Using Activity Theory to Understand Contradictions in 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. In this paper, we use activity theory to analyze the collaborative work of a software development team. Using this framework, we were able to identify different tensions within and contradictions between activities performed by the developers in the team, including software tools and practices. We argue that these tensions and contradictions illuminate opportunities for improvements in the work and for software engineering researchers. Additionally, we believe that the successful application of activity theory to understanding collaborative software development is a step towards further understanding this framework and adapting it to general use.

Keywords*: Computer-supported cooperative work, requirements engineering, human-computer interaction, maintenance and evolution, activity theory, and team work.*

1. Introduction

Software engineers have sought for quite some time to understand their own work of software development as an important instance of cooperative work, especially seeking ways to provide better software tools to support developers [5]. However, before developing tools, one needs to properly understand different factors that influence the adoption and use of these tools, such as organizational aspects, the environment where the work is performed, the developers and their history as a community, and the practices used by these developers and so on [23]. Furthermore, the inclusion of the users has been shown to be an important factor for software tool adoption. All these factors need to be taken into account to avoid that the introduction of the tools disrupt the workplace, leading to unsuccessful tool adoption.

The HCI / CSCW community has applied different approaches to guarantee that the aforementioned factors are not disregarded during the analysis of workplaces. Examples of such approaches are ethnomethodology [13], distributed cognition [15], activity theory [20] [4], among others.

In this paper, we use the activity theory framework to analyze the collective effort of a collaborative software development team. Using this framework, we were able to identify different tensions within the activity and contradictions between activities performed by the developers in the team. We argue that these tensions and contradictions illuminate opportunities for improvements in the work, through better software tools and practices. Additionally, we believe that the successful application of activity theory to understanding collaborative software development is a step for further understanding this framework and adapting it for more general use.

The rest of the paper is organized as follow. Sections 2 and 3 describe the setting where the software development team is located and methods used to study it. Section 4 describes the analysis of our observations using activity theory. More specifically, tensions within elements of the software development activity are described, as well as, the "fixes" that team members adopted to handle these tensions. The following section, 5 presents the discussion about the tensions and fixes identified and their implications for software engineering tools. Section 6 discusses the implications for future use of the activity theory framework in the analysis of software development efforts. Finally, conclusions and future work are discussed.

2. The Setting

2.1. Introduction

The first author spent eight weeks during the summer of 2002 interning as a software developer of a large-scale software development team at the NASA / Ames Research Center. 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 that they used. The software development team develops a software application we will call MVP (not the real name), which is composed of ten different tools that are deployed in different parts of the United

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

States. The source code is approximately one million lines of C and C++.

2.2. MVP Software

As mentioned before, MVP is composed of several different tools. Each one of these tools uses a specific set of "processes." A process for the MVP team is a program that runs with the appropriate run-time options and it is not formally related with the concept of processes in operating systems and/or distributed systems. Processes typically run on distributed Sun workstations and communicate using a TCP/IP socket protocol. Running a tool means running the processes required by this tool, with their appropriate run-time options.

2.3. The Software Development Team

The software development team is divided into two groups: the V&V staff and the developers. The developers are responsible for writing new code, for bug fixing, and adding new features. This group is composed of 25 members, where three of them are also researchers that write their own code to explore new ideas. The experience of these developers range between 3 months to more than 25 years. Experience within the MVP group ranges anywhere between 2½ months to 9 years. This group is spread out into several offices across two floors in the same building.

V&V members are responsible for testing and reporting bugs identified in the software, keeping a running version of the software for demonstration purposes and for maintaining the documentation (mainly user manuals) of the software. This group is composed of 6 members. Half of this group is located on the V & V Laboratory, while the rest is located in several offices located in the same floor and building as this laboratory. Both, the V&V Lab and developers' offices are located in the same building.

2.4. The Software Development Process

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 its code to verify if his integration did not insert bugs in the code, or, "break the code", as informally characterized by MVP developers. After checking-in files in the repository, a developer must send an e-mail to the software development mailing list describing the problem report associated with the changes, the files that were changed, the branch where the check-in will be performed among other pieces of information.

