Reminding and Refinding:
Examining how Software Developers use Annotations

by

Jody Ryall
B.Sc., University of Victoria, 2005

A Thesis Submitted in Partial Fulfillment
of the Requirements for the Degree of

MASTER OF SCIENCE

in the Department of Computer Science

Reminding and Refinding:
Examining how Software Developers use Annotations

by

Jody Ryall
B.Sc., University of Victoria, 2005

Supervisory Committee

Dr. Margaret-Anne D. Storey, (Department of Computer Science)
Supervisor

Dr. Hausi A. Müller, (Department of Computer Science)
Departmental Member

Dr. M. Yvonne Coady, (Department of Computer Science)
Departmental Member

Dr. Raymond G. Siemens, (Department of English)
External Examiner

Supervisory Committee

Dr. Margaret-Anne D. Storey, (Department of Computer Science)
Supervisor

Dr. Hausi A. Müller, (Department of Computer Science)
Departmental Member

Dr. M. Yvonne Coady, (Department of Computer Science)
Departmental Member

Dr. Raymond G. Siemens, (Department of English)
External Examiner

Abstract

Software development requires understanding and navigating complex software spaces. Developers frequently utilize annotations in source code to help them externalize information they need to remember, such as tasks and implementation details. Although some tool support exists in modern integrated development environments for authoring and navigating these annotations, we have observed that they often fail to remind developers about tasks that need to be performed and are sometimes difficult to find. We present the results from four empirical studies designed to better understand how developers create and manage their information using annotations. We also explore the use of hierarchical tagging capabilities to enhance these annotations. Based on the findings from these studies, we provide suggestions on how annotation tools may be improved.

Table of Contents

List of Tables

List of Figures

Acknowledgements

Not long ago this thesis seemed far away and unlikely to be started, let alone finished. It is only with the support of many friends and colleagues that I made it this far.

To my supervisory committee: Peggy Storey for her creativity, guidance and friendship; Yvonne Coady for her unlimited enthusiasm and encouragement; and, Hausi Müller for his unfailing support.

I would like to thank the members of the TagSEA team for taking part in this research. Special thanks go to Del Myers for his work developing and maintaining the TagSEA tool, and for his assistance analyzing the log data it produces. To Janice Singer and Peggy Storey for their time and expertise in coding the qualitative data and making the process seem painless. Thanks also to Li-Te Cheng for recruiting participants in the industry case study and for helping to gather their feedback. To Ian Bull for interviewing the Eclipse developers while on-site in Toronto. And, finally, thanks to Michael Muller for his input on social tagging behaviour. Our combined efforts helped to make this research stronger than it would have been individually.

I can also not forget, or truly thank, all the participants who gave their time to participate in this research. Your feedback and insights are most appreciated.

To members of the Computer Human Interaction & Software Engineering Lab (CHISEL): thanks for making each day fun and enjoyable. There was always laughter to share and when it came time to work, insightful comments and wise advice. Special thanks to Chris Bennett and Trish d'Entremont for their feedback on this thesis.

Finally, I would especially like to thank my friends and family, for their immeasurable support and for providing balance to the sometimes all-consuming nature of graduate school.

Chapter 1:
Introduction

"I long to accomplish a great and noble task,
but it is my chief duty to accomplish small tasks
as if they were great and noble."
- Helen Keller, 'Optimism' (1903)

SOFTWARE developers must understand and navigate complex software spaces to perform their daily tasks. Details for a single task are often scattered across multiple packages, classes, and methods. When a software developer is working on a large project with many tasks to perform, remembering all the relevant details using human memory alone is nearly impossible [48].

The problem of dealing with information overload arises, in part, from multitasking. With multiple ongoing tasks, developers have to choose which task they will perform next, and store the rest for later. Developers must be able to collect, manage, categorize and store these tasks and their associated information, and then recall them as needed. In particular, two parts of this process are challenging: categorizing and recalling information [47][52]. To cope with these difficulties, people have developed various strategies to externalize their memory, such as creating task lists or email reminders.

Annotations have been used as an aid to externalize memory for centuries. For example, annotations can be used to record tasks that still need to be performed, mark locations on a map to visit or revisit, and record thoughts or implications about what has just been read. Software developers have also been known to create annotations to record information for future recovery.

Software developers have created processes and tools for using annotations in their everyday work. The simplest practices available today include creating bookmarks and leaving memorable keywords in the code, such as "todo", "fix me" or the author's initials. Modern IDEs provide tools for

creating and managing annotations; however, these tools do not always adequately support remembering and refinding information. For example, bookmarks are hard to locate and keep synchronized. Similarly, task annotations – specific keywords recognized and highlighted by the compiler – are often rigidly grouped into a handful of predefined categories.

While we have intuitions on how developers use annotations to cope with multitasking, little research has been done in the academic community. We propose to examine how developers create and manage their information using software annotations based on the following research questions:

Q1: What is the content and structure of the annotations being created?

Q2: How are the developers using these annotations?

Q3: Where do they store this information?

Q4: Who are these annotations intended for?

Q5: Why are these annotations beneficial?

By answering these questions we hope to lay the groundwork for further examination of annotation use in software.

Our earliest findings led us to hypothesize that there are two key weaknesses to current tool support: (1) lack of navigational support for linking related code (i.e. cross-cutting concerns), and (2) lack of metadata and structure for the management of these annotations. Seeing these shortcomings, our research group has developed a tool called TagSEA (Tags for Software Engineering Activities) that allows users to employ user-defined keywords [27] and to structure them hierarchically. We aim to explore the potential benefits of these techniques for improving current tool support.

Our research is guided by a mixed-methods design [13]: collecting and combining quantitative and qualitative data from varying sources. In this thesis we present four empirical studies with professional software developers examining the situation in two directions. First, we consider how software developers use current annotation techniques. Second, we examine how software developers use TagSEA, focusing on user-defined vocabulary and hierarchy use.

Based on our findings, we develop a set of implications for software process and tool designers. In the process we uncover additional questions and avenues for future exploration. We hope that by exploring this topic we may improve annotation tools, so that they may better support developers as they externalize their memory.

1.1 Thesis Outline

In Chapter 2, we examine issues that surround managing tasks and information in a multitasking environment. Externalizing memory using annotations is a way of coping with the side effects of multitasking. In Chapter 3, we look at how annotations in three contexts can be used for externalizing memory. By doing so, we can determine what considerations to take into account when designing tools to support software developers. The current tool support available is described in Chapter 4, along with what researchers have learned about the annotation habits of software developers.

Our objective in this research is to better understand what role annotations play in the work practices of software developers. To accomplish this goal, we have designed four studies following a mixed-methods methodology, which are outlined in Chapter 5. There are two phases to this research. First, in Chapter 6, we examine how software developers use current annotation mechanisms in their work by conducting a questionnaire, studying open source code, and conducting developer interviews. Second, in Chapter 7, we explore how developers can use tagging to manage their annotations by studying developers using the TagSEA tool. In Chapter 8, we synthesize and discuss the findings from the studies in light of our research questions. Implications of these findings for annotation tool designers are presented in Chapter 9, along with the limitations and contributions of this work.

As a collaborative approach was taken in this research, my individual contributions are outlined in Appendix A.

Chapter 2:

Multitasking & Memory

"What memory has in common with art is the knack for selection, the taste for detail….
More than anything, memory resembles a library in alphabetical disorder,
and with no collected works by anyone."
- Joseph Brodsky, 'In a Room and a Half', *Less Than One: Selected Essays* (1986)

WHEN we have only a single task in mind, we can easily immerse ourselves in the details at hand and make progress on that task. However, what happens when there is more than one task? Or more importantly, more than one task that demands our attention? We must make decisions about how to manage our time – which task to do first and how much time to spend on it. But, it is not quite that simple. We must also expend considerable mental effort to switch tasks, remember which tasks need to still be completed, and regain any context we lost when we return to an incomplete task. All of this puts a strain on human attention and memory, which while capable in many instances, benefits from external help.

As multitasking is an essential part of everyday work, people have developed various coping strategies to minimize the mental effort involved in managing multiple tasks. These strategies include simple things, like making notes and lists, using a calendar to keep track of deadlines, or using physical items to serve as reminders. These strategies act to externalize our memory, allowing us greater ability to manage and remember tasks and their associated details.

In this thesis, we examine how annotations can be used by software developers for managing development tasks and other related information. Before looking at annotations, we must examine the issues surrounding managing tasks in a multitasking environment. By doing so, we can determine

what considerations to take into account when designing tools to support software developers.

2.1 Multitasking Activities

Activities surrounding multitasking can be divided into various facets. Based on the research literature, I have grouped these activities into four categories, each of which will be discussed below:

1. task switching
2. managing information and tasks
3. task categorization
4. remembering and locating tasks

2.1.1 Switching between Tasks

With multiple ongoing tasks, some switching is inevitable. When switching occurs, we must store the information about the postponed task, as well as remember to return to it [15]. Unfortunately, human memory has limits and can fail. Memory failures can take two forms. We are probably most familiar with *retrospective memory failures* – inability to recall previous information; however, there is another category as well, *prospective memory failures*. Prospective memory involves forming an intention to perform a task and then acting on that intention in the future [20][51]. Inability to remember to perform that task on time is considered a prospective memory failure. For instance, a person could decide to phone their friend on their birthday. Forgetting which date the birthday falls on would be a retrospective memory failure. Forgetting to call on that day would be considered a prospective memory failure. Interestingly, prospective failures occur in everyday life as often, or more often than, retrospective memory failures [46].

Based on the varied reasons that cause interruptions [3][11], and that people tend to interrupt themselves nearly as often as they are interrupted

[16][31], avoiding disruptions and memory failures entirely is not an option. Fortunately, prospective memory failures can be prevented when adequate *reminders* are available. Similarly, retrospective memory failures can be averted if we have support for managing and *refinding* information. People have developed strategies to manage multiple tasks and avoid memory failures, such as creating email reminders, task lists, or calendar entries [6]. These strategies provide varying levels of support for reminding and refinding.

2.1.2 Managing Information and Tasks

Software developers and other knowledge workers must not only manage their tasks, but also the information associated with these tasks. In a broad sense, Barreau and Nardi [5] found that knowledge workers have three types of information: ephemeral, working and archival.

Ephemeral items have a short shelf-life. Bernstein *et al*. [7] referred to this ephemeral information as *information scraps* – "short, self-contained personal notes that fall outside of traditional filing schemes". Examples include: "to do" lists, memos and note pads. Users in Barreau and Nardi's study preferred to keep this information visible rather than filing it. This allows it to be used for quick reference. As a side effect, however, these notes can pile up, creating clutter.

Working items involve information related to current tasks. The shelf life is generally measured in weeks or months. Depending on personal preference, these items are either filed or piled [52][81]. Regardless of how it is stored, it is usually close at hand. Repeated use of this information over its shelf life, usually measured in weeks or months, makes it easy to remember where it is located.

Archival items generally consist of completed work. More often than not, these items are filed and are usually stored for months or years.

An interesting example to demonstrate how people manage information and tasks in an electronic environment involved examining how people managed their email. As with paper documents [52], there are different strategies for managing the email inbox. Whittaker and Sidner [82] identified three categories of email users: no filers, frequent filers and spring cleaners. No filers used the inbox for all of their email, while frequent filers made daily passes of the inbox to keep the number of items down. Spring cleaners periodically cleaned-up their inboxes, usually once every one to three months.

For all types of email users, the authors found that specific types of messages were not discharged immediately. These messages tended to be kept to act as reminders (e.g. to-dos and to-reads) or to provide important information about ongoing tasks (e.g. ongoing correspondence). However, they noted that as the inbox became large, its value as a to-do list faded and there was a shift towards using it for more active tasks. They also found that people did not want to file emails that were active – aiming to keep the information available and readily accessible in the inbox for later refinding. The reasons for not filing included: not wanting to file messages that would not be useful later (especially since usefulness of data changes over time), not wanting to forget where it was stored, and that it was difficult to categorize the information for filing.

2.1.3 Categorizing Activities

Categorizing is traditionally thought of as an activity for archiving information; however, it also can be used for ephemeral or working information to provide more structure, particularly for large amounts of information (e.g. complex task lists or bug tracking systems in software).

Deciding how to categorize items is cognitively difficult [47][52]. Part of this difficulty arises from the actual act of classification. People do not classify information as a librarian would, but rather in the context that the information appears [47]. Items do not always neatly fit into one category. The other part of

the challenge comes from creating a classification that people can use to retrieve the information in the future.

People deal with the challenge in different ways. Ravasio *et al.* [65] reported on an ethnographic study that looked at how users managed and located files on computers. In particular, they found that participants used different strategies for storing documents. For self-created files there was less of a challenge, with 12 of the 16 participants saving the files directly to a location in the filesystem. However, for non-self-created documents, behaviours differed. Five participants saved the documents to the desktop, six saved it to a location that the users thought was approximately the right higher level folder, two saved it to a pre-defined temporary folder, and three saved it directly to the folder they thought was appropriate.

Malone [52] mentioned three ways to assist with classifying items: (1) allowing multiple classifications, so the user does not have to decide on a single rigid classification system, (2) allowing users to defer classifications (i.e. creating piles of information), and (3) allowing the computer to automatically classify items. While the first two options show promise, on this last point, Whittaker and Sidner [82] provided a warning. They observed that users did not want automatic filing of email because they wanted to be aware of what was arriving, and not ignorant of the existence of potentially important information.

2.1.4 Remembering Tasks

A classification system is designed to provide structure for refinding the information that we are storing. Whether we store information by date, keyword or some other dimension, we must still remember roughly where we put it. Retrieving this information requires two psychological processes: *recall* directed search, followed by *recognition* based scanning, also known as browsing [47]. While information retrieval usually involves both processes, one tends to dominate. When the information sought is known or the problem of finding it is

well-structured, the overall strategy we focus on is directed search (i.e. selecting, specification, and ending) [10][79]. Otherwise the overall strategy employed will be browsing (i.e. scanning, learning, recognition, etc.) [79].

In the case of remembering tasks and information that we have stored, the issue tends to be one of refinding information. Refinding is actually a different cognitive process than finding information [10]. With refinding, the user has already seen the information at least once. As such, the user may be able to recall associations, cues or aspects of the information that will aid retrieval. Regardless of what is remembered, refinding still involves searching and browsing [77].

Elsweiler [22] suggested that tools should provide cues along various dimensions to aid in refinding. These dimensions could be visual, spatial, temporal, contextual, etc. This requires providing facilities for storing and manipulating metadata associated with the tasks and information being stored. Since people only remember partial details about what they have stored [47], flexibility in this regard could be beneficial.

2.2 Summary

In this chapter we have looked at some of the issues surrounding multitasking and how there are various techniques for coping with the side effects of switching tasks. The research literature suggests that to minimize the strain on human memory, we need to ensure that classification is as simple as possible and that the techniques available allow for ways to aid reminding and refinding of information. With an idea of the challenges that knowledge workers face, we turn our attention now to how annotations might be used to externalize memory and support reminding and refinding.

Chapter 3:
From Scribbles to Collaborative Annotations

"This making of notes, however, is by no means the making of mere memoranda…
in fact, if you wish to forget anything on the spot, make a note that this thing is to be remembered."
- Edgar Allan Poe, 'Marginalia', *Southern Literary Messenger* (1849)

ANNOTATIONS are a means of supporting human memory, whether to remember tasks or to record details about an item. Once an item is written down it can be largely forgotten, leaving space for other cognitive processes. The annotation, if well formed, can help remind the author of the item and refind any pertinent information.

Whether as simple as a bookmark in a novel to keep one's place, as personal as colour coded notes taken in a textbook, or as brief as a scribbled note on a notepad, annotations take many forms. However, regardless of the forms they may take, annotations are composed of two items: an anchor and content [9]. The anchor can be explicit, as in a circled piece of text, or implicit, such as a scribble in the margin of a book [9][55]. Content is optional for annotations; if present, content either has an explicit or telegraphic (i.e. personal) meaning [55].

To understand more on how people can use annotations in electronic environments, this chapter surveys: paper and digital annotations, web based annotations, and social tagging.

3.1 Paper versus Digital Annotations

Marshall [55] looked at how annotations are changing as we move from a paper-based society to a digital society. In one study the author looked at annotations in used textbooks purchased from a university bookstore. The textbooks were taken from across disciplines. Based on the annotations found in these books, the author developed a list of reasons for making them. Annotations may serve as:

1. procedural signals (acting as a *reminder* of something to read or re-read),
2. placemarks (recording important locations for *refinding* later),
3. a visible trace of a reader's attention,
4. an *in situ* way of working on problems,
5. a record of interpretive activity (interpreting the material), and
6. incidental markings.

