Opportunities for Extending Activity Theory for Studying Collaborative Software Development

Cleidson R. B. de Souza[†] and David F. Redmiles
School of Information and Computer Science
University of California, Irvine
{cdesouza, redmiles}@ics.uci.edu

Abstract

Activity theory is an analytical framework that has been used successfully to understand and explain collective work. Software development is, of course, one particular kind of collective work. We used activity theory to analyze the observations one author made during an internship with a large-scale software development group. We also made some observations about how well suited activity theory was for the analysis. We briefly describe the work setting and the analysis. Then we describe the experiences we had, which indicate possibilities for further developing activity theory for studying collaborative work.

1. An Experience with Collaborative Software Development

The first author spent eight weeks during the summer of 2002 interning as a software developer on a large-scale software development team. As a member of this team, he was able to make observations and collect information about a variety of aspects, including the organization of the team, the formal and informal practices that this team adopted, and the tools they used. The software development team was formed to develop a software application we call MVP (not the real name), which comprises 10 different tools that are deployed in different parts of the United States. The source code is approximately one million lines of C and C++.

Each of the several different tools that compose MVP uses a specific set of "processes." A process for the MVP team is a program that runs with the appropriate run-time options. Processes typically run on distributed Sun workstations and communicate by using a TCP/IP socket protocol. Running a tool means running the processes required by this tool, with their appropriate run-time options.

The software development team is divided into two groups: the developers and the verification and validation (V&V) staff. The developers are responsible for writing new code, fixing bugs, and adding new features. This group comprises 25 members. The V&V staff are responsible for testing and reporting bugs identified in the software, keeping a running version of the software for demonstration purposes, and maintaining the documentation (mainly user manuals) of the software. This group comprises six members.

The MVP group adopts a formal software development process that prescribes the steps that need to be performed by the MVP developers during the software development activities. For example, all developers, after finishing the implementation of a change, should integrate their code with the main baseline. In addition, each developer is responsible for testing the code to verify that his/her integration did not insert bugs in the code, or "break the code," as this is informally characterized by MVP developers. After using a configuration management (CM) tool to check-in files into the repository, a developer must send an e-mail to the software development mailing list describing the problem report (PR) associated with the changes, the files that were changed, and the branch where the check-in will be performed, among other pieces of information.

2. An Activity Theory Analysis

Activity theory allows a variety of ways to analyze phenomena. In this work, Engeström's activity theory model [4] was used in the analysis of findings. This model is presented in Figure 1. Activities are associated with objectives called "outcomes." People working within a community share activities. They work to create objects and rely on tools referred to as artifacts to support their activity. Rules instantiate division of labor and practices of the community.

Figure 2 is basically an "instantiation" of the framework described in Figure 1 as applied to the MVP software development team. The main outcome of the software development activity is high-quality MVP software (i.e., bug-free software that is easy to evolve, delivered on schedule, and meets the customers' specifications). Of course, this includes executables, source code, bug repositories, manuals, specifications, and so on. The *object* of this activity is the MVP software while being modified. This includes, for example, the changes being introduced in the code, reported bugs not yet solved, and so on. The *mediating artifacts*, or *tools*, are the set of tools used by the team to manipulate the object so they achieve their goal or outcome, such as CM and bug tracking tools, e-mail, and the like. *Rules* consist of formal practices

[†] Also at the Department of Informatics, Universidade Federal do Pará, Belém, PA, Brazil.

