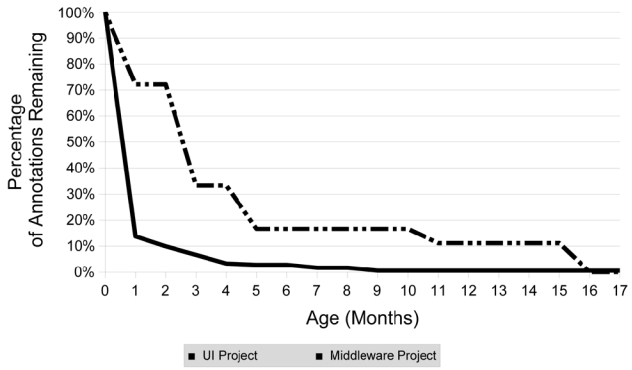

Figure 1. Life Expectancy of Annotations

5. FINDINGS

A number of salient themes emerged from our integrated analysis of the interviews, comment extraction from multiple versions of the three projects, the survey, and task annotation extraction from ten projects. We discuss these themes and link them to the data collected during the two phases of our study as well as to the

research questions. The themes are categorized within four groups: 1) task creation process; 2) TODO or to bug; 3) awareness - self and community; and 4) annotation management.

5.1 Task Creation Process

Our first two research questions ask if developers use task annotations, which ones they use, and why they use them. Seventy-nine of the 81 survey respondents indicated they wrote annotations in their code comments. Of the developers interviewed all but one used TODOs, and one developer used custom source code comments for the same purpose. The following themes relate to the task creation process.

5.1.1 TODOs Support Articulation Work

Articulation work consists of all activities that are needed to coordinate a particular task, manage subtasks, recover from errors and assemble resources [1]. From our analysis of the open ended survey questions and the interviews, we identified several kinds of articulation work that are supported by task annotations. We give examples of these here, drawing primarily from the interviews.

Subtasks: UserInterface Dev1 reported using TODOs for subtasks that were part of a larger task, "... I like to break it down into tasks, and I start with the high level first. And as I'm going through I figure out what I need to do first and then anything that hasn't been implemented I put a TODO, TODO, TODO."

Short term tasks: Middleware Dev2 used TODOs for tasks that he expected to address in "the very, very short future".

Problem indicator: Several of the interviewed developers mentioned leaving TODOs to communicate to team members that something was wrong and that they were aware of the issue.

Edge case: Backend Dev1 and UserInterface Dev1 used TODOs for edge cases. Backend Dev1 said: "I put them in there when there's some piece of code that I know is going to take me a while to write. Usually like some edge case: I don't feel like writing it right now and maybe it doesn't matter today…"

Multi-tasking: UserInterface Dev1, reported using TODOs to avoid switching tasks and interrupting his current task: "You're so focused on the task at hand, you don't want to... divert your focus. So you say okay, you drop a TODO."

Deferring low priority tasks: Developers talked about using TODOs for possible future low priority tasks. As UserInterface Dev1 said: "But, a lot of it is like, re-evaluate this, consider refactoring this, should I merge it with this, things like that. Sometimes they're questions and they're not exactly, obviously 'TODO investigate', or 'TODO should I do this'. And a lot of the ones I leave behind are those type of comments, so if I have time later on I'll come back and revisit it, but it never, you never have time [laughs]".

5.1.2 TODOs are Preferred Over Bookmarks

In addition to the TODO feature in Eclipse, the programmer can bookmark a location in their code. As mentioned before, bookmarks do not change the code, and they cannot be shared because they reside in the user's workspace. Because we were interested in bookmark usage for managing work practices, we asked about bookmark usage in both the survey and interviews. Eighty-four percent of surveyed programmers either never or rarely use bookmarks (see Table 3). UserInterface Dev1 reported using TODOs rather than bookmarks because bookmarks were lost during code refactoring, but not so with TODOs because they moved with the code.

