Task-Directed Software Inspection Technique: An Experiment and Case Study

Diane Kelly and Terry Shepard
Royal Military College Of Canada

Abstract

Research in software inspection has led to the development of inspection techniques focused on providing structure and guidance to the individual inspector, with the goal of improving effectiveness. This paper defines and investigates a new inspection technique, task-directed inspection, specifically developed for inspecting complex computational code, but capable of being applied in other software domains. Students from the Royal Military College of Canada and Queen's University in Kingston, as participants in an experiment, applied two task directed techniques and an industry-standard non-structured inspection technique to a civil engineering code in use in military applications. Results from the experiment were analyzed with a new Orthogonal Defect Classification for computational code developed for this research. Based on this small sample group, the task-directed techniques help software inspectors more thoroughly examine and understand software code. This research also points out the differences between experienced and inexperienced inspectors, and opens up several possibilities for further research.

Keywords
Software inspection, software inspection process, software reading techniques, orthogonal defect classification.

1. INTRODUCTION

The significant advantages of software inspection, the formal human review of software work products, over other verification techniques have been documented in studies such as those carried out at Bell-Northern Research [16]. However, an informal survey [7] indicates that 80% of 90 respondents practiced inspection irregularly or not at all. Johnson [7] comments that some industry practitioners found inspections to be "... difficult, costly, ineffective, and/or excessively time consuming, despite the prospect of quality improvement."

Inspections, being labour intensive, can indeed be costly, but that cost can be more than offset by effective results and reduced by changes in the inspection process.

Since Michael Fagan published his first article in 1976 on inspections [6], extensive research has been focused on various aspects of the inspection process. One particular area of research is on the techniques used by individual inspectors to detect defects in the software work products. Increasing the effectiveness of the individual inspectors means making better use of the individual's time and expertise and also creates opportunities to streamline the overall inspection process.

In 1996, one of the authors (Diane Kelly), while working at Ontario Hydro in the Nuclear Division (OHN), developed a new inspection technique, called task-directed inspection [8]. The technique is designed to provide structure and direction to the individual inspector, enabling a more thorough examination and understanding of the work product and increasing the effectiveness of the inspection.

At OHN, the new inspection technique was combined with an inspection process that decoupled the inspection activity from the correction of the code product. Inspection is considered an evaluation process rather than a corrective process. Findings identified during the inspection exercise exit the inspection process without being fixed. The need for large inspection meetings disappears along with the difficulties associated with such meetings and the inspection process is focused on individual contributions rather than team contributions.

It was within this process framework that we carried out an experiment to evaluate the effectiveness of two kinds of task-directed inspections. The experiment was devised to both measure the effectiveness of task-directed inspections and to observe the use of the task-directed technique in a setting different from where it was developed.

The effectiveness of the task-directed inspection was defined as the ability of the inspector to identify subtle defects in the work product. Measurements commonly taken in empirical studies of inspection techniques involve counting the number of defects and calculating defect rates [11], [14], [15]. Counting number of defects does not provide a clear picture of the inspector's work. Findings can vary from white space inconsistencies to logic errors in a complex calculation. These two extremes illustrate very different levels of understanding of the work product. In complex code, subtle defects are difficult to find and require the inspector to achieve a deep level of understanding.

Since there are different understandings of the word “defect”, we use instead the word “finding” for any issue identified by the inspector during the inspection process. This is to emphasize the idea that an issue identified by an inspector may not be a defect by some interpretations, e.g., may not lead to incorrect operation of the software. Findings can be related to maintainability and other issues. Here we use the more general word finding, and where we use the word defect we mean the word finding.

The task-directed techniques are designed to increase the inspector's understanding of the work product. To evaluate whether that has been achieved, a metric is needed beyond simply counting findings. It must differentiate between findings that address "white space" issues and those that address "logic errors". A tool analogous to the IBM Orthogonal Defect Classification [4] was developed for use in this experiment to provide a means of categorizing the inspector's findings and analyzing the effects of the task-directed techniques on the types of findings identified. Types of findings can be associated with the level of understanding the inspector achieved in identifying that finding. If the inspector identifies more findings that depend on a deeper understanding of the work product, then the task-directed technique has increased the effectiveness of the inspection by this measure. It should be noted that the subtlety of the finding does not correlate with the end consequence of the finding on the operation of the software product; subtle defects may have minor consequences while simple to find defects may have major consequences.

The experiment was carried out with graduate students at Queen's University and the Royal Military College in Kingston. Twelve students used three different inspection techniques to examine civil engineering code currently in use by the military. The students recorded their findings and times taken while examining the code. The recorded findings were used to evaluate and compare the effectiveness of the three techniques.

The results from the two task directed techniques in the experiment were compared to the results from the ad hoc technique. Analysis indicates that experienced inspectors using the task-directed techniques identified a higher proportion of findings that required deeper understanding of the code.

Section II in this paper provides background for the research described in this paper. Section III gives the circumstances in which the task-directed technique was initially developed and used in industry. Sections IV, V, and VI describe the controlled experiment carried out with the graduate students to assess the effectiveness of task-directed inspections in encouraging better understanding of the product under inspection. Section VII describes a new orthogonal defect classification used as a tool to analyze the findings from the inspection exercise in the experiment. The tool was used to categorize findings according to the level of understanding likely achieved to identify that finding. Section VIII discusses the analysis of the findings identified during the experiment. Section IX concludes the paper.

2. Background

We define software inspection, or simply, inspection, as the systematic, static review of a software product to detect and record defects in the product. Static, as opposed to dynamic, does not include executing the product.

The definition of the inspection process, for our purposes, does not include the correction of defects. This is a deviation from the Fagan inspection process [6] where a corrected product exits the process. Inspection here is regarded as an evaluation or measurement process and not as a corrective process. This reduces the inspection interval, focuses attention on the inspection activity, and decouples the correction of defects from the inspection activity. It is particularly appropriate for legacy software, where defects can be fed into a parallel change control process while the inspection exercise is still ongoing.