The first three categories were characterized by the use of highlighting, underlining or symbols – usually simply an anchor without content. The fourth and fifth categories tended to involve words, phrases or equations. The final category involved other markings, like doodles and messages unrelated to the text. Depending on the context, these annotations can act as reminders (e.g. showing what has been read) or as a cue to draw attention for easy refinding later (e.g. an asterisk in the margin).

Marshall also noted the personal nature of many of these annotations. Readers would mark up the texts with various symbols, different pen colours, and shorthand notes. This freedom to choose an annotation scheme that is memorable and informative to the author usually renders the annotations meaningless to others. Even to the creators of these annotations, their notes may not make sense over time, as they may forget their original intent. In a related study, participants of an academic reading group were asked to use a digital tool for making notes [57]. Participants were interviewed about the events of two weekly gatherings. Even after only a week, the specific intent of the annotations was sometimes lost.

In a related paper, Marshall and Brush [56] conducted a study with 11 graduate students taking a seminar in Human-Computer Interaction. The students were asked to read three papers per week and to participate in an online discussion using a web annotation tool. Students had the option of annotating directly on the web or to annotate on paper and then post comments using the tool. Annotations using the tool could be made private. All of the students chose to read and annotate on paper and then transcribe their

comments on to the web. At the end of the study period the researchers collected the private paper-based annotations and compared them to the public web-based ones. The researchers found significant differences between what people wrote on paper and what they chose to share. Students frequently expanded upon the comments they had made on paper. They also wrote comments when previously there had only been an anchor. While one would expect that people might modify their annotation behaviour when they know someone else is reading it, this shows that the normal annotations that people make for themselves tend to be less descriptive than those intended for sharing, and may be unintelligible to others. The researchers suggested that at best 7.8% of the personal annotations translated into collaborative use.

3.2 Bookmarks on the Internet

One of the most common ways of annotating in electronic environments is by bookmarking. In the case of the Internet, the webpage is the anchor and the content is a string chosen by the user to identify the page. By default, the title of the webpage is usually used. The purpose of these types of bookmarks is to allow users to easily refind websites that they think might be important for a current or future task.

Abrams *et al.* [1] studied the bookmarking of webpages. They conducted a survey with 322 respondents, inquiring about the number of bookmarks that participants had accumulated. The authors also collected bookmark files from 56 individuals in 1996. While bookmark use has presumably changed since then, they present some interesting findings. The authors claim that bookmarks can (1) reduce cognitive load of managing URLs, (2) facilitate the return to groups of related pages, and (3) enable users to create a personal information space.

Perhaps more interesting, Abrams *et al.* observed four major metaphors for how participants perceived their bookmark use: identification, collection, movement, and episodes. People that relate to identification would view their

bookmarking as labeling or marking information. The collection metaphor suggests that people use the bookmarks to retrieve information from a vast information space while remaining stationary. The movement metaphor involves navigating or traveling to the information, with the bookmarks acting as landmarks. Finally, the episode metaphor refers to a chronological list of sessions – essentially a history of the browsing activity. All of these metaphors are variations on refinding, but utilize different dimensions for organizing and retrieving the information: classification, task-based, spatial, and temporal. They are all simply different ways of conceptualizing the information.

Abrams *et al.* also examined how participants organized their bookmarks. They found that 30% of respondents were active filers, 44% engaged in sporadic filing, and 26% never organized their bookmarks, thereby keeping them in chronological order. The sporadic filers were found to engage in 'spring cleaning' behaviour occasionally. Those that accumulated lots of bookmarks often created a hierarchy to help with organization. This hierarchy was usually slow to develop and became rigid over time. Finally, in a temporal sense, they found that the median time since a bookmark was last visited was 100 days. This suggests, as the authors noted, that bookmarks are primarily an archival form of information. Furthermore, this suggests that they are ineffective as a tool for actively reminding users or as a tool for refinding pertinent information.

Jones *et al.* [41] observed users engaged in a web-intensive task to understand how people kept track of website locations. The task was self-chosen as something the users wanted to do when they had 30 minutes of free time. While all 11 participants claimed to use bookmarks, only one actually did during the study. Some of the participants claimed their bookmarks needed cleaning up. As a result, the authors concluded that bookmarks have a low reminding value. They suggest that they could be improved by including metadata that would allow users to recall how the webpage was found, why it is relevant, and what actions remain to be taken, if any.

3.3 Social Tagging

Social bookmarking originally was based on the sharing of web bookmarks. However, these systems met with limited success [58]. Recently there has been a resurgence in sharing annotations in the form of tags. A *tag* is simply a user-defined keyword that is meaningful to the individual [29]. Systems have emerged to allow people to tag a variety of items, including: photos, videos, books, product descriptions and people's profiles. When collected together these tags form a large repository of information. From my review of the literature, a few themes emerge that are relevant to this thesis: (1) motivations for tagging, (2) benefits of free-form organization, and (3) the choice of vocabulary.

3.3.1 Motivations for Tagging

As with bookmarks on the Internet, the motivation for tagging is largely personal. People create a web bookmark if they sense it has potential for future use [1]. While web bookmarks are restricted to website URLs, tagging can be applied to almost anything. It is also relatively easy to add tags in most systems, reducing the effort needed to participate [28]. Motivations can range from being solely for the individual to being altruistic for the group [36].

Ames and Naaman [2] studied tagging behaviour for photographs. They found that most people participated for one or two primary reasons, and that few participants tagged solely to organize their own photos. The authors developed a taxonomy based on two dimensions: sociality (self or social), and function (organization or communication). For example, a user may choose to tag photos for themself for the purposes of refinding the photos again later (i.e. self and organization). Or a user may choose to tag photos as a way of providing social signals to others (i.e. social and communication).

3.3.2 Organization

Regardless of the intent, the collective gathering of these tags creates an informal classification system. While professionally created classification systems, such as those found in libraries, are characterized by high quality metadata and a predefined structure and syntax, they do not scale to large information spaces easily [29]. With collaborative tagging systems, people are free to use their own way of thinking and see what develops. This classification system is user-driven following a bottom-up approach [67].

3.3.3 Vocabulary

Sen *et al.* [68] noted that allowing users to invent personally meaningful tags can make tasks, such as organization and refinding, easier for the individual. However, these individual inventions may make it more challenging for other users to locate items. This situation was described by Furnas *et al.* [27] as the *vocabulary problem*. People tend to have different words for the same things. For instance, soda or pop; sofa, couch or chesterfield; and bug, issue or work item. While the creation and tagging of items has become easier in this type of user-defined system, the management and location of terms is less reliable. The multitude of vocabulary weakens the classification system if people must figure out the term to search by [30]. Table 1 shows issues commonly associated with the vocabulary problem.

Table 1: Vocabulary issues related to tagging

Problem	Description	Example
Plurals	The difference between singular and plural forms of a word.	Cat and cats
Spelling	Numerous words have spelling variations.	Theatre and theater
Homonyms	A word has multiple unrelated meanings.	Bow (front of a ship, tied ribbon, weapon, etc.)
Polysemes	A word that has more than one related meaning.	Wood (piece of a tree, and a forest)
Synonyms	Multiple words that have the same meaning.	Student and pupil
Hyponyms	A word that may be replaced by another, but not vice-versa without changing the meaning.	Tulip and flower
Meronyms	A word that represents part of another concept.	Knee and leg

Social tagging researchers have noticed that, despite these differences in vocabulary, usage tends to converge upon common terminology [30][68]. This can be expedited by the use of recommender systems, which encourage users to choose existing terms [68]. Another option is to use thesaurus systems to match similar terms [80]. Despite the work in this area, this problem is often described in the literature, but no solid solutions have been found.

3.4 Summary

All of the annotations we have seen are easy to create and can be useful for externalizing memory to support reminding and refinding. In the following chapter we consider tool support for creating annotations in software that support reminding and refinding, such as bookmarking and tagging in source code.

Chapter 4:

Annotations in Software Development

"The medium is the message."

- Marshall McLuhan, *Understanding Media* (1964)

INTERNAL documentation [66] – comments stored directly in the source code – is often seen as a vital part of the software development process [24]. This type of annotation serves as a way of communicating details about the code to anyone examining it in the future, including the author. Commenting code is a fairly common practice although it is less prominent in some software practices, such as agile software development.

We summarize the state of the art of tool support for creating and managing user defined annotations for reminding and refinding. Following this, we then consider what the research community has learned about how these annotations fit within the work practices of software developers.

4.1 Tools

Modern programming languages and Integrated Development Environments (IDEs), such as Eclipse [19], typically offer various mechanisms to support reminding and/ or refinding. In this section we describe this set of tools and synthesize their underlying design decisions to highlight gaps in reminding and refinding tool support.

4.1.1 Unstructured Source Code Comments

The simplest way for developers to annotate locations of interest involves inserting comments into the source code. These locations can be relocated easily by browsing or searching. These comments can be used to assert information

about the code, or to indicate locations for future work. A familiar example is the prevalence of informal expressions and keywords – such as “hack”, “fix me”, or the developer’s initials – to highlight suspect code. Such comments may be scattered throughout the program if the features being documented cross-cut the established software structure. A drawback of refinding distributed in-line comments is that the developer needs to remember either the keywords or the location in the code.

4.1.2 Programming Language Support for Navigation

To help provide support for navigation via in-line parsable comments, various programming languages have special syntax. For example, Javadoc documentation has the “@see” and “@link” tags, which are accompanied by notation referring to parts of code (e.g. packages, classes, and methods) or URLs [74]. Modern Java IDEs automatically turn these tags into clickable hyperlinks.

Java annotations can provide similar affordances, but can also affect how programs are compiled and run [73]. A developer can define a custom annotation (see Listing 1) and embed it almost anywhere in the code, not only in a comment. These annotations can be used for various purposes, such as to generate reports or to identify the files to be checked-in to a source code management system.

```
@FeatureRequest(
        id    = 247835169,
        message = "Enable teleporter",
        author = "Ethan",
        date   = "4/8/2008"
)
public static void travelToDestination(Date destination) { ... }
```

Listing 1: Example of a Java annotation

4.1.3 Bookmarks

Bookmarks are a feature that allows a developer to mark a location of interest in the code without storing the annotation directly in the code itself. Bookmarks in software are analogous to web bookmarks. For Eclipse Bookmarks (Figure 1), the annotations are stored in the developer's workspace. To refind these bookmarks, there is a Bookmark View. In addition, Eclipse provides visual cues of the bookmarked location in the margin of the editor.

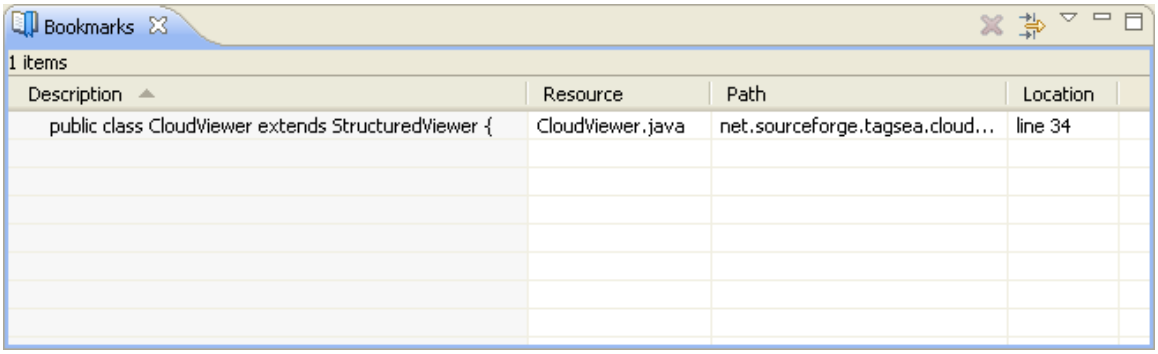


Figure 1: Eclipse Bookmarks View

The research done by Murphy *et al.* [61] suggested that the Bookmarks View is rarely used in Eclipse. The authors conducted a study that logged the interaction of software developers as they used Eclipse in their everyday work. Using the Mylar Monitor, "a standalone framework that collects and reports on trace information about a user's activity in Eclipse", they reported on the activities of developers who had significant use of the Eclipse Java Development Toolkit (JDT). Through an analysis of Murphy *et al.*'s data logs, we saw that bookmark selection events were as low as .02% of total view selection events, and that only four of the 42 programmers used bookmarks at all. We have also received anecdotal feedback from users of other IDEs, such as Visual Studio, that bookmarking features are rarely used.

Our research [72] suggests this low usage is due to four reasons. First, although UI support makes it easy to bookmark an element in the code, bookmarks lack sufficient semantic information to facilitate later recovery. Second, bookmarks suffer from a lack of visibility because they require a specialized viewer to see them and therefore are easily forgotten, difficult to manage, and thus quickly become outdated. Third, bookmarks are typically stored in the user's workspace and cannot easily be shared across teams of developers. Finally, as the code evolves, bookmarks can become unsynchronized and lose their relevance, as they are not anchored to the code. Similar to Abram *et al.*'s result for web bookmarks [1], we find that bookmarks provide weak support for reminding and refinding information.

4.1.4 Task Annotations

Many IDEs allow developers to create annotations for recording tasks by inserting specific keywords into code comments. Certain keywords are predefined by the parser to appear as task annotations. For example, in Eclipse the predefined keywords are "TODO", "FIXME" and "XXX". These keywords may appear in lowercase, although they are traditionally all in capitals. FIXME is assigned a high priority by default. Additional keywords may be defined by the user in a preference dialog; however, this feature appears to be rarely used and many developers are not aware that the keywords may be customized.

Similar to bookmarks, navigation is provided via a specialized view in the IDE that allows for filtered viewing. Clicking on an annotation brings a programmer to the location of that annotation. Since task annotations are inserted directly within code comments, they are implicitly shareable and can be manually updated as the code evolves. Tool support for task annotations was likely added in response to the prevalence of developers adding easily searchable keywords in the code to facilitate refinding (see Section 4.1.1) [84].

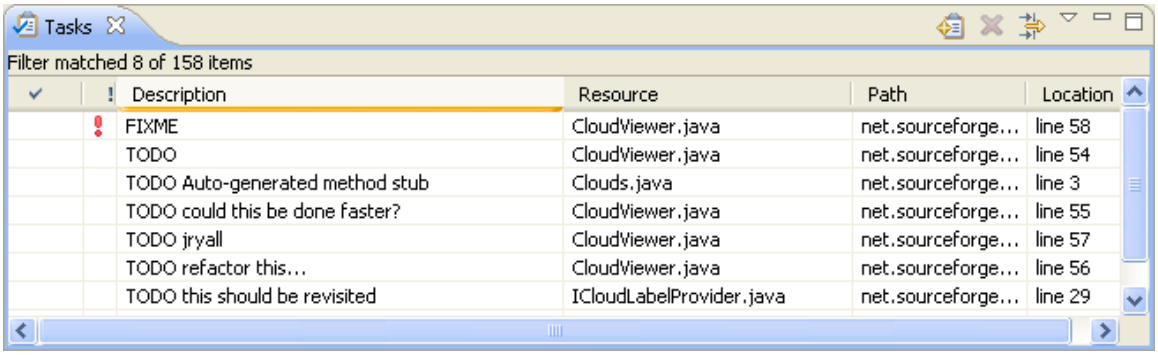


Figure 2: Eclipse Tasks View

To gain insight on how frequently this view is used in Eclipse (Figure 2), we did a further inspection of Murphy *et al.*'s data [61], and saw that task selection events were as low as .2% of the total user view selections, with only 13 of the 42 programmers in that study using them at all during the study period. This low use could be due to limitations of the view. The Tasks View provides limited support for sorting or filtering, with the latter only available through a preference dialog. The limited task vocabulary means that the annotations are grouped into a few broad categories. This lack of sufficient metadata makes it challenging to find them and to determine which ones are important.

Since task annotations also appear directly in the code, the Tasks View is only one way to access and manipulate them. Ying *et al.* [84][85] conducted a study of task annotations from one project, creating a classification of uses. They found that such comments were used by programmers to "talk" to each other as well as to support navigation. Thus, it appears that source annotations do support, at least at a rudimentary level, both reminding and refinding.

4.1.5 TagSEA

TagSEA (Tags for Software Engineering Activities) [69][70][71] is a framework for tagging locations of interest within the Eclipse IDE. Designed by members of our research group as a set of plug-ins, TagSEA integrates ideas from social tagging with the aim of making it easier to find information. There were two

main ideas: (1) providing more structure for organizing annotations by allowing developers to define their own vocabulary, and (2) allowing developers to link cross-cutting concerns in the software.

