A COLLABORATIVE SOFTWARE CODE INSPECTION: THE DESIGN AND EVALUATION OF A REPEATABLE COLLABORATION PROCESS IN THE FIELD

GERT-JAN DE VREEDE*† and PUSHPA G. KONERI†‡

*Faculty of Technology, Policy and Management Delft University of Technology Postbus 5, 2600 AA Delft, The Netherlands

†College of Information Science & Technology University of Nebraska at Omaha 6001 Dodge Street, Omaha, NE 68182, USA *gdevreede@mail.unomaha.edu ‡pkoneri@mail.unomaha.edu

DOUGLAS L. DEAN

Marriott School of Management Brigham Young University, Provo, UT 84602, USA doug dean@byu.edu

ANN L. FRUHLING§ and PETER WOLCOTT¶

College of Information Science & Technology University of Nebraska at Omaha 6001 Dodge Street, Omaha, NE 68182, USA §afruhling@mail.unomaha.edu ¶pwolcott@mail.unomaha.edu

Abstract:

The use of software products in today’s world has increased dramatically making quality an important aspect of software development. There is a continuous need to develop processes to control and increase software quality. Software code inspection is one way to pursue this goal. This paper presents a collaborative code inspection process that was designed during an action research study using Collaboration Engineering principles and techniques. Our inspection process was implemented as a sequence of thinkLets, chunks of facilitation skill, that were subsequently field tested in a traditional paper-based and Group Support System (GSS)-based environment. Four inspections were performed on four different pieces of software code in two different organizations. Results show that regardless of the implementation, the process was found to be successful in uncovering many major, minor, and false-positive defects in inspected pieces of code. Overall observations and feedback suggest that the collaborative inspection process was considered to be productive and satisfactory. GSS inspections were more effective, especially in terms of major defects. GSS inspections were also found to be more efficient. Finally, the GSS inspections outperformed the paper inspections from a practical perspective: logging and managing defects in a GSS was far superior.

Keywords: Collaboration; thinkLets; Collaboration Engineering; Group Support Systems; action research; code inspection; Fagan inspection.
1. Introduction

In software development, up to 50% of all labor-hours are spent on the detection and correction of software defects.\textsuperscript{1,2} A software defect is a “material breach of the contract for sale or license of the software, if it is so serious that a customer can justifiably demand a fix or can cancel the contract, return the software, and demand a refund”.\textsuperscript{3} Software inspections are designed to detect software defects, thereby improving the quality of software products. Inspections can focus on the design documents or the source code itself. Reference 4 defines inspections as a “formal, efficient and economical method of finding errors in design and code”. Well-executed software inspections can result in significant labor savings during the software development process. According to Ref. 5, around 80% of anomalies are removed through inspection; one hour of inspection will uncover as many anomalies as four hours of regression testing. While inspections do not fully replace formal software testing, an hour invested in software code inspection can save up to five hours in later stages.\textsuperscript{6} Moreover, each defect identified during code inspection saves around nine hours of time in later stages of the development process.\textsuperscript{7}

Software can be inspected at different stages of its development.\textsuperscript{8} In the Software requirement specification stage, the specifications requirements document is inspected for its completeness. An inspection plan is developed that includes requirement specification, description of inspection procedures, schedules and milestones. In the Design stage, the adequacy of the software design is determined by checking the consistency of design with the requirements specified. The various design products can be inspected through analysis, simulation, and walkthroughs. In the Software coding stage, the code is checked for its correct syntax, logic and execution. Different aspects of the code such as logic errors, data errors, input/output errors, processing time, security, project programming standards, and conflicts with the specification requirements are also inspected. In addition, inspectors can assess whether the code is written in the most efficient way to accomplish a certain objective and whether there are places where code can be reused. In the Software testing stage, the test cases are inspected for their correct functional objectives. Test cases, expected test results, test procedures, test schedules and test milestones are also inspected for their accuracy. Finally, in the Software Maintenance stage, inspectors check ensure that the software product continues to be in line with the current industry trends and has the latest security updates.

There are a variety of different inspection processes that have evolved over time. One of the most popular is the Fagan inspection,\textsuperscript{9} which is the type of inspection examined in this research. Figure 1 depicts the stages of a Fagan inspection process, which will be discussed in more detail in Sec. 2. Because software
inspections involve multiple people, inspection efficiency depends on proper coordination of the people and activities involved during the inspection process.

Multiple inspectors may inspect the same portion or elements of a software product to obtain more thorough coverage. Inspectors may have to work together to perform a comprehensive inspection if software products are too large to be inspected by a single person. Finally, software inspectors must also coordinate their activities with the developers, who are responsible for correcting defects. In short, software inspection is a collaborative process that has to follow a systematic approach to maximize both inspection efficiency and effectiveness. A well-defined software inspection process lays out the ground rules that allow objective rather than subjective decision-making regarding potential defects. Since many factors may impact the efficiency and effectiveness of a review process, it is important to determine which approaches works best in a given context.

The research reported in this paper focuses specifically on software code inspection. The objective of software code inspection is to identify and remove software defects before the code reaches formal testing. A survey in Ref. 7 showed some striking examples of the value of code inspections:

- IBM managed to remove 82% of all defects before testing even began.
- AT&T found inspections led to 14% increase in productivity and 10-fold increase in quality.
- HP found that 80% of the errors detected during inspections were unlikely to be caught by testing.
• HP, Shell Research, Bell Northern, and AT&T all found inspections 20–30 times more efficient than testing in detecting errors.

