Rapidly Evolving Software and the OVERSEE Environment

by

Steven Wartik
Department of Computer Science
University of Virginia
Charlottesville, VA 22903

Abstract

During its lifetime, a software system will sometimes need to "rapidly evolve," that is, undergo a quick set of changes. Making the changes rapidly is difficult, especially if one's software development policies are rigorous; the need for test reports, signatures, etc., seems to create interminable delays. In this paper, we argue that much of the problem stems not from such policies, but from a lack of consideration to information flow in software environments. We present a configuration management environment called OVERSEE, and discuss how it helps solve the problems of information flow.

1. INTRODUCTION

A software product, towards the end of its development, often experiences a period of growth spurts. Testing in real-life situations—alpha and beta testing, for example—adapts the software, through a rapid series of modifications, from a ragged-around-the-edges product into something useful. This "rapid evolution" phase is, unfortunately, common in software development. Making these changes quickly is a difficult, error-prone task, and the high costs of doing so have long been known [1]. During alpha and beta testing, users become dependent on existing software, even though it is advertised as preliminary. Changes, whether bug fixes or enhancements, must be introduced carefully if compatibility is to be maintained with previous versions. However, it must be possible to make changes rapidly. Anything that inhibits one set of modifications slows down the next set, delaying access to the improved software, not to mention postponing the product's delivery date. Because the delays in introducing a change generally result from the project's software development standards (test case preparation, review boards, etc.), standards are often relaxed when a change is needed quickly, which in the long run usually exacerbates the problem.

Rapid prototyping can help solve this problem. However, while it reduces the need for changes to functionality, it does not necessarily affect the number of bugs [2]. Moreover, the problem appears across the entire life cycle. Frequent changes occur at the end of the requirements and design phases too, since reviews usually find holes in the requirements or the design. Rapid evolution seems endemic to a software project.

In this paper, we argue that many problems during rapid evolution are due to poorly structured information flow within a software development environment. In the following section we describe information flow. We next discuss OVERSEE, a configuration management environment we are building, and then cover how OVERSEE helps alleviate the problems of information flow.

2. INFORMATION FLOWS AND NON-FLOWS

Software development involves much information flow. Flow is usually in the form of files, containing documents, forms, binaries, etc. It occurs due to the need to share information between the members of a software project. For

This research was supported in part by NSF Grant DCR-8602674.

example, most modern operating systems have a hierarchical directory structure. Developers work in their own directory area so as to have an independent development environment, and a central database area serves as a repository for files that are part of the official configuration. Developers send and receive information to and from this area, duplicating most source code and requiring its flow between the two areas.

Because of the difficulties in organizing and tracking information flow, software development organizations usually inhibit information flow except under strictly regulated conditions; most changes require written approval from several parties. While such approaches have been used on many successful projects, they are bureaucratic and waste time and effort. One or more human configuration managers must oversee every step of development. Furthermore, the bureaucracy can discourage entry of important information. Submitting an afterthought—for example, improvements to commenting—becomes irritating and time-consuming; most developers would feel that, because the program already works, the change is not worth the effort.

Automated CM systems help organize information flow. However, information *non-flow* causes equally many software problems. Non-flow is omitted information: something that should be entered into the system but is not, for any reason. Examples of non-flow include commenting improvements, notes on program design, or sending a set of files to a tester but forgetting to include one. Non-flows have both short-term and long-term consequences. A forgotten source code file will cause recompilation to fail but will be rectified within a day or so. A poorly commented program has costs at some undetermined future time for developers attempting to modify it.

Non-flow is often caused by carelessness, but equally often it is caused by the difficulty of initiating flow. The commenting example illustrates this point. CM systems are too often based on a company's non-automated CM practices; instead of passing paper data between offices, they route electronic data to appropriate users. This approach fails to realize that the computer can assume many duties of the CM, and that all parties no longer need to review the data. Existing practices should be re-thought before being automated. In particular, we should concentrate on:

1. *Eliminating unnecessary information flow paths.* Each flow path begins and ends with human interaction, and each human interaction introduces delays.
2. *Encouraging information flow.* Paths should be open and accessible, and submitting useful data of any sort should be simple.

3. AN OVERVIEW OF OVERSEE

OVERSEE is a configuration management environment consisting of a directory structure along with a set of tools and a methodology for maintaining files within that structure. In this section we briefly describe OVERSEE and the operations provided by its major tools.

OVERSEE recognizes three user groups: *developers*, who write or modify the system, *testers*, who test the work of developers, and the *configuration manager* (CM), who manages tested versions.