5.1.3 Vocabulary Meaning is Idiosyncratic

From our survey and the initial search through ten projects, we noticed that developers used a variety of keywords. TODO had the highest frequency of use, with HACK, XXX, FIXME and REVISIT being significantly less common. We asked developers in the interviews what the difference in meaning is between these keywords. It would seem that the meaning they associate with these words is based on informal conventions they have individually assumed or informally agreed on within their teams. For Backend and UserInterface, TODO and FIXME meant the same thing. But, UserInterface Dev1 admitted that when he saw other people's FIXMEs he interpreted them differently, recognizing that other members in his team attached different meanings to the same terms. Backend Dev1 noted that a REVISIT was a much stronger indication than a TODO that the annotated task should be revisited.

5.1.4 Metadata is Added to TODOs

Not surprisingly, *descriptive comments* were added to most task annotations that we viewed in the task and comment extractions. In addition, developers often added their *initials* to a task. Three of the four interviewed developers indicated that they added initials to facilitate identification and navigation to their comments. Middleware Dev1 mentioned that he removed them before checking in his code. But another developer on the same project left them there to indicate an issue that required further consideration.

UserInterface Dev1 also added "an acronym that represents the work item I'm working on at the given point in time". For example, he might insert FE for File Editor[1]. He also associated a *priority* with the task annotation, e.g. LOW for low priority. Two of the interviewed developers also mentioned occasionally adding a *bug number* to their TODOs. But UserInterface mentioned that he no longer did this because hotspots in the code tended to accumulate many comments with bug numbers. We also saw evidence of developers adding bug numbers to their task annotations in the archival data, with five of the ten open source projects containing bug numbers in the current version examined.

While our survey and interview data indicate that some developers add *date* metadata to their comments, we found no evidence that developers added this metadata to the task annotations. Date metadata was most commonly found in the comment at the top of each file. It is also possible that the date information was removed before a commit because it may have been used for a short term task.

With respect to team usage, Backend Dev1 mentioned that other members in the team also put in their ID's, but "some people just put their initials and I'm not even sure whose initials they are sometimes".

5.2 TODO or To Bug

Programmers often manage team tasks through an issue or bug tracking facility. Our third research question explores how task annotations fit within the work practices surrounding the issue or

[1] The actual task and acronym have been changed to preserve the identity of the interviewee.

bug tracking systems. The following themes provide insights on this question.

5.2.1 Project Maturity Influences Number of TODOs

From our analysis of multiple versions of the three projects (as described in Section 4.4), we saw that the number of TODOs stabilized as the projects mature. This finding also emerged in the interviews. Middleware Dev1 commented on the fact that they were more likely to open bugs now than write TODOs because the project was more stable. Bugs are more visible than TODOs and appear to be more appropriate when a project has been released and is in general use.

5.2.2 Granularity of Task Affects TODO Creation

We found that a decision to create a task annotation or open a bug depended on the size and scope of the task. For larger items, the Backend team created bug reports in their issue tracking system, while smaller items were created as TODOs. But one of the developers noted "it's... very subjective on what I choose to open up [as an issue] compared to when I just put REVISITs in the code or not". Similarly, for UserInterface, smaller sub-tasks of a bug report were created as TODOs. In Middleware, all work was driven by bug reports, and any TODOs created were cleaned up before the code was submitted to the repository. TODOs were created for edge cases that were part of a larger task. Two of the interviewed developers also discussed not wishing to interrupt the flow of implementing the majority of the functionality before attending to the edge case. The TODO was put there as a reminder.

5.2.3 Cost/Benefit Analysis and TODO creation

Developers discussed the costs and benefits of creating a bug over a "quick TODO". For tasks that could be fulfilled quickly, the developers stated that there was little point in opening a bug or work item report. There were also concerns with the overhead associated with formalizing a task (emails sent to colleagues being one of them). One developer said that his TODOs were for very small tasks: "But I mean small… I don't mean a five line change. A five line change might have huge implications in the behaviour". UserInterface Dev1 mentioned that because his project was widely used in other projects, he sometimes left a TODO alone because "if there's a problem and somebody comes across it, they have no qualms about opening a bug".

5.3 Awareness, Self and Community

This next theme addresses the third research question: awareness, self and community. We found that the use of TODOs varied according to whether they were for personal, shared, or community use. This brought issues of ownership and privacy into play, as well as highlighting difficulties in communicating and interacting with the community.