Tagging is not a new concept to software engineering. Tags have been used for decades for annotating check-in and branching events in software version control systems, as well as for documenting bugs in issue tracking systems. Brothers' ICICLE was an early exploration of a limited, controlled-vocabulary of tag-like structures during code inspection [8]. More recently, CodeSnippets [24] is an Eclipse plug-in that aims to use Web 2.0 tagging to help software developers retrieve code snippets from a repository. However, the research community has not yet adequately explored tagging as a mechanism for categorizing and retrieving lightweight annotations in source code.

TagSEA has gone through a series of iterations and changes during its lifespan. In this section we describe the current version of TagSEA (0.6.4). As this thesis focuses on the empirical data collection and analysis surrounding the use of annotations in software and not an evaluation of TagSEA, technical details related to the implementation of the tool will not be discussed in this thesis.

TagSEA is based on a *waypoint* metaphor. Waypointing comes from the navigation of routes in geographical spaces [18][64]. Waypoints are created by marking a location of interest. They have associated metadata, such as creation time and author. Waypoints can be shared across users and applications and may be gathered into routes.

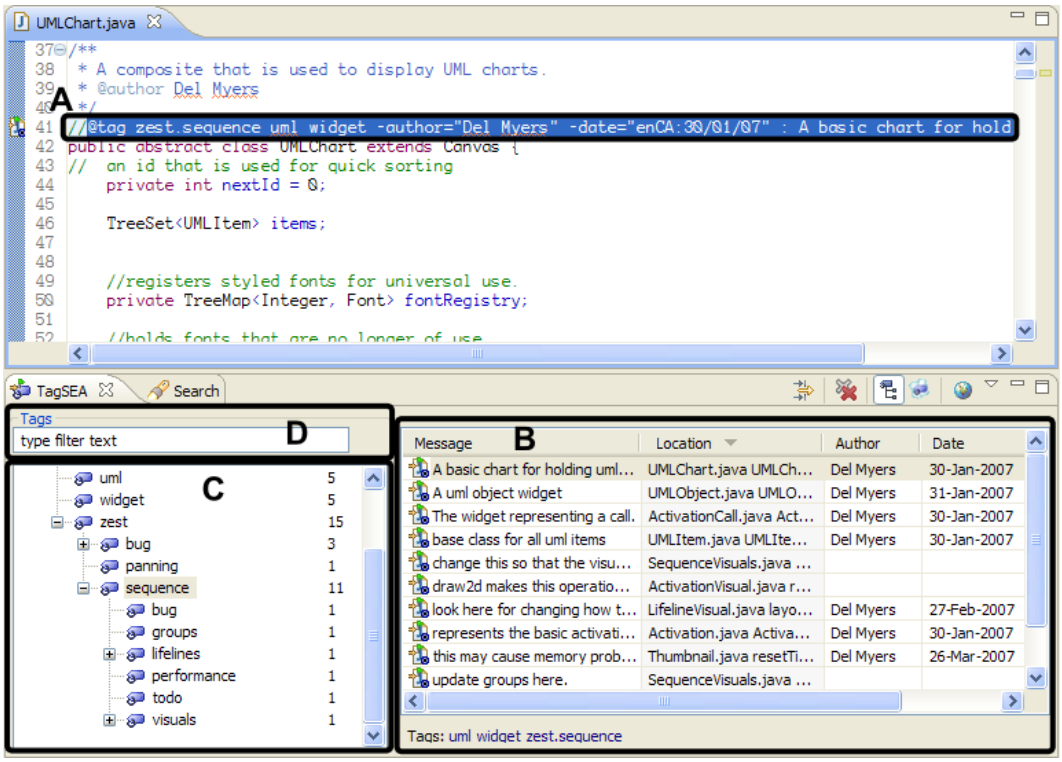


Figure 3: TagSEA 0.6.4 for Eclipse. (A) a waypoint containing tags; (B) a tabular list of waypoints; (C) a tree of hierarchical tags; (D) an input box for filtering the tags that are visible in C.

In TagSEA, a *waypoint* is simply a tagged location in the IDE. Individual locations can be tagged with single or multiple keywords (Figure 3A). In addition to tags, other metadata can be automatically associated with each waypoint; in this version the metadata supported includes a message, the creation date and the author's name. The waypoint, acts as an anchor, while the tags and metadata associated with the waypoint serve as the content.

Developers create waypoints by associating tags with parts of the source code using a Javadoc-style keyword. Specifically, a waypoint is created by the developer typing "@tag" in a comment block, followed by the tag keywords. Descriptive free text messages can also be added after a ":".

Individual tags are delimited by spaces. A developer can also create hierarchical tags using dot-separated notation as follows: "@tag

bug.performance". This indicates that there is a waypoint with the tag "bug" whose subtype is "performance". The Javadoc-style syntax allows for easier adoption of TagSEA by Java developers, who are already familiar with similar conventions, such as "@author" and "@version". The resulting waypoints are automatically associated with the closest Java element (e.g. a method). Once waypoints are created, they can be used to navigate and understand the code, or to share information with others by uploading "waypointed" code into a source code management system.

TagSEA also has support for refactoring tags, allowing them to be easily renamed, reorganized or deleted. Reducing the number of unique tags created can be addressed by using a consistent set of tags over time [36]. TagSEA provides an automatic tag completion (content assist) feature to suggest existing tags based on a partially typed tag.

Waypoints indicated by the "@tag" notation are highlighted in the editor's text, left-margin, and scrollbar so that they stand out as clear landmarks as developers browse through the code (see Figure 3A). Using the Waypoints Viewer (see Figure 3B), developers can search for and navigate to a particular waypoint. Managing a growing sea of tags is a concern in large software systems. To address this concern, TagSEA provides initial support for dynamic filtering of waypoints (see Figure 3D). Every keystroke in the text box immediately filters the list of tags and displays those that partially match the entered query, allowing a user to condense and explore tag spaces. Users can also sort waypoints via metadata.

As an example, a developer could view all locations tagged with a specific bug id or task, or all locations tagged by a particular developer. Selecting one or more tags in the Tag Tree (see Figure 3C) reveals the associated waypointed Java elements in the middle pane. Then, clicking on the entries in the Waypoints pane opens the associated file editor, positions the editor at the appropriate location, and highlights the waypointed Java element. Thus, developers can quickly navigate to places of interest.

At first glance, TagSEA may look similar to the task annotation tool support currently available in most IDEs. However, it is different in several key ways that we feel are important. First, tags are created on the fly by the developer in a bottom-up fashion; they are not predefined nor do they need to be customized in a special dialog before they are embedded in the code. This is consistent with earlier research, in which Furnas *et al.* showed that users prefer to use their own vocabularies for common objects and concepts [27]. Second, the developer can create hierarchical tags, thus producing a lightweight navigational taxonomy. This may also address potential occlusion problems caused by task annotations. The tagging hierarchy can also be refactored, which allows the developer to easily change tagged code, e.g., changing a tag from "bug14.fix-this" to "bug14.fixed". Finally, the waypoints in which the tags appear have additional metadata that further facilitates refinding.

4.2 The Use of Comments in Source Code

While some strategies have been employed to make source code itself clear and understandable, such as literate programming [44], software developers often rely on comments to communicate details. Similar to the annotations we have already seen, comments in source code have an anchor and a message. Van De Vanter [78] refers to these components as content and placement. In general, the link between comments and the source code that it relates to is not explicit. The proximity between these two elements usually forms an implicit link. While coding conventions bring added consistency, in some cases it can be difficult to determine what a comment refers to specifically or when it stops being relevant. Kaelbling [42] lamented the lack of an explicit link. He called for *scoped comments* that would clearly show what each comment refers to.

A side effect of this implicit link is that code can often change without the related comments being updated, causing synchronization problems. Particularly when developers are working towards a deadline, the code will

often take priority over keeping the comments up-to-date [33]. As a result, Grogono [33] suggested that wise programmers will ignore comments when maintaining aged code. Even during regular development, Ko *et al.* [45] found while studying structured editors that comment edits accounted for only 3% of the changes made by developers. And, only around 40% of these comment edits involved creating annotations, with 60% involving developers temporarily commenting out code.

In the software mining community, Jiang and Hassan [39] provided a preliminary result of a study on the evolution of comments in PostgreSQL. They discovered that the number of comments in source code remains constant after the initial stages of development have passed. Fluri *et al.* [23] looked at the ratio between comments and lines of code, based on three open source projects. While they make some interesting assertions related to the ratio of comments to source code and about developers making changes to the comments in the same revision as the code changes, they do not explain the 77% of the comment changes that they claim were not due to code changes. Also, they chose to study only three projects, of which all of them appear to follow agile design processes.

Marin [53] studied what motivates programmers to comment, by examining nine open source projects and extracting their code from the source code management systems. Three of his findings are of interest:

1. the rate of commenting differs not only between programmers, but also for the same programmer working on different projects;
2. larger code modifications tend to be commented more thoroughly; and,
3. programmers tend to comment more in code that is already thoroughly commented.

This suggests that even amongst developers who do comment, how or what is commented depends on the project, team, and individual developer.

Referring to the list of purposes annotations can serve developed by Marshall [55], software comments can act as procedural signals, placemarks, or to record interpretive activity. In general, communication is the primary reason

for creating comments, particularly for recording assumptions that are not explicitly expressed by the source code itself. However, this communication is not perfect. Given that comments are written in natural languages it is common for there to be some misunderstandings caused by these communications. However, as the software evolves, if the comments are not kept in sync with the code, miscommunications and misunderstandings can be exacerbated. Through a preliminary examination of the Mozilla project, Tan *et al.* [75] found that these problems resulted in the introduction of new bugs. Further, in the FreeBSD project, the authors found at least 62 bug reports about comments that are incorrect or confusing.

As previously mentioned, Ying *et al.* [84][85] conducted a study of the TODO task annotations in an IBM project looking at the different characteristics and uses of these annotations. They developed a taxonomy to classify the tasks. Their taxonomy has seven categories: communication, pointer to a change request, bookmark, current task, future task, location marker, and concern tag. Some of these categories were refined further into subcategories. Their work suggests that developers communicate to each other through these annotations. Our work builds on the work by Ying *et al.*, as we attempt to classify the tags created by participants in our studies (see Chapter 7). Our findings expand upon this work and raise some limitations for this form of inquiry.

To my knowledge this is the only attempt made so far to analyze how developers are using these types of annotations in software. Ying *et al.*'s taxonomy provides us with some direction for how these annotations might be used. However, they only looked at one project and did not follow up with the developers to see how accurate their findings were. So, the question remains, how do these annotations fit within the software development process?

In the remainder of this thesis I discuss the empirical studies that we conducted to investigate how developers utilize some of these tools for managing tasks and information in their daily work practices. In particular, these studies focus on task annotations and the tags created using TagSEA.

Chapter 5:

Research Design

"The outcome of any serious research can only be
to make two questions grow where one question grew before."
- Thorstein Veblen, *University of California Chronicle* (1908)

OUR objective in this research is to better understand what role annotations play in the work practices of software developers. We have designed a series of studies using a range of data collection and analysis techniques [13][50]. In this chapter, I reiterate our research questions and outline our approach to answering these questions.

5.1 Research Questions

To understand how developers create and manage annotations, we have set out to answer the following questions:

Q1: What is the content and structure of the annotations being created?

Q2: How are the developers using these annotations?

Q3: Where do they store this information?

Q4: Who are these annotations intended for?

Q5: Why are these annotations beneficial?

These research questions have evolved throughout the various stages of this work. As a result these questions are related, but different, to those that we were attempting to answer at the time of each of these studies (*cf.* [69][72]).

5.2 Overall Approach

Our research has fluctuated between exploratory and explanatory phases [13], as we have looked at the issues underlying how software developers annotate software. In general, we have been guided by a mixed-methods approach [14][40][49]. Mixed-methods research combines the collection and analysis of quantitative and qualitative data [76].

Our purpose for mixing methods has been for complementarity [32]. *Complementarity* aims to elaborate and enhance the results from one method using results from other methods. In our research, the qualitative data allows us to delve into the intricacies of the collaborative software process, the individualities of the software developers, and the practical use of the tools they use. The quantitative data allows us to take the observations we have and validate to some degree how likely our findings are representative of the software development community. While there are weaknesses, as with any research design, the combination of these techniques helps to address the limitations we would find by using any single method.

5.3 Studies

We conducted four empirical studies: two based around developer use of task annotations, and two focused on using tagging for managing annotations. The results and findings from each of these studies helped to construct answers to my research questions (see Table 2).

Table 2: Relationship between study methods and research questions

	Q1	Q2	Q3	Q4	Q5
Task Annotation Research (Chapter 6)					
Survey of Software Developers					
Questionnaire	X		X		
Eclipse Task Annotation Study					
Code analysis	X	X	X	X	X
Interviews	X	X	X	X	X
TagSEA Tag Annotation Research (Chapter 7)					
Industry Case Study					
Pre and post questionnaires	X			X	X
Focus group	X			X	X
Comment change analysis	X	X		X	X
Longitudinal Case Study					
Annotation analysis	X	X	X	X	X
Interviews	X	X	X	X	X

The studies are outlined here and discussed in more detail in Chapters 6 and 7. I synthesize and discuss the findings as they relate to my research questions in Chapter 8. Limitations of this research are described in Chapter 9. Given the overlapping nature of this research, the studies are presented in

thematic order, rather than chronological, for clarity. All of the studies conducted for this thesis have been approved by the Human Research Ethics Board at the University of Victoria.

5.3.1 Survey of Software Developers

We designed a questionnaire to confirm the insights we gained from the literature review and our intuition about how task annotations fit within the work practices of software developers. The online questionnaire was made up of 15 multiple choice and short answer questions. Respondents were asked to provide details about their use of bookmarks, task annotations, and how these annotation mechanisms are influenced by their project and team. The results of the survey, while exploratory, give us insights into the annotation behaviour of 81 developers.

5.3.2 Eclipse Task Annotation Study

From the questionnaire we noticed different practices for using annotations. However, we saw that task annotations were widely used – particularly TODOs. We were curious to confirm that task annotations were indeed in widespread use. We decided to examine ten open-source Eclipse projects that were well-established, widely used and from a variety of domains. We collected a snapshot of the most recent version of the code in each of the projects from their source code repositories to see what types of task annotations existed.

We were able to conduct semi-structured interviews [37] with the developers to gain some context into the annotations we discovered. We were also able to seek information about the individual and group processes related to the creation and management of their annotations.

5.3.3 Industry Case Study

Both of the previous studies looked at the current state of annotation creation and management in Eclipse. With an aim to improve tool support for these processes, our research group developed the TagSEA tool to add support for user-defined tagging and navigation. We conducted studies to examine how this tool might be used and what effects it might cause, starting with a case study [83] of professional software developers employed in industry.

Eight professional software developers from IBM were recruited to participate. We asked developers to share their initial impressions of the tool after a three week period of use, as well as at the end of the eight-week study period. We also asked developers to provide us with periodic snapshots of the tags that they had created and deleted. These tasks were classified using a qualitative coding strategy, involving three researchers including myself. The resulting taxonomy provided us with insight into overall potential uses for the tool.

There are three primary limitations of this approach. First, the tag data were collected at daily intervals and aggregated into three weekly intervals. This decision was made to see the broader trends as developers adjusted to the tool; the three week period was selected to align with the initial focus group. This collection strategy was course grained and could have missed smaller activities that occurred during a work day. Second, the length of the study was limited to eight weeks. We cannot foresee the longer term outcomes of using TagSEA based solely on this study. Third, the taxonomy was created by looking only at the waypoints that were created and deleted, and not the surrounding code. Without this additional context, there is a higher probability of misclassification. We attempted to minimize this last limitation by contacting the participants at a later date to verify our results.

5.3.4 Longitudinal Case Study

Both of the previous studies provided useful insights into tool use; however, we noted that developer use of the tool shifted even over the short eight-week period. We were curious to know what tool use might look like after prolonged use. Also, would the developers' motivation for tagging change over time?

We examined three open source projects – two internal to our research group, including TagSEA, and one external project. We analyzed the task annotations and waypoint use in the code at monthly intervals over a 12 to 24 month period. This approach has been taken by researchers in the software mining community (e.g. [17]). After analyzing the data, we interviewed the lead developer on each of the projects to validate some of our findings and explore their process and decisions.

5.4 Summary

To examine how software developers use annotations in their daily work, we designed and conducted four empirical studies. In the two subsequent chapters, we describe each study in detail, as well as present the results. This is followed by a discussion of the overall findings of the research in Chapter 8, where we synthesize these results as they relate to the research questions presented in this thesis.

