Reducing Inspection Interval in Large-Scale Software Development

Dewayne E. Perry, *Member*, *IEEE Computer Society*, Adam Porter, *Member*, *IEEE*, Michael W. Wade, *Member*, *IEEE*, Lawrence G. Votta, *Member*, *IEEE*, and James Perpich

Abstract—We have found that, when software is developed by multiple, geographically separated teams, the cost-benefit trade-offs of software inspection change. In particular, this situation can significantly lengthen the inspection interval (calendar time needed to complete an inspection). Our research goal was to find a way to reduce the inspection interval without reducing inspection effectiveness. We believed that Internet technology offered some potential solutions, but we were not sure which technology to use nor what effects it would have on effectiveness. To conduct this research, we drew on the results of several empirical studies we had previously performed. These results clarified the role that meetings and individuals play in inspection effectiveness and interval. We conducted further studies showing that manual inspections without meetings were just as effective as manual inspections with them. On the basis of these and other findings and our understanding of Internet technology, we built an economical and effective tool that reduced the interval without reducing effectiveness. This tool, Hypercode, supports meetingless software inspections with geographically distributed reviewers. HyperCode is a platform independent tool, developed on top of an Internet browser, that integrates seamlessly into the current development process. By seamless, we mean the tool produces a paper flow that is almost identical to the current inspection process. HyperCode's acceptance by its user community has been excellent. Moreover, we estimate that using HyperCode has reduced the inspection interval by 20 to 25 percent. We believe that, had we focused solely on technology (without considering the information our studies had uncovered), we would have created a more complex, but not necessarily more effective tool. We probably would have supported group meetings, restricted each participant's access to review comments, and supported a wider variety of inspection methods. In other words, the principles derived from our empirical studies dramatically and successfully directed our search for a technological solution.

Index Terms—Code inspections: web-based, meetingless, asynchronous, natural occurring inspection experiment, automated support for inspections, work, paper and information flow.

1 Introduction

Code inspections are a commonly used quality assurance technique. In fact, in many organizations, all major software artifacts are inspected. Because of this, changes in the cost of individual inspections multiply quickly across an organization. Our previous research, for example, suggests that the inspection interval (calendar time needed to complete an inspection) increases when the activities of multiple developers must be coordinated [2]. We find that more and more large companies are developing software using multiple, geographically separated teams. In this situation, time-zone mismatches, travel, and long-distance mailings dramatically lengthen interval and are having noticeable effects on total development interval.

Although companies would like to preserve the benefits of inspections, the problem of increased interval is becoming acute. Thus, the research challenge is to reduce the calendar time spent doing inspections without sacrificing effectiveness.

Our first thought was that Internet technology should offer some partial solutions because it can help to close the distance between geographically separated groups. However, the best way to use this technology and the effects it might have were not clear. For example, we didn't know which specific technology to use: video-conferencing, groupware, e-mail. Nor could we guess how each technology would affect inspection effectiveness—without, that is, actually building a tool, deploying it, and observing its use.

The general problem is, how do we decide which of many possible tools we should build? And how do we do this both cheaply, minimizing costs, and reliably, knowing the effects beforehand, not by trial-and-error?

Our conclusion is that we cannot solve this problem if we focus only on technology. We have to understand the factors that drive the task's costs and benefits and then determine how technology affects them. In other words, tools should be designed to leverage the cost-benefit drivers, not just to exercise technology.

In this research, we derived inspection cost-benefit drivers from previous empirical studies, conducted some new studies to fill in gaps in our knowledge, and then used this knowledge to reason about the probable effects of candidate tools.

- *D.E. Perry is with the Department of Computer Engineering, University of Texas, Austin, TX 78712-1084. E-mail: perry@ece.utexas.edu.*
- *A. Porter is with the Department of Computer Science, University of Maryland, College Park, MD 20742. E-mail: aporter@cs.umd.edu.*
- *M.W. Wade and J. Perpich are with Lucent Technologies, Naperville, IL 60566. E-mail: {mww, perpich}@lucent.com.*
- *L.G. Votta is with Motorola, Inc., Arlington Heights, IL 60004. E-mail: lvotta1@email.mot.com.*

Manuscript received 25 May 2000; revised 5 Dec. 2000; accepted 11 June 2001.
Recommended for acceptance by P. Johnson.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number 112193.

We derived two important lessons from the work reported in this study. First, experimentation can be used to *drive* tool development, not just validate the use of a finished tool. We used empirical studies to determine the basic requirements of a tool to satisfy the needs of the developers and we used them to determine choices between various inspection approaches.

Second, we abstracted a process for doing this empirical-study-driven tool development. The process is more or less how we did it ourselves, except for the inevitable stops and starts here and there. In other words, we are presenting the polished and ideal version of the process, but, in reality, there will be iterations, stops, and restarts.

1.1 Our Tool Development Approach

Our approach to tool development involves the following steps:

1. **Select improvement criteria.** Software research has mainly focused on reducing the cost and increasing the quality of software. However, there is a growing segment of the industry whose costs and quality are adequate, but which cannot respond quickly to new opportunities. For them, it is very important to reduce development interval (the calendar time needed to produce software). Since different steps may have to be taken depending on the improvement criteria, it is important to define these criteria early in the improvement process.
2. **Model the cost-benefit drivers of the process.**To gain control over software development we need to have validated theories that are 1) general, 2) causal, and 3) suggestive of control strategies. Such theories point out the factors that drive a process' costs and benefits. They also tell us how to manipulate the factors to get a desired outcome. Tools will be effective if they enable us to manipulate a process' drivers to meet our improvement criteria.
3. **Understand the current process and identify key problems.** The starting point of any process improvement activity should be understanding the current process. Once the process is understood, problem areas can be identified and prioritized according to an organization's goals.
4. **Explore and evaluate alternative improvements.** Potential improvements will have different strengths and weaknesses and empirical studies are fundamental to determining them. These studies explore key issues, risks, and costs of alternative improvements and may involve controlled experiments, surveys, process modeling, and prototype development and evaluation.
5. **Build and evaluate preferred improvement.** Based on the previous analyses, one or more preferred improvements will be selected. In many cases, the preliminary evaluation will not be sufficient to determine the actual range of the improvement. In these cases, the improvements must be built and deployed before they can be properly evaluated. Again, empirical studies are one of our basic tools for their evaluation.

