Copyright © 1975 American Telephone and Telegraph Company
THE BELL SYSTEM TECHNICAL JOURNAL
SAFEGUARD SUPPLEMENT
Printed in U.S.A.

SAFEGUARD Data-Processing System:

Software Change Control

By D. VAN HAFTEN

(Manuscript received January 3, 1975)

Large software projects require control procedures to ensure that changes to code can be made systematically. Programmers, however, wish to be able to make changes to their programs without being bothered by administrative considerations. This paper explores the attitudes of people toward change control and the problems associated with establishing a workable system.

I. INTRODUCTION

Software change control—formalizing the identification and resolution of program errors and improvements—has been critical to SAFEGUARD for three reasons. First, software change control promotes systematic communication. Anyone on the project can formally record a problem. A formal resolution is then ensured; the suggested change is either accepted, rejected, or consciously deferred, but not ignored. Second, software change control helps in estimating the maintenance activity required and in scheduling new software releases. Third, software change control provides visibility. It allows one to see what errors have been found and what action is being taken about them. Based on the requests for changes, one can determine which capabilities of the software are being used. Change control can ensure that the design intent of the software is maintained by consistently identifying all changes made.

II. THE THREE PHASES OF SOFTWARE CHANGE CONTROL

Change control on SAFEGUARD has passed through several stages, progressively becoming more formalized. The first stage was essentially *no control at all.* At the beginning of the project, the only evidence of what the final product would be was a requirements document and a high-level design specification. Developers responsible for building the final product as specified by such documents had a very special attitude toward the software they were creating. Typically, a programmer felt he understood the design considerations and implementation details of the program he was coding, and he felt his knowledge of the code was such that he could remain fully aware of all the ramifications of any changes to it. At this time, the programmer was not bothered with any type of change procedures, for he did not yet have a stable product, and possibly not even well-defined requirements. Any constraining procedures would merely have hindered him from doing his job. This phase of no-control continued as long as the programmer did not have to deliver his program to anyone else.

The second phase might be called *informal change control.* The phase began when several programs had to be integrated, and people other than the original programmer became involved. It was now desirable to have problems documented on forms called trouble reports and solutions—though not coding details—on correction reports. The trouble report and the correction report should be on one sheet of paper, to keep all the information about a problem and its solution together. This procedure met a fair amount of resistance, yet it is only consistent with the standard practice of the scientific and business world, where people write down their ideas, agreements, and problems without feeling they are needlessly harassed by paperwork. Experience has taught them the necessity of doing so. The software world is no different, since programmers, like people, cannot remember every problem and situation they encounter.

For two reasons, it is important that trouble-reporting procedures be set up before they are required. First, if they are not defined beforehand, a vacuum will exist when they do become necessary, and each part of the project will be forced to establish its own. Of course, the primary responsibility of groups that are integrating and testing software is to get a working product, not to define procedures. This responsibility will take precedence, and thus the resulting procedures may not be as good as one might expect. A second and more compelling reason for having change-control procedures defined in advance is that, at a later time when all the software on the project is brought together, it is desirable to have a consistent procedure projectwide.

The final phase of change control for the SAFEGUARD project is called *formal change control.* Since the software is being sent to a remote site for testing and eventually will be sent to the customer, stringent control is essential. In this phase, a central control organization having the following objectives is involved. First, the organization provides consistent, complete, and adequately documented deliveries to remote sites or to customers. Second, it accepts trouble reports, noting problems in the software, and keeps a record of what is being done about these problems. Third, the central control organization checks that certain minimal standards are followed in documenting the problems fixed in each software release, and it checks that all programs to be included in a release are properly identified. Finally, it provides historical backup of all SAFEGUARD software, including source code, object code, assembly listings, and load modules.

Whether a central organization is designated to perform change control or whether this responsibility is scattered throughout the design, test, and integration groups, someone will ultimately do it. On SAFEGUARD, a very small central organization was designated, but it did not insert the actual changes into the code. Therefore, at least one group in each process design department evolved into a control group for that department. Each of these groups defined their own procedures, in some ways making the central organization superfluous. Each department felt it had unique change-control problems that could only be solved by a change-control group that reported to that department's management. It was also felt that only under such an arrangement would the change-control group have the requisite interest in meeting the schedules and objectives of the department. These attitudes made transition to formal change control under one central organization difficult.

The transition from informal to formal change control can be smooth only if there is adequate management backing for such a move. This backing is necessary because of the interjection of a central control organization that is in a position to police certain activities of the development groups. Programmers and even managers are reluctant to let this control organization become involved in their activities, and will probably question both its necessity and its competence. The degree of success this central organization has will depend first on management backing and second on the similarity between the existing informal change-control procedures and the desired formal ones.

III. ESTABLISHING A CHANGE-CONTROL SYSTEM

Thus far, we have considered primarily the human aspects of software change control. The problems associated with this part of the subject are difficult to define and the solutions nebulous. Problems of standards and mechanisms for an effective change-control system are easier to solve. As a rule, these standards and mechanisms are necessary but not sufficient for maintaining control of software.