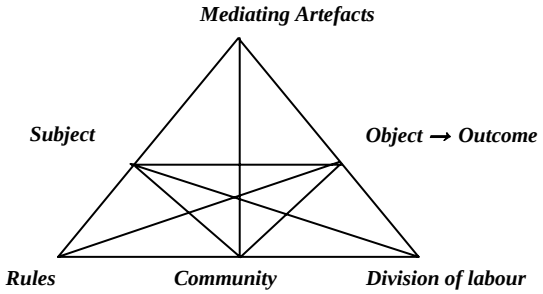

Figure 1: Elements of the Activity Theory Framework (see [4]**).**

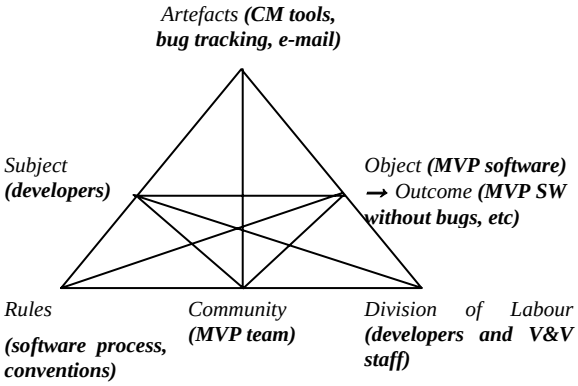

Figure 2: The Software Development Activity as Applied to the MVP Team

(e.g., software development processes) and informal practices (conventions, workarounds, and so on) used by the MVP team. The *community* is the whole MVP team, which is organized according to a specific *division of labor*: There are mainly two groups, namely, developers and V&V staff. But the members of these groups also adopt a division of labor. Specifically, there are process leaders and process developers, the configuration and release manager, the software manager, and testers.

2.1. Tensions and Their "Fixes" in the MVP Team

Contradictions are important aspects in an activity because they might be used as sources of development ([6], pg. 34). In other words, contradictions trigger reflection, thereby helping in the improvement of the activity. Contradictions reveal themselves as breakdowns, problems, tensions, or misfits between elements of an activity or between activities. In our case, we identified several *tensions* within the software development activity developed by the MVP team, but, in addition to that, we also identified the *fixes* that the team adopted to solve them. We identified tensions between different elements, such as between the object and the community, and between the rules and the community.

In the first case, the tension between the object and the community exists due to the effects that the object (e.g., changes in the MVP software) will have on the community. For example, if a change (the object) is introduced in the source code, other members of the MVP team (the community) might need to be informed because they may need to perform additional tasks (e.g., update the documentation) due to that change. The tension exists because developers are not aware of some interdependencies in the software and, therefore, how other members of the community are affected by their work. Despite that, the community must support the evolution of the software and guarantee that the software delivered is not inconsistent with the specifications, manuals and other artifacts.

In the second case, the tension exists basically between rules and the community because one rule suggests that a developer should perform a specific action, but he/she does not want to perform that action out of concern for the effects of the action on the rest of the community. For example, if one developer decides to check-in his/her code into the repository, the other developers (part of the community) might need to recompile their code in order to work with the latest version of the software, and this compilation process is time-consuming.

2.2. Tensions between the Object and the Community

In this case, tensions emerge in the software development activity due to the concern about how the object will affect the community. For example, when the source code is modified, often it is also necessary to modify other software artifacts, such as manuals, documentation, specifications, and so on, or inconsistencies will arise. Although inconsistencies might have positive effects in software development, in general they are not desirable [10]. The MVP software development team already recognized the need to handle this problem (tension) and adopted two different and complementary practices to deal with it: Formal reviews are adopted in the software development process to handle inconsistencies in the source code, and problem reports are structured in such a way that the inconsistencies between source code and other artifacts are easier to manage. Both practices are explained in the following sections.

2.3. Tensions between the Rules and the Community

These tensions occur because a rule might suggest that a developer should perform a specific action, but the developer does not want to perform it due to concern about the effect of this action on the community. As mentioned earlier, an example of such tension occurs when one developer needs to check-in his/her code into the repository, but the other developers would then need to recompile their code in order to work with the latest

their code in order to work with the latest version of the software. Because this compilation process is time-consuming, the developer needs to decide whether to follow the rule and thus cause the whole community to recompile, or to not follow the rule, at least for a while, thereby minimizing the impact of his/her actions in the rest of the community. Typical fixes adopted by the MVP team include changing the order in which some rules are executed or performing additional actions along with the rule to minimize the disruption to the community.

