"Breaking the Code", Moving between Private and Public Work in Collaborative Software Development

Cleidson R. B. de Souza[1,2] David Redmiles[1] Paul Dourish[1]

[1]School of Information and Computer Science
University of California, Irvine
Irvine, CA, USA – 92667

[2]Departamento de Informática
Universidade Federal do Pará
Belém, PA, Brazil - 66075

{cdesouza,redmiles,jpd}@ics.uci.edu

ABSTRACT

Software development is typically cooperative endeavor where a group of engineers need to work together to achieve a common, coordinated result. As a cooperative effort, it is especially difficult because of the many interdependencies amongst the artifacts created during the process. This has lead software engineers to create tools, such as configuration management tools, that isolate developers from the effects of each other's work. In so doing, these tools create a distinction between private and public aspects of work of the developer. Technical support is provided to these aspects as well as for transitions between them. However, we present empirical material collected from a software development team that suggests that the transition from private to public work needs to be more carefully handled. Indeed, the analysis of our material suggests that different formal and informal work practices are adopted by the developers to allow a delicate transition, where software developers are not largely affected by the emergent public work. Finally, we discuss how groupware tools might support this transition.

Categories and Subject Descriptors

H.4.1 [**Office Automation**]: Groupware; H.5.3 [**Group and Organization Interfaces**]: Computer-supported cooperative work;

General Terms

Human Factors

Keywords

Private work, public work, collaborative software development, qualitative studies.

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 [6].

GROUP'03, November 9–12, 2003, Sanibel Island, Florida, USA.

Indeed, they created several different tools, such as configuration management (CM) and bug tracking systems, to facilitate the coordination of groups of developers [14]. However, software development is especially difficult as a cooperative endeavor because of the several interdependencies that arise in any software development effort. To minimize these problems, current CM systems adopt design constructs (like workspaces and branches used in configuration management systems) to shield each individual from effects of other developers' work [5]. These workspaces enforce a distinction between the *private* aspects of work developed by a software engineer and the *public* aspects that occur when this developer shares his work with other developers. Similar approaches have been taken in other categories of collaborative applications (e.g., collaborative writing and hypermedia systems), which have adopted this distinction between private and public work in order to facilitate collaboration. In these applications, this is usually done through the provision of separate private and public (or shared) workspaces. Private workspaces allow users to work in different parts of a document in parallel and contain information that only one user can see and edit allowing him to create drafts that later will be shared with the other co-workers [7]. On the other hand, public workspaces allow all users to share the same information or document and edit it concurrently.

When support for private and public work is provided, it is also necessary to support *transitions* between them. The central issue in systems maintaining separate workspaces is how information or activity moves between them, and similarly, the central mechanism around which CM systems are built is the mechanism for moving information between public and private conditions – checking in, checking out, merging. In cooperative working settings, people selectively choose *when* and *how* to disclosure their private work to others, i.e., they want to be able to control the emergence of public information [1, 26]. CM tools and collaborative authoring tools provide support for these transitions. In collaborative writing, for example, one can basically copy the content of a private workspace and paste into the public workspace. On the other hand, in CM systems, more sophisticated tools involving merging algorithms and concurrency control policies need to be used because of the aforementioned interdependencies in the software.

Transitions between private and public work (and vice-versa) are particularly important in cooperative work and can lead to problematic situations when overlooked. Indeed, Sellen and Harper [28] describe case studies of companies that had problems because they underestimated the delicacy of this transition. Despite that, insufficient analytical attention has

been given to this transition by the CSCW community. In this paper, we will examine this issue with empirical material collected from a collaborative software development effort. The team observed uses mostly three tools for coordination purposes: a configuration management tool, a bug-tracking system, and e-mail. However, these tools alone were not sufficient to effectively support the team; participants needed to adopt a set of formal and informal work practices to properly support private, public work and transitions between them. The adoption of these different work practices suggests that the computational support provided by these systems to support the emergence of private information is still unsatisfactory. Based on these results, we draw more general conclusions about the implications for computer-supported cooperative work.

The rest of the paper is organized as follow. The next section discusses the idea of private and public work in computer-supported cooperative work. Then, sections 3 and 4 present the settings and the methods that we used to study the software development team. After that, Section 5 describes the set of work practices adopted by the team to properly deal with private, public work and transitions between them. Section 6 presents our discussion about the data that we collected. After that, Section 7 discusses implications of our findings in the design of CSCW tools. Finally, conclusions and ideas for future work are presented.

2. PRIVATE AND PUBLIC WORK

In this paper we examine the distinction between private and public work in collaborative efforts. The need for this distinction is widely recognized in CSCW research. According to Ackerman [1], for example, people "(…) have very nuanced behavior concerning how and with whom they wish to share information (…) people are concerned about whether to release this piece of information to that person at this time (…)". Another reason that makes people care about the release of information about them is that they "(…) are aware that making their work visible may also open them to criticism or management (…)" (*ibid.*). Furthermore, one does not make his *entire* work visible because he wants to appear competent in the eyes of colleagues and managers by making their work more complicated than necessary [26]. Indeed, people are not interested in all information that is provided to them. As Schmidt [26] points out:

"(...) in depending on the activities of others, we are 'not interested' in the enormous contingencies and infinitely faceted practices of colleagues unless they may impact our own work (...) An actor will thus routinely expect not to be exposed to the myriad detailed activities by means of which his or her colleagues deal with the contingencies they are facing in their effort to ensure that their individual contributions are seamlessly articulated with the other contributions."

