Presented at the Pacific Northwest Software Quality Conference, Portland, Oregon. October 1993. Introductory material in this paper appeared previously in "Improving Software Quality through Computer Supported Collaborative Review", in the Proceedings of the Third European Conference on Computer Supported Cooperative Work, Milan, Italy, 1993.

Experiences with CSRS: An Instrumented Software Review Environment

Philip M. Johnson
Danu Tjahjono
Dadong Wan
Robert S. Brewer

Department of Information and Computer Sciences
University of Hawaii
2565 The Mall
Honolulu, HI 96822 U.S.A.

Abstract

Formal technical review (FTR) is a cornerstone of software quality assurance. However, the labor-intensive and manual nature of review, along with basic unresolved questions about its process and products, means that review is typically under-utilized or inefficiently applied within the software development process. This paper discusses our initial experiments using CSRS, an instrumented, computer-supported cooperative work environment for software review that reduces the manual, labor-intensive nature of review activities and supports quantitative study of the process and products of review. Our results indicate that CSRS increases both the breadth and depth of information captured per person-hour of review time, and that its design captures interesting measures of review process, products, and effort.

Biographical Information

Philip Johnson is an Assistant Professor of Computer and Information Sciences at the University of Hawaii. Dr. Johnson is Director of the Collaborative Software Development Laboratory (CSDL), which performs research on tools and techniques for group-based software engineering, software quality assurance, and other collaborative activities. Danu Tjahjono and Dadong Wan are members of CSDL and doctoral candidates in the Communication and Information Sciences program at the University of Hawaii. Robert Brewer is a member of CSDL and a graduate student in the Computer and Information Sciences Department at the University of Hawaii. Address e-mail correspondence to `Johnson@Hawaii.Edu.`

1. Introduction

Formal technical review (FTR) is a cornerstone of software quality assurance. While other techniques, such as software testing can help assess or improve quality, they cannot supplant the benefits achievable from well-executed FTR. One reason why review is essentially irreplacable is because it can be carried out early in the development process, well before formal artifacts such as source code are available for complexity analysis or testing. A more important reason is because no automated process can yet provide the two-way quality improvement in both product and producers possible through review.

However, the full potential of review is rarely realized in its current forms. Three significant roadblocks to fully effective review are the following:

Review is extremely labor-intensive. Typical procedures for FTR involve individual study of hard-copy designs or source listings and the creation of hand-generated annotations, followed by a group meeting where the documents are paraphrased line by line, issues are individually raised, discussed, and recorded by hand, leading eventually to rework assignments and resulting changes. For one approach to FTR called *code inspection* (Fagan, 1976), published data indicates that an entire person-year of effort is needed to review a 20 KLOC program by a team of four reviewers (Russel, 1991). Unfortunately, little automated support for the process and products of review is available. What support is available typically involves only a single facet of review (such as the review meeting), or is not integrated with the overall development environment.

Review is not compatible with incremental development methods. Because of their labor-intensive nature, most organizations cannot afford to review most or even many of the artifacts produced during development. Instead, review is deployed as a "hurdle" to be jumped a small number of times at strategic points during development. While this may be a reasonable tactic for development in accordance with a strict waterfall lifecycle model, more modern incremental and maintenance-intensive development methods prove problematic: there is no effective way to optimally position a small number of review hurdles in the development process.

No methods or tools exist to support the design of prescriptive review methods adapted to an organization's own culture, application area, and quality requirements. Research on review tends to fall into two categories, which we will term "descriptive" and "prescriptive". The descriptive literature describes the process and products of review abstractly, advocates that organizations must create their own individualized form of review, but provides little prescriptive support for this process (Schulmeyer, 1987; Dunn, 1990; Freedman, 1990). Such work leaves ill-defined many central questions concerning review, such as: How much should be reviewed at one time? What issues should be raised during review, and are standard issue lists effective? What is the relationship between time spent in various review activities and its productivity? How many people should be involved in a review? What artifacts should be produced and consumed during a review? The prescriptive literature, on the other hand, takes a relatively hard line stance on both the process and products of review (Fagan, 1976; Fagan, 1986; Russel, 1991). Such literature makes clear statements about the process (Meetings must last a maximum of 2 hours; each line of code must be paraphrased; lines of code must be read at rate of 150 lines per hour; etc). The data presented in this literature certainly supports the claim that this method, if followed precisely, can discover errors. However, the strict prescriptions appear to suggest that organizations must adapt to the review method, rather than that the review method adapt to the needs and characteristics of the organization.

This paper describes our initial experiences using CSRS[1], a computer-supported cooperative work system that is designed to address aspects of each of these three roadblocks to effective formal technical review.

First, CSRS is implemented on top of EGRET, a multi-user, distributed, hypertext-based collaborative environment (Johnson, 1992) that provides computational support for the process and products of review and inspection. EGRET runs in a Unix/X window environment, providing a client-server architecture with a custom hypertext database server back-end (Wiil, 1990) connected over an ethernet network to front-end EMACS-based clients. This platform allows an essentially "paperless" approach to review, supports important computational services, and facilitates integration with existing development environments. Our initial experiments indicate that CSRS effectively eliminates many of the manual, clerical tasks associated with traditional review. In addition, our experiments show that CSRS captures significantly more information during review than traditional, manual approaches.

Second, CSRS is designed to interoperate with an incremental model of software development. While simply lowering the cost helps integrate review into incremental models, CSRS additionally provides an intrinsically cyclical process model that parallels the iterative nature of incremental development. Our initial experiments also indicate that CSRS supports interoperation by capturing significant amounts of information relevent to other development phases, such as design rationale information.

Third, CSRS exploits the use of an on-line, collaborative environment for review with instrumentation designed to collect a wide range of metrics on the process and products of review. Such metrics provide historical data about review process and products for a given organization, application, and review group that can provide quantitative answers to many of the questions concerning the basic parameters for review raised above. Our initial experiments indicate that CSRS can provide novel data to organizations at a far smaller grain-size than traditional measurement approaches.

