Let's Go to the Whiteboard: How and Why Software Developers Use Drawings

Mauro Cherubini
CRAFT
École Polytechnique Fédérale de Lausanne
Station1, CH-1015 Lausanne, Switzerland
mauro.cherubini@epfl.ch

Gina Venolia and **Rob DeLine**
Microsoft Research
One Microsoft Way, Redmond, WA 98052
gina.venolia@microsoft.com
rob.deline@microsoft.com

Andrew J. Ko
Human-Computer Interaction Institute
Carnegie Mellon University
5000 Forbes Ave, Pittsburgh PA 15213
ajko@cs.cmu.edu

ABSTRACT
Software developers are rooted in the written form of their code, yet they often draw diagrams representing their code. Unfortunately, we still know little about *how* and *why* they create these diagrams, and so there is little research to inform the design of visual tools to support developers' work. This paper presents findings from semi-structured interviews that have been validated with a structured survey. Results show that most of the diagrams had a transient nature because of the high cost of changing whiteboard sketches to electronic renderings. Diagrams that documented design decisions were often externalized in these temporary drawings and then subsequently lost. Current visualization tools and the software development practices that we observed do not solve these issues, but these results suggest several directions for future research.

Author Keywords
Software visualization, diagrams, exploratory/field study.

ACM Classification Keywords
H.5.3 [Group and Organization Interfaces] Computer-supported cooperative work; D.2.10 [Design] Methodologies, Representation

INTRODUCTION
Diagrams are important tools in every design and engineering discipline. They support reasoning and problem solving [15,21] and in some disciplines, such as civil engineering [7] or mechanical prototyping [10], diagrams are fundamental to practice.

Few studies, however, have investigated diagram use in software development activities. While we might expect a similar use of visual representations in such work, other research suggests that developers are bound to the written form of their code, and so source code editors are the most-used tools for design despite being considered less effective than paper or whiteboards [16]. This suggests that there may be fundamental differences between software engineering and other types of engineering.

To begin to describe these differences, we performed an exploratory study of *how* and *why* developers draw their code. As industrial software development happens in teams, we additionally focused on the social practices around diagrams and visualizations. This provided a social perspective that was essential to understand group dynamics. We started with some research questions:

A. How do engineers use diagrams in their work?

B. Why do engineers use diagrams in their work?

C. What graphical conventions do engineers use?

D. What is the culture around these drawings?

To answer these questions we conducted a field study at Microsoft Corporation to assess developers' perspective on these issues. This involved an initial recruitment survey, a series of interviews, and a final survey, which helped assess the generality of our findings with a larger group of developers. We found that diagrams play largely a supportive role in software design and that drawings are often ephemeral because of the labor involved in translating them into more permanent forms. These findings and others provide useful insights into the design of a wide array of software-visualization tools as well into the use of diagrams in design work in general.

The next section will present some related field studies reporting results of software visualization. Then we will describe our methodology detailing the results of our investigation. We will then discuss the implications of our findings for software development and for collaborative design work in other disciplines.

DEFINITIONS AND RELATED WORK
In this research we will use many synonyms of the word *diagram*, including visualization, sketch, representation, and others. All these words are used to mean a *simplified and structured visual representation that shows entities and relationships representing the architecture or implementation of a software system*. These diagrams might represent any of the architectural aspects of software system, e.g. class inheritance, data flow, flow charts, state

CHI 2007, April 28 – May 3, 2007, San Jose, California, USA.

machines, sequence diagrams, database tables and relationships, architectural layer diagrams, relationships between servers and clients, etc. [19]. For the purposes of this paper, diagrams can be rendered on several media: *sketches* on a whiteboard, in an engineering notebook, on scrap paper, on a Tablet PC, or other medium; *reverse-engineering visualizations* produced by tools such as Source Insight (an integrated development environment), Visual Studio's Class Designer (a diagramming tool in Visual Studio), SQL Server Management Studio's Database Designer, or other reverse-engineering tools; and *drawings* in Visio, PowerPoint, etc., or even ASCII art in source files.

Diagrams of these types have been studied extensively in other design disciplines [10]. There are a number of findings that may be relevant to the study discussed in this paper. For example, in studies of other work domains, designers sketch for four different but intertwined reasons:

- *To share*: Diagrams play a major role in communication [22], as they externalize internal thought making it visible to self and others [21], reifying the mental model for others to act upon.
- *To ground*: Human communication embeds ambiguous interpretations that need to be clarified in conversations [3]: diagrams can serve this purpose.
- *To manipulate*: By externalizing a mental model in a drawing, part of the cognitive process needed to hold it in memory is relieved and other operations can take place, like joining different parts, evaluating the design, checking the consistency, etc. [1]. Once externalized, these phases can happen collaboratively, capturing joint attention and enabling gesturing [1,8].
- *To brainstorm*: Ambiguity in sketches is a source of creativity. Unintended interpretations and ideas can arise when inspecting an initial arrangement of a sketch [20].

The cognitive implications are manifold: diagrams support communicating, capturing attention and grounding conversations [4]. They reduce the cognitive burden of evaluating a design or considering new ideas [13]. Unlike sketching buildings or mechanical parts, code is an abstract entity with few spatial features other than the visual layout of code within a source file. This means that unlike cartography or architectural renderings the representation of code does not follow any intrinsic spatial mapping.

Studies of software development practices also suggest some roles of diagrams in the work. For example, one reason developers might need to create and maintain complex mental models of their code [16] is that design information is not codified in a persistent manner: documentation of the code is scarcely used during development because it is often outdated and the diagrams that are created in this process may not persist. There is also evidence that software developers use whiteboards to support face-to-face conversations in service of awareness and knowledge sharing [18]. One common problem-solving strategy is for a developer to walk to a teammate's office seeking for contextual information and brainstorm over the problem they are facing [2,12,17].

