Interaction Design and Activity Theory: Designing for Social Code Review

by
Randy Souza
December 2010

Presented to the
School of Information Arts and Technologies
University of Baltimore

In Partial Fulfillment
of the Requirements for the Degree
Master of Science

Approved by: _____
Kathryn Summers, IDIA Program Director

Abstract

This thesis applies activity theory to interaction design of a code review tool. The purpose of this project was two-fold: To understand the practice of code review, and to gain insight into the value of activity theory as a framework for interaction design. Decades of research have shown that code reviews are a valuable software engineering tool. But recent studies suggest that many software development organizations do not use code reviews to their full potential. While code review is often acknowledged as a social practice, few studies address the social, cultural, and historical context surrounding programmers as they conduct reviews. This paper uses activity theory to analyze these contextual factors within a department in a commercial software development organization. Code review practice is found to be informal, highly situated, and multi-motivated. Based on these findings, interaction design concepts are proposed for a Web-based tool that supports collaborative construction, execution, and resolution of code reviews.

Acknowledgments

Thank you to Kathryn Summers, Nancy, Kaplan, Stuart Moulthrop, and the School of Information Arts & Technology for the knowledge and opportunity to (finally!) complete this project.

Thank you to M., K., J., A., G., C., L., P., and S. for sharing your code review experience.

Thank you to Matt, Josh, and Luke for letting me build on your work.

Eternal gratitude to Naida Rosenberger for love, patience, support, and everything else. This is for you

Table of Contents

Table of Figures

Introduction

"Reviews are about the people reviewing" (Karlsson, 2010, p. 29).

The practice of code review[1] in software development has been actively studied since the mid-1970s. Researchers and practitioners extoll code review as a driver of software quality and methodological rigor. Reports have scrutinized the management and conduct of reviews, prescribing detailed procedures and suggesting varied techniques. But, as Matthias Karlsson succinctly concludes in the epigraph above, code reviews succeed or fail based on the activities of the programmers who write and review code. Supporting code review requires understanding and supporting a development team's collective goals and actions. Interaction design provides a useful perspective on this effort: Studying human activity in order to create or improve products that support communication, socialization, and work is one of the central components of interaction design (Sharp, Rogers, & Preece, 2007).

This project originated in the author's personal experience as a programmer struggling to integrate code review into regular practice. Reviews involved varied artifacts, including email, face-to-face communication, version control, and a very simple Web-based review tool, none of which truly fit the task at hand. Coordinating and tracking review activity was a manual process based on uncertain social cues—for example, how long was it appropriate to wait for a reviewer to provide feedback before following up? This work began as a response to these challenges, with three goals:

1. To better understand code review practice;

1 While code review is most often categorized as a subset of software inspection, this paper uses the terms interchangeably to mean the review of software source code by one or more programmers other than the original author of the code..

2. To design an improved code review tool;
3. To investigate activity theory as a design tool.

This paper begins with a literature review surveying code review research. Studies that highlight programmer practices are given special attention. After covering the evolution and application of code review, an interaction design perspective is applied to existing code review research. The emergence and growth of interaction design has led some researchers to search for a theoretical foundation. The second half of the literature review investigates one interaction design perspective—activity theory. No studies applying activity theory to code review practice were found, so the literature review concludes with hypothetical applications of activity theory to some key concepts uncovered in code review research.

The second section of this report describes a research project conducted to study the real-world use of code review and artifacts that mediate code review. Contextual interviews structured on insights from the literature review were conducted with nine developers. Data from these interviews were analyzed using models from activity theory, leading to a set of implications for designs meant to support code review.

This thesis concludes with discussion of interaction design concepts based on these implications. The design suggests an architectural approach based on flexibility and emergent integration. User interface components are oriented toward communication, coordination, and shared learning. Context is a pervasive design concept, both for code authors, who must provide reviewers enough context to focus the review, and for reviewers, who must navigate through a review and its supporting material.

Literature Review

Software Inspection

Software inspection research originated at IBM in the early 1970s as part of an overall software quality movement (Fagan, 1976). Fagan (1976) described inspections as "a *formal*, *efficient*, and *economical* method of finding errors in design and code" (p. 189). Fagan (1976) presented software inspection as complementary to other quality assurance techniques like automated testing. Iisakka and Tervonen (2001) found that computer-executed unit, functional, and integration tests efficiently found one set of problems, while human code review identified a different set of defects. Wood, Roper, Brooks, and Miller (1997) recommended a project-sensitive combination of techniques, finding that the nature of the program and the types of defects in the code impacted the effectiveness of inspections, functional testing, and static analysis. Wiegers (1995) recommended inspections as a way to identify module interface errors, excessive complexity, unnecessary functionality, and badly structured code, but reflected that inspections were poor at identifying performance problems.

Ten years after his seminal introduction, Fagan (1986) reflected on the growth of software inspection. Companies expanded inspection beyond its initial focus on code quality to assess freedom from defects after deployment, ease of installation, documentation, portability, maintainability and extensibility, and fitness for use (Fagan, 1986). Nielsen (1994) surveyed seven software-inspection-inspired empirical methods for assessing the usability of graphical user interfaces. Biffl (2000) studied the impact of inspections on developers' estimates and found that inspections led to more accurate future estimates.

Benefits of code review.

The benefits of software inspection are covered by research in two different settings. In industry, inspection research analyzed the impact of inspections on quality and productivity. In academia, inspections were studied as teaching tools. Glass (1999) stated that "inspections, by all accounts, do a better job of error-removal than any competing technology (that is, inspections tend to find more errors), and they do it at lower cost" (p. 17).

Measuring software quality is a hard problem, impacted by intertwined variables including developer ability, technology choice, requirements quality, design quality, and code quality (Kemerer & Paulk, 2009). Inspections, specifically code reviews, have been shown to increase software quality in multiple ways. Code review was initially presented as a way to remove defects, reduce costs, and increase productivity (Fagan, 1976). In empirical studies, code review reduced defects—described as any requirements not satisfied when a project moves between phases—by 60% to 90% (Fagan, 1986; Tyran & George, 2002). Fagan (1976) associated a 38% reduction in production errors with code review. Rombach et al. (2008) reported that inspections were characteristic of successful software organizations, which mandated code review, used well-defined review techniques, had an internal inspection champion, and trained developers in inspection.

Code review reduced costs by finding defects early in the development process, when they required less time to fix (Fagan, 1976). Fagan (1976) associated a 25% savings in programmer resources with code reviews. Hewlett-Packard credited code review with cost savings of $21 million (Johnson, 1998). In a longitudinal study, NASA's Jet Propulsion Laboratory estimated $25,000 savings per code review session (Tyran &

George, 2002).

Programmers also benefited from code reviews. Fagan (1976) saw a 23% increase in programmer productivity due to the early detection of defects through code review. Code reviews improved productivity through "feed-forward," preparing testers and maintenance programmers for the types of defects a program may include (Fagan, 1986). Within an organization, participants learned about different products by reviewing code for those products (Fagan, 1976). Participants in code review sessions experienced educational benefits, learning alternative solutions to problems that other reviewers had solved in different ways (Trytten, 2005; Tyran & George, 2002). Johnson (1998) suggested occasionally reviewing high-quality code to take advantage of the learning benefit that inspections provided.

Recent advances in development tools and methodologies shifted the focus of code review studies from defect reduction to maintainability. New tools, like integrated development environments, and language features, like static typing, eliminated entire classes of bugs that inspections once caught, limiting the ROI of reviews by reducing the number of defects to find (Siy & Votta, 2001). Researchers began to investigate the impact of refactoring—improving the structure of code without changing the observable behavior (Mäntylä & Lassenius, 2006). Mäntylä and Lassenius (2009) suggested dividing defects found in code reviews into two classes: Functional defects, which caused the code to fail, and evolvability defects, which made the code less compliant with standards, more error prone, or harder to understand and change. Despite the early focus on functional defects (Fagan, 1976; Fagan, 1986), later code review research showed that the majority of defects uncovered in inspections were evolvability defects (Mäntylä & Lassenius,

2009; Siy & Votta, 2001). Mäntylä and Lassenius (2009) suggested that evolvability defects were more important for companies that develop in-house or commercial software, due to costs associated with ongoing maintenance of the code base. Siy and Votta (2001) associated the increased attention on evolvability defects with a change from immediate cost savings to future cost savings as the basic framework for evaluating the value of reviews.

In academia, code review was shown to improve learning outcomes in introductory programming classes, where students who reviewed other students' assignments received significantly higher grades (Hundhausen, Agrawal, & Ryan, 2010; Reily, Finnerty, & Terveen, 2009). Turner, Quintana-Castillo, Pérez-Quiñones, and Edwards (2008) reported that code review helped teach complex, abstract concepts by engaging students in evaluation—one of the highest level skills in Bloom's taxonomy of learning. The cooperative nature of code review led to more active learning in a study by Wang, Li, Collins, and Liu (2008). Trytten (2005) used academic code review to build teamwork skills and give students industrial experience. This approach was extended by Hundhausen, Agrawal, Fairbrother, and Trevisan (2009) to support a studio-based course that required students to apply industrial code review.

Limitations of code review.

Despite near-universal acknowledgment that code review is valuable, development teams reported limited adoption (Denger & Shull, 2007; Glass, 1999; Johnson, 1994). Denger and Shull (2007) reported that programming teams that avoided or abandoned inspections experienced difficulty connecting review results to quality in shipping products, mismatches between assumptions in review techniques and team

expertise, and an unclear relationship between reviews and day-to-day work responsibilities. Rombach et al. (2008) found that even among organizations that had adopted software inspection, reviews were not done in all development phases—for example, only 30% of companies did code reviews. Transaction costs—money, schedule time, and programmer motivation lost due to the effort required to prepare, schedule, and conduct inspections—further limited the adoption of code review (Iisakka & Tervonen, 2001; Trytten, 2005; Tyran & George, 2002).

Social dynamics also affected the impact of code reviews. To paraphrase Trytten (2005), "group" does not imply "team." Tyran and George (2002) concluded that problems related to group dynamics were evident in code review teams: Inspections suffered from dominant behavior by some members, sidetracking during face-to-face meetings, information overload when review scope was large, and participants forgetting or neglecting to raise issues. Johnson (1994) reported similar problems, including insufficient preparation by reviewers, domination of meetings by moderators, digression during meetings, and ego and personality conflicts. Group dynamics also led to social loafing, a phenomenon where people work less diligently in a group than they would individually (Trytten, 2005). While Trytten's (2005) report focused on academic code reviews, Iisakka and Tervonen (2001) described social loafing in industry: Authors were frustrated when reviewers did not prepare adequately and reviewers were frustrated when authors did not follow up on suggestions. Inspections of all types are inherently criticisms, so review teams needed to accommodate power imbalances between reviewers and authors, particularly when authors expressed a sense of ownership over the code (Iisakka & Tervonen, 2001).

Johnson (1998) associated low adoption of code review in industry with failure to embrace changes in technology (the emergence of the Web), organization (distributed development), and research (computer supported cooperative work). To address these limitations, Johnson (1998) proposed:

1. Integration of inspection and development processes to connect reviews with day-to-day work;
2. Asynchronous review in place of formal meetings to reduce waste associated with meetings and allow more frequent reviews;
3. Emphasizing learning over defect detection to build programmer skills and preclude repeat defects;
4. Creation of organization-wide review knowledge bases to streamline inspection;
5. Outsourcing review to consultants who teach the development team best practices;
6. Computer mediation to reduce overhead and support analysis and reflection;
7. Increasing the number of reviewers per project to improve coverage.

Evolution of the code review process.

When and how to review code has received more research attention than any other topic. Kemerer and Paulk (2009) reported that there is no consensus on the optimal timing for reviews, though Laitenberger and DeBaud (2001) demonstrated and recommended opportunities for review throughout the software development life cycle. Porter, Siy, Mockus, and Votta (1998) showed that more material in a review yielded more net defects found. Certain functional types (e.g., a parser versus a symbol table in a compiler) were more likely to contain defects, and therefore more important to review,

than others (Porter, Siy, Mockus, & Votta, 1998).

Fagan (1976; 1986) described six stages for any review.

1. During the planning stage the code author collected the material to be reviewed, ensured that the material was ready to inspect, selected participants, and determined a location and time for the review (Fagan, 1976).
2. In the overview stage the author assigned roles, shared the goals of the inspection, and identified the portions of the material under review (Fagan, 1976).
3. During the preparation stage reviewers read the material to become familiar with the code, but were discouraged from looking for defects (Fagan, 1976).
4. During the inspection stage the team held a face-to-face meeting to step through the code, identify defects, and classify defects by type and severity (Fagan, 1976). Fagan (1986) cautioned against determining solutions during this meeting.
5. In the rework stage, the code author evaluated the list of defects and made repairs (Fagan, 1976).
6. Finally, during the follow up stage, the author verified the rework with the original review team (Fagan, 1976).

Subsequent research on the code review process included most or all of Fagan's stages, but generalized them into preparation (Fagan's planning through preparation stages), collection (Fagan's inspection stage), and repair (Fagan's rework and follow up stages) (Porter, Siy, Mockus, & Votta, 1998). Sauer, Ross, Land, and Yetton (2000) refined the common code review process, suggesting a discovery stage conducted by multiple independent expert reviewers, a collection stage conducted by only the author, and a discrimination stage conducted by an expert pair of reviewers. Defects were identified in

the discovery stage, collected in the collection stage, and assessed in the discrimination stage (Sauer, Ross, Land, & Yetton, 2000).

Tailoring the inspection process was proposed as a way to introduce code review to a development organization or to revive waning interest in reviews (Denger & Shull, 2007). Rather than relying on prescribed preparation, collection, report steps, TAQtIC (Tailoring Approach for Quality-Driven Inspections) was "highly customizable, so the resulting inspection techniques best fit current context characteristics, are integrated with other development activities, and focus clearly on improving final-product qualities" (Denger & Shull, 2007, pp. 79-80). Teams that implemented TAQtIC found more high-priority defects and experienced greater programmer acceptance of inspections (Denger & Shull, 2007).

A constant in the preparation stage was the identification of roles. Three roles are common throughout the literature: The code author(s), the reviewer(s), and a meeting or review moderator. Less common inspection roles included tester (Fagan, 1976), designer (Fagan, 1976), producer (Stein, Riedl, Harner, & Mashayekhi, 1997), and recorder (Wiegers, 1995). Fagan (1986) called the moderator the most critical role, and characterized the ideal moderator as a player-coach throughout the inspection process. Mäntylä and Lassenius (2009) reported that certain classes of errors were only found by experienced reviewers. Sauer, Ross, Land, and Yetton (2000) suggested that development organizations could improve code review output by selecting expert reviewers for inspection teams, training reviewers, providing review aids to build experience, and growing inspection groups until performance began to decline.

Sauer, Ross, Land, and Yetton (2000) applied the behavioral theory of group

performance to code review roles. The behavioral theory of group performance suggested that group effectiveness would be determined by the cumulative experience of the participants (Sauer, Ross, Land, & Yetton, 2000). This hypothesis was supported, leading Sauer, Ross, Land, and Yetton (2000) to conclude that individual expertise was the most important factor impacting review effectiveness. Similar findings were reported by Porter, Siy, Mockus and Votta, (1998) and Mäntylä and Lassenius (2009). Sauer, Ross, Land, and Yetton (2000) identified plurality effects in inspections: When a majority of participants agreed on a defect, that defect was included in the review report. Without a majority, agreement from two participants sufficed to identify a defect as important (Sauer, Ross, Land, and Yetton, 2000).

