# Anywhere, Anytime Code Inspections: Using the Web to Remove Inspection Bottlenecks in Large-Scale Software Development


J. M. Perpich
Line Access SW Development
Lucent Technologies Inc
Naperville, IL 60566
perpich@lucent.com

D. E. Perry
Software Production Research
Bell Laboratories
Murray Hill NJ 07974
dep@bell-labs.com

A. A. Porter*
Computer Science Dept
University of Maryland
College Park, MD 20742
aporter@cs.umd.edu

L. G. Votta
Software Production Research
Bell Laboratories
Naperville, IL 60566
votta@bell-labs.com

M. W. Wade
Quality Management Group
Lucent Technologies Inc
Naperville, IL 60566
michaelwwade@lucent.com


March 6, 1997

## 1 ABSTRACT


The dissemination of critical information and the synchronization of coordinated activities are critical problems in geographically separated, large-scale, software development. While these problems are not insurmountable, their solutions have varying trade-offs in terms of time, cost and effectiveness. Our previous studies have shown that the inspection interval is typically lengthened because of schedule conflicts among inspectors which delay the (usually) required inspection collection meeting.

We present and justify a solution using an intranet web that is both timely in its dissemination of information and effective in its coordination of distributed inspectors. First, exploiting a naturally occurring experiment (reported here), we conclude that the asynchronous collection of inspection results is at least as effective as the synchronous collection of those results. Second, exploiting the information dissemination qualities and the on-demand nature of information retrieval of the web, and the platform independence of browsers, we built an inexpensive tool that integrates seamlessly into the current development process. By seamless we mean an identical paper flow that results in an almost identical inspection process.

The acceptance of the inspection tool has been excellent. The cost savings just from the reduction in paper work and the time savings from the reduction in distribution interval of the inspection package (sometimes involving international mailings) have been substantial. These savings together with the seamless integration into the existing environment are the major factors for this acceptance. From our viewpoint as experimentalists, the acceptance came too readily. Therefore we lost our opportunity to explore this tool using a series of controlled experiments to isolate the underlying factors or its effectiveness. Nevertheless, by using historical data we can show that the new process is less expensive in terms of cost and at least as effective in terms of quality (defect detection effectiveness).



*This work is supported in part by a National Science Foundation Faculty Early Career Development Award, CCR-9501354.

## 1.1 Keywords

Code inspections: web-based, meetingless, asynchronous; Natural occurring inspection experiment; Automated support for inspections.


# 2 INTRODUCTION AND BACKGROUND

An increasingly popular trend in large-scale software development is the use of development teams that are geographically separated. Instances of this trend range from groups that are contained in multiple buildings to groups that are located in multiple continents. The former tend to be separated only geographically; the latter tend to be separated temporally as well. Where geographical separation tends to encourage asynchronous activities because of cost factors, temporal separation often prohibits synchronous activities because of non-overlapping work hours.

It is in this context that the dissemination of critical information and the synchronization of coordinated activities are critical problems. While these problems are not insurmountable, their solutions have varying trade-offs in terms of time, cost and effectiveness. These solutions range from the simple form of using speaker-phones to multimedia supported and technologically intensive computer-supported cooperative work — that is, from relatively inexpensive (but primitive) solutions to expensive and sophisticated (but as yet experimental) solutions. Note, however, that temporal separation tends to make these synchronized solutions usable only for short periods during the workday at best and completely impractical at worst. For example, at Lucent technologies, the work hours of developers in Chicago overlap with their partners in Hilversum, Netherlands. However, work hours in Denver are disjoint from those in Sydney, Australia.

Because of these two forms of separation there are often bottlenecks introduced into the project schedules. For example, our previous studies [2] have shown that inspection interval is typically lengthened because of schedule conflicts among inspectors which delay the (usually) required inspection collection meeting. The problems of geographical and temporal separation exacerbate the scheduling problems and result in even greater bottlenecks.

A typical approach to process improvement is to introduce a process change (often incorporating a new tool as part of the change), and then to evaluate the effect of that change. While it is certainly necessary to assess the impact of any process change, these improvements are most often done without understanding thoroughly the existing process, where the important problems are, and what the tradeoffs are among the various alternative solutions. Perry, Staudenmeyer and Votta [7] point out the importance of understanding the existing process before making improvements and discuss a set of related studies aimed at gaining that understanding. Bradac, Perry and Votta [5] report a study to find out how developers spend their time — that is, what they actually do as opposed to what they are thought to do. Only by understanding the current process can one find out where the problems are and which of those are important.