Given this prior work, diagrams and drawings seem to have an important role in software development, but we have little understanding about the extent to which they are used, and how their use compares to the use of diagrams in other disciplines. This knowledge is important in designing any kind of support tool for software development, and it may also help reveal fundamental aspects of diagram use across different engineering disciplines.

We were informed in part by socially distributed cognition theory, which argues that work occurs not only within people's mind, but also between people, artifacts and tools [13]. Unlike other studies from this perspective [2,18], however, our study specifically focuses on diagrams as artifacts in a social context.

METHOD

We used semi-structured interviews and surveys in our investigation. An initial survey allowed us to recruit interview participants. The interviews helped us understand what kinds of representation were used, for which reason, by which modalities, and in which media. From the interview results we developed a model of drawing use, and then validated it with a large-scale survey.

Interview recruitment

Recruitment for interviews was based on whether a developer used visualizations of any sort and whether any of these artifacts were placed in the person's personal or shared workspace. We focused specifically on developers who already used diagrams in order to assess under which conditions diagrams were used for their individual and collaborative work. Investigations of developers who did not use diagrams were left for future work.

To select the participants we deployed a short survey to a randomly drawn sample of 350 Microsoft developers. Besides asking for some biographical facts, the aim of the survey was to gain knowledge on the two factors above. Sixty developers responded to our survey within a week, 45 of whom stated that they used code diagrams in their work. Fourteen respondents stated that they had diagrams

Pseudonym	Title	Historian	Team Size	Product
Andrew	SDE	No	3	Data tools
Geremy	SDE	Yes	7	Communication
Tom	SDE	Yes	30	Entertainment
Colin	SDE	No	6	Mobile device tools
John	Architect	No	> 100	Development tool
David	SDE	No	8	Advertisement
Ray	Lead	No	20	Input device UI
Nigel	SDE	No	20	Multimedia

Table 1: Developers interviewed and details about their role and their team size. These are sorted by interview date.

displayed in their office space. Out of this last group we interviewed 9 developers, stopping after we felt that we were hearing similar answers from the respondents.

Table 1 contains the details of the interview participants along with pseudonyms that we will use in the next sections when describing anecdotes. A group "historian" is the developer lead or the person whom has been with the project the longest.

Interview protocol

The first two authors conducted the interviews, which typically lasted 45 minutes. After introductions, we explained the goals of the study, that their answers would be anonymous, that they could decline to answer any question, and that they could terminate the interview at any time. Additionally, we asked permission to audio record the conversation and photograph drawings that they showed us.

During the interview, we followed a list of questions that was organized in four functional areas: WHAT (e.g., "Please, tell me something about this visualization."), WHY (e.g., "Why did you produce this visualization?"), WHEN (e.g., "When did you use it last? For what purpose?"), and HOW (e.g., "How do you use it?"). We did not ask the questions sequentially but we tried to respect the flow of the conversation, always trying to touch a couple of points in each of the four areas.

Survey

We performed a preliminary analysis of the drawings that the interview participants showed us or described. We clustered them based on the situation in which they were created, and identified nine recurring *scenarios* where drawings were produced by developers. The scenarios are described in the Motivations and Scenarios section, below.

To learn more about the scenarios we performed a survey. We identified over thirty questions we wanted to ask about each scenario. Rather than having survey respondents answer questions about all scenarios, we created a family of six surveys, where each respondent answered regarding only three scenarios, and each scenario appeared in two surveys. We controlled the order of the scenarios within the surveys so that a scenario appeared either in the middle in both surveys, or first in one and last in the other. The surveys were implemented as intranet web pages.

We filtered the Microsoft address book to find the 8,570 full-time employees with titles indicating that they were software developers, development leads, or architects. We deployed each survey to 400 people selected randomly in non-overlapping sets from this list. In an effort to increase participation, we gave US$100 gift certificates to five randomly-selected survey respondents.

We received 427 responses overall (18% response rate), 60-76 responses to each survey, resulting in 130-152 responses per scenario. Respondents were 81% software developers, 11% development leads, 5% architects, and 3% other. Respondents were 7% female, 85% were 20-39 years old, and the median as a professional developer was 7 years.

RESULTS

In this section we describe the visual conventions developers used in their drawings, and then describe the scenarios where developers use drawings. Where possible, we provide quotes from the transcript of the interviews, which have been edited for clarity. *Italics* in the captions indicated vocal emphasis from the speaker.

Visual conventions

Developers used a variety of informal visual conventions, often mixing them freely. Boxes-and-arrows diagrams were by far the most common, representing entities and the relationship between them (**Figure 2**, **Figure 3**, and **Figure 4**). Iconic pictures were used instead of boxes to represent special kinds of entities, e.g. database (cylinder), OLAP data cube (cube), computer (CPU tower), or person (stick figure). Circles were used instead of boxes to represent states in state-transition and security threat model diagrams. Boxes or their equivalents were almost always labeled with text. The size of the box sometimes encoded the importance or size of the entity being represented. They were sometimes grouped into higher-order structures, usually using large boxes or dividing lines.

> During meetings we sketch block diagrams now and then. Not necessarily complicated. In this case the boxes represent components or object entities that can live in this scenario. We tried to distinguish big pieces from small components, highlighting things that are more important. [Colin]

> We used this drawing to explain *who* is contained in *whom*, *who* manages *whom* or *who* maintains *whom*. [Colin][1]

Relationships between entities were usually represented with arrows. They were almost always directed and generally pointed rightward or downward (though some drawing types had different conventions, such as class-inheritance where arrows pointed upwards). Arrows were sometime labeled, or numbered to indicate sequence. Often the type of relationship represented by an arrow was not explicitly stated, even when multiple types of relationships were present.