Furthermore, tensions between these components also arise due to the impact on the community in the execution of the rule. In other words, the developer is concerned that he/she needs to perform a rule but actions of the community (such as check-ins or check-outs) will impact his/her performance of the rule. In this case, those actions influence how the developer performs the rule. Note that in this case, the division of labor also influences this tension because it prescribes how developers should be organized in the community, therefore allowing two or more developers to work and check-in in concurrently.

3. Implications for Activity Theory

3.1. Modeling Human Activity

In software development terms, section 2 of this paper developed a model. The process of developing this model has more similarities to software modeling than one might expect. In particular, we began by choosing a modeling language that seemed appropriate for our application—the language of activity theory, and in particular Engeström's terminology and diagrammatic notation. We then built an instance of a model in this language that served as a first approximation. We then refined it through several iterations. We reached a point at which analysis of the model yielded explanations consistent with the data, as presented above.

Iterative refinement of the model appeared to be an open-ended process. However, the actual observations made during the internship acted in a sense like a "test oracle." Namely, we reached a stopping point when all observed phenomena were accounted for. Moreover, the focus of activity theory on identifying tensions and conflict were useful for understanding what we observed and for highlighting areas where software tools and practices might be improved.

In sum, the attempt to model the human collective activity of collaborative software development did not seem straightforward at first, but required a first approximation and successive refinement. Although frustrating, the challenges did not seem greater than other kinds of modeling, and the results were informative. In the next subsection, we make some observations on how this process may be improved and identify research areas for the methodology.

3.2. Activity Theory: Where Next?

Activity theory has been applied to the design of software systems, and research to date has indicated its usefulness toward collecting requirements for software system design (e.g., [1] and [8]). However, to the authors' knowledge, this paper represents the first application of activity theory to studying collaboration among software developers; previous studies have examined only the collaboration between end users and software developers. Thus, we had to struggle with a finer degree of detail of activity than previous works with respect to the development of software.

One challenge that presented itself was the notion that a single activity might be consistent when observed as a single instance, but might be a source of tension when there were multiple instances of that activity. For instance, in the case of a single developer, even when working with end users and other team members, the activity of checking-in a module revision is consistent within itself. However, multiple instances of this check-in activity create a tension we observed as developers sped up their work to be the first to check-in. This part of the model and the more general issue of multiple instances of activity is one place for further research into the application of activity theory and a potential contribution to improving the methodology.

Another area for research in activity theory is akin to dependency analysis in software testing. Namely, as we identified different activities that comprised the general activity of evolving a software system, we began to observe many interdependencies. For example, rules for applying a specific software tool led to other activities, each with their own associated set of rules, subjects, other tools, and so on. We were intrigued by the notion that a kind of dependency analysis might be developed to help an organization more precisely account for the potential impact of making changes to tools and practices. This kind of work, however, would be a long-term goal. A related issue is that of adoption. Understanding the history of how elements in the activity theory models evolved (e.g., tools, rules, division of labor, and so on) can better enable the responsible introduction of new tools, including involving end users with tool introduction. The basic premise of introducing changes into people's work is the ability to develop the fullest understanding possible of that work. Activity theory, even in its present state of development, is successful in that regard.

Finally, a new line of research is beginning to present itself around the concepts of reflection and awareness. Specifically, various researchers have begun to recognize the value of simply reflecting back to a group or organization the actuality of its various objectives and activities. In a previous study, we used this kind of reflection as a matter of course in reporting findings, but the process of performing this "reporting" led to improvement in the

process of software developers collecting requirements and in the organization's members better understanding one another's roles [2]. Other researchers have observed similar effects, including those at a small scale. Namely, some researchers are developing software tools to help people coordinate their collaborative work by reflecting the current state of a collaborative activity or the state of actual collaborators. Some instances are Portholes systems that reflect the state of collaborators [3] [7], configuration management tools that reflect who is working on what modules [9], and tickertape tools that reflect all activities in a work environment [5]. Thus, another open area is better understanding and better reflecting of actual activity (through manual and automated means) back to participants in that activity, and understanding ways this has positive effects on the collective work.