2.5. Division of Labor in the MVP group

As mentioned before, each MVP tool uses a specific set of processes with their run-time options. Processes are used to divide the work, i.e., each developer is assigned to one or more processes and tends to specialize on it. For example, there are process leaders and process developers, who, most of the time, work only with this process. This is an important aspect because it allows these developers to understand its behavior more deeply and familiarize with its structure, therefore helping them in dealing with the complexity of the code. Indeed, during the software development activity, managers tend to assign work according to these processes to facilitate this learning process. However, it is not unusual to find developers working on different processes. This might happen due to different circumstances. For instance, before launching a new release, the entire workforce is needed to fix bugs in the code. Another reason for allowing one developer to work in a different process is the complexity of the code. One bug might seem to be located in a process and therefore it is allocated to the developer who works with this process. But, later he might find out that the bug actually is located in another process. In this case, it is better to let the developers finish the work since so much time was invested in it. As a side effect, this allows the developers to have a general view of the MVP software, understanding other processes. Indeed, according to the MVP software manager:

> *"(…) while we want to try to keep people concentrated on their process (…) so they get to know them really well, on the other hand, it's always nice for them to go outside of it and take a look and see what's going on in some of the other processes, gives them a better understanding of how MVP works."*

3. Methods

3.1. Data Collection

As mentioned in section 2, the first author spent eight weeks during the summer of 2002 as a member of the MVP team. As a member of this team, he was able to make observations and collect information about several aspects of the team. He also talked with his colleagues to learn more about their work. Additional material was collected by reading manuals of the MVP tools, manuals of the software development tools used, formal documents (like the description of the software development process and the ISO 9001 procedures), training documentation for new developers, problem reports (PR's), and so on.

Some of the team members agreed to let the intern shadow them for a few days so that he could learn about

their functions and responsibilities better. These team members belonged to different groups and played diverse roles in the MVP team: the documentation expert, some V&V members, leaders, and developers.

3.2. Data Analysis: Activity Theory

Activity theory is a way of examining phenomena in the world by considering the relationships of agents, objects, means, and in the case of human collective activity, objectives, community, rules, and division of labor [7]. Activity theory was pioneered in the 1920s and 30s by the psychologists, Vygotsky, Leontiv, and Luria and today is actively being developed into a methodology of analysis and design by a large community of researchers. A good introduction to the origins, methodology, state of the research and applications, as well as current researchers in the field is provided by the collections of articles by Engeström, Miettinen, and Punamäki [9], by Nardi [20], and by Nardi and Redmiles [21].

The activity theory framework allows a variety of ways of analyzing phenomena. For example, Collins *et al.* [4] emphasizes Engeström's triadic model of individual and collective activity [7]. This model suggests important information to the analyst through the identification of contradictions. Contradictions reveal themselves as breakdowns, conflicts, problems, tensions or misfits between elements of an activity or between activities [17]. The tensions identified by Collins *et al.* had important implications for tools, practices, and division of labor for the staff members. A different approach might apply the three-level hierarchical structure of a collaborative activity proposed by Engestrom [8]. This approach is used by Barthelmess and Anderson [2] to analyze the capabilities of software engineering environments.

In this study, Engeström's activity theory model [7] 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.

4. The MVP Software Development Activity

4.1. Introduction

To begin, we will describe the software development activity as performed by the MVP team. Figure 2 is basically an "instantiation" of the framework described in Figure 1 as applied to the MVP software development team.