In addition, this paper reports on the development and evaluation of a collaborative software code inspection process. The design of the process followed Collaboration Engineering principles to create a repeatable inspection process that inspection teams could execute by themselves.\textsuperscript{13,14} Earlier research suggests that code inspection is an example of a collaborative process that can be facilitated by the participants themselves.\textsuperscript{15} The collaborative code inspection process was applied in four cases in two organizations, with both paper-based and Group Support System (GSS) implementations. This study was exploratory in nature and evaluated the following: actual and perceived inspection effectiveness at finding defects, actual and perceived efficiency in terms of time required to complete the inspection meeting process, and satisfaction with the inspection outcome and process.

The remainder of this paper has the following structure. Section 2 discusses code inspection and collaborative support for code inspections in more detail. Section 3 describes our research approach. The design of the collaborative code inspection process is presented in Sec. 4, along with the criteria to evaluate the process. Section 5 presents the results of the evaluation of the Collaboration Engineering process in the field. Finally, Sec. 6 discusses the findings and implications of our research, summarizes the limitations and suggests future research directions.

2. Background

Code inspection methods can be broadly classified into two categories: Informal and Formal Inspection methods. Informal Inspection methods do not follow any standard procedure. They can take place at regular project status meetings or even over a coffee. They do not follow any specific agenda and there is no documented output. Informal inspections are often criticized for their lack of guidelines.\textsuperscript{16} Formal Inspection methods are conducted in a well-defined systematic and structured manner. These methods usually execute a detailed, formal process and keep a record of the meeting results for later analysis to improve both the quality of the code and also of the inspection process itself. Below we describe formal methods in more detail and discuss collaborative support for such methods.

2.1. Formal software code inspection

A typical formal software code inspection consists of six steps (Table 1, Refs. 9 and 17). Formal inspections often are governed by a set of guidelines that are established upfront. Examples of such guidelines include, but are not limited to.\textsuperscript{16}

• Review the product, not the producer.
• Limit debate and rebuttal.

• Enunciate problem areas, but do not attempt to solve every problem noted.

• Insist upon advance preparation by participants.

Fagan inspections have traditionally been performed in the context of traditional life-cycle (waterfall) model and the Unified Modeling Process. Formal code inspection methods have a number of advantages and disadvantages. According to Ref. 8, the advantages include a more thorough coverage of the software code, the identification of more defects, and the evaluation of the code’s understandability and maintainability. Among the disadvantages are the need for thorough preparation prior to the inspection meeting, and the high costs per defect because of the use of multiple inspectors and the slow coverage.\(^8\)

Fagan argues that the main reason some inspections produce inferior results is that the formal inspection process is not well understood and is often poorly executed,\(^4\)

<table>
<thead>
<tr>
<th>Steps</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Planning</td>
<td>The materials to be inspected are arranged, the designated inspectors are checked for their availability, and the meeting place and the time is decided.</td>
</tr>
<tr>
<td>Brief (Overview)</td>
<td>The inspection team is briefed about the scope and purpose of the code to be inspected to help them understand what is expected from them. This may include the “distribution of documents, role assignments and training in code inspection procedure, rules, and checklists”.(^{17})</td>
</tr>
<tr>
<td>Preparation</td>
<td>The inspectors go through the software code individually and make an effort to understand it completely.</td>
</tr>
<tr>
<td>Inspection Meeting</td>
<td>The inspectors meet to identify issues (defects) in the code that is inspected. They are encouraged to find as many major issues as possible.</td>
</tr>
<tr>
<td>Rework</td>
<td>The identified issues are reviewed by the code author and are classified as defects that need to be corrected.</td>
</tr>
<tr>
<td>Follow up</td>
<td>The moderator makes sure that all necessary actions are undertaken on all identified defects. The moderator also has to make sure that no defects are introduced during rework. Finally, he decides if there is any possibility or need for re-inspection.</td>
</tr>
</tbody>
</table>

so research into improving and understanding inspections is important. The satisfactory outcome of an inspection largely depends on how well the purpose of inspection is defined for the inspectors. The purpose may be to understand the code, ensure the requirements are met, guarantee system coding standards are followed, maintain the code, or to identify corrective actions for the defects
uncovered. A clearly understood purpose affects how the inspectors examine the code. Further, the manual process of error documentation is tedious. In addition, defects found may not be logged properly because of miscommunication among programmers, code inspectors, and moderators. Finally, the quality of the inspection can be at risk because some developers do not handle criticism very well or because inspectors do not come prepared. In sum, the entire inspection process can be time consuming and costly.

The roles of inspector, author, and moderator exist in the inspection process. Inspectors actually perform the inspection. Moderators organize and oversee the process. The author of the code usually does not actively participate in the inspection or is even not present. Sometimes however, the author may also play the role of being one of the inspectors.

The preparation step requires inspectors to understand the document and identify defects individually. During the inspection meeting the moderator then collates these defects. A lot of time is wasted here as inspectors often find and record duplicate defects due to the lack of a mechanism to share the defects as soon as they are identified by individual inspectors. Therefore a major amount of inspectors’ time is spent on “redundant search”. During a typical inspection meeting each inspector waits for his turn to report defects to the moderator. This time could be used more effectively for detecting new defects thereby increasing inspection productivity.

Another limitation of traditional inspection process is that it relies heavily on face-to-face meetings. If the inspectors are scattered in different geographical areas or simultaneously working on different projects then it is difficult to find time and place for inspection meetings. Reference 18 proposes a distributed collaborative software inspection environment that removes location constraints and provides sufficient structure to partially relax time constraints. In such an environment, developers and inspectors need not be present in the same location for the code inspection session. This considerably reduces the time and cost, and increases the flexibility of inspections.

