Maintaining Mental Models: A Study of Developer Work Habits

Thomas D. LaToza
Institute for Software Research International
Carnegie Mellon University
5000 Forbes Avenue
Pittsburgh, PA 15213 USA
tlatoza@cs.cmu.edu

Gina Venolia
Microsoft Research
One Microsoft Way
Redmond, WA 98052 USA
gina.venolia@microsoft.com

Robert DeLine
Microsoft Research
One Microsoft Way
Redmond, WA 98052 USA
rob.deline@microsoft.com

ABSTRACT

To understand developers' typical tools, activities, and practices and their satisfaction with each, we conducted two surveys and eleven interviews. We found that many problems arose because developers were forced to invest great effort recovering implicit knowledge by exploring code and interrupting teammates and this knowledge was only saved in their memory. Contrary to expectations that email and IM prevent expensive task switches caused by face-to-face interruptions, we found that face-to-face communication enjoys many advantages. Contrary to expectations that documentation makes understanding design rationale easy, we found that current design documents are inadequate. Contrary to expectations that code duplication involves the copy and paste of code snippets, developers reported several types of duplication. We use data to characterize these and other problems and draw implications for the design of tools for their solution.

Categories and Subject Descriptors

D2.7 [**Distribution, Maintenance, and Enhancement**]: Documentation; D2.9 [**Management**]: Programming teams; D.2.6 [**Programming Environments**]: Integrated environments.

General Terms

Design, Documentation, Experimentation, Human Factors

Keywords

Code duplication, communication, interruptions, code ownership, debugging, agile software development

1. INTRODUCTION

Developers must know or obtain a variety of information to successfully understand and edit code – what code to change, how design decisions are scattered across code [4], the rationale or history behind decisions [7], the slice affecting a variable's value [13], the owner responsible for the code [1], other developers currently editing it, which changes will break code elsewhere, and which changes elsewhere affect it. Developers choose among many strategies to record, communicate, and discover this information. Naming, comments, and design documents allow developers to share their current understanding with future developers, but require an investment of time and knowledge about what future developers will need to learn. Conventions, factoring, and patterns minimize documentation burdens by providing general answers but constrain possible solutions and themselves become more to learn. For many types of information, the simplest solution is frequently to ask a teammate for the answer [2], yet the teammate is interrupted, must change tasks, and forgets goals, decisions, and interpretations relevant to the interrupted task. Modern development environments compute facts from code (e.g. callers of a method, writers to a field, methods overriding a method, average execution time) or other artifacts and require neither interruption nor investment in error prone documentation maintenance, but require a tool vendor or researcher to have anticipated the developer's situation and needs. And computing many types of information may require the developer's assistance.

We performed a series of investigations of developers. The central theme that emerged was the developers' reliance on implicit code knowledge. Developers go to great lengths to create and maintain a mental model of the code, and knowledge is shared between developers through face-to-face communication and the code itself. Developers avoid using explicit, written repositories of code-related knowledge in design documents or email when possible, preferring to explore the code directly and, when that fails, talk with their teammates. Exploring code is made difficult by tool limitations and difficulties traversing relationships. Using the social network as the second line of inquiry causes interruptions and lost work, but those costs are offset by other benefits. Implicit knowledge retention is made possible by a strong, yet often implicit, sense of code ownership, the practice of a developer or team being responsible for fixing bugs and writing new features in a well defined section of code. This increases the payoff from the large investment understanding code. Implicit knowledge retention makes some information difficult to uncover, particularly code duplication. Yet developers view it far more broadly than the clone detection literature.

We used both qualitative data from interviews and quantitative data from two surveys in our investigation. While the breadth of our exploratory approach precludes the detail necessary to fully understand each topic, and we were often left with more questions than answers, we highlight interesting observations and propose promising directions for future investigation.

We first present a taxonomy of developer activity which guided our investigation. We then describe the study design and organizational characteristics of the participants. In the first results section, we describe developers' time usage, tool usage, and tool preferences. In the second, we discuss the nature,

ICSE'06, May 20–28, 2006, Shanghai, China.

Table 1. Descriptions of activities read by respondents. Descriptions ending in […] have been shortened.

Designing	Analyzing a new problem and mapping out the broad flow of code which will be used to solve the problem. […]
Writing	Creating a new method, source file, or script and getting it to a compilable state
Understanding	Determining information about code including the inputs and outputs to a method, what the call stack looks like, why the code is doing what it is doing, or the rationale behind a design decision. […]
Editing	Editing existing code and returning it to a compilable state.
Unit testing	Ensuring that code is behaving as expected. […]
Communicating	Any computer mediated or face-to-face communication about information relevant to a coding task […]
Overhead	Any other code related activities including building, synchronizing code, or checking in changes.
Other code	[No description provided]
Non code	Any other activities included in your work time

motivations, and problems with developers' reliance on implicit code knowledge. Finally, we present design recommendations for tools and conclude.

2. ACTIVITY TAXONOMY

We began our investigation by characterizing developers' interactions with code – their activities, tools, and biggest problems. Rather than bringing a preconceived area of focus, we wished to be more opportunistic and let our users – the developers – guide us in selecting what they perceived as most painful through reports of their time, tool effectiveness perceptions, and problems.

Two previous studies have categorized developers' activities in the field through diaries, observation, and surveys. In the first study [9], thirteen developers on a large software project logged every hour for a year which of 13 activities they were engaged in. The categories distinguished different life cycle activities such as estimation, requirements, high level & low level design, test planning, coding, inspections, and high level & low level testing. Most developers reported being in a coding stage. Despite waterfall or spiral models predicting developers spend their time coding in a coding stage, developers reported coding for only half of their time with the rest spent on activities associated with other life cycle stages.