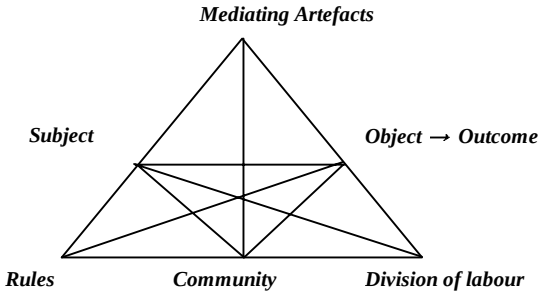

Figure 1: Elements of the Activity Theory Framework (see [7]**).**

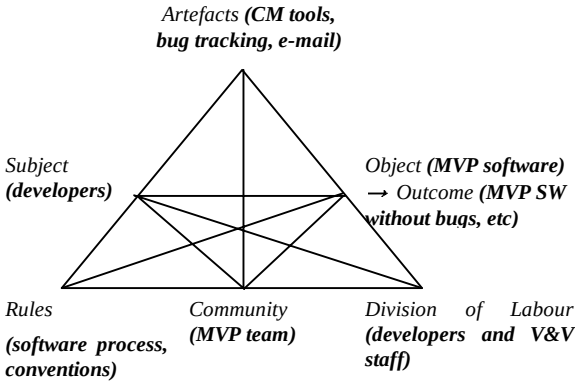

Figure 2: The Software Development Activity as applied to the MVP Team

The main outcome of the software development activity is the high-quality MVP software, i.e., bug-free software that is easy to evolve, delivered on schedule, and meeting the customers' specifications. Of course, this includes executables, source code and 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 that they achieve their goal or outcome, such as configuration management and bug tracking tools, e-mail, etc. *Rules* consist of formal practices (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. There are process leaders and process developers, the configuration and release manager, the software manager, testers, and so

on.

4.2. Tensions and their "Fixes" in the MVP Team

As mentioned in the previous section, according to the activity theory framework, contradictions are important aspects in an activity because they might be used as sources of development ([17], pg. 34). In other words, contradictions trigger reflection; therefore 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: between the object and the community, and between the rules and the community.

In the first case, the tension exists because of 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 because of that (e.g. update the documentation). 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, basically, the tension exists between rules and the community because one rule suggests that a developer should perform a specific action, but he does not want to perform that because he is concerned about the effect of his action in the rest of the community. For example, if one developer decides to check-in his 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. In the rest of this section, we will describe these breakdowns in more detail and the "fixes" that the MVP team adopted.

4.3. Tensions between the Object and the Community

In this case, tensions emerge in the software development activity because of the concern of 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. Otherwise, inconsistencies will arise. While inconsistencies might have positive effects in software development, in general they are not desirable [27]. The MVP software development team already recognized the need to handle this problem (tension) and adopted two different and complementary practices that deal with this problem: formal reviews are adopted in the software development process to deal with inconsistencies in the source code; and problem reports (PR's) that are structured in such a way that the inconsistencies between source code and other artifacts are easier to manage. Both practices will be explained below.

4.3.1. Adoption of Formal Reviews

The software process adopted by the MVP team prescribes the usage of two types of formal reviews: *code reviews* before the integration of any change, and *design reviews* for major changes in the software. Code reviews are performed by the manager of each process. Therefore, if a change involves, two processes, a developer's code will be reviewed twice: one by each manager of these two processes. In addition to that, the code is also reviewed by the general software manager before the closure of the problem report associated with that change. In the first review, one of the goals of the process manager is to guarantee that the change does not affect other parts of the code of that process. In the second review, one of the goals of the software manager is to guarantee that the changes do not "break" the overall architecture of the system, i.e., the software manager will check if the new change will not generate side effects on other parts of the code, which will eventually lead to the recompilation.

On the other hand, design reviews are adopted for changes that involve major reorganizations of the source code. The need of design reviews is decided by the software manager depending on each problem report being worked. Similarly, the purpose of these reviews is to understand, possibly avoid, or minimize the effects of the changes in the source code.

By carefully inspecting the changes (*object*) that are introduced in the source code, the MVP team tries to minimize the side-effects that the *community* is going to experience because of these changes.

