An Observation of Fine-Grain Usage Patterns for Two Configuration Management Tools

Scott Banachowski **Jim Whitehead**
University of California, Santa Cruz
Computer Science Department
1156 High Street
Santa Cruz, CA 95064
{sbanacho, ejw}@soe.ucsc.edu

ABSTRACT
This paper presents the results of a survey of employees from one company that uses different configuration management tools and processes for the development of two separate products. Results from this study suggest that workspace semantics, and build semantics both have a significant impact on the fine grain use pattern of the check-in operation. In contrast, the meaning and use of check-out was the same across configuration management systems. Other results highlight the difficulty of overcoming lock-in to a given configuration management tool, and the desirability of a hybrid change approval process that finds a middle ground between the lightweight and batch processes.

Keywords
Software configuration management, version control, software engineering

1 INTRODUCTION

Today a wide selection of commercially available configuration management systems exist; one web site catalogs over 50 such packages [4]. The technology of these systems is well understood [1, 3, 6, 7, 8, 10], but the actual use patterns and motivations for fine-grain operations such as check-in are not. For example, we expect a check-in when the modification of an item is complete—but motivations for check-ins due to incremental changes are less predictable. We have observed differences in frequency and rationale for fine-grain operations, and seek to understand why. One study by Grinter [5] identifies the interplay between the tool and the organization, and finds that tools provide coordination mechanisms for developers. This suggests that use patterns may be influenced by an organization's adapted processes regardless of the understood capabilities of the configuration management technology.

A configuration management system requires three basic capabilities: *version control*, *check-in/check-out* facilities, and *history comparison* tools [3]. Our study focuses on these fine-grain operations, to determine the criteria used to initiate operations such as creating a new revision. Our paper presents a survey taken at a company that develops two products, but uses different configuration management systems for each product. The participants of the survey were all software developers, as the goal is to discern the users' perspectives and not an experts' knowledge. The study focuses on qualitative aspects of the users' experiences with the CMS tools and processes. Initially, differences in the processes were not well-enough understood to establish useful quantitative measurements for comparison; the motivation for the survey is to uncover unexpected behavior patterns due to differences in tools or processes.

The following sections describe the processes adapted by the teams and the users' perceptions of the tools and processes. The conclusion summarizes the observed trends and offers some future directions for further validating our findings and extending this study.

2 STUDY BACKGROUND

The Company

FooMediCo[1] manufactures medical equipment, including hardware and its associated system software. The company has 1500 employees worldwide, 90 of which are members of the software development organization. There are two products currently on the market that have similar function, but differ in their targeted market segments. The products are based on different system architectures, and therefore do not share any common software libraries. Product *Alpha*'s development began in 1991, and the team of 35 is currently implementing software release 7.0, which is roughly one million lines of code. Product *Omega*'s development began in 1989, and the team of 45 is currently developing release 6.0, which is roughly seven million lines of code.[2]

Although both FooMediCo products are mature, enhancements to software provide significant new capabilities for the equipment, so software evolution continues. This work includes integrating new features into existing code and fixing bugs. Research and marketing drive the enhancement tasks, based on the desirability and feasibility of new features. An independent test group and release manager decide priorities for bug-fix tasks. In this study, we do not distinguish between adding features and fixing bugs; we refer to either effort simply as a *task*.

Survey Methodology

Although development of both products began near the same

[1] actual name withheld for anonymity
[2] including comments, figures based on employee estimates

time, the configuration management software and processes differ. To determine the impact of the tools and processes on fine-grain usage patterns, employees participated in a survey. The survey was conducted through phone interviews with 10 employees of FooMediCo's software engineering organization. All are system software developers, responsible for designing and writing the source code for the embedded processors of the products. One worked exclusively on Alpha, 5 on Omega, and 4 on both. The developers' experience at FooMediCo ranged from 1-10 years, with positions including junior and senior programmers, and low-level managers (who typically split their time between development and managing small teams).

The questions were open-ended,[3] encouraging interviewees to describe usage of the tools and processes in their own words—discussions often deviated from the questions providing valuable insight into the organization. When describing the CMS, interviewees were reminded not to provide abstractions, but specific examples [2]. Answers were transcribed during the interview, and later compiled. The results do not provide data that is easy to quantify; with this study we discover general trends in usage patterns. The open-ended questions allow us to learn about unexpected behavior, without preconceived notions of CMS processes. The remainder of this paper presents a summary and analysis of the results.