> In a *deep dive* of a data structure we use boxes and arrows to show the points of connectivity, inheritance, etc. between two teams, one working on the data structure and the other working on the main system. [John]

> Being a database designer, I use a lot of Visio to produce ERD diagrams. We have projects with thousands of tables partitioned over several

[1] To explain the nature of the relationships in the code and in their drawings, developers often used anthropomorphic metaphors. The "whos" in this quote referred to classes in the code that were represented in the drawing with some boxes. These references were consistent with Herbsleb's observations [11].

databases. Sometimes we produce diagrams to show the data flow in the system. [Daniel]

Boxes were abutted, in lieu of arrows, when there was a simple chain of relationships between the entities, and for architectural layering models. One-to-many relationships were represented by revealing a second box slightly down and right, graphically suggesting a stack of boxes. We observed many types of entities in diagrams, including classes, methods, executable binaries, processes, databases, database tables, hardware devices, UI screens, states, process steps, and people. We also observed many types of relationships, such as inheritance, data reference (e.g. pointer or foreign-key), data access ("talks to"), procedure call, message passing, transition, and containment.

The boxes were arranged such that related things were close in proximity. Boxes were arranged so that relationships "flowed" in a dominant direction, usually left-to-right or top-to-bottom. Colors were rarely used to encode meaning.

Results from the survey showed that the adoption of standards of any sort and the level of accuracy in the diagrams was low across all the scenarios (**Figure 1**f, which is explained in the next section).

Motivations and Scenarios

Based on our interview notes we identified nine scenarios where developers employed drawings. We categorized the scenarios on two independent dimensions: the developer's *motivation* in creating the diagram, and the *stage of investment* in producing it. The association between the scenarios, motivation, and investment is shown **Table 2**, where we identify each scenario with a short phrase. Each scenario is described in detail in this subsection, which is organized by motivation. We will return to the stages of investment in the next subsection.

↓ Investment \ Motivation →	Understand	Design	Communicate
Transient	1) Understand	3) Refactor	
Reiterated	2) Ad-hoc		5) Onboarding 6) Secondary stakeholders
Rendered		4) Design review	7) Customer
Archival			8) Hallway art 9) Documentation

Table 2: The model of diagram use derived from interviews and survey responses. Scenarios are categorized by the developer's motivation for creating the drawing and the developer's investment in the evolution process of the drawing.

We found three main reasons why developers produced visualizations: *to understand*, *to design* and *to communicate*. There is a natural progression to these motivations: a thing must be understood before it can be designed, and must be designed before it can be communicated. We assigned each scenario to one of these *motivations* . There was variation in the importance of each scenario to the developer's work, and in the importance of the drawings in the scenario, as shown in **Figure 1**a.

Scenarios motivated by understanding

Developers maintained complex mental models of their code during development and used diagrams to update and

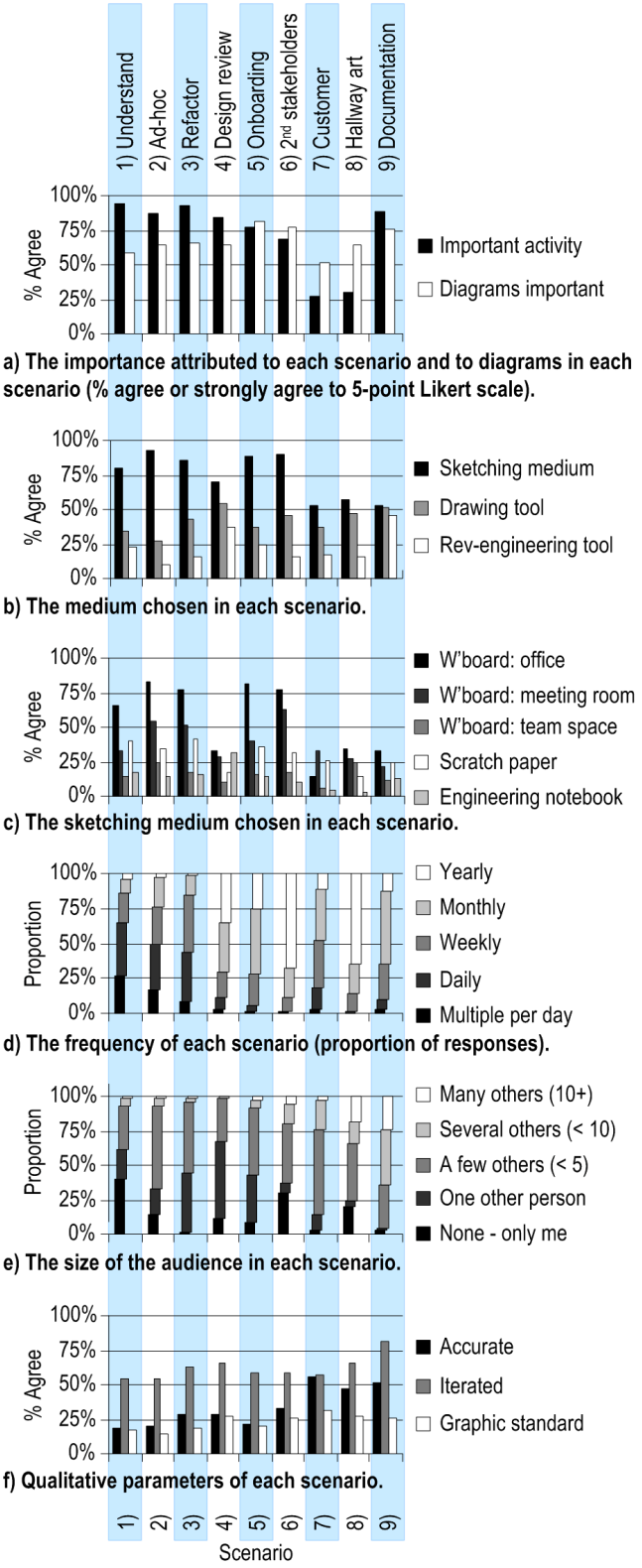

Figure 1: Survey results per scenario (see text).