1.2 Overview

In the following sections, we show how we followed this process to improve the code inspection process. After that, we discuss some open questions and present our conclusions.

2 Selected Improvement Criteria

In this research, we are attempting to develop a tool that allows geographically separated teams to conduct software inspections quickly and efficiently. Inspections conducted with this tool should not have a longer interval nor lower observed defect density than the current manual inspections.

3 Modeling the Drivers of Inspection Costs and Benefits

Over the past 20 years, several inspection methods have been proposed. We have reviewed many of these proposals and have found that, too often, competing methods are based on conflicting rationales. For example, one community argues that groupware technology can greatly improve inspection meetings and, thus, greatly improve overall inspection effectiveness. At the same time, another community argues that even well-conducted meetings discover very few defects, so meetings should be discarded all together.

The existence of these and other competing views indicates a serious problem: We don't know what the fundamental drivers of inspection costs and benefits are. Without this information, we can't tell if we are building new methods based on faulty assumptions, evaluating new methods improperly, or inadvertently focusing on low-payoff improvements. To get this information, we identified several potential drivers of inspection costs and benefits. Then, we conducted a family of experiments to evaluate the effect of different drivers.

3.1 Potential Drivers of Inspection Costs and Benefits

Many organizations use a three-step inspection process: Individual Analysis, Team Analysis, and Repair. Based on the current state of research, we suggest that the costs and benefits of this process are driven by the following factors:

1. structure (how the steps of the inspection are organized into a process),
2. inputs (reviewer ability and code quality),
3. techniques (how each step is carried out),
4. context (interactions with other inspections, project schedule, personal calendars), and
5. technology (tool support).

3.2 Investigating Potential Drivers: The Experiments

We conducted the following family of experiments to understand how each class of drivers affects inspection costs and benefits.

3.2.1 Process Structure

Our first study looked at the effect of process structure. Prior to this study, we reviewed several inspection methods

and identified key differences in their structure. The main structural differences were the size of the review team, the number of teams, and the strategy used to coordinate multiple teams. One of our null hypotheses was that none of these factors drives inspection effectiveness. We tested this hypothesis in an 18-month, controlled experiment on a live development project at Lucent Technologies Inc.

Although the effectiveness of proposed inspection methods is supposed to depend in part on how they are structured, this did not have a significant effect in our experiment. Consequently, we suspect that simply restructuring the inspection process will not significantly increase effectiveness. We also found that inspections with only one reviewer were less effective than those with two, but that inspections with two reviewers were as effective as those with four. Furthermore, reviewers of the same code unit rarely found the same defects (finding many defects in common might indicate that most defects were discovered). These results lead us to believe that the performances of individual reviewers had the largest effect on inspection effectiveness. See Porter et al. [17] for a complete description of the experiment and its results.

In some of these inspections, multiple teams had to be coordinated. We found that higher degrees of coordination led to an increased premeeting interval (the calendar time from the beginning of the inspection to the inspection meeting.) Thus, we concluded that some relatively straightforward changes to process structure had strong, negative, effects on at least a part of the inspection interval.

3.2.2 Process Inputs

The inspection performances described above show considerable variation. This suggests that something other than process structure has a strong effect on inspection effectiveness. One obvious possibility is that the variation is due to differences in the process inputs (e.g., reviewers and code quality). We investigated this possibility by modeling variation in the data as a function of process inputs and process structure. Our goal was to determine the relative effects of process structure and process input on inspection interval and effectiveness.

We found that the code's size, its functionality, and the reviewers who inspected the code explained 50 percent of the variation, while the process structure explained only 3 percent. We also found that, even when the variation due to these inputs was factored out, process structure did not have a significant effect on effectiveness. Our interpretation is that the way the code is constructed and the way it is analyzed have far more influence on effectiveness than does the way the process is structured. See Porter et al. [14] for a more details.

For interval, we found that neither the process structure nor process inputs had a large effect on the inspection interval (although process structure had an effect on premeeting interval). These factors only explained about 25 percent of the variation in the data and the models have some mathematical irregularities indicating that they should be interpreted with caution.

3.2.3 Techniques: Inspections With or Without Meetings

The two previous studies suggest that better techniques for analyzing documents may do more to improve effectiveness than better inspection processes will. There are two contexts in which a review team analyzes a document: individually and as a team. Historically, analysis in the team context has been the focus of inspection research and, in fact, this is often called the *inspection* [6]. Some recent studies suggest that, in practice, team analysis is not essential [19]. Since meetings are expensive, it is important to determine exactly how meetings contribute to inspections and whether superior alternatives exist.

From the point of view of defect detection effectiveness, meetings are essential if 1) many faults are found during meetings and 2) because of these meetings, more faults are found than would be found otherwise. That is, whether a group of reviewers is likely to be more effective working together than working separately. To help answer these questions, we examined three approaches for inspecting software. Two approaches involved meetings; a third did not.

We hypothesized that inspection methods that eliminate meetings are at least as cost-effective as methods that rely heavily on them and probably more so. We expected to see this result because we expected the benefit of holding a meeting to be less than the benefit of letting individuals work alone. To evaluate these hypotheses, we conducted a controlled experiment with 21 graduate students in computer science and 27 professional software developers as subjects. We found that the meetingless inspections found more defects than those with meetings. We also found that, when the effort used to hold the meeting was given instead to additional individual analysis, more defects were found. Finally, we found very few defects that were found with greater frequency by inspections with meetings than by those without. These results are essentially identical to those found by Johnson (see Johnson and Tjahjono [3]).

3.2.4 Techniques: Defect Detection Methods

Preparation, the first step of the inspection process, is done by applying defect detection methods. These methods are composed of defect detection techniques, individual reviewer responsibilities, and a policy for coordinating responsibilities among the review team.

Defect detection techniques range in prescriptiveness from intuitive, nonsystematic procedures (such as ad hoc or checklist techniques) to explicit and highly systematic procedures (such as correctness proofs).