2.2. Collaboration in software inspection

Software code inspection is a collaborative task. When more than one person inspects software code, each person follows a set of inspection guidelines and works together with others to accomplish the common goal of identifying the defects. Multiple people participate in software code inspections. The inspection team consists of the author of the code, a moderator, individual inspectors, and at times even a software manager and a scribe. The moderator is responsible for getting the source code from the author. (S)he coordinates with the inspectors and
distributes the code. The manager describes the code to the inspectors and explains the purpose of the inspection and what is expected from the team. After individual preparation, the individual inspectors assemble to discuss the software code and detect as many defects as possible. During the meeting the defects are reported and transcribed. The scribe records the identified defects in a formal defect log. Finally, the defect log is submitted to the author of the code for review and repair of the defects. Thus, the software code inspection process can be rightly termed as a collaborative process.

Inspection teams may use different types of tools to support their activities. Such tools can support the inspection activities itself, the collaboration between the people involved in the inspection process, or both. Computer support allows the inspection teams to handle inspection artifacts online, to record inspector comments and create project management reports that reduce the volume of bulky printed materials normally generated by inspections. For example, Ref. 20 designed a PC based application that supports verification and validation activities based on Fagan inspections. They feel that a desirable attribute of inspection is “rigor”, which computers can help provide.

Inspection teams also use groupware tools to “communicate, cooperate, coordinate, and solve problems”. Group Support Systems (GSS) are one family of groupware tools that is being used for code inspections. A GSS provides a set of configurable modules that can be used to support different collaborative activities for the collection, categorization, and evaluation of information, such as inspection artifacts. According to recent studies, a GSS supported inspection process can significantly improve the number of defects found during software code inspection and appears to be more effective and efficient in finding defects. The advantages of using GSS in code inspection include:

- GSS provide automatic report generation. All information collected during the inspection session can be formatted into a formal report that summarizes the entire session.

- GSS allow inspectors to work in parallel and to look into other inspectors’ contributions. Thus inspectors can contribute and access information easily and efficiently. Since inspectors working in a group can view defects identified by other inspectors, the number of duplicate defects is reduced.

- GSS allow inspectors to capture their comments themselves as they identify defects. This reduces the likelihood that the defect will be misunderstood during the reporting process. It also reduces the potential for loss due to forgetting them at a later stage in the inspection process.
• GSS support both distributed and face-to-face collaborative work. Inspectors need not be in the same room. They can contribute from their own office desk and see what others have contributed.

A substantial body of research shows that, under the right conditions, groups that use GSS can enhance their productivity and create organizational value.\textsuperscript{24–27} For example, field studies on GSS have reported labor cost reductions averaging 50% and reductions of project calendar days averaging 90%.\textsuperscript{28,29} Yet, GSS are typically complex systems to apply effectively in a sustained fashion.\textsuperscript{30,31} They require the assistance of a skilled facilitator to be configured effectively.\textsuperscript{14,32,33} For example, the GSS feature of anonymous communication appears to impact group performance in inspection tasks: Ref. 34 found that non-anonymous groups were more effective and had a more positive attitude towards the inspection task than anonymous groups. However, most organizations do not have the resources available to employ skilled facilitators to operate GSS for inspection (or other) meetings.\textsuperscript{30,35} Such organizations need a repeatable collaborative script that clearly instructs an inspection team how to perform a collaborative inspection without a facilitator. This research created and applied such a script in a real organization. Early research investigating collaborative support for code inspections was with tools that were more primitive, in some ways, than the groupware used in this study. For example, the ICICLE tool studied by Ref. 36 allowed multiple inspectors to see the same screen and to drop flags, but did not allow the textual input of information about defects. Reference 37 surveys the literature on collaborative support for code inspections. Most controlled experiments with students have not found that tool-supported inspections are more effective and efficient than manually supported inspections (cf. Refs. 18 and 37), though some studies have shown that collaborative tools do results in more defects found in Ref. 38 or time savings in Ref. 21. Very few studies of collaborative tool support in industrial settings have been done, though they have been more encouraging than studies with students. For example, Refs. 39 and 15 found that inspections supported with collaborative tools are more effective in terms of finding more defects and more efficient in terms of saving time than manually supported inspections. However, these studies have not examined the perception of software engineers in terms of perceived effectiveness and efficiency. The current study examines both actual and perceived effectiveness and efficiency in a field setting.

\textsuperscript{9}Bug descriptions are collated in the collaborative software and can be exported as text. There does not currently exist an automated export to bug tracking software such as Bugzilla, but text can be easily pasted into such software.
3. Method

Our research objective was to design and evaluate a repeatable collaborative code inspection method. A field study method appears to be most appropriate in this context as it allows us to gain a richer understanding of the workings of our collaborative process in action. Furthermore, we felt that action research is the most suitable field research methodology. Action research allows for the actual design of an intervention in addition to studying its application in practice. It is the most suitable research method to answer “how to” questions. Other field methods such as case studies, ethnography, or grounded theory studies do not include the design perspective. Action research seeks both to improve practice and to contribute to theory and knowledge. We followed a model (Fig. 2) that states that an action research study may consist of four activities or phases that can be carried out over several iterations: “Plan” concerns exploration of the research site and the preparation of the intervention. “Act” refers to the actual intervention. “Observe” concerns the collection of data during and after the actual intervention to enable evaluation. Finally, the “Reflect” activity analyses the collected data and infers conclusions that may feed into the “Plan” activity of a new iteration.

We selected action research as our research approach for several reasons. First, action research is especially appropriate for addressing “how to” research questions.
Our research aimed to develop a process to perform collaborative code inspections. Second, we felt that collaborative aspects in code inspections are too poor under-stood to study in a constructed, tightly controlled, experimental setting. Finally, action research is very well suited for continuous learning. It allows researchers to continuously evaluate and improve their products during a series of interventions. In our research, we initially produced a few prototypes of the collaborative inspection process.