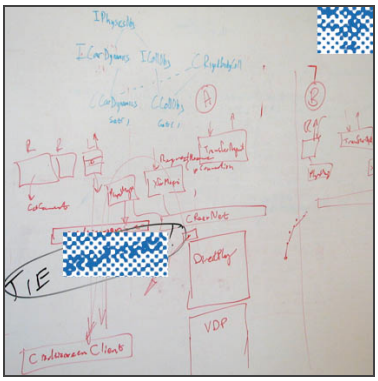

Figure 2: A developer's office whiteboard, with drawings produced during multiple ad-hoc meetings [Tom].

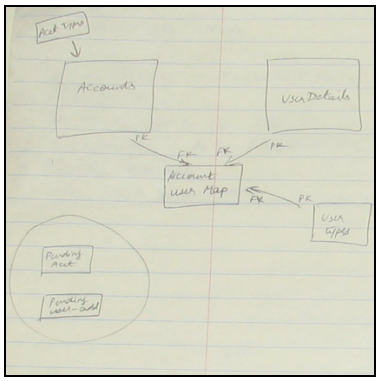

Figure 3: A notebook sketch supporting the design/refactoring scenario [Andrew].

Figure 4: An example of hallway art [Colin]. We masked confidential information (same as in **Figure 2**).

expand these models as familiar code evolved and while exploring unfamiliar code. We observed two scenarios where our interviewees used diagrams in this way.

1) **Understanding existing code**: Developers examined the source code and its behavior in order to develop an understanding of it. Survey respondents rated this as the most important of the nine scenarios (see **Figure 1**a, which shows that 95% of survey respondents agreed or strongly agreed with the statement, "Participating in this activity is an important part of my job function"), as well as the one they engaged in most frequently (**Figure 1**d). It was the scenario most likely to be done alone, but was often performed in small groups (**Figure 1**e). Drawings were not particularly important to save in this scenario (**Figure 1**a).

> Before I go to someone else to ask for specific information I try to understand the thing for myself. In this case I sketch a diagram on everything that is available. In this way I am not wasting someone else's time. [Nigel]

> States are almost invisible in code. We draw state diagrams for threat modeling. [Geremy]

In all scenarios, sketches predominated, reverse-engineering tools were used least, and computer-based drawing tools were used an intermediate amount (**Figure 1**b). Reverse-engineering tools were used to a limited degree in this scenario, but less than many others.

> I remember this one time where I wanted to quickly see the inheritance of a bunch of classes. So I quickly created a diagram with the Object Browser feature of Visual Studio and then I throw it away. [Colin]

As for almost all scenarios, office whiteboards were the most common medium for sketches. This was one of the scenarios in which paper-based sketches were most prevalent (**Figure 1**c). Developers were the least concerned with accuracy in these drawings and the least likely to use a graphic standard (e.g., UML: Unified Markup Language) (**Figure 1**f).

2) **Ad-hoc meeting:** When a developer reached an impasse while trying to understand existing code or needed to vet a design decision with a teammate, he would walk to another developer's office, interrupt her, and then engage her in a brief discussion. Impromptu meetings like this were crucial for transferring knowledge among the development team. This was among the most-frequent scenarios (**Figure 1**c).

As the discussion progressed, sometimes one of the participants turned to the whiteboard to sketch (**Figure 2**), typically drawing a very rough caricature of a portion of the architecture, often with nearly-illegible labels. The drawing was produced during the conversation and was secondary to it. If the other participant engaged in the drawing she typically used a pen of a different color, leading to a kind of informal authorship record.

> When a developer comes to me to discuss a new *Addin* we use this diagram to check whether its implementation respects the criteria. [John]

> I use the whiteboard when I am brainstorming with a colleague. Even the visualization tool Source Insight would not give you multiple inheritance hierarchy. [Tom]

> One of the PMs came to me and drew this picture on the board to ask my opinion on this model. He did that incrementally while he was talking. [John]

> When I need to explain to a colleague how some stuff works then I use the whiteboard. [Nigel]

Developers were more likely to use sketches in this scenario than any other, and the least likely to use reverse-engineering tools and drawing tools (**Figure 1**b). This was among the scenarios where developers were least concerned with the accuracy of the drawings (**Figure 1**f).

Scenarios motivated by designing

There were two scenarios in which developers used drawings in design phases before changing code.

3) **Designing/refactoring**: Developers planned how to implement new functionality, fix a bug, or make the structure better match its existing functionality. This was one of the most important scenarios; diagrams were somewhat important in this process (**Figure 1**a). An example is shown in **Figure 3**.

> I look at the diagram and if I see lots of fields in a certain table I see that is a potential candidate for restructuring. Or maybe I have a small table with lots of joint connections out of it. The diagram helps identify design problems. [Daniel]

This was one of the scenarios in which paper-based sketches were most prevalent (**Figure 1**c). The resulting drawings served as a visual to-do list, and helped to keep the developer oriented in the "big picture" when later implementing the details of the design.