4.3.2. The Structure of Problem Reports

In our analysis we identified that the structure of the problem reports (PR's) in the bug tracking tool is very useful in facilitating the coordination of the MVP team. To be more specific, in addition to support bug tracking, PR's also facilitate the management of interdependencies among the artifacts of the MVP software. PR's are used by end-users liaisons, developers and testers for different purposes. For example, when a bug is identified, it is associated with a specific PR. The person who identified

the bug is also responsible for filling a field in the PR describing 'how to repeat' the bug, i.e., the dataset used, the tools and their parameters, etc. This description is used by the developer assigned to fix the bug to understand the circumstances under which the bug appears. After fixing the bug, this developer must fill a field in the PR that describes how the testing should be performed to properly validate the fix. This field is called 'how to test'. This information is used by the test manager, who creates test matrices that will be later used by the testers during the regression testing. The developer who fixes the bug also indicates in another field of the PR if the documentation of the tool needs to be updated. Then, the documentation expert uses this information to find out if the manuals need to be updated based on the changed the PR introduced. Finally, another field in the PR conveys what needs to be checked by the manager when closing it. Therefore, it is a reminder to the software manager of the aspects that need to be validated. In short, MVP members use information from the PR's depending on the role they are playing.

MVP developers reported that using this approach, they were able to manage the interdependencies between the source code and other artifacts of the software development activity, such as test cases, manuals, and so on. Again, the tension between the object (the MVP software) and the community is addressed, so that changes in the object can be easily accompanied by the respective actions of other members of the community, so that the additional changes in other objects can be performed.

4.4. Tensions between the Rules and the Community

These tensions occur because a rule might suggest that a developer should perform a specific action, but he does not want to perform that because he is concerned about the effect of his action in the community. As mentioned earlier, an example of such tension occurs when one developer needs to check-in his code into the repository, but the other developers will need to recompile their code in order to work with the latest version of the software, and this compilation process is time-consuming. Therefore, the developer needs to decide if he will follow the rule and cause the whole community to recompile, or he is not going to follow the rule, at least for a while, thereby minimizing the impact of his actions in the rest of the community. Typical fixes adopted by the MVP team include change the order in which some rules are executed, and perform additional actions alongside with the rule to minimize the disruption of the community.

Furthermore, tensions between these components also arise because of the impact that the Community will have in the execution of the Rule. In other words, the developer is concerned that he needs to perform a rule but actions of the community (like check-in's, check-out's, etc) will impact his 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. This situation is described in more details in section 4.4.3.

4.4.1. Changing the order of execution of the Rules

The MVP group adopts a formal software development process that prescribes that *after* checking-in files in the repository, a developer must send an e-mail to the software development mailing list describing the problem report associated with the changes, the files that were changed, the branch where the check-in was performed, and other details. However, we found out that MVP developers perform these activities in the inverse order, i.e., they will send e-mail *before*, not after, the check-in. By doing that, MVP developers allow their colleagues to prepare for the changes that they are about to commit. Indeed, developers might even send e-mail to the author of the change asking him to delay its check-in.

We also observed that MVP developers often engage in parallel development, i.e., two or more developers make changes in the same file concurrently. The changes are performed in their local versions, which had been checked-out of the repository. Therefore, if a developer needs to synchronize his modified version of a file with the latest version of that file, he needs to perform a merge between these two versions. However, if several additional versions were created by different check-in's after the file has been checked-out, the "difference" between the working version and the latest version might be too large, and the merging algorithm might not work properly. Furthermore, the working version of a developer might become outdated, which might lead to conflicting changes in the code among other problems. MVP developers are aware of these problems and adopted a practice called "partial check-in's" to handle this situation. Basically, a developer checks files back into the main repository before finishing his entire work with the PR associated with those files, instead of having to wait until his work is entirely done. However, this practice is only adopted in files with a high degree of parallel development, i.e., files that are often modified by different developers, which leads to the creation of several versions. This is another example of tension that is created between tools (CM) and a rule (check-in only after code reviews). Again, the solution adopted by the community is to change the order of the Rules: partial check-in's means that the check-in's are performed

before the code reviews.