Chapter 6:
Exploring Task Annotation Use in Eclipse

"Knowledge is of two kinds.
We know a subject ourselves, or we know where we can find information upon it."
- Samuel Johnson, in James Boswell's *Life of Johnson* (1791)

WE conducted two studies to examine how professional software developers create, manage and use annotations in their daily work.

6.1 Survey of Software Developers

We prepared an online questionnaire to gather insights about the current annotation practices of software developers.

6.1.1 Study Design

The questionnaire asked 15 questions related to how developers use bookmarks and annotations in their individual or group programming practices. Our target audience was professional software developers, with a bias towards Eclipse users. We asked Bjorn Freeman-Benson to blog about our survey, which he did on April 4, 2007. His blog is syndicated on Planet Eclipse [63], a website that combines blog feeds of people interested in the Eclipse community.

6.1.2 Summary of Results

The survey results confirmed that many developers make use of task annotations, but in varying ways.

We present selected results from the questionnaire that are relevant to the research questions in this section, grouped by themes. For a complete set of questions asked, please refer to Appendix B.

Demographics: We received 81 responses to our survey. Eighty-eight percent of respondents reported that they comment their source code. In addition, 88% percent of the survey respondents reported that they were currently working on a team. We also asked participants if their projects were open-source or proprietary. The plurality of respondents (49%) responded that they work on both, followed by only proprietary (37%) and only open source (14%). Our analysis found no noticeable difference in results based either on this measure or the size of the team.

Public versus private annotations: A recurring issue that developers face is whether to store their annotations in the code or privately in their workspace. Integrated Development Environments, such as Eclipse, provides a bookmarking feature for saving code locations in the workspace rather than in the source code itself. Based on the results from Murphy that stated that bookmarks were rarely used [61], we were interested in verifying this result. As Table 3 shows, 84% of respondents rarely or never use bookmarks.

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%)

Vocabulary: Similar to bookmarks, we were interested to find out if developers use task annotations, as well as which vocabulary they prefer (see Table 4). All but one of the respondents noted using task annotations. However, it should be

noted that Eclipse auto-generates TODOs when inserting code based on templates. TODOs were by far the most popular choice of keyword (used by 98% of respondents), followed by FIXME (43%) and XXX (15%). Other custom keywords used included: BUG, NOW, NOTE, and REV.

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%)

Metadata: From a cursory examination of source code comments written by software developers, we observed that developers included metadata, such as initials or bug numbers in their comments. We asked about this activity in the questionnaire; the results are shown in Table 5.

We noticed from these results that 51% of the developers marked their comments with their name or initials. Also, 44% of respondents reported making references in their comments to specific bug reports. This linking is interesting as it provides a reference to conversations that could explain design decisions about the code. Finally of note, only 10% of the developers reported adding memorable keywords. It would be interesting to probe this detail further to see if developers do not add memorable keywords, preferring to use other mechanisms, or if they simply do not perceive the keywords in this way. Unfortunately, this question does not give us an idea on how often or why developers add this metadata.

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%)

Team Process: We were curious what impact teams have on how individuals choose to annotation. We asked what types of group processes teams have around annotations (see Table 6). The majority of respondents reported having an informal agreement to use the same keywords as other members of their team.

Table 6: 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%)

Summary: The survey provided us with some insight into the perspectives of individual developers. To explore these issues further and examine some of the questions arising from the survey, we decided to change our unit of analysis and perform an analysis of the source code of a few projects.

6.2 Eclipse Task Annotation Study

To find out if developers actually use task annotations, we examined the source code from ten open source Eclipse projects and discussed the purposes of task annotations with four professional software developers.

6.2.1 Study Design

The source code we examined was extracted from the source code management systems of ten open source Eclipse projects. The Java files from these projects were then parsed to extract task annotations. Our focus in performing this analysis was to ascertain if developers are using task annotations, how prevalent they are in selected projects, and what vocabulary is being used. To further

understand how and why developers are using these annotations, we conducted interviews with developers from three of these projects.

6.2.2 Code Analysis Results

Corresponding with the results from our survey, we saw from an analysis of the code that the TODO keyword made up a clear majority of the task annotations found in the code (see Table 7). The other keywords that are configured by default in Eclipse (FIXME and XXX) were used significantly less. Custom keywords, such as REVISIT, INTRO, and CONTEXTLAUNCHING were also found in some of the projects.

Table 7: Task annotation 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

Another factor that may influence the presence of task annotations in the code is auto-generation facilities in the 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 removed or mostly removed (see Table 8).

Table 8: Percentage 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%)

This analysis provided a snapshot of the state of annotation use in ten open source Eclipse projects. These results complement those of the questionnaire, suggesting that developers do create these types of annotations.

6.2.3 Developer Interviews

We selected three of the projects to examine more closely, using convenience sampling [13]. Semi-structured interviews were conducted with four software developers. These interviews lasted between 30 minutes and one hour, and were conducted in-person in Toronto. The interviewer, a graduate student in our research group, was somewhat familiar with all three projects. He was given a list of the task annotations present in the source code, which he discussed during the interviews. Audio recordings of these interviews were transcribed for later analysis.

To analyze the interviews, three researchers with qualitative data analysis experience, including myself, coded each of the transcripts. These codes were grouped and regrouped to reveal themes. The researchers then met to agree upon a set of themes that encompassed their shared observations. The resulting themes were checked with the interviewer to verify that they resonated with his impressions from the interviews. These themes, as they relate to my research questions, will be discussed in Chapter 8. For a complete set of themes that emerged from this research, please see [72].

In this section, I present a brief overview of each project and the software practices of each developer as they relate to annotations. All three projects have been open for at least four years and have had at least one million downloads. All of the participants contributed to the projects as part of their professional work positions. To preserve the anonymity of the interviewed developers, we refer to the projects by fictitious names: Middleware, Backend, and UserInterface.

Middleware Project: This project provides an open source framework and Application Programming Interface (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. The team uses CVS to manage its source code 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 that work on this project are well versed in the entire project.

The two developers interviewed (referred to as Middleware-1 and Middleware-2) 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".

Backend Project: This 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 a bug 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-1). He indicated that the developers on this project make use of the keyword REVISIT to indicate that something may need to be addressed in the future. Several reasons were given for marking code 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 was 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.

UserInterface Project: This 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 code. There are seven committers, three of whom contribute on a daily basis. The project uses CVS as the source code management system 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-1) from this project makes heavy use of task annotations for his daily work. The developer added his initials and other metadata to TODOs. The metadata helps him filter TODOs in the Eclipse Tasks View. In this project, TODOs are used both for subtasks - things that must be completed 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 provides some context if the problem is 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 an explanation.

6.3 Summary

Our findings from these two studies – the survey and the Eclipse task annotation study – have provided us with insights into the work practices of software developers. It appears that while the annotations that developers use are easy to create, it can be a challenge to use them to manage tasks and information. In the following chapter, we look at the possibility of using tagging to address this challenge.

Chapter 7:

Examining Tags for Annotations

"Think of the tools in a tool-box:
there is a hammer, pliers, a saw, a screwdriver, a rule, a glue-pot, nails and screws –
the functions of words are as diverse as the functions of these objects."
- Ludwig Wittgenstein, *Philosophical Investigations* (1953)

IN this chapter, we examine the use of tags for creating and managing software annotations. Tagging allows developers to select words that are meaningful to them, and use these words as tags to create a flexible structure for returning to them. We make use of the TagSEA tool to explore our research questions further.

7.1 Industry Case Study

Given the exploratory nature of this research we wanted to see how TagSEA would be used in the field by real software developers. We designed a case study [83] to accomplish this objective.

7.1.1 Study Design

Instead of choosing an observational approach, which traditionally has a limited time frame and predefined tasks, we chose to conduct a logging study. This approach provided data over a longer period of time, using normal everyday tasks. The main weakness of this type of study is that the data can be misinterpreted based on its highly quantitative nature. To augment this data and counteract possible misinterpretations, focus group and questionnaire components were added to the study. This allowed us to validate our analysis of the logged data against feedback from the actual participants, as well as to get participant impressions of the tool.

Convenience sampling [13] was used to recruit participants from IBM Cambridge and IBM Victoria for an eight week study. Eight participants in total agreed to participate; however, only three of these eventually adopted the tool.

We collected data from the participants on a daily basis using diffs – a way of comparing two files. Diffs were performed between the previous and current versions of the code, to determine all of the comments that had been changed over the course of that day. The data from the participants was parsed to extract all of the waypoints. This data was aggregated into three week periods for analysis. We utilized this data to determine quantitative information about waypoint creation and tag reuse. We also had the same three researchers that coded the interviews in the Eclipse task annotation study (see Section 6.2.1) classify the waypoints and tags to develop a taxonomy of tag use.

7.1.2 Tag Taxonomy

Two major purposes emerged for waypoints: to provide information (INFO) and to mark a task (TODO). We further refined this to differentiate between instances, for example, where annotations provide information about who the author is (e.g. @tag ethan) and annotations discussing or interpreting a feature in the code (e.g. @tag copy.paste.position: get position relative to parent). The taxonomy is provided in Table 9. For this study we will use this taxonomy to describe each user's patterns for annotation creation. We revisit this taxonomy in the longitudinal case study (see Section 7.2.4).

Table 9: Tag taxonomy (version 1)

INFO (annotations that assert information)	
INFO-FEATURE	Information about a particular feature
INFO-AUTHOR	Information about a particular author
TODO (annotations that document a task)	
TODO-TEST	Task involving testing code
TODO-BUG	Task involving fixing a bug
TODO-CHANGE	Task involving changing or completing code
TODO-CHECK	Task involving checking something

7.1.3 User Stories

The following user stories are from the three developers who adopted TagSEA during the study period.

Two programmers from IBM Cambridge, C2 and C3, and one user from the IBM Victoria, B2, adopted the tool. We use information from the questionnaires, focus group and the data submitted to us in forming these user stories. The two Cambridge users also submitted snapshots of their source code comments for analysis. Using the tag taxonomy we were able to code the TagSEA waypoints created by the developers.

Table 10 shows a summary of the number and category of waypoints created per user, and how they changed over the study period, as well as a count of the number of waypoints. Table 11 shows the amount of tag reuse, measured as the total number of tags found in the code divided by the number of unique tag names.

Table 10: Classification of user created tags

The number of waypoints for the first data collection period (T1) based on our taxonomy is presented. The change in number of waypoints is indicated at the second (T2) and third (T3) data collection points. Note that waypoints were both added and deleted (indicated with a '+' or '-' sign)

	Types of waypoints								
User:	**INFO**	**INFO AUTHOR**	**INFO FEATURE**	**TODO**	**TODO BUG**	**TODO CHANGE**	**TODO CHECK**	**TODO TEST**	**Total Waypoints**
C2 T1	1	5	11	1	1		4		23
Δ T2		-1	+2/ -3				+5/ -1		+2
Δ T3							+1		+1
C3 T1			2		1	7	3	2	15
Δ T2					+4	-1			+3
Δ T3					+1/ -1	+5	+1		+6
B2 T1			6			12	7		25

Table 11: Tag reuse at the end of the study

User	Tag Occurrences / Unique Tags	Tag Reuse
C2	26/ 14	1.86
C3	45/ 19	2.37
B2	55/ 14	3.93

C2 (female) used the tool over the course of the eight week study and continued to use the tool afterwards. She joined the company shortly before this study and was assigned to extend existing code in a team project. The majority of her waypoints were created to support future tasks (see Table 10). In the interview with C2, we were able to verify that she mostly created waypoints for navigation. She said they were a reminder of the places that needed more examination. From an analysis of her source code, we were able to verify that she does not use task annotations and that very few of her source comments could be classified as supporting future navigation. This user only used one instance of a hierarchical tag, but she did use multiple tags (at most two) per waypoint. She did not add additional comments to the waypoints. There were instances of the same tag being used on more than six waypoints across three files of code.

One interesting tag we observed was called "home". In the interview, C2 indicated it was a special tag to support navigation. She also created one tag type to match her name and it was used to indicate several places where she had made changes in the code. She agreed that the Javadoc @author feature could have been used, but she said she preferred the lightweight approach of TagSEA for indicating her changes. At the end of the study, C2 indicated she would continue to use TagSEA and that it was preferred over bookmarks and Eclipse task annotations. C2 mentioned that "she liked TagSEA for being fast and lightweight to type in things".

C3 (male) joined the company six months before the study. He also used the tool over the course of the eight week study and continued to use the tool afterwards. C3 worked individually on code that provided infrastructure support for a larger project. For this user, there were changes made consistently throughout the study period to his tags and waypoints, indicating that he was able to manage and update the tagged waypoints effectively. When asked in the post-questionnaire if his use of the tool changed over time, he replied: "yes – I started with tags that described the function of the code, but I moved to tags that were

task-based (bugs, todos, API changes, etc.)". This user did not create any hierarchical tags, but used multiple tags frequently and in quite sophisticated ways. He also added comments for all the waypoints. One interesting tag this user added to a pre-existing waypoint was: "badbadbad". The existing tags on this waypoint were used to indicate what he needed to do. The additional "badbadbad" tag was used as a prominent reminder.

There are bursts of commenting and tagging activity that occur just before a major release, change, or refactoring. In the post-questionnaire, said the most useful instance for using waypoints was "tagging particular changes in an API for later integration". He used TagSEA as a "todo/ remember-this-stuff-for-later" tool. He tended to use TagSEA as a status indicator of his TODOs rather than to support navigation. The difference in use between C2 and C3 is reflected in Table 10, where C2's use of waypoints is more focused on asserting information, while C3's use of waypoints is more focused on TODOs. C3 said he would favor using TagSEA over tasks and bookmarks. He mentioned that TagSEA is more flexible because the user can define their own tags directly in code and they show up in the tags view and "you can pivot around them".

B2 (male) was one of two programmers that responded to our email from the BC lab. B2 was only able to use the tool for two weeks due to other deadlines at work. This programmer had over 10 years of programming experience. He worked on a demanding project with two other team members. He was unable to submit his source code to us for analysis. However, we asked him to submit interaction data (logged by TagSEA) on how he used the tool. This data showed us that he used the waypoints to support his navigation activities, and that he also used the refactoring facility for managing his tags. In the interview, he mentioned that the refactoring facility was very powerful. He created hierarchical tags and added additional comments. B2's intent behind tagging is summarized in Table 10. In the questionnaire he commented that he used waypoints for "exploring complex code paths" and "TODOs". He did not use

them for asserting information. A post study interview confirmed that he used the tagged waypoints to support things that needed further examination. He also made several suggestions for improvements, including that TagSEA should "use icons to represent type of waypoint (TODO, documentation, code review, code execution path, etc)" and that "the context sensitive help system needs to reflect the tag hierarchy". We were not able to analyze his code, so we could not determine how he used ordinary comments or task annotations for navigation. However, in the post-study questionnaire, he indicated that he used neither bookmark nor task annotations. He felt that the Eclipse task annotations were worthless because they could be obscured by many general TODO task annotations generated by the IDE. Although he was aware that they could be customized, he indicated in the interview that it was too tedious. When asked about bookmarks in the questionnaire, he replied: "I never use bookmarking. TagSEA augments the IDE tools and should become a standard part of Eclipse".

Non-Adopters: In addition to the three developers who adopted TagSEA, another five participants did not. Technical incompatibilities were cited by three of the participants; one participant rarely comments his code; and, one participant used custom task annotations and did not see the advantage to using TagSEA. While interesting, adoption of TagSEA is not a focus of the research questions posed in this thesis (*cf.* [69]).

7.2 Longitudinal Case Study

While the industry case study provided us with insights into how software developers might use tagging for managing code annotations, we wanted to examine the effects of prolonged use of the tool on annotation creation and software development processes. We conducted a case study of three projects that had used TagSEA for between one to two years.

7.2.1 Study Design

For this study, we examined the task annotations and TagSEA waypoints present in three open source projects, including TagSEA itself. In each project the developers had used TagSEA for between 12 and 24 months. To preserve the anonymity of the developers, we refer to the two other projects as VizTK and PIM. We note that there is some bias with both VizTK and TagSEA because they are both associated with the TagSEA project (see Section 7.2.2). No such bias exists with PIM.

7.2.2 Projects Studied

VizTK was originally developed in the late 1990's as a method for exploring and understanding C source code. It has since expanded to allow for the visualization and comprehension of large information spaces. VizTK is made up of 110,464 lines of Java code, which is maintained by two programmers at the University of Victoria. The developers on the VizTK software started using a prototypical version of TagSEA in February 2006 and still use the tool today. They were early adopters of the tool as they are part of the same research group.