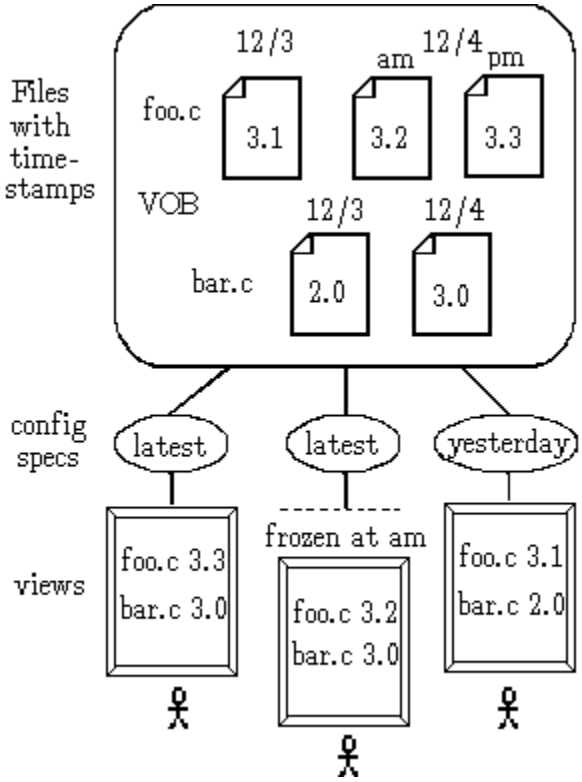

Figure 1: Alpha model using ClearCase

3 SURVEY RESULTS

Tools and Processes

The following discussion is gathered from descriptions of the employees at FooMediCo, and reflects the terminology used by them. Since these descriptions are derived from the perceptions of the user, they may not necessarily reflect the terminology or usage intended by the tools' manufacturers or applied by users at other companies. The survey deliberately included only product developers, and not members of the organization that administer the tools (a team known as the *build group*; the group has other responsibilities as well). A member of the build group reviewed a previous draft of this paper to verify accuracy of the process descriptions. All configuration items include only source code files, as the developers at FooMediCo do not maintain requirement or design documentation with the CMS.

Product Alpha

The Alpha development team uses ClearCase for revision control. ClearCase is a product of the Rational Software Corporation [9]. In ClearCase, all versions of configuration items are stored in databases. ClearCase is feature rich product, and requires site customization. Due to the complexity of the product, most sites using ClearCase dedicate resources for administering the tool and the database servers. At FooMediCo, this responsibility lies with the build group.

The database is accessible through a *vob* (version object base); to the user it looks and behaves like a file system, but it provides work isolation between collaborators. The set of items visible to the user is called a *view*. The view is determined by a *config spec*, a set of rules for selecting versions to present the user. Developers working on different parts of the same feature, or who must fix bugs in the latest release, use identical config specs to see the same view of the system. At FooMediCo specialists in the build group create templates for config specs, and users execute scripts to make individual copies of these templates. For example, the most commonly used config spec sets a rule that all latest versions are visible. Users with this config spec see the same view so that when one checks in a change the new version is immediately visible to the others.

Custom scripts at FooMediCo allow users to freeze views, meaning others' check-ins will not be visible. This is equivalent to creating a branch of the baseline, and is helpful because the developer may change and test items in isolation from the rest of the system. When un-freezing the view, all changes due to others' check-ins become visible, and the developer must resolve conflicts before checking-in their own changes. This step completes the merging of views, and changes become visible to other collaborators.

For most projects, a developer views the latest version of a baseline. Check-ins that conflict with or break existing software must be resolved immediately, since a majority of de-

[3] see appendix A

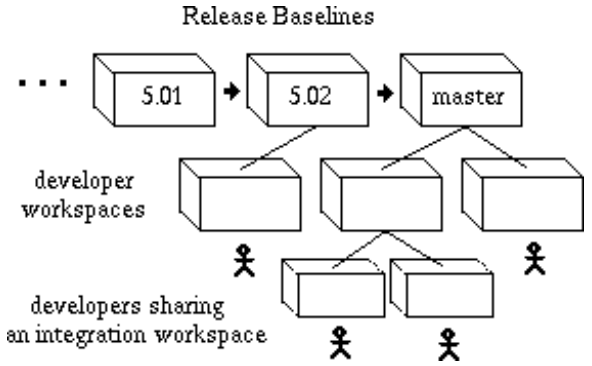

Figure 2: Omega model using CodeMgrTool

velopers require the latest code to be functional. When a check-in prevents code from working, developers revert their view to "yesterday's" release until the problem is resolved. A product release manager outside of the developer's group monitors all check-ins to the baseline, and backs out changes that are unacceptable.