5.3.1 TODOs Used for Self, Team and Community

The survey indicated that many groups shared an informal process for shared vocabulary use (49%) and that some (12%) also had a formal mechanism in place. The three teams we interviewed either used no process or a very informal one. Within the Middleware team, the interviewed developers indicated that they never altered the annotations written by others, and also assumed that other developers would not change their TODOs. There appears to be a sense of ownership around TODOs, much as there is with source code in many team projects.

5.3.2 TODOs Seldom Used for Direct Communication

Ying *et al.* [25] identified in their research that source code comments could often be attributed to communication between developers. While we found some instances of communication, sometimes what appeared to be communication was not. For example, in the UserInterface project, we identified locations where developers asked questions in their comments using the pronoun "we". However, in this case, the developer writing these comments said that when he wrote the word "we", he really meant the "royal" we. This programmer and others we interviewed preferred to send questions by email or to read the code themselves if they had a question. For tasks that required community involvement they would open a bug. Creating a bug report builds awareness of the work to be done and acknowledges where problems exist in the code base. Bug creation is also seen as a facility for building community. TODOs were not seen this way and in several cases the interviewed developers indicated that some of the conventions used in these comments would not be understood by other team members (e.g. feature acronyms). However, developers did mention looking at TODOs if one was nearby when they were fixing a bug.

5.3.3 Bounded Transparency is Important

The term bounded transparency is used in CSCW research literature to reflect that sometimes transparency is needed in information systems, but at other times the same information may need to be hidden due to privacy concerns [1]. When information should be revealed or hidden is highly dependent on a variety of factors. We detected some tensions when we queried one developer on the visibility of his task annotations in the code. On the one hand he wanted to communicate that his work was in progress to fellow team mates, but on the other hand he did not necessarily want members in the community (as it was open source) to know he had not finished implementing all of the required functionality. He noted: "And the other thing, the TODO is a little bit too prominent when you're working on a feature. Especially when you're in open source and everybody's looking at your code. So, I actually thought about this. I'm putting in all these TODOs for major features. Anybody can look at the code and just say, oh my God, he didn't do any of this stuff. And, so like politically, it's sensitive, especially if you attach your name to it". At the same time this user was concerned about the visibility of TODOs, especially given the overhead associated with opening a bug. The theme of bounded transparency also emerged in focus groups we held with software developers on the use of a software tagging tool [18].

5.4 Annotation Management

Research questions four and five address how annotations are managed and used over time. The data collection methods we used are not ideally suited to answer these questions. More effective methods would involve directly observing the programmers or indirectly observing their actions through instrumentation of their IDEs as Murphy did in [13]. However, through the survey and interviews, we gleaned some relevant insights into how programmers navigate and manage annotations. We also gained insights into how out-of-date annotations may negatively influence code comprehension and maintenance.

5.4.1 Varied Processes for Managing TODOs

The Middleware team mentioned that TODOs were dealt with before a commit, yet we observed some unresolved TODOs in their code. The other teams mentioned no formal process for revisiting and dealing with TODOs. The Backend project had many TODOs that did not get revisited and were left in the code. The interviewed developers acknowledged that some of the TODOs in their projects were very old and were probably written by previous team members. The UserInterface developer revisited his TODOs constantly, but some old ones still remained in the code. He mentioned that he sometimes cleaned them up as they were discovered, if they did not interfere with his current task. Sometimes he went back and revisited all of his TODOs to see if any were candidates for being promoted to bugs. Low priority TODOs were left as is. When revisiting TODOs, this developer deleted them if he was unable to quickly reconstruct their meaning.

5.4.2 Maintenance Issues and Out-of-date TODOs

A recurring theme throughout all the interviews was a concern with out-of-date annotations. The developers mentioned that these can clutter the code and obscure more meaningful information, or can even negatively affect code comprehension. Code cleanliness was an important consideration for some of the developers we interviewed, as they believe that "clean" code is easier to understand and maintain. When task annotations are stored directly in the code, they become an artifact that needs to be maintained. This is in contrast to comments stored in CVS or Bugzilla, where the context is stored alongside the annotation.