Critical to making well-founded improvements is understanding the range of alternative changes and assessing their various strengths and weaknesses. Empirical studies are fundamental to determining the characteristics of these changes. For example, Ballman and Votta [11] report that scheduling bottlenecks caused by inspection meetings lengthen the development interval and that meetingless inspections avoid this problem without loss of the important characteristics associated with inspection meetings. To deepen our understanding of inspections, Porter, Votta and Basili [9] empirically (and replicatably) compare and evaluate detection methods for software requirements' inspections. Siy's thesis [10] has the seminal result of showing that the structural changes commonly-proposed to inspection processes do not alter the effectiveness of those processes.

We present and justify a solution using an intranet web that is both timely in its dissemination of information and effective in its coordination of distributed inspectors. First, exploiting a naturally occurring experiment (reported here), we conclude that the asynchronous collection of inspection results is at least as effective as the synchronous collection of those results. Second, exploiting the information dissemination qualities and the on-demand nature of information retrieval of the web, and the platform independence of browsers, we built an inexpensive tool that integrates seamlessly into the current development process. By seamless we mean an identical paper flow that results in an almost identical inspection process. Additionally the new process is consistent with ISO certification.

We provide the context for inspections in general, discuss the inspection process as it was before and after the introduction of the web-based support tool, and describe the technical details of the inspection tool. We then introduce and discuss the empirical basis and justification for the improved process and consider the various cost elements in the new inspection process. Finally, we report the overall results of using the tool in the last section.