Simple metrics were gathered during the experiment, i.e., identifying findings and recording times taken to complete the inspection. Only the findings were subsequently used to analyze the experimental results.

The importance of the work of the individual inspector has been realized for many years. Parnas and Weiss [13] pointed out the need to match the inspector's skills with "... aspects of the [product under inspection] that they are best suited to evaluate." They suggested that inspectors be guided in their individual work by questionnaires, where answering the questions requires the inspectors to work with the product. In other words, the inspectors take an active role in their examination of the product (hence the term "active design review", ADR).

Knight and Myers [10] extended these views in describing their phased inspections (PI). They point out that it is often desirable that software products exhibit various qualities such as maintainability, portability, and reuseability as well as correctness. Each partial inspection, called a phase, focuses on a single specific property of the product. The entire product is inspected for compliance with a specific property, with checklists guiding the inspectors to look for known areas of difficulty. Inspectors are chosen so their qualifications meet the needs of the phase.

An important conclusion came from a large experiment carried out at Lucent Technologies by Porter, Siy, Toman, and Votta where they investigated several aspects of the inspection process [15]. During the 18-month experiment involving professional software development, they studied variations such as numbers of inspectors used in each inspection team, running parallel and sequential inspections, and fixing the product being inspected between sequential inspections. The most significant finding from this experiment is that "...structural changes (team size, number of sessions, etc.) to the [inspection] process ... did not always have the intended effect." Instead, significant improvements to the inspection process "... will depend on the development of new defect detection techniques." In other words, improvements will come with techniques that increase the effectiveness of the individual inspector.

Porter, Votta, and Basili [14] note that in many cases, individual inspections are carried out in a manner that can be characterized as

- non-systematic - the inspector is given no guidance on how to proceed through the product,
- general - the inspector looks for all issues related to the product,
- identical - inspectors in multiple-person teams are reviewing the product for the same issues.

Their preference is to have individual inspectors participate in exercises that are

- systematic - the inspector has clear instructions on how to proceed through the product,
- specific - the inspector is focused on specific classes of defects,
- distinct - there is minimal overlap in responsibilities amongst several inspectors looking at the same product.

Porter et al in [14] carried out an experiment that investigated the effectiveness of a specific, orthogonal, and specialized technique for individual inspections. This technique is called Scenario-based reading. A Scenario describes activities that should be performed by the inspector. As described in [14], the activities are accompanied by questions and provide a procedure for detecting a particular class of defects. Each inspector executes a single Scenario and multiple inspectors are coordinated to achieve broad coverage of the software product. In other words, these are multiple parallel inspections with each inspector focused on a specific, non-overlapping issue. The experiment compared three defect detection methods: ad-hoc, checklist, and Scenario. They came to the following conclusions:

- the defect detection rate for the Scenario technique is superior to that obtained with ad hoc or checklist methods - an improvement of about 35%;
- Scenarios help inspectors focus on specific fault classes;
- the checklist method is no more effective than the ad hoc method;
- inspection meetings contribute nothing to defect detection effectiveness.

On this last point there is conflicting evidence as pointed out in [17].

Extensive work [1][2][3][11][12][19] has since been carried out developing and investigating Scenario based techniques to guide the individual inspectors in their work. Scenarios have been developed for documents such as software requirements specifications, object oriented designs, UML design documents, and code.

Results from the above research indicate that systematic, specific, and distinct techniques for the individual inspectors should be used. However, defining the structured technique can itself be problematic [20] and involve extensive work. Having a structured technique that is easily defined can prove to be a great advantage.

3. A Case Study of the First Use of Task-Directed Inspection

The inspection exercise defined at (OHN) [8] combined the need to update documentation with the need to closely examine a body of complex computational code. One of the authors has observed in her own work that the act of creating careful documentation often shows up defects in the product being documented. By having inspectors produce well-defined documentation, they will be encouraged to examine the source code thoroughly and systematically. With each separate document, inspectors can be given different viewpoints with which to examine the source code. The inspectors can then produce tangible products needed for the continuing health of the software system at the same time as they carry out the inspection. This should make inspection more cost-effective as it is "piggy-backed" on another activity that lends itself to being combined with an inspection exercise.

Three tasks were defined and trials were carried out during the summer of 1996. In March 1997, the inspection exercise was started, using a process shown in Figure [1]. There was a total of twenty people involved over a period of about a year.

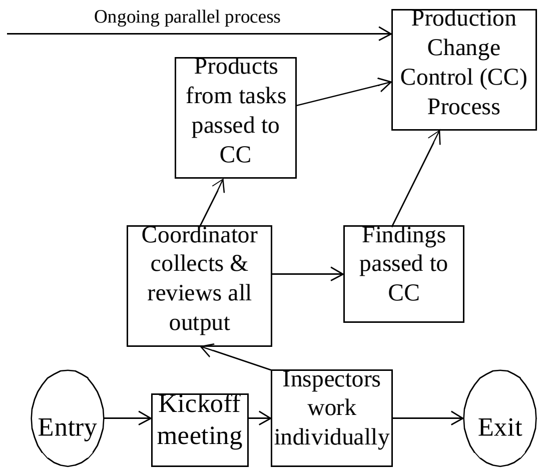

Ongoing parallel process
Production Change Control (CC) Process
Products from tasks passed to CC
Coordinator collects & reviews all output
Findings passed to CC
Entry
Kickoff meeting
Inspectors work individually
Exit

Figure 1: Process for the OHN Inspection Exercise.

The process used for the OHN inspection differs from the Fagan-style inspection in several respects.