Several reasons for out-of-date annotations were described in the interviews. Sometimes the task was done, but the TODO was not removed. In other cases, they were left for future reference. Sometimes the tasks are simply not attended to. As Backend Dev1 reported: "we all put these comments in our code and then don't look back at them. None of them actually remind us...". A further complication arises when a programmer leaves a project and also leaves behind TODOs that others do not understand but cannot easily remove. Some developers seem either reluctant to or ambivalent about changing or removing someone else's task annotations. As UserInterface Dev1 reported: "The ones that stay forever are the ones that are unnamed, like the ones that are 'TODO this', and just the line. And you look at it, and you don't know who it came from or what really it's about or what it was accomplished for. Or what the TODO was relevant for."

5.4.3 Navigating TODOs is Not a Challenge

For every task that is created, it is assumed that the developer or someone else working on the project will return to complete the task. This requires remembering where the task is and navigating to it. With potentially hundreds of TODOs in the workspace, we anticipated that returning to these locations would be challenging. However, all four developers we interviewed indicated that this is not a problem. Developers tend to have a specific task in mind that they need to complete, and given their knowledge of the software system they are working with, they have no difficulty returning to the location that they have annotated. One developer indicated some problems finding tasks in very large classes but mostly when he was new to the project.

The survey revealed that developers use a number of tool features to navigate to annotations, including hyperlinks, the Eclipse Task View, and searching. UserInterface Dev1 mentioned that he used the markers shown in the ruler of the editor to determine if there were TODOs that needed to be revisited. He also filtered on his initials in the Task View to show the TODOs that he had added.

5.4.4 Interesting Mechanisms for Forcing Revisits

An activity that several developers mentioned was the need to pick up a task where they left off, following a break in their work. They discussed how they appropriated tool features, unrelated to task annotations, to help resume a previously initiated task. One of the Middleware developers described inserting a compile time error by putting his initials into his code, so that when he returned he was forced to revisit the error as a reminder of the task he was working on. On the Backend team, the developer mentioned making use of a testing class and a method that threw an exception when the program reached the relevant code. He referred to this as a "very active revisit".

6. DISCUSSION

Software development processes are often structured around the tasks needed to design, build, and deliver a software system. Our findings have implications for both software development processes and the tools that facilitate them. In particular, these findings directly relate to "lightweight" or "agile" methods, where the developers are often required to allocate, prioritize and manage their own units of work. Before discussing these implications, we first present the limitations of our study to provide context.

6.1 Limitations

The main limitation of this study stems from the fact that we looked solely at developers using Eclipse. We may see very different patterns of usage with different development tools and programming languages. Despite this, we note that the task annotation support in Eclipse is similar to that offered by most modern IDEs (e.g. Visual Studio). We also found evidence of annotations used for task management by searching public source code using the Krugle search engine (www.krugle.com).

Because all our data was gathered from open source projects, our findings may not be generalizable to proprietary projects. For example, visibility issues outside the development team would not be relevant. Finally, the interviewees all worked on projects with lightweight development processes that did not have formal agreement on their use of keywords in annotations. We would expect very different results should we examine more structured processes where there is agreement on task annotations. However, only twelve percent of survey respondents indicated that they do have a formal agreement within their team.

Despite these limitations, we have demonstrated that using data from interviews provides rich insights into both the results from the survey and the analysis of the archival data. The interviews also indicated that mining archival data alone is not sufficient because task annotations may be used but not committed to a repository.

6.2 Implications for Software Processes

In lightweight software development processes, such as eXtreme programming or the "Eclipse Way" (www.eclipse.org), software developers are free to manage their own tasks. Moving from *ad hoc* tasks to formal change requests is done in an informal manner. Developers who use code comments or TODOs to track ad hoc tasks often claim they can manage this transition, but in practice, many TODOs are never formally migrated to change

requests and remain hidden in the code for years. We feel this work influences software process in two ways.

Subtask creation. Since developers use task annotations to manage subtasks, software processes could support the notion of subtasks and allow them to be specified in a less formal manner. Software processes could also support subtask completion, i.e., the introduction of a step to ensure that all subtasks have been completed before resolving the parent task.

***Ad hoc* task migration.** Loosely related issues are often discovered while developers are completing tasks. In some cases a formal change request is filed. However, it is often the case that a simple TODO is added with the intention to return to it within a few working days. The developers we interviewed admitted that if an *ad hoc* task was not addressed relatively quickly, it would likely be forgotten. This indicates that a process for dealing with *ad hoc* tasks is necessary. Such a process could also result in a "cleansing" of the old TODOs from the code.