To summarize, people have several contextualized and different strategies to release their *private* information, and they expect that others will do the same, not overloading them with *public* information that is not 'relevant' to their current context or activity. Note that this private information might be collaboratively constructed [16]. In this case, the information is public for those involved in its "construction", but it is private to the other members of the cooperative effort.

CSCW researchers have already recognized the need to support these findings. Indeed, a typical approach to address that is to provide support for private and public (also called shared) windows, or workspaces, to support the collaboration among users [30]. Private workspaces allow users to work in different parts of a document in parallel and contain information that only one user can see and edit, allowing him to create drafts that later will be shared with the other co-workers [7]. On the other hand, public workspaces allow all users to share the same information or document so that, changes in the document are automatically visible to all users. The usage of these workspaces mimic conventions carried over non-technological work, where no one wants to search or look at anyone's private desk or drawer, and conversely wants no one to search theirs, but accepts that when they occur in public spaces. Indeed, Mark and colleagues [21] report how conventions about the use of private and public workspaces implicitly evolved from conventions formed in face-to-face non-technological work after the introduction of a groupware tool.

Often, other mechanisms are present in collaborative systems to make other actions' visible as well. For example, grey 'clouds' were proposed in the collaborative editor Grove to indicate where other co-writers are editing the text [9]. Furthermore, it is also well-known that, in some settings, making others' work public facilitates the coordination of the activities [16] [17] and enables learning and greater efficiencies [20]. Examples of tools that explore such approaches include Portholes [8] and Babble [10].

The underlying distinction between private and public work also implies that in collaborative efforts transitions between these two aspects occur. However, while notions of "public" and "private" have been incorporated into software system design, insufficient analytical attention has been give to the transitions. Field studies such as those of Bowers [4] or Sellen and Harper [28] demonstrate that overlooking these transitions can be problematic. In Bower's study, the disclosure of private data brought about dilemmas of ownership and responsibility among the employees of the organization studied. In Sellen and Harper's study, when the companies tried to go paperless deploying a new information system, the employees' ability to control when to disclosure information was lost and these employees boycotted the system. This happened because paper, as a *medium* on which work was performed, allowed their owners to avoid sharing information with their co-workers until they felt that the information was "ready".

Note that the setting where the collaborative effort takes place is important. For example, in a control room, all workers are collocated, which allows them to use intonations in their voice and/or body language to make their actions visible to other co-workers [17]. On the other hand, Whittaker and Schwarz [34] report an ethnographic study where a large wallboard (containing the schedule of a software development project) is used by the team, which is spread along different cubicles and offices. The public location of this wallboard allowed developers to access information about who was doing which tasks at which times, among other things. In other words, in this setting, information about others' current actions was made public by checking and updating the schedule displayed in the wallboard.

In collaborative software engineering, this distinction between private and private work is not only desirable, but necessary and often enforced by tools. This occurs because of the several interdependencies that arise in any software development effort. In other words, each part of the software depends,

directly or indirectly, on many other parts. Furthermore, these interdependencies are not strictly defined in the artifacts produced, and often are not even known by the developers. To handle this problem, software engineers created tools, such as configuration management (CM) and bug tracking systems, to facilitate the coordination of groups of developers [14]. Current CM systems adopt design constructs (like workspaces and branches) to shield the work of individuals from effects of other developers' work [5]. Basically, these workspaces "create a barrier that prevents developers from knowing which other developers change which other artifacts" [25]. Therefore, CM workspaces allow software developers to work privately. Furthermore, CM systems provide mechanisms to support the transition from private to public work when developers want to make this transition. To be more specific, when a developer finishes his work in his private workspace, he can publicize his work to other software developers through check-in's, check-out's and merging operations. Despite this support, several problems arise in any software development effort. Indeed, based on empirical data that we collected, we identified a set of formal and informal work practices used by a team of software developers to handle these problems. The setting where the data was collected and the methods used to analyze this data are described in the following section.

3. THE SETTING

The team studied is located at the NASA / Ames Research Center and develops a software application we will call MVP (not the real name), which is composed of ten different tools in approximately one million lines of C and C++. 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.

Processes are also 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 deeply understand the process behavior and familiarize with its structure, therefore helping them in dealing with the complexity of the code. During the 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 in different processes. This might happen due to different circumstances. For example, before launching a new release all workforce is needed to fix bugs in the code, therefore, developers might be assigned to fix these bugs.

3.1 The Software Development Team

The software development team is divided into two groups: the verification and validation (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, three of whom are also researchers that write their own code to explore new ideas. The experience of these developers with software development range between 3 months to more than 25 years. Experience within the MVP group ranges anywhere between 2_ 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 MVP 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 in 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.