A reviewer's individual responsibility may be general, to identify as many defects as possible, or specific, to focus on a limited set of issues (such as ensuring appropriate use of hardware interfaces, identifying untestable requirements, or checking conformity to coding standards).

Individual responsibilities may or may not be coordinated among the review team members. When they are not coordinated, all reviewers have identical responsibilities. In

contrast, the reviewers in coordinated teams might have entirely distinct responsibilities.

We hypothesized that nonsystematic techniques with general and identical reviewer responsibilities lead to overlap and gaps in coverage, thereby lowering the overall inspection effectiveness, but that systematic approaches with specific and distinct responsibilities reduce gaps and improve coverage, thereby increasing overall inspection effectiveness. To explore this hypothesis, we prototyped a set of defect detection techniques called Scenarios—collections of procedures for detecting particular classes of defects. We then compared their performance against those of Checklist-driven and Ad Hoc reviewers in a controlled experiment using 48 graduate students in computer science and 21 professional software developers as subjects.

The experimental results showed that 1) the Scenario method had a higher defect detection rate than either ad hoc or checklist methods, 2) Scenario reviewers were more effective at detecting the defects their scenarios were designed to uncover and were no less effective at detecting other defects, and 3) checklist reviewers were no more effective than ad hoc reviewers. These results suggest that improved defect detection techniques may indeed improve overall inspection effectiveness. See Porter et al. [16] for more details.

3.2.5 Process Environment

Our earlier studies looked at the effect of process structure and process inputs on inspection interval. When we consider total interval, there are no significant differences due to these factors. However, we did see some effect on premeeting interval due to process structure. But, the overall mechanisms were still unclear.

Through direct observation and surveys, we found that developers often have to choose which of their many activities to perform at any given time. We hypothesized that the process environment influences these choices and that they, in turn, influence inspection interval.

Our analysis suggests that process environment does indeed influence inspection interval. In particular, we found that different coding and inspection tasks have different priorities. Therefore, when a developer's workload is high, low priority tasks are deferred. Since some inspection tasks have very low priority, this lengthens inspection interval. We also found that approaching deadlines affected work priorities. These effects were somewhat different for premeeting and postmeeting interval. See Porter et al. [15] for more details.

3.2.6 Summary

For effectiveness, we concluded that technical factors supporting individual performances, (e.g., defect detection methods) have more influence on effectiveness than the nontechnical factors (e.g., structure).

For interval, we concluded that interval is driven by different factors than effectiveness is. In particular, we found that certain kinds of process structure (high degrees of coordination) significantly affected the premeeting interval (from distribution of the code to inspection meeting) and that environmental factors (workload, priorities, and deadlines) significantly effect postmeeting interval.

4 Understanding the Current Process: Identify Key Problems

Abstractly, the inspection process is divided into three basic phases: preparation, collection, and repair. The preparation phases includes such things as initiating the inspection process, disseminating the inspection package, and the inspectors preparing (that is, inspecting the artifact) for the collection phase. The collection phase includes the collection, recording, and assessment of defects. The agreed upon defects are then fixed in the repair phase.

4.1 Understanding the Current Process

We first describe the original process in detail and then present some quantitative data about some critical aspects of this process.

4.1.1 The Process Description

For ease of comparison with the improved process discussed below, we present the original inspection process as a sequence of basic steps.

1. Modification Requests (MRs) are issued whenever corrections, additions, or enhancements to code are needed.
2. A developer accepts one or more MRs and develops the necessary code.
3. The author then makes a code unit available for inspection. A code unit may implement one or more MRs.
4. The author selects his or her review team.
5. The author contacts the review team and schedules the inspection meeting. He or she coordinates the proposed schedule with project management.
6. The author prepares the inspection package and distributes paper copies of it to the review team. The inspection package includes the code unit's source text, information about meeting time and location, and all required forms.
7. Prior to the meeting, the reviewers analyze the code unit looking for defects.
8. The author and reviewers conduct the collection meeting. One of the reviewers is assigned to be the moderator, who makes sure the meeting does not get bogged down on any single point of discussion.
9. During the meeting, the author creates the consolidated list of issues. Issues are the potential defects discovered during the inspection.
10. The author determines which issues must be repaired and does so.
11. The author brings the reworked code to the inspection moderator, who ensures that all issues have been addressed and signs off the inspection.

The original process automates much of Step 6: The code and the changes generated by the MRs are automatically generated for printing and, then, manually distributed. The MR and design documents are made available to the reviewers electronically to save paper.

4.2 Identifying Key Problems

We identified two major contributors to delays in the inspection process. One is (unnecessary) overhead created by Lucent's formal development process. The other is blocking due to synchronization and sequencing of inspection subtasks.

4.2.1 Process Overhead

Process overhead comes in several forms. One type comes from having to create and distribute the documents to be inspected, the documents upon which they depend (e.g., design documents when code is being reviewed), and defect report forms. As we mention later, this problem is magnified when inspectors are geographically separated.

Another type of overhead is incurred because inspection data is used by several other development processes and must be collected, processed, and managed. For example, defects must be opened (recorded) in the change management system when inspections find them and closed after they are repaired. This helps developers verify that all known defects have been repaired before shipment. Also, project management uses this information to track the number of defect reports and to ensure that repairs occur in a timely manner. It is also used to document the quality assurance processes that have been applied to the system. These records must be reviewed and signed by management and archived to comply with ISO-900x requirements. This adds substantial overhead to the inspection process because the data must be collected at inspection time.

4.2.2 Blocking Due to Synchronization and Sequencing

Because of its structure, the inspection process is susceptible to blocking which can lengthen its interval. For instance, a previous study [2] found that schedule conflicts among inspectors often delayed the inspection meeting substantially and, thus, lengthened interval.