- The kick-off meeting is the only meeting with all participants present.
- A coordinator was responsible for the process, reviewing products produced by the inspectors, and passing findings identified in the inspection to a production change control process that runs in parallel to the inspection exercise.
- There were no formal inspection meetings: meetings between the coordinator and the inspectors or the coordinator and a software author were held as necessary to answer questions as the inspection proceeded.
- The source code was not fixed as a part of the inspection process; the inspection process was strictly an evaluation exercise.
- The exit criteria for the inspection exercise were the completion of the individual tasks by the inspectors and the hand-off of the inspectors' findings by the coordinator to the production change control process.

- The inspectors produced products required for the continued maintenance of the software system.
- Corrections to the source code were done in a production change control process that was decoupled from the inspection process; the two processes ran in parallel.
- The only consistent metrics kept throughout the exercise was the record of the findings identified by the inspectors.

Two considerations were important in defining the tasks and assigning the inspectors. None of the inspectors had any familiarity with inspection processes, but all had extensive expertise in a variety of engineering fields. Tasks were defined to be simple and to make use of the inspectors' experience. Each inspector was assigned modules in the code based on the engineering model the module was implementing and the inspector's own expertise. For example, a module calculating heat transfer would be assigned to an inspector knowledgeable about heat transfer models.

Three tasks were defined for the inspectors:

- Provide a data dictionary for the module assigned. A template was provided with a list of the variable identifiers in the module. The inspector had to add definitions for each use of each variable identifier, units of measure if appropriate, meanings for lists of discrete values if appropriate, and indicate whether the variable participates as a parameter in the module calling sequence. The completed template was later added to the top of the source code.
- Give a complete description of the logic of the module. The inspectors embedded their understanding of the module's logic in comments in the source code.
- Cross reference the source code with the technical manual that gives the specifications for the models in the code. Inspectors provided cross reference tags, embedded in source code comments and added to the technical manual indicating matches of equations or theory with the source code.

An additional template, called a Review Summary, was provided to the inspectors to record any findings they uncovered while performing these tasks. The inspectors were not limited in what type of finding they were allowed to record.

Given the background of the inspectors, the tasks were straightforward enough that apart from being provided with completed examples, the inspectors had no need for training. The three tasks guided the inspectors through the modules from three different viewpoints, potentially using three different sets of resources to support their work. Any existing resource (manuals, textbooks, papers, colleagues, source code itself) was available for the inspector's use. Most inspectors merged the three tasks into a process most efficient for themselves. The inspection rate for the professional developers doing all three tasks was typically 20 lines of code per hour, that same rate as the student inspectors doing one task in the experiment.

For the sake of efficiency, it made sense to have one developer do all three tasks on a given portion of code. The complexity of the code required specific domain knowledge and substantial lead time for familiarization.

During the design of the exercise, the option of using checklists was considered and rejected. There was no previous track record to know what would be found by the inspection exercise, and hence no basis to create a comprehensive checklist. The tendency for people to complete a checklist and look no further was considered to be too strong. The shortcomings of checklists are discussed in [12]. The task-directed inspection technique was intended to provide a broader, more open context for the inspectors to explore complex issues and examine the source code in possibly new ways.

A number of comments can be made as a result of this inspection exercise. Most are primarily subjective and not supported by hard evidence.

- the task-directed technique provides a structured guide for the inspector without constraining the investigation,
- the inspection is accompanied by tangible deliverables for later use,
- the tasks can be adapted to the inspector's skills, eliminating the need for extensive training,
- tasks can be defined that help focus the inspector on specific concerns,
- the inspectors had a sense of ownership for their part of the process,

- the inspectors had a sense of making useful contributions to the software development process, i.e., producing tangible products needed for the future life of the software system,
- the inspectors had a chance to apply their particular expertise to the exercise,
- the inspectors had a chance to learn more about a part of the software of interest to them,
- the process made good use of the inspector's expertise,
- the process reduced overlap of work,
- only one large group meeting (the kickoff meeting) had to be scheduled,
- one-on-one meetings were scheduled as necessary to address specific questions,
- work was carried on in parallel with little synchronization necessary; bottlenecks were eliminated.

It is important to note that the three products of the inspection tasks were products required for documentation purposes for this particular software system, independent of whether inspection work was done. The inspection exercise described here was an opportunistic activity to introduce inspection on a large scale, with as little impact on existing schedules and work plans as possible. The results were extremely successful based on subjective evaluations of the deliverables and positive comments from the participants.

It was well after this exercise was carried out that the author involved became aware of scenario-based reading techniques. These are similar to task-directed techniques, but there are three main differences:

- Preparation of suitable scenarios can be difficult and time consuming, but once prepared, the scenarios should be reusable. Tasks are based on previously defined products that are part of the software development cycle, so little preparation time is needed to define the tasks. Tasks however may not be reusable, being specific to a particular software system.
- The benefit of a task-directed technique is being examined at the individual level, not at the team level. The benefit of scenario-based techniques is expected to be seen at the level of teams where several different scenarios are combined to improve coverage of the product under inspection [1].
- The products of the tasks in the task-directed inspection are not artifacts of the inspection, but are products to be put into use in the software system. There is an advantage in reducing the number of inspection artifacts from the inspection exercise.

Of highest interest at this point is the effectiveness of the task-directed inspection technique and in particular, establishing more objective measures of effectiveness.

4. Resources for the Experiment

The purpose of the experiment was to further investigate the effectiveness of task-directed inspection. A secondary purpose was to observe the use of the technique in an environment different from the environment where it was developed.

There were three main considerations in the experiment:

- the code to be used for the inspection,
- the inspection techniques to be defined,
- suitable participants.

Since task-directed techniques were first defined for computational code, the preference was to continue with computational code as the inspection vehicle. Since there is a strong interest to apply results directly back to industry, preference was to locate code that was in current use in a "real world" application. A civil engineering code called Canadian Vehicle Military Load Classification (CAVMLC) was offered for our use. The code was about five years old, under continuous development, written in Visual Basic, tested, but never inspected. The author of CAVMLC was available as a consultant through the experiment. The three portions of the CAVMLC code used in the experiment are comparable in complexity and understandability (using the McCabe cyclomatic complexity metric and judgement from CAVMLC's author).