3.2 The Software Development Process

The MVP group adopts a formal software development process that prescribes the steps that need to be performed by the developers during their 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 guarantee that when he integrates his changes, he will not insert bugs in the software, or, "break the code", as informally characterized by the MVP developers. Another part of the process prescribes that, after checking-in files in the repository, a developer must send 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.

The MVP software process also prescribes the usage of 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, e.g. two processes, a developer's code will be reviewed twice: one by each manager of these two processes. On the other hand, design reviews are recommended for changes that involve major reorganizations of the source code. Their need is decided by the software manager usually during the bi-weekly software developers meeting (called pre-design meetings).

3.3 Software Development Tools: CM and Bug tracking

MVP developers employ two software development tools for *coordinating* their work: a configuration management system and a bug tracking system. Of course, other tools are used such as CASE tools, compilers, linkers, debuggers and source-code editors, but the CM and bug-tracking tools are the primary means of coordination [5] [12] [14]. These tools are integrated so that there is a link between the PR's (in the bug tracking system) and the respective changes in the source-code (in the CM tool). Both tools are provided by one of the leader vendors in the market.

A CM tool supports the management of source-code dependencies through its embedded building mechanisms that indicate which parts of the code need to be recompiled when one file is modified. To be more specific, CM tools support both compile-time dependencies, i.e., dependencies that occur when a sub-system is being compiled; and build-time dependencies that occur when several sub-systems or the entire system is being linked [12]. A bug tracking tool, when

associated with the CM tool, supports the tracking of changes performed in the source code during the development effort.

It is important to mention that the MVP team employs several advanced features of the CM tool such as triggers, techniques to reduce compilation time, labeling and branching strategies. Indeed, the branching strategy employed is one of the most important aspects of a CM tool because it affects the work of any group of software developers. This strategy is a way of deciding when and why to branch, which makes the task of coordinating parallel changes easier or more difficult [33]. According to the nomenclature proposed by Walrad and Strom [33], the following branching strategies are used by the MVP team: (1) *branch-by-purpose*, where all bug fixes, enhancements and other changes in the code are implemented on separated branches; (2) *branch-by-project*, where branches are created for some of the development projects; and (3) *branch-by-release*, where the code branches upon a decision to release a new version of the product. The branch-by-purpose strategy is employed by MVP developers in their daily work, while the other strategies are only used by the CM manager. In other words, developers create new branches for each new bug fix or enhancement, while branches for projects and releases are created by the manager only. The *branch-by-purpose* strategy supports a high degree of parallel development but at the cost of more complex and frequent integration work [33]. According to this strategy, each developer is responsible for integrating his changes into the main code. This approach is often called "push integration" [2]. After that, the changes are available to all other developers. Therefore, if one bug is introduced, other developers will notice this problem because their work will be disrupted. Indeed, we observed and collected reports of different instances of this situation. When one developer suspects that there is a problem introduced by recent changes, he will contact the author of the changes asking him or her to check the change, or for more information about it.

4. METHODS

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.

All the members of the MVP team agreed with the author's data collection. Furthermore, 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. We sampled among MVP "processes", developers' experience in software development and with MVP tools (and processes) in order to get a broader overview of the work being performed at the site. A subset of MVP group was interviewed according to their availability. We again sampled them according to the dimensions explained above. Interviews lasted between 45 and 120 minutes. To summarize, the data collected consists in a set of notes that resulted from conversations, documents and observations based on shadowing developers. These notes have been analyzed using grounded theory techniques [31].

5. PRIVATE AND PUBLIC WORK IN SOFTWARE DEVELOPMENT

As mentioned before, software development tools like configuration management systems support private, public work, and transitions between them. Despite using a CM system the MVP team faced several problems when dealing with these aspects. In this section, we present the formal and informal approaches adopted by this team in order to properly perform their work, i.e. develop software. In the sections that follow, we will explore these situations separately: private work, the transition from private to public, public work, and the transition from public to private.

5.1 Private Work

Configuration management tools allow developers to work privately through the implementation of workspaces and branches [5]. These workspaces isolate the changes being created by one developer from other parts of the code. In this case, a developer's 'work-in-progress' is not shared with other developers. Furthermore, these workspaces allow a developer to work without being affected by the changes of other developers. Indeed, when new changes are committed in the repository by other developers, the CM tool lets the user decide if he or she wants to grab these changes. In case one wants to incorporate the changes, he may recompile the software using the embedded building mechanisms on these tools. In case a developer does not want to incorporate the changes, one can continue working and, if necessary, recompile the software with the appropriate run-time options that do not grab these new changes. Of course, this is a risky course of action because it might lead the developer to work with an outdated version of the files, which might potentially make his work less ineffective.

Mechanisms embedded in CM tools are able to identify syntactic conflicts between the developer's 'work-in-progress' and the changes committed into the repository, reporting whether or not the 'work-in-progress' is affected by these changes. However, because CM systems rely on syntactic features of the domain such as files, suffixes and lines of code, they can not identify semantic conflicts [11]. This means that except for these conflicts, current configuration management systems provide extensive and automated support for maintaining the isolation between the work performed by one person from other's work [5].