PIM is an open source personal information management system written in Java using the Eclipse Rich Client platform. It is an extensible application for organizing emails, tasks, and finances. At the time of this study, the PIM project was comprised of 43,021 lines of Java code. It is written and maintained by one developer, with contributions from other developers. The lead programmer started using TagSEA in August 2006. He required that all contributors to PIM also make use of the TagSEA tool. The programmer from PIM recently joined our research group as a PhD student in September 2007, but prior to this he had no connection with us.

TagSEA was originally designed by ten researchers and developers at the University of Victoria. Through the life of the project, it has been actively maintained by one to three developers. The developers have been from the University of Victoria and IBM, with the development distributed between people in Canada, the United States, and Ireland. At the time of this study, TagSEA was made up of 106,501 lines of Java code. This project made use of the TagSEA tool as soon as it was viable – approximately May 2006 – to benefit from the features it offered and also to act as a test case for the tool.

7.2.3 Data Collection and Analysis

Our data were collected by taking snapshots of the state of the code at one month intervals from the source code management system of each project. We were able to retrieve more than two years of history for VizTK, a little less than two years for PIM, and a year for TagSEA. TagSEA underwent a major move and restructuring in CVS in January 2007, making it difficult to extract data beyond one year. PIM switched source code management systems (from CVS to Subversion) in early 2006. Several months of history are missing during this move.

We extracted the comments that contained keywords that are recognized as task annotations by the Eclipse JDT (*TODO*, *FIXME*, and *XXX*), as well as the custom *REVISIT* keyword. We also logged and counted comments that used the TagSEA specific keyword *@tag*. This gave us the ability to measure tag use, reuse, hierarchy depth, and the organization methods that users employ when using tags.

The quantitative data was augmented with interviews conducted with the lead developer from each of the three projects. In the interviews, the developers were asked questions regarding their usage of Eclipse task annotations and TagSEA. As with the industry case study, we manually classified each of the TagSEA waypoints and task annotations. While we were influenced by our

previous classification system, we allowed it to evolve to reflect the annotations from these projects. Two researchers, including myself, independently derived codes and then came to a consensus.

A sample of 20 annotations, stratified based on the classification scheme, were selected from each project to help inform the interview process. After a brief explanation of the classification system, interviewees were asked to classify these annotations using a think-aloud protocol. Since authorship of the annotations was not known at the time of selection, interviewees were asked to identify any annotations that were not theirs.

7.2.4 Tag Taxonomy II

The taxonomy we developed in the industry case study (see Section 7.1.2) was altered to adapt to the annotations created by the longitudinal users (Table 12). We added two subcategories to the INFO designation: INFO-LOCATION and INFO-FIX. INFO-LOCATION encompasses annotations with tags, but no message – essentially a simple placemark. INFO-FIX refers to items that would have originally been a TODO-BUG, but have since been fixed; the annotation has been kept and now serves to express details of the bug fix. For the TODO designation: the TODO-CHECK and TODO-TEST categories were merged and expanded to include annotations that implied an implementation question. TODO-CHANGE was expanded to include annotations that implied an intention that the developer would return to change the code.

Table 12: Tag taxonomy (version 2)

INFO (annotations that assert information)	
INFO-LOCATION	An informational marker with no message
INFO-FIX	Information about a bug fix
INFO-FEATURE	Information about a particular feature
INFO-AUTHOR	Information about a particular author
TODO (annotations that document a task)	
TODO-BUG	Task involving fixing a bug
TODO-CHANGE	Task involving changing or completing code
TODO-CHECK	Task involving checking or testing something, or posing an implementation question

As this was our second time performing this classification, we were aware of the challenge of making a correct classification. It is particularly difficult without examining the code context. We verified our classification with the developers, using a sample of the annotations.

The dynamic between TODO and INFO appeared to be more blurred than we had originally thought. The distinction between a TODO-CHANGE and an INFO-FEATURE depends upon a developer's intention to return and refactor the code. Some of the annotations in the VizTK code that we had labeled as TODO-CHANGE were characterized by the developer as INFO-FEATURE because he did not expect to ever return to the locations. These annotations instead provided information should a change in that area of the code become necessary. We saw a similar transition between TODO-BUG and INFO-FIX, although the boundary is more explicit; either the bug is fixed or it is not.

7.2.5 Results

In this section we discuss our findings from an analysis of the quantitative data followed by the findings from our interview of the developers of the projects.

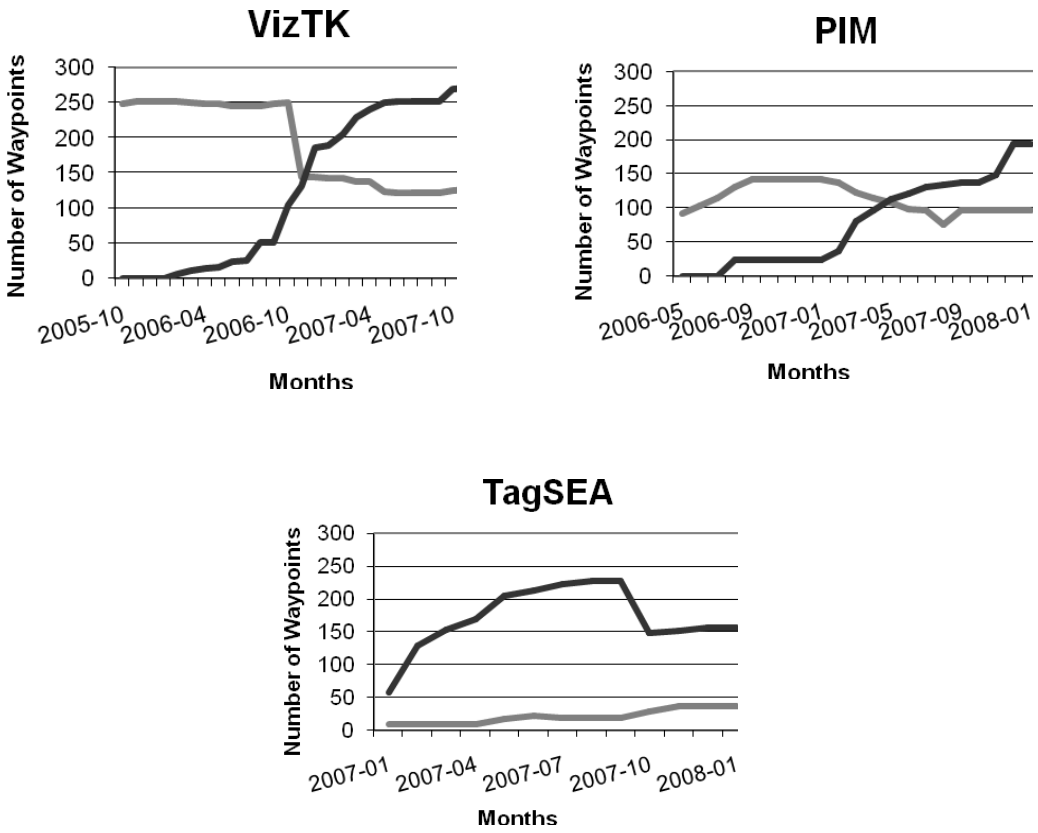
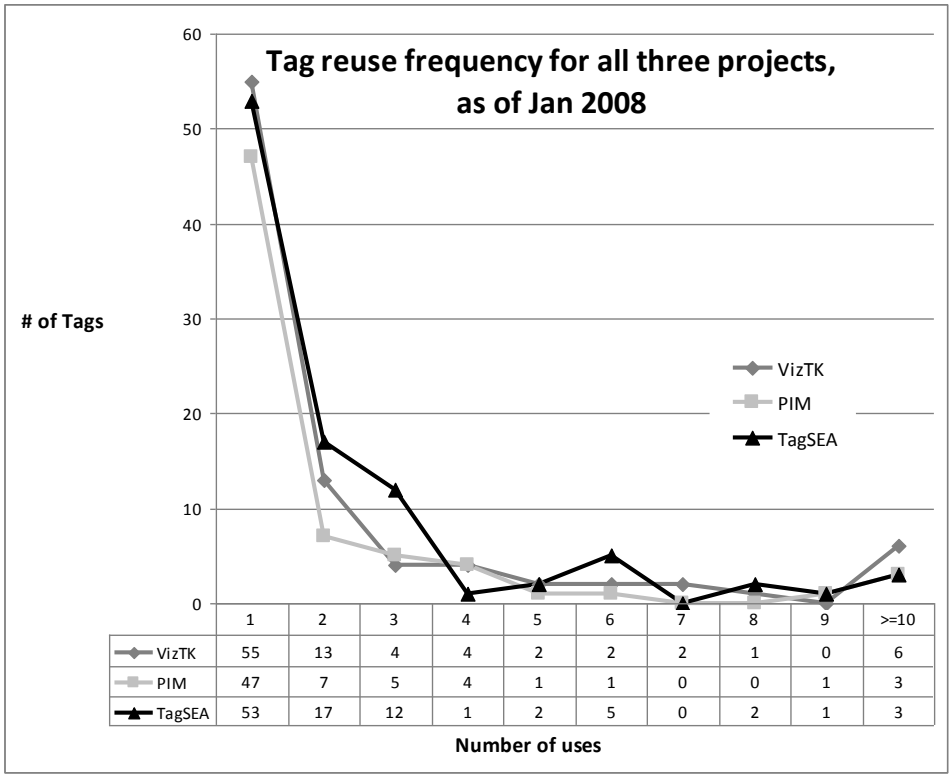


Figure 4: Waypoint (dark) and task annotation (light) use in VizTK, PIM and TagSEA

Waypoint Use: From the extracted comment data, we counted the number of Eclipse task annotations and TagSEA waypoints that were present in the code at one month intervals (see Figure 4). Both the VizTK and PIM projects display a large drop in the number of Eclipse task annotations, as waypoints became the preferred annotation mechanism. The TagSEA project always displays a higher number of waypoints than task annotations because the developers made use of

the tool from the start of our data collection. There is one large drop in the number of waypoints in the TagSEA project in October 2007. This is because one of the developers deleted waypoints that were no longer relevant. These results indicate that waypoints are used at least as frequently as Eclipse task annotations, and may serve additional purposes for the developers.

Tag Reuse: We also calculated the frequency of tag reuse (see Figure 5). That is, we asked the question, "How many tags are used only once, twice, etc.?" We looked at the data extracted from the most recent version of the projects and discovered a common pattern: a small number of tags were used very frequently, and a larger number of tags were used infrequently. It is interesting to note that while most tags are only used once, some tags are highly reused. For example, in VizTK, the tag VizTK.grouping is used over 50 times. We found that, on average, tags in VizTK were reused 3.0 times; 2.8 times in PIM; and 1.7 times in TagSEA.



	1	2	3	4	5	6	7	8	9	>=10
VizTK	55	13	4	4	2	2	2	1	0	6
PIM	47	7	5	4	1	1	0	0	1	3
TagSEA	53	17	12	1	2	5	0	2	1	3

Figure 5: Frequency of tag reuse

We visualized the vocabularies from the latest version of code as tag clouds (see Figure 6-8). Tags shown in larger fonts indicate higher reuse. The tag clouds represent the vocabulary created by all developers on each project and show the unique vocabularies that emerged. The tag clouds reveal that a hierarchical naming scheme was used on each of the projects. For example, nearly all of the tags in VizTK begin with the project name as the prefix. In PIM, many of the tags begin with the prefix “todo” or “tour”. Although the size of the vocabulary for each project is quite large, they are organized into a hierarchical structure that groups similar tags together (the reuse graph distinguishes VizTK.x and VizTK.y as unique tags). The depth of the tag hierarchies indicates the amount of structure and grouping that is used for the tags in each of the projects. The mean and maximum depth of the hierarchies is shown in Table 13. For the most part, tags had a hierarchy depth of at least two and a maximum hierarchy depth of three for VizTK and PIM, and five for the TagSEA project.

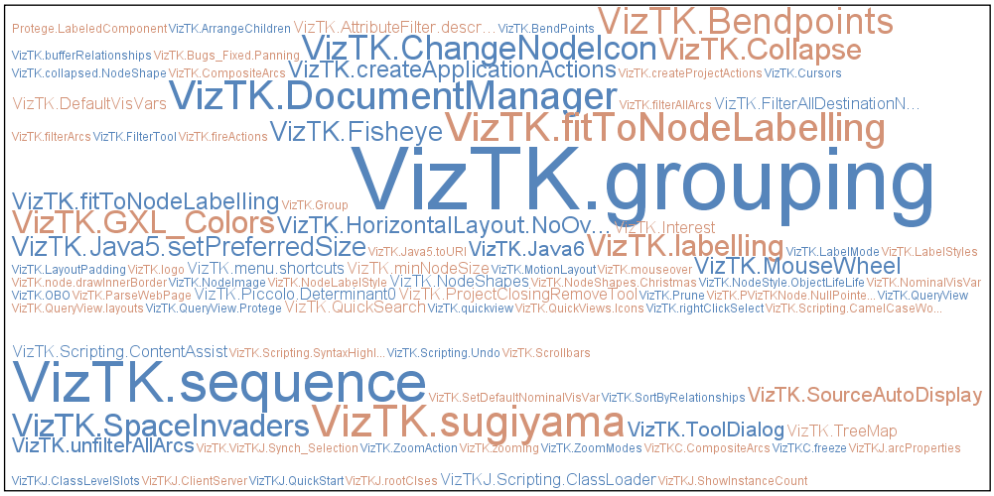


Figure 6: VizTK tags visualized as a tag cloud

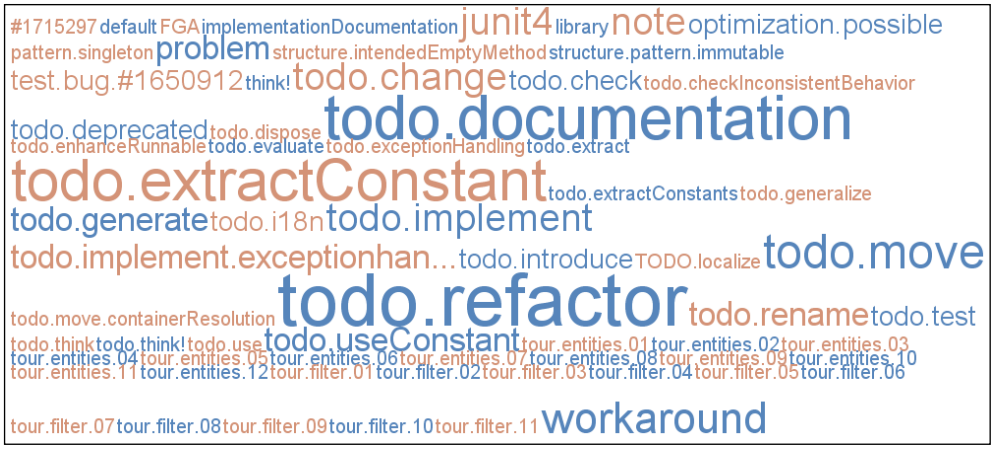


Figure 7: PIM tags visualized as a tag cloud

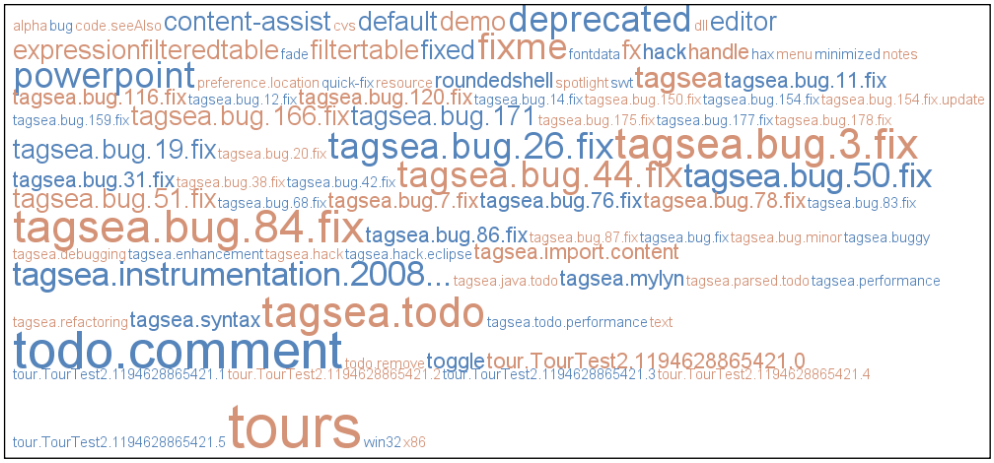


Figure 8: TagSEA tags visualized as a tag cloud

Table 13: Tag hierarchy depth for each project over the course of a year.