A different situation leads MVP developers to change the order in which a rule is executed. In this case, as mentioned in the previous section, MVP developers might add information about the need to recompile part of the system after their check-in is performed. MVP developers are aware that the recompilation process is time-consuming, up to 30 to 45 minutes. They want to finish their changes (PR), but they do not want to disturb the whole community by forcing them to wait for the end of the recompilation. Therefore, MVP developers may hold check-in's until the evening when most of the developers are already gone. Note that the CM tool used by the MVP team allows developers to choose if they want to incorporate other's changes, meaning that they are able to decide if they want to recompile the code or not. Despite that, they still adopt this practice of "holding check-in's". By doing that, they minimize the number of other developers that will be affected by their actions. According to one of the developers:

> *"(…) and the other thing that you find is that when people also know that if they are going to check-in a file they will do in the later afternoon … you're gonna do a check-in and this is gonna cause anybody who recompiles that day have to watch their computer for 45 minutes (…) and most of the time, you're gonna see this coming at 2 or 3 in the afternoon, you don't see folks (….) you don't see people doing* [file 1] *or* [file 2] *checking-in at 8 in the morning, because everybody all day is gonna sit and recompile."*

4.4.2. Performing additional actions alongside the Rule

Our observations suggest that developers, while writing the e-mail to be sent to the mailing list, also describe the impact that their changes will have on others' work. In other words, the software process (rule) prescribes that some information needs to be sent to the mailing list. However, MVP developers include additional information to this e-mail, which allows other developers to prepare and reflect about the effect of their colleagues' changes in their current work, since they are aware of some of the interdependencies in the source-code. Consequently, they might adjust themselves to these changes.

Often, when another developer reads one of these e-mails, he might walk to the co-worker's office to ask about the changes or, if the change has already been committed, browse the CM and bug tracking systems to understand them. The following list presents some usual comments sent by MVP developers:

> *"No one should notice."*
>
> "[description of the change]*: only* [Tool name] *users will notice any change."*
>
> *"Will be removing the following* [x] *files. No effect on recompiling."*
>
> *"Also, if you recompile your views today you will need to start your own* [z] *daemon to run with live data."*
>
> *"The changes only affect* [y] *mode so you shouldn't notice anything."*
>
> *"If you are planning on recompiling your view this evening ([current date]) and running an MVP tool with live* [z] *data you will need to run your own* [z] *daemon."*

Based on our observations, we identified two types of impact statements: changes in run-time parameters of a process and the need of recompilation of parts or the whole source code. The former case is very important because other developers might be running the process that is changed. The latter case is also necessary because when a file is modified, it will be recompiled, as well as, the other files that depend on it.

4.4.3. Speeding-up the Rule

MVP developers sometimes rush to test and check-in their changes because they want to do that *before* somebody else performs another check-in. If somebody checks in any code, the developer needs to repeat his testing to guarantee that his changes will not inexplicably interact with the changes previously checked in and introduce errors in the source code. As one developer plainly pointed out: "This is a race!" We observed that this testing is very informal. For example, developers will sit in the V&V laboratory and compare the current version of MVP to the one with changes. In short, MVP developers do not use regression testing at this moment. This type of testing will be used by the V&V staff at a different time, i.e., before launching a new release.

Although we observed that some check-in's introduced errors in the source code, we do not have evidence that these errors were introduced because of this racing. "Speeding up" the process is a fix employed by the MVP developers because of the tensions between the community (e.g., their actions) and the object (the source code changes, for examples). In other words, the community affects the object because actions performed by other members of the community need to be accompanied by changes in the object.