However, when software developers engage in parallel development, problems arise in the CM tool. Parallel development happens when more than one developer needs to make changes in the same file. This means that the same file is checked-out by different developers and all of them are making changes in the different copies of this file in their respective workspaces. As one might imagine, parallel development might lead to conflicts. They might occur when one developer checks-in his changed version of the file back in the repository, because the versions of the other developers will become outdated. In this case, the changes of these developers might become inappropriate because they are based on a code that is not the latest. To solve this problem, a developer needs to update his version of the file by merging the other developer's

changes into his code. The developers term this operation "back merging"; in CM terminology, it is named "synchronization of workspaces" or "import of the changes". Conflicting changes are more likely to occur in files that are accessed by several developers at the same time. Indeed, in the MVP software some files are used to describe programming language structures that are used all over the code. This means that several different developers often change these files. In this case, "back merges" are problematic because CM tools face difficulties when they need to perform several merges at the same time. To overcome this problem not avoiding parallel development, MVP developers adopted a strategy to deal with these files: they perform "partial check-in's", which consist of checking-in some of the files back in the repository, even when the developers have not finished all their changes yet. This strategy reduces the number of "back merges" needed, therefore overcoming the limitations of CM tools. In addition, they minimize the likelihood of conflicting changes.

In addition to "partial check-in's", MVP developers adopt a different practice during their private work: they "speed-up" to finish some of their activities during the development process to avoid merging. This does not happen all the time though, it occurs only when MVP developers are testing their changes. This activity is performed right before the check-in operations. As one developer plainly pointed out: "This is a race!". According to the software development process, this testing is necessary to guarantee that the changes will not introduce bugs into the system. We observed that, this testing is very informal: developers will sit on the V&V laboratory and compare the current version of MVP with the one with changes. MVP developers do not use more formal techniques, such as regression testing techniques, at this moment. These will be used by the V&V staff before creating a new release of the software.

In contrast, the bug tracking tool does not provide support for the private work of software developers. All the operations made in the problem reports managed by this tool are publicly accessible to all other software developers. For example, when a developer is assigned a bug, he needs to fill some information about the bug indicating how he will proceed to fix that bug. MVP developers usually write the information to be added to the bug tracking system outside the tool in a private file only accessible by themselves. Eventually, this information is added to the bug-tracking tool by the developer, which will automatically make it available to all members of the MVP team. Furthermore, the tool does not avoid that two developers work on the same PR, as reported by one of the developers. Developers themselves have to deal with this problematic situation. The MVP group tries to avoid this problem through the software development process, which prescribes that the software manager is the one responsible for assigning PR to developers. Any assignment needs the approval of the manager. Organizational rules however interact with this process. According to these rules, the software manager can not assign work to the contractors working for the MVP group. This assignment has to be done to the manager of the contracting company, who will be responsible for assigning the work to the developers.

5.2 Moving from Private to Public Work

In this section we discuss the work practices used by the MVP team to support the transition from private to public work, as well as how the software development tools used by the MVP team support this transition. This transition might occur in two situations: when a developer asks for code reviews, or informal comments, in his code; or when a developer commits his work (source-code changes) into the CM repository.

In the first case, MVP developers want to grant others access to their code, meaning that the work will be visible to them so that they can comment on it. In this case, MVP developers simply need to change a setting in their CM workspaces. Although their work is now *public*, it is *not shared by the other developers*, meaning that it will not impact other developers work.

In the second case, after a developer commits his work into the CM repository, this work is made *public* and *shared* meaning that it is visible and might impact the work of the other developers. In order to publicize his work, the author of the changes has to perform, at least, four different operations[1]:

1. Check-in the files that he wants to publish in his own branch;
2. Check-out the same set of files from the baseline;
3. Merge his changed files with the checked-out files available in the baseline; and
4. Check-in the new files generated by the merging operation into the baseline.

From the technical point of view, these tasks are not difficult since check-in's, check-out's and merges are typical operations in CM systems and, therefore, supported by nearly every tool in the market. This means that CM systems provide adequate support for these operations. However, this support is problematic when a developer is, or was, engaged in parallel development. As mentioned in the previous section, MVP developers adopt "partial check-in's" to deal *only* with files with high levels of parallel development. Other files are not "partially checked-in". In this case, if a developer is engaged in parallel development and other developers had checked-in the same files in the baseline before him, then he will need to perform "back merges" before merging his code into the baseline. "Back merges" are supported by the CM tool through the presentation of version trees of the files being merged, which allows developers to identify the need for this task through the observation of the versions on this tree. After that, the operation is a simple merge. Again, the situation becomes problematic only if several "back merges" need to be performed.

During the transition from private to public, there is nothing that other developers need, or are able to do to facilitate this process. The work of performing the transition needs to be done by the author of the changes that will be publicized. However, because of the several inter-dependencies that exist among the several parts of the software (e.g., source-code, manuals, specifications, design documents, and so on), this does not mean that these developers will not be affected by the transition. Indeed, in order to minimize these effects, the developer who is going to perform the transition follows a set of formal and informal practices to facilitate the management of the interdependencies. These practices need to be adopted

[1] These operations might be different in other software development teams since they depend on the branching strategy adopted by the team.

because the tool support to the developers affected by the private work being publicized is minimal. These formal and informal practices are described below.