> I use diagrams to make explicit each assumption I have while I am writing algorithms. I use a block diagram style: each function is represented with a block or eventually a block represents a logical step that my code needs to accomplish. [Andrew]

4) **Design review**: When a proposed design change was complex or far-reaching, developers performed a design review to inform and seek input from the affected people. Design reviews were performed face-to-face, by email, or, in rare circumstances, by teleconferencing. Design reviews were important, but relatively infrequent among the important scenarios (**Figure 1**a and d). Design reviews were often done in pairs and rarely done with more than a few people (**Figure 1**e). Diagrams were used to evaluate the design of the system or to propose changes.

> We did go through different meetings to understand what is what we call *the game* and what we call *the engine*. We wanted to be sure that the core was abstract enough and diagrams helped in figuring out where these boundaries were. [Tom]

> We had many discussions to evaluate different scenarios of implementations at group level. This diagram was a great tool in these situations to keep the focus of the conversation. [Geremy]

> I remember when one of these diagrams triggered a discussion to find a hole in our logic. We had to go back and change the design. [Geremy]

Drawing tools were most likely to be used in this scenario, this was among the top scenarios for reverse-engineering tools, and sketches were used somewhat less than in other scenarios involving team members (**Figure 1**b), suggesting a level of formality and refinement. Engineering notebooks were used more in this scenario than any other (**Figure 1**c).

Scenarios motivated by communicating

Five scenarios involved using drawings to communicate.

5) **Onboarding**: When a developer joined a team he apprenticed with a more-senior developer to acquire a mental model of the code. This process included focused meetings where the mentor explained the code, and ad-hoc meetings and email discussions to answer questions as they occurred. Diagrams were crucial to this scenario (**Figure 1**a). This was one of the scenarios for which reverse-engineering tools were unusually useful (**Figure 1**b). This was one of the scenarios where developers were least concerned with the accuracy of the drawings (**Figure 1**f).

> My manager used this diagram to explain the code to me when I first started. Recently I realized that I used kind of the same diagram to introduce a new hire to the project. [Andrew]

6) **Explaining to secondary stakeholders**: For any particular component in a software system, there were many stakeholders beyond the core development team, including testers, project managers, and internal "consumers" of code. Developers typically communicated with these people through face-to-face, scheduled but informal meetings. Informing these "secondary stakeholders" about the code was less important to the developer's job function than other core scenarios, and it was the rarest among them (**Figure 1**a and d). This was among the scenarios where drawings were most important (**Figure 1**a). Whiteboard sketches were the dominant form of drawing (**Figure 1**b), with an unusually high usage of meeting-room whiteboards (**Figure 1**c).

> I was implementing a new feature and I had to make a design decision and I wanted my PM to approve it. As it was complicated to explain what I had in mind, I sketched it on paper. [Andrew]

Sometime these drawings took a more formal character when the communication spread outside the team to reach other departments of the same company.

> We used the diagram tool of SQL server to reverse engineer the structure of the database and then we stuck the generated diagram in the documentation. [Geremy]

7) **Explaining to customers**: Developers were responsible for presenting the architecture or usage of the software to external customers. This took the form of a live or recorded lecture, hands-on lab, or other setting. Developers judged this to be one of the least important activities, and one for which sketching was least common (**Figure 1**a and c). It was the scenario in which developers most strived for accuracy and to use graphic standards in their drawings (**Figure 1**f), suggesting a high degree of formality.

> I had to use this diagram with customers, but the state diagrams that we were using were too complicated so I had to simplify it focusing on the individual components. [Geremy]

8) **Hallway art**: Developers sometimes tried to foster team awareness of aspects of the architecture by displaying information about the code in the team's space. This was one of the techniques used by team leads to maintain every developer "on the same page," and which encouraged in the Agile Methodology. However, most developers considered it an unimportant activity and it was performed with a low frequency (**Figure 1**a and d). This was among the scenarios for which sketches were used least (**Figure 1**b), although developers had the highest standards of accuracy for hallway art (**Figure 1**d). **Figure 4** shows an example.

> We put these diagrams in the conference rooms and in the hallways so that the developers could stare at them while writing a piece of code. [Geremy]

> When we do planning or spec writing, we come out with this kind of design. Then we dive into implementation. We refer to these diagrams every now and then to communicate with the rest of the team. [Colin]

While we were interested primarily in developers' drawings, much of the hallway art that we observed was created by other stakeholders, particularly program managers and user interface designers, for developers' use. Unlike developers' drawings, these represented the function of the code but not its implementation.

9) **Documentation**: Developers created documents describing the architecture, usage, or internals of the code for teammates, other internal customers, or external customers. This activity was rated as very important, in which drawings played a crucial role (**Figure 1**a). Reverse-engineering tools were used more in this scenario than any other; this was among the scenarios where drawing tools were most likely to be used and sketches least (**Figure 1**b). This was the scenario with the largest audience size (**Figure 1**e), the most likely scenario for a drawing to be refined through iteration, and among the scenarios where care was taken to make the drawings accurate (**Figure 1**f). Together these suggest a high degree of effort, formality, and refinement in drawings made for documentation.

>We have many sectors, which contain a rigid number of servers. We wanted to change that for scalability issues and so I was using these diagrams to explain [to the sustained-engineering group] the inner working of each machine and the proposed change. [Geremy]

Levels of investment

The previous section organized the scenarios by the developer's motivation in creating them. This section presents the same scenarios from the perspective of the effort involved in their production. Most drawings were *transient*, such as a simple sketch or reverse-engineering visualization that served a task and had no later value. Some such sketches were of sufficient value to be *reiterated*, recreated from memory in different contexts. Through repetition these sketches became touchstones for a project. When persistence was needed, some touchstone sketches were *rendered* using computer-based drawing tools. Finally, when these rendered drawings were to be presented in an asynchronous manner or in a more formal setting, greater care was put into creating *archival*-quality renderings (see **Table 2**).