5. Implications for Software Tools

As described in the previous section, we observed two types of tensions in the software development activity performed by the MVP team: between the object and the community and between the rules and the community. By closely examining these tensions, it was possible to identify a common concern with the *impact* of the object (or rule) in the community and the *impact* of the community on the object. For example, PR's used by the team are structured in such a way that all changes in the source code (object) are accompanied by the indications

of their impact on the other developers' work (the community). Similarly, when one developer performs a "partial check-in," he wants to avoid that the community (through their check-in's and check-out's operations) impacts the changes (object) that he is currently developing. In short, the impact on the community is either caused by the *object* or by the *actions* of other members of the community when they follow the rules. On the other hand, the impact on the object is caused by the actions of members of the *community*.

Furthermore, as discussed in the previous section, the MVP software development team adopted fixes in order to minimize the aforementioned tensions. By adopting these fixes, the MVP team is implicitly recognizing the importance of managing such impacts. In other words, the community wants to manage the impact of both other developer's *changes* and *actions* in their own work. However, current software engineering tools, with a few exceptions, have focused exclusively in controlling the impact of the changes that other developers introduce. Indeed, there are several techniques available to support change impact analysis [1]. Impact analysis (IA) it the activity of identifying what to modify to accomplish a change, or of identifying potential consequences of a change. One can easily notice that IA techniques require the existence of a proposed change in the software (an object). Of course, the object is result of other developers' actions, but, these techniques analyze exclusively the object. The impact of other's actions is not addressed, and it is especially important because it helps in the coordination of their work. And, coordination, is one of the major causes of problems in software development [5].

One example of impact analysis techniques is dependency graph approaches, which focus on determining the impact of the code (product) in other's part of the source code. These approaches are usually based on program dependences, which are syntactic relationships between the statements of a program that represent aspects of the program's control flow and data flow [24]. In other words, they focus only in determining the impact of the product in the rest of the cooperative effort. Another example of IA approaches are regression testing techniques (such as [25]) that attempt to minimize the number of the test cases necessary to validate the changes in the software. Although powerful, these techniques could not be used by MVP developers to determine whether the tests that they need to run can be impacted by another developer's changes, since the MVP team only performs regression testing before launching a new release. Other examples of IA approaches are program slicing techniques [28] that might be to determine the program subset that can affect the value of a given variable.

While there are several approaches for dealing with the impact of the *object* (or product) of a software developer's work in other developer's work, there are only a few that support the analysis of the impact of other developers' *actions*. To the best of our knowledge, the closest approach is process-centered software engineering environments [11]. These environments use process models to describe, among other things, agents (that perform process steps) and the order in which these steps need to be performed to achieve team coordination. This means that actions that need to be performed by software developers are considered during the definition of the process model. Furthermore, when these models are built, they also take into account the constraints of the tools used by the members of the community, the expertise and experience of members of that community (i.e., its history), among other aspects. The enactment of the process is then monitored, which means that when developers do not perform these activities, process deviations are reported. However, these deviations are used by project managers to assess high-level attributes of the software development activity such as scheduling and costs. Developers' actions (including deviations) are not used by these environments to assess the low-level impact that they will cause on other developers' work.

Providing information about the impact of other's action is a very difficult problem. Indeed, the current approach is to provide the information to the developer, and let him figure out how his work will be impacted. Examples of this approach can be found on configuration management tools that, in addition to presenting information about the products (files and their versions), also present information about other developers' actions on these products (check-in's, check-out's, merges, etc), therefore increasing the awareness among developers, which ultimately facilitate the coordination of their activities [12]. However, the delivery of this information is problematic, since users need, pro-actively, access the CM tool. Recent work in cooperative software engineering tools (e.g., Palantir [26] and Night Watch [22]) attempt to overcome this limitation by delivering events happening in one CM workspace to other workspaces that are accessing the same files. The problem with these tools, though, is that the information that is delivered is related to only a subset of actions that occur in the software development process: they focus on the actions of developers who are working in the same files (parallel development). Information about the current status of other developers is not presented in these tools, i.e., information about the part of software development process that deals with check-in's, for example, is not presented. By doing that, they present information to the users that is out of context, since as the activity theory illustrates, an activity is a rich set of

interdependent elements that include rules, a community, tools that mediate the interaction of the subject with the object, and so on. Similarly, notification-servers that focus on delivering awareness information, such as CASSIUS [16] and Kronika [19] usually take into consideration only some elements of an activity such as tools, objects and subjects. Other elements that are also important for providing coordination are not considered in these servers.