The code was not seeded with defects for the experiment. Predetermining the types of defects and inserting examples of those into the code would violate the goal of the experiment: to determine the types of defects found using the different techniques. On the other hand, if the code was very clean, then the inspectors would not find anything to report. To assess this risk, we inspected a small portion (about 70 lines of active code) of CAVMLC and readily found 11 issues, suggesting that there would be issues in the code to be inspected.

Three inspection techniques were defined for the experiment.

The first was a method description task-directed technique. A product, a method description, is produced as the driver for the inspection. This involves the inspector recording in a document, an understanding of the source code in pseudocode, table format, and/or natural language. Extensive rigor (such as stepwise abstraction used in [12]) was not demanded of the students.

The second was a white-box test plan task-directed technique. The inspectors produce a test plan, proposing values for variables that would be set outside the code portion and determining the values that the code portion would set.

A third inspection technique was needed to provide a reference point for the two task-directed techniques. A frequently used detection technique is the Ad-hoc technique, a nonsystematic technique where the inspector looks for any defects in the source code or other software product. For the experiment, this technique was referred to as "visual paraphrasing", where the inspector examines the code but has no obligation to record any understanding of the code, other than that implicit in the inspector’s findings.

The comparison for the experiment was between each task-directed technique and the ad hoc technique.

There were twelve participants in the experiment, all students of a graduate Software Verification and Validation course [18]. There was, among the participants, a range of experience and training. Of concern were the effects of these differences on the experiment. The one that was most evident was the split of the class between six students registered in Queen’s graduate programmes (Master’s programmes in either Computer Science or Electrical and Computer Engineering) and six students registered in RMC graduate programmes (either Master’s or PhD programmes in software engineering in the Electrical and Computer Engineering Department). This split was significant because the Queen’s students had little or no industry experience in software development; the RMC students had extensive experience in software development with the Canadian Military. Two of the RMC students had experience doing inspections.

None of the students had any experience with the application domain of CAVMLC, modeling stresses on bridges. It was not expected that the students would understand and be able to comment on the engineering theory underlying the models implemented in CAVMLC. All students had some familiarity with object-oriented programming languages. All the students had an undergraduate degree in either computer science, computer engineering, or electrical engineering, so each had sufficient programming skills to feel comfortable with Visual Basic relatively quickly. Two of the Queen’s students had recently taken a half-year course using Visual Basic; two of the RMC students had used Visual Basic for a brief period, one three years ago and one six years ago.

5. Experimental Design

Several concerns [5], [14] are of particular interest when planning an experiment such as this. The major confounding effect is the wide variation in human performance. Differences have been stated as varying from 4 to 1 to 25 to 1 [12]. This high variability can easily mask any effects from the inspection techniques being studied. Both the design of the experiment and the analysis of the data have to be cognizant of this fact. One of the means of addressing this problem is by a "repeated measures design", where every participant in the experiment uses every inspection technique. This provides an opportunity to observe and perhaps balance out the uncontrolled effects of human performance. Each participant acts as his/her own control.

The maturation of the participants over the course of the experiment is another problem. Each time a participant uses another inspection technique, the participants’ skill level may increase, confounding the results from the use of subsequent techniques. As well, while using different inspection techniques, the participants may apply skills learned earlier. Different approaches can be used here. For example,

- introduce the inspection technique first that is less likely to cause carry-over effects, for example, the visual paraphrasing technique;
- don’t give feedback during the experiment;
- conduct a "warm-up" exercise before the start of the experiment to provide controlled learning;
- use different source code for each inspection done by a participant.

If each participant uses a different portion of source code for each inspection, then variations in the complexity, length, or understandability of the source code portion is another concern. Often impossible to avoid, it can be partially controlled by having all participants work with all portions of source code.

In some experiments [1], a problem occurs where it is unclear if the participants have faithfully followed the experimental process. This can be a difficult problem where either close supervision or an incentive to follow the process may help.

The design of our experiment addressed these concerns as far as possible. Using a repeated measures design, each participant used each of the three inspection techniques once, withthe visual paraphrasing technique always first. Since we wanted each student to use the weakest inspection technique first, we did not use a counterbalanced design. Two students had experience with inspections and had personally developed disciplined inspection habits. Introducing them to a more structured technique first chanced carry-over effects.

We chose three different code portions from CAVMLC so that each inspection done by a participant was conducted on a different portion of code. To prevent any biasing by the order or combination in which a code portion is inspected with a task-directed technique, a partial factorial design was used.

To explain the experimental design, we use the following abbreviations:

Inspection techniques:

P = visual paraphrasing

T = task-directed technique using white box test plan

D = task-directed technique using method description

The three code portions from CAVMLC are designated 1, 2, 3. A participant inspecting a portion of code with a different technique is called a round.

The final design of the experiment is given in Table 1 , where students 1 to 6 are the RMC students and students 7 to 12 are the Queen’s students..

Student	Round 1	Round 2	Round 3
1	**P/3**	**D/1**	**T/2**
2	**P/2**	**D/3**	**T/1**
3	**P/3**	**T/2**	**D/1**
4	**P/1**	**D/2**	**T/3**
5	**P/2**	**T/1**	**D/3**
6	**P/1**	**T/3**	**D/2**
7	**P/2**	**D/1**	**T/3**
8	**P/2**	**T/3**	**D/1**
9	**P/1**	**D/3**	**T/2**
10	**P/1**	**T/2**	**D/3**
11	**P/3**	**T/1**	**D/2**
12	**P/3**	**D/2**	**T/1**

Table 1. Partial Factorial Experimental Design

The body of the table gives the technique/code portion used by each student in each round. The assignment of the students to the participant roles required some care due to the students’ backgrounds. From initial surveys filled in by the students, there were two concerns: the clearly different backgrounds of the RMC and Queen’s students and the two RMC students who had inspection experience. If the students had similar backgrounds, then it would make sense to assign the students randomly to the participant roles. Instead, the assignments were made to evenly distribute the RMC and Queen’s students among the different variations in the experiment.