We executed the four steps as follows. During the first phase, “Plan”, we conducted a literature review and approached other researchers working on software code inspection to understand the limitations of exiting inspection processes. This understanding informed the first rough collaborative code inspection process design, which was modified and improved on in a few iterations. Details of the process are presented in Sec. 4.

The second phase, “Act”, covered the execution of the collaborative code inspection process in the field. We applied the process in four inspections to observe and study it in action. The four inspections were carried out in two organizations. To increase the external validity of this research and because experienced software engineers can focus more on defect finding rather than being overloaded cognitively with the complexity of understanding the code, we conducted this case study with experienced software engineers. The software engineers at both locations were very comfortable using the GSS tool:

• Organization 1: An inspection team consisting of four experienced professional software engineers with sound knowledge of software code inspection inspected two pieces of C++ code in the IT department of a mid-western U.S. logistics firm. In this organization we could observe the performance of the designed inspection process in an industrial setting. The two inspections will be referred to as Case P1 (paper-based) and G1 (GSS-based).

• Organization 2: An inspection team consisting of three experienced student programmers inspected an entire software module in PERL and HTML at the Computer Science Department of a Mid-western University in the United States. The paper inspection and GSS inspection examined different parts of a complete module consisting of different software programs. The module inspected is a part of a larger application developed by the department and commercially applied in the field. Although most inspectors lacked industry experience, they were experienced software developers with sufficient knowledge of software code inspection. This setting allowed us to evaluate the inspection process in an environment with less experienced inspectors. The module was inspected in two parts, referred to as Case P2 (paper-based) and G2 (GSS-based).
During the “Observe” phase, we collected and analyzed explorative data to evaluate how the collaborative inspection process performed and was perceived in the field. We used different instruments, including informal interviews with a selection of the inspectors involved and questionnaires for all participants in the inspection sessions. With the questionnaire we collected both qualitative and quantitative data regarding the perceived effectiveness and efficiency of the inspection process, and the participants’ satisfaction with the inspection process and its outcomes. We also analyzed the actual inspection results on various aspects such as numbers of major/minor defects identified and the time taken for the inspection. Majors are defined as defects that would, if undetected, result in an output error or system error in test or in the field. Minors are all other defects. Finally, it is important to note that this study does represent a laboratory experiment. It represents the application of a collaboration process design in four situations, each of which dealt with a different piece of code. Thus, given the developmental character of the research and the limited sample size, the analyses were exploratory in nature.

During the final phase, “Reflect”, we analyzed our observations. This pointed out some limitations of the design process and set the stage for future research topics.

4. The Collaborative Inspection Process

This section presents the design of the collaborative inspection process. We first elaborate on the design criteria we considered and a standard collaborative inspection process that we used as a starting point for our design. Second, we describe the Collaboration Engineering principles and techniques that we used to create our design. Finally, we present our design and discuss its paper-based and GSS-based implementation.

4.1. Foundations

The collaborative inspection process was designed in a few iterations. Each of these iterations was done with an eye on several design criteria. The iterations were necessary to optimize the different criteria as much as possible. The following criteria were considered:

• Efficiency: The collaborative inspection process should not require a lot of time to arrive at satisfactory results.

• Ease of use by inspectors: The collaborative inspection process should not be complex and should be easily understood by the code inspectors.

• Comprehensiveness of results: The process should be able to identify different types of defects as defined before the inspection execution. Such defects
may include syntax and semantics errors, logic errors and inefficient ways of coding.

- **Non-redundancy of results**: The process should minimize the redundancy of identified defects as much as possible. Having a group of inspectors inspecting the code, the probability of identifying duplicate defects is considerable.

- **Applicability to varying quality of software code**: Software code quality can vary from well-documented superior code to inefficient code with a large number of errors. The collaborative inspection process should be effective for software code of different quality.

We did not design the collaborative inspection process from scratch as researchers had already proposed different processes in the literature. We used the process proposed by Ref. 10 from a case study with the Baan Development Applications Department in the Netherlands as our basis. This process is based on existing industry inspection standards and has two phases, each consisting of a number of activities. In the pre-meeting phase, the code to be inspected is printed with each page numbered and handed over to the inspectors. The inspectors understand the code and the type of errors they have to identify. Then they identify and document the defects that they find. In the discussion-meeting phase, the defects identified by the inspectors are collated into a common defect log. Overlapping defects are removed to arrive at a list of unique defects, which are classified as major defects, minor defects, or false positives. False positives are defects, which the inspectors initially consider as defects but, after discussion, conclude that they are false alarms. During the meeting, the inspectors also discuss any further questions or comments they have with respect to the code. After the meeting is adjourned, the results should consist of a list of clear, unique, and classified list of defects, which is submitted to the author of the code for further review and resolution.

4.2. Collaboration engineering

We designed the collaborative inspection process following Collaboration Engineering (CE) principles and techniques. CE is an approach to designing, modeling and deploying repeatable collaboration processes for recurring high-value collaborative tasks that are executed by practitioners without the ongoing intervention of professional facilitators. This means that with very little training, software engineers can run both the process and collaborative tools required to complete the inspection process. Several design steps take place within CE. The ones relevant for our study are the decomposition of the process into collaborative activities, their classification into patterns of collaboration, and the selection of appropriate group facilitation techniques, called thinkLets, to guide the execution
of each activity. These steps are normally executed iteratively.

The decomposition of the collaborative inspection process into collaborative activities was based on the standard process presented above. Each of the collaborative activities in that process could be classified as aiming to achieve a particular pattern of collaboration. A pattern of collaboration represents observable regularities in the behavior and outcomes that emerge over time in teamwork. Within CE, six patterns of collaboration are identified:

• Generate: To move from having fewer concepts to having more concepts. The goal of generation is for a group to gather or create concepts that have not yet been considered by the group. Brainstorming is an example of a generation process.