All groups manipulate text files. Text files may be *configured*, meaning that they are officially recorded as part of the system. Each configured file consists of one or more *baselined versions*, each of which is always accessible.

Software is developed and tested in the *System Development Area* (SDA). Tested software, namely that suitable for general distribution, is stored in the *System Configured Area* (SCA). Because most data flow occurs in the SDA, we shall concentrate on it. The structures of the SDA and the SCA were adapted from those used by the SPS [4] and SPMS [3]. These in turn resemble the top level of the UNIX™ file system hierarchy with additional directories for information on the requirements and design phases—more precisely, with a directory for each phase of the software life cycle. OVERSEE also adds an area known as the *mini-environment*, explained below, for the testing of products from the current development phase. Finally, an area exists for storing project information and history that does not fit cleanly into any other area. Each of these areas is a hierarchy that reflects the project's structure, with subdirectories for logical subprojects.

Any source file (one that cannot be generated from other files) that is to be regarded as an official part of the system must be configured, whether it is in the SCA or the SDA. OVERSEE permits configured files to be "changed," "baselined," "unit-tested," and "integration-tested." (The "change" operation is a notification of intent to change; "baseline" is a notification that the change has been made. This is analogous to the RCS concept of check-in and check-out [6], for example.) A test phase may "accept" or "reject" a software product. The order in which these operations are performed within the SDA is shown by the data flow graph in Figure 1. This model is our common denominator for

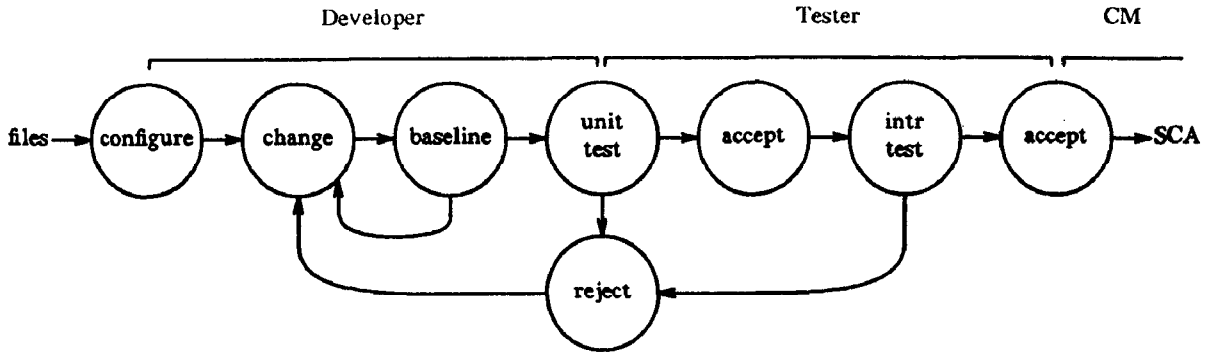

Figure 1 Operation Order

information flow. It assumes only that source is to be tested, first by the developer and then by a tester, and so is applicable to any life cycle model. Note that the nodes are points of interaction with humans. We do not specify exactly the information required at each node (much of it depends on contractual requirements), but instead employ a form-oriented tool called Fillin [7] that can adapt to different data configurations in a simple way. We also do not specify exactly what the interaction consists of, as this depends on a particular company's organization; acceptance might require a single electronic signature, or the joint approval of a change control board. OVERSEE accommodates either scenario.

CM tools have traditionally stored information in a heavily-protected database, where the CM has absolute control on whether information is entered. In TRW's PMDB [5] and in CMS [9], for example, separate areas exist for software under development, software being tested, and software that has been tested and installed. In OVERSEE, however, the development and testing areas are merged into the SDA, and the SCA has a close relationship to the SDA. The following concepts are most important in supporting this. (We explain why in the next section.)

1. All software development relevant to a project is done under a single directory (not spread across different developers' accounts). Each developer creates, configures, and unit-tests his portion of the software in a set of directories with no special access restrictions.
2. When his software is ready for integration testing, the developer places it (through the "integration-test" operation) in a "mini-environment" that mimics the configuration ultimately adopted by the tested, installed software. Note that the developer does not submit files to a CM for testing. Rather, he notifies the tester that a product of his in the SDA is ready for testing. OVERSEE maintains a list of source files relevant to any product, and the tester is given that list; he (the tester) then reviews the source files, in the SDA. File locks prevent modifications to the files until the tester either accepts or rejects the product.
3. The mini-environment stores files that are tested or in the process of being tested. It is accessible to other developers, so object libraries may be placed there. Developers reference only their own source directories and the mini-environment, but not other developers' directories. This avoids the problem of other developers (or the general user community) becoming dependent on private, preliminary versions while giving them access to software that is reasonably stable.
4. Once software is tested and accepted, it is transferred from the SDA to the SCA. This will occur when testing on an entire product is complete. In other words, much of the software in the mini-environment will have passed through testing before the software is moved.