Contrary to early guidelines (e.g., Fagan, 1976), Votta (1993) found that in the collection stage asynchronous reviews, where reviewers independently identified defects and communicated them to the author, were at least as effective as synchronous meetings. Votta (1993) concluded that the transaction costs incurred while scheduling meetings made asynchronous inspection meetings more valuable than face-to-face meetings. Perpich, Perry, Porter, Votta, and Wade (1997) also recommended asynchronous reviews to ameliorate schedule bottlenecks, which were found to be worse in large teams and when geographical and temporal distance increased (significant time differences between teams made synchronous review meetings impractical). Perpich, Perry, Porter, Votta, and Wade (1997) suggested that asynchronous reviews were effective because they did not disrupt existing workflows, which made them appealing to developers and reduced transaction costs. Kelly and Shepard (2003) investigated individual versus group inspection techniques, and found minimally significant gains from face-to-face meetings.

However, distinct findings uncovered during face-to-face meetings were classified as easy to identify, which further limited the value of face-to-face review (Kelly & Shepard, 2003). Kelly and Shepard (2003) concluded that an author reviewing findings individually may be as effective as the author reviewing findings in a group, echoing the conclusions of Votta (1993). Votta (1993) and Sauer, Ross, Land, and Yetton (2000) reported that the primary benefit of a face-to-face inspection meeting was reduction of false positive defects.

The impact of developer skill on code review.

Changes to the code review process, such as the stages used, or a synchronous versus asynchronous collection stage, were shown to have minimal impact on defect detection rates (Porter, Siy, Mockus, & Votta, 1998). Porter, Siy, Mockus, and Votta (1998) concluded that improvements to reviewer skills and defect detection techniques had a greater impact on efficiency than process changes. Porter, Siy, Mockus, and Votta (1998) argued that reviewers had such a strong influence on inspection effectiveness that identification of reviewer skill should become an element of software team management. Reviewer skill was related to experience conducting reviews and to programmer training (Fagan, 1976; Porter, Siy, Mockus, & Votta, 1998; Sauer, Ross, Land, & Yetton, 2000). Devito Da Cunha and Greathead (2007) characterized different programming phases as requiring different activities, where each activity required different skills. A 10:1 difference in skill between programmers was found in certain tasks (Devito Da Cunha & Greathead, 2007). Skill variance in code reviews was investigated by Kemerer and Paulk (2009), who reported that code review ability did not correlate with programming ability, but that reviewer effectiveness was a function of reviewer experience, the technology

used in the review, and the effort expended by the developer. Similarly, Uwano, Monden, and Matsumoto (2008) found that code review ability did not correlate with design review ability, and vice versa.

Despite the rich collection of literature studying code review, sociological aspects such as programmer learning and development have received little attention (Iisakka & Tervonen, 2001). In an investigation of qualitative aspects of code review, Kelly and Shepard (2002) reported that "observations suggest that the success of inspections, both traditional and non-traditional, is influenced strongly by soft issues" (p. 7). According to Kelly and Shepard (2002), reviewer behavior was difficult to shape, and experienced reviewers tended to use their own familiar inspection techniques, even when told to use a structured technique in an academic setting. Given these behavioral observations, Kelly and Shepard (2002) suggested a set of people-centric maxims that may serve as guidelines for code review:

1. Specify the goals of the inspection, defining terms and indicating what is being reviewed and what type of feedback is desired.
2. Identify tradeoffs in reviewer skills, such as domain expertise versus implementation language expertise.
3. Clean up the product to be inspected. Automated tools may fix style issues that can distract reviewers from functional defects.
4. Use a structured inspection technique.
5. Build a process based on inspections. The infrastructure available must identify goals, timing, and participants for an inspection.
6. Give inspectors responsibility and authority. Make the author available to

reviewers. Divide code among reviewers to foster a sense of ownership.

7. Ensure that inspectors have the time to inspect. Schedules must accommodate inspections.
8. Use metrics cautiously when assessing the effectiveness of an inspection activity. Metrics may be skewed by inconsistent definitions of defects, inconsistent granularity between reviewers, variations in the severity of defects, and disagreement over whether a finding is a defect.

Code reading techniques.

To accommodate varying skill levels among programmers, researchers have proposed and evaluated multiple defect detection techniques. Deimel (1985) was one of the first to suggest that program reading be included in the set of skills that programmers are taught. Deimel (1985) identified a range of types of program reading, including technical literature, new programming languages, previously written code, teammates' code, and new code. Basili (1997) pointed out that reading precedes writing in natural language acquisition, but that programmers are taught first to write code and then, if at all, to read code. Basili (1997) suggested that effective code reading required that the reader understood that code reading is effective and that the reader used a sufficiently well-defined reading technique.

In an eye-tracking study that investigated code reading technique, Uwano, Nakamura, Monden, and Matsumoto (2006) discovered a relationship between reviewer ability and review rate. Eye scan patterns were correlated with cognitive effort—more effective reviewers took more time to scan the code and then back-tracked to identify defects (Uwano, Nakamura, Monden, & Matsumoto, 2006). Less effective reviewers

skipped or shortened the scan period, and started looking for defects right away (Uwano, Nakamura, Monden, & Matsumoto, 2006).

Rombach et al. (2008) found that 40% of companies primarily used ad-hoc code reading: Reviewers read code without using any specific reading guidelines. Ad-hoc reading was described as popular, but ineffective unless reviewers were highly experienced (Laitenberger & DeBaud, 2001).

Aside from ad-hoc reading, checklist-based reading was the most popular technique reported by Rombach et al. (2008). Checklist-based reading was prescribed by Fagan (1986) as a way to guide reviewers in what to look for during an inspection. Parnin, Görg, and Nnadi (2008) expanded this definition to include guidance on how to classify defects. In-house coding standards—a specific type of checklist including rules for code structure and definitions of common terms—were identified by Kelly and Shepard (2002) as a way to review house style. Checklist-based reading was criticized as either too fine-grained to catch design defects or too course-grained to catch detailed statement-level bugs (Kelly & Shepard, 2002). Kelly and Shepard (2002) also pointed out that checklist-based reading risked causing tunnel vision in developers, who only reported defects included on the checklist. Laitenberger and DeBaud (2001) claimed that checklists lacked adequate context, as they could only be based on past projects and could not help programmers understand current code.

Reading by stepwise abstraction was highlighted by Deimel (1985), who suggested that to totally comprehend a program an inspector had to understand it at multiple levels of abstraction. In reading by stepwise abstraction, the inspector read a sequence of code statements, abstracted the function those statements provided, and

repeated the process until the entire program had been abstracted (Basili, 1997). At this point the abstracted view of the program was compared with the requirements, and deviations were flagged as defects (Basili, 1997). Reading by stepwise abstraction was shown to be a challenging technique to learn, as programmers had to build different reading skills for different levels of analysis (Deimel, 1985). Kelly and Shepard (2002) reported a similar challenge with task-directed inspection, which increased reviewer effectiveness but to a degree that was highly dependent on reviewer experience.

Scenario-based reading was introduced as a meta-technique that applied different reading techniques to different project scenarios (Basili, 1997). A reading technique could be selected by assessing the project across dimensions including:

- The goals of the review, for example, to find bugs, to understand performance, or to evaluate maintainability;
- The context of the review, for example, the project phase or the availability of certain inspectors;
- The type of input under review, for example, code, requirements, or tests;
- The type of output desired;
- The degree of rigor required;
- The perspective of the reviewer;
- Qualities of the product, for example, the age of the code or level of test coverage;
- Qualities of the overall process, such as reviewer training (Basili, 1997).

Scenario-based reading suggested that there was no single definition of quality (Basili, 1997). This position was adopted separately by Basili et al. (1996) and Laitenberger and DeBaud (2001) in discussions of perspective-based reading.

Perspective-based reading accepted that in the absence of a single definition of quality code should be reviewed from various points of view (Basili et al., 1996). In perspective-based reading the author defined a perspective for each inspector, such as tester, designer, or user (Laitenberger & DeBaud, 2001). A suite of guidelines was distributed for each perspective, and reviewers scoped their comments to those guidelines (Laitenberger & DeBaud, 2001). Perspective-based reading resulted in more in-depth analysis because each reviewer could focus on a narrower view of the code (Basili et al., 1996). Perspective-based reading increased motivation among readers by reducing perceived duplication of effort: Each reviewer was unlikely to overlap another reader's findings (Denger & Shull, 2007). However, perspective-based reading required larger review teams to ensure adequate coverage (Basili et al., 1996).

Thelin, Runeson, and Wohlin (2003) expanded on the user role in perspective-based reading in their description of usage-based reading. Usage-based reading took a set of use cases as the starting point, and asked reviewers to focus on defects that had the most negative impact on end users' perceptions of product quality (Thelin, Runeson, & Wohlin, 2003). In inspections of requirements, Thelin, Runeson, and Wohlin (2003) found that usage-based reading found more defects more quickly than checklist-based inspection of the same documents. For code review, checklist-based reading found more low-level defects than usage-based reading, in part because checklists worked at a low level of abstraction (Deimel, 1985; Thelin, Runeson, & Wohlin, 2003).

Pair programming has been classified as a reading technique, in the sense that it is a real-time review of newly-written code (Williams & Kessler, 2000). Evaluations of pair programming have shown benefits similar to code review—higher efficiency and higher

quality (Williams & Kessler, 2000). The real-time code review provided by the observer on a pair programming team was classified by Williams and Kessler (2000) as one of the key elements in the success of pair programming.

Code review tools.

Two of the most frequently cited findings from the code review literature are that asynchronous and synchronous reviews are at least equally effective (Votta, 1993), and that reviewer skills and techniques have a greater impact on success than changes to the review process (Porter, Siy, Mockus, & Votta, 1998). These findings, combined with the emergence of networked communications platforms and increasingly distributed programming teams, have led to the development of software tools to support code review. These tools include research projects, commercial products (for example, Smart Bear Software Code Collaborator or Atlassian Crucible), and open source applications (for example, Codestriker, Review Board, or Rietveld). This section reviews tools documented in scholarly literature.

ICICLE (Intelligent Code Inspection Environment in a C Language Environment) used groupware technology to enhance defect collection and review meetings in a process following Fagan's (1976) guidelines (Brothers, Sembugamoorthy, & Muller, 1990). ICICLE integrated static analysis tools to help reviewers find routine errors, a shared code window to focus review meetings, and shared comment proposal dialog boxes to support group-wide acceptance of issues (Brothers, Sembugamoorthy, & Muller, 1990). Brothers, Sembugamoorthy, and Muller (1990) reported that programmers altered their review practice by using ICICLE's static analysis feature to refine code before submitting it for review.

CSRS (Collaborative Software Review System) was designed to address problems that made it "difficult to effectively carry out review . . . and difficult to measure the process and products of review in such a manner as to understand review, compare review experiences across organizations, and improve the process" (Johnson, 1994, p. 115). CSRS used hypertext to link between review artifacts, integrate checklists, store issues, suggestions, and comments, and collect votes on which defects should be fixed (Johnson, 1994). Reviewers were able to mark artifacts as reviewed, which gave authors visibility into review progress (Johnson, 1994). CSRS was instrumented with activity and result reports to help development teams reflect on inspection activity (Johnson, 1994).

Perpich, Perry, Porter, Votta, and Wade (1997) studied use of an intranet to support asynchronous code review. The tool centered on two artifacts: An inspection package and email notifications (Perpich, Perry, Porter, Votta, & Wade, 1997). The inspection package included the status of the review, diff-annotated source code, and general or line-specific annotations (Perpich, Perry, Porter, Votta, & Wade, 1997). The inspection package was created by the code author and updated by reviewers (Perpich, Perry, Porter, Votta, & Wade, 1997). Email notifications were used to inform participants of status changes, such as the availability of an inspection package (Perpich, Perry, Porter, Votta, & Wade, 1997). A feature that differentiated the approach studied by Perpich, Perry, Porter, Votta, and Wade (1997) from subsequent tools was the preservation of the moderator role independent of the author role. In Perpich, Perry, Porter, Votta, and Wade's (1997) study the moderator validated the author's repair plans for each defect, and was the only participant who could mark a review complete, ensuring

that all inspectors' comments received attention from the author.

Stein, Riedl, Harner, and Mashayekhi (1997) invented AISA (Asynchronous Inspector of Software Artifacts), a Web-based code review tool. Like the intranet tool described by Perpich, Perry, Porter, Votta, and Wade (1997), AISA used email to synchronize communication between review participants. AISA included design elements intended to preserve positive elements of face-to-face inspection meetings (Stein, Riedl, Harner, & Mashayekhi, 1997). Group decision support features allowed the team to preserve consensus—the feeling that decisions were made and agreed on collaboratively —and coordination—participants felt that there was a clear process to the inspection (Stein, Riedl, Harner, & Mashayekhi, 1997). Communication tools preserved values associated with face-to-face coordination, such as sustainable trains of thought and serial discussions between participants (Stein, Riedl, Harner, & Mashayekhi, 1997). A shared information space preserved visual cues and provided an inspection history (Stein, Riedl, Harner, & Mashayekhi, 1997). To encourage group decisions, participants were required to vote on each others' contributions; this feature was found to be cumbersome and unnecessary (Stein, Riedl, Harner, & Mashayekhi, 1997). Stein, Riedl, Harner, and Mashayekhi (1997) found that the extended time frame provided by AISA reviews led to deeper comments between participants, including rebuttals, examples, and citations.

Laitenberger and Dreyer (1998) investigated the perceived ease of use and usefulness of a Web-based Inspection Process Support tool (WIPS). WIPS supported the defect collection stage of an inspection by allowing reviewers to enter and classify defects (Laitenberger & Dreyer, 1998). During the review meeting, WIPS allowed participants to add, combine, and accept defects (Laitenberger & Dreyer, 1998).

Laitenberger and Dreyer (1998) evaluated WIPS with a questionnaire that assessed ease of use and usefulness; reviewers found WIPS easy to use and preferable to paper forms.

Tyran and George (2002) applied group support system technology as a mediating tool in face-to-face review meetings. In an experimental setting, some standard inspection meetings were augmented with chat windows to provide parallel conversation and a scrolling window of inspection results to provide group memory (Tyran & George, 2002). Control meetings inspected the same code but did not use the group support tools (Tyran & George, 2002). Tyran and George (2002) found that the supporting tools minimized some common meeting problems, like dominant behavior and sidetracking. Additionally, groups using the support tools reported more defects in the code they reviewed, but felt more information overload, and were subjectively less satisfied with the experience (Tyran & George, 2002).