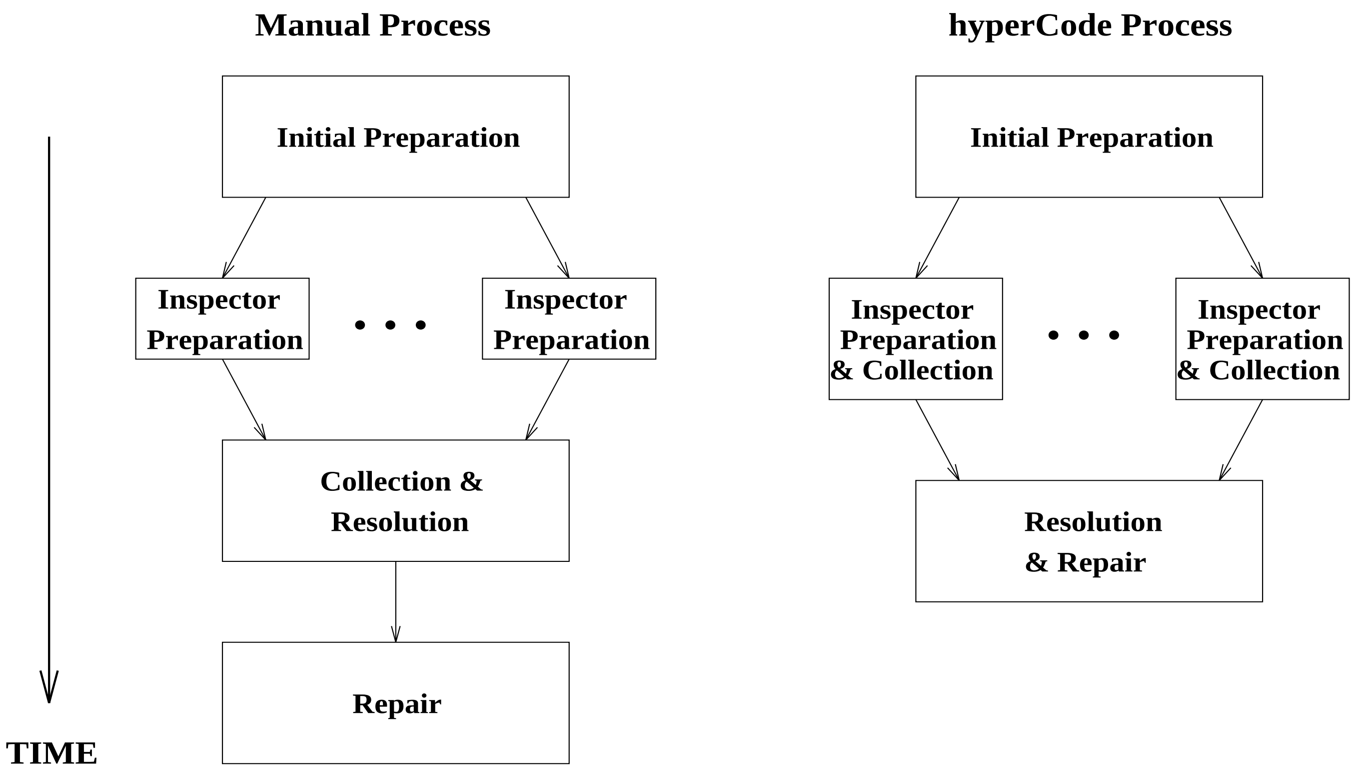


Figure 1: Comparison of Inspection Processes.

# 3 THE INSPECTION PROCESSES

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, assessment and resolution of defects. The agreed upon defects are then fixed in the repair phase.

## 3.1 The Manual Inspection Process

In the initial preparation phase (see Figure 1) the author selects a moderator and inspectors, creates the appropriate versions of the code to be inspected, determines with project management the inspection schedule, and prepares the scenarios to be used during the inspection meeting. The inspection package is then generated and distributed.

The inspectors prepare for the collection meeting by walking through the code following the scenarios provided by the author.

At the collection meeting the moderator coordinates the defect collection process and controls the flow of the meeting to guarantee both thoroughness and completeness. The recorder compiles a written record of the defects and issues. The inspection team completes the process by achieving consensus on resolving the defects and issues.

During the repair phase the author resolves the defects and issues raised in the collection meeting and does the basic bookwork to complete the inspection process which is verified by the moderator.

## 3.2 The `hyperCode` Inspection Process

The initial preparation phase is essentially identical, with a few changes in details: the inspection package when delivered is available on-line rather than as paper with e-mail notification of availability.

The primary difference is in the inspector preparation, collection and repair phases. Here the inspector preparation and collection are done concurrently, with `hyperCode` providing the automatic collection of the annotations, and the resolution of the annotations is done by the moderator and author as part of the repair phase.

# 4 THE hyperCode SYSTEM

We discuss two basic views of `hyperCode`: the process view and the implementation view. In the first, we discuss the observable characteristics of the tool and how they affect the authors, moderators and inspectors. In the second, we discuss various details of how we make things happen, either directly or indirectly.

## 4.1 Process View

`hyperCode` is a web-based code inspection system. During a designated inspection interval, inspectors use the a web browser at their desktop computers to view and annotate the code under inspection (see Figure 2 for an example of the user interface). 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 co-located. 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 (see Figure 3). 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. are provided by a set of CGI scripts maintained at the webserver. No special purpose software is needed by users of `hyperCode` - the only software required of users is the Netscape Navigator web browser (since `hyperCode` makes use of frames, Netscape Navigator version 2.0 or later is required).

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 4 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 par- ties (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 `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.

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.