The Software Development Process

As mentioned before, the software development process adopted by the MVP team prescribes the usage of code and design reviews. One of the reasons reported by the MVP developers for using these formal reviews is the possibility of evaluating the impact that the changes under review will have on the rest of the code. The most experienced software developer of the team, for example, reported that design reviews are used to guarantee that changes in the code do not "break the architecture" of the MVP software. By breaking the architecture, she means writing code that violates some of the design decisions embedded in the MVP software. Code reviews, on the other hand, are responsibility of process leaders, who can evaluate the impact that the changes will introduce in their processes before they were committed in the main repository. This helps each and every process leader to coordinate the work of other developers working in the same process.

E-mail Conventions

In addition to formal reviews, the MVP process prescribes that *after* checking-in code in the repository, a developer needs to send an e-mail about the new changes being introduced in the system to the software developers' mailing list (see section 3.2). However, we found out that MVP developers send this e-mail *before* the check-in. Moreover, MVP developers add a brief description of the impact that their work (changes) will have on other's work in this e-mail sent to the software developers' mailing list. By adopting these practices, MVP developers allow their colleagues to prepare for and reflect about the effect of their changes. This is possible because all MVP developers are aware of some of the interdependencies in the source-code, but not all of them. As an example of this 'preparation', developers might send e-mail to the author of the changes asking him to delay their check-in, walk to the co-worker's office to ask about these changes or, if the changes have already been committed, browse the CM and bug tracking systems to understand them. The following list presents some comments sent by MVP developers:

> *"No one should notice."*
> *"*[description of the change]*: only EDP 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 and running a MVP tool with live* [z] *data you will need to run your own* [z] *daemon."*

These e-mails are also important because they tell (or remind) developers that they have been engaged in parallel development. Often, developers do not know that this is happening[2]. The information in the e-mail is usually enough to tell the developer if he needs to incorporate these changes right away in order to continue his work, or if he can wait until he is ready for check-in. In both cases, the developer needs to "merge back" the latest changes into his version of the file.

Sending e-mail before a check-in is also used by other developers to support expertise identification, and as a learning mechanism. Developers associate the author of the e-mails describing the changes with the "process" where the changes are being performed. In other words, MVP developers assume that if one developer constantly and repeatedly performs check-ins in a specific process, it is very likely that he is an expert on that process. Therefore, if another developer needs help with that process he will look for him for help:

> *"*[talking about a bug in a process that he is not expert] *(...) I don't understand why this behaves the way it does. But, most of these PR's seem to have John's name on it. So you go around to see John. So, by just by reading the headline of who does what, you kind of get the feeling of who's working on what (...).So they* [e-mails] *tend to be helpful in that aspect as well. If you've been around for ten years, you don't care, you already know that* [who works with what]*, but if you've been here for two years that stuff can really make difference (...)"*

On the other hand, the fact that developers read e-mails sent by other developers to assess the impact of others' changes in their code contributes to their learning experience within MVP. Note that developers who reported the aspects described in this section had little experience working at MVP: the first with 2 years and the second with 2 _ months.

Problem Reports

The problem reports (PRs) of the bug-tracking tool are used by different members of the MVP team who play diverse roles in the software development process. Basically, when a bug is identified, it is associated with a specific PR. The tester who identified the problem is also responsible for filling in the PR the information about 'how to repeat' it. This description is then used by the developer assigned to fix the bug to learn and repeat the circumstances (adaptation data, tools and their parameters) under which the bug appears. In other words, the information provided by the tester is then used by the MVP developer to locate, and eventually fix the bug. 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 changes 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 other words, PR's provide information that is useful for different members of the MVP team according to the roles they

[2] Differently than the developers reported by Grinter [14], before checking-out a file, they do not check the version tree that displays information about other developers working on the same file.

are playing. They facilitate the management of interdependencies because they provide information to MVP developers that help them in understanding how their work is going to be impacted by the changes that are going to be checked-in the repository.

Holding check-in's

As mentioned earlier, MVP developers add a brief description of the impact of their changes to the e-mail sent to the developers before checking-in any code. Two types of impact statements are used more often than others: changes in run-time parameters of a process, and the need to recompile parts or the whole source code. The former case is important because other developers might be running the process that will be changed with the check-in. The latter case is used because when a file is modified, it will be recompiled, as well as, the other files that depend on it and this recompilation process is time-consuming, up to 30 to 45 minutes. Developers are aware of the delay that they might cause to others. Therefore, they hold check-in's until the evening to minimize the disturbance that they will cause. According to one of the developers:

> *"(...) people also know that if they are going to check-in a file, they will do in the late 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."*

The transition from private work, then, is recognized as a point at which the work of a single developer can impact the work of others. Developers' orientation is not simply towards the artifacts but towards the work of the group. The subtlety with which the transition is managed reflects this consideration.

5.3 Public Work

The work of one developer becomes public when it is visible to all other co-workers. This happens in two different circumstances: when a developer changes the settings of his workspaces to grant others access to his code and after a developer commits his changes into the repository of the CM tool. These situations raise the question of how the MVP developers handle the new public work (changes)?