This is one example of a more general problem: that dependencies among inspection tasks force some tasks to wait while others catch up. We see this behavior in at least three ways:

1. Many tasks are sequentially ordered. They cannot start until earlier tasks finish. For example, in the standard inspection process, defect repair cannot occur until the collection meeting is finished. Thus, the repair task blocks even though it might be able to continue.
2. Some tasks must be synchronized. Group meetings typically require the simultaneous attendance of all inspectors. First, a mutually convenient meeting time must be found. Then, if one or more inspectors is unable to attend, the meeting cannot occur. Here, the collection meeting may block even though all or most participants are ready to proceed.
3. Inspection tasks are sometimes coordinated. In preparation, inspectors are usually asked to inspect the same documents. In group meetings, it is often important that inspectors are examining the same portions of the document at the same time. In both of these examples, some tasks may be delayed by certain consistency management activities.

These problems exist even when developers are colocated. However, they intensify when developers are geographically separated (sometimes across different time zones). In particular, synchronizing tasks and coordinating the dissemination of critical information becomes more difficult.

5 Exploring Alternatives

We have analyzed the current process and identified several key problems. In this step, we consider some possible approaches for fixing these problems.

5.1 The Solution Space

To reduce delays caused by process overhead and blocking, we considered three approaches.

- Reduction of paper. Put all inspection documents online to allow electronic distribution; require reviewers to record their comments electronically.
- Automatically generate necessary reports. Generate quality assurance records (ISO-900x) and process improvement data from online inspection records.
- Reduce synchronization and coordination. Increase parallelism by reducing synchronous activities and increasing information sharing.

The first two strategies are somewhat straightforward. However, we did take care to make the online process match the existing manual process as closely as possible. The best approaches for reducing synchronization were less obvious. We considered three strategies: eliminating the inspection meeting, sharing preparation results among the review team, and overlapping preparation and repair.

Eliminating the inspection meeting. Our previous research suggests that meetings significantly lengthen inspection interval. In a distributed development environment, this problem becomes even worse. Therefore, we eliminated the inspection meeting.

In terms of effectiveness, this step may significantly reduce the number of defects found and may have other drawbacks as well.

Sharing preparation results. In the manual process, reviewers prepare privately. That is, each reviewer's findings are unknown to the other reviewers until the inspection meeting occurs. With this approach, some effort may be duplicated. Our approach is to make all reviewers' findings public almost immediately. This shares information among all reviewers. In terms of effectiveness, one possible advantage might be that knowledge of the code and its defects are shared among the review team (one reason that meetings are held). On the other hand, this may introduce some bias that inhibits the discovery and disclosure of defects.

Overlapping preparation and repair. Since we elected to eliminate the meeting, the process has only two major phases, preparation and repair (collection becomes a byproduct of preparation). These two phases are normally performed sequentially. That is, preparation is completed before repair can begin. One possible further improvement would be to allow these two phases to overlap. That is, we

TABLE 1
Comparison of Desk and Meeting Inspection Detection Effectiveness for New Code

	Desk	Meeting	Both	Significance
Number of Inspections	202	441	643	NA
Average Faults/Inspection (Faults)	10.1	8.8	9.2	.20
Average Code Size/Inspection (NCSL)	427	327	358	.02
Average Fault Density/Inspection (Faults/NCSL)	.030	.029	.030	.92
Average Repair Interval (Days)	7.1	8.0	7.7	.10

can allow the author to begin repairs as soon as a defect is found.

This may reduce interval, but may also have some effects on effectiveness. In particular, repairing defects implies that the document being inspected will change. If large portions of the document change or if changes are made frequently, then it may make inspection difficult for the reviewers.

5.2 Evaluation and Justification

Eliminating the inspection meeting. We have been unable to conduct a controlled experiment to compare the effectiveness of these two inspection approaches. We do, however, have other data that sheds some light on this topic.

If online inspections are better than manual inspections, then it must be possible to eliminate meetings without decreasing effectiveness. Previous work [16], [13], [18] suggests that this is indeed the case, but, until now, there has been no direct evidence from an industrial environment.

To answer this question, we are exploiting a naturally occurring experiment currently running at Lucent Technologies (see [21] for a similar example). The software development organization was already measuring the effects of, and recording the critical data for, two different inspection processes: desk-based inspections where preparation and collection were done individually and independently and meeting-based inspections which followed the standard process of individual preparation and group collection. Thus, we took advantage of the empirical infrastructure that was already in place.

We compare the results of two classes of inspections: new code (Table 1) and repaired code (Table 2). The significance is calculated using the Wilcoxon-Mann and Whitney Rank Order Test [3], a two-sided test assessing whether the fault densities observed for each inspection when taken from a desk or meeting are drawn from the same distribution.

To determine whether the asynchronous desk inspections are as effective as the meeting collections, we look at inspection statistics taken from almost 3,000 inspections conducted in this environment. Table 1 and Table 2 show these statistics for new and modified code, respectively.

The tables show that there is no difference in the average fault density of new code using desk inspections or meeting-based inspections. There is a significant difference for modified code, but the difference is effectively 0 (.0031 versus .0037). Since this is an order of magnitude smaller than the densities for new code, we conclude that meetingless inspections are no less effective than inspection with meetings.

Sharing preparation results. Having eliminated meetings, the next question is whether to share preparation results. We argue on the basis of Dennis and Valacich [5] that the effects of sharing comments during the inspection process will be no worse than keeping them private. In their paper "Computer Brainstorms: More Heads are Better than One," they argue that there are fewer process losses and useful process gains in sharing information. While the two contexts are not identical, their results do suggest that sharing comments during the inspection process may well have positive results.

TABLE 2
Comparison of Desk and Meeting Inspection Detection Effectiveness for Repaired Code

	Desk	Meeting	Both	Significance
Number of Inspections	2152	197	2152	NA
Average Faults/Inspection (Faults)	.163	.432	.185	$< .01$
Average Code Size/Inspection (NCSL)	26.0	59.4	28.8	$< .01$
Average Fault Density/Inspection (Faults/NCSL)	.0031	.0037	.0031	.03
Average Repair Interval (Days)	1.2	3.3	1.3	$< .01$

Intuitively, the advantages of this shared inspection knowledge are twofold: It incorporates a useful aspect of meetings, namely, the discussion about potential faults among the relevant inspectors and it reduces individual effort by avoiding duplicate work.