The experimental design is described in more detail in [9].

6. Experimental Operation

The participants were supplied with a set of documents to support their understanding of the code portions being inspected and their work during the inspections. At appropriate times through the experiment, the students were supplied with worked examples, electronic templates, electronic copies of the source code, and design description and background information for the CAVMLC application. The students were not given examples of findings, purposely not to bias the type of findings the students were looking for.

Although the experiment design addressed most issues of confounding influences, a concern still remained about the use of the weakest technique (visual paraphrasing) first, possibly introducing a negative bias into the first round. It was decided to add a short exercise before the experiment proper, commonly known as a "warm-up". The warm-up familiarized students with Visual Basic, with the coding style of the code author, with a small segment (about 70 lines of active code) of the CAVMLC application, and with using the Visual Paraphrasing technique.

At the beginning of each round of the experiment proper, the students were assigned their code portion and inspection technique. Classes met weekly and results were handed in the next class. Students were to budget their time for each inspection round to ten hours.

Students were encouraged to record all issues they thought suspicious. It was emphasized that an inspector’s job is to raise issues even if there is uncertainty as to their validity. Issues raise a flag on code areas needing a second look. Inspector’s issues may indicate poor documentation, obscure logic, bad style, as well as code errors. The view taken in this experiment is that there are no false positives, i.e. no findings that are not indicative of some problem with the software product. The inspectors were asked to not self-critique their findings. Their findings may point out other issues not originally identified.

7. Orthogonal Defect Classification (ODC-CC) for Computational Code

The primary goal of this inspection experiment was to measure the difference in effectiveness between an unstructured and a task-directed inspection technique. A means was needed to measure this difference. The strength of the task-directed techniques is to encourage a deeper understanding of the code under inspection. By examining the types of findings recorded by the inspectors while using different inspection techniques, we can assess the level of understanding attained. However, we need a means of translating findings to levels of understanding. We do this by means of a new Orthogonal Defect Classification. The IBM Orthogonal Defect Classification (ODC), described in the paper by Chillarege et al [4] in 1992, was considered for this purpose. However, IBM’s ODC proved to be unsuitable for the needs of this experiment. It did not provide enough detail or cover all defects identified during the OHN inspection exercise. More details are given in [9]. A new Orthogonal Defect Classification for Computational Codes (ODC-CC) was developed from the OHN exercise. This allowed the findings from the experiment to be categorized then analyzed to assess differences in the results from the different inspection techniques.

The ODC-CC is composed of multiple levels of defect type categories, each level adding more detail. The top level is composed of five categories:

- Documentation: documentation against which the code may be compared. The issues classified here are with respect to the documentation. This includes comments in the code.
- Calculation/Logic: issues of implementation related to flow of logic, numerical problems, and formulation of computations.
- Error Checking: issues related to data values (conditions and bounds), pre- and post- conditions, control flow, data size, where specific checks should be included in the code (defensive programming).
- Support Code: issues in supplementary code used for testing, debugging, and optional execution.
- External Resources: issues in interactions with the environment or other software systems.

Defect subtypes are added in successive levels. Each level provides more detail for the defect type, subdividing it into finer categories. In the ODC-CC as it currently exists, four levels were found to be sufficient. The multiple levels help direct the classification of the inspection finding. The multiple levels also allow the flexibility of a very fine grained classification or a very course classification. This has advantages in different ways, for example in training inspectors (the more detailed levels help to make the classification clear) and in adjusting the level of effort desired in an inspection (by requiring more or less detail). The ODC-CC is specific to computational code, but provides a framework that can be followed for classification of software products in other software areas. Potentially, the type classification at the top level can be kept constant for different software areas.

Five defect qualifiers are included in the ODC-CC:

- M = missing
- W = wrong
- S = superfluous
- I = inconsistent
- O = obscure

Each inspection finding is fully categorized by a defect type and a defect qualifier. There should be only one defect type and qualifier into which the finding fits. This was validated to some extent when the findings from the inspection experiment were categorized.

As well as providing a tool for analyzing the types of findings identified by the inspectors, the ODC-CC also provides a means to separate reported findings. Inspectors may record what are actually several findings, rolled into one finding report. By subsequent inspection of the finding reports, the rolled up findings become evident and can be identified as multiple distinct findings. This helps to provide consistency amongst the reports of the various inspectors.

7.1. Levels of Understanding for the ODC-CC

To aid in the analysis of the experimental data, each category in the ODC-CC is associated with a level of understanding. These levels of understanding are referred to as CISL which stands for Comparative, Identifier, Structural, and Logical levels of understanding. They are defined by the depth of understanding an inspector must attain to identify a defect. The lowest level of understanding is at the Comparative Level, where the inspector compares the code to documentation. Conceptually, by experience, the easiest defects to identify in source code are those found by comparing the code against other documentation. The next level in conceptual difficulty is Identifier, where the inspector determines the use of variable identifiers and whether those uses are consistent and unique for each variable. The third level is the Structure of the code, where the inspector obtains an understanding of the structure of the software to identify the coherence of structure with semantics of the different components in the structure. The level requiring the greatest understanding is the Logical Understanding, where the inspector must understand the logical flow, the formulation of equations, and the handling of error conditions. The CISL categorization developed here is admittedly subjective, based on one of the authors' experience as software developer and code inspector. This is currently a limitation in this experiment and needs to be validated by further research.

The following table gives a high level description of the CISL correspondances to the ODC-CC.