Parnin, Görg, and Nnadi (2008) explored automated code inspection tools as a complement to code review. These tools proved useful for assessing design and code quality, in particular for maintainability defects that were overlooked by human inspectors focused on functional defects (Parnin, Görg, & Nnadi, 2008). However, automated inspection tools generated long lists of potential problems, most of which were false positives (Parnin, Görg, & Nnadi, 2008). To address the false positive issue, Parnin, Görg, and Nnadi (2008) designed a series of lightweight visualizations meant to provide additional information about defect warnings. The visualizations helped authors decide at a glance when a warning was likely to be a false positive, which sections of code had the most relevant warnings, and how warnings were spread throughout the code under review (Parnin, Görg, & Nnadi, 2008). While Parnin, Görg, and Nnadi (2008) did not apply their

visualizations to defect lists generated by human code reviews, they also did not provide any evidence that this would be challenging.

Meyer (2008) provided an experience report of code review distributed across a worldwide programming language team. Rather than implementing a code-review-focused tool, the team used a combination of collaboration tools to successfully replicate face-to-face review sessions (Meyer, 2008). Voice over IP telephony was used to communicate in real-time, chat filled in when participants could not connect or desired a side conversation, and a shared document repository and wiki allowed asynchronous review and documentation of inspection findings (Meyer, 2008).

Hedberg (2004) found that no code review tool had broken through in practical or research use. Tool research was criticized for not following advances in reading techniques, not supporting non-text artifacts, and focusing myopically on a single feature or theory (Hedberg, 2004). Hedberg (2004) classified code review tools into four generations, then proposed a set of guidelines for a new generation of tools that incorporated flexibility and integration across artifacts, process, and results. These "comprehensive tools" (Hedberg, 2004, p. 241) should support:

- Inspection of multiple artifact formats, including text and graphics, stored in multiple repository types, including file systems and source code control;
- Asynchronous or synchronous process, including integration with project management tools;
- Multifaceted results, including the severity, type, and resolution of defects (Hedberg, 2004).

Summary.

Software development research has explored increasing tension between emerging agile techniques and established engineering-driven methodologies (Beck, 2004). Code review research has engaged this tension: Fagan's (1976, 1986) pioneering research reported significant value from inspections but prescribed a formalized, standards-based process that organizations struggled to implement. Johnson (1998) and Glass (1999) separately challenged practitioners to reengineer inspections by emphasizing developer skills, embracing asynchronous activity, and integrating computer-mediated review tools. Multiple authors produced code review tools, but no tool achieved wide adoption (Hedberg, 2004). While context may play a critical role in successfully integrating code review into a development organization, few studies have addressed context in inspection practice (Denger & Shull, 2007). Design of new code review tools should address social, behavioral, and contextual factors (Iisakka & Tervonen, 2001; Kelly & Shepard, 2002; Denger & Shull, 2007).

Human-Computer Interaction and the Challenge of Context

Exploration of social, behavioral, and contextual issues in technology design is an ongoing theme in human-computer interaction (HCI) research. HCI research engaging the challenge of context in technology design and use emerged in three cycles: From human factors to human actors, from usability to user experience, and from user-centered design to activity-centered design.

From human factors to human actors.

This section takes its title from Bannon's (1991) widely-cited article, which criticized the then-dominant information processing model of HCI. HCI research evolved

from two precedents: A psychological program with roots in ergonomics and human factors studies during and after World War II, and work research dating to Taylor's scientific management (Kuutti, 1999). Researchers embraced cognitive psychology as a way to harden the science behind HCI, partly to counterbalance the dominance of computer science in information technology research and practice (Newell & Card, 1985).

Newell and Card (1985) attempted to develop an engineering-style theory that would support task analysis through calculation and formal models. This theory supported technology design with symbolic representations and mathematical models of users' tasks (Newell & Card, 1985). This line of research resulted in multiple information processing models, including the Model Human Processor (Card, Moran, & Newell, 1983), GOMS (Card, Moran, & Newell, 1983), the Keystroke-Level Model (Card, Moran, & Newell, 1983), ACT* (Newell & Card, 1985) and applications of Fitt's and Hicks' Laws (Raskin, 2000). Some of these models proved useful in specific design situations—Carroll (1997) reported that GOMS was effective when performance efficiency was critical to usability. Raskin (2000) recommended four quantitative methods, including GOMS and Fitt's Law, for analyzing a user interface.

Bannon (1991) argued that the information processing model of HCI had not proven effective for practical design. Bannon (1991) pointed out that terminology can give insight into a discipline, and that the terms used in the information processing model belied human goals and capabilities. For example, Bannon (1991) claimed that the term "users" neglected participation in work that reached beyond the bounds of the user interface, and that "human factors" reduced people to units of analysis equal to other

components of the system, measured by attention, memory and processing speed. Bannon (1991) criticized the information processing model for de-emphasizing the role of context —motivation, community, and setting—in favor of time consuming and costly laboratory studies.

Bannon (1991) did not prescribe an alternative to the information-processing paradigm, but recommended approaches that focused on human agency in relation to technology. Agency was described as "the ability to act and the need to act" by Kaptelinin and Nardi (2006), and as "the capability to make a difference" by Rose, Jones, and Truex (2005). In a study of ERP implementations, Rose, Jones, and Truex (2005) found that agency was critical, and that, contrary to the assumptions of information processing models, machine and human agency were intertwined, but not equivalent. In successful ERP projects, machines facilitated and constrained human behavior, but human intentions and actions—agency—shaped organizational success (Rose, Jones, & Truex, 2005).

From usability to user experience.

Elaborating on Bannon (1991), Carroll (1997) considered the evolution of HCI beyond cognitive psychology. Carroll (1997) characterized the growth of usability engineering as an evolutionary jump: Projects began to be managed toward explicit usability goals, users were involved throughout the design process, and the focus of design research moved from the laboratory to the field. Tools shifted from formal, rigorous modeling to discount usability testing and rapid prototyping (Carroll, 1997). Carroll (1997) described a second evolutionary step beyond usability engineering that involved increased focus on context—the wider design rationale behind a system. Design teams considered the discussions, debates, and tradeoffs that determined a design, and

created design tools like contextual inquiry and user models (Carroll, 1997).

Hassenzahl and Tractinsky (2006) positioned the evolution described by Carroll (1997) as the emergence of user experience, which "gained momentum in recent years, mostly as a countermovement to the dominant, task- and work-related 'usability' paradigm" (p. 91). Hassenzahl and Tractinsky (2006) characterized user experience as practitioner-driven, and attempted to establish a research agenda focused on three principles. First, user experience research should advance beyond instrumental studies of bounded tasks to address deeper human needs, such as surprise, diversion, stimulation, evocation, and identification (Hassenzahl & Tractinsky, 2006). Second, researchers should consider the affective and emotional impact of user experience (Hassenzahl & Tractinsky, 2006). Finally, researchers should investigate user experiences across time, and address the situated nature of technology use (Hassenzahl & Tractinsky, 2006).

While Hassenzahl and Tractinsky (2006) encouraged designers and researchers to take a wider view of HCI, they questioned the scope of their own recommendation: "Can designers exert enough control over all relevant elements in a way that a positive experience becomes certain? Or do we rather 'design *for* an experience', that is, to take experiential aspects into account while designing, without being able to guarantee a particular experience" (p. 95).

From human-centered design to activity-centered design.

The field of user experience has earned a prominent place in design, but is not without its own challenges. One important challenge was voiced by Norman (2005), who pointed out that many organizations with strong user experience teams continued to ship complex and confusing products. Norman (2005) echoed Hassenzahl and Tractinsky's

(2006) concern that the field of user experience may be focused on an overly ambitious target. A design team that enjoyed deep understanding of users' situations, motivations, and emotions was likely to produce good, but not great, design (Norman, 2005). Norman (2005) expressed concern that while user-experience-oriented designers were likely to have a rich user model, the model might lead to designs that serve one user over another or that do not accommodate advances in users' skills, and that "too much attention to the needs of the users can lead to a lack of cohesion and added complexity in the design" (pp. 16-17).

Activity Theory and Interaction Design

Successful interaction design requires consideration of human agency, context, and activity (Bannon, 1991; Norman, 2005). Activity theory satisfies these considerations, and has been applied as a theoretical foundation for interaction design. This section provides an overview of activity theory and discusses the role of activity theory in interaction design research.

Activity theory evolved within the Soviet school of cultural-historical psychology developed by Vygotsky and his followers, notably Luria and Leontiev (Kaptelinin & Nardi, 2006). Cultural-historical psychology was itself built upon Hegel's and Marx's explorations of dialectical materialism and the philosophical role of breakdowns and contradictions in society (Stetsenko, 2005; Vygotsky, 1978). This Marxist and Hegelian heritage supported a grounding principle in activity theory: That there is no separation between mind and behavior or between mind and society (Gay & Hembrooke, 2004). Beginning in the 1970s Soviet psychological research became more widely available outside of the U.S.S.R., and activity theory grew into an internationalized and

multidisciplinary theory (Engeström, 1999). Scandinavian studies of work practice leaned heavily on ideas from activity theory (Engeström, 1999; Kuutti, 1999). Activity theory was conceptually linked with disciplines including ethnomethodology, pragmatism, and systems theory (Engeström, 1999), ecological psychology (Bærentsen & Trettvik, 2002; Gay & Hembrooke, 2004), phenomenology (Gay & Hembrooke, 2004), and distributed computing and actor-network theory (Kaptelinin & Nardi, 2006). Carroll (1997) positioned activity theory as potentially the most theoretically rich application of these disciplines to HCI: "Activity theory shifts attention from characterizing static and individual competencies toward characterizing how people can negotiate with the social and technological environment to solve problems and learn, which subsumes many of the issues of situated and distributed cognition" (p. 512).

Principles of activity theory.

Activity theorists applied the concept of human activity as the theory's central explanatory idea (Engeström, 1999). Human activity in this context was defined as "doing in order to transform something" (Kuutti, 1996, p. 25). Stetsenko (2005) explained the conceptual impact of centering the theory on human activity: "In even broader terms, the development of the human mind is conceptualized as originating from practical transformative involvements of people with the world, and as a process that can be understood only by tracing its origination in these involvements and practices" (p. 74).

At the intrapersonal level, activity theory dealt with the relationship between subject (a human being) and object (Vygotsky, 1978). Object has been a challenging concept for activity theorists to explain and apply (Miettinen, 2005; Stetsenko, 2005). Leontiev introduced the idea of object-oriented activity as a way to explain motivation:

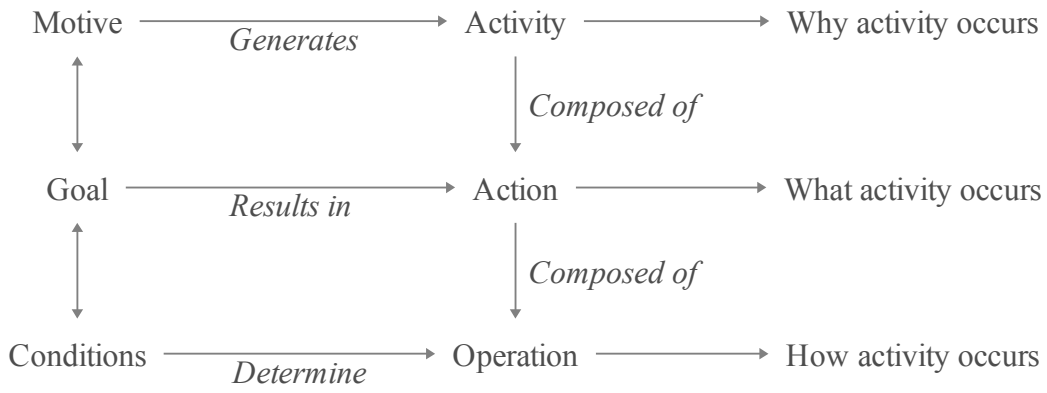
"the object of an activity is its true motive" (Leontiev, 1978, p. 62, quoted in Miettinen, 2005). Kuutti (1996) elaborated that transforming an object into an outcome—some visible transformation in the world or in the head—is what motivates the existence of an activity. According to Kuutti (1996) an object could be anything that may be shared for manipulation and transformation by participants in an activity; an object could be material, minimally tangible (such as a plan), or intangible (such as an idea). Kaptelinin, Nardi, and Macaulay (1999) pointed out that an object is not limited to something with physical, chemical, or biological properties—an object may be culturally or socially determined.

Vygotsky (1978) contended that object-oriented activity did not occur directly between the subject and object, but was mediated by artifacts—tools and signs. Vygotsky focused mainly on semiotic mediation—mediation through signs and symbols, principally spoken language (Wertsch, 2007). Vygotsky (1978) characterized signs and tools as analogous but not equal supporters of mediated activity, with tool-mediated activity externally-oriented and sign-mediated activity internally-oriented. For Vygotsky (1978), mediation allowed social and cultural activity to develop: "The use of signs leads humans to a specific structure of behavior that breaks away from biological development and creates new forms of a culturally-based psychological process" (p. 40). HCI research applying activity theory has concentrated on mediation by tools, though, as pointed out by Bødker (1996), artifacts may mediate activity but may also be the object of activity themselves. Bødker (1996) defined three roles that an artifact may play in object-oriented activity: The object may be the artifact (as in a spreadsheet), the object may be represented in the artifact, (as in a letter), or the object may be outside of the artifact (as

in a control panel). In any of these cases, a mediating artifact "works well in our activity if it allows us to focus our attention on the real object and badly if it does not" (Bødker, 1996, p. 149).

Kaptelinin and Nardi (2006) pointed out that subjects oriented toward objects at different levels in a hierarchical structure (Figure 1). At the highest level, the object represented a motive that fulfilled a conscious or unconscious need or desire, and generated an activity (Engeström, 1999; Wilson, 2006). These activities represented the central unit of analysis in activity theory, and explained the "why" behind a subject's activity (Bødker, 1996). However, human activities were not generally enacted directly at their motive (Kaptelinin & Nardi, 2006). Instead, according to Kaptelinin and Nardi (2006), activities were composed of actions, which were directed at goals. Stetsenko (2005) differentiated between motives and goals, characterizing motives as socialized and goals as individualized. Yet Stetsenko (2005) also illustrated that, while different, goals and their motive did not exist separately, but constructed and transformed each other. Kuutti (1996) claimed that the same action may be different when performed as part of a different activity: For example, a manager may present the same report differently when delivering the report to her team versus when delivering the report to senior management. Just as one action could change in the context of different activities, Bardram (1997) contended that a single action could be polymotivated. A polymotivated action occurred when multiple activities that shared the same goal temporarily overlapped (Bardram, 1997).

Just as activities were composed of actions, actions were composed of operations oriented toward the conditions surrounding an action (Engeström, 1999; Wilson, 2006).



Source: Engeström, 1999; Kaptelinin & Nardi, 2006; Kuutti, 1996; Wilson, 2005

Figure 1: The hierarchical structure of activity

Kaptelinin and Nardi (2006) described operations as generally unconscious "routine processes providing an adjustment of an action to the ongoing situation" (p. 62). Kaptelinin and Nardi (2006) stated that an action could transition to an operation, and become routine, and a previously unconscious operation could become an action when conditions broke down or deviated from what was expected. This relationship between goals and conditions was described by Norman (1993) as reflective (conscious) versus experiential (unconscious) cognition.