CSRS represents two major paradigm shifts with respect to traditional review methods. First, CSRS changes review from an essentially "off-line" process to an essentially "on-

[1]An acronym for Collaborative Software Review System.

line" process[2]. Second, CSRS instrumentation collects measurements at a very small grain-size. (For example, CSRS automatically collects data such as the minutes spent per reviewer on individual functions, as opposed to traditional approaches where reviewers manually record the total elapsed time spent doing preparation, participating in the meeting, and so forth.) Systems involving paradigm shifts are inherently exploratory in nature, requiring an iterative process of design and evaluation to expose and resolve the issues that arise when making paradigmatic change. This paper describes the successes and failures of our experiences with CSRS to date, with the goal of facilitating other research in the design of review support systems.

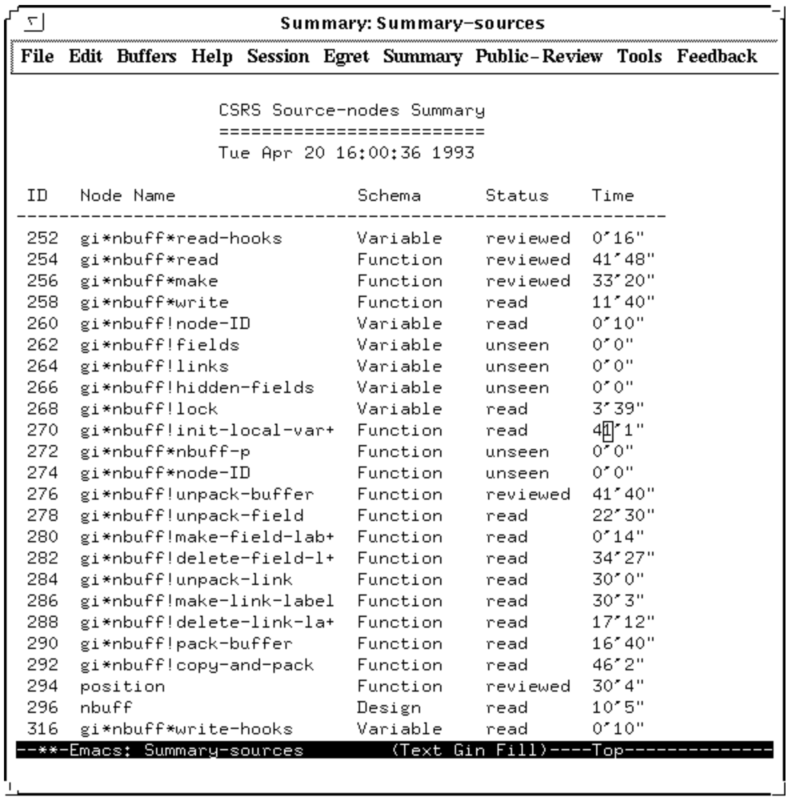
To orient the reader to on-line review, the next section provides a brief introduction to CSRS from a user-level perspective. The following sections discuss the design evolution of CSRS over the past year, detailing our experimental usage of CSRS, the implications of these experiences, and the current status of the system.

2. A review scenario using CSRS

To get the flavor of FTR using CSRS, this section presents excerpts from a recent review. This review cycle involved an object-oriented class implementation called "Nbuff" (short for node-buffer) in the generic-interface subsystem of EGRET. Nbuff defines an abstraction that bridges and combines the hypertext "node" abstraction provided by lower-level subsystems in EGRET and the textual "buffer" abstraction provided by EMACS and higher, application-specific subystems such as those comprising CSRS.

In CSRS, each program object, such as a function, procedure, macro, variable or data type declaration is retrieved from a source code control system and placed into its own node in a hypertext-style database. After an *orientation* session to familiarize each review participant with the system under study, a *private review* phase begins. During private review, each member individually reviews the source code without access to the review commentary of others, although non-evaluative questions and answers about requirements and so forth are publically accessable. CSRS provides facilities to summarize the state of review for the reviewer, such as the window displayed in Figure 1. At this point, the reviewer has partially completed private review, as indicated by the fact that some of the source-nodes are reviewed, some have been read but have not been completely reviewed, and some have not yet even been seen.

[2]Other researchers currently investigating this paradigm shift are (Brothers, 1990), and (Gintell, 1993).


Summary: Summary–sources
File Edit Buffers Help Session Egret Summary Public–Review Tools Feedback

