“Breaking the Code”, Private and Public Work in Collaborative Software Development

Cleidson R. B. de Souza[1,2] and David F. Redmiles[2]
[1]Universidade Federal do Pará, Brazil and [2]University of California, Irvine, USA
cdesouza@ics.uci.edu, redmiles@ics.uci.edu

Abstract. As a cooperative effort, software development is especially difficult because of the many interdependencies amongst the artifacts created during this activity. In order to minimize problems created by these interdependencies, some software development 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, our analysis 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.

Private and Public Work in CSCW

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 (Curtis, Krasner et al. 1988). Indeed, they created different tools, such as configuration management (CM) and bug tracking systems, to facilitate the coordination of groups of developers (Grinter 1995). 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, CM systems adopt design constructs (like branches and workspaces used in configuration management systems) to shield each individual from effects of other developers’ work. These workspaces enforce a distinction between the *private* aspects of work developed by the software engineer and the *public* aspects that occurs when this developer shares his work with the 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. 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. On the other hand, public workspaces allow all users to share the same information or document.

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 (Ackerman 2000). 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 (Sellen and Harper 2002) describe some 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 used three software development tools for coordination purposes. 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.

Setting and Methods

The group studied develops an application called MVP (not the real name) and is divided in two teams: developers and the verification and validation staff (V&V). Developers are responsible for writing new code, for performing bug fixing, enhancements, and so on. There are 25 developers, including researchers that write their own code. The V&V team (6 engineers) is responsible for testing the software, keeping a running version for demonstration and maintaining user manuals.

The first author spent eight weeks during the summer of 2002 as a member of the MVP team. During that time, he was able to interview developers, 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), problem reports (PR's), and so on.

MVP Practices to Handle Private and Public Work

The main tools used by the MVP team to coordinate their activities are the configuration management (CM) and the bug tracking tools (Grinter 1995). Branching in CM tools are used to create shields between developers' workspaces isolating one's work from others (Conradi and Westfechtel 1998). On the other hand, merging mechanisms are created to allow one's work to be combined with other developers' work. In other words, branches support private work, while merging mechanisms support the transition from private to public work. Finally, building mechanisms in CM tools support the public work because they allow developers to automatically recompile the code in order to incorporate changes recently committed in the repository.

In general, we identified that the private and public work are properly supported by the software development tools and by the software development process adopted by the MVP. However, except for

the merging mechanisms embedded in CM tools, the *transitions* between private and public are improved through informal work practices because of the need developers have to manage the interdependencies. Examples of these practices will be briefly discussed in the following paragraphs.

We called the first practice “holding onto check-in’s”. Developers will hold onto check-in’s (and merges) when they realize that their work (in this case, their changes in the software) will imply in the recompilation of the whole source code. They avoid that because they know that the recompilation process is time-consuming usually taking between 30 to 45 minutes. This means that other developers will waste their time waiting for the recompilation of their local copies.

After making their work public by merging it back into the repository, the software development process prescribes that MVP developers must send an e-mail to the whole software development group informing about the new changes in the system. However, these developers will send this e-mail before committing their changes and will also add a brief description of the impact that these changes will cause on their colleague’s work. In this case, because of these e-mail messages, other developers might reflect about the effect of their colleagues’ changes in their current work and prepare for that. This is possible because they are aware of some interdependencies in the source-code. The convention (adding impact statements in the e-mails) is the second practice identified.

A third approach identified was the “partial check-in”, which consists of checking-in files back in the repository, even when the developers have not yet finished their entire work. This is used to deal with parallel development in files that are changed very often. This practice allows developers to reduce the work necessary to make their work public, as it minimizes the number of updates that they need to perform in their files before merging them into the main repository.

Finally, we also identified that problem reports (PR’s) are used by different stakeholders (e.g., end-users liaisons, developers and testers) to manage the software interdependencies. For example, when a bug is identified, it is associated with a specific PR. Whoever identified the problem is also responsible for filling in the PR with information about ‘how to repeat’ it. This description is used by the developer assigned to fix the bug to specify the circumstances (adaptation data, tools and their parameters) under which the bug appears. In short, MVP members use the information from the PR’s in many different ways, according to the role they are playing.

Conclusions

We briefly described some of the work practices adopted by software developers to properly handle the transition between their private and their public work. MVP developers employ these practices because of the interdependencies that exist in the software. As mentioned before, the adoption of these practices suggests that computational support is necessary in cooperative software development tools to support the emergence of private information.

References

Ackerman, M. S. (2000). "The Intellectual Challenge of CSCW: The Gap Between Social Requirements and Technical Feasibility." Human-Computer Interaction **15**(2-3): 179-204.

Conradi, R. and Westfechtel, B. (1998). "Version Models for Software Configuration Management." ACM Computing Surveys **30**(2): 232-282.

Curtis, B., Krasner, H. and Iscoe, N. (1988). "A field study of the software design process for large systems." Communications of the ACM **31**(11): 1268-1287.

Grinter, R. E. (1995). Using a Configuration Management Tool to Coordinate Software Development. Conference on Organizational Computing Systems, Milpitas, CA, 168-177.

Sellen, A. J. and Harper, R. H. R. (2002). The Myth of the Paperless Office. Cambridge, Massachusetts, The Mit Press.