Product Omega
The Omega development team uses CodeMgrTool. CodeMgrTool is a graphical interface wrapper for SCCS [10] developed by Sun Microsystems, which adds the capability for workspace and directory management [11][4]. CodeMgrTool uses the concept of a workspace, which is a directory (or a hierarchy of directories) under SCCS version control. Although SCCS does not provide work isolation, workspaces do; each developer may create their own workspace, or share workspaces among collaborators.

There is a hierarchical relationship between workspaces. The easiest way to visualize this relationship is with a tree structure. At the root of the tree is a workspace containing SCCS managed versions of every configuration item. This root may have multiple children nodes, each containing a copy of a subset of the items. At its creation, a child node contains no files. The developer chooses to *bringover* items from the parent, creating a new copy under SCCS control within the child workspace. Subsequent bringovers of items already existing in the child workspace result in a merge of the parent item with the child's, prompting the user to manually resolve conflicts.

Within a child workspace, an item may be checked-out or checked-in. A parent workspace is unaffected by changes in its children. Changes in a child propagate up the tree through a *putback*. A putback merges changes with the parent's item, creating a new version in the parent workspace.

Every Omega release is contained in its own workspace. Some releases are intended for external distribution to customers, but a majority are internal releases incorporating project milestones or bug-fixes. Rather than storing items from all releases in the same single database, each Omega release is a duplicate of the complete collection of configuration items; any release may act as the root of a workspace tree, and there is no mechanism for sharing a single copy of an item among two releases. Releases are stored in a repository on an NFS file system, and are maintained by the build group. No developer has permission to edit items in a release, meaning no developer has permission to putback from their child into a release workspace. If a developer must change an item from a release, they create a workspace and bringover the item for editing. Within that workspace, they check out the item, edit it, and check it in. Instead of a putback, a request to change the item is submitted through a web-based form called a *girf* (graphical integration request form).[5] Upon submission of the form, the girf scripts run a putback; instead of placing new versions in the parent as with an ordinary putback, the resultant versions are stored in an intermediate workspace.

Approximately once a week, the build group takes changes from the intermediate workspace, and with the approval of the release manager, merges those changes into the root. At this time, if two developers changed the same item, the merge results in a conflict, and the build group notifies the developers who submitted them. The process is held and delayed until a developer resolves the conflict. The name of the latest release is *master*. The process of creating a new release by incorporating the accumulated changes into the latest release is known as *toggling master*. After master toggles this new release becomes the master, and the old release is renamed with a string code containing a release number.

Fine Grain Operations
With an understanding of each product's software processes, we now present the results of our survey on fine-grain configuration management tool usage, and explain the processes' roles on the observed differences.

Check-out
Neither product team expressed different views of the check-out process. A file is checked-out whenever it must be modified to complete a task. When beginning a task, developers do not decide which set of files they need beforehand; in their workspace or view, they begin checking out files as the necessity to edit them arises. The process of checking-out files is complete when no more need for changes is encountered.

Check-in
Alpha engineers check-in files incrementally until they complete a task.[6] For most tasks, the selected view represents the up-to-date state of the software. The developer has the ability to test the interaction of their changes with the current system as soon as they un-freeze their view (often they do not need to freeze a view at all). Therefore a change is completely

[4]Sun packages this product in a suite called TeamWare.

[5]The web form is a combination of a HTML interface and scripts, and is maintained on the company's internal web site.

[6]For less complicated tasks a single check-in may suffice.

tested before check-in. If a check-in "breaks" the system, all other team members immediately suffer the consequence; developers quickly learn that to avoid the embarrassment of reprimand by their peers they must test thoroughly before check-in, and the process is highly self-regulating. Due to the risk of check-ins, developers often check-in changes frequently along the path to completing a task; it is easier to test and manage small incremental changes than large ones.

Due to the hierarchical nature of CodeMgrTool, Omega engineers have different rationale for check-ins on tasks of small scope. The developers work in independent workspaces spawned as children of master, so until a girf is processed check-ins are invisible to others. A girf will not be accepted until the task is complete, so there is little incentive for incremental check-in. Check-in may wait until the task is complete, tested, and ready to girf, with some exceptions. Some developers incrementally check-in to preserve changes worth keeping. The other common check-in occurs when master toggles: to keep the workspace up-to-date with the new master they must initiate a bringover. The tool requires a check-in of any file that changed in master before the bringover completes.