Having said that, we can conclude that providing impact analysis of both actions and objects are necessary to help in the coordination of a collaborative software development effort. Indeed, that is exactly what the fixes adopted by the MVP team indicate to us! Therefore, collaborative software engineering tools need to be able to inform the "context" of the software development effort, so that they can allow developers to determine how best to coordinate. For example, MVP developers speed-up during their informal testing activities because they do not want to redo their work when somebody else's check-in something in the repository. If these developers were aware of the current status of other developers, for example, by visualizing a small process model that describes the check-in activity, they would be able to coordinate with their peers the appropriate moments for check-in's and testing, therefore avoiding the need for rushing.

6. Implications for Activity Theory

6.1. Modeling Human Activity

Section 4 of this paper developed a model. The process of developing this model has more similarities to software modeling than one might expect. In particular, as explained in Section 3.2, 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 where 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 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.

6.2. Activity Theory: Where Next?

Activity theory has been applied to the design of software systems, and research to date has indicated its usefulness towards collecting requirements for software system design (e.g., [21] and [3]). However, to the authors' knowledge, this paper represents the first application of activity theory to studying collaboration among software developers, whereas previous studies have examined the collaboration between end users and software developers. Thus, we had to struggle with a finer degree of detail of activity, with respect to the development of software, than previous works.

One challenge that presented itself was the notion that a single activity might be consistent when observed as a single instance, but be a source of tension when there were multiple instances of that activity. Such was the case with developers speeding up for check-ins. 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, etc. 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 – tools, rules, division of labor, etc. – 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 begin 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 understanding one another's roles better [4]. Other researchers have observed similar effects including 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 [6] [18], configuration management tools that reflect who is working on what modules [26], and tickertape tools that reflect all activities in a work environment [10]. 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.

6.3. Why Not Another Methodology?

Detailed comparisons of activity theory and other methodologies may be found elsewhere and are well done (see e.g. articles by Halverson and Nardi in [21], pp. 243-275). Briefly, activity theory is most often compared to two methods of analysis, distributed cognition [15] and contextual inquiry [14]. Activity theory places an emphasis on identifying activities as primary elements of analysis and organizes other components (such as tools and people) around the activities they are involved in. Distributed cognition places a focus on artifacts in the work environment and how they affect and enable work. Contextual inquiry is a structured interview technique and process to evolve interviews into requirements for systems, especially supportive of user interface design.

Although it is not unusual for researchers to become proponents of one approach over all others, the present authors do not see the need for over specialization of the field at this time. All the aforementioned methods are relatively new. There is much to be learned about their application to work environments. Moreover, the kinds of work computer professionals are delving into are ever changing, placing constant demands not just on technology but on methods for understanding work context. Even though activity theory is one of the oldest methods, being datable to at least 80 years prior, its application to work involving computing tools is only recent. Moreover, different field methods suit different application areas differently. Factors that should be used in making decisions include the objective of the observations and analysis, preferences of participants (observers and observed), and, very importantly, organizational contexts.

Our experiences performing the analysis presented in this paper and previous experiences of our own and our colleagues have shown many positives to activity theory. It is open ended, which though 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, while 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.

7. Conclusions

This paper reported a successful application of the activity theory to analyze the collective effort of a software development team. More specifically, we adopted Engeström's triadic model of individual and collective activity in order to focus on the collaborative aspects of the software development effort. Adopting this framework allowed us to deeply understand the context where the software development activity was being performed: the tools used by the community, the division of labor adopted by the team, the desired out-come of the activity, among other factors. Furthermore, we were also able to better understand this framework and identify aspects that may be refined so that it can be broadly applicable in software development. Finally, by focusing on the tensions and contradictions in the activity, we were able to identify opportunities for improvements in the work, through better collaborative software engineering tools and practices. This is possible because activity theory helps observers identify tensions and contradictions as opportunities for reflection and evolution.