In the second study [12], developers were surveyed, observed, and interviewed to count the number of times they switched between one of thirteen activities. Observations of eight developers for an hour each revealed that they most frequently executed UNIX commands, followed by reading the source, loading or running software, and reading or editing notes. Yet is not clear how activity switches translate to time spent on activities as activities may be frequent and brief or long and infrequent.

We designed our own taxonomy (see Table 1) to focus more specifically on code related activities and the motivation behind these activities. We wished to know specifically for what types of activities developers used development environments and which activities environments were poorest in supporting. We also wanted to know whether developers chose different tools for different activities.

From the our own personal experience as software developers, hypotheses about what developers might find difficult, and topics of ongoing research, we also formulated nineteen hypothesized problems developers might have in obtaining or communicating about code-related knowledge.

3. METHOD

The study consisted of three parts: a survey about activities, tools, and problems (the "activities survey"), a series of semi-structured interviews, and a survey of work practices (the "follow-up survey").

3.1 Organization

The population we selected for study was software developers at Microsoft Corporation. Microsoft is a large software company whose products span a wide range of markets: web portals (MSN); consumer devices (Windows Mobile, Xbox); office productivity applications (Windows, Office); and developer tools and infrastructure (Visual Studio, Great Plains, SQL Server). Of the roughly 63,000 employees, roughly 6,000 are software developers who work on shipping code in product groups. Other developers include those who work on test infrastructure and tools and those in Microsoft Consulting and Microsoft Research. These groups were excluded from our study.

Within a product group, there are three core roles – software design engineer (SDE), program manager, and software test engineer. SDEs are responsible for software design, fixing bugs, and writing new features. Program managers are responsible for specifying and prioritizing features and for writing high level feature specification documents which developers use to write code and testers use to write test cases. Software test engineers translate feature specifications into test cases and manually test the software. A somewhat less common role is software design engineers in test (SDE/T) who write test automation infrastructure. Members of each of these roles work in small teams of "individual contributors." Individual contributors are managed by a lead (e.g. lead software design engineer) who reports to a manager (e.g. software design engineer manager). Other less frequent roles include software architect, product designer, and usability engineer. Nearly all individual contributors have private offices (not cubes) and most do not share an office.

Product group work for a particular release of a product is divided into milestones. In the first milestone, program managers make initial decisions about what features will be in the release, what features developers will work on in subsequent milestones, and write initial feature specification documents. SDEs may work on

bug fixes and patches from the previous release, try out new technologies, or plan major changes. Several milestones of development follow. Each milestone is divided more or less into a coding phase, where features are added, and a stabilization phase, where developers concentrate on fixing bugs. During the last milestone, most of the work involves fixing bugs. As the release nears, most changes become too time consuming and risky to test, and developers spend more time making the next version's code more maintainable (see Figure 1).

3.2 Procedure

Activities survey participants first completed a number of demographic items. They next read the activity descriptions found in Table 1. They were then asked to report the fraction of their past week work time spent on each activity, choosing among 10% increments plus choices for 1%, 2%, and 5%. For each activity, they were asked the percent of time on that activity they used each of a set of tools or techniques, using the same scale. For each tool/technique combination and activity, developers were asked to rate its effectiveness on a seven-point Likert scale. Finally, they rated the seriousness of each hypothesized problem using a seven-point Likert scale. There were 204 questions in all.

While we expect respondents misremembered, misestimated, and misreported the time fractions, we expect they were able to differentiate across large distinctions like values near 0% and 5% or values near 10% and 40%. We normalized each group of fractional responses to sum to 100%.

Before deploying the activities survey we used two techniques to ensure that its design fit the activities, tools, techniques, and problems relevant to our target population. First we ran three experienced developers through the survey using a think-out-loud protocol. We adapted the survey wording and structure based on their feedback. Second, we developed a reduced version of the survey that included extensive opportunities for participants to write in additional activities, tools, and problems. We deployed this pilot survey to 99 randomly selected developers and received 28 responses. Any write-in response from two or more respondents was included in the activities survey. No activities or problems met this criterion, but a few tools did.

We selected the four problems rated as the most serious on average from the activities survey (see Table 2) and designed a series of interview questions to elicit qualitative information about the character and impact of these problems. We added several general, open ended questions on how the participants characterized their work and activities and on team communication patterns. Two authors attended each interview. Ten of the eleven interviewees consented to having the conversation audio-recorded. All three authors used their notes or recordings to generate nearly 1,000 note cards of observations. The cards were then used for a card sort [14] where they were placed on the walls of a ~30 foot hallway to form groups, elicit themes and trends, and consolidate observations across interviwers and interviewees.

From the card sort we identified several preliminary hypotheses. We developed a follow-up survey to assess the hypotheses amenable to surveying. Participants first answered demographic questions. Next they answered questions about the size of their feature team, which was defined as, "the core group of developers that you work with." They then answered a series of questions about communication patterns, code ownership, design documents, understanding unfamiliar code, code duplication, unit testing, and adoption of agile practices. There were 187 questions in all.

3.3 Participants

We drew our participants from the population that deals directly with code: SDEs, SDE/Ts, and architects at both the individual contributor and lead level. After the activities survey we decided to focus on developers working on *shipping* code, and so removed the responses from architects and SDE/Ts from our analysis and the subsequent observations. We felt that our survey questions were most informative about SDEs, and we lacked resources to investigate all three roles. We excluded contractors because of logistical problems and excluded interns because we wished to generalize to professional software developers.