Overlapping preparation and repair. Given that inspection data is available as soon as the first inspector begins reviewing the code, there is the possibility of overlapping the repair cycle with the review cycle and, thus, shortening the overall inspection and repair cycle even more.

There are obviously a number of problems that this may induce. First, premature repair may result in repairs on repairs (which may well increase the potential for code decay). Second, there are problems of consistency: The code being inspected is no longer the latest version of the code and, in fact, may even be inconsistent with it. Third, overlapping repair and review may increase the overall effort of the repairer, even though it may shorten the interval.

Clearly, this is a subject that requires serious study to determine the effects of the various interactions between review and repair. Preliminary analysis, however, indicates that overlap looks like a good idea for shortening the overall interval. It is, obviously, subject to some level of risk.

Given the level of risk and the implications of the code changing during the inspection process on the design and implementation of `HyperCode`, we decided not to incorporate the overlapping of repair and preparation into `HyperCode` as a formal part of the supported process.

5.3 Buy vs. Build

Our next step was to determine whether we could obtain a system that met our needs (possibly with some modification) or whether we would have to build one ourselves. To do this, we analyzed several systems that existed at that time, including: Scrutiny, InspeQ, CSI, ICICLE, ASSIST, and CSRS.

Ultimately, we decided to build the system ourselves. One reason for this was that we discovered several mismatches between our needs and the services provided by commonly available inspection systems. Another reason was that it took too much work to update.

We ruled out several systems because we couldn't extend them to meet our needs. In particular, some systems had inflexible processes. For example, Scrutiny [7] only supported a single, hardcoded inspection process. This process included inspection meetings, which we didn't want to conduct. InspeQ [9], developed by Knight and Myers supports their phased inspection process. This process involves inspections made up of several steps or phases. We, however, want to drastically reduce the number of steps. So, we decided against using this tool.

We ruled out several other systems because they provided extra functionality we didn't want. For instance, CSI [12] provides facilities for managing sessions and committing comments. It also makes use of teleconferencing and audio streams. These facilities are essential for same time, different-place virtual meetings, but are not needed in the inspection process we desire.

ICICLE [4] uses knowledge-based techniques to provide some forms of automated defect detection. We weren't interested in developing the rules needed to do this.

We ruled out several other systems because we couldn't integrate them easily with the processes and tools used in our environment. In particular, we didn't find any system that could easily be interfaced with the proprietary configuration management system used in our environment. For example, ASSIST [11] allows users to define and support their own inspection processes, but the system is built over its own document database, making it difficult to integrate with the CMS used in our environment. This problem of platform dependencies also showed up in our consideration of the CSRS toolset [8]. CSRS allows flexible process definition, but relies heavily on supporting technologies emacs and Unix.

These observations led us to decide that, while we might have been able to modify one or more existing systems to meet our needs, it would have been easier to develop a tool that met only our minimum requirements.

6 Building and Evaluating Improvements

Since desk inspections appear to be as effective as inspections with meetings and since we have to support geographically separated inspectors, we built a Web-based inspection tool that supports distributed, asynchronous code inspections. The tool is called `HyperCode`.

In the following subsections, we discuss two views of `HyperCode`: the process view and the implementation view. For the process view, we discuss the basic `HyperCode` inspection process, describe its characteristics, and show how they support the inspection participants (see Fig. 1). For the implementation view, we discuss various implementation details.

6.1 The `HyperCode` Inspection Process

1. Modification Requests (MRs) are issued whenever additions or enhancements to code are needed.
2. A developer accepts one or more MRs and develops the necessary code.
3. The author then makes a code unit available for inspection by interacting with the `HyperCode` tool.
4. The author selects his or her review team, again by selecting their names from a `HyperCode` form.
5. `HyperCode` then contacts the review team and project management who respond to schedule the closing date of the inspection. (There is no meeting in the `HyperCode` process.)
6. `HyperCode` prepares the inspection package and notifies the review team of the package's location.
7. The reviewers analyze the code unit looking for defects. Reviewers analyze the code concurrently, with `HyperCode` automatically collecting all annotations.
8. Once the inspection is closed the author receives the consolidated list of issues from the `HyperCode` system.
9. The author determines which issues must be repaired and does so.
10. The author brings the reworked code to the inspection moderator, who ensures that all issues have been addressed and signs off the inspection.

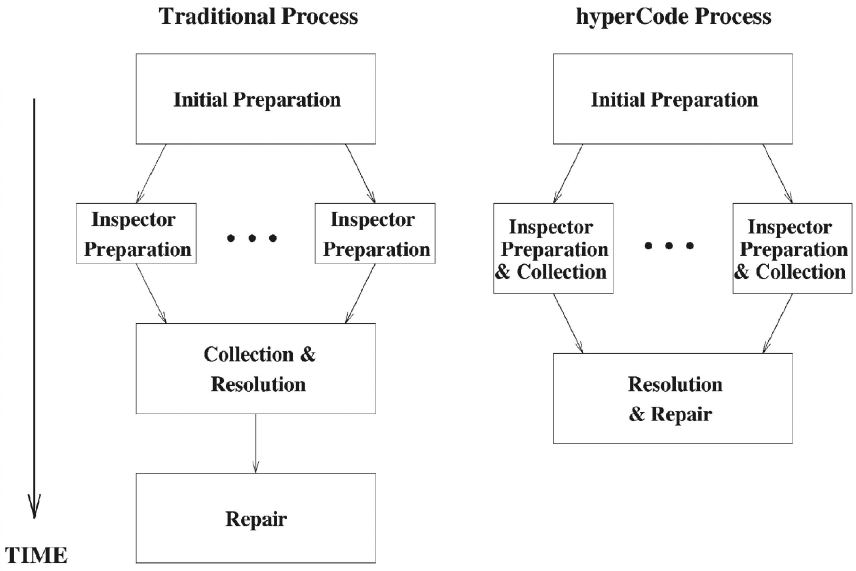

Fig. 1. Comparison of the inspection processes.

The primary differences between the manual and HyperCode processes are as follows:

- automated support for inspector selection,
- automatic notification by e-mail that the package is available,
- all annotations are visible to all reviewers and the author throughout the process (concurrent preparation),
- there is no meeting in the HyperCode process (asynchronous team interaction).