	Defect Description
C	**Comparison of active code to User Documetation, Theory Documentation, internal comments, etc. Naming conventions for variables, modules. Formatting styles.**
I	**Naming of variables and modules versus use.**
S	**Sematic or logical structure of data, active code, and modules. Supporting infrastructure for testing, optional features, and debugging.**
L	**Calculation/logic and error conditions.**

Table 2. CISL correspondance to Defects

8. Experimental Analysis

The primary data from the experiment were the findings identified by each student during the use of each inspection technique. Analysis of this data proceeded in several steps:

- the raw findings were categorized using the ODC-CC;
- all findings reports, plus a summary, were given to the author of CAVMLC for review;
- total findings per student and per technique were graphed and examined for trends;
- the ODC-CC categorized findings were tagged by CISL levels;
- the number of findings in each of the CISL levels were normalized to create C-, I-, S-, and L-proportions; this gave the fraction of each student's findings that were at each of the CISL levels;
- the CISL proportions were analyzed graphically;
- a statistical analysis was done for the C- and L-proportions;
- time reports from the participants were examined and graphed;
- results of the statistical analysis were compared against student comments gathered during the experiment.

Using the most detailed levels of the ODC-CC, all findings from the three rounds of the experiment were categorized by one of the authors and a professional developer. In future replications of the experiment the plan is to have the participants do the categorization.

The total number of findings per technique was:

- P technique: 179
- D technique: 138
- T technique: 129
- Total findings for the experiment: 446

An example of the CISL classification of the findings is shown in Table 3 for the P technique (round 1).

The number of findings identified by each student varied dramatically. There was no consistent trend due to inspection technique, round number, or code portion used. However, there was an obvious trend determined by the program in which each student was enrolled. The RMC students turned in an average of 24, 15, and 17 findings using the P, D, and T techniques. The Queen's students turned in an average of 6, 6, and 5 findings for the three techniques. It was decided after the experiment was completed to do all analysis on the data from the two groups separately.

	C	I	S	L
p1	15	1	2	1
p2	16	4	1	6
p3	18	5	3	3
p4	18	1	2	11
p5	16	7	0	2
p6	4	3	2	1
p7	0	2	0	5
p8	1	0	0	7
p9	2	5	0	4
p10	0	0	0	1
p11	1	1	0	1
p12	4	1	0	2

Table 3: Example of Finding Counts for CISL Levels of Understanding: P Technique

Generally, the raw number of findings for the two task-directed techniques should be lower. The students were given the same amount of time to complete each round, but for the task-directed techniques, the students had to produce a product as well as a findings report. In each case, the product was extensive and required substantial time to prepare. In essence, the students had more time to look for findings using the visual paraphrasing technique than they had with the two task-directed techniques. There is some trend to lower findings counts with the two task-directed techniques for each student, but the trend is not consistent.

8.1. Normalization of the CISL Finding Counts

The number of findings generated by the inspectors using each technique is affected by the technique itself and the performance of each inspector. As discussed earlier, the performance of an individual can vary significantly. This variation can easily mask the influence of the technique on the inspection results.

The repeated measures design of the experiment allows each participant to act as his/her own control. Even at that, we want to normalize the individual variations in findings counts. We do this by dividing each individual's count of findings in each category of CISL by the individual's total findings for that round. This gives a proportion of CISL findings for each round for that individual. The most valid comparisons, then are across the results for each individual.

For example, student p1, using the P technique, has a total of nineteen findings. Fifteen of those are in the Comparative (C) level of understanding, one in the Identifier (I) level of understanding, two in the Structural (S) level of understanding, and one in the Logical (L) level of understanding. For student 1 and the P technique, the counts are divided by nineteen to give the proportions C: 0.79, I: 0.05, S: 0.11, L: 0.05. This normalization is done for each individual in the experiment.

Alternative normalizations were examined and are discussed in [9].

8.2. Graphical Results

The number of contributions in the I- and S-Levels of understanding were low in this experiment, so no further analysis was perfomed on the I- and S- findings. Results in the C-proportions and the L-proportions were analyzed. But in both these categories, the number of findings turned in by the Queen's students were low, making the Queen's results dubious.

The C-proportions represent findings identified through comparisons of the code with some form of documentation (including the code itself). There is a clear split between the RMC and Queen's students in their level of C-proportions, with the RMC students being more consistent. Averages for the RMC students for the P, D, and T techniques are 0.6, 0.6, 0.5; averages for the Queen's students for the three techniques are 0.2, 0.6, and 0.3.

Looking at the results of the C-proportions (see Graph 1) leads to two possible conclusions. For the experienced RMC participants, identifying defects that are at a low level of understanding can be done consistently regardless of inspection technique used. For the inexperienced Queen's participants, the more structured inspection techniques aid even in identifying defects that require less understanding.

The L-proportions represent findings that require the deepest understanding of the source code. The difference between the RMC and Queen's students was again evident (see Graph 2). The results from the RMC students showed a general increase in the L-proportions with the use of the task-directed techniques. Student 4 had inspecton experience and had developed his own disciplined habits. The test plan technique in particular "got in his way". Student 6 had no L findings using the D technique in round 3, even though his total findings were comparable to the other two rounds. In an interview afterwards, it was found he had had family distractions that had made work difficult.