For three reasons, SAFEGUARD follows a standard procedure for identifying program statements that are changed to fix a given problem. First, the programmers responsible for maintaining the code in the future will be better able to determine the intent of previous changes by being able to relate source statements via their "change level" to a specific correction report. The change level of the program is incremented by one for each correction report written against the program, and the change level is placed on each altered statement.* Second, both the programmer and anyone else examining the code can double-check that the intended change was in fact put in. The third reason, which applies only when the object code is being patched, is that a programmer at a remote site who has solved a problem with a patch may want to check the source code change to make certain it does the same thing his patch did. This checking is especially important when the source code is written in a compiler-level language.

Both source and object code are maintained using a projectwide storage and retrieval system. This system allows the automatic insertion of new change levels into the source code as new statements are added to a program or existing statements are changed. These change levels are then carried through to the assembly listings. In addition, this system provides a convenient mechanism for transmitting changed programs from development groups to testing groups and, ultimately, to the central control organization. The library system was available to programmers early in the project.

When a set of programs is first placed under formal change control, a configuration listing is created. This configuration listing specifies, at the very minimum, a list of all the individual programs with their change levels and a precise identification of all support software (compilers, assemblers, linkage editors, etc.) used in creating this release. With each new release, this configuration listing is updated.

SAFEGUARD programmers write a large number of trouble reports, and an automated mechanism is used to keep track of them. This system was designed and built early enough in the project so that it could have been used to record trouble reports during the informal change-control phase. Although the software Status Accounting System (SAS) was available, developers all seemed inclined to invent their own automated systems because SAS was operated by the central organization at a time when each group wanted to maintain its own data base. These groups should have been allowed to maintain individual data bases using SAS.

SAS can create reports by retrieving and sorting on any data parameter stored for each trouble report and correction report, or on any combination of data parameters. The following information is stored for each trouble report: trouble report number, the program in which

* Since source statements have change levels, so also do object decks and assembly listings. The concept is also applied to load modules, patch decks, user and maintenance documentation, etc.

the problem was detected, the date the problem was detected, a functional description of the problem, the originator of the trouble report, the person to whom the problem was referred for resolution, a status indicator showing the current status of the trouble report, the date of last status change, comments about the trouble report, and the date the correction report is due. The following information is stored for each correction report: correction report number, the program in which the problem was corrected (including its change level as discussed previously), the date the correction report was written, the originator of the correction report, an indicator showing the current status of the correction report, the date of the last status change, and the identification of the load module in which the updated program was first released.

The timing of the definition of forms and procedures was important because programmers became accustomed to the forms and procedures used during informal change control and did not want to convert to others. Thus, the official trouble report/correction report form was defined early in the life of the project, avoiding the proliferation of unofficial versions. The procedures followed during informal change control were a subset of those followed during formal change control. The primary difference, of course, is the presence of the central control organization during the formal period.

The major steps of the SAFEGUARD formal change control process are now described. When someone discovers a problem, he writes a trouble report and submits it to the central organization, which logs it in and forwards it to the people responsible for the program, who accept, reject, or defer it when it is received. They tell the change control organization the date by which a correction for the problem will be submitted. After the people responsible for the program have updated their code, they write a correction report describing the change. They test the new release and update the configuration listing to include the new change levels of programs that have changed. They now send the source code, the object code, assembly listings, a load module, and all correction reports relating to this release to the central control organization. The control organization checks that all changes have been documented; that the source code, assembly listing, and the object code of each program in the load module are consistent; that all program change levels are specified; and that the configuration listing is accurate. This organization then prepares copies of the software for shipment to users or remote test sites.

Although it is not being done on SAFEGUARD, the central control organization should be responsible, upon direction from the development areas, for actually making the changes in all released software.

This requires a substantial commitment of manpower to the organization, but it is one way of ensuring that the changes indicated by correction reports are indeed made, and that no others are.

IV. CONCLUSION

Two aspects of software change control that were relatively successful on SAFEGUARD were a projectwide library maintenance system to control source and object code and a standard trouble report form. These two were not developed over a long period of time, but appeared very early in the project. Because of this stability, software developers grew accustomed to using them. The library maintenance system was available during the first phase (no change control), and the trouble report form was available at the beginning of informal change control. It was recognized that early introduction and acceptance would be beneficial, because transition to the later phases would be simplified. Two additional features of the system, change control procedures and software status accounting, proved to be more troublesome to define and implement. Since, early in the period of informal change control, each process area independently developed its own procedures, a certain amount of reexamination and redefinition was required during the transition to formal change control.

Any software change control system is destined to meet with some resistance. Programmers as a rule have very definite ideas about what should be done to their software. This factor combined with the dynamic nature of software makes change control a difficult problem, not so much in establishing the mechanisms and procedures, as more in dealing with human factors and ensuring adherence to procedures. The first step is to recognize that change control is a problem that should be addressed early and, in fact, *will* be addressed either early in a systematic manner or later in a less organized but more costly manner. Only the developers' admitting this and conscientiously addressing the problem will guarantee successful change control. The mechanisms and procedures suggested in this paper are tools, nothing less, but certainly nothing more. The human factors are the more important considerations in successful software change control.