6.2 User Interface

HyperCode is a Web-based code inspection system. During a designated inspection interval, inspectors use a Web browser at their desktop computers to view and annotate the code under inspection. All annotations are viewable by all participants. This inspection process does not require the simultaneous participation of the inspectors nor do inspectors need to be geographically colocated. All that is required for participation is access to the intranet via a Web browser. At the end of the inspection interval, the author and moderator resolve inspector annotations and the author makes code changes as appropriate. All aspects of the code inspection are performed via Web pages. E-mail notification replaces paper meeting notices, status reports, etc.

HyperCode makes use of an already existing tool that generates code inspection packages called sinspect (see Fig. 2). The essential part of the code inspection package is a diff-marked code listing that highlights new and modified lines of source code. Traditionally, this code inspection package is printed on paper and distributed to the inspectors. A HyperCode Web-based inspection package is generated by running the output of the already existing inspection package generation tool through a filter that generates an HTML version of the package (line numbers become hyperlinks that provide the ability to annotate, page numbers in the table of contents become hyperlinks to the corresponding pages, etc.).

The HyperCode inspection package has the same layout as the paper version—experienced developers are therefore immediately familiar with HyperCode inspection packages. The ability to create and view inspection packages, create and manage annotations, send e-mail notifications, etc., is provided by a set of CGI scripts maintained at the Webserver. No special purpose software is needed by users of HyperCode.

An author creates a HyperCode inspection package by bringing up the package creation Web form and entering information about the package, including the usernames of those who are to be inspectors. The author also designates one of the inspectors to be the moderator of the inspection. Standard WWW username/password authentication is used to identify users and control access. The author then submits the form, which causes the Webserver to invoke the standard inspection generation tool and feed the results to the HTML filter, the output of which is the HyperCode inspection package, which is deposited in a node managed by the Webserver.

A HyperCode inspection package goes through a lifetime consisting of four states: pending, in progress, resolution, and done. Packages can be viewed in any state, but annotations can only be made by the inspectors when the package is in the in progress state. A package is initially created by the author in the pending state. The author then moves the package to the in progress state, which causes e-mail notification to be sent to the inspectors and other interested parties (project management, quality team, etc.). The designated inspectors may now inspect the code and make annotations.

At the end of the designated inspection interval, the author moves the package to the resolution state. This state transition again generates e-mail notification to the inspectors and other interested parties. The author then determines the disposition of each annotation and records (via a HyperCode Web page) whether any code changes will be required. After the disposition of all annotations has been determined, the author then informs the moderator via e-mail that the package is ready for moderator sign-off. The moderator then verifies the disposition of the annotations.

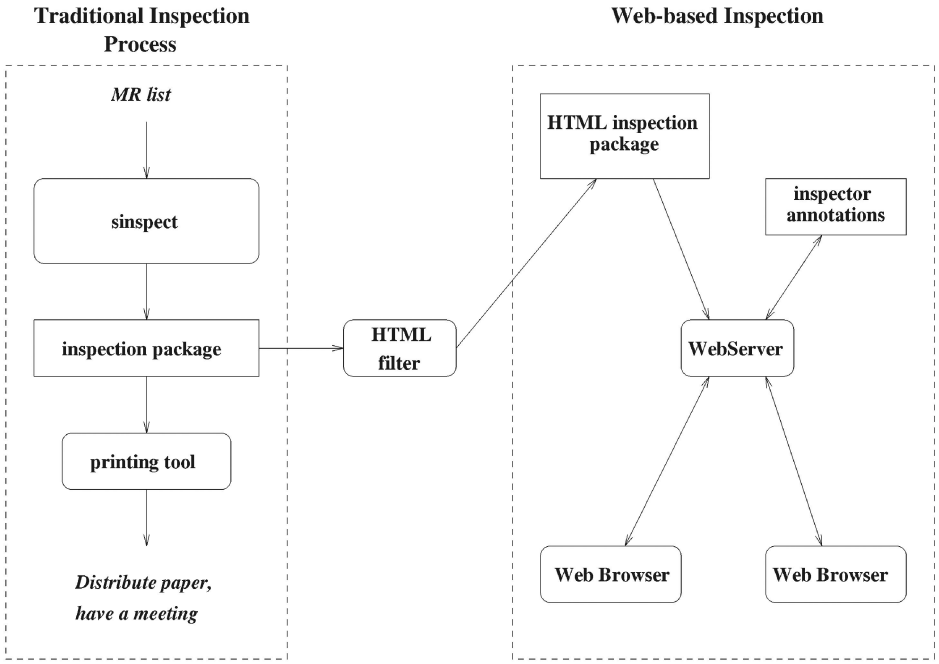

Fig. 2. Generating the inspection packages.

The moderator then moves the package to the done state. This state transition generates a final e-mail notification to inspectors and other interested parties. The inspection report is then generated from the package and its annotations.

6.3 Implementation

Source code line numbers are hyperlinked to a form that allows inspectors to enter annotations. That is, when an inspector clicks on a source code line number, a Web form containing a text input area is presented. The inspector enters the annotation and submits the form, which causes the Webserver to make a record of the annotation. The record contains the username of the inspector, the line number, and source code file name, along with the text of the annotation.

For each inspection package, `HyperCode` provides a page that lists all annotations that have been made to date by the package inspectors. The annotation list contains hyperlinks to the annotation text and to the relevant source code page. The annotation list is ordered by source file and line number. The annotation list page is generated via a CGI script, so the page is up to date each time it is reloaded by a Web browser.

If a source code line has been annotated by an inspector, a graphical element appears in the lefthand margin of the source code display page as a visual cue to inspectors or other viewers of the package. The graphical element is hyperlinked to the text of the corresponding annotations.

In addition to source file-specific annotations, inspectors may also make general annotations that do not refer to any particular line of source code in the package. These types of annotations may be used to record general concerns or issues that are global to the source code under inspection. At the top of each source code display page is a hyperlink to a Web form that allows these types of annotations to be made. General annotations also appear on the annotation list page.