Transient sketches and reverse-engineering visualizations

Most drawings created while understanding code or designing and refactoring (scenarios 1 and 3), were one-off, transient whiteboard or notebook sketches. Visualizations from reverse-engineering tools were also transient.

> Whenever there is something that I am trying to workout in my own head I just write it down, using the whiteboard to map out all the cases. [Tom]

> Diagrams [on the whiteboard] have a short-term functionality. They solve the purpose of discussing or detailing the current problem. They rarely get updated. If a diagram needs to be consulted for a long period of time then it is usually rendered or copied in a notebook. [Andrew]

> I wrote down the diagram that I visualized in Source Insight so that I could annotate it. [Tom]

The value of the diagrams was secondary to that of the setting in which they were generated. As soon as the setting ended, their value decreased immediately.

> Most of the whiteboard drawings that I do are not used in other meetings. They are just useful during the one to one meeting. [Tom]

> The role of the whiteboard drawing during meetings is to direct the natural flow of the conversation. To quickly sketch something on the whiteboard is more convenient that using a laptop and a projector. [Colin]

> Usually I use diagrams in one-to-one meetings. The discussion that I am having while I am drawing is always more important than the drawing. I keep referring to the drawing to remember the discussion. [Ray]

Reiterated sketches

In some cases, a transient sketch was redrawn in different contexts, in whole or in part, evolving over time. This was particularly true in ad-hoc meetings, onboarding, and explaining to secondary stakeholders (scenarios 2, 5, and 6). These drawings typically captured either the high level architecture or some particularly crucial part of the design. Through reiteration they became touchstones for the team.

> The design of this current release started with this diagram. It started on the whiteboard, and then it evolved over time. As it started to become more static, after a month, it was still used to brainstorm new ideas. [Geremy]

> This diagram was developed over three years. It is a canonical version of what we call 'Addin' model. We have copies of this all over. The labels changed over time. We decided to give it this butterfly shape to emphasize that the contract in the center shouldn't change and should be small… [John]

Interviewees reported reiterating these dozens of times.

Rendered drawings

In some cases, a diagram—typically a reiterated sketch—became so important that it warranted the investment of time and effort to transform it to a more permanent form. This occurred particularly in design reviews and when communicating with secondary stakeholders (scenarios 4 and 7). Sketches were sometimes transcribed to another sketch medium, e.g. notebook or Tablet PC. When greater permanence, portability, or malleability was required the drawing was rendered in Visio, PowerPoint or Word, or even ASCII art embedded in code.

> There have been a number of meetings where we had to copy the diagrams done on the whiteboard down on paper because they need to be elaborated by each developer individually. [Colin]

> Sometimes a design refactoring starts in a chalk session. If the changes are fairly complex then we tend to copy the diagram down. [Daniel]

> A Tablet PC is very helpful. Pictures make more sense to me than words, so each meeting I start from a diagram and the tablet helps me to take notes in visual format in the meeting. Later I can render these in Visio. [Geremy]

> I use ASCII art to attach the most relevant diagrams to the code. In this way is very relevant and it is right there where it need to be used. [Tom]

Moving the sketch to electronic form allowed developers to modify it. **Figure 5** shows an example of transformation of a tablet sketch done during a group meeting into a Visio drawing used later as hallway art and in documentation.

Archival drawings

When a diagram was for documentation, greater care was taken to refine its content and visual presentation, and to create surrounding text to explain it (**Figure 1**e).

> The rest of this diagram is there only to give context to the people looking at it, only this portion is relevant. [Geremy]

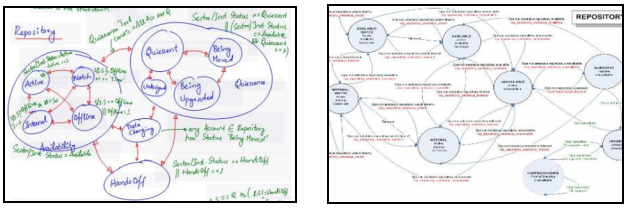

Figure 5: On the left, the hand-sketch created on a Tablet PC during the group meeting; on the right, its rendering in Visio. It is possible to see that the some aspects of the first diagram were changed and elaborated in the second version of it. [Geremy]

> To explain this diagram, I had to write a paragraph to explain what each element does. What was very tough was trying to superimpose multiple instances on top of a sector, which was very difficult in Visio. [Geremy]

> This diagram isn't self-explanatory. People should still come to me and ask for complementary information to understand some design decision. [Geremy]

The maintenance of the diagrams in sync with the evolution of code was easier when the visualization existed in an electronic format. However, even in this scenario maintenance had a high cost, and so these diagrams were often out-of-date.

> These VISIO drawings are mainly used for communications with PMs. I don't think I have ever updated one of these. [Colin]

DISCUSSION

We return to the four questions posed in the introduction.

A. How do engineers use diagrams in their work?

Developers used transient forms for exploration activities, creating diagrams with reverse engineering tools and sketches on whiteboards, scrap paper, or notebooks. On the other hand, they used permanent solutions when communicating with the larger group or with other people.

The majority of diagrams observed were sketched on whiteboards during ad-hoc meetings. The whiteboard offered great advantages as it was ubiquitous and easy to use. The few elements of interest could be easily abstracted and then annotated with additional information focusing on the particular discussion. These results were in line with Dekel's findings [5], which describes how group of developers used sketches while collaborating.

Reverse engineering tools were preferred for quick *solo* code explorations. These visualizations were discarded immediately after the desired information was acquired.

The transience of casual sketches seems to be a difference with other disciplines like architecture, where these are often archived with great care as a record of design process.