• Reduce: To move from having many concepts to having a focus on fewer concepts deemed worthy of further attention. The goal of reduction is to decrease a group’s cognitive load by limiting the number of concepts they must address.

• Clarify: Moving from less to more shared meaning for the concepts under consideration. This is important because people frequently use the same label for different concepts, use different labels for the same concepts, and use labels and concepts that are unfamiliar to others.

• Organize: To move from less to more understanding of the relationships among concepts. The goal of organization is to reduce the effort of a follow-on activity. A group might, for example, organize a list of ideas into a number of categories.

• Evaluate: To move from less to more understanding of the benefit of concepts toward attaining a goal. The goal of evaluation is to focus a discussion or inform a group’s choice based on a judgment of the worth of a set of concepts with respect to a set of task-relevant criteria. For example, an evaluation process may involve a team using a five-point scale to rate a set of alternatives.

• Build Consensus: To move from having more disagreement to having less disagreement among stakeholders on proposed courses of action. The goal of consensus building is to let a group of success-critical stakeholders arrive at mutually acceptable commitments.

To achieve these patterns of collaboration in a group, a facilitator or moderator may apply facilitation interventions called thinkLets. ThinkLets are repeatable, predictable, transferable facilitation techniques to assist a group in reaching its agreed goal. A thinkLet encapsulates an expert facilitator’s best practice for producing a known pattern in the behaviors of a group of people who collaborate. Some thinkLet examples are given in Table 2. A complete description of a thinkLet that was used in the collaborative inspection process, ChauffeurSort, is
given in Appendix A. Thin-k Lets can be used and re-used as building blocks for team process designs in any domain where collaboration is required.\textsuperscript{14} A sequence of thinkLets and the transitions from thinkLet to thinkLet comprise a design for a collaboration process.\textsuperscript{13} We used thinkLets for our collaborative inspection process design to make the design more predictable and repeatable in practice.

4.3. The design

The final design of the collaborative inspection process was the result of three iterations. Early iterations were deemed less desirable because of perceived inefficiencies in the discussion of found defects, the disproportionate amount of time required to complete the process for pieces of code with a large number of defects, and the incomplete recording of all defects. For example, in an early iteration, the inspectors would each introduce and discuss their defects one by one, which made the process slow especially with a large number of found defects. Another iteration focused on speed, allowing inspectors to only discuss what they felt were critical defects as shown on a combined defect log. This iteration was abandoned because the risk was too great that not all important defects were recorded correctly as the selection of defects might be too subjective. The final design is presented in Fig. 3.

<table>
<thead>
<tr>
<th>ThinkLet Names</th>
<th>Patterns of Collaboration</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>Directed-Brainstorm</td>
<td>Generate</td>
<td>To generate, in parallel, a broad, diverse set of highly creative ideas in response to prompts from a moderator and the ideas contributed by team mates.</td>
</tr>
<tr>
<td>LeafHopper</td>
<td>Generate</td>
<td>To generate, in parallel, ideas in depth and detail on a focused set of topics.</td>
</tr>
<tr>
<td>BroomWagon</td>
<td>Reduce</td>
<td>To eliminate the least important ideas from a large set.</td>
</tr>
<tr>
<td>FastHarvest</td>
<td>Reduce &amp; Clarify</td>
<td>To have pairs of team members extract a list of key ideas on assigned topics from a raw set of brainstorming comments.</td>
</tr>
<tr>
<td>ChauffeurSort</td>
<td>Organize</td>
<td>To organize ideas into categories through a short group discussion of each idea.</td>
</tr>
<tr>
<td>Concentration</td>
<td>Organize</td>
<td>To remove overlap among ideas to create a unique set.</td>
</tr>
<tr>
<td>StrawPoll</td>
<td>Evaluate</td>
<td>To evaluate a number of concepts on one or more criteria.</td>
</tr>
<tr>
<td>MoodRing</td>
<td>Build Consensus</td>
<td>To continuously track the level of consensus within the group with regard to the issue currently under discussion.</td>
</tr>
</tbody>
</table>
The process starts with the inspectors individually familiarizing themselves with the code and understanding it. They then are invited to individually identify and record defects. Through the DirectedBrainstorm thinkLet they are stimulated to think about different types of defects in parallel. The DirectedBrainstorm thinkLet prescribes the facilitator to give participants prompts so that they take all relevant aspects into account. Such prompts can represent aspects or types of defects, e.g. “think about syntactical defects” or “think about logical defects”. In a fact-to-face setting, the facilitator gives the participants a new prompt every couple of minutes to keep the flow of contributions going. In an asynchronous dispersed setting, the facilitator provides the participants a list of prompts that they use as a check list of aspects to pay attention to. During the brainstorming activity, the participants use the page and line number on the printed version of the code to record the defects they identified. Compared with traditional, sequential brainstorming where participants speak in turn, the DirectedBrainstorm thinkLet has shown to be more productive and creative. In addition, the DirectedBrainstorm thinkLet helps a team to address all relevant aspects of a problem during a brainstorming activity.

The discussion meeting phase starts with a Concentration thinkLet. The inspectors first discuss their findings for each page of code. Only unique defects are reported, as the inspectors keep track of duplicate defects while listening to other inspectors’ findings. To this end, the facilitator asks each inspector to remove duplicate defects. This is a straightforward activity because the inspectors use the page and line numbers to identify defects. The result of the Concentration thinkLet is therefore a non-overlapping and provides a clean list of defects. During the discussion, the inspectors may identify more defects, triggered by each other’s contributions. If this is the case, the inspectors perform the DirectedBrainstorm thinkLet again followed by a Concentration to remove the duplicates from the additional defects.