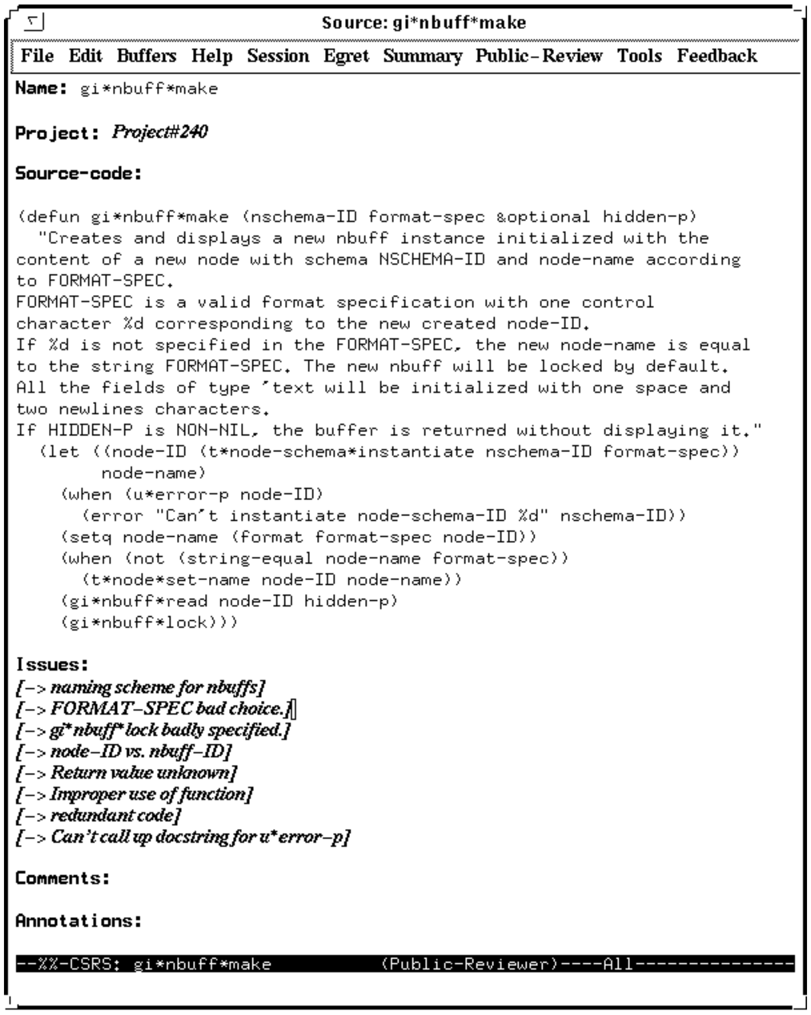
```
                    CSRS Source-nodes Summary
                    =========================
                    Tue Apr 20 16:00:36 1993

 ID   Node Name                 Schema      Status    Time
-------------------------------------------------------------
 252  gi*nbuff*read-hooks       Variable    reviewed  0'16"
 254  gi*nbuff*read             Function    reviewed  41'48"
 256  gi*nbuff*make             Function    reviewed  33'20"
 258  gi*nbuff*write            Function    read      11'40"
 260  gi*nbuff!node-ID          Variable    read      0'10"
 262  gi*nbuff!fields           Variable    unseen    0'0"
 264  gi*nbuff!links            Variable    unseen    0'0"
 266  gi*nbuff!hidden-fields    Variable    unseen    0'0"
 268  gi*nbuff!lock             Variable    read      3'39"
 270  gi*nbuff!init-local-var+  Function    read      41'1"
 272  gi*nbuff*nbuff-p          Function    unseen    0'0"
 274  gi*nbuff*node-ID          Function    unseen    0'0"
 276  gi*nbuff!unpack-buffer    Function    reviewed  41'40"
 278  gi*nbuff!unpack-field     Function    read      22'30"
 280  gi*nbuff!make-field-lab+  Function    read      0'14"
 282  gi*nbuff!delete-field-l+  Function    read      34'27"
 284  gi*nbuff!unpack-link      Function    read      30'0"
 286  gi*nbuff!make-link-label  Function    read      30'3"
 288  gi*nbuff!delete-link-la+  Function    read      17'12"
 290  gi*nbuff!pack-buffer      Function    read      16'40"
 292  gi*nbuff!copy-and-pack    Function    read      46'2"
 294  position                  Function    reviewed  30'4"
 296  nbuff                     Design      read      10'5"
 316  gi*nbuff*write-hooks      Variable    read      0'10"
--**-Emacs: Summary-sources        (Text Gin Fill)----Top--------------
```

Figure 1. A summary window illustrating the state of review for one reviewer.

By mouse-clicking on a line or through menu operations, the reviewer can traverse the hypertext network from this screen to a node containing a source object under review, as illustrated in Figure 2. In this case, the object under review is the operation gi*nbuff*make. Both pull-down and pop-up menus facilitate execution of the most common operations during this phase, such as creating an issue concerning the current source node under review (as illustrated in Figure 3), or proposing a specific action to address an issue. Once the reviewer is finished with a source object, he explicitly marks the node as "reviewed". Since this is the private review phase, only the issues created by this reviewer are accessable.

CSRS assumes that typical programming environment tools are available to the reviewer, such as static cross-referencing and dynamic behavior information, and thus does not attempt to duplicate that functionality. Part of the benefit of an EMACS-based platform is ease of integration with external programming environment tools (for example, EMACS interfaces are provided in the C/C++ environments XOBJECTCENTER and ENERGIZE, as well as in Common Lisp environments by Lucid and Franz.)

Once the source nodes have been privately reviewed, the *public review* phase begins, where reviewers now read and react to the issues and actions raised by others. Each


Source: gi*nbuff*make
File Edit Buffers Help Session Egret Summary Public-Review Tools Feedback

Name: gi*nbuff*make

Project: *Project#240*

Source-code:

```
(defun gi*nbuff*make (nschema-ID format-spec &optional hidden-p)
  "Creates and displays a new nbuff instance initialized with the
content of a new node with schema NSCHEMA-ID and node-name according
to FORMAT-SPEC.
FORMAT-SPEC is a valid format specification with one control
character %d corresponding to the new created node-ID.
If %d is not specified in the FORMAT-SPEC, the new node-name is equal
to the string FORMAT-SPEC. The new nbuff will be locked by default.
All the fields of type 'text will be initialized with one space and
two newlines characters.
If HIDDEN-P is NON-NIL, the buffer is returned without displaying it."
  (let ((node-ID (t*node-schema*instantiate nschema-ID format-spec))
        node-name)
    (when (u*error-p node-ID)
      (error "Can't instantiate node-schema-ID %d" nschema-ID))
    (setq node-name (format format-spec node-ID))
    (when (not (string-equal node-name format-spec))
      (t*node*set-name node-ID node-name))
    (gi*nbuff*read node-ID hidden-p)
    (gi*nbuff*lock)))
```

Issues:
[-> naming scheme for nbuffs]
[-> FORMAT-SPEC bad choice.]
*[-> gi*nbuff*lock badly specified.]*
[-> node-ID vs. nbuff-ID]
[-> Return value unknown]
[-> Improper use of function]
[-> redundant code]
*[-> Can't call up docstring for u*error-p]*

Comments:

Annotations:

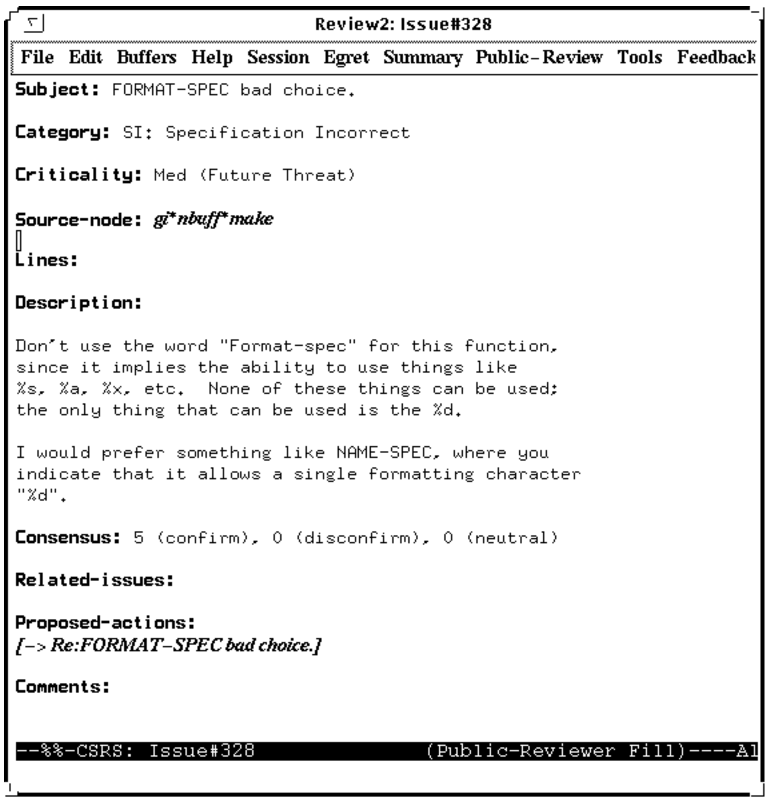
```
--%%-CSRS: gi*nbuff*make          (Public-Reviewer)----All---------------
```