	Jan. 2007		July 2007		Jan. 2008	
	Mean	Max	Mean	Max	Mean	Max
VizTK	2.24	3	2.24	3	2.18	3
PIM	3	3	2.28	3	2.20	3
TagSEA	1.71	4	2.31	4	2.50	5

7.2.6 Developer Interviews

We report on the findings from our interviews with the software developers on each of the projects.

V1 (male) has been the primary software developer working on the VizTK project since November of 2005. Prior to using TagSEA he used TODO task annotations to mark incomplete code, code that could be improved, and places to be tested. He was not aware of Eclipse's bookmarking functionality. He began using TagSEA in the summer of 2006. He chose to use TagSEA so that he could add hierarchical tags to his free-form comments, affording easier navigation in the future. Examining his code, this is clearly the predominant use of waypoints. He also still uses TODOs, but – according to him – a "lot less" than before.

Over time he noticed that he did not actually navigate back to the waypoints often, so he became less motivated to place them in the code simply for the potential benefit of returning to them. Now, he tends to add tags to his comments when he needs to document concepts that cross-cut the software structure, for example, a set of locations that might all be affected by a change. Also he marks locations that require a revisit.

This developer made use of hierarchical tags with the first level of the hierarchy frequently reflecting the project name (e.g. VizTK.something). He

occasionally expanded the hierarchy to three levels to add more distinctions. To navigate back to the waypoints, he used the Waypoint Viewer's filtering capabilities, if he remembered the tag name. Otherwise, using the Tag Tree, he expanded the top level of the hierarchical structure and browsed for the target.

He added author metadata when working with others on similar areas in the system; otherwise, he did not add this additional metadata. To maintain a consistent tag vocabulary he used the content assist features in TagSEA.

V2 (male) is creator of the open-source PIM project. Since using TagSEA, he has stopped using Eclipse task annotations and instead uses hierarchical tags that begin with the todo keyword to mark tasks. His use of TagSEA changed gradually as he became more familiar with the tool. Initially he used TagSEA to create tours for documentation purposes (by using the top-level tag keyword). Subsequently, his use of TagSEA migrated to creating todo's and providing information about the code. To return to these locations, he does not use the Waypoint Viewer for navigation. Instead, he creates his tags in the editor and then browses for them using the editor or search tool. The developers on this project did not make use of author and date metadata. V2 felt that this added metadata adds too much clutter to the comments.

When asked if he used the content assist and refactoring features of TagSEA, he said that he did not but he "should have." He plans to use refactoring in the future to better organize the tags that he creates.

V3 (male) is the primary software developer on TagSEA. He uses auto-generated TODO annotations to mark places in the source code that need to be documented. His initial use of TagSEA centered on marking bugs and fixes in the code, along with their associated bug number. His intent was to link the source code to the bug repository information and associated Mylyn (formerly Mylar [43]) task context. Over time, the developer decided that this provided no future use and was cluttering the code.

When entering waypoints, he rarely used content assist except for adding author and date metadata because he often finds that simply typing the tags is faster. He rarely adds metadata because he believes that it clutters his comments. He has used the refactoring feature for large-scale deletions of waypoints that are no longer needed, but he only occasionally used it for reorganizing his tags.

His main use of TagSEA now is to provide reminders for short term interruptions (e.g. tagsea.starthere) and to mark areas that could be improved (e.g. tagsea.performance). The hierarchical structure of his annotations is similar to the structure used by V1 from the VizTK project (e.g. projectname.reminder).

The main benefits of TagSEA for this developer are its flexible vocabulary and extensibility. In particular, the vocabulary is useful for collaborating with other developers. While not using the navigation facilities of TagSEA for his bug annotations, he has been using it for reminders. To locate tags, he uses a combination of browsing the code, using the Waypoint Viewer, and the visual decorations in the Package Explorer. From examining his code, he predominantly uses tags for the category we coded INFO-FEATURE.

7.3 Summary

By studying the way software developers use TagSEA, we can explore issues surrounding tagging and organization in software annotations. We synthesize the results from the studies presented in Chapters 6 and 7 in the next chapter.

Chapter 8:

Findings

"All our knowledge has its origins in our perceptions."

- Leonardo da Vinci, *The Notebooks of Leonardo da Vinci,* translated by Jean Paul Richter (1883)

GATHERING the evidence from our research studies we present the findings towards each of the research questions.

8.1 Q1: What is the content and structure of the annotations being created?

We draw upon results from all of our studies to discuss the quantity of annotations developers created, which keywords and tags they used, and what other metadata they often added. Where applicable, findings will be presented for task annotations and TagSEA waypoints.

8.1.1 Quantity of Annotations

Bookmarks are infrequently used. Eighty-four percent of respondents to our survey never or rarely use bookmarks. This low interest in bookmarking was confirmed by the developers we interviewed, as well as informally by the Eclipse developer responsible for maintaining the feature in Eclipse. Because these annotations are stored in the developer's workspace, rather than in the code, we are unable to report exact numbers.

Task annotations were present in all ten open source Eclipse projects we examined. The number of task annotations ranged from four to 515, depending on the project. This variation is influenced by the project size, the number of

developers, the individual developers themselves and the state of development on the project.

Waypoints were created by the participants using TagSEA. In the industrial case study, the number of waypoints varied from 24 to 26 after a few weeks of use. For the longitudinal case study, we found much higher numbers, ranging from 157 to 271 at the end of the study. Larger team sizes and a longer opportunity to create waypoints probably led to this higher range. Interestingly the number of Eclipse task annotations actually decreased over time in two of our longitudinal samples (VizTK and PIM), indicating that waypoints were now replacing at least some of the role that task annotations had previously served.

8.1.2 Keywords and Tags

TagSEA tags are similar to keywords for task annotations, with two key differences: (1) the vocabulary is user-defined rather than being limited by default to a handful of words, and (2) developers can add structure by creating hierarchical tags. We consider the influence of vocabulary for both types of annotations and discuss the influence of vocabulary on refinding.

Task annotations: From our survey and the results from our examination of the use of task annotations in ten Eclipse projects, we noticed that developers use a variety of predefined keywords. TODO has the highest frequency of use, with XXX and FIXME 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 agreed on within their teams. For the Backend and UserInterface developers, TODO and FIXME mean the same thing. But, UserInterface-1 admitted that when he sees other people's

FIXMEs he interprets them differently, recognizing that other members in his team attach different meanings to the same terms.

This vocabulary can also be customized. We only observed one project where a custom keyword, REVISIT, was in widespread use (Xerces-J). UserInterface-1 reported creating custom keywords for his own use. He added "an acronym that represents the work item I'm working on at the given point in time". For example, he may insert FE for File Editor. He also associated a priority with the task annotation, e.g. LOW for low priority. Similarly, Backend-1 noted that a REVISIT was a much stronger indication that the annotated task should be revisited. However, while customizable, some developers, such as VizTK, were unaware that other keywords than TODO are recognized by the compiler as task annotations. If developers are unaware of the existing vocabulary choices, it is highly unlikely they will know they can customize the vocabulary.

TagSEA waypoints: With TagSEA waypoints there are no predefined keywords or tags. Developers can also organize their tags hierarchically. All except two of the developers in our studies made use of the hierarchical tagging feature. The longitudinal users and the programmer B2 all found the hierarchies to be a useful way for organizing their tags, particularly based on different projects or components. One interesting result was that the two professional programmers from Cambridge (C2 and C3) did not, for the most part, use hierarchical tags. We speculate this may have been because these two programmers were experienced with social bookmarking tools and there are reports of experienced tagging users creating their own conventions to encode hierarchical relationships across tags [34].

Vocabulary and refinding: Whittaker and Sidner [82] noted in their study of email use that when people filed their emails into only a handful of folders that the complexity of finding information was similar to not filing at all. We see the same situation with task annotations, where they are usually grouped between

one to three keywords (i.e. TODO, FIXME and XXX). By allowing tagging, we saw that developer annotations were spread out amongst various keywords. For the projects using TagSEA in our longitudinal study, we saw that average tag reuse varied from 1.8 to three. In contrast, the keyword reuse for task annotations ranged from four to approximately 171.

Despite the increase in vocabulary choices for tagging, we have yet to see evidence of the negative impacts of the *vocabulary problem* [27] that are often seen in social tagging (e.g. [30] and [60]). We suspect this is largely because the developers who have been using TagSEA during our studies are not doing so collaboratively. We might not expect to see this problem experienced in the same way in code for at least two reasons: (1) items in social bookmarking can be tagged multiple times, while in code this does not appear to be the case, and (2) sharing of tags is important for there to be different terms for the same items; however, we are not sure if or how developers will share their tags.

8.1.3 Additional Metadata

Beyond creating tags or keywords, we observed various other types of metadata being used to describe the annotations, including free text messages, date, and author information. These types of metadata can be important for refinding [77].

Message: Not surprisingly, descriptive text was added to most task annotations that we viewed in the task and comment extractions. These messages tended to be concise and of limited length. Although in a few rare cases, predominantly found in the Backend project in our analysis, the annotations were embedded in block comments with lengthy text, usually followed by a question.

All of the developers using TagSEA added text to their waypoints, except C2 from the industry case study. However, we noted that nearly half of the waypoints in the VizTK project did not have messages. Interestingly, when asked

about two of these tags, the VizTK developer was able to remember the context of the annotation from memory from the tags alone.

Date: While our survey indicated that 19% of developers add date metadata to their comments and our interview data indicated that some developers add date metadata to their comments, we found no evidence that developers are adding this metadata to the task annotations or waypoints (with the exception of the TagSEA developer). Date metadata is most commonly found in the comment at the top of each file. It is possible the date information is removed before a commit as it may be used for a short term task.

Name and initials: In addition, developers often add their initials to task annotations. Three of the four Eclipse developers we interviewed indicated that they add initials to facilitate identification and navigation to their comments. Middleware-1 indicated he removes them before checking in his code. But another developer on the same project leaves them there to indicate an issue that requires further consideration. With respect to team usage, Backend-1 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 [laughs]".

With waypoints, author metadata was not added by the industrial programmers because this feature did not work correctly in the earlier versions of the tool. One of the developers, C2, created a tag to match her name. One of the longitudinal developers (VizTK) noted adding author metadata while working closely on part of the code with another developer.

Bug numbers: Two of the interviewed developers also mentioned occasionally adding a bug number to their TODOs. But UserInterface-1 mentioned that he no longer does this as hotspots in the code tend to accumulate many comments with bug numbers. We also see 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 in our analysis.

8.2 Q2: How are the developers using these annotations?

With an idea of the types of annotations that developers create, we turn our attention to how developers are using them. Based on Marshall's explanation of the purposes of annotations [55], we might expect software developers to create procedural annotations (as reminders), placemarks (as a refinding mechanism), and interpretive annotations (describing or explaining code). We consider developer use of task annotations and TagSEA waypoints to describe developer intentions.

Articulation work consists of all activities that are needed to coordinate a particular task, manage subtasks, recover from errors and assemble resources [4]. From our analysis of the open ended questions in the survey, as well as the interviews, we identified several kinds of articulation work that are supported by task annotations and TagSEA waypoints. We give examples of these here, drawing primarily from the interviews with the Eclipse developers.

1. **Short term tasks** are generally quick items that the developer plans to accomplish in hours or possibly days. Middleware-2 described them as TODOs to be addressed in "the very, very short future".

2. **Subtasks:** UserInterface-1 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".

3. **Problem indicators:** 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.

4. **Multi-tasking:** UserInterface-1, 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... you don't want to divert your focus. So you say okay, you drop a TODO".

5. **Deferring low priority tasks:** Developers interviewed talked about using them for possible future, but low priority tasks. As UserInterface-1 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 ... you never have time [laughs]".

6. **Error handling:** Software developers must write code that handles many possible, but unlikely situations. Some developers, such as Backend-1 and UserInterface-1, delay addressing all of these edge cases. Backend-1 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...". The annotation acts as a reminder that the code needs to be written.

7. **Identifying cross-cutting concerns:** The VizTK developer used TagSEA to tag items in the code that were related and might all need to be changed at once. This allowed him to easily refind all of the locations at once.

This list of examples suggests that annotations in code are primarily being used as reminders or placemarks for short term tasks. Future work is needed to validate, and potentially expand upon, the uses for annotations. In the next section we look at how this short-lived purpose for annotations fits within software development processes for task management.

8.3 Q3: Where do they store this information?

Developers often manage the tasks their team must fulfill through an issue tracking facility. They can also mark locations and tasks with annotations directly in the code. With two popular ways to store task information, how do developers choose between creating an annotation in the code and creating a new work item in the issue tracking system? The following themes look at the types of information that are being stored and some factors, such as the size of the task, that influence the decisions made by developers.

8.3.1 Ephemeral, Working, and Archival Information

Barreau and Nardi [5] suggested that information serves one of three roles: ephemeral (i.e. short lived), working or archival. Developers have tools that support all of these roles. In the previous section, we saw evidence of developers using task annotations and tags for managing short-lived tasks. We also learned that many software teams use issue tracking systems, such as Bugzilla or Jira, for keeping track of issues, tasks and work items. These systems are a common way of managing the many active (i.e. working) tasks that emerge, particularly on large or distributed projects [35]. Finally, source code management systems – such as CVS, Subversion, or ClearCase – are commonly used to archive and manage code that can later be checked to answer a question [59].

The distinction between short-lived and working information provides us with some initial indications of how developers choose between annotations and

issue tracking systems. In the following themes we draw upon the interviews we conducted to explore this issue.

8.3.2 Size and Scope of Task

We found that a decision to create a task or open a bug depended, in part, on the size and scope of the task. For larger items, the Backend team and the PIM developer create bug reports in their issue tracking system, while smaller items are created as task annotations. 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-1, smaller sub-tasks of a bug report are created as TODOs. In the Middleware project, all work is driven by bug reports, and any TODOs that are created are cleaned up before the code is submitted to the repository. TODOs are created for error handling as 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 to attend to these items. The TODO is there to remind them to deal with it later.

8.3.3 Overhead of Work Item Creation

Developers discussed the costs and benefits of creating a bug over a “quick TODO”. For tasks that could be completed 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. 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-1 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”.

8.3.4 Project Maturity and Visibility

Middleware-1 commented that the developers on that project are more likely to open bugs now than write TODOs because the project is more stable. Now that their code is also under close scrutiny, since it is open source and there is no internal code, there is more reason to open bugs that are more visible to the community.

8.3.5 Summary

Keeping track of information for short and long term tasks can be challenging. Ellis *et al.* [21] interviewed eleven participants from the Eclipse, Mozilla and Rational projects. They found that all, but one of them, were worried about losing track of key bugs due to high volume. New bugs are placed in an "inbox", and similar to overwhelmed email users, one user commented about "losing the battle of the inbox". Furthermore, just like an inbox, not all of the incoming issues are relevant. Some include trivial bugs, bugs with bad descriptions, obscure or hard to reproduce bugs, and especially duplicates.

As such, neither annotations in software, nor issue tracking systems are perfect solutions for managing tasks and information. The decision to choose one appears to centre on the size of the task and the level of interaction with the community. In the following section, we will look at who these annotations are for.

8.4 Q4: Who are these annotations intended for?

We found that the use of annotations 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.

8.4.1 Annotations for Self, Team and Community

The survey indicated that many groups share an informal process for shared vocabulary use (49%) and that some (12%) also have 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 [48]. In the longitudinal study, all three developers could identify which TODOs were theirs, even if it was not explicitly noted in the code.

8.4.2 Annotations are Seldom used for Direct Communication

Ying *et al.* [84][85] 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 developer 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. Bug creation is also seen as a facility for building community.

Task annotations 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). This is similar to the notion of public and private annotations that Marshall and Brush [56] explored, and how only a small fraction of the

annotations that were for individual use could be interpreted by outsiders. However, developers did mention looking at TODOs if one was nearby when they were fixing a bug.

8.4.3 Public versus Private Annotations

The term *bounded transparency* is used in Computer Supported Cooperative Work 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 [4]. 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 teammates, 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 public versus private annotations also emerged in focus groups we held with software developers on the use of a software tagging tool. During the focus group discussion at the Cambridge lab there was significant discussion on whether waypoints should be personal or shared artifacts. The non-adopters in particular were concerned about this. One of the developers used the phrase "graffiti" to describe the annotations added to the code. Others liked that the tagged comments were in the source code – e.g. from C3's

questionnaire: “No – public tags work for me. At worst, they're uninteresting to other people. At best, they help document the code.” But there was a need expressed to tag code that has read-only access. Our conclusion from this is that both private and public waypoints are needed, but they should be presented to the user in an integrated way, rather than as separate tools with different user interface affordances. Moreover, the tool should have mechanisms so that annotations, if present in the code itself, can be filtered easily when formalizing a version of the code.