The creators of activity theory used the concepts of internationalization and externalization of activity to describe human development (Kaptelinin & Nardi, 2006). Transformations between levels of activity represented one type of development (Engeström, 1999). Vygotsky (1978) described internalization as a broader developmental process resulting in the "internal reconstruction of an external operation" (p. 56). According to Vygotsky (1978), internalization was the result of a series of developmental events that occurred over a period of time, as a subject first performed an activity with external assistance from people and artifacts, then reconstructed the activity internally,

and finally was able to perform the activity individually. Béguin and Rebardel (2000) characterized internalization as the process that results in development of expertise. The complement to internalization, externalization, was described as a gap in foundational activity theory research (Engeström, 1999; Stetsenko, 2005). Engeström (1999) described externalization as the development of new activities from breakdowns in existing activities. Béguin and Rebardel (2000) positioned externalization as a key creative process, where a search for solutions led to the invention of new artifacts to facilitate performance. In this sense, externalization as described in activity theory echoes Winograd and Flores' (1986) phenomenological approach to HCI: "new design can be created and implemented only in the space that emerges in the recurrent structure of breakdown. A design constitutes an interpretation of breakdown and a committed attempt to anticipate future breakdowns" (p. 78).

Historicity was presented as an important but under-researched corollary to activity theory's investigation of human development through internalization and externalization (Nardi, 2007). Engeström (1999) suggested that researchers identify historical cycles of an activity to understand the evolution of mediating artifacts around the activity. Quek and Shah (2004) saw the elicitation of historical activity as a strength of activity theory when applied to information systems development.

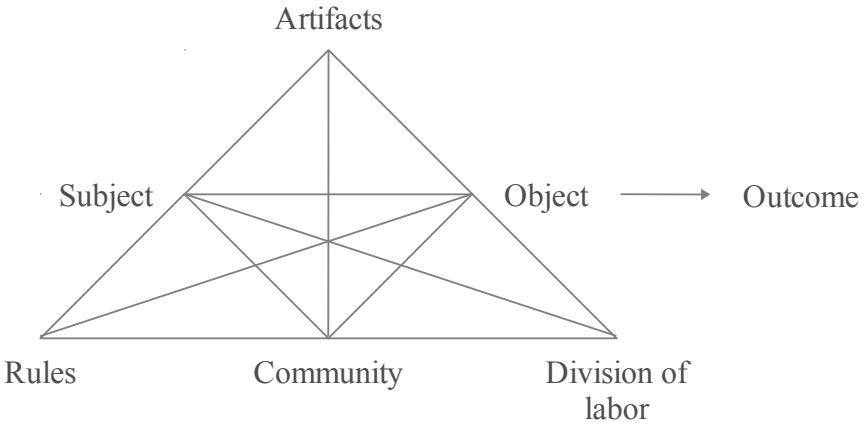
Collective activity and activity systems.

Vygotsky and his followers applied the principles that became activity theory as a socialized, cultural reaction to dualist psychology (Bakhurst, 2007). Engeström (1999) found that this led to a restrictive focus on individual activity, even if that activity was acknowledged as mediated by social and cultural artifacts. Engeström (1999) worried that

without a collective dimension the true motive of activity could be masked:

> The outcomes of my actions [preparing and presenting a speech] appear to be very limited and situation bound: a particular text, a momentary impact on the listeners. If this is all there is to gain, why did I bother to prepare the speech in the first place? Somehow, this level of representation hides or obscures the motive behind the actions. (p. 30)

To address this limitation, Engeström (1999) introduced the activity system as a model of collective activity and a way to connect individual actions to collective activity (Figure 2). Engeström's (1999) approach built upon the central activity theory model of a subject's activity directed at an object and mediated by artifacts, and added concepts describing mediated collective activity. Rules mediated a subject's relationships with the community and object-oriented activity through explicit or implicit norms, conventions, and relationships (Kuutti, 1996). Division of labor provided a way to describe the community's organization with regard to an object (Kuutti, 1996). Kaptelinin and Nardi (2006) used division of labor to explain socially distributed behavior—the common situation where an individual's actions are motivated by one object but directed toward another. In these cases the object was shared (but not necessarily identical), it drove and coordinated the actions of many people in an activity system (Miettinen, 2005). Miettinen (2005) described this phenomenon as "multi-motivated" activity: "Because an object of activity is complex and contradictory by its very nature, various individual aspirations, desires, and motives are attached to it and developed during its creation. In this sense, any collective activity is multi-motivated" (p. 64). Holland and Reeves (1996) described a variation of multi-motivated activity in an academic software project. Holland and



Source: Engeström, 1999

Figure 2: Engeström's model of the activity system

Reeves (1996) assumed that student teams performing the same activity—development of a software system for a class grade—would share an object of their activity—the software program under development. However, Holland and Reeves (1996) found very different motives between different teams—one team's object was the documentation associated with the project, while another team's object was elegant software code.

The role of theory in interaction design.

Human-computer interaction and computer supported cooperative work were two of the first fields to incorporate activity theory into Western research (Kaptelinin & Nardi, 2006). Bødker (1996) identified activity theory as particularly useful to HCI because activity theory supported instrument-level analysis without sacrificing context. Kuutti (1999) positioned HCI research as a central part of a broader challenge: How to handle contextuality in work research. Kuutti (1999) claimed that information systems were a major force driving changes in work, and that "[t]he transforming of work needs new kinds of support and poses new problems. Old conceptual tools are inadequate for solving them, and the IS debate is searching for new frameworks for doing precisely

that." (p. 171). Kuutti (1999) suggested activity theory as a new conceptual tool, mapping activity theory concepts to new research problems in IS and HCI.

Halverson (2002) described four roles that theory may play in HCI research:

- In an applicable role theory informs and guides design decisions;
- In an inferential role theory helps make inferences and predictions;
- In a rhetorical role theory provides a structure for naming things and mapping names to the real word;
- In a descriptive role theory provides a conceptual framework for describing and making sense of the world.

Activity theory has been applied primarily in rhetorical and descriptive roles. For example, Matthews, Rattenbury, and Carter (2007) used activity theory to provide common terminology in their retrospective of the design process for peripheral displays in ubiquitous computing. Fewer studies have investigated activity theory in an applicable role, through, as Winograd (2006) argued, theories have great value in shaping the understanding that a designer brings to a problem. Norman (2005, 2006) pointed out the importance of these applicable theories, arguing that organizations with strong design teams but no theoretical grounding continued to ship poorly-designed products. Norman (2005) called for a new approach to design based upon a shift in designers' theoretical orientation from people to activities.

HCI applications of activity theory.

Kuutti (1996) explored contributions that activity theory could make to HCI. First, activity theory showed that supporting work practices was more important than improving usability—people would adapt to bad UI design but would reject applications

that did not allow them to satisfy their goals and motives (Kuutti, 1996). This argument was reiterated by Norman (2006), who tweaked his own epigraph from *Things that Make Us Smart* (*"*People Propose, Science Studies, Technology Conforms", Norman, 1993, p.253)–stating "[n]one of this tools adapt to the people nonsense—people adapt to the tools" (p.15). Kuutti (1996) suggested that activity theory provided a social context for understanding HCI beyond individuals, a perspective that incorporated change and development over time, and ways to understand activity at technical, conceptual, and contextual levels.

Quek and Shah (2004) surveyed five activity-theory-based information system development methodologies. The majority of these methods focused on early stages of the development process, including domain analysis and requirements elicitation, and none covered the entire development life cycle (Quek & Shah, 2004). Multiple studies used activity theory principles like mediation and the hierarchical structure of activity to classify research data (Quek & Shah, 2004). Others used models based on activity theory, primarily Engeström's model of the activity system, to structure research results (Quek & Shah, 2004). Quek and Shah (2004) criticized the research for failing to apply all of the principles of activity theory and for providing minimal validation of the development methodologies. In Wilson's (2006) review of information systems studies and implementations, activity theory provided a deep understanding of the information needs and uses driving development projects.

Kaptelinin, Nardi, and Macaulay (1999) stated "[w]e see [activity theory's] main potential in supporting researchers and designers in their own search for solutions, in particular, by helping them to ask meaningful questions" (p. 32). Kaptelinin, Nardi, and

Macaulay (1999) introduced the activity checklist to help designers and researchers structure their search for solutions. The activity checklist was designed to make concrete the abstract principles of activity theory (Kaptelinin, Nardi, & Macaulay, 1999). Kaptelinin, Nardi, and Macaulay (1999) stated that because activity theory resists deconstruction—subjects, objects, artifacts, and sociocultural relationships must be considered as a whole—the activity checklist was meant to provide guidance on where researchers should focus their attention.

Barthelmess and Anderson (2002) applied activity theory to design of a software development environment. Activity theory was chosen as a way to contrast the personal nature of programming with the team-oriented nature of software development (Barthelmess & Anderson, 2002). The motivating object in Barthelmess and Anderson's (2002) study was defined as an evolving software system; the research objective was defined as integration of rules and division of labor into a software development environment. Barthelmess and Anderson (2002) focused their design on collaboration support: Providing programmers with a way to modify a document, observe modifications by others, and discuss the modifications within a single tool. Activity theory aided in evaluation of design work, showing that the design broke down by failing to align the programmer community towards a common object (Barthelmess & Anderson, 2002). Barthelmess and Anderson (2002) used activity theory to reflect on and improve the design process, an approach also employed by Collins, Shukla, and Redmiles (2002). Gay and Hembrooke (2004) similarly suggested activity theory as a tool for reflecting on the design process: "[d]esign is situated in a network of influencing social systems, and building any technological system is a socially constructed and negotiable process. By

using an interpretive flexibility framework, developers can understand stakeholders' various goals and apparent inconsistencies" (p. 28).

Halverson (2002) argued that a useful HCI theory required enough descriptive power to structure ethnographic research results. A number of studies used activity theory in ethnographic analysis. Bryant, Forte, and Bruckman (2005) conducted an activity-theoretic ethnography of Wikipedia participation. Miettinen and Hasu (2002) analyzed the design and marketing of a new medical research device. Activity theory was used by Turner, Turner, and Horton (1999) to structure and organize ethnographic data for requirements definition in a software firm. Collins, Shukla, and Redmiles (2002) also used activity theory to generate requirements, but focused on classifying mediating artifacts used in customer support activities. Gay and Hembrooke (2004) applied activity theory in an ethnographic analysis of the role that laptops played in mediating college students' collaborative learning activities.

Bærentsen and Trettvik (2002) repositioned the well-known HCI concept of affordance using activity theory. Norman (1999) expressed concern about misconceptions in the popular understanding of affordance within the HCI community. Bærentsen and Trettvik (2002) contended that these misunderstandings came from a conceptual framework that separated cognition from the physical and social world—in Gibson's original definition an affordance only emerged when organisms were acting in the environment. Activity theory provided a way to clarify the concept:

> In an activity theoretical approach to affordance we find that there are two kinds of use to be considered in artifacts; the possible uses and the intended use. These two uses are not independent but neither are they

> identical, and a lot of problems arise if one assumes that the intended use corresponds to the possible uses. The task of design is in many cases not to eliminate the possible uses, but rather make sure that the intended use is visible for the user. (Bærentsen & Trettvik, 2002, p. 59)

Engeström (1999) argued that contradictions—breakdowns in an activity system—allowed the emergence of new, culturally evolved activities. Turner, Turner, and Horton (1999) used the concept of contradictions as a design tool for derivation of requirements from unstructured ethnographic field data. The research methodology reported by Turner, Turner, and Horton (1999) began with diagrams of the activity systems under review, and identified four types of contradiction. Primary contradictions were defined as breakdowns within a node of the activity system, for example, the subject or object (Turner, Turner, & Horton, 1999). Secondary contradictions represented breakdowns between two nodes, such as when a subject encountered usability problems in an artifact (Turner, Turner, & Horton, 1999). Tertiary contradictions emerged between an activity and a new culturally more evolved activity, such as the common resistance encountered when a new system was introduced into an organization (Turner, Turner, & Horton, 1999). Quaternary contradictions occurred between two concurrent activities, for example, when a team in a meeting attempted to juggle multiple subjects of conversation (Turner, Turner, & Horton, 1999).

Bødker (1996) also used activity theory to analyze video research data, highlighting focus shifts and breakdowns. For each video, Bødker (1996) identified the purpose of the activity and its associated actions, the objects of each activity, and switches between objects. When a shift occurred, Bødker (1996) looked for breakdowns

and causes. Bødker (1996) concluded that "[b]reakdowns and focus shifts provide good pointers for understanding how an application mediates (or does not mediate) work activity. They are useful in identifying problems of mediation and in designing an application as well as understanding when it is brought into use" (p.172).

Reevaluation of code review research in an activity theory context.

No research that directly applied activity theory to code review was identified by this literature review. However, many software inspection studies revealed opportunities for interpretation from an activity theoretic perspective. Brothers, Sembugamoorthy, and Muller (1990) may be taken as an example of why a sociocultural analysis tool like activity theory may be useful. Brothers, Sembugamoorthy, and Muller (1990) intended to implement a tool to support code review without changing existing organizational processes, but found that "the new technological capabilities—which themselves had arisen out of a human process—had an impact back on that human process, changing its structure" (p.178). This is an apt description of how artifacts mediate activity in a dialectical relationship that leads to development and reshapes activity.

The nature of code review and the agenda of activity theory appear to suit each other. Code review is a highly individual activity that occurs within a network of social organizations, including code review teams, project teams, development departments, and corporations. A central concern of activity theory is linking historical activity with social structure (Nardi, 2007). Barthelmess and Anderson (2002) illustrated the importance of historical change in software development tools, specifically with regard to changes in programming languages and methodologies, and changes to the division of labor as programming evolved from an individual to a team activity.

Barthelmess and Anderson (2002) characterized software as a symbolic representation, a fundamentally intangible artifact. Following Kuutti's (1996) characterization of objects of activity, software source code may be considered an immaterial object of inspection activity. Deimel (1985) argued that the meaning of a software program comes from (a) a narrative generated by the computational processes followed when the program executes, and (b) a narrative in natural language made up of, among other sources, comments, variable and method names, tests, and version control commit messages. The symbolic nature of a program means that widely different natural language representations can result in the same computational process (Deimel, 1985). From an activity theory perspective, this suggests that it is useful to evaluate source code and related annotations (like commit messages) as semiotic mediating artifacts. For example, a well-commented and well-named source code file may help a new team member internalize the structure of a system more quickly than a terse file.

Miettinen (2005) considered software an example of a collective object. According to Miettinen (2005), this meant that "individuals contribute with their different capabilities to the construction of object, and attach different expectations to it" (p. 65). Miettinen (2005) suggested that researchers consider how sociocultural rules and division of labor mediated activity toward a collective software object. For example, a programmer's esteem in the community, connected to their personal and professional identity, was driven by contributions to community activity, which, in turn, were mediated by rules and division of labor (Miettinen, 2005).

Seaman and Basili (1998) also studied an aspect of code review that may be considered a mediating artifact or an object of activity: Information flow between

developers. Seaman and Basili (1998) reported that information flow affected productivity and quality, which suggests that treating information flow as an object of activity may uncover opportunities for improvement. Seaman and Basili (1998) also reported that information flow is affected by factors including development process, organizational structure, organizational hierarchy, physical distance between team members, frequency of interaction between team members, and team experience working together. In this context, it could be useful to take collaborative software development as the object of activity and study the role that information flow plays in rules, division of labor, and subject-subject mediation.