In the former case, the work is public but not shared, which means that it is not going to affect other developers' work. Therefore, MVP developers do not need to take any step in order to handle the public work, because it will not affect them. However, in the second case, MVP developers might need to adapt their work based on these changes. Indeed, MVP developers might need to recompile their changes (work) in case they choose to incorporate the new public work or they might need to change the run-time parameters of a process that was altered by the changes. Based on our data, we found out that the configuration management tool provides *some* help to MVP developers handle this situation. As mentioned before, these tools have building mechanisms that help MVP developers, upon request, to incorporate the new changes and identify syntactic conflicts between the developer's 'work-in-progress' and the new changes. However, these tools are not able to detect semantic conflicts since they are purposely created to be independent of programming languages [11].

The bug tracking tool, on the other hand, provides support for public work because all the operations performed in the problem reports are automatically visible to all MVP developers. In addition, this tool implements some accounting features that record the history of a PR including all operations performed on each one of them.

5.4 Moving from Public to Private Work, or "Breaking the code"

According to Walrad and Strom [33], the branch-by-purpose strategy adopted by the MVP team (see section 3.3) assures continual integration of the code, therefore minimizing problems. However, this strategy needs to be complemented by some form of notification that informs all developers that a check-in happened (and therefore that some integration took place). As mentioned before, this is achieved in the MVP team through the e-mail notification sent before the check-in's. Therefore, whenever a new change is introduced in the repository, all developers are notified about it. This affords an easy detection of problems caused by the introduced changes. In other words, if a change introduces a bug in the software, other developers might be able to detect it because: (i) they are aware that a change was introduced in the code by another developer; and (ii) they usually integrate the new introduced changes in their own work. If any abnormal behavior is identified in the software after a check-in, whoever identified that will contact the author of the check-in to verify if the problem is happening because of the check-in. If that is the case, the software is called "broken" and the code that was checked-in must be removed from the repository, corrected, and checked-in again later. In other words, the publicly available work needs to be made private again. The CM tool supports this transition because it provides rollback facilities that allow one to remove committed changes from the repository.

6. DISCUSSION

The notions of private and public work and workspaces are well known ones in the design of collaborative systems. However, our empirical observations draw attention to the complex set of practices that surround the *transition* between public and private. Private information has public consequences, and vice versa.

The different formal and informal work practices arise in the MVP team, especially, because of the interdependencies among the different artifacts created during the software development process. Indeed, these interdependencies make the process of publicizing work so important. A developer can not simply carelessly publicize his work, because this will cause a large impact in other developers' work: some of them will need to go through their testing again, others will spend a lot of time recompiling their changes, others can need to change their own code in order to adapt the new checked-in code, and so on.

Since the MVP developers are aware of some of these interdependencies, they explicitly work to minimize problems that emerge in the relationship between their different working needs. Artifacts such as problem reports facilitate the management of interdependencies of developers from the different groups and with different roles. Problem reports are "boundary objects" in the sense of Star and Griesemer [29]; objects whose common identity is robust enough to support coordination, but whose internal structure, meaning, and consequences emerge from local negotiations between groups.

Viewing PR's as boundary objects draws attention to their role in managing loosely-coupled coordination, and how each developer is able to interpret the information in the PR's that is useful to their current work. Critically, this is achieved without changing the identity of each PR along the whole software development process. Indeed, each PR keeps the same unique identifier.

Interestingly, these formal and informal work practices require that the author of the changes performs *most* of the additional work. However, this author will not get any benefit from that. Indeed, sending e-mail notifications, holding check-in's, and filling the appropriate PR's fields during the implementation are all operations performed by the author of the changes and none of them facilitate or improve his work. There is one developer performing the extra-work who does not gain any benefit of this extra work, and fifteen other developers who benefit from his work[3]. That is exactly one of the situations that lead groupware applications to fail [15]. In this particular software development team though, this does not happen. MVP developers are aware of the extra-work that they need to perform, but they are also aware that this same extra-work is going to be performed by the other developers when necessary, and this is going to help each and every one of them in performing their tasks.

On the other hand, MVP developers also adopt informal practices during their private work. The first one, called "partial check-in's", is especially important because it is used to handle files with a high degree of parallel development and changes in these files positively correlate with the number of defects [23]. "Partial check-in's" are variations of the formal software development process, which establishes that check-ins only will be performed when the entire work is done. They are necessary because of the software development tools adopted are unable to properly handle merging in these files. This is the same reason, according to Grinter [14], that led other team of software developers to either avoid parallel development or rush to finish their work. On the other hand, MVP developers rush because they do not want to repeat their testing when another developer checks-in some code into the repository. In both studies, developers describe their dilemma: they want to produce high-quality code, but they also want to finish fast their changes.

Holding onto check-in's is another informal approach adopted by the MVP developers during their private work. It is adopted because they are aware of some of the existing interdependencies in the software and they want to minimize the impact that their changes will cause on others' work. To be more specific, they understand that some changes cause a lot of recompilation, which might lead other developers to spend time "watching" the recompilation.