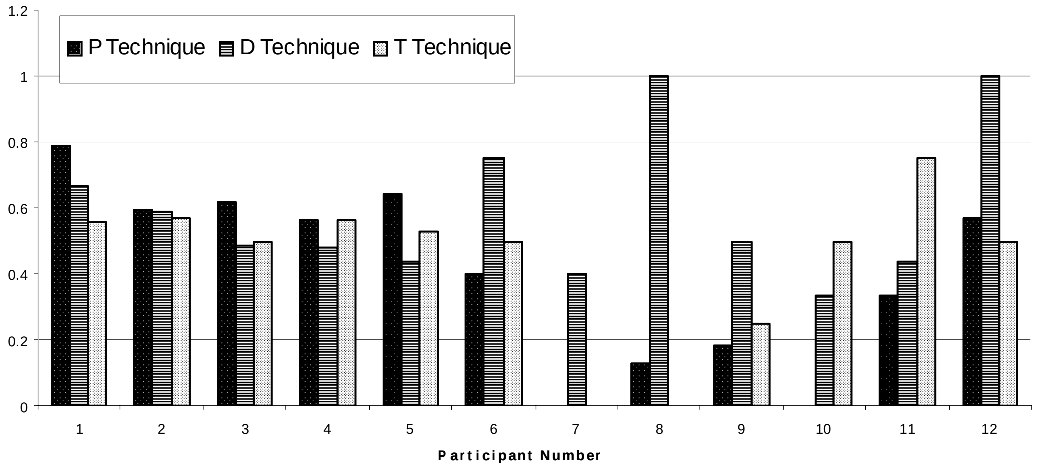
For the RMC students the results suggest that the use of the task-directed techniques increases the proportion of L-level findings. This implies that task-directed techniques encourage the identification of findings that require deeper understanding of the code. For the Queen's students, lack of experience doing documentation impeded effective use of task-directed inspection techniques.

As discussed earlier, two other possible confounding factors in the experiment were the code portions used by each student and the round in which a student used a different technique. A two-way ANOVA examing the interaction between the code portions and the inspection techniques and between the rounds and the inspection techniques showed little interaction between these factors; that is, the code portions used and the rounds in the experiment had no appreciable effect on the results.

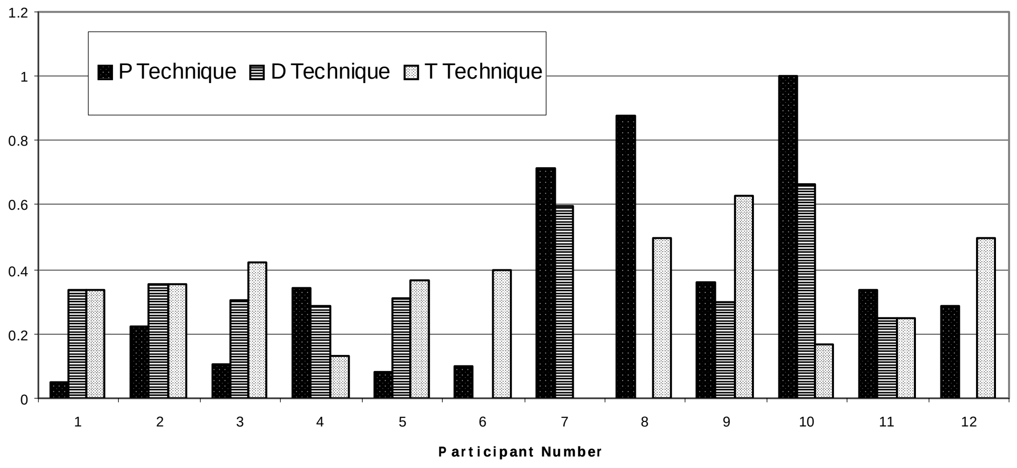
We used a matched pairs one-tailed t-test to compare each task-directed technique with the ad hoc technique, keeping RMC and Queen's as separate groups. The interpretation of the t-test indicates that there was no effect for the Queen's group using the task-directed techniques over the ad hoc. There was an effect for the RMC group using the task-directed techniques over the ad hoc ($p=0.068$ for the D-technique, and $p=0.039$ for the T-technique, power = .59 and .80 respectively).

There seems at this point evidence that for the experienced participants in the experiment, using tasks to structure the inspection resulted in proportionally more L level findings being identified.

.



Graph 1: C-Proportions by Technique



Graph 2: L-Proportions by Technique

8.3. Internal and External Validity of the Experiment

Internal validity is the ability to control unwanted effects on the dependent variable, in our case, the proportion of findings for each individual. Separating the students allows analysis of two diverse groups. The within-subjects design has allowed a statistically valid comparison of an individual's own work across the three rounds.

External validity is based on the generalizability of results to other populations and settings. There are several elements of this experiment that reflect conditions in industry:

- the software under inspection was a working product in a military application with limited support documentation;

- the software under inspection was not seeded with defects for the sake of the experiment;
- products created by the inspectors during the inspection were successfully used by the software code author to improve the application;
- the inspectors had a wide spectrum of external experience;
- the inspectors experienced time pressures and external distractions during the course of the experiment.

All these elements made the experimental data more difficult to analyze, but contributed to the prospect of carrying these results over to industry. More detailed descriptions of the experiment and of the analysis of the results can be found in [9].

9. Conclusions

The performance of the participants in the experiment became an important issue when analyzing the results of the experiment. If all students had lacked outside experience, then the experiment would have produced only limited results.

The analysis of the experimental data provides a number of conclusions on the use of the task-directed technique:

- for experienced inspectors, the task-directed technique appears in most cases to provide positive results in increasing each inspector's understanding of the code;
- tasks must be appropriate to the inspector's background and experience;
- inspectors require a minimum level of experience to be effective.

Apparent from the comments of the participants were the following conclusions:

- task-directed inspections require concentration and inspectors should be provided with the opportunity to accomplish the task free from distractions;
- the intent is that the products of the task provide useful documentation for other stakeholders of the software under inspection, beyind being only inspection artifacts; this was true both at OHN and for the author of CAVMLC;

Task-directed inspection and other structured techniques such as scenario-based reading promise to be powerful tools to improve the inspection process. Further research can address how task-directed techniques can be used with products other than computational codes and how specific inspection goals can be targeted with appropriate tasks. Further research can also address the use of orthogonal defect classifications and levels of understanding to analyze inspection findings. Inspection processes such as the one that served as the framework for this experiment can provide flexibility that can encourage wider use of inspection in industry.