Participants were invited to participate in the surveys by email and sent a reminder email several days before the surveys were closed if they had not yet responded. Respondents were compensated by entry in a drawing for $50 gift certificates. In the activities survey, we randomly sampled 1,000 participants from the participant pool, excluding those invited to take the pilot survey. We received 157 responses, 104 from SDEs, including 18 from lead SDEs. We were somewhat disappointed with the response rate and attribute it to the survey being deployed in early July when many were on vacation, some technical problems with the survey deployment, and sheer size of the survey. In the follow-up survey, we randomly sampled 1,000 from the same pool excluding SDETs and recipients of the activities and pilot surveys. We received 187 responses, 176 from SDEs. For both surveys, we did not measure self selection bias to ensure our sample was truly representative.

The activities survey contained several demographic questions. Since participants from all surveys were randomly sampled from the same population of SDEs, and we expect any self selection bias to apply equally to both surveys, these demographics apply to all study participants. The average respondent is in their 30's with an undergrad degree, 12.1 (± 6.5) years programming, 5.8 (± 4.2) years at Microsoft, and 2.9 (± 2.4) years on their current team; 89% of respondents are male. 37% reported that most of their code base was written in C#, compared to 56% in C or C++, reflecting both older, established code bases and newer code bases written in C#.

We interviewed eleven respondents, five SDEs from the pilot survey and 6 lead SDEs from the activities survey.

4. ACTIVITIES AND TOOLS

Far from spending all of their time understanding or editing existing code, developers reported spending most of their time elsewhere. Developers' tool use was frequently correlated with their tool preferences. This is clearly visible in the positive linear relationship of tool usage to effectiveness (Figure 5). As the study was exploratory rather than being hypothesis driven, results are presented with descriptive statistics. Times are reported using the mean (± standard deviation).

4.1 Time breakdown

Developers reported spending a little less than half of their time (49% ± 39%) fixing bugs, 36% (± 37%) writing new features, and the rest (15% ± 21%) making code more maintainable. This confirmed our expectation that most developers spend much of

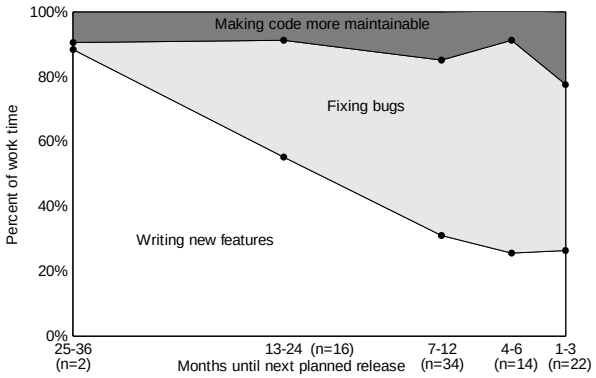

Figure 1. The time spent fixing bugs, making code more maintainable, and writing new features varies with the time until the product is planned to be released.

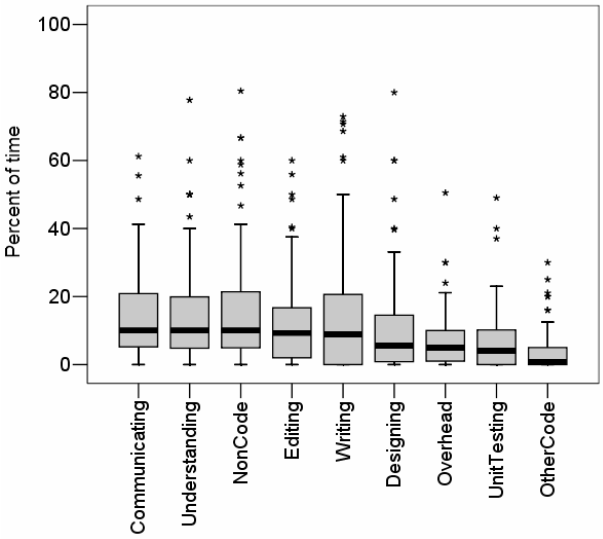

Figure 2. A box plot of activity time. The box bottom, internal line, and top are the first, second, and third quartiles. The exterior lines extend for 1.5 times the interquartile region, with outliers displayed above.

their time fixing bugs. But the vast variability in these numbers also demonstrates that typical development activity varies greatly across teams and across the lifecycle (Figure 1).

Median times spent on each activity (Table 1) are remarkably close (Figure 2), dashing hopes that a single activity accounts for most of developers' time. Most developers engage in multiple activities in a given week (Figure 4). However, most activities still had individual developers who spent most of their week on that activity.

Pairwise correlations of activities (Figure 3) reveal several statistically significant, if not large, activity relationships. Designing code and writing new code are positively correlated. Editing code goes hand-in-hand with overhead tasks like building and source code management. Understanding existing code is negatively correlated with designing code and writing new code, suggesting that one is either working on new code or examining existing code, but not both at the same time. Designing and

Figure 3. Statistically significant correlations between time spent on each activity. Negative numbers indicate inverse relationships. (Spearman's rho, thin lines for $p<0.05$, thick lines for $p<0.01$, $n=104$.)

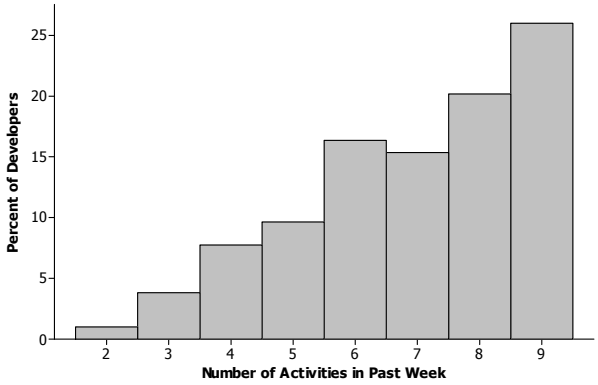

Figure 4. Most developers engage in a number of activities in a given week.

writing are negatively correlated with non-code activities, suggesting that working on new code is an all-consuming activity. The negative correlation between writing new code and communicating about code suggests developers working on new functions or classes need less information from their teammates. Unit testing was the only activity for which we found no correlation to other activities. It is worth noting that analyzing only pairwise correlations neglects any relationships involving multiple activities.