6.4 Evaluation, Experience, and Evolution

The initial acceptance of the inspection tool was excellent. We attribute this to four basic facts: First, the cost savings just from the reduction in paperwork and the time savings from the reduction in distribution interval of the inspection package (sometimes involving international mailings) were substantial. Second, the new intranet tool-based process integrated seamlessly into the existing environment and workflow. This point is both a subtle and a critical one. The disruption of existing workflow almost always causes both resistance and unexpected side-effects. Third, the new process opened up new possibilities for concurrency and inherent speedups of the elapsed time interval. Fourth, the ubiquity of the Web with its distribution and random accessibility, as well as its browser platform independence, made it a natural platform for such an approach as ours.

In the standard inspection cycle, the distribution of the inspection package takes one to two days, two weeks is the standard time for inspector preparation, the collection meeting is usually two hours in length but may take longer, and the overall time interval from generation through the collection meeting is four to five weeks (due to the delay induced by conflicting schedules).

Anecdotal evidence gathered from interviewing users of `HyperCode` indicates that its use has resulted in a reduction of the total inspection interval of about 20-25 percent and has provided a significant improvement in the overall inspection process as well.

- It eliminates the time spent in reproduction and distribution.
- It improves the effectiveness and reduces (on average) the amount of time spent in the defect detection process. This comes about because the other inspector's comments are there for all the inspectors to see and the time spent on each inspector detecting the same defects is eliminated.

- The primary work product of the collection meeting becomes a by-product of the defect detection process (eliminating about one day's worth of work on the part of the author and moderator). The comments are collected as a part of the individual inspections rather than as a major component of an inspection meeting.
- Moreover, the inspection report has improved significantly as a result of HyperCode use.
  - There are more low, medium, and observation category defects, but no fewer severe category defects reported.
  - The defects descriptions for all categories are much more detailed and complete (as opposed to the often cryptic defect descriptions that result from the gathering the defects at the collection meeting).
  - There are recorded discussions of some of the defects because each inspector can see the other's comments. What normally takes place in the collection meeting now takes place as part of the defect detection process.
- The need for a collection meeting is often eliminated, thereby reducing the overall interval of the inspection process because the delay induced by conflicting schedules is eliminated. In those cases where a meeting is held, the use of HyperCode has two important effects.
  - The quality of the meeting is significantly better because the focus is on only the critical issues (the minor issues have already been resolved by inspector discussions recorded in the online comments).
  - The meeting is significantly shorter since there is no time spent in collecting the defects discovered and the time is spent only on those unresolved issues.

  Thus, the time spent in the collection meeting, if there is one, is on average about a half hour instead of two to three. This results in an overall decrease in the delay induced by the conflicting schedules for the scheduling of the meeting because it is easier to find a free half hour rather than a free three hours.

Interest in HyperCode quickly spread beyond its original project. This resulted in several portings to different environmental contexts (different configuration management tools and different Web servers). As a result of these iterations, we created a self-installing portable version of HyperCode in which we separated HyperCode entirely from configuration management issues and evolved the server interfaces so that it now runs on the major Web servers.

The current portable version of HyperCode is now in use in a dozen Lucent projects spread within and across a half dozen countries.

7 Summary

The HyperCode inspection process is clearly more cost effective than the existing paper—driven code inspection process with meetings. First, there is the elimination of the paper production and distribution costs which include such things as special delivery services for time-constrained and critical inspections. Second, there is the elimination of the travel costs that often occur when inspectors are in separated geographical locations for the synchronous meeting to take place.

Furthermore, the HyperCode inspection process is more interval effective. The interval is shortened in several ways. First, the asynchronous approach eliminates the delay inherent in scheduling the inspection meeting [20]. We note that the empirical data we report here is the first such data showing specifically that asynchronous code defect collection is as at least as effective as synchronous code defect collection. Second, we have removed the compartmentalization and sequencing of the typical inspection process and induced a concurrent inspection process in which each inspector proceeds at a pace and time convenient to his or her schedule. Third, if there is a meeting, it is much shorter and easier to schedule.

Finally, there is anecdotal evidence that the HyperCode process is qualitatively better as well. More defects (albeit less significant ones) are detected, the defect descriptions are more complete in the resulting inspection report and, when meetings do occur, they are better because they focus only on unresolved (and, generally, important) issues.

With respect to related work, we note that, while there has been much work on inspection structures, inspection techniques, and automated inspection support, we believe we are the first to report on the use of an intranet-based tool to support asynchronous (that is, meetingless) code inspections. The primary effort in prior automation is in the application of CSCW support for inspection collection meetings—that is, in the support for synchronous meetings (see, for example, [10], [1]). But, as we have shown above, asynchronous code inspections are more cost effective and at least as quality effective as synchronous inspections. Moreover, the cost of asynchronous automated support is significantly less than that of synchronous.

In this context of synchronous versus asynchronous, we also note that our use of an empirically driven tool design process was a significant contributing factor to the success of this research and technology transfer project. Instead of having a preconceived solution to this problem, we empirically isolated the significant factors in the problem and tuned the solution, via experimental techniques, to the problem. Another significant contributing factor to our technology transfer success was the fact that the implementor of HyperCode was a member of both the research team and the project where it was to be used.

In the end, we had a simple and elegant solution that has been easily and quickly adopted in a dozen or so projects with participants in a half a dozen countries and that is being used by over 1,000 people.

References

[1] R.M. Baecker, *Readings in Groupware and Computer-Supported Cooperative Work.* San Mateo, Calif.: Morgan Kaufmann, 1993.
[2] K. Ballman and L.G. Votta, "Organizational Congestion in Large Scale Software Development," *Proc. Third Int'l Conf. Software Process,* pp. 123–134, Oct. 1994.
[3] G.E.P. Box, W.G. Hunter, and J.S. Hunter, *Statistics for Experimenters.* New York: John Wiley & Sons, 1978.
[4] L. Brothers, V. Sembugamoorthy, M. Muller, "Icicle; Groupware for Code Inspection," *Proc. Conf. Computer Supported Cooperative Work,* pp. 169–181, Oct. 1990.