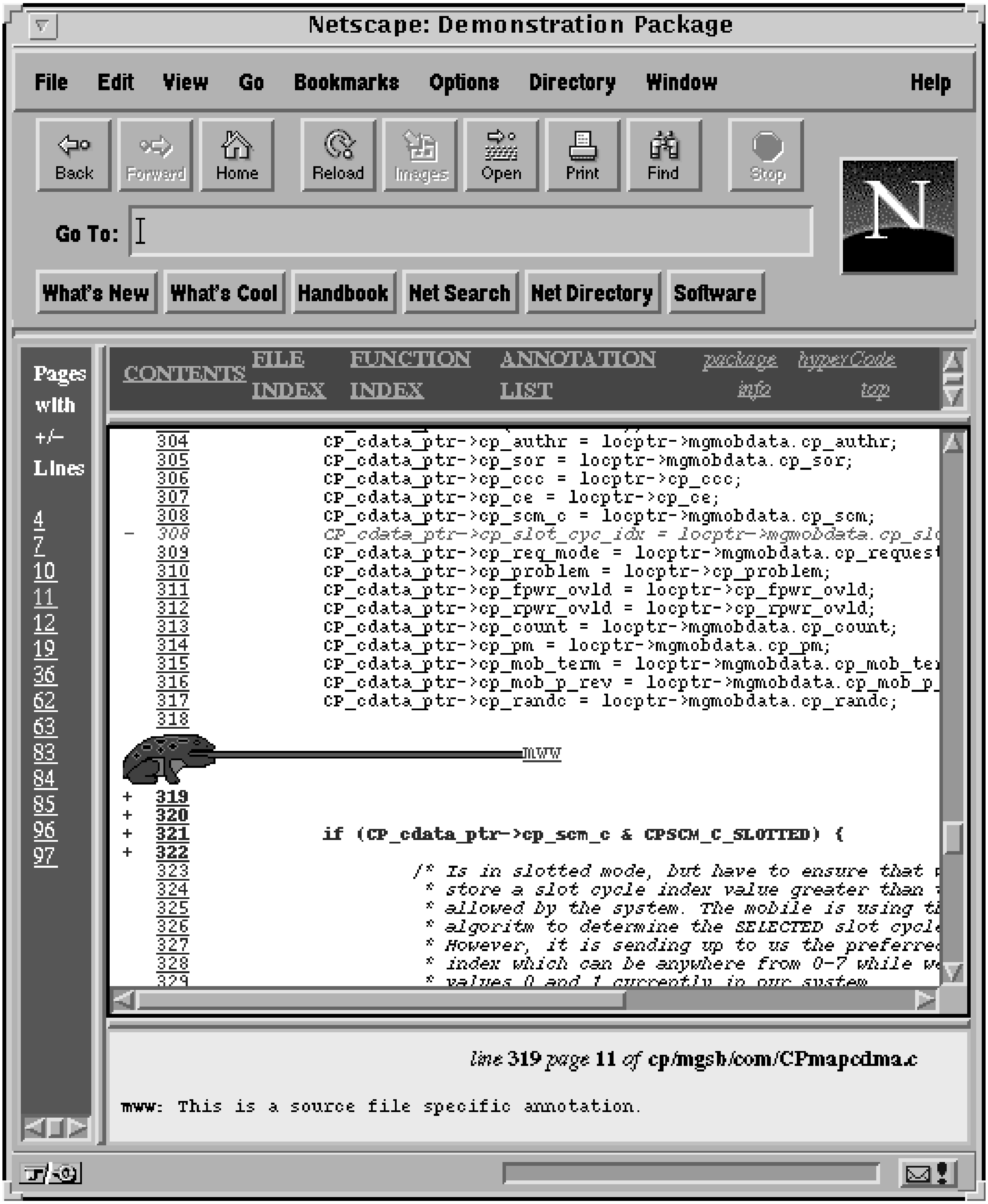


Figure 2: Example of the User's View of hyperCode.