The resulting unique list of defects is then classified into the pre-defined defect types (major, minor, false positive) using the ChauffeurSort thinkLet. In this activity, the facilitator guides a discussion on each defect: (s)he invites the inspectors to determine for each defect what type it is. If inspectors do not agree, the facilitator leads a brief discussion during which opinions are aligned. If a verbal discussion does not result in a quick resolution, the facilitator calls for a vote where the majority opinion wins. In case of a draw, the most senior inspector decides. The agreed upon classification is added to each defect when the inspectors agree. The result is a defect list consisting of unique, non-overlapping defects, which are clearly described and classified into the pre-defined categories. This list is handed to the author of the code for correction.

The collaborative inspection process design was evaluated in two implementations. Each thinkLet specifies which capabilities have to be afforded to
the group so that they can follow the thinkLet’s rules during its execution. We implemented the process on traditional “paper-based” tools (prepared defect sheets, voting stick- ers etc.) and on a GSS, GroupSystems Workgroup Edition 3.4. In GroupSystems, each code page was assigned to a separate entry in an outline. Inspectors could enter defect descriptions with associated source code line numbers into their respective page entry. The discussion of the ChauffeurSort was focused by having the inspectors vote on their preferred classification for each defect in GroupSystems’ Vote module.

Both the paper-based and GSS-based implementations were executed in each organization that participated in the study. The results of the cases are presented in Sec. 5.

5. Results

This section presents the results from the four case studies. Data was collected and analyzed regarding the process’ effectiveness, efficiency, and inspector satisfaction.

5.1. Inspection effectiveness

Inspection effectiveness refers to how well each process found defects. Table 3(a) shows that a considerable number of major defects were uncovered in all the sessions. Table 3(b) shows these values adjusted per thousand lines of code (KLOC) to increase the comparability of the findings. Table 3(a) shows that GSS support found more major defects per KLOC in both locations. In location 1 paper found more minor defects whereas the opposite was true at location 2. In both locations paper inspections resulted in many more duplicates. False positives were higher for paper in location 1 and higher for GSS in location 2.

To measure the inspectors’ perception on the process’ effectiveness, a questionnaire was used on the scale of 1–7, where 7 was the most positive. The reported values in Table 4 are the averages for the responses on perceived process effectiveness. These responses indicate that the inspection process was perceived to be effective in uncovering most of the errors in both locations both with paper and GSS. Some inspectors felt that effectiveness issues actually stretched beyond the inspection meeting. One inspector stated, “Implementation of the corrections is an important aspect of the process”.
5.2. Efficiency

We evaluated process efficiency in two ways. First, we measured the duration of each stage of the inspection processes. Second, we asked inspectors for their perceptions of process efficiency. The efficiency results are similar for both locations as seen in Table 5. As expected paper inspections were less efficient in that significant time was required to merge individual defect logs into an overall defect log during the meeting. In GSS, because inspectors could log their own defects when the defects were found during individual preparation, no meeting time was consumed by logging these defects during the meeting. In Location 1, more time was required to refine the defect list with the paper inspection. In Location 2, defect classification took twice as long in the paper inspection as in the GSS inspection.

To measure the inspectors’ perception on the process’ efficiency, a questionnaire was used on the scale of 1–7, where 7 was the most positive. As shown in Table 6, inspectors rated GSS efficiency slightly higher than paper efficiency.

<table>
<thead>
<tr>
<th>Location</th>
<th>Case</th>
<th>Major Defects</th>
<th>Minor Defects</th>
<th>Duplicates</th>
<th>False Positives</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Paper (P1)</td>
<td>59.9</td>
<td>33.7</td>
<td>56.2</td>
<td>15.0</td>
<td></td>
</tr>
<tr>
<td>GSS (G1)</td>
<td>65.8</td>
<td>9.4</td>
<td>6.3</td>
<td>9.4</td>
<td></td>
</tr>
<tr>
<td>2 Paper (P2)</td>
<td>10.1</td>
<td>82.8</td>
<td>23.6</td>
<td>8.4</td>
<td></td>
</tr>
<tr>
<td>GSS (G2)</td>
<td>36.5</td>
<td>167.9</td>
<td>0</td>
<td>14.6</td>
<td></td>
</tr>
</tbody>
</table>

Table 4. Perceived effectiveness of the process, mean (standard deviation).

<table>
<thead>
<tr>
<th>Location</th>
<th>Paper</th>
<th>GSS</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4.89 (1.13)</td>
<td>4.75 (1.17)</td>
</tr>
<tr>
<td>2</td>
<td>5.83 (1.04)</td>
<td>5.39 (1.42)</td>
</tr>
</tbody>
</table>

Table 5. Measured inspection meeting efficiency.

<table>
<thead>
<tr>
<th>Location</th>
<th>Case</th>
<th>Format Defect Log Generation</th>
<th>Removing Overlaps and Refining the Defect List</th>
<th>Defect Classification</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Paper1 (P1)</td>
<td>30 min</td>
<td>10 min</td>
<td>5 min</td>
<td>45 min</td>
<td></td>
</tr>
<tr>
<td>GSS1 (G1)</td>
<td>0 min</td>
<td>7 min</td>
<td>5 min</td>
<td>12 min</td>
<td></td>
</tr>
<tr>
<td>2 Paper2 (P2)</td>
<td>47 min</td>
<td>5 min</td>
<td>10 min</td>
<td>62 min</td>
<td></td>
</tr>
<tr>
<td>GSS2 (G2)</td>
<td>0 min</td>
<td>5 min</td>
<td>5 min</td>
<td>10 min</td>
<td></td>
</tr>
</tbody>
</table>
Inspectors also found the process to be understandable and versatile. Inspectors stated that they could easily understand the inspection process in all sessions, suggesting that the same process could be used in a variety of settings. Some inspectors felt that the moderator even gave more instructions than the straightforwardness of the process required: “Though instructions certainly help, I felt too much time was spent explaining the instructions, perhaps more brief on instructions supplemented with detailed written instructions would be more efficient”.