Acknowledgments

The authors thank CAPES (grant BEX 1312/99-5) and NASA/Ames for the financial support. Effort 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 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.

8. References

[1] Arnold, R. S. and Bohner, S. A., "Impact Analysis - Towards a Framework for Comparison," International Conference on Software Maintenance, pp. 292-301, 1993.

[2] Barthelmess, P. and Anderson, K. M., "A View of Software Development Environments Based on Activity Theory," *Computer Supported Cooperative Work (CSCW) - Special Issue on Activity Theory and the Practice of Design*, vol. 11, pp. 13-37, 2002.

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

[4] 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.

[5] Curtis, B., Krasner, H., et al., "A field study of the software design process for large systems," *Communications of the ACM*, vol. 31, pp. 1268-1287, 1988.

[6] 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.

[7] Engeström, Y., "Activity Theory and Individual and Social Transformation," pp. 19-38, in [9], 1999.

[8] Engeström, Y., "Coordination, Cooperation, and Communication in the courts," in *Mind, Culture, and Activity*, M. Cole, Y. Engeström, and O. Vasquez, Eds. Cambridge, UK: Cambridge University Press, 1997.

[9] Engeström, Y., Miettinen, R., et al., "Perspectives on Activity Theory." Cambridge University Press, 1999.

[10] 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.

[11] Garg, P. K. and Jazayeri, M., "Process-Centered Software Engineering Environments." Los Alamitos, CA: IEEE Computer Society Press, 1996.

[12] Grinter, R. E., "Using a Configuration Management Tool to Coordinate Software Development," Conference on Organizational Computing Systems, pp. 168-177, 1995.

[13] Heath, C. and Luff, P., "Collaboration and Control: Crisis Management and Multimedia Technology in London Underground Control Rooms," *Computer Supported Cooperative Work*, vol. 1, pp. 69-94, 1992.

[14] Holtzblatt, K. and Beyer, H., "Contextual Design," *ACM Interactions*, vol. 6, pp. 32-42, 1999.

[15] Hutchins, E., *Cognition in the Wild*. Cambridge, MA: The MIT Press, 1995.

[16] Kantor, M. and Redmiles, D., "Creating an Infrastructure for Ubiquitous Awareness," Eighth IFIP TC 13 Conference on Human-Computer Interaction, pp. 431-438, 2001.

[17] Kuuti, K., "Activity Theory as a Potential Framework for Human-Computer Interaction Research," pp. 17-44, in [20], 1996.

[18] 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.

[19] Lövstrand, L., "Being Selectively Aware with the Khronika System," European Conference on Computer Supported Cooperative Work (ECSCW '91), pp. 265--279, Amsterdam, The Netherlands, 1991.

[20] Nardi, B., "Context and Consciousness: Activity Theory and Human-Computer Interaction." Cambridge, MA: MIT Press, 1996.

[21] 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.

[22] O'Reilly, C., Morrow, P., et al., "Improving Conflict Detection in Optimistic Concurrency Control Models," 11th International Workshop on Software Configuration Management (SCM-11), Portland, Oregon, 2003.

[23] Orlikowski, W., "Learning from Notes: Organizational Issues in Groupware Implementation," *The Information Society*, vol. 9, 1993.

[24] Podgurski, A. and Clarke, L. A., "The Implications of Program Dependencies for Software Testing, Debugging, and Maintenance," Symposium on Software Testing, Analysis, and Verification, pp. 168-178, 1989.

[25] Rothermel, G. and Harrold, M. J., "A safe, efficient regression testing selection technique," *ACM TOSEM*, vol. 6, pp. 173-210, 1997.

[26] 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.

[27] 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.

[28] Weiser, M., "Program Slicing," *IEEE Transactions on Software Engineering*, vol. 10, pp. 352-357, 1984.