6.3 Implications for Tool Designers

Developers, who use development environments such as Eclipse, spend approximately 50% of their time in the editor [13]. When developers are not using the editor, they often use navigation views (Package Explorer, Search, Type Hierarchy, Outline, and Call Hierarchy), Debug Views (Debug Trace, Variables, and Breakpoints), the Problems View, and the Console. Very few developers regularly use the Task or Bookmarks View. Since developers spend so much of their time within the editor, it is not surprising that they often make notes for themselves in the source code. Reviewing the implications for software processes and considering how developers use IDEs, we have compiled a number of suggestions for tool designers:

In-code task annotations should support metadata. From our interviews and an examination of project annotations, we can see that developers regularly enter metadata in their comments. An automatic way for entering this information or saving it in the project memory would be useful for both the creators and users of the annotations. A mechanism for easily linking to bugs would also be useful. Mylyn achieves this by adding task annotations that were created or navigated to within the current task context. This may work well for developers that work on one task at a time but not so well for those that multi-task.

Filtering of tasks. One interviewee who used the Task View remarked how it is difficult to view and change the filters. Moreover, metadata had to be added in the same format to assist with the filtering step. The ability to effectively control which tasks are visible directly impacts the usefulness of the view.

Annotations should support linking by task. This is of particular importance to developers who use task annotations as a subtask management tool. Tools should link task annotations to their issue tracking software and allow developers to verify and remove task annotations once the parent task has been completed.

Task annotations should support *ad hoc* task clean-up wizards. To support the migration of *ad hoc* tasks to formal change requests, development tools should provide a mechanism to assist developers with expired task annotations. Some options for dealing with such annotations include: 1) removing the annotation, 2) migrating the task annotation to a formal change request, 3) re-scheduling the annotation to expire at a later date, or 4) removing the expiration. While options three and four will likely result in out-of-date task annotations, a project wide task annotation clean-up could also be scheduled at periodic times throughout the release.

7. CONCLUSIONS AND FUTURE WORK

In this paper, we described an empirical study that explored how task annotations, embedded within the source code, play a role in how software developers manage personal and team tasks.

Our findings helped us describe how task annotations can be used to support a variety of activities that are fundamental to articulation work within software development. We were also able to explore how task management is negotiated between the more formal issue tracking systems and the informal annotations that programmers write. What we saw was that the annotations have different meanings that are dependent on individual, team and community use. Finally, we revealed a number of issues with managing the annotations which may have negative implications on maintenance.

These findings led us to suggest a number of opportunities for improving the processes and the tools that are used for managing these tasks. In particular we saw evidence of a gap between what Robinson calls the "formal level of language" [17] supported by a bug or issue tracking system and the "cultural level", supported by task annotations. Robinson notes that a tool that supports one language of interaction at the expense of the other will not be successful. Eclipse and other modern tools have support for both, but we propose that the migration and navigation between these levels could be improved through tool support.

Future work involves investigating how developers from proprietary projects use annotations, as well as an examination of the more structured processes that some teams may use. We also intend to examine how tools with a closer coupling to issue tracking systems (e.g., Jazz) impact how programmers perform task management. Additionally, we intend to focus on task lifespan to better understand how developers use annotations for long term task management. Finally, we hope to incorporate some of the tool suggestions that resulted from this work within TagSEA [19], a tool we developed for managing and navigating tagged annotations in source code comments.

8. ACKNOWLEDGMENTS

We would like to thank the developers that responded to our survey and participated in interviews. Michael Muller and Li-Te Cheng of IBM provided valuable input on the design of the survey. We also appreciate the feedback received from Chris Bennett that helped improve this paper. This research was funded by the Natural Sciences and Engineering Research Council of Canada.

9. REFERENCES

[1] L. Bannon and K. Schmidt. "CSCW: Four Characters in Search of a Context," in *Proceedings of the European Conference on Computer Supported Cooperative Work*, pp. 358-372, 1989.

[2] G. Button and W. Sharrock. "Project Work: The Organization of Collaborative Design and Development in Software Engineering," *Journal of Computer Supported Cooperative Work,* 5(4): pp. 369-386, 1996.