4. Conclusions

Our experiences in performing the analysis presented briefly in this paper as well as previous experiences of our own and our colleagues have shown many positives to activity theory. It is open ended, which, although a challenge, allows for the introduction of new ideas and refinements. It is noninvasive, using open-ended interviews or even more informal observations of work such as presented in this paper. It readily yields to iterative refinement. When more detail is needed in a model, additional activities may be named and analyzed. Finally, there seems to be some overlap in object-oriented analysis. Although the present authors do not wish to overemphasize the similarities, the overlap is helpful for people with object-oriented experience to engage in learning the methodology. Thus, although there is still a great deal of craft involved in becoming acquainted with and applying activity theory, we have experienced many positives in our analyses in different work settings and anticipate the methodology becoming more refined and documented.

Acknowledgments

The authors thank CAPES (grant BEX 1312/99-5) and NASA/Ames for financial support. This effort was also sponsored by the Defense Advanced Research Projects Agency (DARPA) and Air Force Research Laboratory, Air Force Materiel Command, USAF, under agreement number F30602-00-2-0599. Funding also was provided by the National Science Foundation under grant numbers CCR-0205724 and 9624846. The U.S. Government is authorized to reproduce and distribute reprints for governmental purposes notwithstanding any copyright annotation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Defense Advanced Research Projects Agency (DARPA), the Air Force Laboratory, or the U.S. Government.

5. References

[1] Bodker, S., *Through the Interface: A Human Activity Approach to User Interface Design*, Hillsdale, NJ: Lawrence Erlbaum, 1991.

[2] Collins, P., Shukla, S., et al., "Activity Theory and System Design: A View from the Trenches," *Computer Supported Cooperative Work—Special Issue on Activity Theory and the Practice of Design*, vol. 11, pp. 55-80, 2002.

[3] Dourish, P., and Bly, S., "Portholes: Supporting Distributed Awareness in a Collaborative Work Group," ACM Conference on Human Factors in Computing Systems (CHI '92), Monterey, CA, 1992.

[4] Engeström, Y., "Activity Theory and Individual and Social Transformation," pp. 19-38, in Engeström, Y., Miettinen, R., and Punamäki, R-L., "Perspectives on Activity Theory." Cambridge, UK: Cambridge University Press, 1999.

[5] Fitzpatrick, G., Mansfield, T., et al., "Augmenting the Workaday World with Elvin," 6th European Conference on Computer Supported Cooperative Work, pp. 431-450, Copenhagen, Denmark, 1999.

[6] Kuuti, K., "Activity Theory as a Potential Framework for Human-Computer Interaction Research," pp. 17-44, in Nardi, B., "Context and Consciousness: Activity Theory and Human-Computer Interaction." Cambridge, MA: The MIT Press, 1996.

[7] Lee, A., and Girgensohn, A., "NYNEX Portholes: Initial User Reactions and Redesign Implications," ACM Conference on Human Factors in Computing Systems (CHI '97), pp. 385-394, 1997.

[8] Nardi, B., and Redmiles, D., Eds. Computer Supported Cooperative Work, *The Journal of Collaborative Computing, Special Issue on Activity Theory and the Practice of Design*, Vol. 11, No. 1-2, p. 1-11, 2002.

[9] Sarma, A., Noroozi, Z., et al., "Palantír: Raising Awareness among Configuration Management Workspaces," Twenty-fifth International Conference on Software Engineering, pp. 444-453, Portland, Oregon, 2003.

[10] Spanoudakis, G., and Zisman, A., "Inconsistency Management in Software Engineering: Survey and Open Research Issues," in *Handbook of Software Engineering and Knowledge Engineering*, vol. 1, S. K. Chang, Ed.: World Science Publishing Co., 2001, pp. 329-380.