4.2 Communicating

Developers both preferred and spent more time using face-to-face communication than electronic communication (Figure 5a), replicating a 1994 finding [9] of a strong preference for face-to-face over email. Yet, email has since increased in prominence and sophistication and instant messaging has made possible short response time, interactive communication. Developers gave a number of reasons for preferring face-to-face communication. Developers reported that email questions often took hours or days to receive a response, that developers frequently misinterpreted emails' meanings, writing an email without immediate feedback

often resulted in explanations with more or less detail than the recipient required, and that email was just tedious to write. We believe many of these problems generalize to other electronic communication such as documentation, bug databases, and IM. Developers still use email when the issue is of low priority, involves multiple people, or involves non-teammates, averaging 16.1 (± 14.5) emails sent to teammates in the prior week and 5.9 (± 11.5) to non-teammates. The preference for face to face communication over email might limit benefits from systems helping developers locate old emails, and the barriers discouraging email use might make it difficult to encourage more retention of knowledge in emails. Unplanned, face-to-face meetings happen frequently with teammates, averaging 8.4 (± 11.7) per week, and much less frequently with non-teammates, averaging 2.6 (± 4.0). Communication within the team is much more common than communication across teams, indicating that the culture of informal communication works well and that the team boundaries are typically in the right places.

Most developers reported using IM only infrequently for code related tasks. It was more frequently used to contact teammates for social functions (e.g. going to lunch) or to talk to family. Use of the telephone for code-related communication was similarly rare.

4.3 Designing

Despite the availability of high-level views of code and visual editors such as tools for UML, developers remain focused on the code itself. Developers reported using a source code editor the most for design while paper and whiteboards were perceived most effective (Figure 5b). We hypothesize that the need to find details about the existing design by using a source code editor discourages increased use of paper or whiteboards, even though both were viewed as more effective tools.

4.4 Perceived problems

Table 2 lists the problems we proposed in the survey and the percent of respondents who agreed that the problem is a "serious problem for me." The top four are: understanding the rationale behind existing code, having to switch tasks because of manager or teammate requests, being aware of changes elsewhere, and finding code duplicates. We focused our semi-structured interviews on these problems to discern what makes them difficult. Several themes emerged:

- Developers go to great lengths to create and maintain rich mental models of code that are rarely permanently recorded.
- Understanding the rationale behind code is the biggest problem for developers. When trying to understand a piece of code, developers turn first to the code itself and, when that fails, to their social network.
- Developers and development managers use a variety of tools and work practices and are actively looking for better solutions.

We present these themes with support from our follow-up survey.

5. MAINTAINING MENTAL MODELS

Developers create and maintain intricate mental modes of the code. Through our interviews, we know that developers, without referencing written material, can talk in detail about their product's architecture, how the architecture is implemented, who owns what parts, the history of the code, to-dos, wish-lists, and meta-information about the code. For the most part this

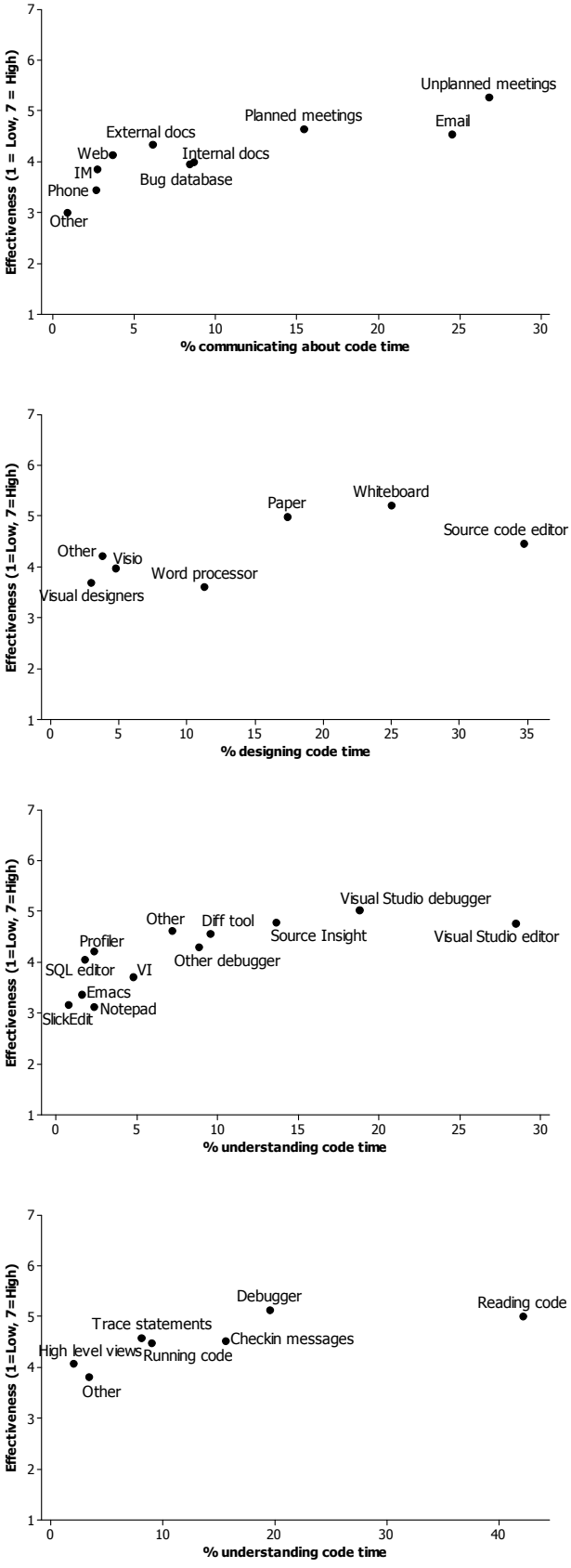

Figure 5a-d. (See text.)