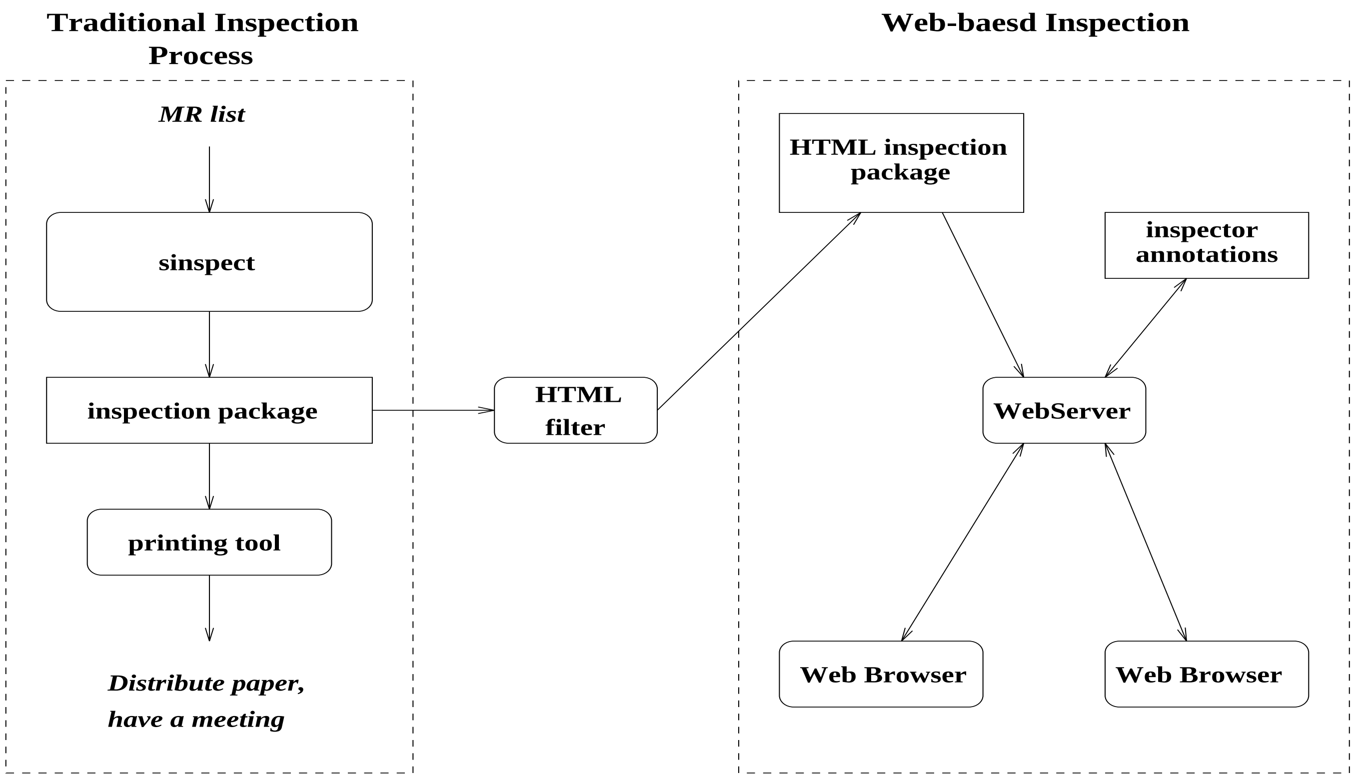


Figure 3: Generating the Inspection Packages.

### 4.2 Implementation View

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. This contains hyperlinks to the annotation text and to the relevant source code page, and 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 left hand 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 type 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 enables these types of annotations to be made. General annotations also appear on the annotation list page.

## 5 EMPIRICAL ASSESSMENT

Given the geographical and temporal separation of many of our projects, it is immediately obvious that electronic distribution saves both delivery time and distribution costs, especially when several continents are involved.

If on-line inspections are better than manual inspections, then it must be possible to eliminate meetings without decreasing effectiveness. Previous work [9, 8, 10] suggests that this is indeed the case, but until now there has been no direct evidence from an industrial environment.

One of the advantages of conducting software engineering research in the context of a number of very large

| | 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 |

Table 1: Comparison of Desk and Meeting Inspection Detection Effectiveness for New 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 | $< .01$ |

Table 2: Comparison of Desk and Meeting Inspection Detection Effectiveness for Repaired Code.

software developments at Lucent Technologies is the possibility of gathering important data and insights via retrospective studies and naturally running experiments. Thus we are fortunate to have data available from one of these existing experiments that enables us to compare the effectiveness of synchronous vs. asynchronous inspections (see [12] for a similar example). The advantage of this approach is that the empirical infrastructure is already in place – that is, the software development organization was already measuring the effects of two different inspection processes (desk-based collection versus meeting-based collection) and recording critical data for the two processes. Hence, there was no intrusion on the part of the experimenters and our role was that of interpretation.

We compare the results from these two classes of inspections: new code (Table 1 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. The smaller the value, the more significant. For this article, we consider values between 0.1 and 0.05 to indicate a mild significance and values less than 0.05 indicate significance.

For example, in Table 1 the row labelled "Average Faults/Inspection" indicates that desk-based inspections (10.1 faults/inspection) and meeting-based inspections (8.8 faults/inspection) are not significantly different since the "significance" is 0.2. Conversely, the difference in the "Average Code Size/ Inspection" between desk- and meeting-based inspections is significant because 0.02 is less than 0.05. Finally, the"Average Repair Interval" is mildly significant (0.1).

To determine whether the asynchronous desk inspections are as effective as the meeting collections, we look at inspection statistics taken from almost 3000 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[1] measures of defects of new code inspections found by desk inspections or meeting-based inspections. There is a significant difference for modified code, but the difference is effectively 0 (.0031 vs. .0037). Since this is and order of magnitude smaller than the densities for new code we conclude that meetingless inspections are no less effective than inspection with meetings.