Omega engineers collaborating on larger tasks typically share a child of master as an integration workspace, and create grandchildren workspaces for independent work. A putback from the grandchild propagates changes to the child workspace, making them visible to the collaborators. Before a putback, all files must be checked-in. Therefore check-ins occur when the developer is ready to putback, either because they completed a task or need to incrementally share changes with collaborators. In this collaborative effort, the rationale resembles Alpha's, as the team must maintain the integrity of a shared workspace themselves.

Version Comparisons

Surveyed developers express a wide variety of views about version comparison, but these views are not product dependent. Some indicate that they rarely-to-never look at previous versions of code (with the exception of resolving merge conflicts). Only 30% refer to past versions more often, citing reasons such as "software archeology," regression testing, and bug-hunting. Software archeology is a term to describe the process of reconstructing the history of the system as an aid for understanding its evolution or unraveling mysterious implementations. Regression testing ensures that a new version meets the same standards as a previous version. Past versions aid bug-hunting when a latent bug introduced in earlier versions later appears.

Tool Evaluation

90% of survey participants expressed a preference for ClearCase over CodeMgrTool. Even those who never used ClearCase desired to switch based on its reputation. Cited advantages of ClearCase include technological superiority, shorter build times, better graphical interface, expanded auditing, easier access to previous versions, safer prevention of misuse, improved abstraction, and more efficient use of storage. The drawbacks of ClearCase are a higher learning curve and more administration overhead.

Cited advantages of CodeMgrTool include ease of use and intuitive interface. Most developers consider CodeMgrTool less sophisticated. The most significant disadvantage is build time. Although developers only operate directly on user-created items in the CMS, we find that the tools' maintenance of derived objects (such as compiler outputs) is an important factor in performance. The build system for Omega uses a make tool that resolves links to items absent in a workspace by locating the parent's version of the item. The resolution of this inheritance is limited to whole libraries, so a minor change may require the bringover and compilation of an entire colossal library, significantly increasing both workspace maintenance overhead and build time. The ClearCase build system shares object files in a view, so a single compilation updates the object files for all sharing that view, resulting in faster incremental builds.

When asked why the teams use different tools, most responses indicate political reasons (only 1 employee claimed to know the full history of the decisions, yet all participants offered speculative reasons). The teams chose different tools due to priorities of the projects at the time; the Omega project conservatively chose CodeMgrTool because they felt ClearCase was not yet proven in the market. Most employees feel that decisions were most influenced by the personalities of those who championed the tools; they also cite breakdown of communication among the teams. When questioned why Omega does not switch tools, considering the overwhelming support for ClearCase, answers provide more pragmatic than political reasons. Plans to switch were underway for several years but not implemented due to limited time, budget and manpower, lack of priority, reluctance to change, and technical challenges of conversion.

Change Incorporation Process Evaluation

Alpha's change incorporation process allows developers to introduce changes into the baseline at any time; developers call it a "lightweight process" because configuration updates are distributed among all users. Omega follows a batch process: a group is responsible for periodically incorporating multiple changes into the system at once. Preference for a process is less divided than for the tool (60% favored Alpha's process); survey participants noted advantages and disadvantages for both processes, and interestingly wished to find a medium between the two.

The most notable problem with Omega's process is not the process itself, but its implementation. The master toggle rarely occurs on schedule, creating intolerable latencies. The opportunity to synchronize with the latest release comes infrequently, and often the long lapses tend to break more software when updated, so incompatibilities are expected rather

than prevented. Because responsibility for maintaining the integrity of the release lies with the build group, the process encourages less consideration of the impact of changes on other parts of the system. Alpha's developers introduce changes one at a time making them easier to manage. In Alpha's process, version updates are synchronous and conflicts detected and resolved immediately by the developer, who is in a better position to fix problems than an independent group.

Although batch processing has weaknesses, it also has advantages for large projects. The build group acts as gatekeeper, protecting the main baseline from instability and coordinating changes from multiple activities on different components of a complex system. Also due to awareness of the latency, only fully developed and reviewed changes are submitted.