B. Why do engineers use diagrams in their work?

Similarly to the studies of Tversky [21], developers produced visualization for three main reasons: *to understand*, *to design* and *to communicate*. However, while for other disciplines diagrams might be the standard way of communicating (e.g., architecture, mechanical engineering, etc.), this is not the case for computer science, where the "code is the king" [16]. In our observations, this resulted in the tendency to adopt informal, ad-hoc notations.

C. Which graphical conventions are used?

During solo explorations or peer-to-peers meetings, developers did not follow any graphical standard, and used an informal, ad-hoc style. In the majority of cases the developers used a simple boxes-and-arrows visual language whose elements assumed meanings depending on the context. This may be because of convenience rather than preference or specific requirement. However, for documentation, a more formal style was chosen, though standard notational systems were rarely used. Visualizations in documentation were often out-of-date and they were rarely used for the core of the development process.

The use of formal graphical modeling languages, such as UML, was very low. We do not have precise results to explain why this was so, but we can speculate that UML requires too much effort to learn and that is too formal for the majority of the visualizations that were produced in informal settings. It is clear that UML does not, in general, reach a sufficient cost-benefit ratio in the minds of developers to warrant its use. However, we have to say that all the diagrams in the situations we observed *could* have been rendered proficiently using an UML notation. Some of the developers we talked to said explicitly that there was not a culture of modeling languages in their team.

Of course, another issue is whether the use of such standards would be beneficial at all. Our results suggest the importance of context in drawings and the necessity of rendering multiple levels of detail within a single drawing. These are properties that UML and other such notations provide little support for.

D. What is the culture around these drawings?

Our findings show a limited adoption of drawing tools, and adherence to standards of any sort. This result is consistent with the work of LaToza et al., who report that despite the availability of visual editors such as tools for UML, developers remain focused on the code itself [16]. Particular diagrams retained a high value for the group, which brought the developers to reiterate their design in a similar way as described by Henderson [10].

Production costs

Today's tools make the production of diagrams easy, but despite this, the effort required was *perceived* as exceeding developers' time resources. Whiteboards were the most adopted tool for producing visualizations because they were ubiquitous and their perceived cost of use was extremely low. A smaller number of visualizations were produced for internal communication purposes. Here the production costs

increased, as the images had to be rendered electronically for sharing. These images did not require lots of details because the contextual information necessary to understand them was already part of the group knowledge. Finally, few diagrams were produced for an external audience. This because the drawings included in documentation had a high production cost due to the effort necessary for providing contextual information to make them intelligible.

Optimal collaborative effort

In this study we found some evidence that when diagrams were generated automatically, they seemed to be regarded as less interesting than diagrams that were produced manually in a collaborative effort. At group level it is important to maintain an *Optimal Collaborative Effort* [6] to ensure a proper grounding process: when the task is too challenging, people might be overwhelmed; while if the task is not challenging enough, as in the case of automatically provided information, participants may not be fully engaged in the cognitive activity. Manually produced diagrams could capture the developer's attention and efficiently scope to the information necessary to the task or conversation. On the other hand, automatically-produced visualizations do not require developers to externalize their mental models nor do they allow for flexibility in the level of detail. This may be why digital diagrams resulted in less useful and less appreciated diagrams.

Validity

Several factors influence the validity of our results. We developed the entire model within a single company, which might not be representative of companies with different cultures or practices. The use of drawings of code in highly-distributed or open-source software development is likely to be different. Our interview sample is small compared to Microsoft's developer population and it was also biased by our selection for developers actively using visualizations in their work. However, in our validation survey, we had a much broader and representative sample. Finally, surveys and interviews are well known to be subject to respondents' self-perceptions. Our findings should be compared to studies of software development work in the field.

TOOL SUPPORT FOR CODE DRAWINGS

Our results reveal many opportunities for tools that help developers capture, create, and share their drawings

Capture

Many design decisions are made during one-to-one meetings [14, 16], In many of these meetings developers produced a diagram as the conversation was occurring. Developers might benefit from recording these events. Such a recording could encompass the conversation, the sketch as it evolves, and the deictic references made to the sketch. These recordings could be especially relevant to ad-hoc meetings (scenario 1) and design reviews (scenario 4).

Many of the developers interviewed suggested that they desired some sort of "intelligent whiteboard" to augment the drawing process and capture the result in electronic form. Tablet PC's, which are capable of exactly that, are in wide deployment in Microsoft, yet few developers used them for this purpose. Digital cameras make whiteboard capture trivial, and are ubiquitous, yet we saw little use of them for this purpose. There are many commercial products for digitizing whiteboards, yet we saw no adoption of these technologies. There are many potential explanations for this lack of adoption, such as the perceived cost of their use we found in our surveys. These explanations should be investigated further.

Virtually all developers had whiteboards in their offices, every team had whiteboards in the hallways, and every meeting room had a substantial portion of its wall space dedicated to whiteboards. Were a whiteboard capture system implemented it would ultimately be rolled out to the tens of thousands of whiteboards, with an exorbitant cost of deployment and maintenance. Despite these problems, and lack of adoption of existing capture techniques, the benefits of capturing meetings (including the whiteboard contents as it evolves) could be substantial.

Integrating reverse-engineering and sketching

In their sketches developers often combined aspects of the current state of the code with proposed changes when understanding existing code, designing/refactoring, and onboarding (scenarios 1, 3, and 5). This need might be addressed by a tool that combines reverse-engineering with sketching or drawing. Furthermore such a tool might be able to reduce the barriers to iterating and rendering drawings, aiding the transition to design reviews, explaining to stakeholders and customers, and creating documentation (scenarios 4, 6, 7, and 9).