8.5 Q5: Why are these annotations beneficial?

We have seen what types of annotations developers create, how they use them, where they create them and who they are for. In this question, we ask: why are these annotations beneficial?

8.5.1 Short-term and Long-term Reminding

The reminders that developers create are for both short-term and long-term tasks. Since software developers are often interrupted during their daily work [38][54], several developers mentioned the need to pick up a task where they left off, following a break in their work. In terms of short-term reminders, we saw evidence of users creating ephemeral annotations to help them return to the location they were working on prior to an interruption.

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. Similarly, the TagSEA developer created a tag called “tagsea.starthere” to mark a single place to resume his interrupted task. With a unique tag name, it is easy to remember and refind. We also saw evidence of developers creating tags for short-term tasks, so that they could not be easily forgotten.

As mentioned previously, one developer (C3) created a tag named "badbadbad", signifying a location that he needed to come back to and fix. Interestingly, the developer started off naming the tag "badbad", and added an additional "bad" later on, presumably to increase its ability to remind.

Reminders for long-term tasks are often less memorable and usually form a short question or statement of what needs to be done. Examples include reminders to identify code that needs to be completed, code that should be refactored or improved, or code that needs to be tested.

From the code analysis we saw that many annotations that express tasks are still found in the code. Are they simply not removed when the task is completed? Sometimes this is the case, such as when the annotation is intentionally left for future reference. Sometimes the tasks that are tagged are simply not attended to, and so the annotations remain. As Backend-1 reported: "we all put these comments in our code and then don't look back at them. None of them actually remind us...".

From interviews with Eclipse developers, we noticed that many task annotations do not serve as active reminders. With our interviews in the longitudinal study, we found instances where developers would identify waypoints that had been in the code for some time that contained information about a task that was still outstanding. The VizTK developer noted he has no intention of returning to many of the annotations that are in the code that indicate a task to be completed. However, he indicated that these waypoints were more to remind the developer of the context of the code, rather than to remind him of work that remains to be done.

8.5.2 Refinding Relevant Information

Annotations are often used to assert information about locations in the code. Frequently, these annotations also act as reminders, providing warnings about

potential problems in the code or additional context that could potentially help developers understand the rationale behind the code.

We saw instances where developers would create waypoints for these purposes, which we confirmed in the interviews. We also saw uses of tags that were chosen to aid in refinding. For example, one user used the tag “home” to facilitate the rapid navigation to the focal point of a task under way.

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 Tasks View, and searching. UserInterface-1 mentioned that he uses the markers shown in the ruler of the editor to determine if there are TODOs that needed to be revisited. He also filters on his initials in the Tasks View to show the TODOs that he had added.

Through the interviews in the longitudinal case study, we found evidence that the hierarchical structure of tags allowed developers to refind annotations. Interestingly, each of the developers used different techniques for navigation. The TagSEA developer usually only returned to certain waypoints, those that he could remember. He utilized filtering to find and navigate back to these waypoints. The VizTK developer sometimes does the same, but more often opened the Waypoint Viewer, chose the project tag from the Tag Tree and browsed through the sub-list to find the tag he wanted. The PIM developer

rarely uses the Waypoint Viewer at all, choosing to refind his tags either by text-based search or simply browsing through the source code in the editor.

The reuse of tags indicates that tags are sometimes being used to document cross-cutting concerns in the code. The VizTK developer originally marked locations that he thought he might return to. However, as he noticed that he was rarely returning to them, he started to only tag locations that cross-cut the software space. These tagged locations mark places where a change in one place would require changes to other parts of the code, providing him with an easy way to navigate back to all of these locations.

8.5.3 Just-in-case Tool Support for Refinding and Reminding

Navigation events through the Waypoint Viewer appear to occur less frequently than we had originally thought. At first we interpreted this lack of use to mean that TagSEA's value may lie in its documentation ability rather than its navigational features. However, we feel that we may have perhaps, through our coding and analysis, drawn too strong a line between the purposes of providing information (reminding) and supporting tasks (refinding).

To organize our knowledge in the world we often file things away and store similar information together to support future "just-in-case" refinding. In many cases, we are not sure we will need that resource again, but we file it anyway. We organize these informational resources in such a way that we can refind them relatively easily. Sometimes we annotate them with additional information that reminds us of important related information and adds value to a possible future task. Similarly, we believe that the TagSEA tool was used in this manner. The information was added to support not only future refinding (through the use of tags), but also to support reminding of why the waypoint is there (through the tag vocabulary, message and metadata).

8.6 Summary

Through this research we have seen that developers create annotations in the code to record and manage short term tasks. These annotations can act as reminders or placemarks for future refinding. By storing them in the code, developers avoid some of the overhead associated with more formal processes, such as using an issue tracking system. They also strike a balance creating between private annotations stored in the workspace (e.g. bookmarks) and the easily accessible public annotations in the issue tracking system. Ultimately, we found that developers are creating these annotations to externalize their memory for the purposes of reminding and refinding.

Chapter 9:
Contributions and Future Work

"Now this is not the end.
It is not even the beginning of the end.
But it is, perhaps, the end of the beginning."
– Winston Churchill (1942)

OUR work has aimed to understand what role annotations play in the work practices of software developers. Our findings have provided insights that may enable tool designers to improve the features available for reminding and refinding in software development environments. In this chapter, we present implications for software process and software tool design, as well as contributions of this work. As our findings are based on empirical research, we outline some limitations and point to future work to resolve them.

9.1 Implications

9.1.1 Implications for Software Process

In a lightweight software development process, such as eXtreme programming, 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 task annotations to track *ad hoc* tasks often claim they can manage this transition, but in practice, many annotations are never formally migrated to issue tracking systems and remain hidden in the code for years. We feel our work addresses this issue in two ways.

Subtask creation: Because developers use task annotations to manage subtasks, software processes could support the notion of subtasks and allow their

specification in this less formal manner. This would allow for easier grouping and management of tasks. 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:** Related issues are often discovered by developers with they work. In some cases a formal work item is filed in an issue tracking system. 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 migrating *ad hoc* tasks to an issue tracking system is necessary. Such a process could also result in a removal of the old TODOs from the code.

9.1.2 Implications for Tool Designers

Developers, who use development environments such as Eclipse, spend approximately 50% of their time in the editor [61]. It is not surprising, then, that they often make notes for themselves in the source code. Considering the implications for software process and how developers use IDEs, we have compiled a number of suggestions for tool designers.

In-code task annotations should have improved metadata support: 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 in the issue tracking system would also be useful.

Filtering of task annotations: One interviewee who used the Eclipse Tasks View remarked how it was 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 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 issue tracking systems 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 a release.

9.2 Contributions

We believe that our work makes the following contributions to software engineering research:

Synthesis of multidisciplinary literature: We have synthesized literature from studies of human memory, task and information management strategies, and software practices to understand some of the underlying factors influencing the use of annotations in software development. In particular, we examined how

annotations can be used by software developers to aid reminding and refinding of information in source code.

Examination of developer processes for annotations: We have examined how developers use annotations in their daily work by interviewing developers, analyzing source code, and conducting a questionnaire. Our findings build upon the work by Ying *et al.* [84][85].

Evaluating the use of tagging in software: We have evaluated the TagSEA tool, looking at how a more flexible vocabulary and greater ability to structure annotations might aid developers in the management of their software tasks and information. While much research has been carried out on social tagging on the web, we are not aware of any research looking at how tagging can be used by developers for annotating software.

Uncovering implications for software process and tool designers: We have presented a list of suggestions that designers of tools for supporting annotations should consider. We also have provided suggestions for improving the software process for managing task-based annotations.

9.3 Limitations

In this section, we discuss weaknesses of our current research, and point toward future research to resolve them.

First, the questionnaire we conducted intentionally recruited developers from the Eclipse community. We wanted to minimize the effects of different development environments on our results, particularly as the TagSEA tool is built for Eclipse. As a result of this decision, our results may not be generalizable to other integrated development environments, such as Visual Studio. Potential other biases also exist with this focus on Eclipse developers, such as a possible

bias towards the Java programming language or agile work practices. While we cannot confirm these biases exist based on the questions we asked, we are aware of their possible existence.

Second, we extracted source code from repositories in both the Eclipse task annotation study and the longitudinal case study. The timing of these extractions could have affected our results, depending on the stage in the development lifecycle at each extraction point. Additionally, an analysis of this kind only captures those annotations that are committed to the source code, and not those that are kept in a developer's workspace or those that are deleted before they can be committed to the repository. We mitigated the impact of these limitations by conducting interviews with developers and specifically asking about their software practices.

Third, although we studied real developers doing real work, we hope to increase the number and diversity of case study sites. We hope to find more teams, unrelated to our own that are willing to discuss how task annotations fit in their work practices. We also hope to find more teams willing to experiment with the newer, more powerful version of TagSEA. And we hope to obtain data from multiple members of each team.

Finally, we have observed a tendency for annotations, particularly TagSEA waypoints, to become more complex over time. We may need to consider the "lifecycle" of a waypoint as it accumulates more meaning and significance. Are there distinct "classes" of waypoints, for example individual versus group annotations? We hope that further longitudinal study, in more diverse teams, will clarify these questions.

9.4 Future Work

There are several avenues open to further exploration, including the following:

Expand the task annotation analysis: Our analysis of task annotations found that different developers often have very different practices around how they annotate. Our work could be expanded by conducting a further archival analysis of source code comments. First, we could examine how many developers on a team commit task annotations to the code. For example, does one developer account for the majority of task annotation in a project, or is it evenly spread between team members? Second, does the life span of a task annotation change depending on the stage in application development?

Further examine the uses of task annotations: While we have begun to answer how developers use task annotations, further examination of potential use cases should be conducted. The resulting list of use cases should then be validated by soliciting feedback from professional software developers. This may lead to further implications for tool designers and the software process.

Explore collaborative tagging in software: We have wanted to study software tagging in a collaborative setting; however, it has been difficult to arrange for a development team to use TagSEA. We believe that annotation patterns and practices might be different for developers working together on the same or closely related code. In particular, we would be interested to see if there are any changes in keyword or metadata use.

Interview additional developers: One way to improve the generalizability of our results would be to interview more developers. For instance, we have considered interviewing more Eclipse developers, as well as Jazz developers [26]. To further improve upon this, we should aim to employ a purposive sampling technique [13][83], so we can make some distinction about the population we are examining. Since our research thus far has focused on open-source developers using the Eclipse IDE, we would recommend that any further interviews remain within this scope, so as not to generalize beyond our results.

Explore tours through software: We have begun to explore other uses for tagging within the domain of software engineering. This includes looking at how a series of tagged locations can form a tour or path through source code. Similar to the work by Oezbek and Lutz [62], we expect that this may have possible uses for program comprehension. The idea of tours has also been used to implement a plug-in allowing developers to use the Eclipse IDE as a presentation tool [12].

9.5 Conclusion

Software development research often focuses only on the code that developers work on. However, secondary information, such as documentation in the form of annotations, plays an important role helping developers manage complex software development tasks. Our research highlights how an improved understanding of how this information is created and used provides useful insights that inform software process and software tool design.

References

[1] Abrams, D., Baecker, R., Chignell, M. "Information Archiving with Bookmarks: Personal Web Space Construction and Organization," in *Proceedings of the Conference on Human Factors in Computing Systems,* Los Angeles, California, United States, pp. 41-48, 1998.

[2] Ames, M. and Naaman, M. "Why We Tag: Motivations for Annotation in Mobile and Online Media," in *Proceedings of the Conference on Human Factors in Computing Systems,* pp. 971-980, 2007.

[3] Bannon, L., Cypher, A., Greenspan, S., Monty, M.L. "Evaluation and Analysis of Users' Activity Organization," in *Proceedings of the Conference on Human Factors in Computing Systems,* pp. 54-57, 1983.

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

[5] Barreau, D. and Nardi, B.A. "Finding and Reminding: File Organization from the Desktop," *SIGCHI Bulletin*, 27(3): 39-43, July 1995.

[6] Belotti, V., Dalal, B., Good, N., Flynn, P., Bobrow, D.G., Ducheneaut, N. "What a To-Do: Studies of Task Management Towards the Design of a Personal Task List Manager," in *Proceedings of the Conference on Human Factors in Computing Systems,* Vienna, Austria, pp. 735-742, 2004.

[7] Bernstein, M.S, Van Kleek, M., Schraefel, M.C., Karger, D.R. "Management of Personal Information Scraps," in *Proceedings of the Conference on Human Factors in Computing Systems,* pp. 2285-2290, 2007.

[8] Brothers, L., Sembugamoorthy, V., Muller, M. "ICICLE: Groupware for Code Inspection," in *Proceedings of the Conference on Computer Supported Cooperative Work,* pp. 169-181, 1990.

[9] Brush, A.J.B., Bargeron, D., Gupta, A., Cadiz, J.J. "Robust Annotations Positioning in Digital Documents," in *Proceedings of the Conference on Human Factors in Computing Systems,* Seattle, Washington, United States, pp. 285-292, 2001.

[10] Capra, R. *An Investigation of Finding and Refinding Information on the Web.* Doctoral dissertation, Department of Computer Science, Virginia Polytechnic Institute and State University, Blacksburg, Virginia, United States, February 2006.

[11] Card, S.K. and Henderson, A. “A Multiple, Virtual-workspace Interface to Support User Task Switching,” in *Proceedings of the Conference on Human Factors in Computing Systems,* Toronto, Ontario, Canada, pp. 53-59, 1997.

[12] Cheng, L.-T., Desmond, M., Storey, M.-A. “Presentations by Programmers for Programmers,” in *Proceedings of the International Conference on Software Engineering,* Minneapolis, Minnesota, United States, pp. 788-792, 2007.

[13] Creswell, J.W. *Educational Research: Planning, Conducting, and Evaluating Quantitative and Qualitative Research, 2nd ed.* Pearson Education Inc., 2005.

[14] Creswell, J.W. *Qualitative, Quantitative, and Mixed Methods Approaches, 2nd ed.* Sage Publications, 2003.

[15] Czerwinski, M. and Horvitz, E. “An Investigation of Memory for Daily Computing Events,” in *Proceedings of the Conference on Human Factors in Computing Systems,* pp. 230-245, 2002.

[16] Czerwinski, M., Horvitz, E., Wilhite, S. “A Diary Study of Task Switching and Interruptions,” in *Proceedings of the Conference on Human Factors in Computing Systems,* pp. 175-182, 2004.

[17] den Besten, M., J.-M. Dalle, Galia, F. “Collaborative Maintenance in Large Open-Source Projects,” *IFIP International Federation for Information Processing,* Volume 203, pp. 233-244, 2006.

[18] Darken, R.P. and Peterson, B. “Spatial Orientation, Wayfinding, and Representation,” *Handbook of Virtual Environment Technology,* Lawrence Erlbaum Associates, Inc., Philadelphia, Pennsylvania, United States, 2001.

[19] Eclipse, The Eclipse Foundation, http:/ / www.eclipse.org.

[20] Ellis, J. and Kvavilashvili, L. “Prospective Memory in 2000: Past, Present, and Future Directions,” *Applied Cognitive Psychology,* 14(7): pp. S1-S9, 2000.

[21] Ellis, J.B., Wahid, S., Danis, C., Kellogg, W.A. “Task and Social Visualization in Software Development: Evaluation of a Prototype,” in *Proceedings of the Conference on Human Factors in Computing Systems,* pp. 577-586, 2007.

[22] Elsweiler, D., Ruthven, I., Jones, C. “Towards Memory Supporting Personal Information Management Tools,” *Journal of the American Society for Information Science and Technology,* 58(7): pp. 924-946, 2007.

[23] B. Fluri, M. Würsch, and 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.

[24] Forward, A and Lethbridge, T.C. “The Relevance of Software Documentation, Tools and Technologies: A Survey,” in *Proceedings of the Symposium on Document Engineering,* pp. 26-33, 2002.

[25] Forward, A., Lethbridge, T., Deugo, D. “CodeSnippets Plug-in to Eclipse: Introducing Web 2.0 Tagging to Improve Software Developer Recall,” in *Proceedings of the International Conference on Software Engineering Research, Management and Applications,* pp. 451-460, 2007.

[26] Frost, R. “Jazz and the Eclipse Way of Collaboration,” *IEEE Software,* 24(6): pp. 114-117, 2007.