4 CONCLUSIONS

The employees of FooMediCo were surveyed to determine the use patterns of two different configuration management systems applied to different products. Section 3 presents the tools and processes applied to configuration management, and the opinions of the developers. The major patterns discovered by the study indicate that:

- Workspace semantics have a direct impact on the use of check-in. For example, since workspaces in Alpha immediately made any checked-in object visible to all developers, check-ins occurred more frequently; in Omega, the batch process used to bringover objects into the build workspace led to less frequent check-ins, with each check-in typically representing a completed task.
- Build semantics are tightly linked to workspace semantics. In Alpha, changes appear immediately in the shared development workspace, and builds are performed against this workspace. This leads to a strong desire to avoid breaking the build, and contributes to the observed rapid check-in behavior. In Omega, the build tool's missing object resolution algorithm leads directly to slower builds, and contributes to the observed slow release-to-release cycle time. This, in turn, leads to less frequent check-in. Hence, workspace and build semantics are intertwined, and both have a direct effect on the check-in use pattern. This result is significant, since workspaces, build capability, and check-in are typically portrayed as independent operations in descriptions configuration management systems; these results suggest the situation is much more complex.
- Once a tool is in place it is difficult to replace, even with much transfer of experience from another tool. One striking result of the study is Omega's long-sustained resistance to switching tools, considering the disdain for CodeMgrTool and the fact that ClearCase may adapt to Omega's process. Tool dependency becomes entrenched independent of process.
- Developers expressed an interest in a hybrid change incorporation process, combining the advantages of the batch and lightweight processes. Finding the proper balance of combining lightweight and batch processes would be a beneficial undertaking for FooMediCo.

Future Work

One of the weaknesses of these conclusions is that they are based on the observation of a small population within a single organization. Further research to increase the number of people interviewed, and the number of organizations considered, will increase our confidence in these results. A more general questionnaire, instead of one tailored to a single company, is necessary to broaden the scope of the survey.

Our study strongly suggests that a thorough examination of the interplay between workspace semantics, build semantics, and check-in behavior will yield a deeper understanding of the forces driving use patterns of configuration management tools. Future work may quantitatively measure this interplay by collecting check-in and build frequency data, and benchmarking product quality and developer productivity. It will be enlightening to learn if our observed user's perception of each process and tool correlates with measurable performance data. We hope this research will provide valuable input into the design of next generation configuration management systems.

ACKNOWLEDGMENTS

We would like to thank the FooMediCo employees for their participation and valuable insight. We would like to thank our anonymous reviewers, who created more questions than our study answered, paving the way for future work.

REFERENCES

[1] E. H. Bersoff. Elements of software configuration management. *IEEE Trans. on Software Engineering*, 10(1):79–87, Jan. 1984.

[2] H. Beyer and K. Holtzblatt. Apprenticing with the customer: A collaborative approach to requirements definition. *Communications of the ACM*, 38(5):45—52, May 1995.

[3] F. J. Buckley. Implementing a software configuration management environment. *Computer*, 27(2):56–61, Feb. 1994.

[4] D. Eaton. Configuration management tools summary, Aug. 2000. At the printing of this paper, the comp.software.config.mgmt newsgroup FAQ is online at `http://www.landfield.com/faqs/sw-config-mgmt/cm-tools/`.

[5] R. Grinter. Supporting articulation work using configuration management systems. *Computer Supported Cooperative Work: The Journal of Collaborative Computing*, 5(4):447–465, 1996.

[6] W. Keuffel. Configuration management. *Computer Language*, 9(11):31–34, 1992.

[7] M. C. Paulk, B. Curtis, M. B. Chrissis, and C. V. Weber. The capability maturity model for software. *IEEE Software*, 10(4):18–27, July 1993.

[8] J. Plaice and W. W. Wadge. A new approach to version control. *IEEE Trans. on Software Engineering*, 19(3):268–276, Mar. 1993.

[9] Rational Software Corporation, 18880 Homestead Road, Cupertino, CA 95014 USA. *Rational ClearCase Manual*, 1999.

[10] M. J. Rochkind. The source code control system. In *IEEE Trans. on Software Engineering*, pages 364–370, 1975.

[11] Sun Microsystems, 901 San Antonio Road, Palo Alto, CA 94303 USA. *Sun WorkShop TeamWare: User's Guide*, A edition, Dec. 1996. At the printing of this paper an online version of the manual is available at `http://docs.sun.com`.

A SURVEY QUESTIONS

1. What FooMediCo products do you work on (now and in the past)? What are (or were) your roles?
2. In your own words, what is the purpose of software configuration management?
3. For the products you are familiar with, please describe in detail the CM process (please walk through an example).
4. How do you know when to check-out a file? When is check-out complete?
5. How do you know when to check-in a file? How are check-in conflicts handled?
6. When and why did you examine a file's version history?
7. Are you familiar with the tool of both products? If so, which do you prefer (independent of process)?
8. Are you familiar with the process of both products? If so, which do you prefer (independent of tool)?
9. Do you know why the tools were selected?
10. Do you know why product teams do not switch to the same tool/process?