Though in this research the inspection process was used only to inspect software codes, the inspectors suggested that it could be easily used to inspect software requirement documents, functional design documents, and other artifacts by defining new defect categories relative to the inspected document. This illustrates the process’ flexibility in terms of its applicability in different settings for different types of code.

5.3. Satisfaction

Inspectors indicated that they were more satisfied with the GSS-supported inspection process. Table 7 (scale of 1–7, 7 most positive) shows that the GSS-supported inspectors were more satisfied with both the process and the outcomes than paper-supported inspectors. They indicated that with this process, inspectors were more likely to attain stated inspection goals. One particular concern that they expressed with the GSS-based process was the amount of discussion that was possible on each defect that individual inspectors had found during the pre-meeting phase. Some inspectors felt the process introduced some time pressure which could result in the defects descriptions in the system not being completely clear to everyone: “Less discussion during GSS process might lead to poorly identified problems/errors, which means they would not be fixed as effectively”. In the end, a number of inspectors felt that the process would benefit from more discussion in general. One inspector said, “More discussions on each error!”

<table>
<thead>
<tr>
<th>Location</th>
<th>Paper</th>
<th>GSS</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>5.33 (1.76)</td>
<td>5.40 (1.30)</td>
</tr>
<tr>
<td>2</td>
<td>5.47 (1.77)</td>
<td>6.07 (1.42)</td>
</tr>
</tbody>
</table>
6. Discussion

The finding that GSS inspections found more major defects per KLOC in both locations suggests that GSS inspections are more effective for major defects. Moreover, the vast reduction in duplicates suggests that the group memory — that is the ability of inspectors to see defects found by other inspectors in the GSS, resulted in far fewer defects. In this research, the mixed results on minor defects and false positives should be further examined in future research.

Actual inspection effectiveness and perceived effectiveness were in opposite directions in this research. GSS inspections were more effective in finding major defects and yet inspectors perceived that paper inspections were more effective. Future research should further examine this phenomenon in order to better understand why the perceptions are inconsistent with measured effectiveness. For efficiency, the measured and perceived measures were consistent. GSS supported inspections required less time because no time was required to produce a defect log during the meeting. This savings makes inspections more efficient. This savings could be used to allow inspectors to search for more defects or to discuss and categorize defects. Inspectors likewise perceived GSS inspections to be more efficient. Finally user satisfaction in both the process and outcomes of the GSS inspections were higher than for paper-based inspections.

While most research on software code inspections focuses on formal inspection methods and tool support for defect recording and tracing, our study concentrated on the design of collaborative procedures to execute an inspection. We built on the research by Ref. 10 and found similar encouraging results. Yet, there are a number of noteworthy differences. First, in addition to minor and major defects, we also included a focus on false positives and duplicate defects. In line with Ref. 47, we felt this would benefit the quality of the inspection results. Second, our process was executed by a dedicated moderator who was not part of the inspection team itself. In Ref. 10 work, the moderator was one of the software professionals. The use of a non-dedicated moderator was found to impact the fluency of the
inspection process as the moderator had to divide his attention between finding defects and guiding the process.

We focused on the design of the process and not directly on the collaborative tool support. The process design was evaluated through a dual implementation in a paper-based and GSS-based environment. In fact, this research’s contribution is that it showed that a sound process design can be implemented and executed successfully in different technological environments. Both implementations were found to produce, to a large extent, similar results.

Yet, the GSS-implementation proved to be more useful for a number of reasons. First, consistent with Ref. 47, it is apparent that using the GSS-environment saved significant time on process execution. While generating a formal defect log, all identified defects by the inspectors were discussed and collated into a common file. In a paper-based method this was executed as a separate activity that took around 30–50 minutes of the execution time. Moreover, the time spent on this activity is directly proportional to the number of defects identified by the inspectors. Second, the GSS allowed inspectors to work in parallel and view fellow inspectors’ lists of defects. This transparency avoided to some extent recording duplicate defects as well as reduced the time required to remove the overlapping defects from the defect log. Third, the voting feature of the GSS supported the categorizing of identified defects. Finally, the GSS offered automatic report generation documenting the entire session data into a formal report. However, even though the paper-based implementation may have been more time-consuming, it still yielded satisfactory results. The inspectors pointed out that they were relaxed and at ease while working truly alone in the pre-meeting part of the inspection. This could be attributed to the absence of peer pressure as no inspector was aware of the productivity of their colleagues.

The findings of this research help provide some insight into how GSS may impact software inspections relative to traditional, manual support approaches. Given the exploratory nature of this research, few groups were used, so this implies caution should be used in interpreting the results because of limited statistical conclusion validity. Future research should study these phenomena with larger sample sizes.

7. Conclusion

Software code inspections are a collaborative technique to improve the quality of software products. We designed a repeatable collaborative code inspection process, which was subsequently field tested in four inspections in two software development organizations. In addition, we investigated two implementations of our process: a paper-based and Group Support System (GSS)-based version. Both implementations were found to be successful in uncovering many major, minor, and false-positive defects in inspected pieces of code. The inspection process was
also flexible in that it could be successfully applied to inspect four different pieces of code. The GSS inspections were more effective in terms of major defects per KLOC. The GSS inspections were also more efficient, both in terms of actual results and inspectors’ perceptions. Finally, the GSS inspections outperformed the paper inspections from a practical perspective: logging and managing defects in a GSS was far superior.