All this extra-work performed by the different members of the MVP team is another form of articulation work [27] that occurs in cooperative software development. It is different from the recomposition work [13], which is the coordination required to assemble software development artifacts from their parts. Recomposition work focuses on choosing the right components to create a software artifact due to source-code dependencies, while this extra work that we report focuses on the management of all interdependencies that exist in a software development effort.

After any code is checked-in into the CM repository, the other MVP developers are able to detect problems, or, detect if the MVP software is "broken". As noted in other settings such as ship bridges [19] or aircraft cockpits [20], this can be achieved because work artifacts and activities are visible to all. By creating a public space, the CM repository supports collective error detection and correction.

7. IMPLICATIONS FOR TOOLS

Software engineers have been developing tools to help co-workers in analyzing the impact of others' work in their own work. In this case, the support is provided to the developers after the transition from private to public work has been made. This approach, called change impact analysis [3], uses several techniques. One example is dependency graph approaches, which focus on determining the impact of the changed 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 representing aspects of the program's control flow and data flow [24]. In other words, they focus only in determining the impact of the changes in the product in the rest of the cooperative effort. Although powerful, these techniques are also computationally expensive and very time-consuming to be used by developers in their daily work. Consequently, they do not completely support the transition from private to public work, and as we've seen, this is a very subtle step in cooperative software development. Although these techniques have their limitations, they are evidence that the dependencies between developers' working activities are a cause for concern and attention. We argue that other cooperative efforts, especially those with several interdependencies, could greatly benefit from such approaches, if they were arranged to support the emergence of public information.

Recent approaches in software engineering attempt to provide useful information to developers so that they can better coordinate. In other words, these approaches try to increase the awareness [7] of software engineers about the work of their colleagues. They differ, however, on the type of information that is provided. A first approach is based on the idea of facilitating the dissemination of public information by collocating software developers in warrooms [32]. In this case, companies expect to achieve the same advantages that the public availability of others' actions has brought to other settings such as ship bridges [19], aircraft cockpits [20], transportation control rooms [17] and city dealing rooms [16]. Indeed, early results of this approach have been encouraging [32]. However, there are practical limitations in the size of the teams that can be collocated, which suggests that tool support is still necessary. Indeed, new tools like Palantir [25] and Night Watch [22] adopt a different approach that focuses on constantly publicizing information(like CM commands) collected from a CM workspace to other workspaces that are accessing the same files. In this case, instead of focusing in the transition between private and public aspects of work, these tools basically eliminate the private work by making all aspects of the work publicly available to others. However, as discussed in section 2, the need for privacy and for controlling the release of private information is an important aspect in any social setting; which therefore needs to be addressed in the design of cooperative tools.

[3] The MVP group is composed of 16 developers. One of them is performing the check-in; therefore 15 others are being helped by the extra-work.

Finally, our data suggests that a software developer might use different sources of information at different times in order to assess the current status of the work. As mentioned before, the MVP team uses information from e-mail messages, the configuration management tool and the bug tracking system. By reading e-mail, MVP developers are aware of future changes in the CM tool because somebody else is going to check-in something. By inspecting only the CM tool, a developer can be aware of partial check-ins in the repository that are not reported by e-mail. And finally, the bug-tracking tool, through its PR's, provides information about how a developer's work is going to be impacted by the problem report associated with the check-in. These are three different tools that a MVP developer has to use. We believe that a possible improvement is to use event mechanisms (such as event-notification servers) to integrate these different sources of information, and then provide a unique interface and tool to assess the relevant information. Furthermore, abstraction techniques [18] could be employed to generate high-level information (e.g., status of the work) from low-level information like recent check-ins and check-outs, e-mails exchanged among software developers, information added to the bug-tracking tool, etc. This is an interesting research area that we plan to explore.

8. CONCLUSIONS AND FUTURE WORK

This paper examined the transitions between private and public work based on empirical material collected from a large-scale software development effort. The team studied, called MVP, uses mostly three tools to coordinate their work: a configuration management (CM) tool, a bug-tracking system, and e-mail. These tools provide support for private and public work, as well as some technical support that facilitates the transition from the former aspect to the latter. However, MVP developers also adopted a set of formal and informal work practices to manage this transition. These transitions are necessary to facilitate the management of the interdependencies among the different software artifacts. The following practices were identified and described in the paper: partial check-in's, holding onto check-in's, problems reports crossing team boundaries, code and design reviews, "speeding-up" the process, and finally, the convention of adding the description of the impact of the changes in the e-mail sent to the group. These practices suggest that analytical attention needs to be given to these transitions in order to enhance our understanding of cooperative work. Furthermore, computational support also needs to be provided so that this task can occur properly.

We plan to study other software development teams in order to understand how they deal with the aforementioned transition and their work practices to perform that. By doing that, we expect to learn important characteristics that can help us in understand other cooperative efforts.

9. ACKNOWLEDGMENTS

The authors thank CAPES (grant BEX 1312/99-5) and NASA/Ames for the financial support. Effort 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, 9624846, IIS-0133749 and IIS-0205724. 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.

10. REFERENCES

[1] Ackerman, M. S., "The Intellectual Challenge of CSCW: The Gap Between Social Requirements and Technical Feasibility," *Human-Computer Interaction*, vol. 15, pp. 179-204, 2000.