10. Biographies

Diane Kelly is an instructor and Ph.D. candidate at the Royal Military College of Canada. Previously, Diane worked in the Nuclear Division at Ontario Hydro where she participated in a wide variety of roles in software development, from programmer to project leader, trainer to QA advisor. Diane has an M.Eng. in Software Engineering from the Royal Military College, a B.Sc in Mathematics and a B. Ed. in Mathematics and Computer Science, both from the University of Toronto, Canada.

Terry Shepard is a professor in the Department of Electrical and Computer Engineering at the Royal Military College of Canada, where he has played the lead role in creating strong software engineering programs. This includes working extensively with a number of Canadian military software projects, and creating and teaching graduate and undergraduate courses on software design, V&V, and maintenance. He has over 25 years of software experience in industry, government and academia. Terry received his B.Sc. and M.A. from Queen's University in Kingston, and his Ph.D. from the University of Illinois, all in Mathematics. He is a Registered Professional Engineer. He has published a number of papers in software design and verification, and has worked with ObjecTime Ltd. for several years.

11. References

[1] Basili, Victor R., Scott Green, Oliver Laitenberger, Filippo Lanubile, Forrest Shull, Sivert Sorumgard, Marvin V. Zelkowitz; "The Empirical Investigation of Perspective-Based Reading",Empirical Software Engineering: An International Journal, 2(1), 1996, pp. 133-164

[2] Basili, Victor, Gianluigi Caldiers, Filippo Lanubile, and Forret Shull; "Studies on Reading Techniques", Proceedings of the Twenty-First Annual Software Engineering Workshop, Goddard Space Flight Centre, Greenbelt, MD, December 1996 (Software Engineering Laboratory Series, SEL-96-002, pp 59-62)

[3] Cheng, Benjamin and Ross Jeffery; "Comparing Inspection Strategies for Software Requirement Specifications", Proceedings of the 1996 Australian Software Engineering Conference, pp. 203-211

[4] Chillarege, Ram, Inderpal S. Bhandari, Jarir K. Chaar, Michael J. Halliday, Diane S. Moebus, Bonnie K. Ray, Man-Yuen Wong; "Orthogonal Defect Classification - A Concept for In-Process Measurements", IEEE Transactions on Software Engineering, Vol. 18, No. 11, November 1992, pp. 943-956

[5] El Emam, Khaled and Isabelle Wieczorek; "The Repeatability of Code Defect Classifications", Proceedings of the 9th International Symposium on Software Reliability Engineering, 1998, pp322-333

[6] Fagan, M.E.; "Design and Code Inspections to Reduce Errors in Program Development", IBM Systems Journal, Vol.15, No.3, 1976, pp. 182-211

[7] Johnson, Phillip M.; "Reengineering Inspection", Communications of the ACM, February 1998/Vol 41, No.2, pp. 49-52

[8] Kelly, Diane and Terry Shepard; "A Novel Approach to Inspection of Legacy Code", Proceedings of PSQT’00, Austin Texas, March 2000

[9] Kelly, Diane; “An Experiment to Investigate a New Software Inspection Technique”, Master’s Thesis, Royal Military College of Canada, July 2000.

[10] Knight, John C. and E. Ann Myers; "An Improved Inspection Technique", Communications of the ACM, November 1993/ Vol.36, No.11, pp. 51-61

[11] Laitenberger, Oliver, Colin Atkinson, Maud Schlich, and Kahled El-Emam; "An Experimental Comparison of Reading Techniques for Defect Detection in UML Design Documents, December 1999, NRC 43614

[12] Laitenberger, Oliver, Khaled El Emam, Thomas Harbich; “An Internally Replicated Quasi-Experimental Comparison of Checklist and Perspective-based Reading of Code Documents”, [online], http://www.iese.fhg.de/network/ISERN/pub/technical_reports/isern-99-01.pdf; 1999

[13] Parnas, David L. and David M.Weiss; "Active Design Reviews:Principles and Practice", Proceedings 8th International Conference on Software Engineering , London UK, August 1985

[14] Porter, Adam A., Lawrence G. Votta, Jr., Victor R. Basili; "Comparing Detection Methods for Software Requirements Inspections: A Replicated Experiment", IEEE Transactions on Software Engineering, Vol. 21, No. 6, June 1995, pp. 563-575

[15] Porter, Adam A., Harvey P. Siy, Carol A. Toman, Lawrence G. Votta; "An Experiment to Assess the Cost-Benefits of Code Inspections in Large Scale Software Development", IEEE Transactions on Software Engineering, Vol. 23, No. 6, June 1997, pp. 329-346

[16] Russell, Glen W.; "Experience with Inspection in Ultralarge-Scale Developments", IEEE Software, Vol.8, No.1, Jan. 1991, pp. 25-31

[17] Sauer, Chris, D.Ross Jeffery, Lesley Land, Philip Yetton; "The Effectiveness of Software Development Technical reviews: A Behaviourally Motivated Program of Research", IEEE Transactions on Software Engineering, Vol. 26, No. 1, January 2000, pp. 1-14

[18] Shepard, Terry; “On Teaching Software Verification and Validation”, Proceedings of the 8th SEI Conference on Software Engineering Education, New Orleans, LA, 1995, pp. 375-386

[19] Travassos, Guilherme H., Forrest Shull, Jeffrey Carver, Victor R. Basili; "Reading Techniques for OO Design Inspections", Proceedings of the Twenty-fourth Annual Software Engineering Workshop, Goddard Space Flight Centre, Greenbelt MD, December 1999

[20] University of Maryland, Notes on Perspective Based Scenarios; http://www.cs.umd.edu/projects/SoftEng/ESEG/manual/pbr_package/node21.html,[online], November 1999.