Laitenberger and DeBaud (2001) contended that reading technique was the most important factor driving code reviewer effectiveness. However, reading techniques suffered breakdowns: Checklists, for example, were found to lack instructions, not include sufficiently specific questions, and cause tunnel vision in reviewers (Laitenberger & DeBaud, 2001). An activity theory approach might treat the checklists as artifacts mediating the subject-object relationship between a reviewer and the code under review, which could lead to analysis of breakdowns in the activity.

Vygotsky (1978) used the zone of proximal development to help explain the social nature of learning and development. A person inside the zone of proximal development learns more quickly by using other people as sources of knowledge (Vygotsky, 1978). Johnson (1998) recommended that teams review defects in inspection meetings and focus on good code as well as bad. The value of this approach may be partly explained by the zone of proximal development: In a code review with more experienced programmers, a developer can generate more advanced solutions than they could on their own.

Research Project

Background

The literature reviewed above suggests that code review is a valuable software development activity. However, adoption of code review by programming teams is limited by methodological guidelines that conflict with real-world practice and supporting artifacts that provide incomplete support for collaboration and learning. Code review research has not adequately addressed the social and behavioral context surrounding reviewers and authors.

Understanding the context surrounding technical activity is an ongoing subject of HCI research. Activity theory has potential as an orienting theory for HCI research and design that seeks to understand context. The central principles of activity theory have been applied as evaluation tools and as a framework for requirements analysis. Few studies applied activity theory late in the interaction design process, as requirements were translated into user interface concepts, prototypes, and implementations.

This research provides a case study applying activity theoretic concepts to the interaction design of a code review tool. The research was conducted in three phases: A series of contextual interviews with programmers, analysis of those interviews with activity theory models, and design of user interface components informed by activity theory principles.

Setting.

Research was conducted at an established medium-sized software company headquartered in the United States. The company has a large, worldwide development organization divided into multiple departments. The department studied creates and

maintains a portfolio of Web-based products and online community, documentation, and support tools. The department staff consists of multiple teams of programmers, quality engineers, marketing professionals, program managers, and development managers. The department was created less than two years ago in a merger of staff from the desktop software department and the internal business systems group. The department uses Java, Ruby on Rails, and Adobe Flex to develop and deliver products. Teams use an internally-developed bug tracking system, and a commercial version control system. Developers use a heterogeneous collection of development environments, operating systems, and programming tools.

Code review is only mandatory for bug fixes made in a period between code freeze and a release of new software. In these cases, developers must have another developer certify that the code was reviewed by adding an annotation to the relevant bug report. The development organization has not formally adopted any processes or tools for software inspection. However, in 2007 a group of programmers created a simple Web-based code review tool. This tool allows authors to cut and paste one or more code samples for review and send email invitations to reviewers. Reviewers are able to add comments on lines of code. The code review tool has been used sporadically, with roughly 200 reviews conducted between 2007 and 2010. The original developers make occasional bug fixes and enhancements.

Design Research

To understand current code review practices field research was done within the development department discussed above. Interviews were conducted using contextual interview techniques (Beyer & Holtzblatt, 1998). Interview data was analyzed using

Kaptelinin, Nardi, and Macaulay's (1999) activity checklist as an organizational aid; those findings helped model a code review activity system (Engeström, 1999).

Methodology.

The research method used in this project is similar to the process employed by Collins, Shukla, and Redmiles (2002) to define requirements for a customer support knowledge authoring tool. Collins, Shukla, and Redmiles (2002) conducted a series of interviews and applied activity theory models to analyze the data. According to Collins, Shukla, and Redmiles (2002), these models were an efficient way to organize data, focus analysis, and elicit requirements. Contradictions proved to be a particularly useful tool for uncovering requirements that participants did not explicitly identify.

This study used the contextual interview introduced as a component of Beyer and Holtzblatt's (1998) contextual inquiry. Contextual inquiry was created to help reveal unarticulated knowledge and work structures within groups (Beyer & Holtzblatt, 1998). Contextual inquiry is based on a redefinition of the master/apprentice model, with the researcher acting as an apprentice to the interview participant (Beyer & Holtzblatt, 1998). This shares a conceptual foundation with ethnomethodology, which suggests that, because social order is an ongoing creation of people in an activity, investigators should employ the same sense-making actions as participants (Goguen & Linde, 1993).

A contextual interview is based on four principles: context, partnership, interpretation, and focus (Beyer & Holtzblatt, 1998). Context is established by going to where the participant works and observing situated behavior (Beyer & Holtzblatt, 1998). Direct observation of real work allows researchers to identify unconscious operations associated with actions, and removes the burden on the interviewee to try to remember all

elements of an activity (Beyer & Holtzblatt, 1998). As Goguen and Linde (1993) stated, "[p]eople know how to do many things they can't describe" (p. 155). The partnership principle allows a contextual interviewer and interviewee to concentrate on work practice, as the conversation is centered on the interviewee teaching the researcher how to do the work (Beyer & Holtzblatt, 1998). In a contextual interview, the principle of interpretation surfaces as part of the interview process. Interviewers are encouraged to immediately validate their interpretations with the participant, rather than delaying analysis until after all the facts are collected (Beyer & Holtzblatt, 1998). Researchers share hypotheses about the facts with participants and incorporate any corrections or elaborations (Beyer & Holtzblatt, 1998). Finally, the principle of focus encourages the interviewer to define a point-of-view, set goals for the interview, and enter an interview assuming that everything the participant does will be unique (Beyer & Holtzblatt, 1998).

Kaptelinin, Nardi, and Macaulay's (1999) activity checklist was used to analyze contextual interview data. The activity checklist was designed for use with ethnographically-inspired research techniques like contextual inquiry as a "guide to the specific areas to which a researcher or practitioner should be paying attention when trying to understand the context in which a tool will be or is used" (Kaptelinin, Nardi, & Macaulay, 1999, p. 28). Kaptelinin, Nardi, and Macaulay (1999) provided two versions of the activity checklist, one for evaluation of existing systems and one for design of new systems. The design version of the checklist was used in this project and is included as Appendix 1. Each version of the checklist provides a set of contextual factors grouped into activity-theoretic "perspectives" of means/ends, environment, learning/cognition/articulation, and development (Kaptelinin, Nardi, & Macaulay, 1999).

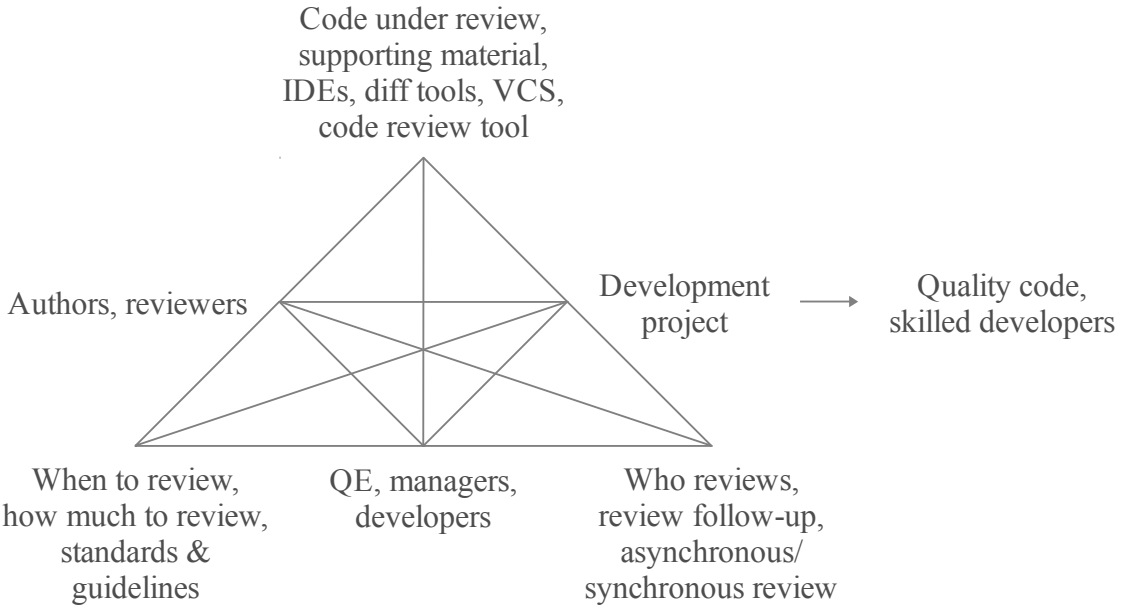


Figure 3: A code review activity system

This project used the activity checklist before conducting contextual interviews to ensure that the interview plan addressed issues of learning, development, and historicity. The activity checklist was then used to identify gaps in interview data and to prioritize interface design concepts.

Results.

Over the course of two weeks interviews were conducted with nine developers across seven teams. Interviews lasted between 30 and 60 minutes, and were recorded in notes taken by the interviewer. Interviews followed the structure for a contextual interview found in Beyer and Holtzblatt (1998). An interview guide was used to ensure discussion of critical topics (Appendix 2).

Analysis of the interviews revealed a collective activity system surrounding code review (Figure 3). The subject is a group of collaborating developers who act alternatively as code review authors or reviewers. The collective object of their activity is

an evolving software system. Broader activity oriented at the same software system may include creating enhancements or maintaining and fixing existing code. The projected outcomes—software quality and developer skill—suggest that code review activity in this department is polymotivated. Most developers were motivated by improving software quality: "[T]he point is that everyone is trying to make the system better." This objective motivates activity over time. A participant expressed that "part of efficiency is getting it right, and we *always* find something not right in a code review. Even if it only improves marginally, that reduces your technical debt." Echoing conclusions drawn by Johnson (1998), Trytten (2005), and Tyran and George (2002), multiple participants were motivated by the learning opportunities that code review presented. A participant described how educational benefits accrued for her entire team: "We try to keep the format of our reviews predictable because there's a cross-training element. The other two people on my team are learning the technology, so it helps to have the time to go through each review item as needed." Another developer tailored his approach as a reviewer based on the opportunity to teach: "If the author has less experience I'll focus less on patterns and more on syntax. If I can read the code beforehand I can go into the review knowing whether I want to focus more on the quality of the code versus the quality of the developer."

Activity theory suggests analysis at the level of actions—conscious, goal-directed hierarchies of processes that are enacted to achieve the object (Kaptelinin & Nardi, 2006). Code review in the department is summarized as sequences of actions in Figure 4. Lightning symbols (↯) in Figure 4 represent contradictions identified by interviewees.

Code review authors expressed tension between recruiting the most qualified

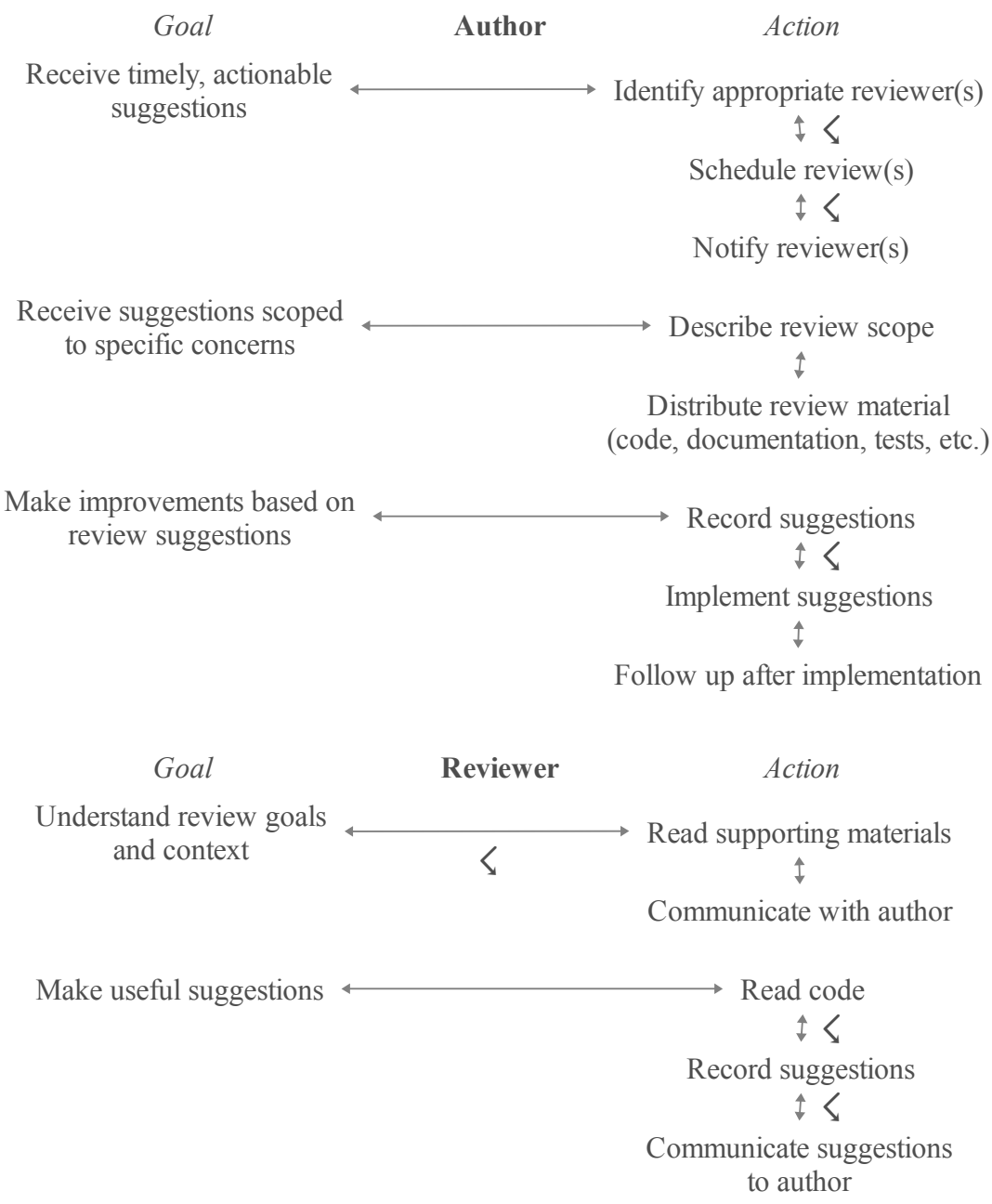


Figure 4: Code review goals and actions

reviewers versus reviewers who were immediately available. Many teams used "over-the-shoulder" reviews, calling in a fellow developer as needed. This breaks down when the reviewer is busy or out of the office. A participant reported difficulty using the Web-based code review tool because there was no way to set a deadline for reviewers, and no way to

receive notification when reviewers added suggestions. Authors sometimes ignored or deferred suggestions, indicating breakdown moving from the act of recording suggestions to implementing those recommendations: “Sometimes I pick the low-hanging fruit, skip some suggestions that might take hours of refactoring that we don’t have time budgeted for.” Multiple code reviewers complained that authors did not provide sufficient context to understand the goals of the review. In some cases the author failed to relay the purpose of the code—what it was meant to do and why. In other cases the author did not provide enough supporting material, such as use cases and unit tests: “I always have to ask other authors for background material.”