Figure 2. A source node illustrating one of the functions under review.

reviewer responds to the issues and actions raised by others through the creation of new issues or actions, creation of confirming or disconfirming evidence nodes to extant issues or actions, and by voting for one or more actions to be taken during the rework phase.

Following public review, the moderator uses CSRS to *consolidate* the review state. Consolidation involves the restructuring of information captured during review into a written report that delineates the proposed actions, agreements, and unresolved issues resulting from the private and public review phases. CSRS provides automated support to the moderator in traversing the hypertext database and generating a LaTeX document containing the consolidated report.

If all issues arising from the on-line phases have been satisfactorily resolved, then this consolidation report is also the final review report that both specifies the issues raised and the rework required. Consolidation reports are far more comprehensive and detailed than typical review reports from traditional review methods, such as those described in


Review2: Issue#328
File Edit Buffers Help Session Egret Summary Public-Review Tools Feedback

```
Subject: FORMAT-SPEC bad choice.

Category: SI: Specification Incorrect

Criticality: Med (Future Threat)

Source-node: gi*nbuff*make

Lines:

Description:

Don't use the word "Format-spec" for this function,
since it implies the ability to use things like
%s, %a, %x, etc.  None of these things can be used;
the only thing that can be used is the %d.

I would prefer something like NAME-SPEC, where you
indicate that it allows a single formatting character
"%d".

Consensus: 5 (confirm), 0 (disconfirm), 0 (neutral)

Related-issues:

Proposed-actions:
[-> Re:FORMAT-SPEC bad choice.]

Comments:

--%%-CSRS: Issue#328            (Public-Reviewer Fill)----Al
```

Figure 3. An issue node containing an objection to an aspect of gi*nbuff*make.

(Pressman, 1992; Freedman and Weinberg, 1990; Humphrey, 1989). If the consolidation report reveals issues unresolved during public review, then the moderator schedules a face-to-face meeting to resolve these issues, or else decides them unilaterally. (CSRS is not currently used in face-to-face review meetings. Lack of automated support for this phase of CSRS-style review has not been a problem in practice, since such meetings are typically very short or avoided altogether.)

(Johnson and Tjahjono, 1993) provides a detailed description of the data and process model used in CSRS. The next section reports on our experiences in the use of CSRS, and provides rationales and results from our design decisions.

3. Initial experiences with CSRS

Our research project on computer-supported FTR will be two years old at the time of publication, and we will have been performing review experiments with running versions of CSRS for over one year of that period. CSRS has evolved and matured significantly during this time. To most clearly present the successes and failures of CSRS to date, we will present the major releases of CSRS, interesting facets of their design rationale, and our experimental data in chronological order. As the next sections will reveal, we first focussed on the design of a data and process model appropriate for on-line review, then incrementally explored the design space for automated review instrumentation.

3.1 CSRS I: Process/data model design

In the first release of CSRS, we concentrated upon the impact that specializing our computer-supported cooperative work environment, EGRET (Johnson, 1992), to FTR would have upon the process and products of review. It became clear that the "classic" FTR method–Fagan-style code inspection (Fagan, 1976; Fagan, 1986)–cannot be straightforwardly applied to a collaborative work environment.

First, the manual nature of Fagan-style inspection means that certain concepts, such as the role of scribe, do not make sense in an automated environment where each participant's actions are captured automatically.

Second, since EGRET is oriented toward asynchronous communication, our support for review is similarly biased. However, the primary focus of Fagan-style inspection is on the nature of the synchronous, face-to-face meeting and its attendent issues. A change in process orientation from synchronous to asynchronous has profound implications. First, it leads to extensive change in the role of moderator. While the primary responsibilities of a Fagan-style moderator is to ensure reviewer preparedness and maintain order and effectiveness during a face-to-face meeting, a CSRS moderator's task involves two completely different issues: creating a well-structured hypertext database of source artifacts, and maintaining order and effectiveness while participants asynchronously read source code and other postings and reply to them on-line.

The change from synchronous to asynchronous interaction also allows change in the scope and content of review. One of the fundamental guidelines for effective synchronous review is "raise issues, don't resolve them." In other words, in a face-to-face meeting, it is important to keep focussed directly on the generation and recording of issues, and to avoid discussion of resolutions. Such a focus is needed because review meetings may cost 4-6 person-hours of skilled technical staff per hour of elapsed time, and resolution conversations are frequently not only time consuming, but may involve only a small subset of those attending.

Avoiding resolution-oriented discussion significantly improves the group process by preventing conversational digression and by improving the usage of human resources.

In an asynchronous environment, however, this rationale for restricting the scope of review no longer applies. Since participants are working asynchronously, time spent by one participant generating a potential resolution to an issue does not incur cost to others. During public review, only those participants qualified to evaluate an issue or action proposal need perform detailed analysis—others can simply indicate their neutrality.