Table 2. Developer ratings of proposed problems. In the survey, problems were presented without headings and in a different order.

This is a serious problem for me	% agree
Code Understanding	
Understanding the rationale behind a piece of code	66%
Understanding code that someone else wrote	56%
Understanding the history of a piece of code	51%
Understanding code that I wrote a while ago	17%
Task Switching	
Having to switch tasks often because of requests from my teammates or manager	62%
Having to switch tasks because my current task gets blocked	50%
Modularity	
Being aware of changes to code elsewhere that impact my code	61%
Understanding the impact of changes I make on code elsewhere	55%
Links between Artifacts	
Finding all the places code has been duplicated	59%
Understanding who "owns" a piece of code	50%
Finding the bugs related to a piece of code	41%
Finding code related to a bug	28%
Finding out who is currently modifying a piece of code	16%
Team	
Convincing managers that I should spend time rearchitecting, refactoring, or rewriting code	43%
Convincing developers on other teams within Microsoft to make changes to code I depend on	42%
Getting enough time with senior developers more knowledgeable about parts of code I'm working on	34%
Expertise Finding	
Finding the right person to talk to about a piece of code	39%
Finding the right person to talk to about a bug	38%
Finding the right person to review a change before check-in	19%

knowledge is never written down, except in transient forms such as sketches on a whiteboard. One interviewee summed it up well - "Lots of design information is kept in peoples' heads."

5.1 Personal Code Ownership

Mental models are expensive to create and maintain. Developers have a strong notion of *personal code ownership*, which constrains the amount of code they have to understand in detail. In our follow-up survey, 77% of respondents agreed[1] with the statement, "There is a clear distinction between code that I own and the code owned by my teammates." On the other hand some teams have a policy to avoid personal code ownership because it makes individuals too indispensable and promotes, in the words of one of our interviewees, "too much passion around the code." Code ownership is a long-term proposition, reducing the number of times that a developer has to learn a new code base. In the activities survey, the average time on the current code base was 2.6 years, with 32% reporting 6 years or more. Personal code ownership is usually tacit, i.e. part of the mental model. Written records of ownership, when present, are often out-of-date and distrusted.

We received conflicting information about design documents for issues within a team. Design documents are usually written by a developer immediately prior to implementing a larger change that affects other developers to solicit other developers' input on important decisions. In the interviews, design documents were described almost as write-only media, serving to structure the developer's thinking and as an artifact to design-review, but seldom read later and almost never kept up-to-date. On the other hand our follow-up survey respondents reported a different picture of design documents for issues within the team: their feature teams wrote an average of 7.6 (± 10.2) documents in the prior year, and kept 51% of them up-to-date. We were surprised with these numbers and can't reconcile them with the results of the interviews.

5.2 Team Code Ownership and the "Moat"

Even stronger than personal code ownership is a notion of *team code ownership*. An overwhelming 92% agreed with the statement "There is a clear distinction between the code my feature team owns and the code owned by other teams." Feature teams are small. 93% stated that their feature team consisted of 2-4 people (including the respondent). There seems to be a sweet spot at three-person feature teams, reported by 49%. Feature teams are almost always collocated, facilitating informal knowledge sharing.

One of the ways developers maintain their mental model of their team's code is by subscribing to check-in messages by email, though several interviewees expressed dissatisfaction with the lack of detail provided by teammates.

Small feature teams' strong code ownership forms a kind of *moat*, isolating them from outside perturbations. The moat is defined, in part, by design documents, which specify the interface across the moat. Design documents related to cross-team issues were less common than those relevant to issues within the team. Although the average number of design documents written in the last year for cross-team issues was 4.5 (± 7.8), significantly less than the 7.6 (± 10.2) for within-team issues (two-tailed *t*-test, $p<0.01$, $t=4.78$), cross-team design documents are significantly more likely to be kept up-to-date (61% versus 51%, two-tailed *t*-test, $p<0.01$, $t=-3.58$). The greater care taken with cross-team design documents reflects their important role in defining the moat.

Unit tests, used by 79% of our respondents, are an important part of the development process for many reasons. One surprising

[1] Throughout this paper, the word *agree* means that the participant selected either "Somewhat agree", "Agree", or "Strongly agree" from a seven-point Likert scale.

Table 3. Forms of code duplication reported by interviewees with frequency and importance from follow-up survey respondents.

	Repeated work	Example	Scattering	Fork	Branch	Language
Creation	Separate developers implement same functionality	Copy and paste of example code	Design decision distributed over multiple methods	Copy of other team's code base	Branch maintained separately	Reimplementation by same developer in different language
Aware when created	No	Yes	Yes	Yes	Yes	Yes
Refactoring challenge	Awareness at creation; different design decisions	Investment creating abstraction	Changing architecture	Convincing other team to make changes	Combining released branches	Changing architecture or implementation language
Size of clone	Members, classes	Members, classes	Members, classes	Many classes, code base	Code base	Members, classes
Repeated change	24%	44%	29%	13%	25%	29%
Refactoring	19%	39%	14%	5%	6%	15%
Agree problem	42%	41%	37%	29%	28%	29%

function is to defend the moat from outside perturbations – 54% of respondents agreed that an important benefit of unit testing is that "they isolate dependencies between teams."

Almost all teams have a *team historian* who is the go-to person for questions about the code. Often this person is the developer lead and has been with the code base the longest.