Asynchronous review introduced contradictions in developers’ actions. Authors worried that the distributed nature of asynchronous review would lead to social loafing because “there’s no way to guarantee a thorough review, if there are other people reviewing you may not flag everything.” This led some participants to complain that asynchronous reviews provided less feedback than expected. However, one developer saw this as an advantage: “You can use the shotgun approach, send a review out to four people and if only two reply that’s OK.”

Engeström (1999) suggested that activity has its social basis in the wider community surrounding subjects’ actions. In the department studied, community was defined by development teams and by technology expertise. Developers most often reviewed code with members of their own team. Review within the team was seen as the most practical approach because team members were likely to be nearby and familiar with the components under review. Team-based review also supported development of mediating artifacts and social relationships. One participant stated that “having code

review as part of the team process helps maintain standards and strengthens traditions that other developers keep." Expertise also defined code review communities; participants were likely to recruit reviewers who had experience with specific code libraries or technologies. Occasional participants in the code review community included quality engineers, who might have questions for a reviewer, and managers, who sometimes tracked the action items that emerged from code reviews.

The relationship between subjects and communities is mediated by rules and social norms (Engeström, 1999). Only one company-wide rule affected participants' code review activity: Code changes made in a specific period of time before a release must be approved by an engineering code reviewer. In most cases this rule affected small changes handled by one-on-one, over-the-shoulder code reviews. Beyond this, the rules mediating code review were very informal. Some participants felt that code review should be more formal. Often this was based on experiences at other firms. For example, one participant said "Our team's code review can be too informal, compared with other places I've worked. What we do is closer to pair programming, fix-as-you-go type review." Others disagreed about the level of formality: "We instituted a lightweight process in our team, we want to be as agile as possible but still have some checkpoint. So we just do a one-on-one review before anyone checks in code." Even among participants who thought that the process was too informal there was tension around making the process more formal. One developer stated "I'm a fan of enforcing code review whenever you check in code, but the Catch-22 is that I want control without being controlled." This tension over formality was reflected in other rules that mediated code review activity.

Some participants valued rules enforced by coding standards. However, the

standards in use tended to be informal documents maintained by individual teams to enforce naming and formatting conventions. Rules about when to review code varied from participant to participant. One developer reviewed his code before any commit to version control, but stated "reviews for one-line changes can feel silly" and "you may not always want to review things when you feel like they're not quite done yet." Another developer reviewed code after it had unit tests but before it was built, though urgent changes were often built and then reviewed when time permitted.

Collective activity is also mediated by division of labor (Engeström, 1999). From an activity theoretic perspective, division of labor is a useful way to analyze tradeoffs between synchronous and asynchronous reviews. As described above, some interview participants feared that the asynchronous model would allow reviewers to ignore requests, or to provide cursory feedback. The lack of back-and-forth discussion he experienced in asynchronous reviews led one participant to state "if the author and reviewer agree on the problem and the approach to the solution, then an asynchronous approach might work; then it becomes 'is this correct' rather than 'is this appropriate?' Anything other than 'check if I goofed' needs conversation."

Code review work was divided into actions conducted by the author, actions conducted by the reviewer, and actions conducted between authors and reviewers. Seniority and experience imbalances sometimes led to contradictions in this division of labor. One participant stated "code review can look threatening at first, having three other developers look at your work tends to feel personal. But developers get more comfortable over time, they see that everyone goes through it, you just have to keep it professional." Another participant preferred in-person reviews because "it feels more personal, I can

defend myself, and provide the rationale behind a decision." Knowledge of a code base also determined how work was distributed within the department, or even within teams. One participant described challenges related to the distribution of experience within his team: "Each person has specialties, but we try to get everyone involved. Sometimes you have to expect to bring the reviewer up to speed about the context and the code they're looking at. For now our code base is small enough that it's not a burden.

Participants described many artifacts that mediated their code review activity (Figure 5). The code under review was the most visible artifact. Code is an abstract artifact, so participants relied on different representations to support their reviews. In some cases changed files were simply printed out and read. This practice was improvisational for one participant, who reported that "if the change is large or we have too many files we will put the code in a Microsoft Word document and review that in a meeting room." Other participants gained an understanding of the changed code by running it, either on a development server or on their personal workstation. Treating the changed code as a runnable artifact allowed one participant to learn about dependencies within the system, in addition to simply seeing if the code would execute properly. In some cases reviewing changes to the code sharpened the developer's understanding of the wider system: "The act of code review can tell you a lot about the code. Is that necessary complexity or just failure to simplify?"

Unit tests, class diagrams, white board sketches, and technical specifications were mentioned as supporting artifacts that help reviewers understand the code under review. Most reviewers wanted unit tests included in the code review, and considered it a breakdown when unit tests were missing. Some participants ran the unit tests to verify the

Subject	Action	Mediating artifacts
Author	Identify appropriate reviewer(s)	Past reviews, discussions with other developers, intranet
	Schedule review(s)	Email, calendar software, discussion
	Notify reviewer(s)	Email, instant messenger (IM), discussion
	Describe review scope	Code under review, email, presentation software, word processing software, white boards, code comments, IDE/text editor, design documentation, unit tests, Web-based review tool, discussion
	Distribute review materials	Code under review, email, Web-based review tool, paper printouts, white board, IM, version control software (VCS)
	Record suggestions	Code under review, IDE/text editor, white board, notepads, Web-based review tool
	Implement suggestions	IDE/text editor, recorded suggestions
	Follow up after implementation	Email, IM, VCS
Reviewer	Read supporting materials	Code under review, email, Web-based review tool, paper printouts, white board, word processing software
	Communicate with author	Email, IM, discussion
	Read code	Code under review, email, IDE/text editor, Web-based review tool, paper printouts, VCS, white board
	Record suggestions	Code under review, IDE/text editor, white board, notepads, Web-based review tool
	Communicate suggestions to author	Email, IM, discussion, Web-based review tool

Figure 5: Mediation of code review actions by artifacts

code's behavior, others read the tests in parallel with the code. One participant's team distributed a template-based technical specification document with every code review. However, most documents supporting reviews were created ad-hoc: "Sometimes, for larger [reviews], we create a design doc for the review, with the notable things on it. We track what will be deployed, lists of configuration changes, and code changes, anything

we touched."

Code authors used varied tools to collect and organize the suggestions reviewers made. Most interview participants took paper notes in face-to-face meetings or while reviewing feedback sent via email or through the Web-based review tool. In reviews that used code printouts, some authors added notes directly to the printouts. Notes were mainly for the author's benefit, as follow-up on suggestions was uncommon and informal. One participant described follow up as "the author's responsibility, after they make a change they might get back together and go over how the change was implemented." Most participants said that if follow-up was required the author would send an email containing the new code or a link to changes in version control.

The version control system (VCS) played an important role as a mediator of code review activity. A relatively new feature allowed programmers to check changed code into a staging area, making the changes visible to anyone without affecting the main development branch. The VCS vendor implemented this feature specifically to help with code review. One participant commented that before this feature was available the VCS was a bottleneck to review, because an author had to commit volatile changes to the main branch. Despite the perceived value, only two participants described actively using the new staging feature; two participants were unaware of the feature.

The VCS incorporated a diff tool that all participants found valuable. One participant claimed that a good diff tool was the most important part of code review, because "the diff is the thing that *proves* that only one thing changed." In this instance, the diff tool served to keep the author from intentionally or accidentally making changes that were outside the scope of the review. For other participants, the diff tool structured

the review. The author and reviewer opened the diff tool, then stepped through and analyzed each change. The diff tool reshaped the participants' view of the code under review, shifting the perspective from files and classes to sets of changes. However, the diff tool caused some low-level breakdowns. One participant reported that the keyboard bindings in the diff tool were non-standard, making navigation challenging. In activity theoretic terms, this tool caused a normally unconscious operation, keystroke-based navigation, to become a conscious action. Another participant used a diff tool built into his interactive development environment, which was not as feature-rich as the VCS diff tool but which he had better internalized as a part of routine use.

Interactive development environments (IDEs) are important mediators of programmers' work (Barthelmess & Anderson, 2002). IDE features helped interview participants internalize the code reading process. Syntax highlighting simplified the structure of code. Search tools allowed programmers to find code in multiple scopes. Some IDEs could automatically link from a method call to the method's definition, which "helps you figure things out in the context of the code, it's a more natural way to navigate." However, development environments could introduce contradictions as well. One participant detailed a development tool that complicated code reading actions: "There's nothing useful, no syntax highlighting, some of the fields are very cramped, and the code in those fields are some of the hardest to get right, the ones we want to review the most. Once in a while we'll copy the code into a text editor just to get the formatting useful." A different team attempted to adopt a code review plug-in for their IDE, but gave up because it was cumbersome to install and poorly matched their workflow.

Implications.

From an interaction design perspective, code review practice in the department studied is informal, situated, mediated, and developmental. Tools meant to support code review activity should address each of these perspectives.

The informal nature of code review means that tools must allow authors and reviewers to apply varying amounts of structure. For example, an author may wish to send a review to many reviewers hoping for some commentary in a short time frame. Another author may want suggestions from a specific group of reviewers, but be open to a longer time frame. The ad-hoc nature of many code reviews suggests that authors should be able to initiate a review from multiple contexts—for instance, from an IDE or a bug report. Authors tend to work with the same small group of colleagues; tools should reflect this by making it easy for authors to choose from common collaborators or, when needed, find potential reviewers based on technical experience. Informality led some interviewees to worry about the amount of focus and rigor that reviewers applied when reading code. A review tool could help mitigate this by allowing the code author to set deadlines, which would push reviewers to either read the code or refuse to participate in the review. Additionally, the tool could reward reviewers for their contributions, adding an informal indicator of status and accomplishment to a highly informal activity.

A code review tool must also accommodate the situated nature of reviews. Many reviews happen with two programmers sitting side-by-side reading code on the same screen. The interface should be flexible enough to support co-reading comfortably. The tool should support situations where a reviewer and author are in separate locations but want to communicate, in real time or asynchronously. Finally, a code review tool must

handle not only changes to existing code, which may be represented in a diff tool that compares past and current versions, but also new code, which usually requires external documentation to provide context.

As a mediating artifact, a code review tool is integral in users' progress toward their goals. A well-designed tool will help users internalize common operations, such as selecting reviewers or navigating through source code. In addition, the tool should be flexible enough to support externalization, as users evolve their existing activity. One possible avenue for externalization is the relationship between a code review tool and the wider development process—integrating a code review tool with version control and deployment systems could shape new ways of doing work. A code review tool may mediate reviewers' efforts to analyze code by integrating supporting documentation, such as specifications provided by authors and artifacts like checklists and author-reviewer conversations.

Several interviewees listed development—personal or collaborative—as an objective of code review. Tools can facilitate development by, for example, grouping and recording reviewers' suggestions across reviews, forming a list of anti-patterns. Reviewers may also highlight code samples that they find exemplary, allowing new developers to see examples of elegant or canonical solutions. By supporting discussion and proposed solutions, a code review tool may become a space for problem solving in addition to problem identification.

Interaction Design

This section describes a prototype user interface based on the implications above. The prototype was designed within a conceptual architecture informed by Hedberg's

(2004) guidelines for next-generation code review tools:

- Inspection of multiple artifact formats;
- Integration of multiple repository types;
- Asynchronous or synchronous process;
- Integration with project management tools;
- Results that include severity, type, and resolution of defects.

Architecture.

Figure 6 divides the architecture of the user interface into seven conceptual layers, each of which illustrates a concrete or conceptual tool mediating code review. The code review tool itself is at the center of the architecture, providing a representation of reviews and review items (for example, files, consolidated change lists, or uploaded graphics). The code review architecture specifies an interface connecting the review tool to version control, enabling developers to create a review directly from a VCS change list. This will also allow reviewers to browse a file's version history within the code review tool. The review tool will also integrate with IDEs and editors that support remote editing. In face-to-face reviews, this feature will allow developers to make immediate edits and submit those edits in VCS.

The architecture also incorporates a presence layer, which will provide information about developers' most common collaborators. While review participation will not be limited to these collaborators, research with developers showed that most reviews were conducted within teams or between closely aligned teams. The presence-based component of the architecture will simplify the process of initiating a review with teammates. Additionally, the review tool will integrate with open messaging systems like

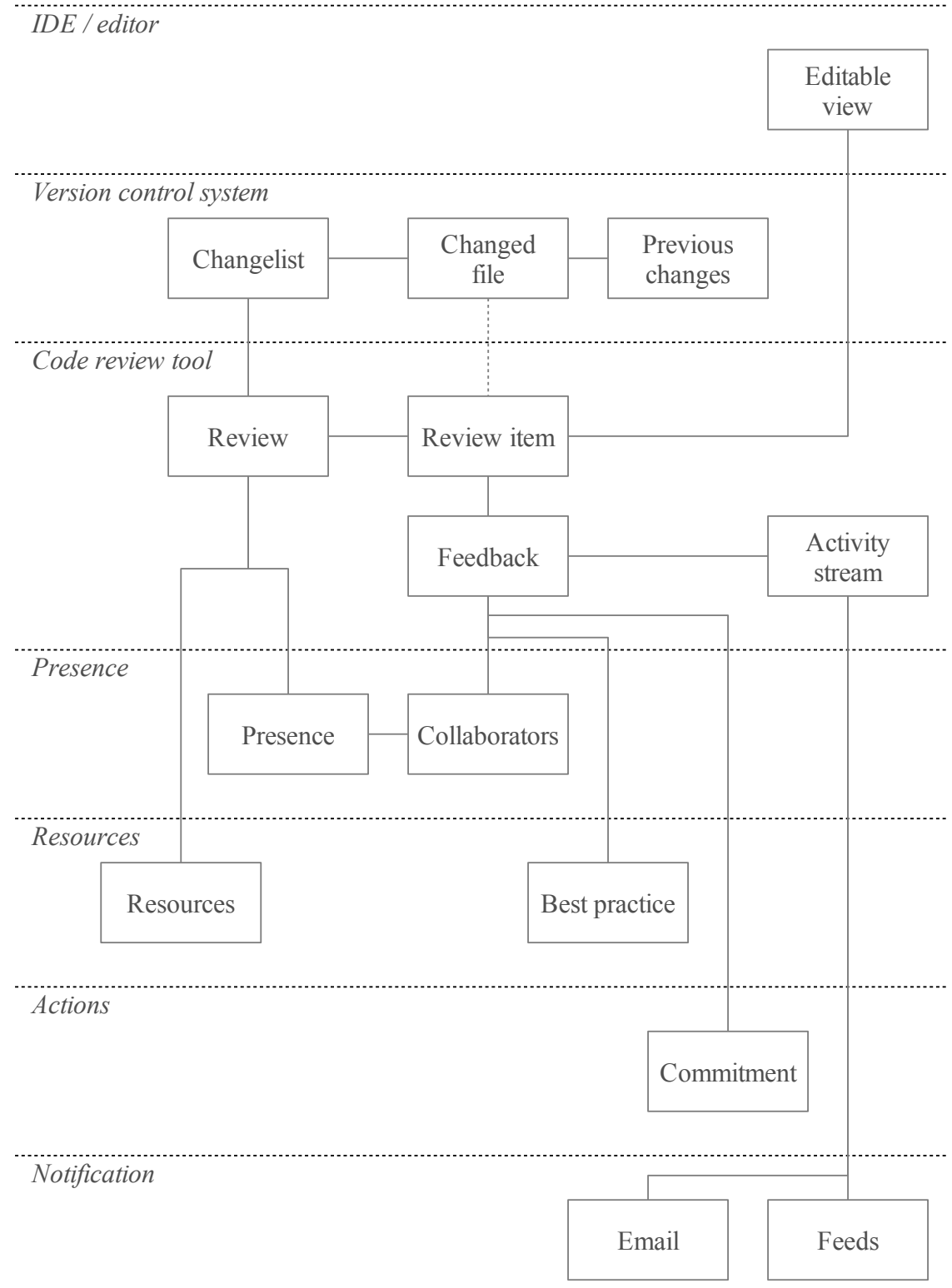


Figure 6: Conceptual architecture for a Web-based code review tool

Figure 7: Presence awareness, showing available, offline, and busy users

Jabber, allowing asynchronous reviews that include live chat when author and reviewer are both online (Figure 7).