[3] J.W. Creswell. "Research Design: Qualitative, Quantitative, and Mixed Methods Approaches," *Sage Publications*, 2002.

[4] S. de Souza, N. Anquetil, K.M. Oliveira. "A Study of the Documentation Essential to Software Maintenance," in *Proceedings of the International Conference on Design of*

Communication: Documenting and Designing for Pervasive Information, pp. 68-75, 2005.

[5] B. Fluri, M. Würsch, H. Gall. “Do code and comments co-evolve? On the relation between source code and comment changes,” in *Proceedings of the IEEE Working Conference on Reverse Engineering*, pp. 70-79, 2007.

[6] R. Grinter. “Supporting Articulation Work Using Software Configuration Management Systems,” *Journal of Computer Supported Cooperative Work*, 5(4): pp. 447-465, 1996.

[7] P. Grogono. “Comments, assertions and pragmas,” *SIGPLAN Notices*, 24(3), pp. 79-84, 1989.

[8] Z. Jiang and A. Hassan. “Examining the Evolution of Code Comments in PostgreSQL,” in *Proceedings of the International Workshop on Mining Software Repositories*, pp. 179-180, 2006.

[9] M. Kaelbling. “Programming languages should NOT have comment statements”, *SIGPLAN Notices*, 23(10), pp. 59-60, 1988.

[10] M. Kersten and G. Murphy. "Mylar: A degree-of-interest model for IDEs," in *Proceedings of Aspect Oriented Software Development*, pp. 159-168, 2005.

[11] J. Maletic, M. Collard, A. Marcus. “Source Code Files as Structured Documents,” in *Proceedings of the International Workshop on Program Comprehension*, pp. 289-292, 2002.

[12] D. Marin. “What Motivates Programmers to Comment?” Technical Report No. UCB/EECS-2005018, University of California at Berkeley, 2005.

[13] G.C. Murphy, M. Kersten, L. Findlater. "How Are Java Software Developers Using the Eclipse IDE?" *IEEE Software*, 23(4), pp. 76-83, 2006.

[14] E. Nurvitadhi, W. Leung, C. Cook. “Do Class Comments Aid Java Program Understan*ding?” Frontiers in Education,* Volume 1, pp. 5-8, 2003.

[15] M.P. Robillard and G. Murphy. "Automatically Inferring Concern Code from Program Investigation Activities," in *Proceedings of the International Conference on Automated Software Engineering*, pp. 225-234, 2003.

[16] P.-N. Robillard. “Automating Comments,” *SIGPLAN Notices*, 24(5), pp. 66-70, 1989.

[17] M. Robinson. “Double-level languages and co-operative working,” *AI & Society*, 5(1), pp. 34-60, 1991.

[18] M.-A Storey, L.-T. Cheng, J. Singer, M. Muller, D. Myers, J. Ryall. "How Programmers can Turn Comments into Waypoints for Code Navigation", in *Proceedings of the International Conference on Software Maintenance*, pp. 265-274, 2007.

[19] M.-A. Storey, L.-T. Cheng, I. Bull, and P. Rigby. “Shared Waypoints and Social Tagging to Support Collaboration in Software Development,” in *Proceedings of Computer Supported Cooperative Work,* pp. 195-198, 2006.

[20] A. Strauss. “Work and the Division of Labor”, *The Sociological Quarterly*, 26(1), pp. 1-19, 1985.

[21] L. Tan, D. Yuan, Y. Zhou. “HotComments: How to *Make Program Comments More* Useful?” in *Proceedings of Hot Topics in Operating Systems*, pp. 49-54, 2007.

[22] T. Tenny. “Program Readability: *Procedures vs. Comments,” Transactions of Software Engineering,* 14(9), pp. 1271-1279, 1988.

[23] L. Weissman. “Psychological Comp*lexity of Computer Programs,” SIGPLAN Notices, 9(6),* pp. 25-36. 1974.

[24] S. Woodfield, H. Dunsmore, V. Shen. “The effect of modularization and comments on program comprehension.” in *Proceedings of the International Conference on Software Engineering,* pp. 215-223, 1981.

[25] A. Ying, J. Wright, S. Abrams. “Source code that talks: an exploration of Eclipse task comments and their implication to repository mining”, *Workshop on Mining Software Repositories,* pp. 1-5, 2005.