Configuration management systems can help enforce software development standards. OVERSEE allows this through the use of *policies*, a concept similar to that found in MCS [8]. An OVERSEE policy is a boolean-valued operation applied to any component or set of components in the SDA. Each OVERSEE command has an associated set of policies that are tested whenever the command is invoked. The exact set of policies associated with a given command is project-specific and depends on the particular life cycle and standards in effect. The following are examples of policies that have been implemented in OVERSEE:

1. Each subroutine must have a comment header in a standard format. The header contains at least the routine's name, authors

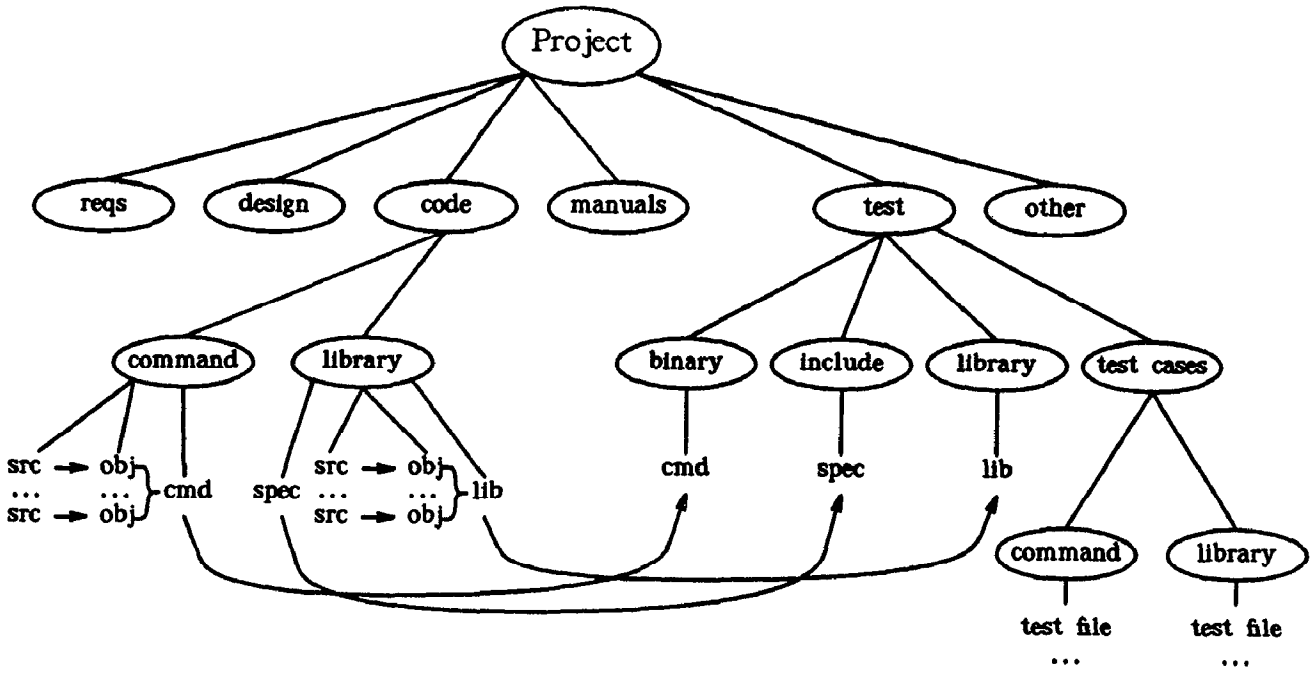

Figure 2 Sample Project Structure

and a copyright notice.

2. All permanent file names must be implemented as defined constants, not hard-coded into a program.
3. Each (sub)system must contain a "makefile" with a standard set of directives to for compilation and installation.

Figure 2 presents the general structure of a software development project configured under OVERSEE that illustrates many of the concepts presented in this section. The project has passed through requirements and design into coding (else there would be no files under the code directory). The project is to produce:

- A library of software modules. Each module is obtained from one or more configured source files, which compile into relocatable object files, which are then placed in the library.
- A shell-level command. This command is built from configured source files, which compile into relocatable object files. These files, along with several objects from the library, are linked to form the command.

The testing area contains versions of software from the code area in the integration testing phase. Test files in this area are stored in a hierarchy that duplicates the subproject organization in the coding area; the similarity helps in locating appropriate test files and plans. The other parts of the testing area represent the mini-environment. The curved lines show what types of files are taken from the developers' areas and where they reside in the mini-environment.