5.3 New team members

Creating a mental model from scratch requires a lot of energy for the new team member and the team as a whole. Often the newcomer is assigned a mentor, often the team historian, designated as the first point of contact for questions about the code. The mentor helps to jumpstart the newcomer's mental model and social network. Newcomers are much more likely to read the team's design documents than seasoned team members. Some teams maintain online documents specifically for newcomers. Unguided exploration of the code is rare; more commonly the newcomer is assigned bugs specifically to introduce them to the code while minimizing risk. While all changes are code reviewed before checkin, newcomers receive extra attention and feedback on early changes they make. Several interviewees viewed fixing bugs as requiring less design knowledge than implementing new features. Bug fixing allows newcomers to do useful work while still learning the code base.

5.4 Code duplication

Two previous studies [5] [10] and the focus of clone detection tools (e.g. CCFinder [3]) led us to expect that when developers were asked about code duplication, they would discuss copying and pasting example API usage code, subclasses, or other hard-to-understand example code or even regale us with stories of hard to refactor clones. When pressed, a few admitted to copying and pasting code in dubious ways. Yet most responded with stories that had nothing to do with finding example code or copy and paste.

From our interviews, we identified six distinct forms of code duplication (Table 3), corresponding to columns in the table. Each clone type can be characterized by its creation mechanism, whether developers are aware they are creating clones, the refactoring challenges to remove the clones, and the size of the clones. Our follow-up survey also revealed the percentage of developers who had made changes repeated in multiple places or refactored or otherwise eliminated duplication during a one week period. Finally, developers rated the difficulty maintaining their code base caused by each type of clone.

In *repeated work clones*, multiple developers separately and unknowingly reimplement the same functionality. One developer reported that he had been implementing a small piece of functionality that another developer was also working on for a different problem until a program manager suggested that he talk to a second developer. After creation, interviewees viewed these clones as being difficult to refactor as each developer may have made subtly different decisions that are difficult to change.

The most studied clone type, *example clones*, occurs when some usage context code which illustrates how to create or make use of some code is copied and pasted and modified. We expect that this usually involves a small amount of code. Kim *et al.* [6] argue that copies frequently diverge and that it is difficult to predict whether the clones would be better off factored into a new abstraction.

Scattered clones, or logical clones, involve crosscutting changes in the aspect oriented programming sense [4]. Here, changing a particular decision requires making changes to many widely dispersed areas of code. One developer reported that correctly changing one method required changing another method that was hidden several calls deeper into the component. Another reported that they would sometimes make a change, hope for the best, and rely on testers to find any other necessary related changes.

Fork clones occur when a team takes a large portion of code from another team. One developer reported doing this when they wished to use code that the original team was not ready to ship. They subsequently heavily modified the code to remove functionality they didn't need. Forks occur when a consuming team wishes to use functionality provided by a producing team in ways that the producing team is unable to support. Interviewees, when asked, all agreed that it was best to avoid forked code whenever possible. Yet, when faced with the alternative of reimplementing the functionality from scratch, forking is frequently a better alternative. Particularly difficult are bug fixes. The consuming team must monitor bug fixes made by the producing team and reimplement the fixes themselves, taking on much of the maintenance burden of the producing team.

Branch clones occur when developers must reimplement their change in several branches of the same code base. They aren't

clones in the strict sense of duplicate code but rather copies of the entire code base in various stages of release. One developer reported fixing a bug in both code used in production and the current version under development.

Language clones involve the same code implemented in multiple languages. One developer reported having the same methods in both C++ and C#.

In contrast to the clone detection literature's narrow view of cloning as syntactically similar code, developers viewed cloning as making the same change several times. This includes many cases involving code not syntactically similar in a single code base but cloned across code bases or repeated in multiple languages or branches. From the developer's perspective, many of these problems seem similar in that individual bugs have to be fixed in several places, new feature work involves changes in many different places, or changes crosscut the strong team code ownership boundary. Future empirical work might be best served by focusing on this broader definition of repeating the same work.

6. RATIONALE AND COMMUNICATION

Understanding the rationale behind code is the most serious problem developers face among the problems activities survey respondents were asked about. 66% of the respondents agreed that "understanding the rationale behind a piece of code" was a serious problem (see Table 2). There are many facets to the rationale problem: 82% agree that it takes a lot of effort to understand "why the code is implemented the way it is," 73% "whether the code was written as a temporary workaround," 69% "how it works," and 62% "what it's trying to accomplish."

Consideration of rationale led us to understanding how developers understand and explore code. We found that developers had many complaints about using their tools to explore code, eschewed design documents for interrupting teammates, had code ownership boundaries to minimize how much they must understand, and rarely documented their understanding for others. This led to the second most serious problem - developers felt they were too frequently interrupted by their teammates. We also explored how developers maintained awareness of changes affecting their code and what developers meant by code duplication.

6.1 Investigating Code Rationale

When investigating a piece of code, developers turn first to the code itself: on average respondents spent 42% (± 29%) of their understanding time examining the source code, 20% (± 17%) using the debugger, 16% (± 19%), examining check-in comments or version diffs, 9% (± 10%) examining the results, 8% (± 12%) using debug or trace statements, and 3% (± 14%) using other means (Figure 5d). In other words, the code itself is the best source of information about the code. However it is not flawless. Developers commonly become disoriented in unfamiliar source code, and discerning the relationship between observed program behavior and the source code is often difficult.

When the code itself does not give the answers the developer needs, one might expect them to turn next to the vast amount of information that's written about it – the bug reports, the specs, the design documents, the emails, etc. This is emphatically not the case. Several factors combined to dissuade most developers from using design documents for understanding code. First, finding design documents was frequently difficult. Design documents were stored on internal websites without a usable search facility, forcing developers to manually navigate hierarchic collections looking for the appropriate design document. Thus, even if developers thought there was a possibility of a design document containing the information they cared about, it was not worth looking for. If search were available, it was not clear that developers would know the correct search terms. Second, design documents were not reliably updated. Thus, developers consulting a document would not be sure if the code still conformed to the document and would be forced to inspect the code.