Reviewers will be able to add multiple types of feedback to review items, including questions, comments, and flags—a collective label for suggestions and defects. Reviewers may also highlight good examples, which are collected across reviews into sets of language- or subject-specific best practices. All activity is reported in an activity stream for each user. The activity stream is available through the Web user interface, but, as represented in the notification layer of the architecture, users may elect to subscribe to their activity stream in Atom Publishing Protocol format and optionally receive email alerts about specific activities.

Multiple interview participants expressed dissatisfaction with review follow-up. Authors used ad-hoc methods to track recommendations, and reviewers voiced concerns that their suggestions might be ignored. The architecture discussed here focuses on actions resulting from reviews in the form of commitments that authors make based on reviewers' suggestions.

User interface.

Specific elements of the prototype user interface address breakdowns identified in contextual interviews, opportunities for mediation identified as implications of those breakdowns, and recommendations from scholarly code review research (Figure 8).

The review tool applies lightweight process mechanics to the predominantly ad-

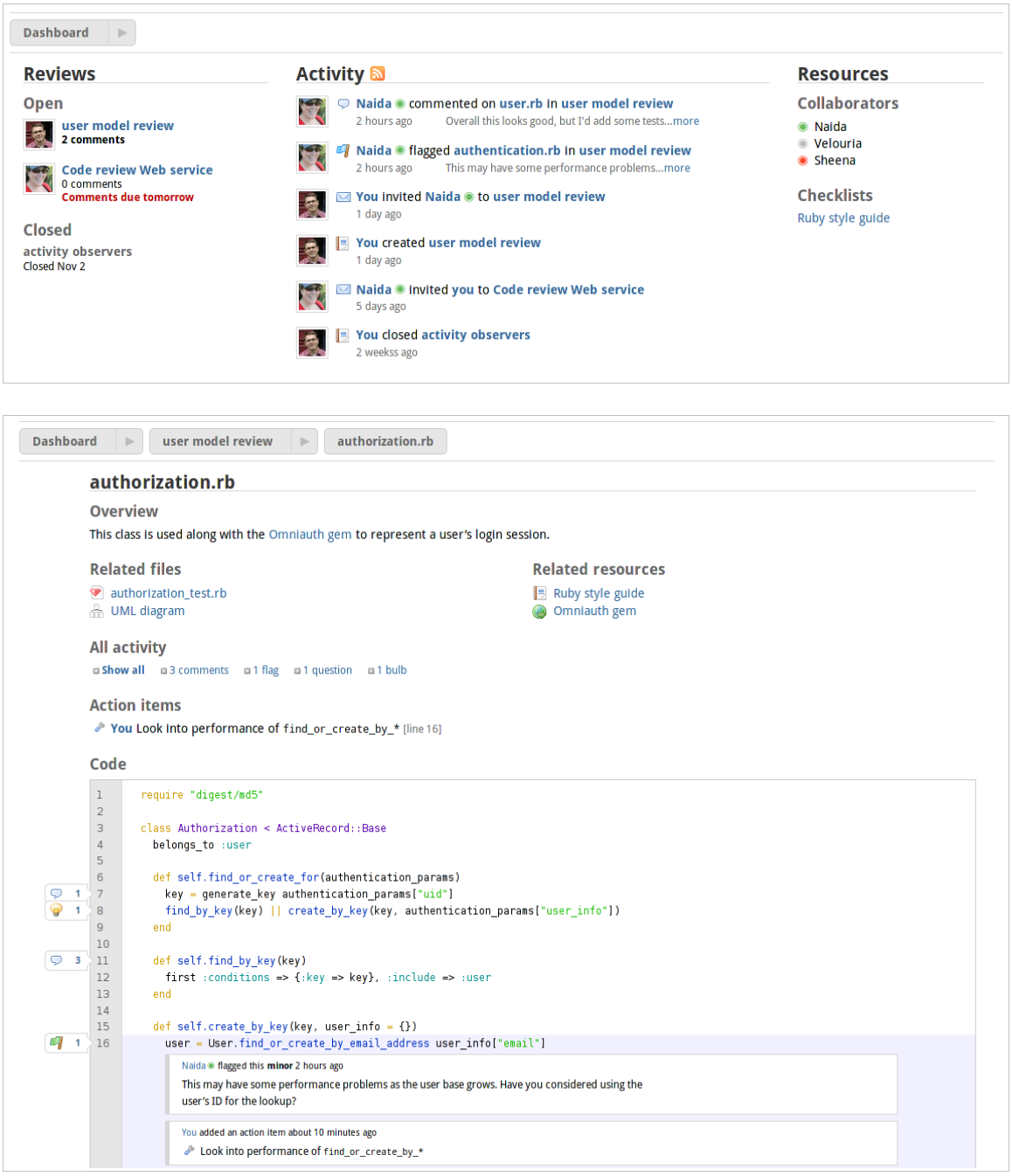


Figure 8: Screen shots of the code review dashboard (top) and a review item (bottom)

hoc review process. As seen in the review dashboard in Figure 8, review authors may set deadlines for reviews—the dashboard indicates when contributions are due soon. Authors may also close reviews, which de-emphasizes the closed review and allows the

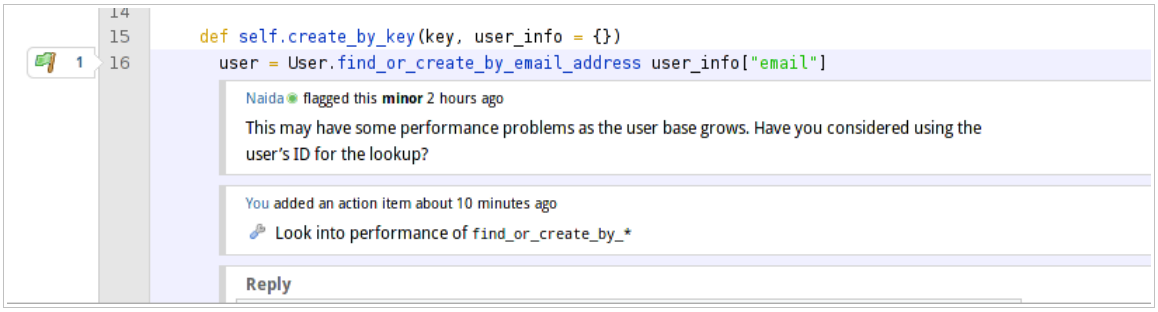


Figure 9: Collaboration between author and reviewer

author/reviewer to attend to active reviews. The dashboard also highlights the user's frequent collaborators and centers on the activity stream discussed above.

Figure 9 reflects the primary mode of interaction between reviewers and authors. Reviewers may flag sections of code, suggest improvements, and indicate the importance of the suggestion; either minor, moderate, or critical. The author can reply in-line, or agree to implement the suggestion. If the author agrees to the suggestion, it is added to a review-wide list of action items. When aggregated across review files, this list of action items becomes a representation of activity that will improve the overall quality of the software project.

Discussion

This thesis addressed code review from an activity theoretic perspective. The literature review showed that activity theory could be a compelling tool for engaging the human- and context-centered challenges of modern code review. Research conducted within a software development organization was analyzed using tools from interaction design and activity theory. Implications derived from this research were applied to create an interaction design model and user interface prototypes for a Web-based code review tool.

Activity theory proved to be a useful tool for framing interaction design questions, shaping research objectives, and analyzing research data. However, as the design became more concrete, the direct applicability of activity theoretic concepts waned. Future extensions of this research may proceed in at least two directions. First, a more complete implementation of the code review tool should be deployed and observed in use. This will allow further validation of the premise that activity-theory-driven requirements result in usable, desirable, and useful products. Finally, specific user interface design decisions should be more closely associated with concepts from activity theory. This research may be conducted in concert with the observational studies proposed above, but with a sharper focus on specific actions and operations. For example, a more focused future study could investigate whether checklists and collections of reviewer-identified best practices mediate increased learning in programmers who are new to a technology or project.

References

Bærentsen, K. B., & Trettvik, J. (2002). An activity theory approach to affordance. *Proceedings of the second Nordic conference on Human-computer interaction* (pp. 51-60). New York: ACM.

Bakhurst, D. (2007). Vygotsky's Demons. In H. Daniels, M. Cole, & J. V. Wertsch (Eds.), *The Cambridge companion to Vygotsky* (pp. 50-76). New York: Cambridge University Press.

Bannon, L. J. (1991). From human factors to human actors: The role of psychology and human-computer interaction studies in systems design. In J. Greenbaum & M. King (Eds.), *Design at work: Cooperative design of computer systems* (pp. 25-44). Hillsdale, NJ: Erlbaum.

Bardram, J. E. (1997). Plans as situated action: An activity theory approach to workflow systems. *Proceedings of the 5th European Conference on Computer Supported Cooperative Work* (pp. 17-32). Norwell, MA: Kluwer Academic Publishers.

Barthelmess, P., & Anderson, K. M. (2002). A view of software development environments based on activity theory. *Computer Supported Cooperative Work, 11*, 13-37.

Basili, V. R. (1997). Evolving and packaging reading technologies. *Journal of Systems and Software, 38*, 3-12.

Basili, V. R., Green, S., Laitenberger, O., Lanubile, F., Shull, F., Sørumgard, S., & Zelkowitz, M. V. (1996). The empirical investigation of perspective-based reading. *Empirical Software Engineering: An International Journal, 1*, 133-164.

Beck, K. (with Andres, C.). (2004). *Extreme programming explained* (2nd Ed.). Boston,

MA: Addison-Wesley.

Beyer, H., & Holtzblatt, K. (1998). *Contextual design: Defining customer-centered systems*. San Francisco: Morgan Kaufmann.

Béguin, P., & Rebardel, P. (2000). Designing for instrument-mediated activity. *Scandinavian Journal of Information Systems, 12*(1-2), 173-190.

Biffl, S. (2000, November-December). Using inspection data for defect estimation. *IEEE Software*, 36-43.

Bødker, S. (1996). Applying activity theory to video analysis: How to make sense of video data in human-computer interaction. In B. A. Nardi (Ed.), *Context and consciousness: Activity theory and human-computer interaction* (pp. 147-174). Cambridge, MA: MIT Press.

Bragdon, A., Zeleznik, R., Reiss, S. P., Karumuri, S., Cheung, W., Kaplan, J., . . . LaViola, J. J., Jr. (2010). Code Bubbles: A working set-based interface for code understanding and maintenance. *Proceedings of the 28th International Conference on Human Factors in Computing Systems* (pp. 2503-2512). New York: ACM.

Brothers, L., Sembugamoorthy, V., & Muller, M. (1990). ICICLE: Groupware for code inspection. *Proceedings of the 1990 ACM Conference on Computer-Supported Cooperative Work* (pp. 169-181). New York: ACM.

Bryant, S. L., Forte, A., & Bruckman, A. (2005). Becoming Wikipedian: Transformation of participation in a collaborative online encyclopedia. *Proceedings of the 2005 International ACM SIGGROUP Conference on Supporting Group Work* (pp. 1-10). New York: ACM.

Card, S. K., Moran, T. P., & Newell, A. (1983). *The psychology of human-computer*

interaction. Hillsdale, NJ: Erlbaum.

Carroll, J. M. (1997). Human-computer interaction: Psychology as a science of design. *International Journal of Human-Computer Studies, 46*, 501-522.

Collins, P., Shukla, S., & Redmiles, D. (2002). Activity theory and system design: A view from the trenches. *Computer Supported Cooperative Work, 11*, 55-80.

Deimel, L. E., Jr. (1985). The uses of program reading. *SIGCSE Bulletin, 17*(2), 5-14.

Denger, C., & Shull, F. (2007, March-April). A practical approach for quality-driven inspections. *IEEE Software*, 79-86.

Devito Da Cunha, A., & Greathead, D. (2007). Does personality matter? An analysis of code-review ability. *Communications of the ACM, 50*(5), 109-112.

Engeström, Y. (1999). Activity theory and individual and social transformation. In Y. Engeström, R. Miettinen, & R-L. Punamäki (Eds.), *Perspectives on activity theory* (pp. 19-38). New York: Cambridge University Press.

Fagan, M. E. (1976). Design and code inspections to reduce errors in program development. *IBM Systems Journal, 15*, 182-211.

Fagan, M. E. (1986). Advances in software inspections. *IEEE Transactions on Software Engineering, 12*, 744-751.

Gay, G., & Hembrooke, H. (2004). *Activity-centered design: An ecological approach to designing smart tools and usable systems*. Cambridge, MA: MIT Press.

Glass, R. L. (1999). Inspections—some surprising findings. *Communications of the ACM, 42*(4), 17-19.

Goguen, J. A., & Linde, C. (1993). Techniques for requirements elicitation. *Proceedings of the 1993 IEEE International Symposium on Requirements Engineering* (pp.

152-164). Los Alamos, CA: IEEE Computer Society Press.

Halverson, C. A. (2002). Activity theory and distributed cognition: Or what does CSCW need to DO with theories? *Computer Supported Cooperative Work, 11*, 243-267.

Hassenzahl, M., & Tractinsky, N. (2006). User experience—a research agenda. *Behaviour & Information Technology, 25*(2), 91-97.

Hedberg, H. (2004). Introducing the next generation of software inspection tools. In F. Bomarius, H. Iida (Eds.), *Product Focused Software Process Improvement: 5th International Conference, PROFES 2004* (pp. 234-247). Berlin, Germany: Springer.

Holland, D., & Reeves, J. R. (1996). Activity theory and the view from somewhere: Team perspectives on the intellectual work of programming. In B. A. Nardi (Ed.), *Context and consciousness: Activity theory and human-computer interaction* (pp. 257-281). Cambridge, MA: MIT Press.

Hundhausen, C., Agrawal, A., Fairbrother, D., & Trevisan, M. (2009). Integrating pedagogical code reviews into a CS 1 course: An empirical study. *SIGCSE Bulletin, 41*(1), 291-295.