The software in the mini-environment is being referenced by several people. It has already been through unit-testing (a necessary precondition to being exported outside the area from which it originates) and is now being used by testers. Also, the person developing the command is extracting the necessary modules from the library.

4. SUPPORTING RAPIDLY EVOLVING SOFTWARE

We claim a configuration management system must do the following to support rapidly evolving software:

1. It must minimize information loss.
2. It must reduce the overhead of initiating testing and installation.
3. It must allow the developer to experiment with different versions.
4. It must allow simple recovery of previous versions.

This section describes OVERSEE's support for the above.

4.1 Minimizing Information Loss

Information is lost in two ways. The first is through leaks within the system. Losses of this sort are minimal in an automated environment, so we shall not comment on them further. The

second, more serious way, is through the non-flow concept discussed earlier. Any place where flow may be initiated represents a potential non-flow point. OVERSEE attempts to eliminate information flow in three major ways.

4.1.1 Reducing Data Flow

One of the first projects in OVERSEE was to build a general-purpose model for software development and to analyze data flow within that model. The model assumes little more than that if a product is to be part of the installed configuration (which includes much more than just code) then it be tested in some manner. Even so, the complexity of information flow was soon apparent. The data flow diagram of Figure 1 is accurate only for a project with one developer, one tester, and one CM. In reality, a highly complex set of interactions exists between a multitude of project members, each interaction a possible non-flow point. Simplifying the data flow is an important goal.

Using the mini-environment and merging the testing and development areas are the principle techniques for reducing non-flow. They eliminate a large number of flow paths and information entry points; since less information flows, less can be lost. Software that compiles in the mini-environment needs little transformation when it is moved to the SCA. Similarly, there is no flow between development and testing areas, since they are the same. Projects are often delayed for a day or more due to non-flows of this nature; OVERSEE solves the problem by eliminating the possibility of their occurring.

4.1.2 Note-Taking Commands

OVERSEE also provides special note-taking commands to allow for spur-of-the-moment thoughts. These are shell-level commands, and, combined with Unix's context-switch facilities, are simple to access at any time. While the notes entered are not necessarily organized in any logical fashion, they exist, which is better than typical scenarios. For example, a developer modifying code might enter some thoughts on understanding the design that were not covered in the design document, which is easier and quicker than modifying the design document itself (and hence more likely to be done). We are presently studying techniques for categorizing and reporting such notes; currently, they are timestamped to help maintain project history. OVERSEE simply requires that all notes be re-read before integration testing ends. The intent is that re-reading a design note will encourage it to be added to the design document.

4.1.3 Accounting for Unconfigured Files

OVERSEE allows any file to become part of the SDA through a single command. Not all of these files become part of the SCA, but OVERSEE tracks all files in the SDA and can notify a user of the presense of a file that does not fit into the expected configuration. Suppose a developer forgets to configure a source file; since such a file must be tested, it is lost information. The notification gives the developer a chance to recover the information before the software leaves his control (which is when lengthy delays are possible).

4.2 Reducing Testing Overhead

Passing software through the testing phase—preparing the necessary forms, getting proper approvals, etc.—can create lengthy delays. This is undesirable for rapidly evolving software. However, relaxing the testing standards is not acceptable; small changes should be subjected to the same review process as large ones.

OVERSEE's solution to this problem is to increase the developer's control over the testing process. In most CM scenarios, a developer must inform the CM that testing is to begin. The CM will then copy a set of files into the testing area, and notify all parties involved in the testing. However, in an automated CM environment, the (human) CM's presense is not needed until after testing is completed. Thus developers control when testing commences, as discussed above; OVERSEE is responsible for notification and file locking. This eliminates several information flow paths between the developer, the tester, and the CM.

A potential drawback to this scheme is that the developer can, by placing a new version of software in the mini-environment during integration testing, affect another developer who is referencing the old version. We do not view this problem as serious; it involves no information loss and, as explained below, can always be corrected with a single recompilation command. Software under development is subject to bugs in any case, and in our experience there does not appear to be a significant increase in lost time from changes to the mini-environment. To help guard developers from surprises, OVERSEE sends an electronic mail message to all developers connected with a project when any part of the software within that project is about to be tested.

Policies also reduce testing overhead by eliminating some of the most tedious and time-consuming parts of testing. The examples in the previous section are usually accomplished by manual code reviews. Their implementations were not foolproof—checking adherence to standards is a complex pattern-recognition problem, and there is still no substitute for a careful code review—but they uncovered many simple errors caused by

carelessness. OVERSEE helps developers pinpoint and fix such problems before a tester sees the code. This saves testers' time and eliminates a large information flow between developers and testers.