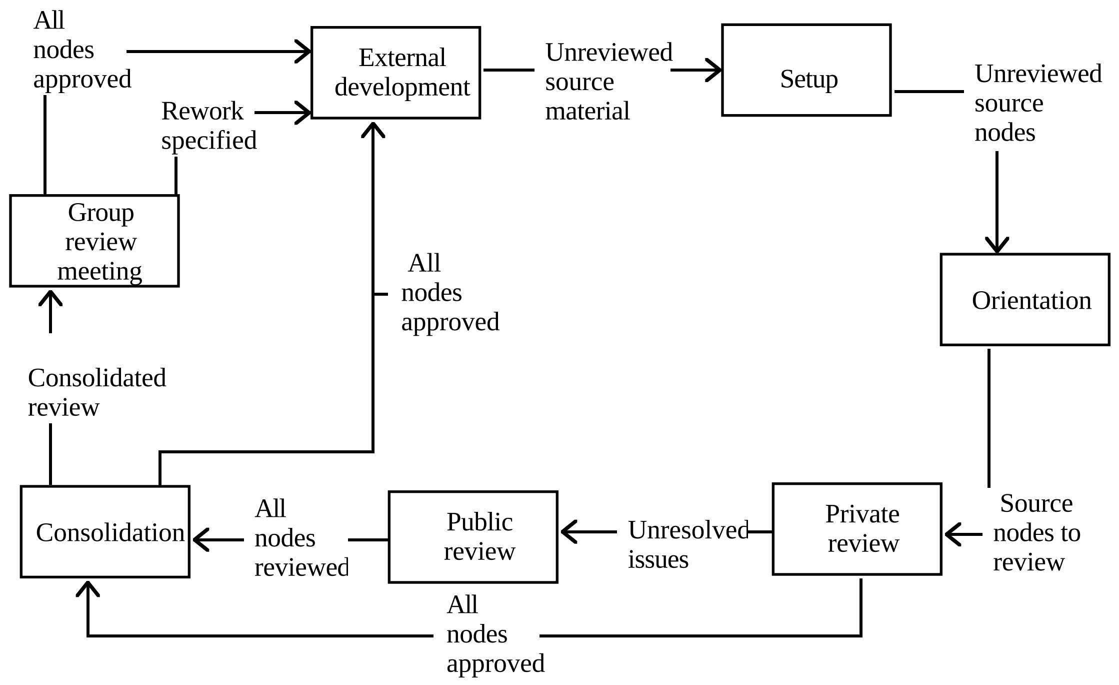


Figure 4. The CSRS process model. During the Setup phase, the Moderator and Producer use tools to build a hypertext database of source material. During Orientation, the producer provides the Reviewers with a high-level overview of the review materials. During Private Review, reviewers analyze source material and generate issue and action proposals. During Public Review, reviewers analyze the issues and actions generated by other reviewers, and attempt to build consensus. During Consolidation, the Moderator analyzes the current state of review. If all issues have been resolved in a consensual manner, then the review session terminates and a final report is generated. If controversies remain, a Group Review Meeting is called to resolve the issues in a face-to-face manner, followed by termination of review and generation of the final report.

Moreover, allowing review activities to include resolution discussion has significant advantages. In many cases, exploring resolutions provides useful additional insight into the nature of the issue and its interdependencies with other issues. Resolution discussion is also a "natural" part of review: as any attendee to a code inspection meeting knows, it takes conscious effort *not* to propose solutions to problems. Finally, resolution discussion during review can be more efficient: those people most qualified to review a resolution action are frequently part of the review, and incorporating resolution reduces or eliminates the additional meetings and scheduling typically required after a Fagan-style inspection.

For these reasons, we decided in the design of the asynchronous review environment to explicitly encourage the generation of resolutions to raised issues from participants, viewing this as an opportunity to exploit the power of group work that is unfortunately but necessarily lost from traditional synchronous meetings.

As a result of this research, we designed a data and process model for review that we have continued to use, with only minor changes, in all subsequent versions of CSRS to date. The process model consists of seven basic phases, as illustrated in Figure 4 and outlined in Section 2. The process model is coupled with a data model that describes the set of node and link types that can be defined and the legal relationships between them. A detailed description of this representation appears in (Johnson and Tjahjono, 1993).

3.2 CSRS I: Data-oriented instrumentation

From the start, we viewed measurement and instrumentation as a fundamental part of the design of CSRS. In the first version, we designed measurement from a "data-oriented" perspective: the measures were generated by querying the hypertext database at the conclusion of review for the number of nodes of a given type, or for those containing a specific value for a given field, or for those partaking in a specific relationship to other nodes.

Data-oriented measures are very useful: at a minimum, the number and severity of identified defects provides a first-order estimate of the quality of the software under review. Data-oriented measures can also reveal important characteristics of the review team and review process. For example, unproductive members of the review team might be identified (after a sufficient number of review instances) as those who contribute little, who contribute non-productively (by simply affirming comments made by others), or who use review for political purposes.

Finally, data-oriented measures can form the basis for controlled experimentation: given sufficient time and resources, an organization can fine-tune certain characteristics of review, such as the number of participants or the number of lines of code under review, by collecting data across a range of review instances and correlating these factors to, for example, the number of productive issues raised.

We experimented on the initial version of CSRS in the summer of 1992 by performing a review of a module called "Gtable" from the EGRET implementation. Gtable allows its clients to define a high-level abstraction for associating keys and values in a distributed environment, without concerning themselves with the details of replicating the tables to local hosts, maintaining consistency as updates are made by clients, and providing recovery procedures. The top-level of the Gtable implementation under review consisted of approximately 500 lines of Lisp macros and functions.

In the Gtable review, which lasted approximately three weeks, over 90 nodes were created by five reviewers during the private and public review phases. Following these phases, the moderator assessed the state of review to determine if a group meeting was necessary. It was found that a group meeting was required to resolve approximately only half of the 25 primary issues raised about the Gtable implementation.

This usage of the initial CSRS system convinced us that a computer-supported, asynchronous approach to review was legitimate and useful, and that the EGRET environment was an effective platform for this application. This usage also revealed certain engineering issues that required immediate attention. During the fall of 1992, we devoted considerable effort to improving the user interface, the efficiency of both CSRS and EGRET, and the set of services provided (for example, facilities to automatically generate LaTeX hard-copy reports)

3.3 CSRS II: Elapsed-time instrumentation

A significant result from the first experiment was the insight that CSRS could be further instrumented to capture time-related data. By recording the time that a node is retrieved from the database by the client and the time that the node is closed, we expected to determine how long the participants spent reviewing each function, generating each issue, and so forth.

Capturing such time-related data could greatly enhance the analysis potential of CSRS. With this data, it would be possible to perform "fine-grained" analysis of review. For example, one could study the relationship between code size and time required for review at the grain-size of individual functions.