[27] Furnas, G.W., Landauer, T.K., Gomez, L.M., and Dumais, S.T. “The Vocabulary Problem in Human-System Communication,” *Communications of the ACM,* 30(11): pp. 964-971, 1987.

[28] Furnas, G.W., Fake, C., von Ahn, L., Schachter, J., Golder, S., Fox, K., Davis, M., Marlow, C., Naaman, M. “Why do Tagging Systems Work?” in *Proceedings of the Conference on Human Factors in Computing Systems,* pp. 36-39, 2006.

[29] Gendarmi, D. and Lanubile, F. “Community-Driven Ontology Evolution Based on Folksonomies,” *Lecture Notes in Computer Science,* 4277, pp. 181-188, 2006.

[30] Golder, S.A. and Huberman, B.A. “The Structure of Collaborative Tagging Systems,” *Journal of Information Science,* 32(2): pp. 198-208, 2006.

[31] Gonzalez, V.M. and Mark, G. “'Constant, Constant, Multi-tasking Craziness': Managing Multiple Working Spheres,” in *Proceedings of the Conference on Human Factors in Computing Systems,* Vienna, Austria, pp. 113-120, 2004.

[32] Greene, J.C., Caracelli, V.J., Graham, W.F. “Towards a Conceptual Framework for Mixed-Method Evaluation Designs,” *Educational Evaluation and Policy Analysis,* 11(3): pp. 255-274, 1989.

[33] Grogono, P. “Comments, Assertions and Pragmas,” *SIGPLAN Notices,* 24(3): pp. 79-84, 1989.

[34] Guy, M. and Tonkin, E. “Folksonomies: Tidying up Tags?” *DLib Magazine,* 12(1), January 2006.

[35] Halverson, C.A., Ellis, J.B., Danis, C., Kellogg, W.A. "Designing Task Visualizations to Support the Coordination of Work in Software Development," in *Proceedings of the Conference on Computer Supported Cooperative Work*, Banff, Alberta, Canada, pp. 39-48, 2006.

[36] Hammond, T., Hannay, T., Lund, B., Scott, J. "Social Bookmarking Tools (I): A General Review," *D-Lib Magazine,* 11(4), April 2005.

[37] Hove, S.E. and Anda, B. "Experiences from Conducting Semi-structured Interviews in Empirical Software Engineering Research," in *Proceedings of the International Software Metrics Symposium,* 2005.

[38] Iqbal, S.T. and Horvitz, E. "Disruption and Recovery of Computing Tasks: Field Study, Analysis, and Directions," in *Proceedings of the Conference on Human Factors in Computing Systems*, pp. 677-686, 2007.

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

[40] Johnson, R.B. and Onwuegbuzie, A.J. "Mixed Methods Research: A Research Paradigm Whose Time Has Come," *Educational Researcher,* 33(7): pp. 14-26, 2004.

[41] Jones, W., Bruce, H., Dumais, S. "Keeping Found Things Found on the Web," in *Proceedings of Conference on Information and Knowledge Management*, pp. 119-126, 2001.

[42] Kaelbling, M.J. "Programming Languages Should NOT have Comment Statements," *SIGPLAN Notices,* 23 (10): pp. 59-60, 1988.

[43] Kersten, M. and Murphy, G. "Mylar: A Degree-of-Interest Model for IDEs," in *Proceedings of Aspect Oriented Software Development*, pp. 159-168, 2005.

[44] Knuth, D. "Literate Programming," *The Computer Journal,* 27(2): pp. 97-111, 1984.

[45] Ko, A.J., Aung, H.H., Myers, B.A. "Design Requirements for More Flexible Structured Editors from a Study of Programmers' Text Editing," in *Proceedings of the Conference on Human Factors in Computing Systems,* pp. 1557-1560, 2005.

[46] Kvavilashvivi, L. "Remembering Intentions: Testing a New Method of Investigation," *Applied Cognitive Psychology,* 12(6): pp. 533-554, 1998.

[47] Lansdale, M. "The Psychology of Personal Information Management," *Applied Ergonomics,* 19(1): pp. 55-66, 1988.

[48] LaToza, T.D, Venolia, G., DeLine, R. "Maintaining Mental Models: A Survey of Developer Work Habits," in *Proceedings of the International Conference on Software Engineering,* pp. 492-501, 2006.

[49] Leech, N. and Onwuegbuzie, A.J. "A Typology of Mixed Methods Research Designs," *Quality and Quantity,* ISSN 0033-5177.

[50] Lethbridge, T.C., Sim, S.E., Singer, J. "Studying Software Engineers: Data Collection Techniques for Software Field Studies," *Empirical Software Engineering,* 10(3): pp. 311-341, 2005.

[51] McDaniel, M.A. and Einstein, G.O. "Strategic and Automatic Processes in Prospective Memory Retrieval: A Multiprocess Framework," *Applied Cognitive Psychology,* 14(7): pp. S127-S144, 2000.

[52] Malone, T.W. "How do People Organize their Desks? Implications for the Design of Office Information Systems," *ACM Transactions on Office Information Systems,* 1(1): pp. 99-112, 1983.

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

[54] Mark, G., Gonzalez, V.M., Harris, J. "No Task Left Behind? Examining the Nature of Fragmented Work," in *Proceedings of the Conference on Human Factors in Computing Systems,* pp. 321-330, 2005.

[55] Marshall, C. "Annotation: From Paper Books to the Digital Library," In *Proceedings of the International Conference on Digital Libraries,* pp. 131-140, 1997.

[56] Marshall, C. and Brush, A.J.B. "Exploring the Relationship between Personal and Public Annotations," in *Proceedings of the International Conference on Digital Libraries,* pp. 349-357, 2004.

[57] Marshall, C., Price, M.N., Golovchinsky, G., Schilit, B.N. "Introducing a Digital Library Reading Appliance into a Reading Group," in *Proceedings of the International Conference on Digital Libraries,* pp. 77-84, 1999.

[58] Millen, D.R., Feinberg, J., Kerr, B. "DogEar: Social Bookmarking in the Enterprise," in *Proceedings of the Conference on Human Factors in Computer Systems,* pp. 111-120, 2006.

[59] Mockus, A. and Votta, L.G., "Identifying Reasons for Software Changes Using Historic Databases," in *Proceedings of the International Conference on Software Maintenance,* pp. 120-130, 2000.

[60] Muller, M.J. "Comparing Tagging Vocabularies among Four Enterprise Tag-Based Services," in *Proceedings of the International Conference on Supporting Group Work,* pp. 341-350, 2007.

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

[62] Oezbek, C. and Lutz, P. "JTourBus: Simplifying Program Understanding by Documentation that Provides Tours Through the Source Code," in *Proceedings of the International Conference on Software Maintenance,* pp. 64-73, 2007.

[63] Planet Eclipse, http:/ / planeteclipse.org/ planet/ .

[64] Raubal, M. and Winter, S. "Enriching Wayfinding Instructions with Local Landmarks," *Lecture Notes in Computer Science,* 2478, pp. 243-259, 2002.

[65] Ravasio, P., Schär, S.G., Krueger, H. "In Pursuit of Desktop Evolution: User Problems and Practices with Modern Desktop Systems," *ACM Transactions on Computer-Human Interaction,* 11(2): pp. 156-180, 2004.

[66] Sachs, J. "Some Comments on Comments," *SIGDOC Asterisk Journal of Computer Documentation,* 3(7): pp. 7-14, 1976.

[67] Shirky, C. "Ontology is Overrated: Categories, Links, and Tags," http:/ / www.shirky.com/ writings/ ontology_overrated.html, last accessed 18 March 2008.

[68] Sen, S., Lam, S.K., Rashid, A.M., Cosley, D., Frankowski, D., Osterhouse, J., Harper, F.M., Riedl, J. "tagging, communities, vocabulary, evolution," in *Proceedings of the Conference on Computer Supported Collaborative Work,* pp. 181-190, 2006.

[69] Storey, M.-A., Cheng, L.-T., Singer, J., Muller, M., Myers, D., Ryall, J., "How Programmers can Turn Comments into Waypoints for Code Navigation", in *Proceedings of the 23rd IEEE International Conference on Software Maintenance,* pp. 265-274, 2007.

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

[71] Storey, M.-A., Bull, I., Rigby, P., Cheng, L.-T. "Waypointing and Social Tagging to Support Program Navigation," in *Proceedings of the Conference on Human Factors in Computer Systems*, pp. 1367-1372, 2006.

[72] Storey, M.-A., Ryall, J., Bull, R.I., Myers, D., Singer, J. "TODO or To Bug: Exploring how Task Annotations play a Role in the Work Practices of Software Developers," in *Proceedings of the International Conference on Software Engineering*, 2008.

[73] Sun Microsystems. "Java Annotations," http:/ / java.sun.com/ j2se/ 1.5.0/ docs/ guide/ language/ annotations.html, 2004.

[74] Sun Microsystems. "Javadoc 5.0 Tool," http:/ / java.sun.com/ j2se/ 1.5.0/ docs/ guide/ javadoc/ index.html, 2004.

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

[76] Tashakkori, A. and Teddlie, C. *Handbook of Mixed Methods in Social & Behavioural Research*, Sage Publications, 2003.

[77] Teevan, J. *Supporting Finding and Re-Finding through Personalization*. Doctoral dissertation, Massachusetts Institute of Technology, Boston, Massachusetts, United States, 2007.

[78] Van De Vanter, M.L. "The Documentary Structure of Source Code," *Information and Software Technology*, 44(13): pp. 767-782, 2002.

[79] Vakarri, P. "Task Complexity, Problem Structure and Information Actions: Integrating Studies on Information Seeking and Retrieval," *Information Processing & Management*, 35(6): pp. 819-837, 1999.

[80] Voss, J. "Collaborative Thesaurus Tagging the Wikipedia Way," http:/ / arxiv.org/ abs/ cs/ 0604036, 27 April 2006.

[81] Whittaker, S. and Hirschberg, J. "The Character, Value, and Management of Personal Paper Archives," *ACM Transactions on Computer-Human Interaction*, 8(2): pp. 150-170, 2001.

[82] Whittaker, S. and Sidner, C. "Email Overload: Exploring Personal Information Management of Email," in *Proceedings of the Conference on Human Factors in Computing Systems*, Vancouver, British Columbia, Canada, pp. 276-283, 1996.

[83] Yin, Robert K. *Case Study Research: Design and Methods, 3rd ed.* Sage Publications, 2002.

[84] Ying A., Wright, J., and Abrams, S. "Source Code that Talks: An Exploration of Eclipse Task Comments and their Implication to Repository Mining", in *Proceedings of the Workshop on Mining Software Repositories*, pp. 1-5, 2005.

[85] Ying A., Wright, J., and Abrams, S. "An Exploration of How Comments are Used for Marking Related Code Fragments", in *Proceedings of the Workshop on Modeling and Analysis of Concerns in Software*, pp. 1-4, 2005.

Appendix A:
Summary of Individual Contributions

As mentioned earlier, this is a collaborative project that has benefited from the contributions of members of the TagSEA research team. This appendix aims to clarify my individual contributions to this research. I have organized the contributions based on four parts of the thesis: background, research design, studies and results, and findings and conclusions.

Background (Chapters 1-4)

- I conducted the multidisciplinary literature review.
- Janice Singer and Margaret-Anne Storey made contributions to the literature review of software tools and the use of annotations in source code.
- Del Myers, as lead developer of the TagSEA tool, contributed significantly to the description of the tool presented in this thesis. Li-Te Cheng, Michael Desmond, and other members of the research group also contributed to the design and development of the TagSEA tool.

Overall Research Design (Chapter 5)

- The research methodology was chosen collaboratively between Margaret-Anne Storey and me.
- The research questions in this thesis differ from those posed while the studies were being conducted. Those questions were derived collaboratively, while the ones in this thesis were chosen by me, in consultation with members of the research group.

Studies and Results (Chapters 6-7)

- Survey of Software Developers

    - For the questionnaire, I designed the initial questions and sought feedback from the team, created the questionnaire, and analyzed the results.
- Eclipse Task Annotation Study
    - I designed the overall strategy for this study.
    - I conducted the initial extraction of task annotations from the ten projects. The numbers were confirmed by Del Myers.
    - For the interviews, I developed questions with Margaret-Anne Storey. The interviews were conducted by Ian Bull. Del Myers and I created the interview transcripts. Janice Singer, Margaret-Anne Storey and I analyzed the transcripts to uncover themes.
- Industry Case Study
    - This study was designed by Margaret-Anne Storey.
    - The study was run by Li-Te Cheng at IBM, who was on site in Cambridge, Massachusetts.
    - Data collection and extraction was led by Del Myers.
    - I led the development and analysis of the pre- and post-study questionnaires.
    - As part of the data analysis, Janice Singer, Margaret-Anne Storey and I coded the annotations to develop our tag taxonomy.
- Longitudinal Case Study
    - This study was designed by me.
    - I developed the questions and interviewed the developers
    - Janice Singer and I coded the annotations and refined the tag taxonomy.
    - Data extraction and quantitative analysis were performed by Del Myers and Ian Bull, with feedback from Margaret-Anne Storey and me.

Findings and Conclusions (Chapters 8-9)

- Ian Bull prepared the initial list of implications. The rest of the group and I expanded and refined this list.
- The findings from the studies were produced collaboratively, arising from the quantitative results and the qualitative themes that emerged during the studies. I contributed to the development of these findings, and in this thesis, I brought together the results and synthesized them as they relate to the questions posed in this thesis.

Appendix B:

List of Questions from the Questionnaire

1. How many years have you been programming professionally?

2. What kinds of software projects do you work on?
   a. Open source
   b. Proprietary
   c. Both

3. Are you currently working on a team?
   a. No
   b. Yes

4. Based on your most recent team experience, how many programmers work in your team (including yourself)?

5. Based on your most recent team experience, how are your team members distributed?
   a. N/ A – I am working on my own
   b. Everyone is in the same hallway or office
   c. Everyone is in the same building
   d. The team is distributed across two buildings

6. When you are collaborating on a team has your team agreed to use the same keywords?
   a. I use my own keywords
   b. I use a mixture of my own keywords and my team's keywords
   c. My team has an informal agreement to use the same keywords
   d. My team has a formal commitment (or coding practices) to use the same keywords

7. What common development tools and coding conventions do you use in your team? (Select all that apply)
   a. N/ A – Team members use their own tools and conventions
   b. Team members use the same source code management (SCM) system
   c. Team members use the same bug tracking system
   d. Team members use a common wiki to document their code

e. Team members use a mailing list or chat to announce changes and new pieces of code to keep track of
f. Team members enter bug # information in source code management (SCM) check-in comments
g. Team members use javadoc to document their code
h. Team members use Mylar's shared task contexts
i. Team members use a common class and variable naming scheme
j. Team members follow a common scheme for folder hierarchies and folder names
k. Team members use common keywords in their comments (e.g. TODO, FIXME, XXX)
l. Other (please specify)

8. Do you use Eclipse bookmarks in your source code?

a. Never
b. Rarely
c. Sometimes
d. Often

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

a. FIXME
b. HACK
c. TODO
d. XXX
e. Other (please specify)

10. Do you comment your code?

a. No
b. Yes

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

a. My name or initials
b. Name or initials of other people
c. Name of my team
d. Name of another team
e. Reference to another class, method, plugin, or module
f. URL to a related web page (e.g. article, wiki page, specification, etc)
g. Bug id

h. Date
i. Memorable keywords
j. None of the above (I do not add additional details)
k. Other (please specify)

12. If you add memorable keywords, please give examples, along with their definitions. (Please write as long an answer as you think appropriate. Each text field will expand to accommodate your response.)

13. Which of the following tool features do you use to navigate to source code comments? (Select all that apply)

a. Browse to find them
b. Hyperlinks
c. Search for keywords
d. Tasks/ Bookmarks view
e. Ctrl-Shift-T (Open Type) and then browse/ search for comment
f. Use an external email or personal "todo.txt"-like file to find the file and then browse/ search for comment
g. None of the above (I do not navigate source code comments)
h. Other (please specify)

14. When you scan through other people's code, what features of their comments would attract your attention and make you stop and read more detail? (Select all that apply)

a. My name or initials
b. Names or initials of other people
c. Id of a bug I am currently, or had worked on
d. Id of a bug relevant to my team
e. Id of any bug
f. Date
g. Memorable keywords
h. Comment description
i. Reference to another class, method, plugin, or module
j. URL to a related web page (e.g. article, wiki page, specification, etc)
k. None of the above
l. Other (please specify)

15. Other feedback on commenting style or tools?