4.3 Experimenting with Versions

While developers use the mini-environment for software that is being tested, they can easily create different versions of the software in their own areas. They therefore doe not interfere with files of theirs that other developers are using. Experiments with variations on interface styles or functionality can be conducted in isolation from anyone else's work.

4.4 Recovering Previous Versions

Since the developers do their work in the SDA, and since they have normal file access permissions, they can damage the configuration. We now discuss two issues: first, how much trouble a developer can cause, and second, how difficult it is to recover from that trouble. We consider carelessness rather than maliciousness, but note in passing that an ordinary developer lacks the ability to destroy anything except his own files, and only the latest versions thereof. Hence, even maliciousness cannot necessitate more than a recompilation of the previous version.

OVERSEE permits the developer to modify only the set of objects that he is developing. Modification is done through extension, so previous versions can never be lost. This in itself means that problems can cause delays no longer than recompilation time. It is also important to understand exactly what the developer can affect. Aside from his own files, he may cause some modification to software that other developers (but not users) require. Insuring that access to this common area does not disrupt other software developers is therefore important. Suppose, for example, that a developer is creating some object file that is to be part of a publicly accessible object library. OVERSEE requires (in a way that can be automatically verified) that a developer test the file before it is placed in the public testing area. Furthermore, the last version of the public file is preserved, and developers can resort to it if bugs impact their work.

Finally, all software that has passed through testing is stored in the SCA. Here, it is completely removed from the developer's responsibility, and so cannot be harmed by him. Most of installation is automated, invoked by a single command. It is based on the Unix "make install" convention, but extended to account for installing particular versions. This minimizes the possibility for error on the part of the CM.

5. CONCLUSIONS

Few software projects avoid a rapid evolution phase. In this paper, we have discussed two methods used by the OVERSEE environment to reduce the consequent problems: the elimination of excess information flow, and attempts to prevent non-flow. The reader should not conclude that the problems discussed are unique to rapid evolution phases, nor that OVERSEE handles only this part of the software life cycle. Rapidly evolving software exposes problems in one's software configuration procedures more than other times, due to the need for haste; since we do not have space for a complete discussion of OVERSEE, we have concentrated on this one aspect.

OVERSEE is the result of an attempt to rethink CM practices by asking "How can we manage software on a computer?" rather than asking "How can we automate current software development practices?" We have answered this question largely in terms of our information flow analysis. Several ideas unique to OVERSEE have resulted, including merging the testing and development areas, keeping test data out of the central database (the system configured area) until the testing phase is completed, and allowing the developer to control the testing phase. These ideas might, at first, seem to border on heresy. In practice, we have not seen them cause trouble; rather, they have greatly simplified software development. A prototype version of OVERSEE has been used in the development of several small software systems with encouraging results, and the information flow seems natural for an automated environment.

REFERENCES

[1] B. Boehm, *Software Engineering Economics*, Prentice Hall, Englewood Cliffs, NJ, 1981.

[2] B. Boehm, T. Gray and T. Seewald, *Prototyping vs. Specifying: A Multi-Project Experiment*, UCLA Technical Report, Computer Science Dept., University of California, Los Angeles, CA, 1982.

[3] P. Nicklin, *The SPMS Software Project Management System*, Unix Programmer's Manual (4.2 Berkeley Software Distribution), Berkeley, CA, Aug. 1983.

[4] M. Penedo and A. Pyster, "Software Engineering Standards for TRW's Software Productivity Project," *Proc. 2nd Software Engineering Standards Application Workshop*, San Francisco, CA, May 1983.

[5] M. Penedo and E. Stuckle, "PMDB—A Project Master Database for Software Engineering

Environments," *Proc. 8th Int. Conf. on Software Eng.*, London, UK, Aug. 1985.

[6] W. Tichy, "Design, Implementation, and Evaluation of a Revision Control System," *Proc. IEEE 6th Int. Conf. on Software Eng.*, Tokyo, Japan, Sep. 1982.

[7] S. Wartik and M. Penedo, "Form-Oriented Software Development," *IEEE Software* 3, 2 (Mar. 1986), pp. 61-69.

[8] A. Wasserman, "The Unified Support Environment: Tool Support for the User Software Engineering Methodology," *Proc. Softfair*, Washington, D.C., June 1983, pp. 145-153.

[9] S. Zucker, "Automating the Configuration Management Process," *SOFTFAIR*, Arlington, VA, June 1983, pp. 164-172.