[2] Appleton, B., Berczuk, S., et al., "Streamed Lines: Branching Patterns for Parallel Software Development," vol. 2002, 1998.

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

[4] Bowers, J., "The Work to Make the Network Work: Studying CSCW in Action," Conference on Computer-Supported Cooperative Work, pp. 287-298, Chapel Hill, NC, USA, 1994.

[5] Conradi, R. and Westfechtel, B., "Version Models for Software Configuration Management," *ACM Computing Surveys*, vol. 30, pp. 232-282, 1998.

[6] 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.

[7] Dourish, P. and Bellotti, V., "Awareness and Coordination in Shared Workspaces," Conference on Computer-Supported Cooperative Work (CSCW '92), pp. 107-14, Toronto, Ontario, Canada, 1992.

[8] 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.

[9] Ellis, C. A., Gibbs, S. J., et al., "Groupware: Some issues and experiences," *Communications of the ACM*, vol. 34, pp. 38-58, 1991.

[10] Erickson, T. and Kellogg, W. A., "Social Translucence: An Approach to Designing Systems that Support Social Processes," *Transactions on HCI*, vol. 7, pp. 59-83, 2000.

[11] Estublier, J., "Software Configuration Management: A Roadmap," Future of Software Engineering, pp. 279-289, Limerick, Ireland, 2001.

[12] Grinter, R., "Supporting Articulation Work Using Configuration Management Systems," *Computer Supported Cooperative Work*, vol. 5, pp. 447-465, 1996.

[13] Grinter, R. E., "Recomposition: Putting It All Back Together Again," Conference on Computer Supported Cooperative Work (CSCW'98), pp. 393-402, Seattle, WA, USA, 1998.

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

[15] Grudin, J., "Why CSCW applications fail: Problems in the design and evaluation of organizational interfaces," ACM Conference on Computer-Supported Cooperative Work, pp. 85-93, Portland, Oregon, United States, 1988.

[16] Heath, C., Jirotka, M., et al., "Unpacking Collaboration: the Interactional Organisation of Trading in a City Dealing Room," Third European Conference on Computer-Supported Cooperative Work, pp. 155-170, Milan, Italy, 1993.
[17] 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.
[18] Hilbert, D. and Redmiles, D., "An Approach to Large-scale Collection of Application Usage Data over the Internet," 20th International Conference on Software Engineering (ICSE '98), pp. 136-45, Kyoto, Japan, 1998.
[19] Hutchins, E., *Cognition in the Wild*. Cambridge, MA: The MIT Press, 1995.
[20] Hutchins, E., "How a Cockpit Remembers its Speeds," *Cognitive Science*, vol. 19, pp. 265-288, 1995.
[21] Mark, G., Fuchs, L., et al., "Supporting Groupware Conventions through Contextual Awareness," European Conference on Computer-Supported Cooperative Work (ECSCW '97), pp. 253-268, Lancaster, England, 1997.
[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 (to appear).
[23] Perry, D. E., and, H. P. S., et al., "Parallel Changes in Large-Scale Software Development: An Observational Case Study," *ACM Transactions on Software Engineering and Methodology*, vol. 10, pp. 308-337, 2001.
[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] 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.
[26] Schmidt, K., "The critical role of workplace studies in CSCW," in *Workplace Studies : Recovering Work Practice and Informing System Design*, P. Luff, J. Hindmarsh, and C. Heath, Eds.: Cambridge University Press, 2000, pp. 141-149.
[27] Schmidt, K. and Bannon, L., "Taking CSCW Seriously: Supporting Articulation Work," *Journal of Computer Supported Cooperative Work*, vol. 1, pp. 7-40, 1992.
[28] Sellen, A. J. and Harper, R. H. R., *The Myth of the Paperless Office*. Cambridge, Massachusetts: The Mit Press, 2002.
[29] Star, S. L. and Griesemer, J. R., "Institutional Ecology, Translations and Boundary Objects: Amateurs and Professionals in Berkeley's Museum of Vertebrate Zoology.," *Social Studies of Science*, vol. 19, pp. 387-420, 1989.
[30] Stefik, M., Foster, G., et al., "Beyond the Chalkboard: Computer Support for Collaboration and Problem Solving in Meetings," *Communications of the ACM*, vol. 30, pp. 32-47, 1987.
[31] Strauss, A. and Corbin, J., *Basics of Qualitative Research: Techniques and Procedures for Developing Grounded Theory*, Second. ed. Thousand Oaks: SAGE publications, 1998.
[32] Teasley, S., Covi, L., et al., "How Does Radical Collocation Help a Team Succeed?," Conference on Computer Supported Cooperative Work, pp. 339-346, Philadelphia, PA, USA, 2000.
[33] Walrad, C. and Strom, D., "The Importance of Branching Models in SCM," *IEEE Computer*, vol. 35, pp. 31-38, 2002.
[34] Whittaker, S. and Schwarz, H., "Meetings of the Board: The Impact of Scheduling Medium on Long Term Group Coordination in Software Development," *Computer Supported Cooperative Work*, vol. 8, pp. 175-205, 1999.