The second recourse for investigating the rationale behind code is the social network. If the developer thinks a teammate might be able to provide the needed information (or the name of the person who might), she will walk down the hall to talk with them.

Once the developer has the desired information, she returns to her office, applies the newfound information, and gets on with her work. This information is precious: it is demonstrably useful, demonstrably hard to ascertain from the code, and was obtained at a high cost. Yet it is exceedingly rare for this developer to then write this information down. The next person who needs the same information must go through the same laborious discovery process. There are plenty of reasons that a developer would choose to not record the information. The overhead of checking the code out, editing it, and checking it back in (possibly triggering check-in review processes, merge conflicts, test suite runs, etc.) is enough to dissuade the developer from recording the information as a comment in the code. Some interviewees expressed the concern that the newfound information was not authoritative enough to add permanently to the code or that checking in the comment under their own names would inappropriately make them experts. Hence the information tends to remain in the developers' heads, where it is subject to institutional memory loss.

6.2 Interruptions

Each of these unplanned, face-to-face meetings represents an interruption of at least one person. Recovering from these interruptions is a substantial problem, ranking second with 62% of developers agreeing that this is the case (Table 2). Recovering from an interruption can be difficult. Developers must remember goals, decisions, hypotheses, and interpretations from the task they were working on and risk inserting bugs if they misremember.

Developers have adopted various strategies to mitigate the effects of interruptions on themselves, such as using a closed office door or other social cues to deflect interruptions, working on complicated tasks at times of the day when interruptions are infrequent, staving off an interruption for a moment while finishing a thought, or scheduling "office hours." Sometimes the interrupter mitigates the impact of interruption by using email instead of face-to-face for low-priority issues or emailing a warning 10 minutes before the interruption to give the interrupted person a chance to save his working context by writing down notes.

While many (though not all) interviewees indicated that they received too many interruptions, all acknowledged that interruptions were a valuable part of the work culture. Interestingly, two interviewees indicated that interruptions had become more of a problem since their teams had adopted agile processes.

6.3 Bug Investigation Example

Developers reported spending nearly half of their time fixing bugs. A bug investigation helps illustrate how their tools, activities, and problems interact to make fixing bugs possible but also suboptimal. When asked to describe an instance of a difficulty understanding the rationale behind a piece of code, one developer responded with a bug investigation narrative. While this is but a single story and not necessarily general and based on a recollection of events and not completely accurate, it illustrates several themes supported by interview and survey data.

After being assigned a new bug through a bug tracking tool, the developer first reproduced the bug by navigating to a webpage and ensuring that *error 500 – internal error* was triggered as reported in the bug. Next, the developer attached the Visual Studio debugger to the web server, set it to break on exceptions, reproduced the error again, and was presented with a null reference exception in Visual Studio. From an inspection of the call stack window, the developer considered the functions that might be responsible for producing the erroneous value. The developer switched to emacs to read the methods and used ctags.exe to browse callers of methods. The developer then switched back to the Visual Studio debugger to change values at run time and see the effects. The developer made a change, recompiled, and found that the same exception was still being produced. Finally, the developer browsed further up the call-stack, tracing the erroneous value to one object, then to another object, and finally to a third object protected with mutexes.

By this time, the developer had wandered into code that he did not understand and did not "own" – or have primary responsibility for making changes. But a second developer was working on a high profile feature that touched this code, so he immediately knew that this second developer would understand this code. He went to the second developer's office, interrupted the second developer, and engaged him in a discussion about the rationale behind the code. He walked back to his office, made a change based on this information, and determined that the change wouldn't work, leaving him with a new problem. He then walked back to the second developer's office who then him that the functionality causing the problem was actually related to code that a third developer was working on. They both went to visit the third developer's office only to find the third developer away for lunch. The first developer, now blocked, switched to another task. After lunch, both developers returned to the third developer's office, had a design discussion about how the functionality should behave, and finally passed the first developer's bug to the third developer to make the fix.

This story illustrates several themes in our surveys and interviews:

- Developers rapidly switch between multiple tools.
- When looking for detailed information about code, developers first explore the code by reading it and using a debugger.
- When unable to find answers exploring code, developers consult knowledgeable teammates rather than specs, design documents, email, or other artifacts.
- Face-to-face communication is strongly preferred over email or IM.
- Developers switch tasks when blocked or interrupted by teammates seeking code knowledge.
- Software development is a highly social process.
- While code ownership within a team is well understood, changes crosscut ownership boundaries.
- Developers spend vast amounts of time gathering precious, demonstrably useful information, but rarely record it for future developers.

Table 4. Agile practices adopted by respondents.

Does your team use	% agree
Collective code ownership within the team	49%
"Sprints," i.e. a development cycle that last four (or so) weeks	42%
An intentional policy to involve customers (internal or external) deeply into design and planning	33%
"Scrum meetings," i.e. a brief daily status meeting including all stakeholders	25%
"Burndown" estimate or chart, i.e. a measure of the time remaining in the sprint	24%
An intentional policy of preferring face-to-face over electronic communications	16%
Pair programming, i.e. developers working together, shoulder-to-shoulder on a problem	16%
A "bullpen" or other open-floorplan space for the team	10%

7. OPENNESS TO CHANGE

Developers and development teams are constantly trying new tools and work practices to optimize their work. Developers use a variety of tools to do their job. When writing code, 49% use two or more tools, and 19% use three or more.