Hundhausen, C., Agrawal, A., & Ryan, K. (2010). The design of an online environment to support pedagogical code reviews. *Proceedings of the 41st ACM Technical Symposium on Computer Science Education* (pp. 182-186). New York: ACM.

Hyysalo, S. (2005). Objects and motives in a product design process. *Mind, Culture, and Activity, 12*, 19-36.

Iisakka, J., & Tervonen, I. (2001). The darker side of inspections. In M. Lawford & D. L. Parnas (Eds.), *WISE'01: Proceedings of the 1st Workshop on Inspection in*

Software Engineering (pp. 99-104), Hamilton, Canada: McMaster University.

Johnson, P. M. (1994). An instrumented approach to improving software quality through formal technical review. *Proceedings of the 16th International Conference on Software Engineering* (pp. 113-122). Los Alamitos, CA: IEEE Computer Society Press,

Johnson, P. M. (1998). Reengineering inspection. *Communications of the ACM, 41*(2), 49-52.

Kaptelinin, V., & Nardi, B. A. (2006). *Acting with technology: Activity theory and interaction design*. Cambridge, MA: MIT Press.

Kaptelinin, V., Nardi, B. A., & Macaulay, C. (1999, July-August). The activity checklist: A tool for representing the "space" of context. *Interactions*, 27-39.

Karlsson, M. (2010). Code reviews. In K. Henney (Ed.), *97 things every programmer should know* (pp. 28-29). Sebastopal, CA: O'Reilly Media.

Kelly, D., & Shepard, T. (2003). An experiment to investigate interacting versus nominal groups in software inspection. *Proceedings of the 2003 Conference of the Centre For Advanced Studies on Collaborative Research* (pp. 122-134). Indianapolis, IN: IBM Press.

Kemerer, C. F., & Paulk, M. C. (2009). The impact of design and code reviews on software quality: An empirical study based on PSP data. *IEEE Transactions on Software Engineering, 35*, 534-550.

Kuutti, K. (1996). Activity theory as a potential framework for human-computer interaction research. In B. A. Nardi (Ed.), *Context and consciousness: Activity theory and human-computer interaction* (pp. 17-44). Cambridge, MA: MIT Press.

Kuutti, K. (1999) Activity theory, transformation of work and information systems design. In Y. Engeström, R. Miettinen, & R. Punamäki (Eds.), *Perspectives on Activity Theory* (pp. 360-376). Cambridge, UK: Cambridge University Press.

Laitenberger, O., & Dreyer, H. M. (1998). Evaluating the usefulness and the ease of use of a web-based inspection data collection tool. *Proceedings of the 5th International Symposium on Software Metrics* (pp. 122-134). Washington, DC: IEEE.

Laitenberger, O., & DeBaud, J-M. (2001). An encompassing life-cycle centric survey of software inspection. *Journal of Systems and Software, 50*, 5-31.

Leontiev, A. N. (1978). *Activity, consciousness, and personality*. Hillsdale: Prentice-Hall.

Matthews, T., Rattenbury, T., & Carter, S. (2007). Defining, designing, and evaluating peripheral displays: An analysis using activity theory. *Human-Computer Interaction, 22*(1), 221-261.

Mäntylä, M. V., & Lassenius, C. (2006). Drivers for software refactoring decisions. *Proceedings of the 2006 ACM/IEEE International Symposium on Empirical Software Engineering* (pp. 297-306). New York: ACM.

Mäntylä, M. V., & Lassenius, C. (2009). What types of defects are really discovered in code reviews? *IEEE Transactions on Software Engineering, 35*, 430-448.

Meyer, B. (2008). Design and code reviews in the age of the Internet. *Communications of the ACM, 51*(9), 67-71.

Miettinen, R. (2005). Object of activity and individual motivation. *Mind, Culture, and Activity, 12*, 52-69.

Miettinen, R., & Hasu, M. (2002). Articulating user needs in collaborative design:

Towards an activity-theoretical approach. *Computer Supported Cooperative Work, 11*, 129-151.

Nardi, B. A. (2007). Placeless organizations: Collaborating for transformation. *Mind, Culture, and Activity, 14*, 5-22.

Newell, A., & Card, S. K. (1985). The prospects for psychological science in human-computer interaction. *Human Computer Interaction, 1*, 209-242.

Nielsen, J. (1994). Usability inspection methods. *Conference Companion on Human Factors in Computing Systems* (pp. 413-414). New York: ACM.

Norman, D. A. (1993). *Things that make us smart: Defending human attributes in the age of the machine*. Cambridge, MA: Perseus Books.

Norman, D. A. (2005, July-August). Human-centered design considered harmful. *interactions,* 14-19.

Norman, D. A. (2006, November-December). Logic versus usage: The case for activity-centered design. *interactions*, 45, 63.

Omoronyia, I., Ferguson, J., Roper, M, & Wood, M. (2009). Using developer activity data to enhance awareness during collaborative software development. *Computer Supported Cooperative Work, 18*, 509-558.

Parnin, C., Görg, C., & Nnadi, O. (2008). A catalogue of lightweight visualizations to support code smell inspections. *Proceedings of the 4th ACM Symposium on Software Visualization* (pp. 77-86). New York: ACM.

Perpich, J. M., Perry, D. E., Porter, A. A., Votta, L. G., & Wade, M. W. (1997). Anywhere, anytime code inspections: Using the Web to remove inspection bottlenecks in large-scale software development. *Proceedings of the 19th International*

Conference on Software Engineering (pp. 14-21). New York: ACM.

Porter, A., Siy, H., Mockus, A., & Votta, L. G. (1998). Understanding the sources of variation in software inspections. *ACM Transactions on Software Engineering and Methodology, 7*, 41-79.

Quek, A., & Shah, H. (2004). A comparative survey of activity-based methods for information systems development. *Proceedings of the 6th International Conference on Enterprise Information Systems* (pp. 221-232). Lisbon, Portugal: INSTICC.

Raskin, J. (2000). *The humane interface: New directions for designing interactive systems*. Reading, MA: Addison-Wesley.

Reily, K., Finnerty, P. L., & Terveen, L. (2009). Two peers are better than one: aggregating peer reviews for computing assignments is surprisingly accurate. *Proceedings of the ACM 2009 International Conference on Supporting Group Work* (pp. 115-124). New York: ACM.

Rombach, D., Ciolkowski, M., Jeffery, R., Laitenberger, O., McGarry, F., & Shull, F. (2008). Impact of research on practice in the field of inspections, reviews, and walkthroughs: Learning from successful industrial uses. *SIGSOFT Software Engineering Notes, 33*(6), 26-35.

Rose, J., Jones, M., & Truex, D. (2005). Socio-theoretic accounts of IS: The problem of agency. *Scandinavian Journal of Information Systems, 17*(1), 133-152.

Sauer, C. D., Ross, J., Land, L., & Yetton, P. (2000). The effectiveness of software development technical reviews: A behaviorally motivated program of research. *IEEE Transactions on Software Engineering, 26*, 1-14.

Seaman, C. B., & Basili, V. R. (1998). Communication and organization: An empirical study of discussion in inspection meetings. *IEEE Transactions on Software Engineering, 24*, 559-572.

Sharp, H., Rogers, Y., & Preece, J. (2007). *Interaction design: Beyond human-computer interaction* (2nd Ed.). West Sussex, UK: John Wiley & Sons.

Siy, H., & Votta, L. (2001). Does the modern code inspection have value? *Proceedings of the IEEE International Conference on Software Maintenance* (pp. 281-290). Washington, DC: IEEE.

Stein, M., Riedl, J., Harner, S. J., & Mashayekhi, V. (1997). A case study of distributed, asynchronous software inspection. *Proceedings of the 19th International Conference on Software Engineering* (pp. 107-117). New York: ACM.

Stetsenko, A. (2005). Activity as object-related: Resolving the dichotomy of individual and collective planes of activity. *Mind, Culture, and Activity, 12*, 70-88.

Storey, M-A., Cheng, L-T., Bull, I., & Rigby, P. (2006). Shared waypoints and social tagging to support collaboration in software development. *Proceedings of the 2006 20th Anniversary Conference on Computer Supported Cooperative Work* (pp. 195-198). New York: ACM.

Thelin, T., Runeson, P., & Wohlin, C. (2003). An experimental comparison of usage-based and checklist-based reading. *IEEE Transactions on Software Engineering, 29*, 687-704.

Trytten, D. A. (2005). A design for peer code review. *Proceedings of the 36th SIGCSE Technical Symposium on Computer Science Education* (pp. 455-459). New York: ACM.

Turner, P., Turner, S., & Horton, J. (1999). From description to requirements: An activity theoretic perspective. *Proceedings of the International ACM SIGGROUP Conference on Supporting Group Work* (pp. 286-295). New York: ACM.

Turner, S. A., Quintana-Castillo, R., Pérez-Quiñones, M. A., & Edwards, S. E. (2008). Misunderstandings about object-oriented design: Experiences using code reviews. *Proceedings of the 39th SIGCSE Technical Symposium on Computer Science Education* (pp. 97-101). New York: ACM.

Tyran, C. K., & George, J. F. (2002). Improving software inspections with group process support. *Communications of the ACM, 45*(9), 87-92.

Uwano, H., Monden, A., & Matsumoto, K. (2008). Are good code reviewers also good at design review? *Proceedings of the Second ACM-IEEE International Symposium on Empirical Software Engineering and Measurement* (pp. 351-353). New York: ACM.

Uwano, H., Nakamura, M., Monden, A., & Matsumoto, K. (2006). Analyzing individual performance of source code review using reviewers' eye movement. *Proceedings of the 2006 Symposium on Eye Tracking Research & Applications* (pp. 133-140). New York: ACM.

Votta, L. G. (1993). Does every inspection need a meeting? *Proceedings of the 1st ACM SIGSOFT Symposium on Foundations of Software Engineering* (pp. 107-114). New York: ACM.

Vygotsky, L. S. (1978). *Mind in society: The development of higher psychological processes*. Cambridge, MA: Harvard University Press.

Wang, Y., Li, Y., Collins, M., & Liu, P. (2008). Process improvement of peer code review

and behavior analysis of its participants. *Proceedings of the 39th SIGCSE Technical Symposium on Computer Science Education* (pp. 107-111). New York: ACM.

Wertsch, J. V. (2007). Mediation. In H. Daniels, M. Cole, & J. V. Wertsch (Eds.), *The Cambridge companion to Vygotsky* (pp. 178-192). New York: Cambridge University Press.

Wiegers, K. E. (1995). Improving quality through software inspections. *Software Development, 3*(4), 55-64.

Williams, L. A., & Kessler, R. R. (2000). All I really needed to know about pair programming I learned in kindergarten. *Communications of the ACM, 43*(5), 108-114.

Wilson, T. D. (2006). A re-examination of information seeking behaviour in the context of activity theory. *Information Research 11*(4). Retrieved from http://informationr.net/ir/11-4/paper260.html

Winograd, T. (2006). Designing a new foundation for design. *Communications of the ACM, 49*(5), 71-73.

Winograd, T., & Flores, F. (1986). *Understanding computers and cognition: A new foundation for design*. Norwood, NJ: Ablex.

Wood, M., Roper, M., Brooks, A., & Miller, J. (1997). Comparing and combining software defect detection techniques: A replicated empirical study. *Proceedings of the 6th European Software Engineering Conference Held Jointly with the 5th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (pp. 262-277). New York: Springer-Verlag.

Appendix 1: The activity checklist (design version)

Means/ends

- People who use the target technology
- Goals and subgoals of the target actions (target goals)
- Criteria for success or failure of achieving target goals
- Decomposition of target goals into subgoals
- Setting of target goals and subgoals
- Potential conflicts between target goals
- Potential conflicts between target goals and goals associated with other technologies and activities
- Resolution of conflicts between various goals
- Integration of individual target actions and other actions into higher-level actions
- Constraints imposed by higher-level goals on the choice and use of target technology
- Alternative ways to attain target goals through lower-level goals
- Troubleshooting strategies and techniques
- Support of mutual transformations between actions and operations
- Goal that can be changed or modified, and goals that have to remain after new technology is implemented
- Parties involved in the process of design
- Goals of designing a new system
- Criteria of success or failure of design
- Potential conflicts between goals of design and other goals (e.g., stability of the organization, minimizing expenses)

Environment

- Role of existing technology in producing the outcomes of target actions
- Tools, available to users
- Integration of target technology with other tools
- Access to tools and materials necessary to perform target actions
- Tools and materials shared between several users
- Spatial layout and temporal organization of the working environment
- Division of labor, including synchronous and asynchronous distribution of work between different locations
- Rules, norms, and procedures regulating social interactions and coordination related to target actions
- Resources available to the parties involved in design of the system
- Rules, norms, and procedures regulating interaction between the parties

Learning/cognition/articulation

- Components of target actions that are to be internalized
- Time and effort necessary to learn how to use existing technology
- Self-monitoring and reflection through externalization
- Possibilities for simulating target actions before their actual implementation
- Support of problem articulation and help request in case of breakdowns

- Strategies and procedures of providing help to colleagues and collaborators
- Coordination of individual and group activities through externalization
- Use of shared representation to support collaborative work
- Representations of design that support coordination between the parties
- Mutual learning of the content of the work (designers) and possibilities and limitations of technology (users)

Development

- Use of tools at various stages of target action "life cycles"—from goal setting to outcomes
- Transformation of existing activities into future activities supported with the system
- History of implementation of new technologies to support target actions
- Anticipated changes in the environment and the level of activity they directly influence (operations, actions, or activities)
- Anticipated changes of target actions after new technology is implemented
- Anticipated changes in the requirements to the system

Source: Kaptelinin, Nardi, & Macaulay, 1999

Appendix 2: Interview guide

Activity: Reviewing code

- Can you show me the last code you reviewed?
- Can you show me how the author asked you to review the code?
- Can you show me how you got access to the code to review?
- Can you show me what you did once you had access to the code?
- Did you record your feedback while reading the code?
- Did you use a checklist or anything else to decide what to look for?
- Can you show me how you gave your feedback to the author?
- Did the author follow up after you gave your feedback?

Activity: Initiating a code review

- Can you show me the last code you had reviewed?
- Why did you decide to have it reviewed?
- Did you stop working on the code while it was being reviewed?
- How did you decide who should review it?
- What did you do to make the code available to the reviewer(s)?
- Can you show me what you made available to the reviewer(s)? Did you post complete files, or snippets of code?
- Can you show me how you instructed the reviewer(s)?

Activity: Acting on review feedback

- Can you show me changes that you made based on the last code review you ran?
- Can you show me any examples of feedback that you chose not to make?
- Did you tie any changes explicitly back to the review, like in your source control change logs?
- Did you follow up with the reviewer(s)?

Perspective: Goals, and actions oriented toward those goals

- Why did you do it that way?
- Were there any other ways you could have done it?

Perspective: Division of labor; rules and procedures

- Was anyone else involved in that?
- Why did you do it that way?

Perspective: Learning, contradictions, and breakdowns

- Has that become easier/harder over time?
- Did you experience any problems?
- Were there any tools that helped you do that?
- Were there any other ways you could have done it?

Perspective: Development

- Can you show me any examples where you did that differently?
- Were there any other ways you could have done it?