[1]Porter et al. [2] describes several approaches for measuring and estimating defect detection ratio. We use the observed defect density estimate they recommended.

Moreover, there is very little difference in the time needed to repair new code, though the slightly less time take might be due to overlapping repair with collection.

# 6 RELATED WORK

While there has been much work on inspections 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 [4, 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.

The empirical data we report here is the first such data showing specifically that asynchronous code defect collection is as effective as the synchronous code defect collection.

What has not been taken advantage of is the possibility of further concurrency in the inspection process — namely, that the resolution and repair phase can proceed concurrently with the inspector preparation and collection phase (probably because work patterns are hard to change). While there are undoubtedly cases where defects interact and the expense of coordinated changes is less than separate changes, in most cases the changes are independent and hence concurrent repair would be cost effective[2].

# 7 RESULTS

The acceptance of the inspection tool has been excellent. We attribute this to four basic facts. First, the cost savings just from the reduction in paper work and the time savings from the reduction in distribution interval of the inspection package (sometimes involving international mailings) have been substantial. Second, the new intra-net tool-based process integrates 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 opens up new possibilities for concurrency and inherent speedups of the elapse time interval. Fourth, the ubiquity of the web with its distribution and random accessibility as well as its browser platform independence makes it a natural platform for such an approach as ours.

From our viewpoint as experimentalists, the acceptance has come too readily and easily: we have lost our opportunity to control the important empirical variables and adequately assess the impact of the tool experimentally (see [6] for a description of our desired experimental structure). Because of its immediate acceptance at the grass roots level, the prototype has become a de facto product.

What, then, do we do about this situation? How do we evaluate the effects of a new process when we cannot do the controlled experiments we had originally wanted to do? While not without its drawbacks, the use of historical data (which we do have for a large number of products and their numerous releases) can show that the new process is at least as good as the existing one if there is no drop in cost, interval and quality measures. The primary drawback of course is that we do not have control over the experimental variables which limits the validity of our results.

# References


[1] R. M. Baecker. *Readings in Groupware and Computer-Supported Cooperative Work.* Morgan Kaufmann, San Mateo, CA, 1993.

[2] K. Ballman and L. G. Votta. Organizational congestion in large scale software development. In *Third International Conference on Software Process*, pages 123–134, October 1994.

[3] G. E. P. Box, W. G. Hunter, and J. S. Hunter. *Statistics for Experimenters.* John Wiley & Sons, New York, 1978.

[4] R. E. Kraut and L. A. Streeter. Coordination in software development. *Communications of the ACM*, 38:3:69–81, March 1995.


[2] In software developments where the fault density is higher before inspections, this may not be a good assumption.

[5] P. McCarthy, A. Porter, H. Siy, and L. G. Votta. An experiment to assess cost-benefits of inspection meetings and their alternatives. In *Proceedings of the International Metrics Symposium, Berlin*, March 1996.

[6] D. E. Perry, A. A. Porter, L. G. Votta, and M. M. Wade. Evaluating workflow and process automation in wide-area software development. In *Software Process Technology, Fifth European Workshop – EWSPT'96*. Springer Verlag, October 1996.

[7] D. E. Perry, N. Staudenmayer, and L. G. Votta. People, organizations, and process improvement. *IEEE Software*, pages 36–45, July 1994.

[8] D. E. Perry, N. A. Staudenmayer, and L. G. Votta. Understanding and improving time usage in software development. In A. Wolf and A. Fuggetta, editors, *Software Process*, volume 5 of *Trends in Software: Software Process*. John Wiley & Sons., 1995.

[9] A. Porter, L. G. Votta, and V. Basili. Comparing detection methods for software requirement inspections: A replicated experiment. *IEEE Transactions on Software Engineering*, 21(6):563–575, June 1995.

[10] H. P. Siy. *Identifying the Mechanisms Driving Code Inspection Costs and Benefits*. PhD thesis, University of Maryland, College Park, MD, June 1996.

[11] L. G. Votta. Does every inspection need a meeting? In *ACM SIGSOFT Software Engineering Notes*, volume 18, pages 107–114, December 1993.

[12] L. G. Votta and M. L. Zajac. Design process improvement case study using process waiver data. In *Proceedings of the Fifth European Conference in Software Engineering*, volume 989 of *Lecture Notes in Computer Science*, pages 44–58. Springer-Verlag, September 1995.