In our interviews, we found several development teams experimenting with "agile practices," a collection of behaviors intended to make software development more efficient[2]. Some teams were gingerly dipping a toe into the agile water, while a small number were jumping in with both feet (see Table 4). 48% of respondents reported that their team was using two or more of the eight practices, 32% three or more, and 20% four or more. A few respondents (3%) reported that their teams used seven or all eight of the practices. Most developers wanted to continue adopting agile practices (53% agreed that they thought their team "should adopt agile software development methodologies more aggressively") while a few were skeptical (14% agreed that their team should adopt *less* aggressively).

Developers adopted specific agile practices when they felt their benefits were compelling. Developers shunned design documents in favor of face-to-face communication, designed minimally rather than up front, and employed unit testing. Developer leads reported preferring daily standup team meetings over weekly team meetings. Daily meetings encouraged teammates to help each other and assisted the lead in responding to problems blocking individual developers' progress. Several teams had gone further

[2] http://agilemanifesto.org/

by adopting an entire agile process, Scrum [11], and reported using radical collocation, collective ownership, and sprints.

8. DESIGN RECOMMENDATIONS

Several of the problems we observed might benefit from tool solutions, although further empirical work is first necessary.

Problem: Developers don't write down knowledge in design documents, resulting in constant rediscovery of knowledge known by developers working on the code in the past.

Solution: Reduce the cost of using design documents by (1) providing hyperlinks in code to design documents or (2) tools that capture informal whiteboard or paper designs. Two empirical questions that must first be answered are how readable informal notes would be for others and how much of what subsequent developers need to understand was ever explicitly considered by the original developer.

Problem: Interrupted developers lose track of parts of their mental model, resulting in laborious reconstruction or bugs and discouraging more frequent interruptions.

Solution: Externalize developer's task context – methods they've examined, decisions in progress, and other information – in a tool. This information could also be useful as documentation for future developers. The central empirical question is determining what information developers consider during a modification task.

Problem: It is difficult to discover and consistently change clones.

Solution: Embed hyperlinks between clone instances with editor support for navigating between clone instances.

9. CONCLUSIONS

Our exploratory study of developers' typical activities, tools, and problems led to a finding that is likely surprising to few – software development relies heavily on implicit knowledge. Yet, a detailed examination yielded more interesting findings – barriers preventing design document and email use, problems with interruptions, causes of duplication, and the deeply social nature of software development. We feel that wide-ranging, exploratory studies like ours have an important place within software engineering to keep tool development rooted in real problems developers face and fight the perceived irrelevance of academic software engineering research [8]. While many of our findings help inform tool development, many also need much more study. Finally, it not clear how this study of software development at Microsoft generalizes to software development in other professional environments. Given the diversity of environments – large software companies, small software companies, software developers in companies whose product is not the software itself, open-source development of commercial software – future work is needed to understand the generality of these findings.

10. ACKNOWLEDGEMENTS

Many thanks to Nachi Nagappan for his skills with statistical analysis, Miryung Kim for very helpful discussions including suggesting interview questions and an initial form of our clone taxonomy, and Andrew Ko, Marwan Abi-Antoun, Jim Herbsleb, and Brad Myers for careful readings of earlier drafts and helpful suggestions. The authors gratefully acknowledge support from a National Science Foundation Fellowship awarded to the first author and by NSF research grant IIS-0534656. This paper is based on work carried out while the first author was an intern on the Human Interactions in Programming team at Microsoft Research.

11. REFERENCES

[1] de Souza, C. R., Redmiles, D., Cheng, L., Millen, D., and Patterson, J. 2004. Sometimes you need to see through walls: a field study of application programming interfaces. In *Proceedings of the 2004 ACM Conference on Computer Supported Cooperative Work* (Chicago, Illinois, USA, November 06 - 10, 2004), 63-71.

[2] Hertzum, M. & Pejtersen, A. M. The information-seeking practices of engineers: searching for documents as well as for people. *Information Processing and Management*, 36, 5, 761-778, 2000.

[3] Kamiya, T., Kusumoto, S., and Inoue, K. CCFinder: a multilinguistic token-based code clone detection system for large scale source code. *TSE,* 28, 7 (Jul. 2002), 654-670.

[4] Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Videira Lopes, C., Loingtier, J.-M., and Irwin, J. Aspect Oriented Programming. In *Proceedings of ECOOP*, 1997.

[5] Kim, M., Bergman, L, Lau, T., and Notkin, D. An Ethnographic Stud of Copy and Paste Programming Practices in OOPL. *International Symposium on Empirical Software Engineering*, 2004.

[6] Kim, M., Sazawal, V., Notkin, D., and Murphy, G.C. An Empirical Study of Code Clone Genealogies. FSE 2005.

[7] Moran, T. P. and Carroll, J. M., Eds. *Design rationale: concepts, techniques, and use*. Lawrence Erlbaum Associates, Inc, 1996.

[8] Parnas, D.L. On ICSE's "Most Influential Papers". In *ACM Software Engineering Notes*, 20, 3, July 1995, 29-32.

[9] Perry, D., Staudenmayer, N., and Votta, L. G. People, Organizations, and Process Improvement. *IEEE Software, 11*, 4, 36-45, 1994.

[10] Rosson, M.B., and Carroll, J.M. The Reuse of Uses in Smalltalk Programming. *ACM Transactions on Human-Computer Interaction, 3,* 3, 219-253, 1996.

[11] Schwaber, K, & Beedle, M. *Agile Software Development with Scrum*. Prentice Hall, 2001.

[12] Singer, J., Lethbridge, T., Vinson, N., and Anquetil, N. An Examination of Software Engineering Work Practices. In *Proceedings of CASCON '97*, 209-223, 1997.

[13] Weiser, M. 1981. Program slicing. In *Proceedings of the 5th International Conference on Software Engineering* (San Diego, California, United States, March 09 - 12, 1981), 439-449.

[14] Wright, G. and Ayton, P. Eliciting and Modeling Expert Knowledge In *Decision Support Systems*, Vol. 3, 13-26, 1987.