The results of this study show the value in designing a repeatable collaboration process based on collaboration techniques (thinkLets) that can be implemented in paper-based and GSS-based environments. The study shows that while GSS may outperform the paper-implementations, a dependable process will deliver positive results regardless of the platform on which it is implemented. This suggests that it is worthwhile for software development organizations to invest in developing dedicated collaborative software inspection technologies to further improve the measured and perceived efficiency and effectiveness of collaborative inspections.

A number of limitations have to be taken into account when interpreting the results of our research. First, the mindset of the participating inspectors may play quite a significant role in relation to the quality of the inspection. Unenthusiastic inspectors may not be willing to give their 100% to the inspection session, affecting the measured and perceived quality of the inspection process. Although we felt we were working with enthusiastic inspectors, we were not able to take this into account. Second, the GSS application used in this research is a general purpose tool; it is not specifically designed for code inspection, for example it does not support hypertext-like browsing through the code. Therefore, inspectors may have experienced sub-optimal functionality concerning the recording of defects. Finally, the sample size of the study is limited. However, we feel that the limited sample size is offset by the high level of realism that we achieved by executing the process in two software development organizations.

We foresee two key directions for future research. First, our collaborative inspection process was implemented in a co-located collaborative environment. It is useful to explore the value of the collaborative inspection process in a distributed environment. Such an exploration could focus on comparing the efficiency, effectiveness, and satisfaction between the co-located and dispersed executions. Second, our research design did not allow for an in-depth comparative analysis of the paper-based vs. GSS-based implementation. Our results only let us conclude that our process design worked well in both implementations. An in-depth comparison between the paper and GSS version of our process would require, for example, an experimental design in which a number of inspection groups all inspect the same piece of code, half of the groups in the paper environment and the other half in the GSS environment.
Appendix A. ChauffeurSort thinkLet

Below is the standard description of the ChauffeurSort thinkLet.

Choose this thinkLet . . .

. . . when you want to assure that the placement of every item in a category is carefully considered by the team

. . . when creating a shared understanding of the categories is as important as the actual placement of an item in a category

Do not choose this thinkLet . . .

. . . when time is of the essence. A sequence of PopcornSort followed by BucketWalk is far faster.

. . . if the appropriate placement of each item is straightforward or not likely to spark discussion.

Overview

Team members discuss the placement of each item within a pre-defined set of categories. Categories may have been previously derived with thinkLets like ThemeSeeker or RichRelations, or may be pre-defined in a methodology.

Inputs

(1) A set of brainstorming comments. (2) A list of categories for organizing the brainstorming comments.

Outputs

A set of brainstorming comments organized into categories.

How to use ChauffeurSort

Setup

(1) Post the category names as buckets in Categorizer. (2) Post the brainstorming comments as list items in a bucket in the same tool.

Steps
. (1) For each comment on the list, ask the group, a. “In which bucket does this comment belong, and why?”

. (2) Facilitate a verbal discussion about the reasons for placing the comment into a bucket.

. (3) When there is sufficient consensus drag-and-drop the comment into the bucket (category) to which it belongs.

. (4) Repeat steps 1–3 until all comments have been placed in the appropriate buckets.

Insights on ChauffeurSort

In a chauffeured sort the magic is in the discussion of the placement of every item. By the time the task is completed the team will have developed a solid understanding of what each category does and does not mean. That understanding may shift and deepen as the task continues, so it may be worth doing a final BucketWalk thinkLet at the end to verify that the items placed early in the process wound up in the right categories.

A ChauffeurSort is not the fastest way to organize comments from a brainstorming session. The PopcornSort/BucketWalk combination will be far faster. However, with that combination you will only discuss the items that somebody believes to be misplaced.

ChauffeurSort Success Story

The Criminal Investigations Division of the Amsterdam Police Department in the Netherlands was reorganizing its Information Department. This department gathers intelligence and background materials on organized crime, and prepares cases for Investigative Project Teams to investigate. The Information Department needed to reorganize because the way in which they were fighting organized crime was changing. They were moving from a functional bureaucracy to a project-based approach were people from many specialties joined a team for a particular investigation.

The reorganization team used the FreeBrainstorm thinkLet to develop a comprehensive set of information access problems they had experienced in their careers. They used an ExpertChoice thinkLet to develop a set of categories representing all the different kinds of organized crime they encountered — computer crime, drugs, prostitution, and so on. The team then used a ChauffeurSort thinkLet to decide which information problems were associated
with which kinds of crime. In this case the team often put the same item in several
categories. Items that applied to all categories they placed in a “General Problems”
category.

The results served several purposes. Based on the contents of each bucket, they
were able to decide what kinds of specialist should be assigned to teams
investigating each kind of crime. The results also became a checklist for the
Information team to make sure they gathered all types of information they needed
to build a case.

What’s in a Name?

A chauffeur drives your car for you, so you don’t have to think about the
machinery. In ChauffeurSort a moderator drives the technology for the group, so
they don’t have to worry about the physical placing of the ideas in categories, and
to assure that they consider each item in its turn.

References

   84–85.
2. R. L. Glass, Inspections — Some surprising findings, Communications of the
4. M. Fagan, Hitting the bull’s eye — Improving your quality, schedule, cost, and
5. L. Harjumaa, H. Hedberg and I. Tervonen, A path to virtual software inspection,
   in Proc. of Asian-Pacific Conference on Quality Software, IEEE Computer
8. K. E. Wiegers, Improving quality through software inspections, Software


19. H. P. Sify, Identifying the mechanisms to improve code inspection costs and benefits, PhD. Dissertation, University of Maryland, Maryland (1996).


41. M. D. Myers, Qualitative research in information systems (2005), http://www.qual.auckland.ac.nz/.