To explore this functionality, we enhanced CSRS with a counter for each node that accumulated the elapsed time that the node was retrieved. We then performed a review experiment on the "Nbuff" module mentioned above. The review involved 18 source code nodes, totalling 435 lines of code. Like the Gtable review, this experimental usage involved five participants over approximately three weeks. It generated 104 nodes that were eventually consolidated down to 19 actions. Six of these were controversial, requiring only a 35 minute meeting to resolve before beginning rework.

After review, we examined the timing data, and discovered to our surprise that it did not in all cases reflect the time spent in review. The essential problem is that recording the elapsed time that a node is retrieved by a client is only an indirect measure of the time spent reviewing.

Recall that CSRS is implemented using Unix and X windows, and consider, for example, the following scenarios:

- The participant takes a phone call while reviewing a node, or a colleague walks into the office and begins a lengthy discussion.
- The participant receives e-mail while reviewing a node, and switches to a simultaneously running mailer process to read and reply to it.
- The participant goes home for the weekend, leaving in the middle of reviewing a node with the intent to finish the review first thing on Monday. Since the participant has a dedicated workstation in his office, he leaves CSRS up and running.

In fact, the incorrect elapsed time values in Figure 5 for reviewer 1 on nodes gi*nbuff*read and gi*nbuff*make are due to the first and second scenarios.

These results led us back to the proverbial drawing board. We proposed and immediately discarded the idea of telling reviewers that they could not read their mail or

Source code name	Size	Reviewer 1		Reviewer 2		Reviewer 3		Reviewer 4	
		Time	Iss	Time	Iss	Time	Iss	Time	Iss
gi*nbuff*read	25	2:28:47	3	0:49:08	2	0:28:39	0	0:40:43	1
gi*nbuff!pack-buffer	57	0:08:59	0	0:41:33	2		0	0:08:35	1
gi*nbuff!copy-and-pack	51	0:24:18	1	0:29:07	3		0	0:02:37	0
gi*nbuff!init-local-variables	19	0:09:06	1	0:16:57	1	0:02:49	0	0:24:00	0
gi*nbuff!unpack-field	46	0:10:19	1	0:32:28	0		0	0:08:22	0
gi*nbuff*make	20	2:04:44	5	0:36:18	2	0:14:50	0	0:13:38	1
gi*nbuff!make-link-label	34	0:01:39	0	0:40:45	1	0:00:22	0	0:04:15	0
gi*nbuff!unpack-buffer	26	1:46:29	3	0:03:42	0	0:04:22	0	0:17:22	2
gi*nbuff!make-field-label	18	0:00:42	0	0:25:39	0		0	0:03:06	0
gi*nbuff*write	33	0:06:50	1	0:09:54	2	0:07:48	0	0:14:38	0
gi*nbuff!delete-field-label	18	0:00:34	0	0:25:56	1	0:03:44	1	0:02:38	0
gi*nbuff!unpack-link	12	0:20:09	1	0:00:40	0	0:01:48	0	0:06:09	1
gi*nbuff*nbuff-p	12	0:05:11	0	0:01:08	0	0:04:16	0	0:04:15	0
gi*nbuff*node-ID	12	0:00:23	0	0:02:20	0	0:04:40	0	0:08:12	0
gi*nbuff!delete-link-label	31	0:02:08	1	0:02:08	0		0	0:02:38	0
gi*nbuff*read-hooks	5	0:00:11	0	0:07:53	1	0:00:32	0	0:00:35	0
position	12	0:00:50	0	0:00:12	0	0:03:12	1		
gi*nbuff!node-ID	4	0:00:20	0	0:00:08	0	0:00:49	1	0:00:13	0
Total	**435**	**7:51:39**	**17**	**5:25:56**	**15**	**1:17:51**	**3**	**2:41:56**	**6**

Figure 5. Elapsed time data generated from the Nbuff review.

otherwise interrupt a CSRS review session; such a restriction would be impossible to enforce and the resulting data would be no less suspect.

3.4 CSRS III: Improved elapsed-time instrumentation

In the next version of CSRS, we redesigned the time-based instrumentation to address this issue. Instead of simply maintaining a counter with each node that would accumulate a single elapsed time value, we implemented a more sophisticated and general purpose event-based timestamp facility within EGRET. This facility allows EGRET-based applications such as CSRS to insert timestamp calls at strategic points within their code to record the occurrence of arbitrary events of interest. (For example, when the user opens a node, writes a node, traverses a link, and so forth.) EGRET provides the underlying mechanisms to fast-cache the sequence of timestamped events at the local client during the session, and write the cache out to the server database at disconnect time. Timestamping inevitably incurs some overhead, and it is possible that over-zealous insertion of timestamp calls by an application can visibly degrade the responsiveness of the system. (We have not, however, observed degradation in performance due to timestamping in CSRS or two other applications developed using EGRET.)

Using this redesigned instrumentation, we performed a review experiment on a prototype newsreader system called URN. The review involved 53 source code nodes, totalling 478 lines of code. This review lasted 10 days, generated 75 reviewer-based nodes, and 35 issues. We timestamped database connection and disconnection, node

ID	Operation	Node	Name	Date	Time	Event Interval	Screen	Misc
60	connect		csrs	20-May-93	10:09:48			Private
60	summarize-sources		Summary-sources	20-May-93	10:10:13	0:00:25	Summary	
60	read-node	350	uin*key!close-article	20-May-93	10:10:20	0:00:07	Source	
60	close-node	350	uin*key!close-article	20-May-93	10:30:49	0:20:29	Source	
60	close-node	362	overview	20-May-93	10:30:50	0:00:01		
60	disconnect		csrs	20-May-93	10:30:50	0:00:00		Private
60	summarize-sources		Summary-sources	20-May-93	10:36:21		Summary	
60	connect		csrs	20-May-93	10:36:21	0:00:00		Private
60	read-node	350	uin*key!close-article	20-May-93	10:36:30	0:00:09	Source	
60	close-node	350	uin*key!close-article	20-May-93	10:38:59	0:02:29	Source	
60	summarize-sources		Summary-sources	20-May-93	10:39:02	0:00:03	Summary	
60	read-node	260	uts*article*make	20-May-93	10:39:19	0:00:17	Source	
60	summarize-sources		Summary-sources	20-May-93	10:39:39	0:00:20	Summary	

Figure 6. A portion of the event log generated during the URN review. Over 1000 events were logged during the URN review.

reading, writing, creation, deletion, and setting the status field in source code nodes (the status field is used by reviewers to explicitly signal when finished reviewing a node).