Levels of abstraction

When understanding existing code, designing/refactoring, and onboarding (scenarios 1, 3, and 5) developers need to understand both the *microscopic* details of the code and the *macroscopic* conceptual structure. The microscopic level of abstraction includes the mechanics of classes and methods, which can be examined in the text of the code or reverse-engineering tool. The macroscopic level of abstraction includes concrete higher-level concepts such as modules and systems and conceptual structures that are not manifest directly in the code. No current view conveys both levels of abstraction simultaneously. Developers might benefit from an interactive visualization that allowed them to explore the microscopic details while remaining oriented in the macroscopic structures. Such a tool might be particularly effective if combined with sketching capabilities as suggested above.

Staying oriented using spatial memory

A visualization that was spatially stable, yet up-to-date with the evolution of the code, could help a developer stay oriented while understanding existing code and designing/refactoring (scenarios 1 and 3). If the visualization were shared among the development team then ad-hoc meetings, design reviews, and especially onboarding (scenarios 2, 4, and 5) could benefit from the common ground that it would create, which might be enhanced by using it as hallway art (scenario 8).

CONCLUSION

This study tried to answer two basic questions on software visualizations: *how* and *why* developers use diagrams. Our results indicate that in most cases, informal notation was used to support face-to-face communication and that current tools were not capable of supporting this need because they did not help developers externalize their mental models of code. Our results also suggest some ways in which the role of diagrams in software development differs from other engineering disciplines. For example, not only is code lacking many spatial features to support its rendering, but it also lacks any conventional level of abstraction. Instead, developers reported that the level of abstraction differs with every conversation and even within a conversation.

Our approach of performing a series of interviews and then validating our results with a large-scale structured survey gives us confidence in our findings. However, this technique offered only a static, introspective, and retrospective view. There is much more to discover in observing the active production of visualizations, especially in moments of conflict that reveal mismatches in developers' mental models. We hope to explore this in our future work.

REFERENCES

1. M. W. Alibali, M. Bassok, K. O. Solomon, S. E. Syc, and S. Goldin-Meadow. Illuminating mental representation through speech and gesture. *Psychological Science*, 10:327–333, 1999.
2. V. Bellotti and S. Bly. Walking away from the desktop computer: Distributed collaboration and mobility in a product design team. In *Proc. CSCW*, pp. 209–218.
3. M. Cherubini and J. van der Pol. Grounding is not shared understanding: Distinguishing grounding at an utterance and knowledge level. In *CONTEXT 2005*.
4. H. H. Clark and E. F. Shaeffer. Contributing to discourse. *Cognitive Science*, 13:259–294, 1989.
5. U. Dekel. Supporting distributed software design meetings: What can we learn from co-located meetings? In *Proc. HSSE 2005*. ACM.
6. P. Dillenbourg, D. Traum, and D. Schneider. Grounding in multi-modal task-oriented collaboration. In *Proc. EUROAIED 1996*, pp. 415–425.
7. E. Do and M. D. Gross. Reasoning about cases with diagrams. In J. Vanegas and P. Chinowsky, editors, *ASCE*, pp. 314–320, 1996.
8. S. Goldin-Meadow. *Hearing gesture: How our hands help us think*. Belknap Press, Cambridge, MA, USA, 2003.
9. J. Heiser, B. Tversky, and M. Silverman. *Visual and spatial reasoning in design III*, chapter Sketches for and from collaboration, pp. 69 – 78. 2004.
10. K. Henderson. On Line and On Paper: Visual Representations, Visual Culture, and Computer Graphics in Design Engineering. Cambridge, MA: MIT Press, 1999.
11. J. D. Herbsleb. Metaphorical representation in collaborative software engineering. In *Proc. WACC 1999*, pp. 117–126. ACM.
12. M. Hertzum and A. M. Pejtersen. The information seeking practices of engineers: Searching for documents as well as for people. *Information Processing and Management*, 36(5):761–778, 2000.
13. E. Hutchins. Cognition in the Wild. MIT Press, Cambridge, MA, USA, 1995.
14. A.J. Ko, H.H. Aung, B.A. Myers (2005), Eliciting Design Requirements for Maintenance-Oriented IDEs: A Detailed Study of Corrective and Perfective Maintenance Tasks, In *Proc. ICSE 2005*. ACM.
15. J. H. Larkin and H. Simon. Why a diagram is (sometimes) worth ten thousand words. *Cognitive Science*, pages 65–99, 1987.
16. T. D. LaToza, G. Venolia, and R. DeLine. Maintaining mental models: a study of developer work habits. In *ICSE '06: Proc ICSE 2006*, pp. 492–501. ACM.
17. D. W. McDonald and M. S. Ackerman. Just talk to me: A field study of expertise location. In *Proc. CSCW 1998*, pp. 315–324.
18. D. E. Perry, N. A. Staudenmayer, and L. G. Votta. People, organizations and process improvement. *IEEE Software*, pages 36–45, July 1994.
19. B. A. Price, R. M. Baecker, and I.S. Small. A Principled Taxonomy of Software Visualization. *Journal of Visual Languages and Computing*, 4(3):211-266, 1999.
20. M. Suwa, J. Gero, and T. Purcell. Unexpected discoveries and s-invention of design requirements: Improving vehicles for a design process. *Design Studies*, 18(4):539–567, 2000.
21. B. Tversky, M. Suwa, M. Agrawala, H. J, C. Stolte, P. Hanrahan, D. Phan, J. Klingner, M. Daniel, P. Lee, and J. Haymaker. Human behavior in design: Individuals, teams, tools. In *Sketches for Design and Design of Sketches*, pp. 79–86. Springer, 2003.
22. B. Tversky. Spatial schemas and abstract thought. In *Spatial Schemas in Depictions*, pp. 79–111. MIT, 2001.