[5] A.R. Dennis and J.S. Valacich, "Computer Brainstorms: More Heads Are Better than One," *J. Applied Psychology,* vol. 78, no. 4, pp. 531–537, Apr. 1993.
[6] M.E. Fagan, "Design and Code Inspections to Reduce Errors in Program Development," *IBM Systems J.,* vol. 15, no. 3, pp. 182–211, 1976.
[7] J. Gintell, J. Arnold, M. Houde, J.K. McKenney, R. McKenney, and G. Memmi, "Scrutiny: A Collaborative Inpection and Review System," *Proc. Fourth European Software Eng. Conf.,* Sept. 1993.
[8] P.M. Johnson and D. Tjahjono, "Assessing Software Review Meetings: A Controlled Experimental Study Using CSRs," *Proc. 1997 Int'l Conf. Software Eng.,* pp. 118–127, May 1997.
[9] J. Knight and E. Meyers, "An Improved Inspection Technique," *Comm. ACM,* vol. 36, no. 11, pp. 51–61, Nov. 1993.
[10] R.E. Kraut and L.A. Streeter, "Coordination in Software Development," *Comm. ACM,* vol. 38, no. 3, pp. 69–81, Mar. 1995.
[11] F. Macdonald and J. Miller, "A Comparison of Tool-Based and Paper-Based Software Inspection," technical report, Dept. of Computing Science, Univ. of Strathclyde, Glasgow, Scotland, 1997.
[12] V. Mashayekhi, J. Drake, W.-T. Tsai, and J. Riedl, "Distributed Collaborative Software Inpection," *IEEE Software,* vol. 10, no. 5, Sept. 1993.
[13] D.E. Perry, N.A. Staudenmayer, and L.G. Votta, "Understanding and Improving Time Usage in Software Development," *Trends in Software: Software Process,* A. Wolf and A. Fuggetta, eds., vol. 5, John Wiley & Sons., 1995.
[14] A.A. Porter, A. Mockus, H.P. Siy, and L.G. Votta, "Understanding the Sources of Variation in Software Inspections," *ACM Trans. Software Eng. and Methodology,* vol. 7, Jan. 1998.
[15] A.A. Porter, H. Siy, and L. Votta, "Understanding the Effects of Developer Activities on Inspection Interval," *Proc. 20th Int'l Conf. Software Eng.,* May 1997.
[16] A.A. Porter, L.G. Votta, and V.R. Basili, "Comparing Detection Methods for Software Requirements Inspections: A Replicated Experiment," *IEEE Trans. Software Eng.,* vol. 21, no. 6, pp. 563–575, June 1995.
[17] A.A. Porter, L.G. Votta, H.P. Siy, and C.A. Toman, "An Experiment to Assess the Cost-Benefits of Code Inspections in Large Scale Software Development," *IEEE Trans. Software Eng.,* vol. 23, no. 6, pp. 329–346, June 1997.
[18] H. Siy, "Identifying the Mechanisms Driving Code Inspection Costs and Benefits," PhD thesis, Univ. of Maryland, College Park, June 1996.
[19] L.G. Votta, "Comparing One Formal to One Informal Process Description," *Proc. Eighth Int'l Software Process Workshop,* pp. 145–147, Mar. 1993.
[20] L.G. Votta, "Does Every Inspection Need a Meeting?" *Proc. ACM SIGSOFT '93 Symp. Foundations of Software Eng.,* Dec. 1993.
[21] L.G. Votta and M.L. Zajac, "Design Process Improvement Case Study Using Process Waiver Data," *Proc. Fifth European Conf. Software Eng.,* Sept. 1995.

Dewayne E. Perry is currently the Motorola Regents Chair of Software Engineering at the University of Texas at Austin. The first half of his computing career was spent as a professional programmer, with the latter part combining both research (as a visiting faculty member in Computer Science at Carnegie-Mellon University) and consulting in software architecture and design. The last 16 years were spent doing software engineering research at Bell Laboratories in Murray Hill, New Jersey. His appointment at UT Austin began in January 2000. His research interests (in the context of software system evolution) are empirical studies, formal models of the software processes, process and product support environments, software architecture, and the practical use of formal specifications and techniques. He is particularly interested in the role architecture plays in the coordination of multisite software development, as well as its role in capitalizing on company software assets in the context of product lines. He is a coeditor-in-chief of Wiley's *Software Process: Improvement & Practice*, a former associate editor of the *IEEE Transactions on Software Engineering*, a member of ACM SIGSOFT and the IEEE Computer Society, and has served as organizing chair, program chair, and program committee member on various software engineering conferences.

Adam Porter earned the BS degree summa cum laude in computer science from California State University at Dominguez Hills, Carson, in 1986. In 1988 and 1991, respectively, he earned the MS and PhD degrees from the University of California at Irvine. He has been an associate professor with the Department of Computer Science and the Institute for Advanced Computer Studies at the University of Maryland since 1992. His current research interests include empirical methods for identifying and eliminating bottlenecks in industrial development processes, experimental evaluation of fundamental software engineering hypotheses, and development of tools that demonstrably improve the software development process. Dr. Porter is a member of the ACM, the IEEE, and the IEEE Computer Society.

Michael W. Wade obtained the PhD degree from the University of Illinois, Urbana-Champaign, in 1984. He came to Bell Labs as a software engineer. At Bell Labs, he has designed and implemented large scale, fault-tolerant real-time databases, and human interface software for cellular switching systems. He has also designed and implemented Web-based software development tools. He is a member of the IEEE.

Lawrence G. Votta received the BS degree in physics from the University of Maryland, College Park, in 1973 and the PhD degree in physics from the Massachusetts Institute of Technology, Cambridge, in 1979. He currently leads the performance and availability modeling and analysis group of the Common Platform Development Department in Motorola's Network Systems Sector. His research interests are high availability computing (new) and empirical software engineering (his old favorite). Larry has authored or coauthored more than 40 articles and book chapters in software engineering, including empirical studies of software development from highly controlled experiments investigating the best methods for design reviews and code inspection to anecdotal studies of a developer's time usage in a large software development. He is a member of the IEEE and ACM and is currently serving as an associate editor of the *IEEE Transactions on Software Engineering*.

James Perpich's biography is not available.