Figure 6 illustrates a portion of the time-stamped event log for the URN review experiment. Participants did not report any noticable degradation in the responsiveness of the system from the timestamping mechanism.

The timestamp mechanism significantly improves the CSRS instrumentation. First, timestamps can recreate the elapsed time data we obtained through the previous mechanism.

More importantly, timestamps allow us to assess the accuracy of the elapsed-time information, by helping detect the occurrence of review interruptions due to scenarios like those noted above. This is accomplished by calculating the inter-event intervals. The reasoning goes as follows. If CSRS is instrumented "correctly", then, under typical usage patterns, events should be generated relatively frequently and consistently. A significant interruption in review, due to answering the phone, leaving the office, and so forth can be detected by an abnormally large value for an inter-event interval. However, if CSRS is instrumented "incorrectly" (i.e. the timestamp calls are too sparsely distributed in the application code), then not enough timestamps will be generated to distinguish such interruptions from normal patterns of timestamp generation during review.

Figure 7 shows a histogram of inter-event intervals collected during one phase of the URN review. This data shows that timestamps were generated less than 30 seconds apart over 80% of the time, less than a minute apart over 90% of the time, and less than three minutes apart over 98% of the time.

It is important to be precise about what can and cannot be inferred from inter-event interval data. While a sequence of small inter-event interval values (say, less than or equal to 10 seconds) does effectively indicate essentially uninterrupted use of CSRS, the

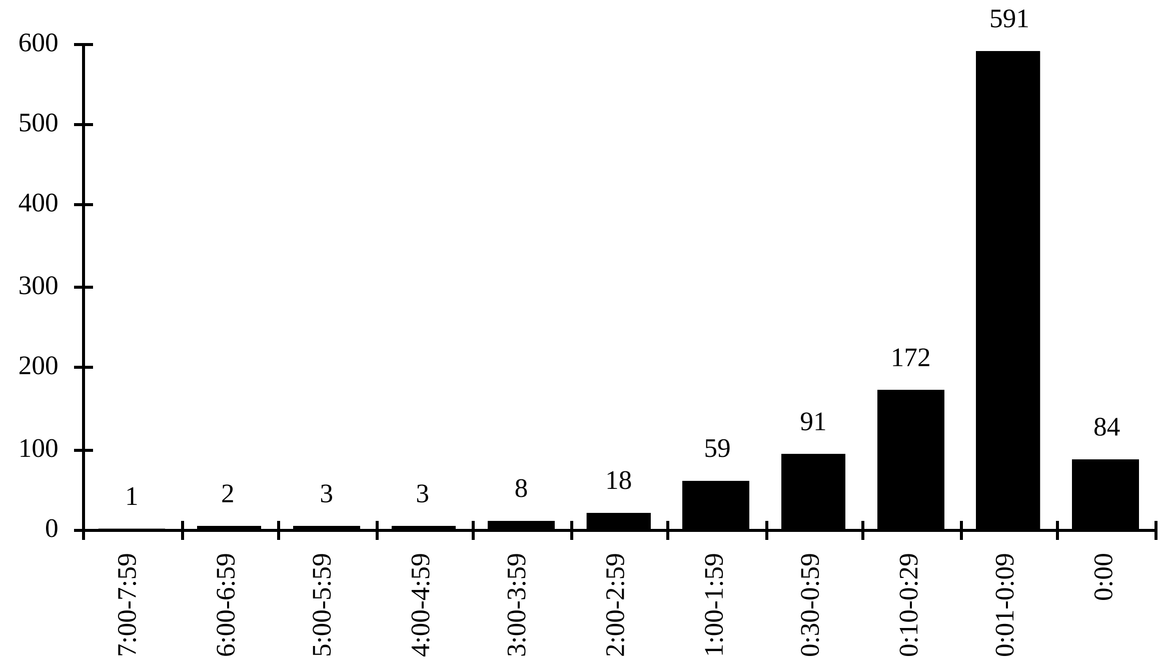


Figure 7. A histogram illustrating the frequency of inter-event intervals in the URN private review phase. For presentation purposes, a single interval with value 20:29 has been omitted from this figure.

converse is not true: a high inter-event interval value does not necessarily indicate the occurrence of interruption.

It is possible, for example, that a reviewer could simply stare at a section of source code for many minutes without performing any CSRS-related action that would trigger an event, although this seems somewhat improbable. A more likely scenario is one in which a reviewer spends a lengthy period of time carefully composing an issue in a CSRS editor buffer without retrieving, saving, traversing, or otherwise interacting with CSRS. With the timestamp calls used in this experiment, the event interval data would be identical to an alternative scenario in which the reviewer spends a couple of minutes composing the issue, leaves for lunch, and then returns and saves the issue to the database.

While the event interval data might be identical in these two cases, one would hope that the two issues would differ qualitatively. Thus, an appropriate way of regarding event interval data is as a means to pinpoint areas of ambiguity in the timestamp data: places where further study is needed to assess the accuracy of the information.

As a concrete example, in the URN review, the highest recorded interval was one of 20 minutes and 29 seconds. Figure 6 illustrates a portion of the event interval log containing this value, which occurred between 10:10am and 10:30am on May 20, 1993. Notice the context in which this event interval occurred: the reviewer connected to the system at 10:09 a.m., retrieved the node containing the source code uin*key!close-article at 10:10 a.m., and then closed the node and immediately disconnected at 10:30 a.m. Given that this particular function is only five lines long and reasonably straightforward, and that no other activity occurred during this entire session except to retrieve and close this node, the interval value is suspect. An interview with the reviewer confirmed that this particular elapsed-time value for uin*key!close-article could be thrown out, since the reviewer was indeed interrupted during this time period.

In a very recent version of the system, we incorporated a timer-based mechanism that wakes up once-per-minute and checks to see if any low-level editor activity (such as a keystroke or mouse click) has occurred in CSRS during the preceeding two minutes. If such activity occurred, it writes out a generic "busy" timestamp event. This mechanism may have substantially solved the problem of detecting idle time, since we believe it to be extremely unlikely that a reviewer will simply stare at a CSRS screen for over two minutes without so much as scrolling a window.

3.5 CSRS III: Process-oriented instrumentation

Moving to an event log-based instrumentation mechanism has an even more profound impact than improving the accuracy of the elapsed-time data: it instruments the *process* of review at a fine-grained level. By analysis of the event log, it is possible to reconstruct the sequence by which a reviewer traversed the hypertext database, even reconstructing the set of nodes displayed concurrently in different windows at any point in time.

Such forms of analysis carry with them new and significant quality assessment issues. For example, to reconstruct this process-level information accurately, users much not change the default screen orientation manually during review.

However, we believe this process-level analysis of CSRS review has significant potential to aid in process maturation of FTR, as developed in the SEI Capability Maturity Model (Humphrey, 1989). By analyzing the sequence of actions taken by reviewers as they perform FTR, we expect to improve the user interface to CSRS and the set of services provided. More fundamentally, we believe that high-quality, fine-grained, process-oriented measures, when combined with high-quality, fine-grained data-oriented measures, will provide new and efficient support to organizations in designing and improving FTR methods custom-suited to their organizational and application-level needs. This will bring our research full-circle, as we use the instrumentation data to return to the initial focus of our design: a data and process model more efficiently suited to on-line, collaborative review.

4. Current status

We are currently using CSRS in-house on a nearly continuous basis, not simply to further refine the paradigm, but because we are firmly convinced that it provides the most cost-effective way of improving the software quality of CSRS itself and the other applications under development in our research group.

While we have not yet collected enough quantitative data under controlled conditions to present statistically significant conclusions concerning the process and products of FTR using CSRS, our data collected to date does support some general observations.

First, the data indicates that CSRS review appears to proceed at approximately the same rate (100-250 lines of code/person-hour) as that reported for Fagan-style code inspection. However, as alluded to above, CSRS review captures the review "discussion" much more completely than can a scribe manually writing notes during a meeting, and also includes discussion of resolutions that are typically excluded from manual review methods. Thus, more information in greater detail is produced in the same amount of time using CSRS.

To provide some perspective of the range of artifacts captured during review, we manually categorized the types of issues and commentatary raised during the URN review, and found that approximately 20% were either: design rationale related; clarified the specifications or behavior of the application under development; or clarified the specifications or behavior of EGRET or some other underlying infrastructure (such as the source language). The presence of such captured artifacts (which would not typically be recorded in a conventional review–indeed, they would be viewed as a "digression") have helped us to improve the specifications and documentation of EGRET and other systems, and provided significant aid to the developers.

The current users of CSRS completed a questionnaire asking subjective questions about their satisfaction with the system. Aside from the generic eternal user plea for faster response times, high satisfaction was indicated. (These responses, however, are from a small and biased population.) Better subjective data will soon be forthcoming, however, as CSRS is scheduled for use in an external software development group during the summer of 1993.

In conclusion, to summarize our lessons learned:

- CSRS formal technical review requires a different data and process model than those designed for manual FTR methods.
- User satisfaction with CSRS appears high.
- CSRS does not appear to appreciably change the rate of review as compared to traditional FTR. However, this rate comparison is misleading, since CSRS review encourages resolution activities that typically take place downstream from traditional FTR.
- CSRS review captures a much broader range of information than application-level defects, the focus of traditional FTR.
- The EGRET timestamp mechanism used in CSRS provides both fine-grained data on the process and products of review, and supports quality assurance activities on elapsed time data. However, this quality assurance currently requires manual post-processing, analysis, and interpretation.

5. Acknowledgments

Support for this research was provided in part by the National Science Foundation Research Initiation Award CCR-9110861 and the University of Hawaii Research Council Seed Money Award R-91-867-F-728-B-270. We thank the anonymous reviewers for many helpful comments that improved the quality of this paper.

6. References

L. Brothers, V. Sembugamoorthy, and M. Muller (1990): ICICLE: Groupware for code inspection. In *Proceedings of the Conference on Computer-Supported Cooperative Work 1990,* pp. 169-181. ACM Press.

Lionel E. Deimel (1990): *Scenes of Software Inspections: Video Dramatizations for the Classroom.* Software Engineering Institute, Carnegie Mellon University.

Robert Dunn (1990): *Software Quality: Concepts and Plans.* Prentice Hall.

Michael E. Fagan (1976): Design and code inspections to reduce errors in program development. *IBM System Journal,* 15(3):182--211.

Michael E. Fagan (1986): Advances in software inspections. *IEEE Transactions on Software Engineering,* SE-12(7), pp. 744-751.

D. P. Freedman and G. M. Weinberg (1990): Handbook of Walkthroughs, Inspections and Technical Reviews. Little, Brown.

John Gintell, John Arnold, Michael Houde, Jacek Kruszelnicki, Roland McKenney, and Gerard Memmi (1993): Scrutiny: A Collaborative Inspection and Review System. In *Fourth European Software Engineering Conference*, Garwisch-Partenkirchen, Germany, September 1993.

Watts S. Humphrey (1989): *Managing the Software Process.* Addison-Wesley.

Philip M. Johnson (1992): Supporting exploratory CSCW with the EGRET framework. In *Proceedings of the Conference on Computer-Supported Cooperative Work 1992,* ACM Press.

Philip M. Johnson and Danu Tjahjono (1993): Improving Software Quality through Computer Supported Collaborative Review. In the *Third European Conference on Computer Supported Cooperative Work,* Milan, Italy, September, 1993.

Roger S. Pressman (1992): *Software Engineering: A Practitioner's Approach.* McGraw-Hill, Inc.

Glen W. Russel (1991): Experience with inspection in ultralarge-scale developments. IEEE Software, (9)1.

G. Gordon Schulmeyer and James I. McManus (1987): Handbook of Software Quality Assurance. Van Nostrand Reinhold.

U. Wiil and K. Osterbye (1990): Experiences with hyperbase-a multi-user back-end for hypertext applications with emphasis on collaboration support. *Technical Report 90-38,* Department of Mathematics and Computer Science, University of Aalborg, Denmark.

Edward Yourdon (1989): *Structured Walkthrough.* Prentice-Hall, Fourth Edition.